PureBasic - A Beginners Guide.

Разделы книги

Перевод осуществил **Станислав Будинов**

глава оглавление стр.

1

Теперь, после введения давайте начнем эту главу с самого простого, а именно с типов данных. Как вы знаете, в компьютерных программах можно манипулировать данными и обрабатывать их. В этой главе я покажу вам все доступные встроенные типы данных и полностью объясню, каким образом и когда их использовать. Чтобы вам начать работать как можно скорее, я включил много примеров и попробую их объяснить понятным для вас языком.

Встроенные типы данных

Типы данных можно рассматривать как способ хранения данных. Основная идея вводить типы данных, это дать некоторые различия тому, что в конечном счете все равно будет в двоичном коде, а нам попросту будет легче понять и различать их, а значит удобнее ими манипулировать. Данные хранятся в памяти компьютера.Объем оперативной памяти, необходимый для каждого типа данных зависит от того, какой тип данных используется.

Числа

Первые типы данных, которые мы будем рассматривать, это числа. Числа могут использоваться, чтобы сохранить что-нибудь, например дату, длину или даже сумму какого нибудь вычисления. Вы всегда используете числа в реальном мире. Почему бы вам не использовать числа в PureBasic для хранения данных.Числа делятся на две разновидности в PureBasic: целые числа и дробные(числа с десятичной или плавающей точкой). Целые числа не имеют десятичной точки и могут быть положительными или отрицательными. Вот некоторые примеры целых чисел:

16543 -1951434 100 -1066 0

Дробные числа, которые содержат десятичную точку, также могут быть положительным или отрицательным. Вот несколько примеров дробных чисел:

52.887 -11.0005 1668468.1 -0.000004 0.0

PureBasic предусматривает двенадцать числовых типов данных для вашего использования в программировании, каждый из которых использует
различный объемом оперативной памяти и все они имеют различные численные ограничения. Описания числовых типов данных:

Тип	Суфикс	Использование памяти	Числовой диапазон
Byte	.b	1 байт (8 бит)	-128 to +127
Ascii	.a	1 байт (8 бит)	0 to +255
Character	.c	1 байт (8 бит) (ascii)	0 to +255
Word	.w	2 байта (16 бит)	-32768 to +32767
Unicode	.u	2 байта (16 бит)	0 to +65535
Character	.c	2 байта (16	0 to +65535

		бит) (unicode)	
Long	.l	4 байта (32 бита)	-2147483648 to +2147483647
Integer	.i	4 байта (32 бита) x86	-2147483648 to +2147483647
Integer	.i	8 байт (64 бита) x64	-9223372036854775808 to +9223372036854775807
Float	.f	4 байта (32 бита)	Без лимита*
Quad	.q	8 байт (64 бита)	-9223372036854775808 to +9223372036854775807
Double	.d	8 байт (64 бита)	Без лимита*
String	.s	Длина строки + 1 байт	Нет лимита
Fixed String	.s{длина}	Длина строки	Нет лимита

***В главе 13 будет дано пояснение**

Числовые границы

В таблице вы смогли заметить, что многие типы данных имеют численное ограничение, это напрямую связано с суммой памяти, выделяемой определенному типу. Объем оперативной памяти выделяется на числовые типы данных более или менее так же, как на языке С. Но обратите внимание, что в языке С вы найдете еще много различных типов данных, отличных от перечисленных здесь, но PureBasic сделан просто, чтобы не сгибать голову над сотнями прогрессивных видов. Для начинающих все, что вам нужно помнить, это о численных пределах каждого типа и понимать, что превышать недопустимо. Чтобы объяснить, почему память, выделенная для каждого типа влияет на количественный предел, я должен объяснить, как эти номера хранятся в памяти на основе бинарных данных. Это вы можете найти в главе 13 (пристальный взгляд на числовые типы данных).

Строки

Строки такой важный и полезный тип данных, что они существуют почти во всех доступных языках программирования. Как следует из их названия, строки просто набор символов. В отличие от чисел, для того чтобы определить строку, ее надо записывать в кавычки. Вот несколько примеров такого синтаксиса:

"abcdefghijklmnopqrstuvwxyz" "Mary had a little lamb" "123456789"

Обратите внимание на последнюю строку, состоящую из цифр. И все равно это строка, поскольку числа записаны в кавычках. Строки, вероятно, простейший тип данных для понимания, потому что они очень просты в использовании. Надо лишь помнить, что символы записанные в двойные кавычки это строка.

Строковые типы PureBasic

Типы	Суффикс	Использование памяти	Кол-во символов
String	**.s**	4 байта (32 бит)	Без лимита

String	**$**	4 байта (32 бит)	Без лимита
Fixed Length String	**.s{длина}**	4 байта (32 бит)	Определяется длиной*
Fixed Length String	**${длина}**	4 байта (32 бит)	Определяется длиной*

***Определяется длиной, заданной пользователем.**

Строки могут быть составлены из любого символа в кодировке ASCII, включая управляющие символы(используемые в памяти) кроме нулевого символа, поскольку он используется, чтобы показать конец строки. Не растраивайтесь, если что то непонятно в этом предложении, просто читайте дальше, со временем все поймете.

Переменные и константы

Чтобы сохранить и управлять данными в любой программе, вы должны использовать правильный тип данных в памяти, но вы также нуждаетесь в способе легко найти эти данные в памяти. Переменные и Константы решают эту проблему. Назначив имя на специфическую часть данных, к ним можно легко обратиться позже. Если переменной присвоить какие то данные, то их впоследствии можно легко изменить. Объявленное же значение константы изменить уже нельзя.

Переменные

Как правило, имя переменной привязано к конкретной области и объему памяти (определяется по его типу данных) , и любые операции с переменной будут манипулировать с областью памяти. Имя переменной задается по желанию и может быть любым, но многие люди делают его максимально информативным, чтобы передать смысл того, что в ней содержится. Переменные являются строительными блоками любой компьютерной программы, так как они хранят данные, которыми можно манипулировать. Переменные, необходимы для организации и хранения данных. Ладно, давайте наконец перейдем к практике в PureBasic. Откроем среду разработки(IDE) PureBasic и создадим нашу первую переменную. Для того чтобы создать переменную, нужно ввести имя переменной. После имени указать суффикс, которым определяется тип переменной, далее оператором = присваиваем значение:

```
NumberOfLinesOfCode.b = 1
```

Обратите внимание на имя переменной, в ней нет пробелов и быть не должно! Если вы хотите разделять слова в переменной для большего понимания кода, тогда используйте нижний дефис:

```
Number_Of_Lines_Of_Code.b = 1
```

Вы можете использовать любое имя для переменной, но есть несколько ограничений. Имена переменных не должны начинаться с числа. Имя не может быть таким же как у любого из операторов (Dim,Print, и т д). Также специальные символы не допускается.

В конец имени переменной добавлен суффикс **.b** Он сообщит компилятору, что эта переменная будет использовать 1 байт памяти.
Если тип суффикса не используется, то есть запись выглядит следующим образом:

```
NumberOfLinesOfCode = 1
```

Тогда переменная будет иметь тип *Long* это устанавливается по умолчанию компилятором. PureBasic предоставляет возможность изменять тип по умолчанию с помощью
оператора *Define:*

```
Define.b
NumberOfLinesOfCode = 1
TodaysDate = 11
```

Команда *Define* имеет суффикс на конце, нужный для определения типа по умолчанию для кода целиком.Но можно определять и для отдельных переменных, тогда данная команда будет действоввать только для них:

```
Define MyChar.c
Define MyLong.l
Define MyWord.w
```

Если вы хотите объявить несколько переменных для последующего использования, но не хотите давать им значения тогда можно использовать этот стиль синтаксиса:

```
Define.w Day, Month, Year
```

Этот код присваивает тип Word для трех переменных Day, Month и Year. Поскольку они задаются без значений, то им присваивается значение 0 (ноль).
А теперь пример, показывающий, создание всех типов переменных в PureBasic:

```
ByteVariable.b = 123
CharVariable.c = 222
WordVariable.w = 4567
LongVariable.l = 891011
QuadVariable.q = 9223372036854775807
FloatVariable.f = 3.1415927
DoubleVariable.d = 12.53456776674545
StringVariableOne.s = "Test String One"
StringVariableTwo$ = "Test String Two"
StringVariableThree.s{6} = "abcdef"
StringVariableFour${3} = "abc"
```

Вы заметили, последние четыре переменные являются строками, но все они имеют разные суффиксы. Первые два с неограниченной длиной строки, а в двух последних с фиксированной длиной. Суффиксы **.S** и **$** . Оба идентичны во всех отношениях. Просто

один является старым стилем, а другой новым. Можно применять любой или оба в вашей программе. Но помните: они не являются взаимозаменяемыми. Например, эти две переменные различны:

```
StringVariable.s = "Test String One"
StringVariable$ = "Test String Two"
```

Даже если они имеют одинаковое название, различные суффиксы показывают, что они являются различными переменными. Не верите? Ну это можно проверить:

```
StringVariable.s = "Test String One"
StringVariable$ = "Test String Two"
Debug StringVariable.s
Debug StringVariable$
Debug StringVariable$+" "+StringVariable.s
```

А теперь по порядку. Присваиваем переменным *StringVariable.s* и *StringVariable$* разные значения. А затем с помощью команды *Debug* выводим результаты в окно отладки. Запустите код нажав F5 . В окне отладки должно быть три результата:

Test String One

Test String Two

Test String Two Test String One (результат сложения двух строк). **" " используется для разделения пробелом**

Команда *Debug* используется исключительно для отладки. В конечном EXE файле эти команды будут автоматически удалены.В дальнейшем эта команда сослужит вам огромную службу. Вы будете использовать ее очень часто, в особенности когда результаты будут отличные от тех, которые вы бы хотели видеть.

И еще один момент хотелось бы отметить. Переменные не чувствительны к регистру. Посмотрите на следующий пример:

```
TestVariable.s = "Test String One"
testvariable = "Test String Two"
TeStVaRiAbLe = "Test String Three"
Debug    tEsTvArIaBlE
```

Вам наверно показалось, что я присвоил трем разным переменным разные значения. На самом деле, я каждый раз присваивал разные значения одной и той же переменной. Фишка вся в том, что если мы какой либо переменной присвоили тип (в нашем случае String), то на протяжении всей последующей жизни программы, эта переменная будет иметь этот тип. И нам не надо каждый раз писать суффикс. А сейчас пример неправильного кода, на который будет ругаться компилятор:

```
StringVariable.s = "Test String One"
StringVariable = 100
```

Если вы попробуете запустить, вы получите вежливое послание IDE о том, что вы не можете писать числовые значения в переменную String. Следующий пример будет работать:

```
StringVariable.s = "Test String One"
StringVariable = "One Hundred"
```

Поскольку переменная *StringVariable* была первоначально объявлена как строка, то только строки и можно в нее заносить.
А теперь обобщим основные правила для переменных:

1). Переменные не должны содержать пробелов.
2). Имена переменных не должны начинаться с цифры, но могут их содержать.
3). Имена переменных не должны быть сходны ни с одним из любых операторов.
4). Имена переменных не должны содержать какие-либо специальные символы.
5). По умолчанию, если не указывается суффикс для переменных, то его тип данных Long.
6). Если переменная была объявлена, его тип данных не может быть изменен во время выполнения.
7). После объявления переменной ее можно использовать свободно, без суффикса, компилятор помнит его тип.

Константы

Константы похожи на переменные тем, что они обеспечивают простой способ обращения к данным и могут быть названы как угодно, но на этом сходство заканчивается. Константы используются, когда вы хотите, чтобы определенная часть данных, назначенная однажды, более не изменялась на протяжении выполнения программы. Посмотрите на следующий пример:

```
#DAYS_IN_THE_YEAR = "365"
```

Мы знаем, что количество дней, например в этом году стандартно и не изменится и мы можем использовать константу, чтобы выразить это. Если мы попытаемся изменить это значение, как переменную, мы получим сообщение об ошибке. Хорошо в константах то, что они не используют память, потому что они никогда не компилируются как таковые, их начальные значения заменяются в коде перед компиляцией. Посмотрите на следующий пример:

```
#DAYS_IN_THE_YEAR = "365"
Debug "There are " + #DAYS_IN_THE_YEAR + " days in the year."
```

Этот пример, в нашей программе, для компилятора будет таким:

```
Debug "There are 365 days in the year."
```

поскольку заменяется постоянной величиной, в данном случае '365 ', а затем компилируется.

Все константы точно следуют тем же правилам именования, как переменные, за исключением суффиксов, константы не используют их даже независимо от того, какой тип данных им назначен. Все объявленные константы используют префикс (#).

Перечисление Констант

Если вам нужен блок констант в котором к примеру присваиваются числовые значения, перечисляемые один за
другим, то вы можете использовать команду *Enumeration*.

```
Enumeration
#ZERO
#ONE
#TWO
#THREE
EndEnumeration


Debug #ZERO
Debug #ONE
Debug #TWO
Debug #THREE
```

Вы увидите в окне вывода отладки, что каждая константа имеет большее значение, чем предыдущая, а отсчет идет с '0 '. Если вы хотите, начать перечисление с другого номера, вы можете использовать опциональный числовой параметр с командой *Enumeration*, вот так:

```
Enumeration 10
#TEN
#ELEVEN
#TWELVE
EndEnumeration


Debug #TEN
Debug #ELEVEN
Debug #TWELVE
```

Теперь вы можете увидеть, что у первой константы значение '10', а остальные увеличиваются, начиная с нее(то есть 11,12).
Вы даже можете использовать шаг увеличения констант. Посмотрите на следующий пример:

```
Enumeration 10 Step 5
#TEN
#FIFTEEN
#TWENTY
EndEnumeration


Debug #TEN
```

```
Debug #FIFTEEN
Debug #TWENTY
```

Теперь константы увеличивается на '5 ', начиная с '10'.

Если присвоить какое нибудь значение в любом месте перечисления блока, это установит новое значение
перечисления:

```
Enumeration 5
#FIVE
#ONE_HUNDRED = 100
#ONE_HUNDRED_AND_ONE
#ONE_HUNDRED_AND_TWO
EndEnumeration


Debug #FIVE
Debug #ONE_HUNDRED
Debug #ONE_HUNDRED_AND_ONE
Debug #ONE_HUNDRED_AND_TWO
```

Здесь вы можете увидеть что, после строки: *'# ONE_HUNDRED = 100',* все константы далее пронумерованы от '100 '. Нумерованные констант используются в основном для программирования графического интерфейса пользователя (см. главу 9), где каждому окну или гаджету нужен собственный идентификатор, перечислить которые константами очень удобно. Это помогает избежать трудностей, в первую очередь анализируя свой собственный код, в особенности когда гаджетов и окон много.

Ну вот я надеюсь, что вам все понятно из данной главы. Если все же есть пробелы, не печальтесь, все со временем заполнится. Я (Станислав Будинов) пытался переводить достаточно понятным языком. Иногда правда приходилось отходить от авторской мысли, чтобы вам казалось, что текст написан изначально на русском языке и вам не пришлось ничего домысливать, но в основном стиль автора сохранен. Автор же данного материала Gary Willoughby.

2

Операторы

Операторы используют для присвоения значений переменным и манипулирования данными, которые в этих переменных содержатся. В этой главе я познакомлю вас со всеми операторами PureBasic и для каждого из них я дам краткий пример описания его функций и использования. Также я подготовил несколько таблиц, показывающих работу более продвинутых операторов, и их работу с данными на уровне двоичных файлов. И

конечно же расскажу о приоритетах операторов(то есть какие из них выполняются в первую очередь, вторую, третью и т. д.)

Введение для операторов

Операторы представляют собой набор функций, которые могут выполнять арифметические операции над числовыми данными, логические операции и выполнять операции со строками. Некоторые операторы являются универсальными, а это означает, что они могут быть использованы на разные типы данных и могут выполнять различные функции. Например, оператор **(=)** равно может быть использован для присвоения значения переменной. Но его также можно использовать как оператор равенства для проверки, что две переменные или значения равны.

Оператор = (Равно)

Итак, вот его первое использование (Присвоение значения переменной):

```
LongVariable.l = 1
```

Второй способ (Сравнение равенства между двумя выражениями, переменными или значениями):

```
LongVariable.l = 1
If LongVariable = 1
  Debug "Yes, LongVariable does equal 1"
EndIf
```

Вы в первый раз, увидели команду ***IF***, но не стоит пугаться. Эта команда позволяет выполняться коду **Debug "Yes, LongVariable does equal 1"** только при условии если **LongVariable.l = 1**. Команда ***Endif*** является неотъемлемой к команде ***IF .*** Она показывает конец всех действий связанных с условной командой ***IF*** . Обязательно попробуйте поменять значение в первой строчке кода (например так: **LongVariable.l=4**) и посмотрите результат.

Оператор + (Плюс)

Оператор + используется для связки(суммирования) строк, а так же для суммирования чисел.Вот пример суммирования чисел:

```
NumberOne.l = 50
NumberTwo.l = 25
NumberThree.l = NumberOne + NumberTwo
Debug NumberThree
```

Число, которое будет в окне отладки: 75. То есть мы присвоили значение первой переменной 50, затем присвоили второй переменной 25. А третьей переменной назначили сумму первых двух переменных. А вот еще один способ сложения двух чисел и присвоение переменной *NumberOne.l*:

```
NumberOne.l = 50 + 25
Debug NumberOne
```

А вот еще один способ использования оператора:

```
NumberOne.l = 50
NumberOne + 25
Debug NumberOne
```

Здесь мы присвоили переменной значение 50, а потом прибавили еще 25. В итоге в окне отладки будет 75.

Теперь пример сложения строк:

```
StringOne.s = "Mary had a"
StringTwo.s = " little lamb"
StringThree.s = StringOne + StringTwo
Debug StringThree
```

Мы двум переменным присвоили по строке, а затем объединили в общую строку. Обратите внимание на пробел в строке **" little lamb"** и попробуйте убрать его и запустить. Понятно теперь зачем нужен пробел? Ну ладно, вот еще один пример:

```
StringOne.s = "Mary had a" + " little lamb"
Debug StringOne
```

Если вам надо строковую переменную сложить с каким нибудь текстом, то это можно сделать примерно так:

```
StringOne.s = "Mary had a"
StringOne + " little lamb"
Debug StringOne
```

Обязательно попрактикуйтесь со складываением строк, чисел, перед изучением следующего оператора.

Оператор - (Минус)

Оператор - работает так же как оператор плюс, но вместо операции складывания использует операцию вычитания. Кроме того этот оператор не работает со строками, вот пример:

```
NumberOne.l = 50
NumberTwo.l = 25
NumberThree.l = NumberOne - NumberTwo
Debug NumberThree
```

В итоге в переменной NumberThree.l окажется 25 То есть разность двух переменных NumberOne - NumberTwo . А вот еще один пример:

```
NumberOne.l = 50
NumberOne - 10
Debug NumberOne
```

Здесь NUMBERONE присваивается значение 50, затем NUMBERONE уменьшается на 10, используя оператор минус . Новое значение NUMBERONE (40) и будет в окне вывода отладки.

Оператор * (Умножение)

Оператор умножения используется для умножения двух значений и как оператор минус не может работать со строками. Демонстрация работы этого оператора:

```
NumberOne.l = 5
NumberTwo.l = 25
NumberThree.l = NumberOne * NumberTwo
Debug NumberThree
```

Окно отладки покажет результат 125 , поскольку в этом примере мы умножили NUMBERONE на NumberTwo (5 * 25 = 125). И еще один пример перемножения переменной:

```
NumberOne.l = 50
NumberOne * 3
Debug NumberOne
```

Здесь NUMBERONE присваивается значение 50 , затем NUMBERONE умножается на 3, используя оператор умножения. Новое значение NUMBERONE (150) и будет в отладочном окне.

Оператор / (Деление)

Оператор / математической оператор, который используется для деления чисел. Он тоже не работает со строками.
Вы, наверное, догадались, как использовать его, читая прошлые примеры, но все же я покажу примеры:

```
NumberOne.l = 100
NumberTwo.l = 2
NumberThree.l = NumberOne / NumberTwo
Debug NumberThree
```

Здесь NUMBERONE присваивается значение 100. NumberTwo присваивается значение 2. Затем мы
делим NUMBERONE (100) на NumberTwo (2) и сохраняем результат в NumberThree. И в отладочном окне мы можем видеть 50
И еще один пример:

```
NumberOne.l = 50
NumberOne / 5
Debug NumberOne
```

Здесь NUMBERONE присваивается значение 50 , далее используя оператор деления разделим это значение на 5. В переменной NUMBERONE будет значение 10, о чем и свидетельствует окно отладки.

Оператор & (Битовый AND)

Битовые операторы манипулируют числами на уровне двоичных файлов. Если вы мало знакомы с бинарными данными, вы можете обратиться к главе 13 (пристальный взгляд на числовые типы данных), где дается полное объяснение. Битовые операторы не могут быть использованы с дробными числами или строками.

Как работает:

Битовый оператор **&** сравнивает два двоичных числа и возращает **1 (TRUE)** только в том случае, если у обоих чисел биты равны **1 (TRUE).** Во всех других случаях оператор возращает **0 (FALSE)** Все же для вас подготовлена таблица из которой все станет более понятно.

	False	True	False	False	False	True	False	True
Бинарное значение 77	0	1	0	0	1	1	0	1
Бинарное значение 117	0	1	1	1	0	1	0	1
Конечный результат 69	0	1	0	0	0	1	0	1

В таблице можно увидеть два числа 77 и 117, которые будут обрабатываться оператором &. После обработки оператором &, конечным результатом будет число 69 . Чтобы объяснить, как это значение достигается, нужно смотреть на каждую колонку битов сверху вниз. Взгляните на крайний правый столбец: этот столбец при обработке оператора &, устанавливается как TRUE (истина), поскольку в строчках обоих чисел для этого столбца стоят единички. Если вы посмотрите на крайний левый столбец, то увидете, что название у него False (ложь), поскольку у чисел в этом столбце значение ноль. Просмотрите внимательно таблицу, а так же прочитайте еще раз внимательно как работает оператор. Теперь можно привести пример использования оператора **&**

```
NumberOne.b = 77
NumberTwo.b = 117
NumberThree.b = NumberOne & NumberTwo
Debug NumberThree
```

В этом маленьком примере двум переменным присваиваются числа, которые будут обрабатываться оператором &, а в переменную NumberThree запишется результат и будет выведен в окно отладки. Значение будет равно 69. А вот еще один пример:

```
NumberOne.b = 77
NumberOne & 117
Debug NumberOne
```

Здесь NUMBERONE присваивается значение 77, и обрабатывается с помощью оператора & значением 117, а результат сохраняется в NUMBERONE. И значение сохраненное в NUMBERONE выводится в окно отладки. Далее идет обобщающая таблица работы оператора &.

Левый бит	Правый бит	Результат
0	0	0
0	1	0
1	0	0
1	1	1

Оператор | (Битовый OR)

Битовый оператор | сравнивает два двоичных числа и возращает **1 (TRUE)** в том случае, если хотя бы у одного из чисел бит равен **1 (TRUE).** Во всех других случаях оператор возращает **0 (FALSE)** И конечно же для вас подготовлена таблица из которой все станет более понятно:

	False	True	True	True	False	True	True	False
Бинарное значение 54	0	0	1	1	0	1	1	0
Бинарное значение 102	0	1	1	0	0	1	1	0
Конечный результат 118	0	1	1	1	0	1	1	0

В таблице можно увидеть два числа 54 и 102, которые будут обрабатываться оператором |. После обработки оператором |, конечным результатом будет число 118 . Чтобы объяснить, как это значение достигается, нужно смотреть на каждую колонку битов сверху вниз. Взгляните на крайние столбцы: эти столбцы при обработке оператора |, устанавливается как FALSE (Ложь), поскольку в строчках обоих чисел для этого столбца стоят нули. Если вы посмотрите на любые столбцы с названием TRUE, то увидете, что хотя бы в одном строчке столбца есть единичка. Просмотрите внимательно таблицу, а так же прочитайте еще раз внимательно как работает оператор. Теперь можно привести пример использования оператора |

```
NumberOne.b = 54
NumberTwo.b = 102
NumberThree.b = NumberOne & NumberTwo
Debug NumberThree
```

В этом маленьком примере двум переменным присваиваются числа, которые будут обрабатываться оператором |, а в переменную NumberThree запишется результат и будет выведен в окно отладки. Значение будет равно 118. А вот еще один пример:

NumberOne.b = 54
NumberOne | 102
Debug NumberOne

Здесь NUMBERONE присваивается значение 54, и обрабатывается с помощью оператора | значением 102, а результат сохраняется в NUMBERONE. И значение сохраненное в NUMBERONE выводится в окно отладки. Далее идет обобщающая таблица работы оператора |.

Левый бит	Правый бит	Результат
0	0	0
0	1	1
1	0	1
1	1	1

Оператор ! (Побитовый XOR)

Оператор ! сравнивает два двоичных числа, и результат его работы будет FALSE (ложь), если у двух чисел биты равны единице, либо у двух чисел биты равны нулю. Во всех остальных случаях оператор возращает TRUE (Истина). И конечно таблица:

	False	True	True	False	True	True	False	False
Бинарное значение 38	0	0	1	0	0	1	1	0
Бинарное значение 74	0	1	0	0	1	0	1	0
Конечный результат 108	0	1	1	0	1	1	0	0

В таблице можно увидеть два числа 38 и 74, которые будут обрабатываться оператором !. После обработки оператором !, конечным результатом будет число 108 . Чтобы объяснить, как это значение достигается, нужно смотреть на каждую колонку битов сверху вниз. Взгляните на столбцы в которых сверху идут одинаковые биты: эти столбцы при обработке оператора !, устанавливается как FALSE (Ложь). Если вы посмотрите на любые столбцы с названием TRUE, то увидете, что биты в этих столбцах разные. Просмотрите внимательно таблицу, а так же прочитайте еще раз внимательно как работает оператор. Теперь можно привести пример использования оператора !

NumberOne.b = 38
NumberTwo.b = 74
NumberThree.b = NumberOne ! NumberTwo
Debug NumberThree

В этом маленьком примере двум переменным присваиваются числа, которые будут обрабатываться оператором **!**, а в переменную NumberThree запишется результат и будет выведен в окно отладки. Значение будет равно 108. А вот еще один пример:

NumberOne.b = 38
NumberOne ! 74
Debug NumberOne

Здесь NUMBERONE присваивается значение 38, и обрабатывается с помощью оператора **!** значением 74, а результат сохраняется в NUMBERONE. И значение сохраненное в NUMBERONE выводится в окно отладки. Далее идет обобщающая таблица работы оператора **!**.

Левый бит	Правый бит	Результат
0	0	0
0	1	1
1	0	1
1	1	0

Оператор ~ (Битовый NOT)

Оператор **~** представляет полное отрицание битов. То есть то что было нулем, станет единицей и наоборот. Вот небольшой пример его применения:

NumberOne.b = 43
NumberTwo.b = ~NumberOne
Debug NumberTwo

Здесь переменной 'NUMBERONE' присваивается значение '43 '. Далее мы создадим переменную' NumberTwo и присвоим ей значение 'NUMBERONE ', но в перевернутом бинарном уровне с использованием оператора ' ~ '. Это значение (которое должно быть '-44 ') затем выводится в окно вывода отладки.

	Инв	ерти	рова	нное	зна	чени	е	
Бинарное значение 43	0	0	1	0	1	0	1	1
Конечный результат -44	1	1	0	1	0	1	0	0

Как вы поняли каждый бит инвертируется или говоря проще становится противоположным. Если вы число -44 обработаете опять оператором ~ то получите число 43.

Оператор << (Сдвиг влево)

Операторы сдвига бит похожи на побитовые операторы тем, что они манипулировать числами в бинарной уровне. Как следует из их названия они сдвигаются все биты влево или вправо в зависимости от используемого оператора. Вот код, демонстрирующий использование '<<' оператора:

```
NumberOne.b = 50
NumberTwo.b = NumberOne << 1
Debug NumberTwo
```

В этом примере мы переменной NUMBERONE присваиваем значение 50. Затем мы создаем переменную NumberTwo и присваиваем ей значение NUMBERONE, но обработанное оператором <<. Результат этой операции 100 мы и наблюдаем в окне вывода отладки. Чтобы лучше понимать функции этого оператора, надо взглянуть на таблицу, представленную ниже. Как вы можете видеть результат вычисления имеет просто смещение двоичных битов влево от их исходного положения, в данном случае на одно место. При переключении битов влево создаются нули справа от имеющихся битов, а крайние слева биты будут сдвинуты и будут потеряны навсегда.

Значение 50	0	0	1	1	0	0	1	0
Значение 100	0	1	1	0	0	1	0	0

Оператор >> (Сдвиг вправо)

Оператор >> работает точно так же, как оператор <<, но в противоположном направлении. Вот код демонстрирующий использование этого оператора:

```
NumberOne.b = 50
NumberTwo.b = NumberOne >> 1
Debug NumberTwo
```

В этом примере мы назначаем переменной NUMBERONE значение 50. Затем мы создаем переменную NumberTwo и присваиваем ей значение NUMBERONE, но обработанное оператором >>. Результат этой операции 25 мы и наблюдаем в окне вывода отладки.И конечно же посмотрите таблицу:

Значение 50	0	0	1	1	0	0	1	0
Значение 25	0	0	0	1	1	0	0	1

Как вы можете видеть результат вычисления имеет просто смещение двоичных битов вправо от их исходного положения, в данном случае на одно место При переключении разрядов вправо, (как в нашем случае на 1) крайний правый бит пропадает навсегда, а крайний левый заполняется нулем, остальные сдвигаются.

Оператор < (Меньше)

Оператор < используется при сравнении двух переменных или выражений. Если значение слева меньше чем справа, то этот оператор возвращает "True" (1),
в противном случае он вернет FALSE (0). Вот фрагмент кода, демонстрирующий его использование:

```
NumberOne.l = 1
NumberTwo.l = 2
If NumberOne < NumberTwo
Debug "1: NumberOne is less than NumberTwo"
Else
Debug "2: NumberTwo is less than NumberOne"
EndIf
```

Здесь мы проверяем значения двух переменных. Если NUMBERONE меньше NumberTwo (что конечно же так), то выполнится **1:NumberOne is less than NumberTwo** в окне отладки. Если же мы изменим значение 'NUMBERONE' на 3 , вот так:

```
NumberOne.l = 3
NumberTwo.l = 2
If NumberOne < NumberTwo
Debug "1: NumberOne is less than NumberTwo"
Else
Debug "2: NumberTwo is less than NumberOne"
EndIf
```

То в окне отладки мы увидим другое предложение **2: NumberTwo is less than NumberOne**

Оператор > (Больше)

Оператор > используется при сравнении двух переменных или выражений. Если значение слева больше чем справа, то этот оператор возвращает "True" (1),
в противном случае он вернет FALSE (0). Вот фрагмент кода, демонстрирующий его использование:

```
NumberOne.l = 2
NumberTwo.l = 1
If NumberOne > NumberTwo
Debug "1: NumberOne is more than NumberTwo"
Else
Debug "2: NumberTwo is more than NumberOne"
EndIf
```

Здесь мы проверяем значения двух переменных. Если NUMBERONE больше NumberTwo (что конечно же так), то выполнится **1:NumberOne is more than NumberTwo** в окне отладки. Если же мы изменим значение 'NUMBERONE' на 0 , вот так:

```
NumberOne.l = 0
NumberTwo.l = 1
If NumberOne > NumberTwo
Debug "1: NumberOne is more than NumberTwo"
Else
Debug "2: NumberTwo is more than NumberOne"
EndIf
```

То в окне отладки мы увидим другое предложение **2: NumberTwo is more than NumberOne**

Оператор <= (Меньше или равно)

Этот оператор используется при сравнении двух переменных или выражений. Если значение слева меньше или равно правому, то этот оператор вернет "True" (1), в противном случае он возвратит "False" (0). Вот фрагмент кода, демонстрирующий его использование:

```
NumberOne.l = 0
NumberTwo.l = 1
If NumberOne <= NumberTwo
Debug "1: NumberOne is less than or equal to NumberTwo"
Else
Debug "2: NumberOne is NOT less than or equal to NumberTwo"
EndIf
```

Здесь, как вы сами понимаете, в окне отладки будет **1: NumberOne is less than or equal to NumberTwo** А вот если мы изменим **NumberOne** и сделаем его к примеру 2 или 3 или больше:

```
NumberOne.l = 2
NumberTwo.l = 1
If NumberOne <= NumberTwo
Debug "1: NumberOne is less than or equal to NumberTwo"
Else
Debug "2: NumberOne is NOT less than or equal to NumberTwo"
EndIf
```

То в окошке отладки будет: **2: NumberOne is NOT less than or equal to NumberTwo**

Оператор >= (Больше или равно)

Этот оператор используется при сравнении двух переменных или выражений. Если значение слева больше или равно правому, то этот оператор вернет "True" (1), в противном случае он возвратит "False" (0). Вот фрагмент кода, демонстрирующий его использование:

```
NumberOne.l = 2
NumberTwo.l = 1
If NumberOne >= NumberTwo
Debug "1: NumberOne is more than or equal to NumberTwo"
Else
Debug "2: NumberOne is NOT more than or equal to NumberTwo"
EndIf
```

Здесь,в окне отладки будет **1: NumberOne is more than or equal to NumberTwo** А вот если мы изменим **NumberOne** и сделаем его равным 0:

```
NumberOne.l = 0
NumberTwo.l = 1
If NumberOne >= NumberTwo
Debug "1: NumberOne is more than or equal to NumberTwo"
Else
Debug "2: NumberOne is NOT more than or equal to NumberTwo"
EndIf
```

То в окне отладки будет **2: NumberOne is NOT more than or equal to NumberTwo**

Оператор <> (Не равно)

Этот оператор используется при сравнении двух переменных или выражений. Если значение слева не равно правому, то этот оператор вернет "True" (1), в противном случае он возвратит "False" (0). Вот фрагмент кода, демонстрирующий его использование:

```
NumberOne.l = 0
NumberTwo.l = 1
If NumberOne <> NumberTwo
Debug "1: NumberOne does not equal NumberTwo"
Else
Debug "2: NumberOne does equal NumberTwo"
EndIf
```

Здесь,в окне отладки будет: **1: NumberOne does not equal NumberTwo** поскольку **NumberOne** и **NumberTwo** не равны друг другу, а теперь давайте изменим наш пример так:

```
NumberOne.l = 1
NumberTwo.l = 1
If NumberOne <> NumberTwo
Debug "1: NumberOne does not equal NumberTwo"
Else
Debug "2: NumberOne does equal NumberTwo"
EndIf
```

Здесь оператор возратит False а в окне отладки мы увидим: **2: NumberOne does equal NumberTwo**

Оператор **AND** (логическое и)

Логические операторы применяются для объединения сравниваемых логических выражений. Делается это для большего удобства в написании кода. Данный оператор возратит TRUE, только если оба его сравниваемых значения соответствуют запрашиваемой логике. Посмотрите на этот фрагмент кода:

```
StringOne.s = "The quick brown fox"
NumberOne.l = 105
If StringOne = "The quick brown fox" And NumberOne = 105
Debug "1: Both expressions evaluate to true (1)"
Else
Debug "2: One or both of the expressions evaluate as false (0)"
EndIf
```

По моему все просто, но я попробую объяснить как можно проще:

1) первой переменной мы присвоили строку The quick brown fox.

2) второй переменной присоили число 105

3) далее мы спрашиваем: если строка равна **StringOne = "The quick brown fox"** ***а так же*** **NumberOne = 105,** то оператор вернет True и в окне отладки будет: **1: Both expressions evaluate to true (1)** Но ели бы это было не так то оператор вернет False и в окне мы увидели бы **2: One or both of the expressions evaluate as false (0)**

Оператор **Not** (логическое НЕ)

Оператор **Not** используется для выполнения логического отрицания для выражений или логических значений. Другими словами все, что оценивается как истинно справа возвращается как ложное и наоборот. См. пример:

```
One.l = 1
Two.l = 2
If Not One = 5
  Debug "1: One = 5 is evaluated as true (1)"
Else
 Debug "2: One = 5 is evaluated as false (0)"
EndIf
If Not Two = 2
 Debug "1: Two = 2 is evaluated as true (1)"
Else
 Debug "2: Two = 2 is evaluated as false (0)"
EndIf
```

Двум переменным присваиваются значения. Далее идет условие: **If Not One = 5** Давайте прочитаем его более понятно. Мысленно переставим два слова Not и One, и прочитаем:

ЕСЛИ One НЕ РАВНО 5

ТОГДА **Debug "1: One = 5 is evaluated as true (1)"**

ИНАЧЕ **Debug "2: One = 5 is evaluated as false (0)"**

С остальными условиями так же.

Оператор Or (логическое или)

Данному оператору нужно хотя бы одно выполняемое условие из двух или более и он вернет TRUE. См пример:

```
StringOne.s = "The quick brown fox"
NumberOne.l = 105
If StringOne = "The quick brown fox" Or NumberOne = 100
Debug "1: One or more expressions evaluate to true (1)"
Else
Debug "2: Both of the expressions evaluate as false (0)"
EndIf
```

Здесь сравниваются два условия и поскольку одно из них истинно, этого достаточно для того чтобы оператор возратил True, а в окне отладки высветилась надпись: **1: One or more expressions evaluate to true (1)**

Оператор XOr (Логический XOR)

Оператор 'XOR' используется для проверки сразу двух выражений. Посмотрите на этот кусок кода:

```
StringOne.s = "The quick brown fox"
NumberOne.l = 105
If StringOne = "The quick brown fox" XOr NumberOne = 105
  Debug "1: Only one expression is true (1)"
Else
  Debug "2: The expressions are either both true (1) or both false (0)"
EndIf
```

Если только одно из выражений является True, то результат справедлив. Если оба выражения являются либо True либо False, то оператор Xor возвращает false. Для того чтобы разобраться в этом примере, вам надо запустить этот код и поизменять значения переменных.

Оператор % (Остаток)

Этот оператор делит два числа и возращает остаток от деления. Вот пример:

```
NumberOne.l = 20 % 8
Debug NumberOne
```

Если 20 разделить на 8, то остаток будет 4

Оператор () (Скобки)

Скобки не похожи на другие операторы, поскольку они никогда не возвращают никаких результатов. Они используются для определения порядка выполнения операций. Общее правило гласит, что выражение в скобках выполняется в первую очередь. А теперь примеры:

```
NumberOne.l = 2 * 5 + 3
Debug NumberOne
```

В данном случае будет 13, а теперь другой пример:

```
NumberOne.l = 2 * (5 + 3)
Debug NumberOne
```

А теперь мне кажется вам должно быть понятно, что результат 16

Приоритеты операторов

Приоритет операторов это термин, который означает порядок, в котором будут выполняться действия во время компиляции, а так же при выполнении программы. Посмотрите на таблицу, самый высокий приоритет у оператора под номером 1, а самый низкий у номера 8. Отсюда думается мне все понятно:

Приоритет	Операторы
1	( )
2	~
3	<< >> % !
4	| &
5	* /
6	+ -
7	> >= < <= = <>
8	And Or Not XOr

В этом примере:

```
Debug 3 + 10 * 2
```

оператор умножения имеет больший приоритет, поэтому выполняется первым. Результатом будет число 23. Чтобы настроить приоритет операторов, используйте скобки.Например, если мы хотим того, чтобы выполнить оператор сложения сначала, то это будет таким образом:

```
Debug (3 + 10) * 2
```

Результатом будет 26

Примечания по выражениям

Когда компилятор оценивает выражение между целыми числами и дробными, то иногда изменяет тип данных , чтобы определять их должным образом. Если выражение содержит плавающую точку или говоря простым языком является дробным, то тогда, каждая часть выражения преобразуется в дробное по правилам компилятора. В таблице ниже есть пример того, как PureBasic оценивает эти выражения при разных определенных условиях.Если Вы находите, что странные результаты возвращаются из операторов или выражений, или что число возвращенное не имеет ожидаемого типа, то неплохо было бы проверить выражение, чтобы удостовериться, что они соответствуют правилам компилятора.

Правила вычисления выражений

Примеры выражений	Правило оценки
a.l = b.l + c.l	‘b’ и ‘c’ остаются нетронутыми как прежде во время оценки, тип Long назначен 'a'.
a.l = b.l + c.f	Поскольку 'c' дробное, 'b' преобразуется в дробное перед оценкой. Далее ‘b’ суммируется с ‘c’ и результат присваивается в 'a' с типом Long
a.f = b.l + c.l	‘b’ и ‘c’ и остаются как прежде во время оценки. Получающееся число с типом Long, возвращенное операцией сложения, преобразуется в дробное и будет назначено 'a'
a.l = b.f + c.f	‘b’ и ‘c’ остаются нетронутыми. Получающееся дробное число, возвращенное операцией сложения, преобразуется в тип Long и присваивается 'a'.

Краткий справочник по операторам

Операторы	Описание
=	Равно. Можно использовать двумя способами: 1) назначение одного значения другому. 2) когда нужно проверить равенство значений, в зависимости от этого оператор возвращает True или False
+	Используется для суммирования чисел или строк
-	Используется для вычитания значений (не используется со строками)
*	Используется для умножения значений (не используется со строками)
/	Используется для деления значений (не используется со строками)
&	Битовый оператор **&** сравнивает два двоичных числа и возвращает **1 (TRUE)** только в том случае, если у обоих чисел биты равны **1 (TRUE).** Во всех других случаях оператор возвращает **0 (FALSE)**
|	Битовый оператор | сравнивает два двоичных числа и возвращает **1 (TRUE)** в том случае, если хотя бы у одного из чисел бит равен **1 (TRUE).** Во всех других случаях оператор возвращает **0 (FALSE)**
!	Оператор **!** сравнивает два двоичных числа, и результат его работы будет

	FALSE (ложь), если у двух чисел биты равны единице, либо у двух чисел биты равны нулю. Во всех остальных случаях оператор возвращает TRUE (Истина).
~	Оператор ~ представляет полное отрицание битов. То есть то что было нулем, станет единицей и наоборот.
<	Оператор возвращает True если левое значение меньше правого.
>	Оператор возвращает True если левое значение больше правого.
< =	Оператор возвращает True если левое значение меньше или равно правому
> =	Оператор возвращает True если левое значение больше или равно правому
<>	Оператор возвращает True если левое значение не равно правому
And	Логические операторы применяются для объединения сравниваемых логических выражений. Данный оператор возратит TRUE, только если оба его сравниваемых значения соответствуют запрашиваемой логике.
Or	Данному оператору нужно хотя бы одно выполняемое условие из двух или более и он вернет TRUE.
Not	Оператор используется для выполнения логического отрицания для выражений или логических значений. Другими словами все, что оценивается как истинно справа возвращается как ложное и наоборот.
Xor	Оператор используется для проверки сразу двух выражений.
<<	Оператор сдвига битов влево
>>	Оператор сдвига битов вправо
%	Оператор возращает остаток от деления.
( )	Они используются для определения порядка выполнения операций.

3

Условные операторы и циклы

В этой главе я расскажу вам про условные операторы и циклы. Они - главные части любой программы и являются определяющими элементами выполнения программы. Я начну с объяснения булевых значений и как PureBasic обрабатывает их и после этого перейду к условным операторам таким как, '**IF**' и '**Select**', которые используются, чтобы определить выполнение программы на основе различных условий. Закончу главу с объяснениями и примерами различных циклов, которые доступны для использования в PureBasic. Как всегда будут даны полные объяснения наряду с большим кол-вом примеров.

Булевая логика

Сначала давайте окунемся в историю. Джордж Буль был математиком и философом, который изобрел форму алгебры, называющуюся теперь булевой алгеброй. Логику этой формы алгебры назвали Булевой в честь Джорджа Буля. Эта форма давно стала основой всех современных компьютеров.
Что удивительно, так это то, что Джордж Буль изобрел эту форму примерно семьдесят лет назад до создания первого компьютера, который начал использовать его! По сути, вся система вращается вокруг двух значений, TRUE(истина) и False(ложь). Эти два значения, проверяются с помощью логических операций, для определения результата. На самом деле это не так сложно. Три самые основные логические операции были (и остаются) AND, OR и NOT. Именно эти три оператора, легли в основу алгебры Буля, добавим операции, необходимые для выполнения сравнений и элементарную математику. (В главе 3 подробно рассказывается про эти логические операции). У PureBasic нет булевого типа данных , в отличие от некоторых языков, такие как C++. Так в PureBasic, чтобы выразить истину или ложь мы используем числа. ‘1’ истина и ‘0’ ложь, помните это! Поскольку мы используем эти числовые значения для представления TRUE(истина) и FALSE(ложь), то неплохо было бы использовать в PureBasic встроенные константы. Это поможет сделать ваш программный код легко читаемым и понимаемым в дальнейшем. Вот две константы:

```
#True
#False
```

True имеет значение '1' и # False имеет значение '0'.

Почти все команды в PureBasic возвращают значение. Иногда это значение-математическая функция или это может быть состоянием окна, которое Вы только что создали. Эти значения возвращаются, чтобы иметь возможность проверить их при случае. Посмотрите пожалуйста на этот код(авторский код изменен на русский для большего понимания):

```
if Создано окно True
    тогда я размещу на нем мои кнопки и графику
else
    есть какие то проблемы с созданием окна
end the program
```

Это не реальный компилируемый код, это только идея, в которой я пытаюсь объяснить суть логики. И так я спрашиваю:

Создано ли мое окно?

Если значение равно **TRUE** , тогда я могу разместить на моем окне мою графику и кнопки.

Если значение не равно **TRUE**, то есть оно равно **FALSE**, то вылетит сообщение что есть какие то проблемы.

Это первое впечатление о важности наличия истинных и ложных значений. А теперь пора в следующий раздел.

Оператор IF

Команда IF используется для управления потоками программы на основе необходимой логики.Иногда, когда программа еще не запущена, вы
можете получить неправильный запуск или ошибки и неплохо было бы иметь возможность для обработки таких вещей.

Конструкция оператора IF

Оператор используется для проверки на истинность значения. Если он получает значение TRUE(истина), то он сразу же выполняет кусок кода после первой строки: **IF.....** Если он не получит этого значения, он будет выполнять другой отдельный кусок кода сразу же после **Else**. Давайте посмотрим на примере:

```
a.l = 5
If a = 5
    Debug "A true value was found"
Else
   Debug "No true value was found"
EndIf
```

Здесь оператор 'IF' тестирует равна ли переменная "a" числу "5". Он возвращает значение TRUE, поэтому выполняется код в сроке сразу после строки IF. Если бы это сравнение оказалось FALSE, то выполнился бы код сразу после строки с командой 'Else'. Чтобы определить конец работы для оператора IF, нужна команда ENDIF.

О значениях

В PureBasic ‘1’ =True и ‘0’ =False. Оператор '**IF**' распознает эти два значения и использует в своей структуре в зависимости от логики.

Обратите внимание при изучении 'IF' : команда ENDIF обязательна при создании IF, другое дело команду ELSE можно использовать по надобности. Так например следующий пример покажет вам это:

```
a.l = 5
If a = 5
  Debug "A true value was found"
EndIf
```

Единственный недостаток этого примера: нет команд при возврате оператором False . Вам конечно решать нужны ли команды при срабатывании False в вашем коде оператором. А мы попросту возьмем следующий пример, в котором задаем условия для оператора, определяющего: имеет ли переменная какое либо значение?

```
Beads.l = 5
If Beads
 Debug "переменной присвоено значение"
Else
```

```
  Debug "переменной не присвоено значение"
EndIf
```

Здесь оператор IF проверяет имеет ли переменная BEADS значение отличное от 0. В нашем примере переменная имеет значение и соответствующий кусок кода выполняется. Попробуйте изменить значение на 0 и запустить снова.

Давайте взглянем на более сложный пример с использованием оператора IF. Если вам сложны для понимания некоторые операторы, прочитайте главу 3.

```
Value1.l = 10
Value2.l = 5
Value3.l = 1
If  Value1 >= 10 And (Value2 / 5) = Value3
  Debug "The expression evaluates as true"
Else
  Debug "The expression evaluates as false"
EndIf
```

Здесь мы проверяем: **value1** больше или равно 10 а так же **Value2** разделенное на 5 равно **Value3**. Как вы можете видеть: выражения проверяемые могут быть довольно сложными.

Команда ELSEIF

Другая команда, которая может быть использоваться в рамках оператора "IF", это команда ElseIf. Как следует из названия команды, она представляет собой сочетание Else и IF. Если оригинал IF оценивается как False, то проверяется кусок кода со строкой ElseIf. Однако, в отличие от Else, для того чтобы оценивать альтернативные условия, можно использовать несколько команд ElseIf. Я покажу пример с использованием одной команды ElseIf, но вам ничего не мешает попрактиковаться и добавить еще несколько условий с командами ElseIf:

```
NumberOfBeads.l = 10
If NumberOfBeads < 5
  Debug "The variable has a value below '5'"
ElseIf NumberOfBeads > 5
  Debug "The variable has a value above '5'"
Else
  Debug "The variable has a value of '5'"
EndIf
```

Здесь мы проверяем значение переменной NumberOfBeads. В первой строке проверки тестируется: является ли это значение меньше "5". Поскольку возращается False, программа переходит к части кода ElseIf. В строке ElseIf проверка возвращает TRUE, потому что NumberOfBeads больше, чем "5". Значит кусок кода со строкой ELSE выполняться не будет.
Команда ElseIf по сути является отличным способом расширить оператор IF, чтобы проверить множество значений. Единственным недостатком является то, что когда

большое число этих команд используется, код станет слишком огромен. Когда нужно проверить много значений, предпочтительней использовать оператор SELECT

Немного обобщим:

IF.....

код

ELSEIF.....

код

ELSEIF.....

код

ELSE.....

код

ENDIF

1)Проверяется строка IF

2) Если возратило False, то проверяем строки ELSEIF, пока не возратится TRUE

3) Если везде возратилось FALSE, то выполняем команду ELSE

4) Выходим из проверки командой ENDIF

Оператор SELECT

Оператор SELECT можно считать прямым дополнением к "IF". Он удобен только если необходимо проверить несколько условий . Позвольте мне показать вам пример правильного синтаксиса и объяснить его использование:

```
Days.l = 2
Select Days
  Case 0
    Debug "0 Days"
  Case 1
    Debug "1 Day"
  Case 2
    Debug "2 Days"
  Default
    Debug "Over 2 Days"
EndSelect
```

Оператор начинается с команды SELECT, которой задается выражение или переменная для тестирования, в данном случае это переменная **Days**. Командам CASE задаются значения. Если какое нибудь значение равно переменной, то выполняется соответствующий код ниже команды Case, задающей это значение. В нашем примере переменная **Days** имеет значение 2. Сначала выполнится сравнение Case 0, потом Case 1, далее Case 2 и поскольку **Days** имеет значение 2**,** то выполнится код **Debug "2 Days".** И оператор SELECT закончит свою работу завершающей командой ENDSELECT. Вы заметили, что последней командой в сравнении стоит Default. Это кусок кода, который выполняется, если все другие CASE возвратят FALSE, своего рода как Else в рамках IF.

Проверка на наличие нескольких значений

SELECT можно использовать для проверки на множество разных значений и при этом код будет краток. Вот пример:

```
Weight.l = 12
Select Weight
  Case 0
    Debug "No Weight"
  Case 1, 2, 3
    Debug "Light"
  Case 4 To 15
    Debug "Medium"
  Case 16 To 30
    Debug "Heavy"
  Default
    Debug "Massive"
EndSelect
```

Здесь вы можете увидеть, что после команды CASE можно задавать несколько значений через запятую, и конечно если какое нибудь из этих значений будет равно тому, что задано в SELECT, выполнится соответствующий код ниже CASE. Кроме того вы наверно заметили, что после CASE, можно задавать интервал значений: **4 TO 15,** то есть от 4 до 15. Интервал задается командой TO. А теперь еще один пример оператора SELECT в консольном приложении.

```
 If OpenConsole()
    PrintN("1. Official PureBasic Home")
    PrintN("2. Official PureBasic Forums")
    PrintN("3. PureArea.net")
    PrintN("")
    PrintN("Enter a number from 1 To 3 and press Return: ")
 Destination.s = Input()
Select Destination
  Case "1"
    RunProgram("http://www.purebasic.com")
  Case "2"
    RunProgram("http://forums.purebasic.com")
  Case "3"
    RunProgram("http://www.purearea.net")
```

```
EndSelect
EndIf
End
```

В этом примере я использовал несколько новых команд, с которыми вы не знакомы, но я думаю, что это не так сложно для понимания. Программа четко демонстрирует использование SELECT. Новые команды будут объяснены попозже, но я думаю, можно понять, что происходит, глядя на их описательные названия. По сути программа ожидает от вас нажатия на клавиши 1 или 2 или 3, в зависимости от того на какой сайт вам надо попасть, а оператор SELECT обрабатывает ваш выбор и функция RunProgram() запускает нужный сайт.В качестве примечания хотелось бы сказать, что я использовал оператор IF для проверки возможности запуска консоли. Если функция OpenConsole () возвращает TRUE, откроется окно консоли и выполнится код внутри оператора IF-ENDIF. Если функция вернет FALSE,приложение просто завершится. Не печальтесь, если чего нибудь из прошлого примера вы не поняли, со временем к вам придут знания и опыт.

Циклы

Чтобы иметь возможность непрерывно получать и обрабатывать данные, используют циклы. Во всех программах, которые используют GNU, используются циклы для управления и разработкой интерфейса, а так же для слежением за вводом от пользователя. Например, в IDE PureBasic используется очень много циклов, для контроля за нажатиями клавиш и кнопок мыши, для обновления дисплеев и т.д. Циклы также отличный способ обработки больших объемов данных, в том числе массивов или связанных списков.

Цикл FOR

Первый цикл, о котором я буду говорить, является вероятно самым известным и возможно наиболее используемый из всех циклов это **FOR.** Эти циклы, иногда называемые циклами FOR / NEXT, являются отличным способом выполнения обработки данных. Когда вы нуждаетесь в переменной приращения, то есть использовать ее как счетчик например для индекса массива . Вот пример простого цикла:

```
For x.l = 1 To 10
  Debug x
Next x
```

В этом примере мы создали цикл, используя команду FOR. Переменная X с типом long будет началом нашего цикла и мы присвоили этой переменной "1", а следовательно и старт нашего цикла будет с числа "1". Далее вводится диапазон командой TO, верхним пределом которого я установил ‘10’ , это число будет концом нашего цикла. Последней строчкой нашего кода, мы устанавливаем команду NEXT(СЛЕДУЮЩИЙ), все с той же переменной X. Эта команда будет увеличивать цикл на единицу. Когда переменная достигнет большего числа ,чем то что заложено верхним пределом то есть больше десяти "10", цикл завершит свою работу. Ну и для наглядности, внутрь цикла мы запехнули команду Debug X, которая выводит результат работы цикла в отладочное окно.Может не совсем понятно? Попробую разжевать: И так цикл- это петля. Все что лежит внутри цикла, выполняется столько раз, сколько мы заложим в строчке с командой FOR. В нашем случае 10. А теперь поменяйте строчку кода **x.l = 1 To 10** вот на такую **x.l = 5 To 11** и

посчитайте сколько результатов в отладочном окне.Думаю теперь понятно.
Вот другой пример использования цикла, для легкого перечисления массива:

```
Dim Names.s(3)
 Names(0) = "Gary"
 Names(1) = "Sara"
 Names(2) = "Stella"
 Names(3) = "MiniMunch"
For x.l = 0 To 3
  Debug Names(x)
Next x
```

Все значения массива доступны с использованием индексов, и эти показатели всегда начинаются с нуля. Циклы FOR легко использовать с массивами, это дает нам небольшой код по размеру, независимо от того, какой размер массива. Больший массив просто потребует больший диапазон в первой строке цикла FOR
Циклы FOR могут быть построены так же с использованием выражений:

```
  StartVar.l = 5
  StopVar.l = 10
For x = StartVar - 4 To StopVar / 2
 Debug x
Next x
```

и, конечно, циклы могут быть вложенными, если вам нужно обработать многомерные массивы:

```
Dim Numbers.l(2, 2)
Numbers(0, 0) = 1
Numbers(0, 1) = 2
Numbers(0, 2) = 3
Numbers(1, 0) = 4
Numbers(1, 1) = 5
Numbers(1, 2) = 6
Numbers(2, 0) = 7
Numbers(2, 1) = 8
Numbers(2, 2) = 9
For x = 0 To 2
   For y = 0 To 2
     Debug Numbers(x, y)
   Next y
Next x
```

В прошлом примере получился цикл в цикле. Чтобы вам было проще понять поясню на примере часов: представьте себе что, большой цикл это часовая стрелка, а маленький цикл это минутная стрелка помотрим сколько движений пройдет минутная стрелка(это будет у нас Z) за два часа (X):

```
  z=1
For x=1 To 2
  For y=1 To 60
```

```
    Debug z
    z=z+1
  Next y
Next x
```

То есть за одно прохождение большого цикла, маленький цикл пройдет 60 раз.А поскольку в нашем примере диапазон большого цикла равен 2, то маленький цикл пройдет два раза по 60, то есть 120. Данный пример обязательно загрузите, если что то непонятно с циклами.

Вы можете сделать столько циклов сколько пожелаете без всяких ограничений. До сих пор вы видели, что нарастание цикла было на 1, но мы можем это изменить по своему желанию с помощью команды STEP (шаг):

```
For x.l = 0 To 10 Step 2
  Debug x
Next x
```

Обратите внимание на команду STEP, ее мы записываем в той же строчке что и FOR. Команда STEP указывает нам в данном примере, что счетчик должен увеличиваться на 2 при каждом прохождении цикла.Если вы запустите этот пример и посмотрите в окно отладки, вы увидите значения, кратные 2.

Цикл ForEach

Этот вид цикла отличается от всех других тем, что он работает только со связанными списками. Синтаксис очень похожий на "FOR" за исключением того, что не требуется указывать диапазон, достаточно указать имя списка. Вот простой пример:

```
NewList Shopping.s()
  AddElement(Shopping())
  Shopping() = "Bunch of bananas"
  AddElement(Shopping())
  Shopping() = "Tea bags"
  AddElement(Shopping())
  Shopping() = "Cheese"

ForEach Shopping()
  Debug Shopping()
Next
```

В этом примере определен связанный список и добавлено несколько элементов. Я использую цикл ForEach к этому списку и все его содержимое вывожу в окно вывода отладки. Как вы можете видеть, синтаксис очень простой. Цикл начинается с команды ForEach , за которым идет название списка. Конец цикла определяется с помощью команды Next . Код, который находится между этими двумя командами выполняется, пока цикл не дойдет до конца связанного списка. Далее цикл завершит работу. Цикл ForEach работает для всех типов связанных списков, в том числе и для структурированных связанных списков. Об этом больше будет рассказано в главе 5.

Цикл While

Этот цикл довольно удобный для определенных целей. Команда While задает начало цикла. В этой же строчке задается условие при котором цикл готов работать и выполняться до тех пор, пока это условие истинно. Команда Wend показывает конец петли цикла. То есть все, что находится между командами While - Wend будет выполняться до тех пор, пока выполняется условие, записанное в строке вместе со строкой While. Вот пример:

```
Monkeys.l = 0
While Monkeys < 10
  Debug Monkeys
  Monkeys + 1
Wend
```

Этот цикл будет выполняться до тех пор, пока переменная **Monkeys** будет меньше 10. Сначала эта переменная равна 0. При Каждом прохождении цикла **Monkeys** увеличивается на единицу, то есть как только **Monkeys** достигнет 10, цикл прервется. Запустите данный код и будет все понятно.А теперь пример цикла который не будет работать, поскольку условие не соответствует:

```
 Monkeys.l = 20
While Monkeys < 10
   Debug "This code is never executed"
Wend
```

Цикл Repeat

Такие циклы почти противоположны циклам While. Цикл начинается с Repeat и заканчивается одним из двух способов. Первый способ заключается в использовании условного выражения Until, а второй способ использует безусловное выражение Forever. Я полностью объясню оба направления , начиная с Repeat-Until:

```
Bananas.l = 0
Repeat
  Debug Bananas
  Bananas + 1
Until Bananas > 10
```

В отличии от цикла While, цикл Repeat будет выполняться пока условие записанное в строке Until не станет истинным. То есть пока условие не выполняется**,** цикл будет работать. Кроме того циклы Repeat, вне зависимости от условий, выполняются хотя бы один раз.Вот пример:

```
Bananas.l = 20
Repeat
  Debug Bananas
Until Bananas > 10
```

Кол-во **Bananas** = 20 и это совпадает с условием, записанным в строке с командой Until. Значит, как только цикл пройдет один раз, он завершится.Далее пример бесконечного цикла, но прежде чем запустить его: ВНИМАНИЕ! Зайдите в меню Меню **Debugger**, там есть строка меню **Kill Program**. Это команда - закилять программу вне зависимости от выполнения. Поскольку как только вы включите бесконечный цикл, выйти из него

сможете только закиляв программу, либо выключив системными средствами. Ну ладно теперь пример:

```
Counter.l = 0
Repeat
  Debug Counter
  Counter + 1
ForEver
```

Ручная остановка циклов

Иногда, когда вы используете циклы в вашей программе вы можете столкнуться с проблемой непреднамеренными непрерывными циклами. Это может вызвать проблемы в ваших программах, поскольку эти петли могут остановить всю остальную работу до их выхода.В PureBasic IDE делается это легко. Если вам нужно остановить запущенные программы вручную, то просто нажмите **Kill Program** на панели инструментов IDE, это тот, который выглядит как череп (в PureBasic 4.40, значок черепа, был заменён на красный крестик) или используйте команду меню Debugger-> Kill Program. Это не только остановит цикл, но и сделает выход из всей программы.

Управление циклами используя Break и Continue

В любое время все эти различные типы циклов могут управляться двумя командами: Break и Continue.Я объясню комаанду Break в первую очередь.
Если команда Break используется в любом месте любого типа циклов, то цикл завершает работу как только столкнется с этой командой. В случае вложенных циклов, есть необязательный параметр, который может быть добавлен к команде Break, который указывает, скольким циклам завершиться. А теперь пример команды Break:

```
For x = 1 To 10
  If x = 5
    Break
  EndIf
    Debug x
Next x
```

В данном примере цикл прервался, как только сработало условие x=5. А теперь пример вложенных циклов с командой Break и с необязательным параметром для нее:

```
For x.l = 1 To 10
    Counter = 0
Repeat
   If x = 5
      Break 2
   EndIf
    Counter + 1
Until Counter > 1
    Debug x
Next
```

Как только x будет равен 5, завершатся оба цикла, поскольку параметр Break равен 2.

Далее будем рассматривать команду Continue. Она позволяет в любой момент выпрыгнуть из текущей итерации(значению петли) и
перейти к следующей в текущем цикле. Это более просто, чем кажется:

```
For x.l = 1 To 10
  If x = 5
    Continue
  EndIf
    Debug x
Next
```

Здесь вы можете увидеть, что когда X равно 5, команда Continue прерывает пятую итерацию и продолжает сверху в начале шестого итерации, где X станет равно 6. Из-за этого прыжка можно заметить, в окне отладки число 5 не отображается.
Циклы могут быть использованы для различных целей в программировании. В первую очередь для сокращения утомительного кода и для перебора огромных объемов данных. Надеюсь, теперь вы поняли, как можно их использовать.

4

В этой главе я расскажу, как создать и использовать другие методы для хранения и управления данными, такие,
как структуры, массивы и связанные списки. Данные, такие как структуры необходимы для программирования игр и приложений,
поскольку они позволяют получить более легкий и быстрый доступ к нескольким значениям связанных и не связанных данных.
Как всегда, даются разъяснения и несколько примеров.

Структуры:

Ранее, во 2 главе, я представил вам встроенные типы данных, *Byte, Character, Word, Long, Quad, Float,*
Double и String. Используя в структуре слова с ключами(.s,.l,.b,.w,.c,.q), вы можете определить свой собственный
структурированный тип данных, а затем присвоить этот тип переменной(ым). Создание собственных структурированных
переменных удобно, особенно если вам понадобятся много общих имен переменных в рамках одной структуры.
Непонятно? Тогда давайте посмотрим на примере структуры, которая содержит несколько полей:

```
Structure PERSONALDETAILS
FirstName.s
LastName.s
Home.s
EndStructure

Me.PERSONALDETAILS
Me\FirstName = "Gary"
```

```
Me\LastName = "Willoughby"
Me\Home = "A House"

Debug "First Name: " + Me\FirstName
Debug "Last Name: " + Me\LastName
Debug "Home: " + Me\Home
```

Структура *'PERSONALDETAILS'* создается с использованием ключевого слова **Structure**. Далее идут компоненты структуры
которые определяются точно так же, как обычные переменные. Ключевое слово **EndStructure** используется для определения
конца структуры. После того как структура объявлена, она готова к использованию. Мы придали этой структуре тип так же, как мы
назначаем любой тип переменной, пример:

```
Me.PERSONALDETAILS
```

Здесь имя переменной *'Me'* и его тип *'PERSONALDETAILS'*. Чтобы присвоить значения отдельным
переменным (иногда называемые полями) в рамках новой *'Me'* структурированной переменной, мы используем символ '\'. Если
вы посмотрите на пример выше, то заметите, что ‘\’, символ также используется, чтобы восстановить данные из индивидуальных полей
также, как здесь:

```
Father.PERSONALDETAILS
Father\FirstName = "Peter"
Debug Father\FirstName
```

Здесь, в этом маленьком примере, мы создали новую структурированную переменную *"Father"*
с структурированным типом *"PERSONALDETAILS "*. Мы придаем значение *'Peter'* для *Father\FirstName*.
Затем мы вывели это значение в окне отладки. Конечно по этим примерам вы возможно не увидели пользы от структур, но они невероятно полезны.

Рассмотрение памяти

Размер памяти структурированной переменной зависит от полевой переменной, используемой в первоначальном
определении структуры. В структуре *"PERSONALDETAILS"* определены три переменные с
типом *String*, каждая из которых имеет размер 4 байта (см. Рис.3 ранее в главе 2 для размеров, типов String).
Таким образом, вновь заявленная переменная *'Me'* занимает 12 байт (3 x 4 байта) в памяти. Мы можем проверить это,
с помощью функции *'SizeOf ()'*.

```
Structure PERSONALDETAILS
  FirstName.s
```

```
  LastName.s
  Home.s
EndStructure
Debug SizeOf(PERSONALDETAILS)
```

Функция *'sizeof ()'* возвращает значение '12', которое показывает, сколько байт использует структура в памяти.

Функция 'SizeOf()'

Эта команда возвращает размер любой структуры или переменной, определяемой в байтах. Она не работает с массивами, связанными списками или интерфейсом. Эта команда имеет неоценимое значение для программирования в Windows, поскольку некоторые Win32 API функции требуют размера конкретной структуры или переменной в качестве параметра. Больше о Win32 API мы узнаем позже в главе 13.

Дополнение полей из другой структуры

Структуры можно дополнить полями из другой структуры с помощью параметра ***'Extends'***

```
Structure PERSONALDETAILS
  FirstName.s
  LastName.s
  Home.s
EndStructure


Structure FULLDETAILS Extends PERSONALDETAILS
  Address.s
  Country.s
  ZipCode.s
EndStructure

User.FULLDETAILS
User\FirstName = "John"
User\LastName = "Smith"
User\Home = "A House"
User\Address = "A Street"
User\Country = "UK"
User\ZipCode = "12345"

Debug "Users First Name: " + User\FirstName
Debug "Users Last Name: " + User\LastName
Debug "Users Home: " + User\Home
Debug "Users Address: " + User\Address
Debug "Users Country: " + User\Country
Debug "Users Zip Code: " + User\ZipCode
```

В этом примере структуру *'FULLDETAILS'* расширяет структура *'PERSONALDETAILS'* и вновь получаемая структура дополняется данными из структуры *'PERSONALDETAILS'* , причем так, что взятые данные появляются первыми в нашей новой структуре. Мы присвоили этой новосозданной структуре переменную *'User'*, ну и присвоили значения во всех ее сферах. Затем эти данные проверили в окне отладки.

Структуры Union (объединенные)

Структуры *Union* - способ сохранить память, вынуждая группы переменных совместно использовать в пределах структуры тот же самый адрес памяти. Возможно я залез немного вперед, но поверьте мне пришлось это сделать для законченности.
Вы можете просмотреть Главу 13 (Pointers), чтобы понять лучше, как это работает. Вот простой пример:

```
Structure UNIONSTRUCTURE
  StructureUnion
    One.l
    Two.l
    Three.l
  EndStructureUnion
EndStructure

Debug SizeOf(UNIONSTRUCTURE)

UnionVariable.UNIONSTRUCTURE
UnionVariable\One = 123

Debug UnionVariable\One

UnionVariable\Three = 456

Debug UnionVariable\One
```

После объявления *'UNIONSTRUCTURE'* мы использовали **StructureUnion** и **EndStructureUnion** для инкапсуляции переменных, которым мы хотим использовать одну выделенную область памяти.
Когда мы запускаем эту небольшую программу,первое что появляется в окне отладки это '4 '(байта), поскольку структура имеет размер одной переменной из-за того, что все переменные в такой структуре задействуют только одно место в памяти.
Далее в программе мы присвоиваем *UnionVariable* тип *UNIONSTRUCTURE* и назначаем значение '123' для *UnionVariable\One,* затем считываем его в окне отладки. После мы присвоиваем новое значение '456' для *UnionVariable\Three*, и снова считываем старую переменную *UnionVariable\One*. Но так как используется одна и та же область памяти для всех переменных в структуре то переменной *UnionVariable\One* присваивается новое значение '456' которое как мы помним, мы присвоили *'UnionVariable\Three'*.
Структуры могут содержать так называемые статические массивы, но мне нужно

объяснить массивы, прежде чем мы
может применять эти знания для структур. Массивы и статические массивы объясняются в полном объеме в следующем разделе.

Массивы:

В PureBasic, массивы могут вмещать определяемое пользователем кол-во переменных, одинакового типа данных. Каждая переменная
в массиве имеет свой индекс, который лежит в последовательном диапазоне целых чисел. Массивы могут также быть определены для структурированных
переменных. Этот раздел научит вас всему, что нужно знать о массивах в PureBasic.

DIM

Массивы создаются с помощью команды *DIM*:

```
Dim LongArray.l(2)
```

Позвольте мне объяснить эту строку кода более понятно. Во-первых мы используем команду *Dim* , чтобы сообщить компилятору, что
мы собираемся определить массив. Затем мы даем имя массиву. В данном случае, я назвал его *LongArray*. После названия, мы аналогично переменным,
присваиваем тип массиву с помощью суффикса *'. l'*. Итак, массив имеет тип *Long*. После того, как определен тип, нам надо определить, сколько индексов
будет содержать массив.Для определения индекса используются целые числа в круглых скобках. В приведенном выше примере мы использовали (2). Это означает,
что массив сможет вместить три переменных. Почему три, а не две? Потому, что отсчет в массивах всегда начинается с нуля, а цифра (2) показывает последний индекс
массива.После того как массив был создан, все его переменные будут иметь тип *Long*. Это простой массив. Чаще такой массив называют одномерным. Давайте посмотрим на наглядном примере в котором мы определим массив и присвоим значения всем его индексам:

```
Dim LongArray.l(2)
  LongArray(0) = 10
  LongArray(1) = 25
  LongArray(2) = 30
Debug LongArray(0) + LongArray(1)
Debug LongArray(1) * LongArray(2)
Debug LongArray(2) - LongArray(0)
```

После того как мы присвоили значения переменным массива, мы произвели с ними математические операции и вывели результат в окно отладки. Например, первым результатом будет сложение '10 + 25 ',поскольку мы сложили индексы '0 'и '1'. Вторым результатом умножение '25' * '30 ', третьим '30'-'10'. Индексы могут задаваться не только с помощью выражений, но и с помощью переменных:

```
   LastIndex.l = 2
   FirstIndex.l = 0
Dim StringArray.s(LastIndex)
```

```
  StringArray(FirstIndex) = "One is one and all alone"
  StringArray(FirstIndex + 1) = "Two, two, the lily-white boys"
  StringArray(FirstIndex + 2) = "Three, three, the rivals"
Debug StringArray(FirstIndex)
Debug StringArray(FirstIndex + 1)
Debug StringArray(FirstIndex + 2)
```

Здесь мы определили массив с тремя индексами, каждый из которых содержит строковую переменную(обратите внимание на суффикс **.S**). С помощью переменной LastIndex мы сумели задать три индекса. Затем мы использовали переменную FirstIndex чтобы заполнить массив тремя переменными. Далее попросту считали данные из массива в отладочное окно. См. таблицу ниже для большего понимания соответствия индексов и значений:

Индекс	Значение
0	One is one and all alone
1	Two, two, the lily-white boys
2	Three, three, the rivals

Поскольку массивы аккуратно отсортированы в индексы, это дает возможность для перебора их с помощью циклов, очень быстро. Вот пример массива с 1000 индексами. Сначала с помощью первого цикла мы заполняем массив, а вторым циклом считываем данные из массива в окно отладки.

```
Dim TestArray.l(999)
For x = 0 To 999
  TestArray(x) = x
Next x
For x = 0 To 999
  Debug TestArray(x)
Next x
```

Запустите код и посмотрите в окно отладки. Как вы можете видеть, с помощью циклов легко и быстро заполнить массив, а потом так же быстро считать из массива.

Многомерные массивы

Лучший способ описать многомерные массивы,это создать таблицу.Чтобы понять как размещаются данные, мы
просто укажем число столбцов и строк, которые имеет массив.(Но это чуть ниже). А сейчас посмотрим пример
где создадим массив,так называемый *"Animals"*,который состоит из трех индексов,каждый

из которых содержит
еще три индекса.

```
Dim Animals.s(2, 2)

  Animals(0, 0) = "Sheep"
  Animals(0, 1) = "4 Legs"
  Animals(0, 2) = "Baaa"

  Animals(1, 0) = "Cat"
  Animals(1, 1) = "4 Legs"
  Animals(1, 2) = "Meow"

  Animals(2, 0) = "Parrot"
  Animals(2, 1) = "2 Legs"
  Animals(2, 2) = "Screech"

Debug Animals(0, 0) + " has " + Animals(0, 1) + " And says " + Animals(0, 2)
Debug Animals(1, 0) + " has " + Animals(1, 1) + " And says " + Animals(1, 2)
Debug Animals(2, 0) + " has " + Animals(2, 1) + " And says " + Animals(2, 2)
```

И так давайте посмотрим в обещанную таблицу, в которой все наглядно видно и понятно:

ИНДЕКС	0	1	2
0	Sheep	4 Legs	Baaa
1	Cat	4 Legs	Meow
2	Parrot	2 Legs	Screech

Из таблицы видно, как располагаются данные в массиве.

Допустим мы хотим сменить данные в нулевой строке. Тогда так:

```
Animals(0, 0) = "Tripod"
Animals(0, 1) = "3 Legs"
Animals(0, 2) = "Oo-la"
```

В итоге будет следующее:

ИНДЕКС	0	1	2
0	Tripod	3 Legs	Oo-la
1	Cat	4 Legs	Meow
2		2	

	Parrot	Legs	Screech

Еще один способ объяснить многомерные массивы в том, что их принцип массив в массиве. Вспомним, что
в каждом массиве индекса содержится другой массив, и вы получите идею многомерных массивов.
В следующем примере показано , как определить один, два, три, четыре и пятьмерных массивов:

```
Dim Animals.s(5)
Dim Animals.s(5, 4)
Dim Animals.s(2, 5, 3)
Dim Animals.s(1, 5, 4, 5)
Dim Animals.s(2, 3, 6, 2, 3)
```

После двумерных массивов, трехмерные и т д, кажутся сложными, но если вы вспомните, массив
в массиве, то становится более менее понятно. Хотя максимальный ряд аспектов, который
может быть отнесен к элементам массива двести пятьдесят пять (255), использование массивов
более двух или более трех измерениях необычное в повседневной практике программирования.

Структурированный тип массивов

До сих пор мы видели, как применять различные массивы, используя только стандартные средства, но у нас есть
возможность применять массивы, используя структуру. Давайте рассмотрим простой пример использования одномерного массива:

```
Structure FISH
  Kind.s
  Weight.s
  Color.s
EndStructure

Dim FishInTank.FISH(2)
 FishInTank(0)\Kind = "Clown Fish"
 FishInTank(0)\Weight = "4 oz."
 FishInTank(0)\Color = "Red, White and Black"
 FishInTank(1)\Kind = "Box Fish"
 FishInTank(1)\Weight = "1 oz."
 FishInTank(1)\Color = "Yellow"
 FishInTank(2)\Kind = "Sea Horse"
 FishInTank(2)\Weight = "2 oz."
 FishInTank(2)\Color = "Green"


Debug FishInTank(0)\Kind+" "+FishInTank(0)\Weight+" "+FishInTank(0)\Color
Debug FishInTank(1)\Kind+" "+FishInTank(1)\Weight+" "+FishInTank(1)\Color
```

Debug FishInTank(2)\Kind+" "+FishInTank(2)\Weight+" "+FishInTank(2)\Color

После того как мы определили структуру *FISH,* мы определили массив, используя команду *DIM,* а суффикс *FISH* - тип массива.
Все так же, как мы использовали .S (string) в массиве Animals. Кроме того мы использовали '2' в качестве последнего индекса в этом массиве.
Для присвоения значений полей каждому индексу массива, надо просто объединить синтаксис задания массивов и структур:

FishInTank(0)\Kind = "Clown Fish"

Давайте попробуем разбить эту часть кода на куски для большего понимания. Во-первых, имя массива, в данном случае это
"FishInTank. Затем идет текущий индекс, содержащийся в скобках, в данном случае индекс '0 '. Далее мы используем
символ '\' для доступа к области, называемой "KIND" в пределах FISH структуры, которая была возложена на массив
FishInTank. Затем мы используем оператор '=', чтобы присвоить строковое значение. Для получения значения, которое мы только что назначили,
мы просто используем точно такой же синтаксис, как при назначении. В данном случае мы выводим значение в окно отладки:

Debug FishInTank(0)\Kind

Если нам необходимо получить и другие значения индексов массива, мы делаем это так:

Debug FishInTank(0)\Kind
Debug FishInTank(1)\Kind
Debug FishInTank(2)\Kind

Это был список полей Kind всех индексов массива FishInTank. Чтобы получить значение из других областей,
мы просто используем их имена:

Debug FishInTank(1)\Kind
Debug FishInTank(1)\Weight
Debug FishInTank(1)\Color

Здесь мы получаем все поля, одного из индексов массива '1 '. Чтобы было более понятно давайте опять создадим графическую таблицу

ИНДЕКС	Структура FISH
0	Kind: Clown Fish Weight: 4 oz. Colour: Red, White and Black
1	Kind: Box Fish Weight: 1 oz. Colour: Yellow
2	Kind: Sea Horse Weight: 2 oz. Colour: Green

Как и в случае с одномерными массивами, вы можете указать многомерные массивы с помощью структуры. Вы получите доступ к полям структуры в рамках каждого из индексов внутри многомерных массивов. Чтобы определить многомерный структурированный тип массива, делайте это в точности так же, только дополнив индексы.

```
Structure FISH
  Kind.s
  Weight.s
  Color.s
EndStructure
Dim FishInTank.FISH(2, 2)
.....
```

Я не буду набирать код массива, думаю и так все понятно, что следует вместо многоточия. С другой стороны мы создадим таблицу многомерного массива.

ИНДЕКС	0	1	2
0	Kind: Clown Fish Weight: 4 oz. Colour: Red, White and Black	Kind: Box Fish Weight: 1 oz. Colour: Yellow	Kind: Sea Horse Weight: 2 oz. Colour: Green
1	Kind: Parrot Fish Weight: 5 oz. Colour: Red	Kind: Angel Fish Weight: 4 oz. Colour: Orange	Kind: Shrimp Weight: 1 oz. Colour: Pink
2	Kind: Gold Fish Weight: 2 oz. Colour: Orange	Kind: Lion Fish Weight: 8 oz. Colour: Black and White	Kind: Shark Weight: 1 lb. Colour: Grey

Для получения значения из этого вида массива мы должны указать в скобках два индекса, то есть так:

```
Debug FishInTank(1, 1)\Kind
```

Если мы хотим изменить данные в массиве, то это можно сделать так:

```
FishInTank(1, 1)\Kind = “Devil Fish”
FishInTank(1, 1)\Weight = “6 oz.”
FishInTank(1, 1)\Color = “Dark Red”
```

Мы изменили все поля структура *FISH*, которые расположены в средней части массива под индексом '1, 1 '. Смотрим ниже:

ИНДЕКС	**0**	**1**	**2**
0	Kind: Clown Fish Weight: 4 oz. Colour: Red, White and Black	Kind: Box Fish Weight: 1 oz. Colour: Yellow	Kind: Sea Horse Weight: 2 oz. Colour: Green
1	Kind: Parrot Fish Weight: 5 oz. Colour: Red	**Kind: Devil Fish Weight: 6 oz. Colour: Dark Red**	Kind: Shrimp Weight: 1 oz. Colour: Pink
2	Kind: Gold Fish Weight: 2 oz. Colour: Orange	Kind: Lion Fish Weight: 8 oz. Colour: Black and White	Kind: Shark Weight: 1 lb. Colour: Grey

Вы, вероятно, используете только одномерные структурированные массивы в ваших программах на данный момент, но
знайте, что многомерные массивы структурированного типа дадут вам хорошее представление о более
расширенном коде.

Переопределение созданного массива

Стандартные массивы в PureBasic не статичны, то есть они могут быть переопределены двумя разными способами. Первый способ заключается в использовании команды *Dim* которая переопределяет массив, но в процессе уничтожает все предыдущие данные, имеющиеся в нем. Второй способ заключается в использовании команды *ReDim*, которая переопределяет массив, но сохраняет предыдущие данные нетронутыми. Давайте посмотрим на переопределение массива с помощью команды *Dim:*

```
Dim Dogs.s(2)

  Dogs(0) = "Jack Russell"
  Dogs(1) = "Alaskan Husky"
  Dogs(2) = "Border Collie"

Debug Dogs(0)
Debug Dogs(1)
Debug Dogs(2)
```

```
Dim Dogs.s(2)

Debug Dogs(0)
Debug Dogs(1)
Debug Dogs(2)
```

Здесь после создания и заполнения массива, мы переопределив его попросту стираем все данные в нем. Это может быть очень полезно. К примеру надо очистить лишнюю используемую память после массива.Но знайте, что переопределяя массив, надо применять тот же тип, иначе будет ошибка. А теперь давайте воспользуемся командой *ReDim* и переопределим массив, дополнив его, причем старые данные сохранятся:

```
Dim Dogs.s(2)

  Dogs(0) = "Jack Russell"
  Dogs(1) = "Alaskan Husky"
  Dogs(2) = "Border Collie"

For x.l = 0 To 2
  Debug Dogs(x)
Next x

Debug ""

ReDim Dogs.s(4)

  Dogs(3) = "Yorkshire Terrier"
  Dogs(4) = "Greyhound"

For x.l = 0 To 4
  Debug Dogs(x)
Next x
```

В примере выше мы дополнили массив двумя ячейками, в результате их стало пять.

Правила пользования массивов

Хотя массивы являются очень гибкими, существует несколько правил, которые должны учитываться при их использовании.
Правила должны быть соблюдены при использовании массивов в своих программах.
1). Если массив повторно определяется с помощью команды *Dim*, массив теряет свои предыдущие данные.
2). Если массив повторно определяется с помощью команды *ReDim*, то предыдущие данные сохраняются.
3). Массивы могут быть сделаны только из одного типа переменной (структурированного или стандартного типа переменной).
4). Массивы могут быть глобальными(**Global**), защищенной(**Protected**), статические(**Static**) и общие(**Shared**) (. См. главу 6).

5). Размер массива ограничен только оперативной памятью текущей машины.
6). Многомерные массивы могут иметь размер 255.
7). Массивы можно динамически определить с помощью переменной или выражения указывая размер.
8). При определении размера , вы определяете последний номер, а отсчет начинается с нуля.
9). Индексы могут быть различных размеров в многомерных массивах.

Статические массивы в структурах

Статические массивы в структурах немного отличаются от обычных массивов, которые были
описаны ранее. Статические массивы в самой своей природе являются статичными и поэтому не могут быть изменены,
после того, как они определены. Эти типы массивов существуют только в рамках структуры.

Статические массивы также имеют различный набор правил, которые учитывается при их использовании:

1). Как только статический массив определен, его внутренняя структура не может быть изменена.
2). Статические массивы (как структуры) не могут быть переопределены.
3). Они могут быть сделаны только из одного типа переменной (структурированного или стандартного типа переменной).
4). Размер массива ограничен только установленной оперативной памятью текущей машины.
5). У статических массивов может только быть одна размерность.
6). Они могут быть динамически определены, используя переменную или выражение, в которой задается размер
7). При определении размера , вы определите кол-во индексов и оно показывает реальный размер массива.
8). Статические массивы могут быть доступны только через структуру переменных, в которых они определены.

После того как я дал вам основные правила, позвольте мне привести вам пример того, как они используются:

```
Structure FAMILY
  Father.s
  Mother.s
  Children.s[2]
  Surname.s
EndStructure

  Family.FAMILY
  Family\Father = "Peter"
  Family\Mother = "Sarah"
  Family\Children[0] = "John"
  Family\Children[1] = "Jane"
  Family\Surname = "Smith"
```

```
Debug "Family Members:"
Debug Family\Father + " " + Family\Surname
Debug Family\Mother + " " + Family\Surname
Debug Family\Children[0] + " " + Family\Surname
Debug Family\Children[1] + " " + Family\Surname
```

В этом примере, структура *"FAMILY"* имеет поле с названием *"Children"*, которое представляет собой статический строковой массив.
Когда мы определили этот массив, мы использовали цифру '2 '. Это означает, что в массиве два индекса.
Это не похоже на стандартные массивы, в котором вы определяете последний индекс.
В нашем новом статическом массиве два показателя, '0 'и '1'.
Далее в примере я назначил значения всех полей в структурированную переменную *FAMILY* , в том числе и два индекса в *Children* статического массива.
Вы заметите, что статические массивы имеют немного другой синтаксис для назначения и используют квадратные скобки:

```
Family\Children[0] = "John"
```

Вы также заметили, что вам не нужно использовать команду *Dim*, когда вы определяете статический массив.
Вы просто добавляете квадратные скобки . В квадратных скобках вы определяете нужный размер. В структуре *FAMILY* выше, я
использовал строковой тип для статического массива, но вы можете использовать любой тип и конечно же любую структуру.
Давайте рассмотрим еще один простой пример:

```
Structure EMPLOYEES
  EmployeeName.s
  EmployeeClockNumber.l
  EmployeeAddress.s
  EmployeeContactNumbers.l[2]
EndStructure

Dim Company.EMPLOYEES(9)

  Company(0)\EmployeeName = "Bruce Dickinson"
  Company(0)\EmployeeClockNumber = 666
  Company(0)\EmployeeAddress = "22 Acacia Avenue"
  Company(0)\EmployeeContactNumbers[0] = 0776032666
  Company(0)\EmployeeContactNumbers[1] = 0205467746
  Company(1)\EmployeeName = "Adrian Smith"
  Company(1)\EmployeeClockNumber = 1158
```

Здесь я создал структуру под названием *EMPLOYEES*, чтобы описать небольшую компанию сотрудников.
Затем создал стандартный массив, который содержит десять таких записей. Внутри структуры *EMPLOYEES* я
использовал статический массив с типом *Long* для хранения двух номеров контактного телефона.
Ну и начал определять индивидуальные сведения о сотрудниках, начинающихся с

Company(0) \ ... , затем Company(1) \ ..., и т.д.
Я не завершил этот пример, но я уверен, что вы поняли идею того,что я хотел показать.

Связанные списки

Связанные списки похожи на массивы тем, что они могут сослаться на множество данных с использованием одного имени.
Вместе с тем, они отличаются от массивов тем, что они не используют индекс чтобы назначать и получать данные.
Эти списки похожи на книгу, в которой вы можете пролистать данные от начала до конца или просто перейти к нужной странице
внутри, ну и конечно считать данные оттуда. Связные списки полностью динамичны. Это означает, что они могут расти
или уменьшаться в зависимости от того, сколько данных вам нужно хранить в них. При увеличении размеров в связанных
списках не будет вреда. Кроме того, если вам надо добавлять или изменять какие-либо другие данные, хранящиеся в них,
вы можете спокойно это делать, не боясь за остальные данные, причем в любом месте связанного списка.

Связные списки являются отличным способом хранения и управления данными неопределенной длины и могут быть отсортированы
несколькими способами. Существует также встроенные библиотеки, которые предоставляют функции для выполнения команд
добавления, удаления и замены элементов. Так же внутри библиотеки имеются две функции, которые используются исключительно
для сортировки связанных списков, но об этом я упомяну позже.Общий обзор встроенных команд, будет дан позднее в главе 7.

Функция NewList

Связанные списки в PureBasic создаются с помощью функции *NewList*, как в следующем примере:

```
NewList Fruit.s()
```

Определение связанного списка очень похоже на определение массива, используя команду *Dim*. В начале мы используем
команду *NewList*, чтобы сообщить компилятору, что мы собираемся определить связанный список. Далее, мы определяем имя
нашего списка, в данном случае мы называли это *Fruit*. После названия мы определяем его тип, в нашем случае *String*
то есть строковой. Скобки используются для определения списка. В них не надо ничего заносить, поскольку списки
динамичны и будут расти по мере добавления элементов. Давайте посмотрим, как мы добавим новый элемент в список:

```
NewList Fruit.s()
  AddElement(Fruit())
  Fruit() = "Banana"
  AddElement(Fruit())
  Fruit() = "Apple"
```

Поскольку связанные списки не имеют индексов, использование их на первый взгляд кажется странным, потому что неизвестно
где какой элемент расположен. В приведенном выше примере я добавил два новых элемента в список *Fruit ()* .
Для этого я использовал функцию *AddElement ()*. Когда мы добавляем новый элемент, с помощью этой функции,
она не только автоматически определяет новый элемент, но также делает связанную точку в списке имен,создавая пустой элемент.
А дальше мы просто используем имя списка, чтобы присвоить этому элементу часть нужных нам данных в список.
Вместе с именем списка обязательно используем круглые скобки:

```
Fruit() = "Banana"
```

Когда мы добавляем еще один элемент помощью функции *AddElement ()*, то происходит точно такой же процесс:

```
Fruit() = "Apple"
```

Можно подумать, что это неправильно, потому что мы присваиваем текст *Apple* тому же имени, которому присвоили
текст *Banana*. Но если вспомнить, что мы каждый раз добавляем новый элемент связанного списка с помощью функции *AddElement (),*
при всем этом сохраняя старые,то думаю все становится понятным. Мы всегда можем проверить, сколько элементов в нашем списке, используя
функцию *CountList ()* например:

```
Debug CountList(Fruit())
```

Если вы выполнили код выше, и дописали туда *Debug CountList(Fruit())* то количество элементов, в списке *Fruit()* будет показано в окне отладки и равно 2.

Давайте добавим побольше элементов в этот список, а затем с помощью цикла выведем их в окно отладки:

```
NewList Fruit.s()
  AddElement(Fruit())
  Fruit() = "Banana"
  AddElement(Fruit())
  Fruit() = "Apple"
  AddElement(Fruit())
  Fruit() = "Pear"
  AddElement(Fruit())
  Fruit() = "Orange"
ForEach Fruit()
   Debug Fruit()
Next
```

В этом примере, мы создали новый связанный список, называемый *Fruit()* и в нем мы создали четыре элемента
и назначили им индивидуальные значения. Затем используя цикл *ForEach* мы вывели эти значения в окно отладки.

Команда ForEach используется для определения цикла, который используется только для связанных списков.

Ниже дается краткий обзор команд для связанных списков.
Более продвинутые команды могут быть найдены в PureBasic Helpfile.

Функция	Описание
AddElement(List())	Добавляет элемент в список.
ClearList(List())	Очищает список всех элементов.
CountList(List())	Подсчитывает кол-во элементов внутри списка.
DeleteElement(List())	Удаление текущего элемента в списке
FirstElement(List())	Перейти к первому элементу в списке.
InsertElement(List())	Вставляет один элемент в список перед текущим элементом, или в начало списка, если список пуст.
LastElement(List())	Переход к последнему элементу в списке.
ListIndex(List())	Возвращает позицию текущего элемента в списке. (позиция начинается с '0 ').
NextElement(List())	Переход к следующему элементу в списке.
PreviousElement(List())	Переход к предыдущему элементу в списке.
ResetList(List())	Сбросить позиции списка в '0 'и сделать первый элемент текущим.
SelectElement(List(), Position)	Сделать текущим элементом тот, что указан в параметре 'Position'.

Структурированные Связные списки

Теперь, когда я объяснил стандартные связанные списки, давайте перейдем к структурированным. Они аналогичны
структурированным массивам в том, что данные определяются с помощью структуры вместо встроенных переменных.
Давайте возьмем пример с структурированными массивами, но переделаем код под структурированные связанные списки.

```
Structure FISH
  Kind.s
  Weight.s
  Color.s
EndStructure


NewList FishInTank.FISH()
AddElement(FishInTank())
  FishInTank()\Kind = "Clown Fish"
  FishInTank()\Weight = "4 oz."
```

```
  FishInTank()\Color = "Red, White and Black"
AddElement(FishInTank())
  FishInTank()\Kind = "Box Fish"
  FishInTank()\Weight = "1 oz."
  FishInTank()\Color = "Yellow"
AddElement(FishInTank())
  FishInTank()\Kind = "Sea Horse"
  FishInTank()\Weight = "2 oz."
  FishInTank()\Color = "Green"
ForEach FishInTank()
  Debug FishInTank()\Kind+" "+FishInTank()\Weight+" "+FishInTank()\Color
Next
```

Вы можете видеть из этого примера, что после создания списка, он очень похож на структурированный
массив. Основное различие здесь в том, что индексы массива не используются.
Помните, что при использовании *AddElement (FishInTank ())* команда создает новый элемент
с использованием структуры. Обратите внимание, нам не надо каждый раз писать *AddElement (FishInTank ()).*
Вот в чем прелесть структуры! Она расширяет наш код, притом в очень удобном виде.

Связанные списки за или против?

Связные списки прекрасно подходят для хранения данных, когда вы не знаете, сколько их будет. Например
ранее я написал программу для отслеживания бытовых расходов, и использовал структурированные связанные списки.
Подробная информация об этих расходах. Использование связанного списка, было более удобным, чем массив, поскольку
проще простого было добавлять, удалять и сортировать данные.
Во время написания этой программы я думал, что я должен сделать эту программу гибкой, чтобы работать с новыми расходами
когда они встречаются, и чтобы иметь возможность удалить старые и т.д.
Это очень хорошо обрабатывается в связанных списках. Когда мне нужно добавить запись я использую функцию *AddElement (),*
когда мне нужно удалить запись я использую функцию *DeleteElement ()* . После добавления и
удаления в списке, я передаю все эти данные в приятный графический интерфейс пользователя (GUI)
чтобы увидеть и взаимодействовать. Более подробно о GUI в главе 9.
Связные списки являются более гибкими, чем массивы в том, что они могут расти и уменьшаться в размерах более быстро и просто.
Массивы же всегда будет использовать меньше памяти для хранения того же объема информации, чем связанные списки. Это происходит потому,
что массивы используют непрерывную область памяти, используя стандартный объем оперативной памяти для каждого
индекса. Связные списки отличаются таким образом, чтобы каждый элемент использует примерно в три раза больше
RAM для определенного типа. Это происходит потому,что связанные списки не находятся в непрерывном куске памяти.
Имейте в виду при работе с огромными массивами данных, так как ваши требования к

памяти могут быть тройными,
если вы будете использовать связанные списки.

Сортировки массивов и связанных списков

Массивы и связанные списки прекрасно подходят для хранения всех видов данных, и эти данные проходящие через структуры могут быть легко
получены. Хотя иногда вам может потребоваться реорганизовать данные содержащиеся в массиве или связанном списке в алфавитном порядке или численно.
Есть несколько примеров (Helpfile:Reference Manual->General Libraries->Sort) для сортировки массивов и связанных списков.

Сортировка стандартных массивов

Сортировка стандартных массивов чрезвычайно проста. Прежде всего, вам необходимо иметь массив с заполненными значениями.
Затем использовать функцию SortArray () для сортировки. Вот примерный синтаксис:

SortArray(Array(), Options [, Start, End])

Первым делом мы задаем команду *SortArray*, , что массив будет отсортирован. Обратите внимание за фигурными скобками после имени массива стоят еще одни скобки.
Они необходимы для правильной передачи массива в качестве параметра. Второй параметр является опциональным, чтобы указать, как массив будет отсортирован.
Вот опциональные значения для второго параметра:

'0 ': Сортировка массива в порядке возрастания быть чувствительным к регистру.
'1 ': Сортировка массива в порядке убывания быть чувствительным к регистру.
'2 ': Сортировка массива в порядке возрастания, не быть чувствительным к регистру.(' A 'такой же, как ' a ').
'3 ': Сортировка массива в порядке убывания, не быть чувствительным к регистру.(' A 'такой же, как ' a ').

Квадратные скобки, с последними двумя параметрами показывают, что эти не являются обязательными. Последние два параметра используются для указания позиции внутри массива.Используя выше информацию, мы можем сортировать массив в порядке возрастания и быть чувствительным к регистру, с использованием
команды *SortArray(Fruit(), 0)*:

```
 Dim Fruit.s(3)
   Fruit(0) = "Banana"
   Fruit(1) = "Apple"
   Fruit(2) = "Pear"
   Fruit(3) = "Orange"
SortArray(Fruit(), 0)

For x.l = 0 To 3
   Debug Fruit(x)
Next x
```

Сортировка структурированных массивов

Это будет посложнее, поскольку она использует немного более сложную команду сортировки SortStructuredArray ().
Вот примерный синтаксис:

SortStructuredArray(Array(), Options, Offset, Type [, Start, End])

Первым параметром является имя массива со скобками. Вторым способ сортировки, это точно так же, как в *SortArray ()*. Третим параметром является смещение (позиция в теле структуры) то есть поле, которое вы хотели бы сортировать. Можно задавать с помощью функции *OffsetOf ()*
Функция *OffsetOf ()* возвращает количество байт какой-либо переменной от начала структуры. Четвертый параметр определяет, какой тип переменной находится на месте смещения. Вы можете использовать встроенные константы для указанных параметров, для описания типа переменной:

‘#PB_Sort_Byte’
‘#PB_Sort_Character’
‘#PB_Sort_Word’
‘#PB_Sort_Long’
‘#PB_Sort_Quad’
‘#PB_Sort_Float’
‘#PB_Sort_Double’
‘#PB_Sort_String’

Последние два параметра в скобках, точно так же как и *SortArray ()*. А теперь пример:

```
Structure WEAPON
  Name.s
  Range.l
EndStructure

Dim Weapons.WEAPON(2)
  Weapons(0)\Name = "Phased Plasma Rifle"
  Weapons(0)\Range = 40
  Weapons(1)\Name = "SVD-Dragunov Sniper Rifle"
  Weapons(1)\Range = 3800
  Weapons(2)\Name = "HK-MP5 Sub-Machine Gun"
  Weapons(2)\Range = 300

SortStructuredArray(Weapons(), 0, OffsetOf(WEAPON\Range), #PB_Sort_Long)

For x.l = 0 To 2
  Debug Weapons(x)\Name + " : " + Str(Weapons(x)\Range)
Next x
```

В этом примере я выбрал поле *"Range"*, чтобы упорядочить структурированный массив.В команде *SortStructuredArray*
я определил смещение с помощью функции *OffsetOf (WEAPON\Range)* и указал тип переменной поля с помощью константы *# PB_Sort_Long*.

Сортировка стандартного связанного списка

Сортировка стандартных связанных списков предельно проста. Прежде всего, вам потребуется связанный список предварительно заполненный значениями. Затем использовать функцию SortList () для сортировки. Вот примерный синтаксис:

SortList(ListName(), Options [, Start, End])

Вначале идет название функции, и оно говорит о том что связанный список будет отсортирован. Далее в скобках задаются параметры:

1) Имя связанного списка со скобками
2) Опция сортировки (так же как у массивов выше)
3) Необязательные параметры начала и конца сортировки внутри списка

Как вы успели заметить все достаточно схоже с массивами, разница лишь в имени функции!

Использовав приведенную выше информацию, мы отсортируем список в возрастающем порядке с опцией: чувствительность к регистру:

```
NewList Fruit.s()
  AddElement(Fruit())
  Fruit() = "Banana"
  AddElement(Fruit())
  Fruit() = "Apple"
  AddElement(Fruit())
  Fruit() = "Orange"


SortList(Fruit(), 0)


ForEach Fruit()
  Debug Fruit()
Next
```

Сортировка структурированных связанных списков

Сортировка структурированых связанных списков как и массивов немного сложнее, поскольку она использует немного более сложную функцию *SortStructuredList ()* для сортировки. Вот пример синтаксиса этой команды:

SortStructuredList(List(), Options, Offset, Type [, Start, End])

Первой идет имя функции *SortStructuredList ()* Далее в скобках задаются параметры:

1)Имя списка
2)Опция сортировки(так же как у массивов выше)
3)Смещение в структуре, то есть нужное поле(задается с помощью функции *OffsetOf ()*)
4) Тип переменной, которая хранится в нужном поле структуры (можно задавать константой, см. выше список констант)
5) Необязательные параметры начала и конца сортировки внутри списка

Ну и конечно пример:

```
Structure GIRL
  Name.s
  Weight.s
EndStructure


NewList Girls.GIRL()
  AddElement(Girls())
    Girls()\Name = "Mia"
    Girls()\Weight = "8.5 Stone"
  AddElement(Girls())
    Girls()\Name = "Big Rosie"
    Girls()\Weight = "19 stone"
 AddElement(Girls())
    Girls()\Name = "Sara"
    Girls()\Weight = "10 Stone"
SortStructuredList(Girls(), 0, OffsetOf(GIRL\Name), #PB_Sort_String)


ForEach Girls()
  Debug Girls()\Name + " : " + Girls()\Weight
Next
```

В этом примере я выбрал поле *"Name"* для сортировки структурированного связанного списка. Определить смещение мне помогла функция *OffsetOf (Girl \ Name),*
а тип переменной я задал с помощью константы *# PB_Sort_String.*

Как маленькое заключение по массивам и спискам

Выбор массивов, связанных списков, структур должен зависеть от задачи поставленной программистом. Хотя зачастую на практике, каждый программист любит работать с более привычными для него методами хранения данных, на мой взгляд надо все же уметь выбирать для каждой задачи более уместный метод.
Сортировка массивов и связанных списков, созданных с помощью структур или без них, требует правильности синтаксиса, нужных
функций и главное вашей практики. Попрактикуйтесь, и вы быстро овладеете этим в принципе нехитрым, но мощным оружием в программировании.

5

Процедуры и подпрограммы

В этой главе я расскажу о процедурах и подпрограммах. Процедуры являются неотъемлемой частью любого языка программирования и предоставляют возможность для

аккуратного структурированного кода и создания благоприятных повторов использования кода. PureBasic считается структурированным и процедурным языком программирования, обеспечивающим средства для создания четкой структуры программы. Также я упомяну о подпрограммах. Однако они являются далеко не основным элементом во многих языках программирования. Поскольку они все таки играют определенную роль в PureBasic, они упоминаются здесь для полноты картины.

Для чего нужны процедуры или подпрограммы?

В принципе, я думаю, вы можете не использовать процедуры или подпрограммы. Вы можете просто написать очень большой кусок кода, другой вопрос как вы будете потом в нем разбираться в случае ошибок в программе, которые уверяю вас будут по мере написания программы. Процедуры и подпрограммы обеспечивают способы вызова отдельных кусков кода в любое время из вашего основного кода. Например можно вызвать процедуру: проигрывание звука или обновление дисплея. Использование процедур и подпрограмм являются хорошим способом понятного и ухоженного кода, а так же предоставление ему четкой структуры. Хорошо структурированный код всегда лучше для чтения в дальнейшем, к примеру для обновлений программы, или если вы работаете в команде, тогда многим людям будет проще понять его.

Подпрограммы

Подпрограммы не очень часто используются в PureBasic, а некоторые считают их плохим стилем, но иногда они могут быть полезны, если вы хотите быстро вызвать некоторый код. Подпрограммы обеспечивают возможность прыжка в другой кусок кода ниже в исходном коде, а затем возврат к точке после прыжка. Для того, чтобы перейти к отдельному куску кода с помощью этого метода, нужно указать метку, на которую необходимо сделать переход. Вы можете использовать любое имя для метки, но вы должны придерживаться тех же принципов именования, изложенных для переменных, за исключением того, что вместо использования суффиксов для определения метки, используется двоеточие, вот так:

```
Label:
```

Эта подпрограмма может быть вызвана из любой точки в вашем коде, используя следующую команду:

```
Gosub Label
```

Обратите внимание, что после команды Gosub идет название метки подпрограммы на которую следует прыгнуть. После того, как будет отработан код подпрограммы, вернуться назад в основной код программы поможет команда Return:

```
Return
```

Ну и конечно пример использования:

```
a.l = 10
Debug a
Gosub Calculation
Debug a
End
```

```
Calculation:
  a = a * 5 + 5
Return
```

В этом примере, мы присваиваем переменной величину 10 и выводим в отладочное окно. Далее с помощью команды Gosub, мы заставляем программу сделать прыжок на подпрограммную метку *Calculation:* . В подпрограмме производятся вычисления с переменной **a**, в результате которых в переменной окажется число 55. Командой же RETURN делается возврат из подпрограммы в основной код. И программа продолжит выполнение кода сразу после вызова *Gosub Calculation*. То есть с инструкции *Debug a*. Которая выведет результат 55 в отладочное окно. Следующая инструкция *END* завершит выполнение программы.

Примечание относительно позиции подпрограмм в коде

Хотелось бы отметить одну важную вещь: вызовы подпрограмм могут находится где вам необходимо, но само тело подпрограммы должно находится после команды END, иначе ваша программа может работать неправильно.

Прыжки из подпрограмм

Хоть подпрограммы считаются плохой практикой кодирования, но раз уж начали описывать их, то я опишу их в полном объеме. Каждая подпрограмма должна содержать команду Return, для перехода обратно в основной код, но в некоторых случаях, при каких нибудь там условиях, нужно выскочить из подпрограммы досрочно. Это достигается с помощью команд *FakeReturn* и *GOTO*. Вот пример:

```
a.l = 10
Debug a
Gosub Calculation
Debug a
End


RunAgain:
a.l = 20
Debug a
Gosub Calculation
Debug a
End


Calculation:
If a = 10
FakeReturn
Goto RunAgain
Else
a = a * 5 + 5
```

```
EndIf
Return
```

Как вы можете видеть из этого небольшого примера, скачки из подпрограмм в зависимости от условий, делают код уродливым. Этот пример не труден для понимания. Переменной a присваивается значение 10. Далее идет вызов подпрограммы *Calculation*. В ней проверяется условие равенства *"a=10"*, и поскольку условие выполняется, выход из подпрограммы осуществляется досрочно с помощью *FakeReturn* и *GOTO RunAgain*, в результате программа начнет выполнение кода с метки *RunAgain*. В ней переменной присваивается значение 20, выводится в окно отладки и производится повторный вызов подпрограммы *Calculation* Как вы наверно поняли теперь условие равно False и программа делает вычисление с переменной. В результате a=105 и с помощью Return производится возврат из подпрограммы. Далее отображение значения в окне отладки и завершение программы. Надеюсь, что эти примеры дали вам понимание применения подпрограмм, но так же я надеюсь, что вы не будете злоупотреблять этой практикой. Вы можете улучшить структуру кода, используя процедуры.

Основы процедур

Структурное программирование является термином, который ясно определяет PureBasic, который может быть описан как 'Процедурный' и 'Структурный' язык программирования. Его архитектура просто насыщена процедурами. Процедуры (иногда вызываемые 'Функции' на других языках) являются самой важной частью функциональных возможностей, которые обеспечивает PureBasic. Процедуры по сути только держатель части кода, который можно выполнить в любой момент. Они чем то похожи на подпрограммы. Их можно вызывать сколько угодно раз так же, как и подпрограммы, но в отличие от них процедурам можно задавать множество стартовых параметров, а так же возвратить значение любого типа. Давайте начнем с простого примера процедуры:

```
Procedure NurseryRhyme()
  Debug "Mary had a little lamb, its fleece was white as snow."
  Debug "And everywhere that Mary went, that lamb was sure to go."
EndProcedure


NurseryRhyme()
```

Для того чтобы использовать процедуру в коде, вы должны ее сначала определить. Здесь я определил процедуру под названием *NurseryRhyme()*, используя ключевое слово **Procedure**. Конец процедуры определяется с помощью ключевого слова **EndProcedure**. Мы можем вызвать кусок кода, содержащийся в рамках этой процедуры в любое время, просто используя ее имя, например:

```
NurseryRhyme()
```

Прежде чем мы продолжим, я должен отметить несколько моментов в этом примере, которые требуют дальнейшего разъяснения. Во-первых, вы наверно заметили, что есть скобки после имени процедуры. Они нужны для использования каких-либо параметров, которые должны быть переданы процедуре. Даже если вы не используете никаких параметров, как в этом примере, все равно необходимо использовать скобки. Кроме того,

при вызове процедуры, так же используются скобки как часть вызова процедуры с параметрами или без. Кроме того хотелось бы отметить, что имя процедуры, например, NurseryRhyme() не должно содержать пробелов и необходимо придерживаться тех же принципов именования, изложенных для переменных. Если вы запустите приведенный выше пример, вы увидите в окне вывода отладки детский стишок, который содержится в теле вызванной процедуры NurseryRhyme(). Вы можете сколько угодно выполнять код этой процедуры, для этого достаточно просто вызывать ее снова и снова, используя ее имя.

```
Procedure NurseryRhyme()
  Debug "Mary had a little lamb, its fleece was white as snow."
  Debug "And everywhere that Mary went, that lamb was sure to go."
EndProcedure


For x.l = 1 To 5
  NurseryRhyme()
Next x
```

Процедуры можно вызвать из любого места программы, как и в приведенном выше примере, где я вызвал несколько раз процедуру NurseryRhyme() из цикла. Процедура может быть вызвана так же из других процедур:

```
Procedure NurseryRhyme2()
  Debug "And everywhere that Mary went, that lamb was sure to go."
EndProcedure

Procedure NurseryRhyme1()
  Debug "Mary had a little lamb, its fleece was white as snow."
  NurseryRhyme2()
EndProcedure

NurseryRhyme1()
```

В приведенном выше примере видно, как сначала вызывается процедура **NurseryRhyme1(),** а из нее уже идет вызов процедуры **NurseryRhyme2()**

Примечание относительно позиции процедур в коде

Приведенный выше пример четко показывает позиции процедур. Вы наверно заметили, что первой в коде объявлена процедура *NurseryRhyme2()* а уже потом процедура *NurseryRhyme1()*. Это сделано намеренно и было необходимо в силу того, что вы всегда должны объявлять процедуру перед ее вызовом. PureBasic компилятор считывает исходный код сверху вниз, и по этой причине компиляция вызовет ошибку, если вызов процедуры будет до ее объявления. Этот простой пример показывает также, почему почти всегда процедуры определяются в верхней части исходного кода. Как и в обычной жизни, есть исключения и из этого правила. Для того чтобы иметь возможность вызывать процедуру прежде ее объявления, надо использовать ключевое слово *Declare*. На самом деле Declare не объявляет процедуры, оно просто позволяют компилятору знать, какие процедуры будут вызываться. А теперь пример:

```
Declare NurseryRhyme1()
Declare NurseryRhyme2()


NurseryRhyme1()


Procedure NurseryRhyme1()
  Debug "Mary had a little lamb, its fleece was white as snow."
  NurseryRhyme2()
EndProcedure

Procedure NurseryRhyme2()
  Debug "And everywhere that Mary went, that lamb was sure to go."
EndProcedure
```

При использовании *Declare* определяется только первая строка процедуры, примерно так же, как ключевое слово Procedure. Далее вы вызываете процедуру в любом месте, но ниже *Declare*. Ну и не забываем реально объявлять процедуру в каком нужно месте, поскольку *Declare* как бы симулирует объявление процедуры.

Программная область

Используя процедуры, также очень важно понять различные области доступности переменных, массивов, или связанных списков в программах. Позвольте мне объяснить это на простом примере, используя переменную:

```
a.l = 10


Procedure DisplayValue()
  Debug a
EndProcedure


DisplayValue()
```

Здесь я определил переменную с типом *Long*, и присвоил ей значение ‘10’. Так же я определил процедуру ‘*DisplayValue()*’ и в этой процедуре всего одна строка программы: отобразить в окне отладки число присвоенное переменной. Последняя строка этого примера вызывает процедуру. Теперь, если вы выполните этот пример, то наверно ожидаете, что в отладочном окне должно быть отображено 10, но вместо этого вы видите 0. Позвольте мне все же объяснить в чем дело. Переменные в процедурах являются локальными, то есть доступные только в теле процедуры. А это значит, что переменная **a** которой присвоено значение 10 и переменная **a** находящаяся в процедуре это две разные переменные. Для того же чтобы переменная **a** стала общей или единой для основного кода и процедуры ее необходимо сделать глобальной. То есть вот так:

```
Global a.l = 10
```

```
Procedure DisplayValue()
  Debug a
EndProcedure


DisplayValue()
```

Сравните эти два примера и вы поймете, что переменная не объявленная глобальной в основном коде, будет не видна в процедуре, так же и переменная, которой присвоено какое либо значение в процедуре, но не являющаяся глобальной, не видна в основном коде:

```
Procedure DisplayValue()
  a.l = 10
EndProcedure


DisplayValue()

Debug a
```

Как вы поняли из этого примера, в окне отладки будет ноль. А теперь с командой *Global*

```
Procedure DisplayValue()
   Global a.l = 10
EndProcedure

DisplayValue()

Debug a
```

Теперь в окне отладки будет 10.

Читая эти последние несколько примеров вы можете подумать, почему бы не сделать все переменные глобальными? Когда программы достигают больших масштабов, код может стать запутанным, если все переменные являются глобальными. Использование различных областей в рамках вашей программы также позволяет использовать временные имена переменных в рамках процедуры для расчетов или циклов, и вы можете быть уверены, что они не повлияют на любые переменные извне. Некоторые программисты стремятся использовать как можно меньше глобальных переменных, поскольку это делает отладку программы не такой сложной. При использовании массивов и связанных списков с процедурами, они тоже могут иметь разные диапазоны в рамках вашей программы так же, как переменные. До PureBasic V4, все массивы и связанные списки являлись глобальными по умолчанию. С приходом PureBasic V4, массивы и связанные списки могут быть локальными и глобальными, то есть такие же как и переменные. Далее я опишу все более расширенно, с большим количеством примеров, чтобы продемонстрировать использование переменных, массивов и связанных списков в различных рамках программы.

Команда GLOBAL

Глобальные переменные

Я уже выше дал пример команды *Global* , но здесь повторюсь, чтобы сделать обзор полным .

```
Global a.l = 10

Procedure DisplayValue()
  Debug a
EndProcedure

DisplayValue()
```

Команда *Global,* используемая для переменной, делает ее общей, или правильней сказать глобальной для всего кода, включая процедуры. Синтаксис команды несложен, как вы уже видели выше.

Глобальные массивы

```
Global Dim Numbers.l(1)


Procedure ChangeValues()
  Numbers(0) = 3
  Numbers(1) = 4
EndProcedure


ChangeValues()
Debug Numbers(0)
Debug Numbers(1)
```

В этом примере, как и с глобальными переменными, я использовал команду *Global* перед массивом. Без команды *Global* процедура не сможет видеть или использовать его.

Глобальная Связные списки

```
Global NewList Numbers.l()
AddElement(Numbers())
Numbers() = 1

Procedure ChangeValue()
  SelectElement(Numbers(), 0)
  Numbers() = 100
EndProcedure


ChangeValue()
SelectElement(Numbers(), 0)
Debug Numbers()
```

В этом примере подобно массивам, я использовал команду *Global* перед связанным списком. Без команды *Global* процедура не сможет видеть или использовать его.

Команда Protected

Protected(местные) переменные

Команда *Protected* стоящая перед переменной внутри процедуры, делает ее местной только для этой процедуры, даже если она была ранее объявлена глобальной. Это очень полезно для определения временных имен переменных внутри процедур, или для пущей уверенности, что переменные процедуры никогда не вмешаются в любые глобальные переменные в основном коде.

```
Global a.l = 10

Procedure ChangeValue()
  Protected a.l = 20
EndProcedure

ChangeValue()
Debug a
```

Вы можете видеть, что даже если местная переменная внутри процедуры имеет такое же имя, что и глобальная переменная, она считается другой переменной или переменной в другой области. Если вы запустите выше пример, то увидите результат 10 в окне отладки, хоть мы и вызываем процедуру *ChangeValue()* в которой, мы вроде как присваиваем переменной с таким же именем число 20. Поскольку перед переменной стоит команда *Protected,* процедура ни как не влияет на переменную в основном коде.

Protected(местные) массивы

```
Global Dim Numbers.l(1)

Procedure ChangeValues()
  Protected Dim Numbers.l(1)
  Numbers(0) = 3
  Numbers(1) = 4
EndProcedure

ChangeValues()
Debug Numbers(0)
Debug Numbers(1)
```

В этом примере мы используем команду *Protected* точно так же, как с защищенными переменными. Если вы запустите пример выше, результаты в окне вывода отладки будут нули хоть мы и вызвали процедуру ChangeValues(), в которой меняются данные в ячейках массива. Но как вы поняли это ни как не влияет на глобальный массив в основном коде.

Protected(местные) связанные списки

```
Global NewList Numbers.l()
AddElement(Numbers())
```

```
Numbers() = 1

Procedure ChangeValue()
  Protected NewList Numbers.l()
  AddElement(Numbers())
  Numbers() = 100
EndProcedure

ChangeValue()
SelectElement(Numbers(), 0)
Debug Numbers()
```

Опять же, если вы запустите приведенный выше пример, результат в окне вывода отладки будет 1, поскольку мы сделали местным связанный список внутри процедуры, и он ни как не влияет на глобальный список основного кода .

Команда Shared

Shared (общие) переменные

Иногда в вашем коде вам может понадобиться доступ к переменным внутри процедуры, которая не была определена как глобальная. Для этого используется команда *Shared*. Вот пример:

```
a.l = 10


Procedure ChangeValue()
  Shared a
  a = 50
EndProcedure


ChangeValue()
Debug a
```

Здесь, несмотря на то что переменная изначально не была определена как глобальная, процедура получила доступ к ней с помощью команды *Shared*. При запуске приведенного выше примера, процедура изменяет значение переменной, как будто она объявлена глобальной.

Shared (общие) массивы

```
Dim Numbers.l(1)

Procedure ChangeValues()
   Shared Numbers()
   Numbers(0) = 3
   Numbers(1) = 4
EndProcedure

ChangeValues()
```

```
Debug Numbers(0)
Debug Numbers(1)
```

В этом примере, хотя массив и не определен как глобальный, процедура получила к нему доступ с помощью команды *Shared*. При определении массива общим, нужно просто указать имя массива со скобками без необходимости указывать суффикс.

Shared (общие) связанные списки

```
NewList Numbers.l()

Procedure ChangeValue()
  Shared Numbers()
  AddElement(Numbers())
  Numbers() = 100
EndProcedure

ChangeValue()
SelectElement(Numbers(), 0)
Debug Numbers()
```

В этом примере, хотя связанный список и не определен как глобальный, процедура получила к нему доступ с помощью команды *Shared*. При определении связанного списка общим, нужно просто указать имя связанного списка со скобками без необходимости указывать суффикс.

Команда Static

Статические(постоянные) переменные

Каждый раз, после выхода из процедуры, будут потеряны все значения переменных, определенных в этой процедуре. Если Вы хотите, чтобы процедура помнила значения своих переменных после выхода, то вам нужно использовать команду *Static* перед этими переменными. См. пример:

```
Procedure ChangeValue()
  Static a.l
  a + 1
  Debug a
EndProcedure

For x.l = 1 To 5
  ChangeValue()
Next x
```

Здесь, в процедуре *ChangeValue()* я установил статическую переменную, используя команду *Static*. После этого я увеличил значения этой переменной на 1 и выводил в окно отладки. Я вызвал эту процедуру пять раз с помощью стандартного цикла. Если вы посмотрите на значения, они все разные, и увеличивается на 1 каждый раз. Это происходит потому, что значение переменной процедура запоминает при выходе из нее.

Статические (постоянные) массивы

Массивы, как и переменные тоже могут быть статичными. Ячейки массива сохраняются в процедуре, если использовать для массива команду *Static:*

```
Procedure ChangeValue()
  Static Dim Numbers.l(1)
  Numbers(0) + 1
  Numbers(1) + 1
  Debug Numbers(0)
  Debug Numbers(1)
EndProcedure

For x.l = 1 To 5
  ChangeValue()
Next x
```

В приведенном выше примере я использовал команду *Static* для того чтобы сохранить значения массива между вызовами процедуры так же, как и со статическими переменными. Если вы посмотрите на значения они все разные, и увеличивается на 1 .

Статические(постоянные) связанные списки

```
Procedure ChangeValue()
   Static NewList Numbers.l()
    If CountList(Numbers()) = 0
      AddElement(Numbers())
    EndIf
   SelectElement(Numbers(), 0)
   Numbers() + 1
   Debug Numbers()
   EndProcedure

For x.l = 1 To 5
   ChangeValue()
Next x
```

В этом примере я использовать команду *Static,* чтобы сохранить значения связанного списка между вызовами процедуры точно так же, как и со статическими массивами и переменными. Если вы посмотрите на значения они все разные, и увеличиваются на 1.

Передача переменных процедуре

Как я уже говорил, одна из самых полезных возможностей процедуры является то, что они могут принять первоначальные параметры запуска. Эти параметры могут быть любыми переменными встроенного типа, массивов или даже связанных списков. Параметры используются в качестве способа передачи значений из основной программы в любую процедуру для обработки. Процедуры могут быть определены и использованы снова и снова, по всей программе, с различными значениями. Таким образом, можно определить процедуру добавив параметры:

```
Procedure AddTogether(a.l, b.l)
   Debug a + b
```

```
EndProcedure

AddTogether(10, 5)
AddTogether(7, 7)
AddTogether(50, 50)
```

Все параметры, которые должны быть переданы процедуре должны быть в скобках. И если нужно передать несколько параметров необходимо разделять их запятыми. Когда определяются параметры , как в прошлом примере, им нужно определять тип. Здесь, я определил два параметра: A и B , оба из которых имеют тип *Long*. При вызове процедуры, определенной ранее с параметрами, нужно не забывать про них:

```
AddTogether(10, 5)
```

После этого вызова , значение 10 передается в переменную A, значение 5 передается в переменную B. Затем процедура складывает эти переменные вместе и отображает результат в окне вывода отладки.Вот еще один пример, но с использованием строк:

```
Procedure JoinString(a.s, b.s)
  Debug a + b
EndProcedure


JoinString("Mary had a little lamb, ", "its fleece was white as snow.")
JoinString("And everywhere that Mary went, ", "that lamb was sure to go.")
JoinString("..", "..")
```

Здесь мы используем оператор + для складывания строк в одну. Я изменил название процедуры, а так же тип переменных. Теперь я могу вызывать эту процедуру из любой точки моей программы в соответствии с определенными параметрами. Параметры процедуры могут быть разных типов и их кол-во не ограничено:

```
Procedure LotsOfTypes(a.b, b.w, c.l, d.f, e.s)
   Debug "Byte: " + Str(a)
   Debug "Word: " + Str(b)
   Debug "Long: " + Str(c)
   Debug "Float: " + StrF(d)
   Debug "String: " + e
EndProcedure

LotsOfTypes(104, 21674, 97987897, 3.141590, "Mary had a little lamb")
```

Здесь я использовал несколько различных типов переменных в качестве параметров в мою процедуру *LotsOfTypes()*, чтобы продемонстрировать, что параметры не все должны быть одного типа. Значения, которые передаются при вызове процедуры назначаются в установленном порядке соответственно. Вы также заметили, что все значения совпадают с соответствующими им типами параметров, в противном случае будут синтаксические ошибки. Вы также, вероятно, заметили, я использовал две встроенных функций, которые вам могут быть не знакомы. Эти две функции Str() и StrF() будут подробно описаны в главе 7 (примеры общих команд), но все же объясню проще, они превращают числовые типы в строковые.

Правила передачи значений процедурам

При вызове процедуры необходимо убедиться в том, что параметры указанные в процедуре соответствуют параметрам вызываемой процедуры. Например, если процедура определена так:

AddTogether(a.l, b.l)

тогда только переменные с типом Long могут быть переданы в качестве параметров. Если переменные строкового типа определяются как параметры, вы должны вызвать процедуру применяя строковые параметры. Процедуры с использованием массивов и связанных списков в качестве параметров так же должны быть вызваны правильно и все параметры должны быть в правильном порядке, и иметь правильный тип. Я знаю, это кажется очевидным, но я думаю, что стоит отметить.

Даже из этих простых примеров видно большая полезность процедур. Они не только экономят время с вводом кода, но они позволяют программистам расширять функциональность PureBasic.

Дополнительные параметры переменных

Новыми возможностями в PureBasic V4 стало включение дополнительных параметров в процедуры. Это очень легко объяснить и продемонстрировать. В общем, вы можете задать начальное значение для любого параметра процедуры. Вот пример:

```
Procedure DisplayParameters(a.l, b.l, c.l = 3)
  Debug a
  Debug b
  Debug c
EndProcedure

DisplayParameters(1, 2)
```

Если вы посмотрите на определение процедуры в приведенном выше примере, вы увидите, что последний параметр был определен иначе, чем другие. Если последний параметр не указан в вызове процедуры, он должен быть указан по умолчанию. Я вызываю процедуру *DisplayParameters(1, 2)* без последнего параметра, но вы заметили, что в окне отладки отобразилось три параметра, поскольку один из параметров был указан по умолчанию. Вы должны понимать, что все дополнительные параметры, указанные по умолчанию, должны всегда определяться справа. Это происходит потому, что значения в процедуру передаются слева направо.

Передача массивов процедуре.

Вы также можете передать массивы и связанные списки процедурам в качестве параметров. Их использование является простым. Чтобы передать массив в качестве параметра необходимо сначала определить массив, используя команду Dim. Вот простой одномерный массив строк определенный с четырьмя индексами (помните, что индексы начинаются с 0) я назвал его Countries

```
Dim Countries.s(3)
  Countries(0) = "England"
  Countries(1) = "Northern Ireland"
```

```
  Countries(2) = "Scotland"
  Countries(3) = "Wales"
```

После того как массив был определен, давайте определим процедуру:

```
Procedure EchoArray(Array  MyArray.s(1))
  For x.l = 0 To 3
   Debug MyArray(x)
  Next x
EndProcedure
```

Чтобы определить процедуру с использованием массива в качестве параметра, вам надо определить массив со скобками, в которых указывается размерность массива(в данном случае у нас одномерный), поэтому в скобках стоит 1. Кроме того необходимо перед именем массива использовать ключевое слово **Array.** Далее после имени нужно указывать суффикс ожидаемого типа массива. Параметр-массив выглядит следующим образом:

```
Array MyArray.s(1)
```

После того как процедура была правильно определена, вы можете вызвать ее, передав массив в качестве параметра:

```
EchoArray(Countries())
```

Для передачи массива мы используем только его имя со скобками на конце. Не надо писать типов, индексов или размеров. А теперь этот пример в собранном виде:

```
Dim Countries.s(3)
Countries(0) = "England"
Countries(1) = "Northern Ireland"
Countries(2) = "Scotland"
Countries(3) = "Wales"

Procedure EchoArray(Array MyArray.s(1))
For x.l = 0 To 3
Debug MyArray(x)
Next x
EndProcedure


EchoArray(Countries())
```

В окне отладки должны отобразится четыре страны

Передача многомерных массивов

Можно также передавать многомерные массивы в качестве параметров, как показано в следующем примере:

```
Dim Countries.s(3, 1)
  Countries(0,0) = "England"
```

```
  Countries(0,1) = "57,000,000"
  Countries(1, 0) = "Northern Ireland"
  Countries(1,1) = "2,000,000"
  Countries(2, 0) = "Scotland"
  Countries(2,1) = "5,200,000"
  Countries(3, 0) = "Wales"
  Countries(3,1) = "3,100,000"

Procedure EchoArray(Array MyArray.s(2))
  For x.l = 0 To 3
    Debug MyArray(x,0) + " - " + "Population: " + MyArray(x,1)
  Next x
EndProcedure


EchoArray(Countries())
```

Хотя это двумерный массив, мы по-прежнему используем те же правила, как и для одномерного, с небольшой разницей: в параметре-массиве, передаваемом процедуре используется цифра 2, указывающая на то, что будет передаваться двумерный массив в качестве параметра:

```
Array MyArray.s(2)
```

Далее вызываем процедуру передавая массив как и прежде, без каких-либо типов, индексов или размеров:

```
EchoArray(Countries())
```

Передача массивов структурированных типов

Структурированные массивы так же могут быть переданы процедуре в качестве параметра. Все делается так же как и с простыми массивами, меняется лишь тип массива в соответствии со структурой. Посмотрите на следующий пример:

```
Structure COUNTRY
  Name.s
  Population.s
EndStructure

Dim Countries.COUNTRY(3)
 Countries(0)\Name = "England"
 Countries(0)\Population = "57,000,000"
 Countries(1)\Name = "Northern Ireland"
 Countries(1)\Population = "2,000,000"
 Countries(2)\Name = "Scotland"
 Countries(2)\Population = "5,200,000"
 Countries(3)\Name = "Wales"
 Countries(3)\Population = "3,100,000"

Procedure EchoArray(Array MyArray.COUNTRY(1))
  For x.l = 0 To 3
```

```
    Debug MyArray(x)\Name + " - " + "Population: " + MyArray(x)\Population
  Next x
EndProcedure

EchoArray(Countries())
```

Здесь я использовал структурированный массив в качестве параметра: Array MyArray.COUNTRY(1)

Передача Связанных списков процедуре

Иногда при программировании понадобится использовать связанный список в качестве параметра, передаваемого процедуре. Это очень похоже на передачу массива:

```
NewList Numbers.l()
  AddElement(Numbers())
  Numbers() = 25
  AddElement(Numbers())
  Numbers() = 50
  AddElement(Numbers())
  Numbers() = 75

Procedure EchoList(List MyList.l())
  ForEach MyList()
    Debug MyList()
  Next
EndProcedure

EchoList(Numbers())
```

Здесь я создал стандартный связанный список, называемый Number.l(), с типом Long. После добавления трех элементов в этот список, я определил процедуру Echolist() со связанным списком в качестве параметра. Как и с массивами, параметр - связанный список состоит из имени параметра, за которым следует тип и, наконец, набор скобок. Связанные списки не используют индексы, как массивы, так что вам не нужно вводить номер в скобках. Так же необходимо перед именем в связанном списке использовать ключевое слово **List .** Когда все это сделано, можно вызвать процедуру, передав связанный список как массив, без какого-либо типа и т.д. :

```
EchoList(Numbers())
```

В принципе, если вы усвоили передачу массивов в качестве параметров, у вас не должно возникнуть недопонимания с передачей связанного списка.

Передача структурированных связанных списков

Структурированные связанные списки так же легко передаются в качестве параметров процедуре. Все как и с массивами меняется только тип списка в соответствии со структурой:

```
 Structure FLAGS
   Country.s
```

```
  Flag.s
EndStructure

NewList Info.FLAGS()
  AddElement(Info())
  Info()\Country = "Great Britain"
  Info()\Flag = "Union Jack"
  AddElement(Info())
  Info()\Country = "USA"
  Info()\Flag = "Stars And Stripes"
  AddElement(Info())
  Info()\Country = "France"
  Info()\Flag = "Tricolore"

Procedure EchoList(List MyList.FLAGS())
  ForEach MyList()
    Debug MyList()\Country + "'s flag is the " + MyList()\Flag
  Next
EndProcedure

EchoList(Info())
```

И сам параметр из кода: List MyList.FLAGS()

Возврат значения из процедур

Еще одна интересная особенность процедуры: они могут возвращать значения. Это значение может быть любого встроенного типа. Возврат значений из процедуры хорош для вычисления или манипулирования данными в рамках процедуры, а затем возвращения чего-нибудь полезного. Это может быть расчет результата, код ошибки и т д. Синтаксис для возврата значений из процедуры прост. Сначала нужно определить, какой тип должен быть возвращен из процедуры. Это делается при определении процедуры. Посмотрите на следующий пример:

```
Procedure.l AddTogether(a.l, b.l)
  ProcedureReturn a + b
EndProcedure

Debug AddTogether(7, 5)
```

После слова Procedure следует суффикс того типа, который следует возвратить из процедуры. Для возврата значения используется команда ProcedureReturn . Если вы хотите использовать для возврата другой тип данных, то это можно сделать примерно так:

```
Procedure.s JoinString(a.s, b.s)
  ProcedureReturn a + b
EndProcedure

Debug JoinString("Red ", "Lorry")
Debug JoinString("Yellow ", "Lorry")
```

Обратите внимание на тип возвращаемого значения, указанного в ключевом Procedure.s. Это указывает, что теперь эта процедура будет возвращать строку. Процедуры, возвращающие значения, могут использоваться везде, где есть выражения. Например их можно использовать так:

```
Procedure.l AddTogether(a.l, b.l)
  ProcedureReturn a + b
EndProcedure

Debug AddTogether(AddTogether(2, 3), AddTogether(4, 1))
```

В данном примере попросту складываются четыре числа, но довольно изощренно.

Параметры не влияют на тип возврата

Следует помнить, что при использовании возврата из процедуры тип возврата может быть отличным от параметров процедуры. Например:

```
Procedure.s DisplayValue(a.l, b.s)
  ProcedureReturn Str(a) + b
EndProcedure

x = 5
While x >= 0
 Debug DisplayValue(x, " green bottles hanging on the wall.")
 x - 1
Wend
```

Хоть я и использовал тип Long в параметрах процедуры, она должна вернуть строковой тип данных, именно поэтому в возвращаемом значении одна из переменных с помощью функции Str() преобразована в строковой тип. А потом уже складываются обе строки в одну и делается возврат значения из процедуры.

Ограничения связанные с возвратом результатов

При использовании возврата значений из процедур существуют два ограничения. Во-первых, процедура может возвращать только один результат. Это означает, что вы не можете вернуть два или более значений в одном вызове процедуры. Во-вторых, вы не можете вернуть массив, связанный список или любой определяемый пользователем тип. Вы можете возвращать только встроенный в PureBasic тип. Вместе с тем, возможно обойти эти ограничения, но узнать об этом можно из главы 13.

6

Использование встроенных команд

Языки программирования ничто без встроенной библиотеки полезных команд. Так и PureBasic располагает более 800 встроенных команд для использования в рамках своих

программ. Они используются для манипуляций со строками, математическими вычислениями, для обработки файлов и даже графического интерфейса. Они охватывают практически любые мыслимые задачи программирования и если чего то нет, отвечающего вашим потребностям, вы можете создать собственные процедуры. В этой главе я расскажу и объясню наиболее часто используемые встроенные команды PureBasic. Хотя это далеко не полный список, все же это введение должно послужить вам верой и правдой. Эта глава начинается с описания синтаксиса команд из файла справки PureBasic, а затем переходит к фактическому описанию команд и пояснению их. Закончу же эту главу разделом о том, как работать с файлами, например: загрузка, запись или чтение, все с использованием встроенных команд PureBasic.

Использование файла справки PureBasic

Здесь мы постараемся понять как организованы записи на страницах файла справки, как их читать и что более важно, как использовать встроенные команды, которые описаны там. При просмотре страниц в справке вы столкнетесь с такой формой записи:

Синтаксис - Описание - Пример - Поддерживаемые операционные системы.

Все разделы по сути сами объясняют себя, но давайте сконцентрируемся на синтаксисе. В верхней части идет название команды, например: SaveImage(). Давайте найдем ее в файле справке: (Helpfile:Reference Manual->General Libraries->Image->SaveImage). Далее идет раздел синтаксис, примерно так:

SaveImage(#Image, FileName$ [, ImagePlugin [, Flags]])

Первым идет фактическое имя команды, в нашем случае *SaveImage* после чего идут пара скобок, в которых следуют параметры. В синтаксическом примере показано, что эта команда может принимать четыре параметра. Вы также заметили, что последние два параметра показаны в квадратных скобках. Это означает, что эти последние два параметра являются необязательными, и могут не использоваться для вызова SaveImage(). Давайте посмотрим на примеры с использованием необязательных параметров и без них:

```
SaveImage(1, "Image.bmp")
SaveImage(1, "Image.jpg", #PB_ImagePlugin_JPEG)
SaveImage(1, "Image.jpg", #PB_ImagePlugin_JPEG, 10)
```

Первый пример будет сохранить изображение с именем **image.bmp** по умолчанию в 24 битовый растровый формат. Второй пример будет сохранить изображение в формате JPEG и использовать стандартную степень сжатия. Третий пример будет сохранить изображение в формат JPEG и использовать максимальное значения сжатия 10!

Квадратные скобки в рамках примеров

Квадратные скобки, показанные в синтаксических примерах никогда не используются при использовании этих команд. Они только отделяют в примерах необязательные параметры для лучшего понимания, но в реальном коде их быть не должно. Квадратные скобки в реальном коде используются только для статических массивов см. главу 5.

Вызов встроенных команд чем похож на вызов процедур. Каждая из встроенных команд должна вызываться с правильным синтаксисом передачи параметров, иначе компилятор оповестит вас об ошибках в коде. Структура приведенного выше примера синтаксиса

доминирует по всему файлу справки, поэтому если вы поняли синтаксис этот примера, вам не составит труда понять и другие.

PB Числа и OS идентификаторы

При использовании встроенных команд, важно понимать роль нумерации объектов PureBasic а так же идентификаторов операционной системы , так как они непосредственно используются при управлении программой. Оба являются не более чем цифры, и оба используются для идентификации объектов в программах. Например, чтобы определить части графического интерфейса пользователя или различные изображения. Знание как и когда использовать их, имеет важное значение не только для PureBasic но в программировании в целом. Справка PureBasic имеет большое кол-во информации, но она может быть немного непонятной при использовании нумерации объектов PureBasic и идентификаторами операционной системы . Попробуем разобраться получше:

PB числа

PureBasic работает на нескольких системах и использует свою нумерацию для идентификации каждого объекта в вашей программе. Этим объектом может быть окно, гаджет или файл изображения. PB числа используются в программе для того, чтобы ссылаться на эти объекты позже, при выполнении определенных действий, связанных с ними. Многим командам необходимо один или два PB числа в качестве параметров. Они приводятся в справке PureBasic в синтаксическом примере любой команды. Числа PB обычно показаны как константы. Начинаются с решетки (#) и заканчиваются названием библиотеки, которая находится в команде. Вот пример (Helpfile:Reference Manual->General Libraries->Image->CreateImage):

```
CreateImage(#Image, Width, Height)
```

Как вы можете видеть, первым параметром, показана константа #Image. Но это не означает, что использование константы является обязательным. Можно использовать любое целое число для объекта, лишь бы оно было уникальным среди других объектов такого типа. Это число для созданного объекта будет закреплено за ним на протяжении всей программы.

Одинакового типа объекты могут конфликтовать с одинаковыми PB числами

PureBasic числа обеспечивают отличный способ для обозначения всего, что создает PureBasic на индивидуальной основе. Из-за этого вы не можете иметь два одинаковых объекта с одним номером. Если вы создадите изображение с номером 1, а затем вы создадите еще одно изображение с номером 1, PureBasic автоматически уничтожает первое и освободит память для создания второго.Эта функция очень удобна, для замены объектов в любое время. Одинаковые числа можно использовать для объектов из разных библиотек. Они не будут конфликтовать. Но в коде это может вызвать путаницу для вас самих. Поэтому многие программисты, при использовании разных команд для создания объектов, используют константы вместо чисел. Эти константы обычно определяются в блоке перечисления. Вот пример:

```
Enumeration
  #IMAGE_DOG
  #IMAGE_CAT
  #IMAGE_BIRD
```

```
EndEnumeration
CreateImage(#IMAGE_DOG, 100, 100)
CreateImage(#IMAGE_CAT, 250, 300)
CreateImage(#IMAGE_BIRD, 450, 115)
```

После того как константы были определены, я могу использовать их для создания изображений и не беспокоиться о конфликтах с индефикацией. В течение программы я могу использовать эти константы для обозначения изображения прямо по имени. Например, теперь я могу сослаться на первое изображение с помощью константы #IMAGE_DOG . Этот метод использования констант для чисел PureBasic гарантирует четко организованный и читаемый код.

Динамические PB числа

В качестве альтернативы использованию блоков перечисления для обработки вновь созданных объектов, вы можете использовать специальную константу. Эта константа:

```
#PB_Any
```

Эта константа может использоваться везде, где PB число ожидается в команде создания объекта. Это бывает нужно для построения динамических программ, где вы не можете знать, сколько объектов будет создаваться. Каждой в конце концов будет присвоен свой иденфикатор Вот пример:

```
ImageDog.l = CreateImage(#PB_Any, 100, 100)
```

Иденфикаторы операционной системы

Некоторым объектам, создаваемым с помощью встроенных команд PureBasic, присваиваются номера операционной системы. За этими значениями следит операционная система. Это например окна, шрифты и изображения, и т.д. В справке PureBasic они указываются как ID. При программировании в PureBasic вы заметите, что некоторые команды потребуют в качестве параметров идентификаторы ОС. Вот несколько команд, которые возвращают идентификатор ОС:

```
WindowOSId.l = WindowID(#Window)
GadgetOSId.l = GadgetID(#Gadget)
ImageOSId.l = ImageID(#Image)
```

Я перечислил здесь три , но есть еще несколько. Названия команд, которые заканчиваются обычно ID() возвращают ОС идентификаторы. Вот пример команды, которая использует ОС идентификатор в качестве параметра:

```
CreateGadgetList(WindowOSId)
```

Эта команда используется для создания списка гаджетов в окне и использует операционную систему для определения идентификаторов, в которой создается список. Идентификатор ОС используется с этой командой вместо PB числа для максимальной совместимости, только в случае, если окно было создано с помощью API операционной системы. Если вы хотите использовать эту команду, чтобы создать список гаджетов окна, созданного с помощью системы счисления PureBasic, то можно так:

CreateGadgetList(WindowID(#Window))

Мы используем команду WindowID() в качестве параметра, который возвращает идентификатор окна ОС с помощью PB числа . Каждая операционная система имеет собственный интерфейс прикладного программирования или API для краткости. Это встроенная команда используется для всех языков программирования и служит для создания интерфейса. Идентификаторы используются для отслеживания и доступа ко всем объектам, которые содержит операционная система . Все ОС идентификаторы являются уникальными для каждой программы, даже не написанной в PureBasic. Операционная система для отслеживания тысяч наименований, использует длинные номера из 8 цифр, так что не удивляйтесь. ОС идентификаторы могут быть не использованы для создания программ PureBasic, но начинающие должны знать, что когда нибудь настанет время для использования их. ОС идентификаторы играть важную роль при использовании API любой операционной системы, в особенности Windows API, об этом чуть больше в главе 13.

Примеры общих команд

В этом разделе я собираюсь продемонстрировать некоторые из наиболее часто используемых команд в PureBasic. Обычно эти команды используются в большинстве программ PureBasic, поэтому обучение их синтаксиса будет только полезным. Все эти команды существуют в различных библиотеках. Здесь они представлены ниже в алфавитном порядке.

Asc()

(Helpfile:Reference Manual->General Libraries->String->Asc)

Синтаксис: ASCIIValue.l = Asc(Character.s)

Эта команда возвращает ASCII значения строки. В стандартный наборе символов ASCII цифры от 0 до 255 используются для представления символов и кодов управления компьютером. Передаваемым параметром этой команды является строка, и команда Asc возращает ASCII код одного символа этой строки, в данном случае первого. На следующем примере возвращаются все символы строки:

```
Text.s = "This is a test"
For x.l = 1 To Len(Text)
 Debug Mid(Text, x, 1) + " : " + Str(Asc(Mid(Text, x, 1)))
Next x
```

С помощью цикла *FOR-NEXT* и команды *Mid*, передаются по очереди все символы в качестве параметров команде ASC. Она выдает ASCII коды для них и конечно все они выводятся в окно отладки.

Chr()

(Helpfile:Reference Manual->General Libraries->String->Chr)

Синтаксис: Character.s = Chr(ASCIIValue.l)

Эта команда возвращает один символ из полученного кода ASCII. Эта команда прямая противоположность ASC. Параметром служит число от 0 до 255.

```
Text.s = Chr(84) + Chr(104) + Chr(105) + Chr(115) + Chr(32)
Text.s + Chr(105) + Chr(115) + Chr(32)
Text.s + Chr(97) + Chr(32)
Text.s + Chr(116) + Chr(101) + Chr(115) + Chr(116)
Debug Text
```

Приведенный выше пример строит строку This is a test за счет сложения нескольких Chr() команд. Например, первая команда Chr(84) возвращает символ T. См. приложение файла справки полного списка ASCII таблицы, показывающей все символы и связанных с этими символами числа от 0 до 255.

Delay()

(Helpfile:Reference Manual->General Libraries->Misc->Delay)

Синтаксис: Delay(Milliseconds)

Эта команда паузы для деятельности всей программы на срок, указанный в параметре в миллисекундах. (Миллисекунда является одной тысячной секунды).

```
Debug "Start..."
Delay(5000)
Debug "This is executed 5 seconds later"
```

После запуска программы высветится в окне отладки **Start..** Затем ожидание 5 секунд. Далее высветится **This is executed 5 seconds later.** И программа завершится.

ElapsedMilliseconds()

(Helpfile:Reference Manual->General Libraries->Misc->ElapsedMilliseconds)

Синтаксис: Result.l = ElapsedMilliseconds()

Эта команда возвращает количество миллисекунд, прошедших с определенного ранее времени.

```
Debug "Start..."
StartTime.l = ElapsedMilliseconds()
Delay(5000)
Debug "Delayed for "+Str(ElapsedMilliseconds() - StartTime)+" milliseconds."
```

Эта команда может понадобиться, чтобы выполнить любой вид синхронизации в вашей программе. Например, если нужно измерить какой то промежуток времени, вы первоначально задаете команду ElapsedMilliseconds(), далее запускаете таймер, и снова с помощью команды ElapsedMilliseconds() делаете вычисление промежутка времени. Чтобы

получить результат, вычтите из времени окончания начальное время. Все продемонстрировано в вышеупомянутом примере. Когда вы используете эту команду впервые, вы можете быть потрясены возращенным большим числом. Вы должны помнить, что запись делается миллисекундах.

FindString()

(Helpfile:Reference Manual->General Libraries->String->FindString)

Синтаксис: Position.l = FindString(String.s, StringToFind.s, StartPosition.l)

Эта команда ищет параметр StringToFind в рамках параметра String, начиная с позиции, указанной параметром StartPosition. Если с параметром 1, он находит эту строку сразу возвращает позицию 1.

```
String.s = "I like to go fishing and catch lots of fish"
StringToFind.s = "fish"
FirstOccurrence.l = FindString(String, StringToFind, 1)
SecondOccurrence.l = FindString(String, StringToFind, FirstOccurrence + 1)
Debug "Index of the first occurrence: " + Str(FirstOccurrence)
Debug "Index of the second occurrence: " + Str(SecondOccurrence)
```

Этот пример показывает, как найти строку в другой строке. Первым делом команда FindString() пытается найти строку *FISH* начиная с позиции 1. Поиск успешен и в переменную FirstOccurrence заносится значение 14. Это как раз место в строке *I like to go fishing and catch lots of fish* , по которому стоит искомая *FISH*. Следующий поиск ведется с 15 места. Поскольку FirstOccurrence+1=15. Поиск опять оказался успешным, и на этот раз строка *FISH* найдена на позиции 40. Это число заносится в переменную SecondOccurrence И под конец оба результата выведены в окно отладки.

Len()

(Helpfile:Reference Manual->General Libraries->String->Len)

Синтаксис: Length.l = Len(String.s)

Эта команда возвращает длину строки, то есть кол-во символов в строке, которая идет в качестве параметра для команды.

```
Alphabet.s = "abcdefghijklmnopqrstuvwxyz"
LengthOfString.l = Len(Alphabet)
Debug LengthOfString
```

Этот очень простой пример команды Len(). Переменной Alphabet присваивается строка с символами. Далее переменной LenghtOfString присваивается значение, которое будет обработано командой Len() с параметром Alphabet . И значение переменной LengthOfString выводится в окне вывода отладки.

MessageRequester()

(Helpfile:Reference Manual->General Libraries->Requester->MessageRequester)

Синтаксис: Result.l = MessageRequester(Title.s, Text.s [, Flags])

Возможные Flags:
#PB_MessageRequester_Ok
#PB_MessageRequester_YesNo
#PB_MessageRequester_YesNoCancel
Возможные возращаемые значения:
#PB_MessageRequester_Yes
#PB_MessageRequester_No
#PB_MessageRequester_Cancel

Эта команда используется для создания небольшого окна, которое может показывать любую информацию в вашей программе. Она может быть использована в любом месте программы. Первый параметр в этой команде, это строка, отображаемая в строке заголовка окна. Вторым параметром является фактическая строка сообщения, отображаемая в самом окне. Третий и последний параметр для необязательных флагов. С помощью разных флагов в последнем параметре, вы можете изменить стиль окна запроса, включением различных кнопок. Ваша программа будет остановлена, пока одна из этих кнопок не будет нажата. Чтобы понять лучше назначение каждой из них пример:

```
Title.s = "Information"
Message.s = "This is the default style message requester"
MessageRequester(Title, Message, #PB_MessageRequester_Ok)
Message.s = "In this style you can choose 'Yes' or 'No'."
Result.l = MessageRequester(Title, Message, #PB_MessageRequester_YesNo)

If Result = #PB_MessageRequester_Yes
  Debug "You pressed 'Yes'"
Else
  Debug "You pressed 'No'"
EndIf

Message.s = "In this style you can choose 'Yes' or 'No' or 'Cancel'."
Result.l = MessageRequester(Title, Message, #PB_MessageRequester_YesNoCancel)

Select Result
 Case #PB_MessageRequester_Yes
   Debug "You pressed 'Yes'"
 Case #PB_MessageRequester_No
   Debug "You pressed 'No'"
 Case #PB_MessageRequester_Cancel
   Debug "You pressed 'Cancel'"
EndSelect
```

Этот пример показывает различные способы использования команды MessageRequester. Он также показывает вам понять, как использовать константы возвращаемого значения для определения того, какая была нажата кнопка. Использование констант, помогает не беспокоиться по поводу запоминания числовых значений. Вам не нужно знать, какие значения присваиваются этим константам, важно знать равны они или нет возвращаемому значению.

Mid()

(Helpfile:Reference Manual->General Libraries->String->Mid)

Синтаксис: Result.s = Mid(String.s, StartPosition.l, Length.l)

Эта команда возвращает строку, которая вырезана из другой строки. Первым параметром является строка из которой нужно вырезать нужный отрывок. Вторым параметром является стартовая позиция для будущей вырезанной строки. Третий параметр это длина будущей строки. Вот пример:

```
StartingString.s = "Hickory Dickory Dock"
ExtractedString.s = Mid(StartingString, 17, 4)
Debug ExtractedString
```

Здесь я извлекаю строку Dock, задав начальную позицию 17, с длиной 4 символа. Подобно команде FindString() позиция в исходной строке измеряется в символах.

Random()

(Helpfile:Reference Manual->General Libraries->Misc->Random)

Синтаксис: Result.l = Random(Maximum.l)

Эта команда просто возвращает случайное число между нулем и значением, определенным в параметре включительно.

```
Debug Random(100)
```

В приведенном выше примере команда Random() вернет случайное значение в диапазоне от 0 до100 включительно.

Str(), StrF(), StrQ(), StrD()

(Helpfile:Reference Manual->General Libraries->String->Str)

Синтаксис:

```
Result.s = Str(Value.b)
Result.s = Str(Value.w)
Result.s = Str(Value.l)
Result.s = StrF(Value.f [, DecimalPlaces.l])
Result.s = StrQ(Value.q)
Result.s = StrD(Value.d [, DecimalPlaces.l])
```

Эти четыре команды по сути одинаковы. Их основное использование: переводить числа в строки. **Str()** для обработки чисел с типом **.b (byte) .w (word)** и **.l (long).** **StrF()** используется для обработки чисел с типом **.f (Float)**. **StrQ()** для обработки чисел с типом **.q (Quads)**. **StrD()** для преобразования чисел с типом **.d (Doubles)** В случае с StrF() и StrD() вы заметили, что существует также дополнительный параметр называющийся DecimalPlaces. Если используется этот параметр, то он определяет, сколько знаков после запятой преобразовать с округлением, остальные будут отрезаны

```
Debug "Long converted to a String: " + Str(2147483647)
Debug "Float converted to a String: " +StrF(12.05643564333454646, 7)
Debug "Quad converted to a String: " +StrQ(9223372036854775807)
Debug "Double converted to a String: " +StrD(12.05643564333454646, 14)
```

Приведенный пример показывает, как преобразовать четыре различные числовые значения в строки. В каждом случае после преобразования они выводятся в окне вывода отладки. Важно помнить: если вы не используете параметр DecimalPlaces, то в случае с StrF() по умолчанию будут преобразованы только шесть знаков после запятой, а в случае с StrD() десять знаков.

Val(), ValF(), ValQ(), ValD()

(Helpfile:Reference Manual->General Libraries->String->Val)

Синтаксис:

```
Result.l = Val(String.s)
Result.f = ValF(String.s)
Result.q = ValQ(String.s)
Result.d = ValD(String.s)
```

Эти четыре команды по сути прямая противоположность команде STR(). То есть эта команда возвращает числовые значения из строк. Вот простой пример:

```
LongTypeVar.l = Val("2147483647")
FloatTypeVar.f = ValF("12.05643564333454646")
QuadTypeVar.q = ValQ("9223372036854775807")
DoubleTypeVar.d = ValD("12.05643564333454646")


Debug LongTypeVar
Debug FloatTypeVar
Debug QuadTypeVar
Debug DoubleTypeVar
```

В этом примере я использовал все четыре различных типа данных с командой Val() для преобразования четырех строк в их числовые типы. Вы должны помнить, что если в заложенных типах используются значения большие чем должны быть, то конечный результат сбивается, то есть становится не корректным.

Работа с файлами

Использование файлов является естественным для любой программы, как для хранения так и для извлечения данных. PureBasic обеспечивает полную поддержку для чтения и записи файлов, и может читать и записывать любое количество файлов практически одновременно. Описание всех команд обработки файлов вы можете найти в файле справки:

(Helpfile:Reference Manual->General Libraries->File)

Я не буду объяснять все команды описанные в вышеупомянутом разделе, но объясню команды для чтения и записи файлов. А остальные команды вы легко поймете, при первом же использовании их.

Запись в файл

Давайте начнем с примера о том, как писать некоторые данные в файл, то здесь мы напишем несколько строк в простой текстовый файл.

```
#FILE_RHYME = 1
Dim Rhyme.s(3)
Rhyme(0) = "Baa baa black sheep, have you any wool?"
Rhyme(1) = "Yes sir, yes sir, three bags full!"
Rhyme(2) = "One for the master, one for the dame,"
Rhyme(3) = "And one for the little boy who lives down the lane."

If CreateFile(#FILE_RHYME, "Baa baa.txt")
    For x.l = 0 To 3
      WriteStringN(#FILE_RHYME, Rhyme(x))
   Next x
  CloseFile(#FILE_RHYME)
EndIf
```

Для начала я использовал константу #FILE_RHYME для нумерации(идентификации) нашего файла. Далее я определил строковой массив, содержащий четыре ячейки со строками, в которых записан детский стишок. Далее перейдем к записи его в файл. Если бы я хотел записать стишок в существующий файл, тогда я бы использовал команду OpenFile(), но в приведенном выше примере, я хочу записать в новый файл, поэтому я использую команду CreateFile() Эта команда имеет два параметра: первый уникальный идентификатор файла, а второй имя создаваемого файла. После проверки на создания файла и возврата TRUE(истина), программа перейдет к циклу FOR-NEXT. Если не использовать проверку, может случится так, что файл по каким то причинам не создается. Это вызовет ошибки в программе, что чревато вылетом или зависанием программы.

Для записи строк в файл, я использовал команду WriteStringN(). Она имеет два параметра. Первый: идентификатор файла, в который будет осуществлена запись. А второй собственно сама строка, которую надо записать. Я использовал цикл для считывания из массива, и записи считанных строк в файл. Команда WriteStringN() используемая в этом примере представляет собой расширенную команду WriteString(). На самом деле единственное различие заключается в букве N перед скобками. Этот символ в команде означает, что при следующем вызове команды (**WriteString() илиWriteStringN()**) запись будет с новой строки. Когда запись в файл закончена, я закрываю файл, используя команду CloseFile()У нее есть всего один параметр - это идентификатор файла , который надо закрыть.

Различные способы создания или открытия файлов

При использовании команд для работы с файлами (чтения или записи) нужно выбрать правильную команду в зависимости от того, как вы хотите использовать этот файл. В приводимом ниже списке описываются команды для записи и чтения:

ReadFile() Открывает файл только для чтения

OpenFile() Открывает файл для чтения или записи. Так же создает его, если он не существует.

CreateFile() Создает пустой файл для записи. Если файл уже существует, он заменяет его пустым.

Все команды имеют по два параметра. Первым параметром является идентификатор файла, с которым будет работать программа, вторым параметром является фактическое имя файла на диске.

Этот файл, содержащий детский стишок, должен сейчас быть где-то на вашем жестком диске. Если вы создаете файл, используя относительное имя файла, такое как Baa baa.txt , то созданный файл будет находиться в том же каталоге, что и файл исходного кода. Если же вам нужен файл, который будет создан в каком-то конкретном месте, вы должны указать полный путь, который должен включать имя диска, директорий и имя файла. Пример:

```
...
If CreateFile(#FILE_RHYME, "C:\My Directory\Baa baa.txt")
...
```

При использовании этого метода, всегда нужно убедиться, что все каталоги, указанные в пути существуют, прежде чем записать файл, в противном случае создание файла не удастся.

Важность команды CloseFile()

Когда вы закончили чтение, и особенно запись файлов, вы должны закрыть файл командой CloseFile(). Эта команда не только закрывает выбранный файл, но освобождает его для других программ. Еще одна важная роль команды CloseFile() в том, что она полностью записывает в файл любые данные, которые были оставлены в буфера файла . PureBasic использует системный буфер для увеличения производительности доступа к файлу, так что если вы открываете файл в другом редакторе, а некоторых данных будет там не хватать, вы должны проверить, что вы закрыли файл правильно используя CloseFile(). Для новичков в программировании, я думаю не стоит забивать голову знаниями о буфере файла, нужно просто закрывать файлы каждый раз, когда вы закончите с ним работу, во избежание любых ошибок.

Запись других встроенных типов данных

Запись других типов данных происходит точно так же как и запись строк. Теперь, когда вы понимаете, как работает WriteString(), по аналогии не трудно понять и другие команды для записи данных. Вот пример синтаксиса для написания других встроенных типов данных:

```
WriteByte(#File, Value.b)
WriteChar(#File, Value.c)
WriteWord(#File, Value.w)
WriteLong(#File, Value.l)
WriteQuad(#File, Value.q)
WriteFloat(#File, Value.f)
WriteDouble(#File, Value.d)
```

Эти команды могут быть использованы для записи любого встроенного типа значений в файл, а все команды можно использовать столько раз, сколько захотите. Параметров как и у WriteString() два. аналогично первым идет идентификатор файла, вторым данные, которые надо записать.

Чтение из файлов

Чтение из файлов так же просто, как и запись в них. Посмотрите на следующий пример: в нем читаются строки из файла с именем Report.txt, и помещаются в связный список с именем FileContents. Содержание этого связанного списка затем выводится в окне вывода отладки просто для наглядности.

```
#FILE_REPORT = 1
NewList FileContents.s()
If ReadFile(#FILE_REPORT, "Report.txt")
  While Eof(#FILE_REPORT) = #False
    AddElement(FileContents())
    FileContents() = ReadString(#FILE_REPORT)
  Wend
 CloseFile(#FILE_REPORT)
EndIf


ForEach FileContents()
   Debug FileContents()
Next
```

В начале кода я использовал константу, для идентификации(нумерации) файла. Затем создал связанный список с именем FileContents. Вы можете не использовать связанный список для чтения строк, просто я считаю, что это удобно. Я использовал связанный список здесь, потому что они легок в использовании и может расти вместе с новыми пунктами, в отличие от массивов, размерность которых определена. Открываем файл, используя команду ReadFile(). Как вы помните, это открывает его только для чтения. Разрешения на запись отсутствуют. Как и прежде с CreateFile(), я использовал команду ReadFile() в рамках логического оператора IF-ENDIF. После того как файл открыт, мне нужна информация только для чтения. Я использовал цикл While-Wend для повторных команд чтения строки. Цикл работает пока не будет достигнут конец файла. В этом помогает команда EOF(). Как только эта команда вернет истину(то есть достигнут конец файла), чтение прекратится. Каждый раз при прохождении цикла, читается новая строка и заносится в связанный список. После прекращения работы цикла командой CloseFile()

закрывается файл и передается управление другому циклу ForEach-Next, который считывает данные из связанного списка и выводит их в окно отладки.

Чтение других типов данных из файла

Хотя последний пример касается строк, можно также легко прочитать другие типы данных из файла. PureBasic предусматривает конкретные команды для каждого из встроенных типов данных:

```
ReadByte(#File)
ReadChar(#File)
ReadWord(#File)
ReadLong(#File)
ReadQuad(#File)
ReadFloat(#File)
ReadDouble(#File)
```

Для простоты, все эти команды схожи с командой ReadString(). Единственное различие между этими командами то, что ReadString() считывает всю строку, а эти команды считывают только одно значение из файла, которое записано в имени команды. То есть команда ReadLong() считает только четыре байта. Далее указатель доступа к файлу перемещается вперед на 4 байта, для чтения других 4 байт.

Указатель доступа к файлам

В рамках каждого открытого файла существует невидимый указатель доступа к файлу. Эта мнимая позиция, откуда вы будете читать или записывать данные в открытом файле. При использовании ReadFile(), OpenFile() или CreateFile() команд, доступ указателя начинается с начала файла, и готов к следующей операции. Как только вы начинаете чтение или запись, указатель доступа к файлу начинает движение. Если идет запись в файл, то доступ указателя будут осуществляться на основе ее, двигаясь на позицию сразу после получения последнего записанного значения, готовый к следующей операции. Если вы читаете из файла, то доступ указателя будет двигаться к концу файла и достигнет его как только команда Eof() = 0(#False)

Получение позиции указателя

Во время чтения или записи в файл, вы можете получить расположение указателя доступа к файлам с помощью команды Loc(). Возвращаемое значение измеряется байтами.

(Helpfile:Reference Manual->General Libraries->File->Loc)

Синтаксис: Position.q = Loc(#File)

Эта команда вернет позицию указателя доступа к файлу (в байтах), указанного в параметре #File. Значение позиции возвращается восьмибайтовым числом(Quad) для поддержки больших файлов.

Перемещение указателя в файле

Указатель доступ к файлам может быть перемещен в любое время с помощью команды FileSeek() .

(Helpfile:Reference Manual->General Libraries->File->FileSeek)

Синтаксис: FileSeek(#File, NewPosition.q)

Эта команда имеет два параметра: первый номер(идентификатор)файла, вторым, является новая позиция (в байтах) указателя доступа к файлу. Она позволяет читать с любой позиции, в рамках какой-то файла в любое время. Параметр NewPosition должен быть восьмибайтовым(Quad) числом для поддержки больших файлов.

Выяснение текущего размера файла

Чтобы узнать размер файла, можно использовать команду Lof(), высчитывающую длину файла в байтах.

(Helpfile:Reference Manual->General Libraries->File->Lof)

Синтаксис: Length.q = Lof(#File)

Эта команда вернет размер файла, указанного в параметре #FILE в байтах. Длина возвращается числом с типом Quad для поддержки больших файлов.

Пример для Loc, Lof и FileSeek

В следующем примере я собираюсь продемонстрировать использование Loc(), Lof() и FileSeek(). В приведенном ниже коде я читаю музыкальный файл MP3 и пытаюсь обнаружить, ID3 (V1) тег, встроенный внутри него. Эти теги обычно содержат информацию, такую как имя исполнителя, название песни и жанр, и т.д. После чтения спецификаций в формате MP3 ID3 тэгов в Интернете, я обнаружил, что теговая информация всегда добавляется в конец MP3-файла и этот тег всегда 128 байт. Спецификация упоминает, что первые 3 байта этого тега символы Tag. Вот это и проверяется данным примером:

```
#FILE_MP3 = 1
MP3File.s = "Test.mp3"

If ReadFile(#FILE_MP3, MP3File)
    FileSeek(#FILE_MP3, Lof(#FILE_MP3) - 128)
        For x.l = 1 To 3
          Text.s + Chr(ReadByte(#FILE_MP3))
        Next x
 CloseFile(#FILE_MP3)
        If Text = "TAG"
            Debug "'" + MP3File + "' has an ID3v1 tag embedded within it."
        Else
            Debug "'" + MP3File + "' does not have an ID3v1 tag embedded within it."
        EndIf
EndIf
```

После открытия файла с помощью ReadFile() перемещаем указатель в нужное место. Выяснив длину файла отнимаем от нее 128 байт и перемещаем указатель на это значение:

```
FileSeek(#FILE_MP3, Lof(#FILE_MP3) - 128)
```

После того как указатель оказался на нужном месте, мы используем цикл для чтения первых трех байт из 128 последних. Эти три байта помещаем в переменную Text. Потом закрываем файл с помощью CloseFile(). И с помощью команд IF-ENDIF в зависимости от условий выводится соответствующее сообщение в окне отладки.

Читайте файл справки

Все больше становится хороших программистов особенно на интернет-форумах. Чтобы по-настоящему понять что то, надо учиться. И чтобы выучиться, надо прочесть немало. В действительности не скрывают того факта, что люди, которые читают больше о предмете, как правило, лучше понимают его. Это особенно подходит для языков программирования. Лучший совет, который я могу дать вам, для обучения PureBasic, это чтение справки к PureBasic. Прочитать все о каждой команде. Это звучит очень скучно, но поверьте это приведет к существенному увеличению вашего понимания о PureBasic и даст вам отличное представление, что просто можно с помощью этого великого языка. Надеемся, эта глава даст вам достаточно понимания, чтобы без проблем читать справку и понять, как использовать большинство встроенных команд.

7

Хороший программный стиль

До сих пор я сосредотачивался на объяснении начальных основ программирования. Я думаю, настало время поговорить о стиле или говоря проще о том, как программа должна быть написана. Обучение программированию может быть несложным, но вскоре, когда начнете сами писать свою программу, вы можете столкнутся с плохим стилем написания собственного кода. Аккуратно отформатированный код, не только делает ему более профессиональный вид, но и помогает вам легко читать и понимать его. Далее я опишу способы удобного форматирования своего кода для максимального удобства чтения. Это не является обязательным, скорее предпочтительным. Надеюсь, что после прочтения моих советов, вы организуете свой собственный стиль и будете придерживаться его в дальнейшем. Кроме того я дам вам советы и приемы, которые помогут вам избежать ошибок в коде, и даже если они произойдут, с легкостью с ними справляться. Так же будут некоторые рекомендации по использованию отладчика PureBasic для отслеживания проблем, происходящие в вашем коде.

Почему следует беспокоиться об аккуратности форматирования кода?

Этот вопрос иногда задают начинающие. Один из возможных ответов например, если бы я писал эту книгу без заголовков, подзаголовков, пунктов и слил бы всю книгу в один большой блок, легко ли было бы читать, как сейчас? Аккуратно форматированный код не влияет на производительность программы и делается только для удобной читаемости. Вроде как зачем беспокоиться? Ну, я могу гарантировать, что в будущем вы будете работать над различными исходными кодами, которые не сможете закончить за один день. Вы вернетесь к нему через несколько дней, неделей или месяцев и уже не будете помнить

полностью его. И вот теперь представьте себе если код будет уродливый, плохо читаемый. Вам придет в голову мысль создать лучше заново, чем читать подобную галиматью. И даже если вы создадите за один день свой код, где гарантия, что вы не захотите обновить его через некоторое время?

В вашей программе могут случаться ошибки и при плохо отформатированном программном коде появятся трудности с их обнаружением. Бывают случаи, когда вы сами не можете обнаружить ошибки и попросите помощи у других. Как другие смогут читать ваш сумбурный код? Работа в коллективе является еще одной веской причиной для написания хорошо отформатированного кода. Если вы работаете над проектом в команде, важно чтобы вся команда использовала один стандарт форматирования. Это сделает работу в коллективе максимально эффективной.

Ценность Комментариев

Первое, о чем стоит упомянуть для понимания хорошего стиля программирования, это использование комментариев. До сих пор, комментария не были использованы в моих примерах. Я решил подождать с ними до этой главы.Вы можете использовать их сколько хотите и они никогда не будут вмешиваться в выполнение программы. Комментарии могут дать подробное объяснение вашего кода. При том можно делать пояснения хоть каждой команды или действия . Вот короткий пример:

```
;The procedure Pi() is called to return the value of Pi.
;This calculation is only accurate to six decimal places.

Procedure.f Pi()
  ProcedureReturn 4 * (4 * ATan(1/5) - ATan(1/239))
EndProcedure

CircleDiameter.l = 150
Debug "A circle which has a diameter of " + Str(CircleDiameter) + "mm"
Debug "has a circumference of " + StrF(CircleDiameter * Pi()) + "mm."
```

Здесь я использовал свои комментарии, чтобы описать действия процедуры, а так же упомянуть об ограничении в шесть знаков после запятой для данного расчета.

Комментарии всегда начинаются со знака **;** Вот пример:

```
;This is a comment
```

Комментарии могут появиться абсолютно в любом месте вашего исходного кода, даже на одной строке с другой командой, вот так:

```
Procedure.l AddTogether(a.l, b.l)   ;Add two numbers and return the result
   ProcedureReturn a + b
EndProcedure
```

Вы должны помнить, что точка с запятой в начале определяет где начинается комментарий. Комментарий лучше делать кратким, но достаточно понятным. Если нужно сделать один комментарий на нескольких строчках, надо на каждой строчке перед записью ставить точку с запятой.

Комментарии могут быть использованы для:

1). Добавление лицензий или информации об авторских правах.

2). Объяснять, почему был использован конкретный подход.

3). Уведомления о том, где код может быть улучшен.

4). Уточнение функций сложных процедур.

5). Уточнение внутренности процедур , используя формулы.

Мой формат кодирования

Мой план - определить структуру плохого кода и подвести под стандартный формат кода других языков программирования. Вся эта книга была написана в таком стиле, который я лично считаю легко читаемым и понимаемым.

Переменные, Массивы, Связанные списки и Процедуры

Имена переменных, массивов, связанных списков и процедур я стараюсь писать в формате, который считаю более читабельным. Имя у меня состоит из строчных и прописных букв. Такое имя быстрее находится в тексте кода, да и более примечательно глазу. Пример:

```
NumberOfDays.l = 365

Dim MonthsOfTheYear.s(11)

NewList Days.s()
```

Константы

Для формата констант я использую стандартный C подобный и Windows API стиль, в которых все буквы прописные. Если мне нужно различать отдельные слова, заключенные в имени констант, я использую подчеркивание. Пример:

```
#SPEED_OF_LIGHT = 299792458 ; Meters per second
```

Структуры

Для имен структур я использую тот же формат, что и для констант. То есть буквы все прописные, и если надо отделять слова использую подчеркивание. Пример:

```
Structure USER_SERVICE
  ServiceType.s
  ServiceCode.l
EndStructure
```

Отступы

Отступы дают возможность увидеть структуру кода для процедур, циклов и условных операторов. Отступы очень часто используются для хорошего визуального восприятия кода. Ниже приведены примеры отступов:

```
; Returns a String containing a floating point number that's been rounded up.
; 'Number' = Number to round up and return as a String.
; 'DecimalPlaces' = Number of decimal places to round up to.

Procedure.s StrFRound(Number.f, DecimalPlaces.l)
  Protected R.f
  Protected T.f
  If DecimalPlaces < 0
    DecimalPlaces = 0
  EndIf
  R.f = 0.5 * Pow(10, -1 * DecimalPlaces)
  T.f = Pow(10, DecimalPlaces)
  ProcedureReturn StrF(Int((Number + R) * T) / T, DecimalPlaces)
EndProcedure

Debug StrFRound(3.1415927, 4)
```

В этом примере вы могли видеть, как я с отступом написал код. Это четко определяет код между началом и окончанием команд процедур. А так же вы наверно заметили отступы с условным оператором IF_ENDIF. Также с отступами я применяю написание кода для циклов. Отступы особенно полезны с вложенными циклами:

```
For x = 0 To 2
  For y = 0 To 2
    z.l = 0
    While z =< 10
      Debug x * y + z
      z + 1
    Wend
  Next y
Next x
```

В данном примере ясно видно границы каждого из циклов. Идет четкое разграничение каждого из циклов путем сдвига внутренностей каждого из них. Команда NEXT стоит четко под командой FOR. А команда WEND четко под командой WHILE.

Несколько команд в одной строке

Иногда я пишу несколько команд в одной строке, чтобы сделать некоторые исходные коды визуально меньше размерами . Для отделения команд используется двоеточие:

```
Dim Birds.s(3)
Birds(0)="Sparrow" : Birds(1)="Wren" : Birds(2)="Cuckoo" : Birds(3)="Owl"
```

Некоторые программисты начинают хмуриться на такой род форматирования, поскольку считают что это может помешать читабельности. В чем то они правы, но все же бывают моменты, когда такое написание выглядит довольно читабельным.

Разбитие исходного кода на несколько файлов

Когда я пишу программу, как правило разбиваю его на несколько отдельных файлов исходного кода. Эти файлы являются стандартными PureBasic файлами с расширением .PB или .PBI файл IDE. Эти файлы всегда нетрудно включить в основной код программы с помощью команды XIncludeFile или IncludeFile. Например, если я определяю много процедур, я создаю отдельный исходный файл с именем Procedures.pbi , где будут все мои процедуры. Затем в верхней части моего основного исходного кода я записываю:

```
IncludeFile "Procedures.pbi"
```

Это влечет за собой принятие всех строк кода из файла Procedures.pbi и вставляет их в мой основной исходный код, начиная с этой линии. При использовании нескольких файлов, может случится так, что в них могут встречаться много раз оператор IncludeFile с одним и тем же именем. Будут лишние дубликаты, и чтобы этого не было, надо использовать команду XIncludeFile. Все как и раньше:

```
XIncludeFile "Procedures.pbi"
```

Команда XIncludeFile будет включать код из файла Procedures.pbi только если он не был включен ранее. Я редко использую команду IncludeFile. Я предпочитаю использовать XIncludeFile, так как это сократит количество возникающих ошибок.

Правильное окончание программы

Чтобы завершить программу правильно, вы должны использовать команду END. Это закрывает программу и освобождает всю память, используемую ею. Опциональный выход с использованием кода выхода так же возможен.

Без использования кода выхода:

```
End   ; Immediately ends the program and frees all memory used by it.
```

С использованием кода:

```
End 1   ; Immediately ends the program, frees all memory and returns 1.
```

Золотые правила написания легко читаемого кода

Вот список золотых правил, которым следуют при написании программы. Я придерживаюсь этих правил, даже при написании очень маленьких программ. Если будете следовать этому списку, вы тоже сможете написать хороший и понятный код. Используя стандартный способ форматирования для ваших программ, дает четкую структуру для вашего кода и людям читающим его, легко будет дать вам совет или даже помочь с поиском ошибок.

1). Дайте всем переменным, процедурам, массивам, и т.д. точные, удобопроизносимые, подробные имена.
2). Для группы переменных подключайте массивы или структуры.
3). Процедуры должны выполнить только одну функцию и выполнять ее хорошо.
4). Используйте отступ, чтобы показать структуру кода.
5). Используйте круглые скобки в выражениях, чтобы избежать любого беспорядка в

значениях.
6). Используйте пустые строки, чтобы отделить различные процедуры и другие блоки программы.
7). Попытайтесь не использовать команды ‘Goto’ или ‘Gosub’.
8). Используйте комментарии, чтобы помочь людям и вам лучше понимать код.
9). Пытайтесь плохой код с комментариями, переписать должным образом.
10). Работая в группе, согласуйте стиль форматирования перед стартом, затем придерживайтесь его.

Как свести к минимуму появление ошибок

В этой главе я хочу поговорить о методах, которые помогут вам найти ошибки в коде. Даже самые опытные и преданные делу программисты делают ошибки. Здесь я покажу хорошую практику работы наряду с некоторыми ценными советами и рекомендациями. Это позволит вам быть более бдительными в программировании, сводя к минимуму вероятность каких-либо проблем.

Используйте процедуру-обработчик ошибок

Когда будете писать большие проекты, вы поймете, что будет слишком много тестов для подтверждения значений. Это связано с тем, что много команд возвращает значение. Эта величина почти всегда целое число и всегда выше 0 . Некоторые программисты используют логический оператор IF для этих целей. Я нахожу этот подход правильным только для маленьких программ. Поскольку в большом коде использование множества IF-ENDIF может привести к путанице. Гораздо лучше использовать процедуру обработки ошибок. Это не только делает ваш исходный код более легким для чтения, но так же сообщает пользователю о всех неисправностях в программе. А теперь я представляю вашему вниманию два исходных кода без использования процедуры-обработчика и с использованием:

```
#TEXTFILE = 1
If ReadFile(#TEXTFILE, "TextFile.txt")
  Debug ReadString(#TEXTFILE)
CloseFile(#TEXTFILE)
Else
  MessageRequester("Error", "Could not open the file: 'TextFile.txt'.")
EndIf
```

Здесь тестируется чтение файла *TextFile.txt*. Хотя этот метод хорош для небольших программ, я предпочитаю другой:

```
#TEXTFILE = 1
Procedure HandleError(Result.l, Text.s)
  If Result = 0
    MessageRequester("Error", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure


HandleError(ReadFile(#TEXTFILE,"TextFile.txt"),"Couldn’t open: 'TextFile.txt'.")
```

```
Debug ReadString(#TEXTFILE)
CloseFile(#TEXTFILE)
```

Здесь я использовал процедуру под названием HandleError() для тестирования возвращенного значения любых команд, переданных ей. Первый параметр называется Result, куда посылается возвращенное значение из команд. Второй параметр Text.s это строка,которую вы желаете для отображения ошибки при параметре равном нулю. Процедура получит в качестве параметров два значения, которые передаст ей этот код:

```
ReadFile(#TEXTFILE, "TextFile.txt")
```

Если файл *TextFile.txt* не существует на вашем жестком диске или что то идет не так, команда ReadFile() вернет 0. Это значение передается первым параметром процедуре обработки ошибок. Внутри процедуры этот параметр проверяется, если его значение равно 0, будет вызов сообщения и программа завершится. Это очень удобно, выдавать четкие и краткие сообщения об ошибках для каждой команды, которая вызывает ошибку. Если же команда возвращает значение выше 0, то процедура не предпримет никаких действий. Использование процедуры обработки ошибок становится действительно полезным в крупных программах, где много команд должны быть проверены одна за другой. После того как процедура определена, вы можете использовать ее для тестирования многих команд и каждая из них будет испытана в одной части кода, в отличие от использования способа с оператором IF. Взгляните на этот пример тестирования нескольких команд:

```
HandleError(InitEngine3D(), "InitEngine3D() failed to initialized!")
HandleError(InitSprite(), "InitSprite() failed to initialized!")
HandleError(InitKeyboard(), "InitKeyboard() failed to initialized!")
HandleError(OpenScreen(1024, 768, 32, "Game"), "A screen could not be opened!")
```

А теперь представьте себе сколько кода бы понадобилось для проверки этих четырех команд с использованием оператора IF. Данный пример позволяет легче обнаружить ошибки и неожиданные проблемы. Единственная проблема такого подхода является то, что HandleError() появляется в начале каждой строки, и это некоторые люди считают навязчивым.

Использование команды EnableExplicit

Эта команда является находкой для некоторых людей и помехой для других, и поэтому она не является обязательной для использования. Данная команда позволяет явное определение переменных в течение всей программы. Так что же это значит? В главе 2, я объяснил, что если вы не указали тип суффикса для переменной, то это тип переменной по умолчанию имеет тип LONG. Команда EnableExplicit меняет это правило и требует, чтобы все новые переменные были определены. Позвольте мне показать вам, как эта команда может оказаться полезной. Скажем, например, у меня есть процедура, в которую необходимо передать значение переменной и получить результат вычислений:

```
WeeklyAmount.l = 1024
Procedure CalculateYearlyAmount(Value.l)
  YearlyAmount.l = Value * 52
  ProcedureReturn YearlyAmount
EndProcedure
Debug CalculateYearlyAmount(WeaklyAmount)
```

Вроде как выглядит нормально, и если вы быстро просмотрите этот пример, вы не увидите никаких проблем. Но запустив пример увидите, что процедура CalculateYearlyAmount() возвращает 0. Такого не должно быть, скажете вы! Ведь процедуре передается переменная WeeklyAmount, которая имеет значение 1024. Если присмотреться внимательно на вызов процедуры можно заметить, что на самом деле в качестве параметра передается переменная WeaklyAmount. Обратите внимание, что имя переменной другое, и поэтому она рассматривается в качестве новой переменной. В PureBasic если переменная не была объявлена, создается новая переменная автоматически и в нашем примере такая переменная передается в качестве параметра. Значение такой переменной равно нулю. Отсюда вычисление равно нулю и возвращает процедура тоже нуль.

Автоматическое определение переменных, по мнению некоторых программистов очень подвержено ошибкам, поскольку сказывается человеческий фактор. Это можно изменить с помощью команды EnableExplicit. Если бы мы использовали эту команду в приведенном выше примере, мы получили бы сообщение компилятора о том, что переменная должна быть объявлена.Это означает, что все переменные, используемые после команды EnableExplicit должны быть определены с использованием любой из команд для определения переменных: ‘Define’, ‘Global’, ‘Protected’, ‘Static’ и ‘Shared’ С помощью этой команды выявилась ошибка в написании имени переменной. Пример с использованием команды EnableExplicit. В нем все переменные определены:

```
EnableExplicit
Define WeeklyAmount.l = 1024
Procedure CalculateYearlyAmount(Value.l)
   Protected YearlyAmount.l = Value * 52
   ProcedureReturn YearlyAmount
EndProcedure
Debug CalculateYearlyAmount(WeeklyAmount)
```

С помощью этой команды можно писать код не боясь делать орфографические ошибки, потому что каждый раз, когда появится орфографическая ошибка, компилятор будет рассматривать имя как отдельную переменную. И вам будет предложено определить эту переменную, давая вам шанс исправить переменную, тем самым восстанавливая любые значения, которые могут быть утрачены за счет этих ошибок. В любое время, если вы хотите вернуть правила объявления переменных по умолчанию, вы можете выключить команду EnableExplicit. Это делается командой DisableExplicit.

Определение переменных, используя команду DEFINE

Команда DEFINE может быть использована двумя способами. Первый это определение типа по умолчанию:

```
Define.s
MyString = "Hello"
```

Второй с использованием команды EnableExplicit. Если команда EnableExplicit была использована, все переменные с тех пор в вашей программе, должны быть строго определены. В этом случае, команда DEFINE помогает вам в этом:

```
EnableExplicit
Define MyVar.b = 10
```

Обратите внимание, что при использовании таким образом, вы не должны использовать тип суффиксов в конце команды DEFINE, потому что мы определяем тип каждой переменной с использованием собственного суффикса.

Представление Отладчика PureBasic

PureBasic предоставляет отладчик, который помогает найти ошибки в коде. Это отладчик неоценим, так как он дает вам возможность программного управления потоком, а так же оценивать значения переменных, массивов и связанных списков в любое время в течение выполнения своей программы. Он также предоставляет расширенные функции для программистов, например чтобы изучить и изменить регистры процессора или просмотр значений, хранящихся в смежных адресах памяти. Также можно просмотреть загрузку процессора вашей программы, используя встроенный монитор процессора. Если вы запустите программу и отладчик сталкивается с ошибкой, программа будет остановлена и строка, где произошла ошибка будет отмечаться в IDE (красной линией), и ошибка будет отображаться в журнале и в строке состояния IDE. Когда обнаружена ошибка, вы можете использовать функции программного управления или завершения запущенной программы. Для завершения запущенной программы используют команду Kill Program, которая находится в меню : Debugger/ Kill Program, или связанную с ней кнопку на панели инструментов. Если отладчик отключен, то ошибки не будут пойманы и могут привести к краху программы.

Во время написания программы в IDE, отладчик включен по умолчанию. Это вы можете увидеть, взглянув на кнопочку переключения *Debugger Enable/Disable Toggle* на панели инструментов IDE, см. Рис.ниже. Если эта кнопка отображается нажатой значит отладчик включен. Так же это можно увидеть(Menu:Debugger->Use Debugger). Отладчик также может быть включен в опции компилятора для вашей программы (Menu:Compiler->Compiler Options...->Compiler Options->Enable Debugger). Все эти различные способы переключения состояния отладчика связаны. Если используется одно, другие зеркала его статус. Пакет PureBasic для Windows поставляется с тремя различными видами отладчика. Они могут быть использованы для отладки программы, но некоторые имеют не одинаковую функциональность. Во-первых, встроенный отладчик является наиболее богатым в плане функций и используется по умолчанию, он встраивается непосредственно в IDE. PureBasic IDE работая на некоторых операционных системах, не поддерживает эту встроенную версию, но поддерживает другую автономного отладчика, он также поставляется с установкой. Эта автономная версия имеет почти такой же набор возможностей. Третий отладчик запускается в консольной версии. Первичное использование этой версии является для не-графических сред. Оно основано для операционных систем Linux или удаленно разработанных приложений, работающие с клиентами с помощью SSH протокола. Доступные отладчики могут быть выбраны по умолчанию в настройках IDE, (Menu:File->Preferences->Debugger).

Несмотря на то, отладчик это отличный инструмент для отслеживания проблем, отладка понижает функциональность программы. Вы увидите, что программы, запущенные под отладчиком работают медленнее, чем при запуске их без него. Некоторым программам необходима максимальная скорость, которую можно развить только в окончательных приложениях (EXE). Вы всегда можете отключить отладчик, используя кнопку DisableDebugger, а потом включить при необходимости кнопкой EnableDebugger. Все таки вы должны помнить, что если вы отключите отладчик, будут отключены и все команды DEBUG, это потому, что команда DEBUG является частью отладчика.

Использование отладчика

Функции отладчика могут быть использованы в любое время, пока ваша программа выполняется. Они могут быть доступны из меню отладчика, а также с помощью соответствующих кнопок. Журнал ошибок (Menu:Debugger->Error Log) или монитор процессора (Menu:Debugger->CPU Monitor) всегда доступны. Пока вы используете отладчик, все файлы исходного кода, которые подключены к запущенной программе немедленно включены только для чтения, пока программа не завершилась. Это гарантирует, что используемый в настоящее время код не будет изменен. Обзор отладчика я собираюсь начать с объяснения программы управления функциями, которые он обеспечивает. Эти элементы управления позволяют остановить программу в любой момент времени и изучить значения любой переменной, массива или связанного списка .Состояние любой программы при использовании отладчика будет показан в строке состояния IDE и в журнале ошибок. Обзор отладчика с кнопками на панели инструментов приведен ниже на рис.

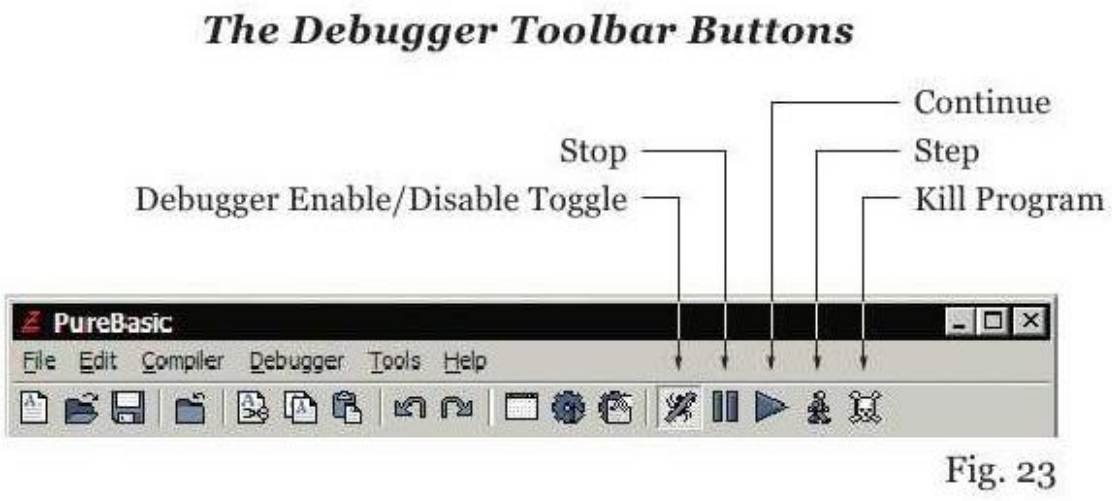


Fig. 23

Чтобы остановить программу в любое время с использованием отладчика, вы можете использовать команду CallDebugger в вашем коде. Или используйте Stop в меню отладчика (Menu:Debugger->Stop) или нажмите кнопку Stop на панели тулбара во время выполнения программы. Вы можете также использовать точки останова(Брекпоинты), чтобы остановить исполнение программы. Для использования брекпоинтов в IDE, поместите курсор на строку, в которой вы хотели бы сделать остановку программы для дальнейшего управления с отладчиком. Затем выберите в меню команду Breakpoint (Menu:Debugger->Breakpoint). Вы заметите, что строка поменяла цвет в IDE. Это визуальная индикация для того, чтобы показать, где определены точки останова. Когда программа дойдет до этого места, она остановится и будут доступны все функции управления отладчика. Далее вы пошагово можете изучать выполнение программы. Вот краткое описание программы контрольных функций:

STOP Эта команда останавливает программу и отображает текущую строку.

CONTINUE Продолжает выполнение программы, пока не встретит другую точку остановки программы

STEP Выполняется одна строка исходного кода, и останавливает выполнение.

Kill Program Полностью завершает работу программы

Вот короткий пример остановки выполнения программы командой CallDebugger

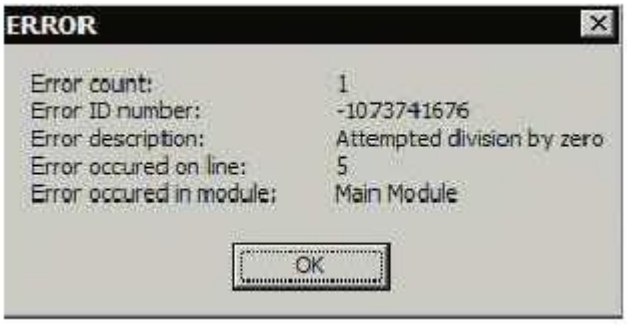
```
CallDebugger
For x.l = 1 To 10
  Debug x
Next x
End
```

После запуска программа была приостановлена командой CallDebugger. Вы можете перемещаться по ней, используя кнопку STEP на панели инструментов. Нажмите на нее десять раз, и вы увидите различные значения X в окне вывода отладки, они будут увеличиваться на единицу при каждом прохождении цикла. Программа будет остановлена и вы можете просматривать значения любой переменной, массива или связанного списка , открыв Variable Viewer (Menu:Debugger->Variable Viewer). Этот небольшой обзор должен был научить вас основам работы с отладчиком. Для более продвинутой информации о том, на что он способен, смотрите справку PureBasic.

OnError Library

Внимание! От версии к версии, меняется набор команд в PureBasic. Так и с библиотекой OnError. Все нижеописанные коды, будут безошибочно работать в версии компилятора 4.00 Но в более поздних версиях некоторые команды, потеряли свой функционал, а некоторые попросту исключили. Но это не значит, что не стоит изучать этот раздел.

Встроенная OnError Library позволяет выполнять проверку ошибок в окончательном исполняемом файле при отключенном отладчике. Обычно вы будете использовать отладчик при разработке для поиска ошибок, но когда вы запланируете скомпилировать исполняемый файл, отключения отладчика принесет вам выигрыш по скорости примерно в шесть раз. Есть много очень сложных команд, которые могут быть использованы с этой библиотекой, но это выходит за рамки этой книги, так что я просто расскажу про наиболее часто используемые и легкие для понимания. Во-первых, я покажу вам, как эффективно поймать ошибку, не используя отладчик. Взгляните на следующий пример:




```
;Set the error handler
OnErrorGoto(?ErrorHandler)
;Trigger a classic 'divide by zero' error.
Null.l = 0
TestVariable.l = 100 / Null
;Handle any system error that occurs
```

```
ErrorHandler:
Text.s + "Error ID number:" + #TAB$ + #TAB$ + Str(GetErrorNumber()) + #CRLF$
Text.s + "Error description:" + #TAB$ + #TAB$ + GetErrorDescription() + #CRLF$
Text.s + "Error occurred on line:" + #TAB$ + Str(GetErrorLineNR()) + #CRLF$
Text.s + "Error occurred in module:" + #TAB$ + GetErrorModuleName() + #CRLF$
MessageRequester("ERROR", Text)
End
```

Здесь я использую команду *OnErrorGoto* с указанием места, куда нужно будет перейти, если произойдет ошибка. Параметр этой команды МЕТКА , куда нужно прыгать. Если вы внимательно посмотрите на параметр, вы заметите, что я приложил вопросительный знак перед именем. Это необходимо команде OnErrorGoto(). Использование знака вопроса, возвращает адрес памяти метки, а не саму метку. В главе 13 даются более подробные объяснения указателей адресов памяти. Подобно использованию команд Gosub или GOTO, мы создаем метку в нашем коде на которую ссылается команда OnErrorGoto и в конец имени ставим двоеточие чтобы дать понять компилятору, что это метка ErrorHandler: . В приведенном выше примере я использовал эти команды:

GetErrorNumber() Эта команда возвращает уникальный номер последней ошибки.

GetErrorDescription() Эта команда возвращает строку с полным описанием произошедшей ошибки.

GetErrorLineNR() Эта команда возвращает номер строки, где произошла ошибка в исходном коде, либо в исходном файле или во включенных файлах. Для правильной работы этой команды, необходимо включить опцию компилятора (Enable OnError Lines Support) перед компиляцией программы.

GetErrorModuleName() Эта команда возвращает строку, говорящую вам адрес исходного файла, в котором находится код с произошедшей ошибкой. Это очень полезно, если вы используете много включенных файлов для исходного кода. Для правильной работы этой команды необходимо включить опцию компилятора (Enable OnError Lines Support) перед компиляцией программы. Это находится в настройках компилятора IDE, (Menu:Compiler->Compiler Options...->Compiler Options->Enable OnError Lines Support).

Чтобы показать работу OnError я специально создал ошибки в коде. В данном случае я написал код с ошибкой (деление на ноль):

```
;Trigger a classic 'divide by zero' error.
Null.l = 0
TestVariable.l = 100 / Null
```

Если отладчик включен, то он показал бы конечно ошибку, но можно и без него поймать ошибку. Мы можем использовать библиотеку OnError для этого. Чтобы запустить пример и увидеть ее нормальную работу, вам необходимо сначала отключить отладчик и в настройках компилятора указать поддержку OnError . Теперь при компиляции программы, она должна поймать ошибку (деление на ноль) и в сообщении вывести подробный отчет. На рисунке ниже примерный вариант того, что должен отобразить компилятор, после компиляции нашего примера.

В последнем примере я использовал команду OnErrorGoto() для перехода на заданную метку в случае ошибки. Эта команда, однако, не позволяет сделать возврат и продолжить программу, даже если вы используете команду Return. Более удобно для таких целей использовать команду OnErrorGosub(). С помощью этой команды вы можете указать метку или процедуру, как обработчик ошибок и сделать возврат в программу для дальнейшего продолжения. Вот пример использования:

```
;Handle any system error that occurs
Procedure ErrorHandler()
  Text.s + "Error ID number:" + #TAB$ + Str(GetErrorNumber()) + #CRLF$
  Text.s + "Error description:" + #TAB$ + GetErrorDescription() + #CRLF$
  Text.s + "Occurred on line:" + #TAB$ + Str(GetErrorLineNR()) + #CRLF$
  Text.s + "Occurred in module:" + #TAB$ + GetErrorModuleName() +
#CRLF$+#CRLF$
  Text.s + "Would you like to continue execution of the program?"
  ReturnValue.l = MessageRequester("ERROR", Text, #PB_MessageRequester_YesNo)
  If ReturnValue = #PB_MessageRequester_No
    End
  EndIf
EndProcedure


;Set the error handler
OnErrorGosub(@ErrorHandler())
;Trigger a classic 'divide by zero' error.
Null.l = 0
TestVariable.l = 100 / Null
;If the program reaches here then the program was resumed
MessageRequester("STATUS", "The program was allowed to continue running.")
End
```

Здесь я продемонстрировал, как возобновить выполнение программы после сообщения об ошибке. То есть то, что вы не можете делать, используя команду OnErrorGoto() . Существует одна проблема с этим подходом. Если вы решите продолжить выполнение программы после произошедшей ошибки, есть риск сделать эту программу нестабильной, что может привести к зависанию и пр. Поэтому, если вы даете вашей программе возможность возобновиться после сообщения об ошибке, вне зависимости от серьезности ошибки, вы должны сообщить пользователю, что лучше перезапустить программу. Возобновление выполнения программы в идеале может быть использовано, для сохранений данных перед перезапуском программы. Скажем, например, вы написали текстовый редактор и Боб использует его для написания письма. Через час после написания, необъяснимая ошибка, и программа выводит сообщение об ошибке. Но Боб не хочет потерять свое письмо из-за ошибки. В этом случае, Боб может быть проинформирован о случившемся и будет возможность возобновить выполнение программы, чтобы сохранить его письмо в файл до перезагрузки программы.

Эти обе команды очень похожи. Есть только одна разница: OnErrorGoto() завершает программу после ошибки, а OnErrorGosub() позволяет продолжить.

Ловим ошибки, создаваемые пользователем

Пока что я показал вам способ ловли и документирования системных ошибок, которые могут возникнуть в вашей программе. Этот способ будет ловить практически любую серьезную ошибку. Однако вы можете использовать в вашем коде обработчик пользовательских ошибок с помощью OnError library . Существует команда SetErrorNumber(), которой требуется всего один параметр для правильной работы. Я обычно использую константы, определяющие пользовательские ошибки:

```
#ERROR_READFILE = 1
#ERROR_WRITEFILE = 2
#FILE_TEXT = 1
;Set the error handler
OnErrorGoto(?ErrorHandler)
If ReadFile(#FILE_TEXT, "Report.txt") = #False
;If the file read fails then fire an error
SetErrorNumber(#ERROR_READFILE)
EndIf
End
;Handle any custom error that occurs
ErrorHandler:

Text.s + "Error ID number:" + #TAB$ + #TAB$ + Str(GetErrorNumber()) + #CRLF$
Select GetErrorNumber()
Case #ERROR_READFILE
Description.s = "The file could not be read."
Case #ERROR_WRITEFILE
Description.s = "The file could not be written."
EndSelect
Text.s + "Error description:" + #TAB$ + #TAB$ + Description + #CRLF$
Text.s + "Error occurred on line:" + #TAB$ + Str(GetErrorLineNR()) + #CRLF$
Text.s + "Error occurred in module:" + #TAB$ + GetErrorModuleName() + #CRLF$
MessageRequester("ERROR", Text)
End
```

В приведенном выше примере, если файл Report.txt нельзя прочитать с помощью команды ReadFile() я подключаю команду SetErrorNumber(), которая вызывает код обработки пользовательской ошибки. Эта ошибка определяется константой #ERROR_READFILE которая имеет значение 1. Когда ошибка срабатывает, вызывается метка обработчика ошибок или процедура. В обработчике ошибок с помощью команды GetErrorNumber() отлавливается параметр ошибки, переданный командой SetErrorNumber(). Текстовые результаты, вы можете выводить те, которые будут уместны в каждом конкретном случае. Как видно из этих примеров, используя OnError library можно сделать неплохой обработчик ошибок, с использованием очень простого синтаксиса.

GUI

В следующем разделе я собираюсь поговорить о графическом пользовательском интерфейсе(GUI) и как он создается в PureBasic. Почти все современные операционные системы имеют GUI, что позволяет пользователю гибко взаимодействовать с программами, которые предпочитают использовать его. Операционная система предоставляет пользователю интерфейс для программы через интерфейс прикладного программирования (или API), благодаря которому программа может сообщить операционной системе, как оформить свой пользовательский интерфейс. Это звучит очень сложно, но это просто и изящно делается в PureBasic с помощью различных библиотек (Window, Menu, а так же Gadgets). Данный раздел начинается с объяснения и демонстрации программ, которые используют консоль, так как их пользовательский интерфейс более понятен. Позднее я перейду к объяснению, как создавать программы с меню и графикой. В последнем разделе я дам вам обзор визуального конструктора PureBasic. С помощью этого инструмента можно разрабатывать интерфейс визуально. После прочтения этого раздела вы поймете как создать графический пользовательский интерфейс для ваших программ а так же как он взаимодействует с пользователем.

8

Создание пользовательского интерфейса

В этой главе я расскажу, как создать графический пользовательский интерфейс (далее я буду называть GUI) для своих программ. PureBasic с помощью своих простых команд делает эту задачу невыразимо легкой. Я поясню код создания меню и графики. Мы так же рассмотрим обработку событий интерфейса вашей программы, например, когда пользователь нажимает на кнопку или выбирает пункт меню. Надеюсь, что после прочтения этой главы, вы будете хорошо осведомлены для создания GUI любой программы, а так же уже сами будете развиваться в этом направлении.

Консольные программы

Мы начнем с простого, а именно с консоли. Консольные программы, как следует из названия, это программы, которые используют консоль для своего пользовательского интерфейса. Консоль представляет собой текстовый интерфейс, который может принимать вводные параметры и отображать с помощью текстовых символов. В некоторых операционных системах консоль может отображать графические символы. Они взяты из набора символов ASCII и попросту заменяют простые символы. Интерфейс консоли обычно используются в программах, где мало требуется вмешательство пользователя. Это обычно утилиты командной строки, которые запускаются из консоли или к примеру CGI программы, которые выполняются в фоновом режиме на веб-серверах и т.д. В основном, консоль используется для вывода информации из программы а так же для того чтобы принять ввод текста от пользователя. Команды, которые создают и работают с консольным интерфейсом описаны в справке PureBasic (Helpfile:ReferenceManual->General Libraries->Console). Эта библиотека предлагает программистам различные команды для печати текста в консоли, ввода информации от пользователя, очистки консоли и даже изменения цвета. Вот пример того, как создать консольную программу в PureBasic:

```
If OpenConsole()
  Print("This is a test console program, press return to exit...")
  Input()
  CloseConsole()
EndIf
End
```

В этом примере я использовал команды OpenConsole() и CloseConsole() , чтобы открыть и закрыть окно консоли, фактически они говорят сами за себя. Второй командой Print со строковым параметром, я указал программе напечатать строку в консоли. Эта команда практически идентична команде PrintN() , которая также будет печатать строку текста, но она будет добавлять конец строки после последнего символа в параметре. То есть следующая команда Print или PrintN будет печатать с новой строки.Это очень похоже на поведение команд WriteString() и WriteStringN(). Третьей командой, используемой в приведенном выше примере, является Input(). Эта команда останавливает выполнение программы до нажатия клавиши ENTER. Кроме того перед нажатием клавиши ENTER можно вывести любой символ(ы) на экран, и это будет параметром возврата команды INPUT(). В моем примере, я использую эту команду исключительно для того чтобы держать окно консоли открытым, чтобы люди смогли прочитать текст, который я вывел в окно консоли. Если в данном примере исключить эту команду, то консоль почти сразу же закрывается после открытия. Использование Input() позволяет держать консоль открытой, для чтения, а потом сообщить пользователю, что для продолжения выполнения программы нужно нажать на клавишу ENTER.

Чтение пользовательского ввода

Иногда в консоли нужно получить то, что ввел пользователь. Это может быть простым числом, либо строкой текста. Принять ввод от пользователя несложно, но вы всегда должны помнить о том, что команда Input() возвращает только строки. Следующий фрагмент кода показывает это, представляя новые консольные команды:

```
If OpenConsole()
 EnableGraphicalConsole(#True)
  Repeat
    ConsoleColor(10, 0)
    PrintN("TIMES TABLES GENERATOR")
    PrintN("")
    ConsoleColor(7, 0)
    PrintN("Please enter a number, then press Return...")
    PrintN("")
    Number.q = ValQ(Input())
    If Number = 0
      ClearConsole()
      Continue
    Else
      Break
   EndIf
 ForEver

PrintN("")
```

```
  For x.l = 1 To 10
    PrintN(Str(x) + " x " + StrQ(Number) + " = " + StrQ(x * Number))
  Next x

  PrintN("")
  Print("Press Return to exit...")
  Input()
  CloseConsole()
EndIf
End
```

Этот пример выглядит довольно сложным, на первый взгляд. Я использовал здесь новую команду EnableGraphicalConsole(). Она включает или выключает отображение графических возможностей консоли, путем передачи этой команде констант #TRUE или #False. Поскольку мы хотим использовать команду ClearConsole(), которая работает только в графическом режиме консоли, мы зададим команде EnableGraphicalConsole() значение #TRUE.

Различия между текстовым и графическим режимами консоли

Использование команды EnableGraphicalConsole() дает возможность переключаться между режимом текста и графическим режимом в консоли. Ниже приведены различия для каждого режима:

Текстовый режим (по умолчанию):

ASCII символы отображаются правильно (ASCII в диапазоне от ‘0’ до ‘31’).

Редиректы(перенаправление) работают правильно (необходимость для программ CGI).

Длинные строки напечатанного текста переносятся на новую строку, при достижении границы окна консоли.

Вы можете читать и записывать данные в консоль, которые могут быть вовсе не обязательно текстовыми.

Графический режим (меняется с помощью EnableGraphicalConsole (# True)):

Текстовые символы ACSII вне диапазона от ‘33’ до ‘126’ отображены в режиме малой графики.

Длинные строки напечатанного текста обрезаются, при достижении границы окна консоли.

ClearConsole() полностью очищает консоль вывода информации.

ConsoleLocate() перемещает курсор в любое заданное место в окне консоли.

Этот список содержит незнакомые команды, но я включил их в этот список, чтобы в будущем понимать в каком режиме они работают.

С помощью команды ConsoleColor() я изменил цвет текста консоли в некоторых местах . Первым параметром является цвет текста, вторым цвет фона текста. Параметры цифры в диапазоне от 0 до 15 которые представляют различные цвета. Чтобы понять какое число с каким цветом ассоциируется см. справку (Helpfile:Reference Manual->General Libraries->Console->ConsoleColor). Кроме того, в этом примере я использовал цикл, для проверки вводимой пользователем информации. Это нужно, если пользователь введет неверное значение. То есть мне нужно, чтобы пользователь вводил число. Даже если пользователь введет число, это все равно будет строковым типом данных, возвращенным командой INPUT(). Для того, чтобы преобразовать его в число, я использую функцию ValQ(), которая преобразует его в тип Quad. Это возвращаемое значение проверяется. Если оно равно 0, то программа очистит консоль и выведет на печать те строки, которые были сразу после запуска и конечно снова перейдет в режим ожидания ввода от пользователя. В том же случае, если пользователь введет число, программа прервет цикл и перейдет к остальной части программы, в которой нарисует таблицу умножения для вводимого числа.

Чтение пользовательского ввода в режиме реального времени

Последний пример продемонстрировал работу команды Input(), которая хороша, если вам необходимо ввести строку, и все же у нее есть единственное ограничение: обязательное нажатие клавиши ENTER. Что делать, если вы хотите получить ввод пользователя в реальном времени, то есть сразу же обнаружить, когда клавиша была нажата, чтобы вызвать действие? Это может быть достигнуто с помощью Inkey() и RawKey() команд. Посмотрите на следующий кусок кода для примера использования их обоих:

```
Procedure DisplayTitle()
  ConsoleColor(10, 0)
  PrintN("KEY CODE FINDER")
  PrintN("")
  ConsoleColor(7, 0)
EndProcedure

Procedure DisplayEscapeText()
  PrintN("")
  ConsoleColor(8, 0)
  PrintN("Press another key or press Escape to exit")
  ConsoleColor(7, 0)
EndProcedure

If OpenConsole()
    EnableGraphicalConsole(#True)
    DisplayTitle()
    PrintN("Press a key...")
  Repeat
    KeyPressed.s = Inkey()
    RawKeyCode.l = RawKey()
    If KeyPressed <> ""
      ClearConsole()
      DisplayTitle()
      PrintN("Key Pressed: " + KeyPressed)
      PrintN("Key Code: " + Str(RawKeyCode))
      DisplayEscapeText()
```

```
    ElseIf RawKeyCode
      ClearConsole()
      DisplayTitle()
      PrintN("Key Pressed: " + "Non-ASCII")
      PrintN("Key Code: " + Str(RawKeyCode))
      DisplayEscapeText()
    Else
     Delay(1)
    EndIf
  Until KeyPressed = #ESC$
    CloseConsole()
EndIf
End
```

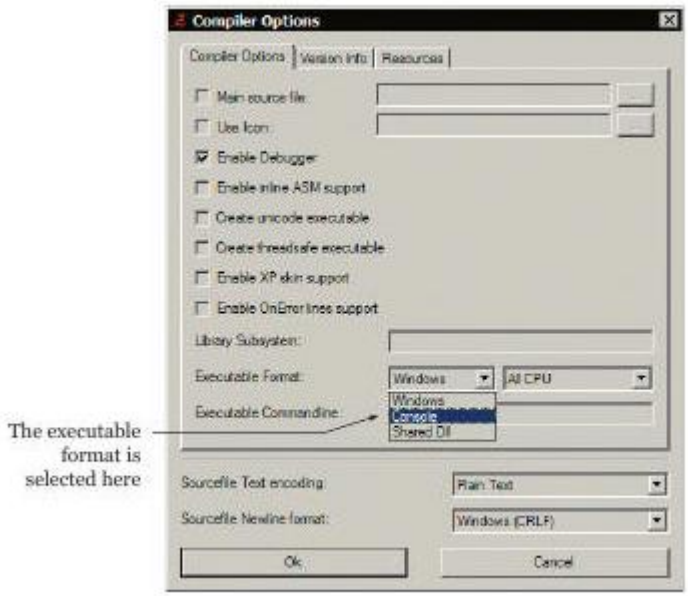
Этот пример похож на прошлый. Мы так же открываем консоль, устанавливаем графический режим, печатаем текст. Основное различие в цикле Repeat, где идет отлов клавиши с помощью команд Inkey() и RawKey(). Первая команда Inkey() возвращает строковой параметр: сам символ при нажатии на него. Так если я нажму клавишу D на клавиатуре, то команда Inkey() вернет строку D. Если я нажму на эту клавишу с нажатой клавишей Shift, то команда вернет d. Так же если я переключу на русский язык, команда будет возвращать уже русские символы Если при нажатии будет не-ASCII символ, то команда вернет пустую строку. Вторая команда RawKey() тоже работает в режиме реального времени и возвращает код нажатой клавиши. Важно понимать, что команда RawKey() не различает русские и английские символы, а так же строчные и прописные. Она считывает ASCII код лишь в одном регистре (он совпадает с прописными английскими клавишами). В приведенном выше примере, можно увидеть, как использовать эти две команды. Это отличный пример ожидания ввода от пользователя в режиме реального времени и соответствующим образом реагировать, когда конкретная клавиша нажата.

Использование команды DELAY()

Вы могли заметить, что в приведенном выше примере я использовал команду Delay() с помощью которой я определил программе задержку на 1 миллисекунду. Хотя это выглядит немного странным и ненужным, команда Delay() разумна для данного кода. Поскольку вы будете использовать многозадачность операционной системы для запуска программ, практика использования команды задержки позволит не монополизировать(только для себя) процессорное время. Использование задержки дает возможность другим программам использовать процессор в это время. Даже задержка 1 миллисекунда освобождает ресурсы процессора для других программ. Эта команда особенно полезна при использовании циклов. Данная команда может использоваться даже для заморозки своей программы на нужное время.

Правильная компиляция консольных программ

Запуск консольной программы в IDE, будет строить программы, которые выглядят и действуют как реальные консольные программы, но не будут консольными программами в прямом смысле, пока вы их не скомпилируете правильно в формат консоли.



И это все есть в программировании интерфейсов консольных программ на самом деле. Они не должны быть использованы для сложных интерфейсов (хотя в прошлом, и игры были написаны с использованием графического режима командной строки). Эти интерфейсы используется в настоящее время для отображения простых утилит из командной строки.

Создание обычного пользовательского интерфейса

В этом разделе я собираюсь показать вам, как создавать привычные вам пользовательские интерфейсы. PureBasic использует API операционной системы для привлечения компонентов интерфейса. В качестве примера такого интерфейса, посмотрите на PureBasic IDE. Среда IDE написана полностью на языке PureBasic, который в свою очередь, использует API операционной системы. Все эти разговоры об API операционной системы могут вас обескуражить , заставит вас думать, что все очень сложно, но не паникуйте. PureBasic всю сложность превратил в набор понятных команд и функций. По набору этих функций PureBasic может соперничать с любым другим языком программирования. В среднем программы, написанные на PureBasic оказываются быстрее и имеют меньшие размеры, чем у многих языков программирования. Это было доказано много раз через небольшие соревнования, созданные пользователями на официальных форумах PureBasic. Итак, давайте продолжим и создадим наш первый пользовательский интерфейс.

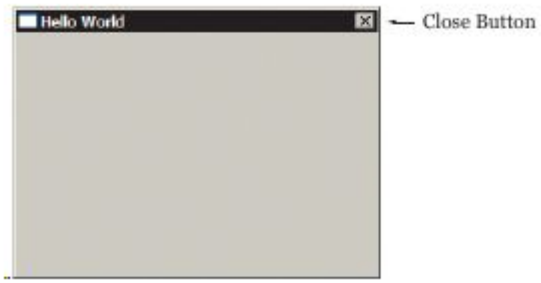
ПРИВЕТ МИР!

Как это принято в мире программирования, первый пример интерфейса команд на любом языке программирования обычно программа Hello World(ПРИВЕТ МИР!) Он состоит из простого окна, приветствующего пользователя и предоставляющего возможность выхода из программы. И так:

```
#WINDOW_MAIN = 1
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
```

```
If OpenWindow(#WINDOW_MAIN, 0, 0, 300, 200, "Hello World", #FLAGS)
  Repeat
    Event.l = WaitWindowEvent()
 Until Event = #PB_Event_CloseWindow
EndIf
End
```

Если вы запустите этот кусок кода, вы увидите окно, похожее на то, что ниже на рисунке. Чтобы закрыть это окно нажмите кнопку Закрыть в верхнем правом углу. Этот пример состоит из пустого окна и это станет началом пользовательских интерфейсов в PureBasic. В приведенном выше примере, я использую функцию OpenWindow() для открытия окна и определения ее атрибутов. Если открыть справку и найти функцию OpenWindow() (Helpfile:Reference Manual->General Libraries->Window->OpenWindow), можно четко увидеть синтаксис этой функции. В данном примере мы использовали флаги #PB_Window_SystemMenu и #PB_Window_ScreenCentered, которые определяют, что окна должны иметь функционирующую систему меню, в комплекте с кнопкой Закрыть и быть в центре экрана, после запуска. При использовании нескольких флагов, нужно использовать битовый оператор OR (|). Этот оператор объединяет все значения (более полное объяснение см. главу 3). После команды OpenWindow() я использовал цикл Repeat в качестве основного цикла для обработки любых событий.



Почему параметры позиции окна нулевые? При открытии окна с помощью OpenWindow() параметры команд определения координат игнорируются, поскольку используется флаг #PB_Window_ScreenCentered. Как и флаг #PB_Window_WindowCentered (для дочернего окна)они имеют больший приоритет. И поскольку это так, вы можете смело использовать параметры #PB_Ignore для этих двух позиций.

Основной цикл

Основной цикл в программе необходим, чтобы программа непрерывно работала. Это позволяет ей иметь неизменный вид при перерисовке интерфейса (если он двигался на рабочем столе) и обрабатывать любые события, генерируемые интерфейсом. К счастью PureBasic обрабатывает все автоматически, при правильном использовании основного цикла. В программе ПРИВЕТ МИР я использовал цикл Repeat - Until в качестве моего главного цикла , это дает мне возможность держать цикл, пока определенное условие соблюдено, в нашем случае, пока программа не обнаружила тип события # PB_Event_CloseWindow, которое срабатывает при нажатии на кнопку Закрыть. После получения этого события, цикл завершает свою работу, таким образом идет прекращение работы программы.

Понимание событий

Программа, написанная на любом из языков программирования, будет использовать примерно одинаковый способ обработки событий. Все основное действие происходит в главном цикле, и лишь в зависимости от событий, программа может перейти к нужной нам части кода. Какого рода события существуют? События могут быть вызваны многими действиями в рамках вашей программы, например событие нажатия на гаджет, или нажатие кнопки на клавиатуре, выбор пункта из меню, или же просто закрытие программы. Самое главное, надо уметь привязать к нужному событию нужный кусок кода. Но для этого надо его идентифицировать. В Purebasic для этого есть команды: WaitWindowEvent() и WindowEvent().

Что такое гаджет?

В PureBasic гаджет представляет собой графический объект пользовательского интерфейса, который нужен для интерактивности вашей программы. Microsoft Windows называет их Controls, а некоторых дистрибутивы Linux называет их Widgets. Гаджеты представляют собой кнопки, поля ввода, ползунки, фреймы, прогресс бары и т.д. Все различные интерактивные компоненты, которые составляют интерфейс.

Эти две команды практически идентичны, поскольку обе они возвращают идентификатор в виде длинного номера, когда происходит событие. Разница между ними в том, что команда WaitWindowEvent() циклично приостанавливает на короткие промежутки времени вашу программу до обнаружения события, и лишь когда оно происходит, возвращает его идентификатор и позволяет программе продолжиться как обычно. (Это очень удобно для использования в GUI, так как позволяет программе использовать мало ресурсов процессора.). Команда WindowEvent() не останавливает программу, то есть она на всю мощь работает, отлавливая события, при этом съедая невыразимо много ресурсов процессора. Эта команда очень редко используется в пользовательских интерфейсах так как это уменьшает вычислительные мощности компьютера по отношению к другим запущенным процессам в системе. WindowEvent() может использоваться, когда необходимо сохранить работу основного цикла на всю мощь, например, при показе динамической графики в вашей программе, при ее постоянной перерисовке и т.д. Если же все таки вы используете WindowEvent(), то лучше это делать в паре с командой Delay (1). Если нет событий осуществляется задержка 1 миллисекунда, для освобождения ресурсов процессора.

В программе "Hello World" я использовал команду WaitWindowEvent() в цикле Repeat для обнаружения вызова события. После вызова события WaitWindowEvent(), возвращает идентификатор, который передается на хранение в переменную с типом Long и названием **Event.l** . Цикл проходя по коду, сравнивает возвращенный идентификатор со всеми событиями имеющимися в коде, и если он равен числу в константе # PB_Event_CloseWindow, которая содержит идентификатор закрытия окна, то окно закрывается. Поскольку ниже цикла нет кода, программа завершает свою работу.

События происходят все время

Хотя мы не назначили никаких других действий программе, это не означает, что не происходит никаких других событий. Если мы возьмем ту же программу "Hello World" и добавим еще одну строку, содержащую команду Debug Event,мы увидим, как программа будет отображать идентификаторы событий в окне вывода отладки во время выполнения.

Запустите следующий кусок кода, а затем переместите мышку или щелкните внутри и вокруг окна. Вы увидите, что происходит много событий.

```
#WINDOW_MAIN = 1
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, 300, 200, "Hello World", #FLAGS)
  Repeat
    Event.l = WaitWindowEvent()
    Debug Event
 Until Event = #PB_Event_CloseWindow
EndIf
End
```

Команда WaitWindowEvent() возвращает все идентификаторы для всех вызванных событий, даже если они не будут задействованы в вашей программе. То есть совершенно нормально будет немного замедлить работу программы для освобождения ресурсов процессора, с помощью команды WaitWindowEvent().

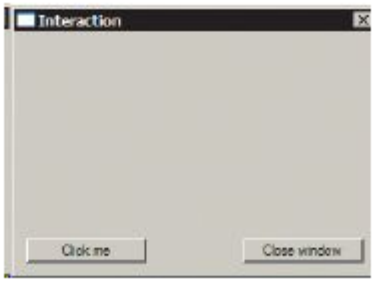
Добавление гаджетов

До сих пор мы видели простое окно, которое отображает Hello Word в заголовке. В следующем разделе я покажу вам, как вы можете добавить гаджет к окну вашей программы, чтобы повысить его функциональность и интерактивность. В следующий кусок кода я добавил две кнопки и продемонстрировал, как выявлять события от нажатия на них.

```
Enumeration
  #WIN_MAIN
  #BUTTON_INTERACT
  #BUTTON_CLOSE
EndEnumeration

Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Interaction", #FLAGS)
  If CreateGadgetList(WindowID(#WIN_MAIN))
    ButtonGadget(#BUTTON_INTERACT, 10, 170, 100, 20, "Click me")
    ButtonGadget(#BUTTON_CLOSE, 190, 170, 100, 20, "Close window")
    Repeat
      Event.l = WaitWindowEvent()
      Select Event
        Case #PB_Event_Gadget
          Select EventGadget()
            Case #BUTTON_INTERACT
              Debug "The button was pressed."
            Case #BUTTON_CLOSE
              Quit = #True
        EndSelect
      EndSelect
  Until Event = #PB_Event_CloseWindow Or Quit = #True
  EndIf
```

```
EndIf
End
```

Чтение этого кода не должно вызвать у вас затруднений, так что я просто поясню те области, которые будут для вас в новинку. Рисунок ниже показывает вам, как это окно выглядит.



В этом примере после команды создания основного окна, я использовал команду CreateGadgetList(). Это необходимо, чтобы ваше окно могло размещать гаджеты. Эта команда принимает только один параметр, который является идентификатором ОС: ID окна. Я использую команду WINDOWID() для его получения:

```
...
CreateGadgetList(WindowID(#WIN_MAIN))
...
```

ОС идентификаторы были рассмотрены в главе 7 (PB числа и ОС идентификаторы). После создания списка гаджетов, можно размещать гаджеты на окно. Я создал две кнопки, при помощи команды ButtonGadget(). Если открыть файл помощи и посмотреть на синтаксис этой команды (Helpfile:Reference Manual->General Libraries->Gadget->ButtonGadget) вы сможете увидеть все возможные параметры относящиеся к этой команде.

С помощью параметров к команде ButtonGadget() я разместил кнопки по нижнему краю окна. Как и в первом примере с пустым окном, мы переменной Event присваиваем идентификаторы, возвращаемые функцией WaitWindowEvent(), а командой SELECT мы тестируем какому числу равен идентификатор:

```
Select Event
Case #PB_Event_Gadget
...
EndSelect
```

Вы заметили, что первый тест с помощью SELECT указывает на константу #PB_Event_Gadget, которая является идентификатором события гаджета. Другими словами, проверяется равенство события константе #PB_Event_Gadget. Далее, когда мы знаем, что событие произошло от гаджета, мы должны узнать от какого именно. Для этого применяется функция EventGadget(), которая возвращает номер гаджета задействованного в событии. И я тестировал его с помощью другого SELECT.

```
...
Select EventGadget()
  Case #BUTTON_INTERACT
```

```
    Debug "The button was pressed."
  Case #BUTTON_CLOSE
    Quit = #True
EndSelect
...
```

Здесь у меня два тестируемых значения, одно #BUTTON_INTERACT другое #BUTTON_CLOSE. Если возвращаемое значение из функции EventGadget() равно #BUTTON_INTERACT, то была нажата кнопка с названием Click me. А значит по условию будет выполняться код: **Debug "The button was pressed."** Если возвращаемое значение равно #BUTTON_CLOSE, то значит что была нажата кнопка Close window и переменной Quit будет назначено значение #True, а значит произойдет выход из цикла и завершение программы. Именно поэтому использование констант в блоке Enumeration очень полезно. После определения числовых констант для всех гаджетов в начале исходного кода, можно сослаться на каждый из них, вместо номера, что делает исходный код несравненно более четким и читаемым.

Принимаем ввода текста

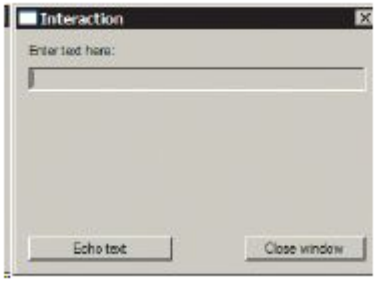
Давайте изменим последний пример и добавим StringGadget, для того чтобы мы могли ввести некоторый текст, который можно использовать в программе. StringGadget очень удобен, поскольку он обеспечивает простой способ программе принимать строку текста. Следующий пример покажет вам, как можно получить значение этого гаджета т.е. строку. и использовать ее в вашей программе. Давайте создадим один StringGadget практически на полную ширину окна:

```
Enumeration
   #WIN_MAIN
   #TEXT_INPUT
   #STRING_INPUT
   #BUTTON_INTERACT
   #BUTTON_CLOSE
EndEnumeration


Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Interaction", #FLAGS)
   If CreateGadgetList(WindowID(#WIN_MAIN))
      TextGadget(#TEXT_INPUT, 10, 10, 280, 20, "Enter text here:")
      StringGadget(#STRING_INPUT, 10, 30, 280, 20, "")
      ButtonGadget(#BUTTON_INTERACT, 10, 170, 120, 20, "Echo text")
      ButtonGadget(#BUTTON_CLOSE, 190, 170, 100, 20, "Close window")
      SetActiveGadget(#STRING_INPUT)
      Repeat
         Event.l = WaitWindowEvent()
         Select Event
          Case #PB_Event_Gadget
            Select EventGadget()
               Case #BUTTON_INTERACT
                  Debug GetGadgetText(#STRING_INPUT)
```

```
            Case #BUTTON_CLOSE
              Quit = #True
          EndSelect
        EndSelect
      Until Event = #PB_Event_CloseWindow Or Quit = #True
    EndIf
EndIf
End
```

В этом обновленном коде, я добавил еще пару констант в блок Enumeration, в результате я могу легко добавить еще два гаджета . Один из гаджетов, который я добавил, является **TextGadget**. Он выводит не изменяемую для пользователя строку текста в окно. Я его использовал для отображения строки: **Enter text here:** Вторым гаджетом является **StringGadget** Он создает поле ввода, которое позволяет вам ввести свой собственный текст.

Если вы откроете справку PureBasic для StringGadget() (Helpfile:Reference Manual->GeneralLibraries->Gadget->StringGadget) то можете увидеть параметры этого гаджета. С помощью параметров, я поставил **StringGadget** в верхней части окна, чуть ниже **TextGadget**. В качестве последнего параметра я принял пустую строку, которая выражается как набор двойных кавычек, между ними ничего нет. Это тоже строка, которая не имеет содержания. Мне захотелось, чтобы изначально в нашем поле ввода ничего не было.



Как только эта программа будет запущена, вы заметите, что текстовый курсор мигает в StringGadget. Это не по умолчанию, как правило, вам придется нажать на гаджете, для того чтобы появился курсор. Но в моей программе я установил так, что гаджет станет активным при запуске с помощью команды SetActiveGadget(). Эта команда имеет всего один параметр: гаджет, который вы хотите сделать активным. Я сделал это для удобства пользователя. Для проверки интерактивности этой программы, введите текст в StringGadget, а затем нажмите на кнопку **Echo text**. Теперь вы увидите, что текст появился в окне вывода отладки. Этот текст может изменяться по вашему желанию и каждый раз будет выводиться в окно вывода отладки, то что вы напечатали. Считывает текст из StringGadget функция GetGadgetText, у которой всего один параметр: гаджет, из которого надо считать текстовую информацию. Эта функция может быть использована со многими гаджетами в PureBasic, но со StringGadget она просто возвращает строку.

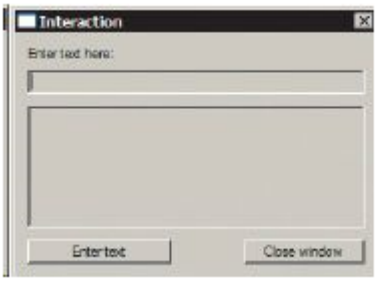
Отображение текста

Давайте расширим эту программу еще дальше и вместо того, чтобы выводить текст из StringGadget в окно вывода отладки, давайте добавим ListViewGadget для отображения текста внутри нашей программы, а не в процессе отладки. В следующем примере я добавил ListViewGadget для отображения ввода текста, с невозможностью редактировать строки. ListViewGadget является отличным способом для отображения списка строк, поскольку каждая строка имеет свою собственную линию в рамках гаджета. PureBasic предлагает множество команд для работы с ListViewGadget, начиная с добавления и удаления, далее для подсчета и сортировки элементов, содержащихся в этом типе гаджета. В следующем небольшом примере, я использовал только команду AddGadgetItem() , чтобы добавлять новые строки:

```
Enumeration
   #WIN_MAIN
   #TEXT_INPUT
   #STRING_INPUT
   #LIST_INPUT
   #BUTTON_INTERACT
   #BUTTON_CLOSE
EndEnumeration

Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Interaction", #FLAGS)
   If CreateGadgetList(WindowID(#WIN_MAIN))
      TextGadget(#TEXT_INPUT, 10, 10, 280, 20, "Enter text here:")
      StringGadget(#STRING_INPUT, 10, 30, 280, 20, "")
      ListViewGadget(#LIST_INPUT, 10, 60, 280, 100)
      ButtonGadget(#BUTTON_INTERACT, 10, 170, 120, 20, "Enter text")
      ButtonGadget(#BUTTON_CLOSE, 190, 170, 100, 20, "Close window")
      SetActiveGadget(#STRING_INPUT)

      Repeat
         Event.l = WaitWindowEvent()
         Select Event
            Case #PB_Event_Gadget
               Select EventGadget()
                  Case #BUTTON_INTERACT
                     AddGadgetItem(#LIST_INPUT, -1, GetGadgetText(#STRING_INPUT))
                     SetGadgetText(#STRING_INPUT, "")
                     SetActiveGadget(#STRING_INPUT)
                  Case #BUTTON_CLOSE
                     Quit = #True
               EndSelect
         EndSelect
      Until Event = #PB_Event_CloseWindow Or Quit = #True
   EndIf
EndIf
End
```

Этот пример очень похож на предыдущий. На самом деле единственное реальное различие является включение ListViewGadget. Чтение справки PureBasic для ListViewGadget() (HelpFile: ReferenceManual->General Libraries->Gadget->ListViewGadget)

даст вам хороший обзор этого гаджета. Вы узнаете о синтаксисе и параметрах этой команды. Я использовал параметры для этого гаджета таким образом, чтобы этот гаджет оказался в центре главного окна, над кнопками, но под StringGadget.



После запуска программы, введите какой-нибудь текст в StringGadget и нажмите на кнопку **Enter text**. Это действие добавит в наш ListViewGadget() строку, которую мы ввели. После этого программа очищает StringGadget и он готов к дальнейшему вводу. Все это достигается благодаря тому, что я добавил в наш код добавление в StringGadget пустой строки и фокусировку гаджета, после нажатия на кнопку **#BUTTON_INTERACT**

```
...
Case #BUTTON_INTERACT
AddGadgetItem(#LIST_INPUT, -1, GetGadgetText(#STRING_INPUT))
SetGadgetText(#STRING_INPUT, "")
SetActiveGadget(#STRING_INPUT)
...
```

В первой строке после команды CASE идет команда ADDGadgetItem. Если вы откроете страницу в справке для AddGadgetItem() (Helpfile:Reference Manual->General Libraries->Gadget->AddGadgetItem), вы увидите параметры, которые использует эта команда. Первым параметром является номер гаджета, в который вы хотели бы добавить элемент, в нашем случае это константа #LIST_INPUT, которая содержит номер нашего гаджета. Второй параметр определяет позицию в списке (существующего или несуществующего значения). Когда будете добавлять, вы должны помнить, что индексы начинаются с нуля. В приведенном выше примере, я хочу, чтобы этот текст был добавлен в конец списка независимо от того, какой индекс, поэтому я задал параметр -1. Такой способ помогает не возиться с индексами, просто добавляет пункт в конец текущего списка каким бы большим он ни был. Третий параметр является фактической строкой, которую вы хотите добавить в гаджет. Я использовал команду GetGadgetText() возвращающую значение из StringGadget и передающую его в качестве параметра. AddGadgetItem() может иметь больше параметров, чем я использовал здесь, но другие не являются обязательными. Второй строкой после команды Case является команда SetGadgetText(), которая устанавливает значение StringGadget в пустую строку. Это так же делается для удобства пользователя. Последней строкой идет команда SetActiveGadget(), которая делает StringGadget активным.

Чему мы научились?

Надеюсь, последние несколько примеров были хорошим введением в GUI и обеспечивает хорошую отправную точку для дальнейшего обучения. В этой книге я не могу дать вам примеры работы каждого гаджета. Все, что могу сделать, это дать вам основные так

сказать строительные блоки интерфейса, и это будет вам опорой для дальнейших ваших разработок. Принцип GUI одинаков. Сначала необходимо открыть окно с помощью команды OpenWindow() . Далее, по желанию нужно разместить гаджеты в этом окне. Но перед этим надо создать список гаджетов с помощью команды CreateGadgetList(). И конечно же создать цикл, в котором будут обрабатываться все события программы. Не забываем установить в цикл команду WaitWindowEvent() или WindowEvent() для обнаружения событий.

Внимание: в новых версиях компилятора, начиная с версии 4.30 команда **CreateGadgetList()** встроена в компиляцию GUI приложений, поэтому для размещения гаджетов ее использовать нет необходимости!

Изучаем гаджеты в справке.

Если вы поняли, что было написано до сих пор, вам не трудно будет создать свой собственный интерфейс. Для того чтобы больше узнать о гаджетах, нужно обратиться в справку PureBasic (Helpfile:Reference Manual->General Libraries->Gadget). Я рекомендую прочесть этот раздел, поскольку он даст знания о том как взаимодействуют различные команды с различными гаджетами. Возьмем, к примеру команду SetGadgetText(). Она может быть использована с различными гаджетами. Если вы посмотрите в справку для этой функции (Helpfile:Reference Manual->General Libraries->Gadget->SetGadgetText), там есть список всех гаджетов, с которыми эта команда взаимодействует. Чтобы охватить все гаджеты и команды, которые работают с ними, нужно написать еще одну книгу, поэтому пока она не написана, мой вам совет: ознакомиться с разделом GADGET в справке. С другой стороны, если вам нужно поверхностно узнать о каком то гаджете, вы можете обратиться к приложению B этой книги. Там есть полный список всех гаджетов и краткое описание каждого из них.

Добавление меню

Добавить меню в интерфейс так же просто, как и добавление гаджетов. Когда меню создано, события для него обрабатываются так же как и у гаджетов в главном цикле.Все пункты меню определяются по номерам, как и гаджеты, и так же тестируются для дальнейших действий, связанных с ним. Вот небольшой пример:

```
Enumeration
  #WINDOW_MAIN
  #MENU_MAIN
  #MENU_QUIT
  #MENU_ABOUT
EndEnumeration


Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, 300, 200, "Menu Example", #FLAGS)
   If CreateMenu(#MENU_MAIN, WindowID(#WINDOW_MAIN))
      MenuTitle("File")
      MenuItem(#MENU_QUIT, "Quit")
      MenuTitle("Help")
      MenuItem(#MENU_ABOUT, "About...")
      Repeat
```

```
        Event.l = WaitWindowEvent()
        Select Event
          Case #PB_Event_Menu
            Select EventMenu()
              Case #MENU_QUIT
                Quit = #True
              Case #MENU_ABOUT
                MessageRequester("About", "This is where you describe your program.")
            EndSelect
        EndSelect
      Until Event = #PB_Event_CloseWindow Or Quit = #True
   EndIf
EndIf
End
```

Этот маленький пример показывает создание меню в пустом окне. Есть два названия меню и каждое название содержит один пункт меню. Если вы посмотрите на приведенном выше примере, можно увидеть, как создать такое меню. Для начала, вам необходимо открыть окно. После того как окно было создано, мы создаем основу меню с помощью команды CreateMenu() . Эта команда имеет два параметра, первый номер PB этого меню, а второй идентификатор окна ОС, к которому оно должно быть привязано. Это очень похоже на синтаксис команды CreateGadgetList(), там я также использовал команду WINDOWID() чтобы получить идентификатор ОС для нашего окна. После того как основа меню была успешно создана, можно заполнить ее заголовками меню. Для этого я использую команду MenuTitle(). Эта команда создает заголовок меню, которое отображается в верхней части окна (или в случае с Mac OS X, в верхней части экрана). Для этой команды предусмотрен всего один параметр-название меню. После того, как заголовок определен, можно создать пункты меню. Пункты меню создаются с помощью команды MenuItem(). Эта команда имеет два параметра, первый номер PB, а второй его фактическое название. В моем примере, я определил по одному пункту меню для каждого заголовка. Для заголовка FILE пункт Quit, а для заголовка HELP пункт ABOUT.

Обнаружение событий меню

После того как было создано меню, включая все меню и названия, мы можем отлавливать события этого меню с помощью команды **#PB_Event_Menu**. Все в точности так же как и с гаджетами, разница лишь в том что в случае с гаджетами мы использовали **#PB_Event_Gadget.** Если событие отлавливаемое с помощью WaitWindowEvent() равно #PB_Event_Menu то это событие из пункта меню.

```
...
Select Event
  Case #PB_Event_Menu
    Select EventMenu()
      Case #MENU_QUIT
        Quit = #True
      Case #MENU_ABOUT
        MessageRequester("About", "This is where you describe your program.")
    EndSelect
EndSelect
```

...Для определения, какой именно пункт меню был вызван, мы должны использовать команду EventMenu(). Эта команда возвращает номер меню, который был вызван данным

событием. И так сначала мы тестируем: равно ли событие **#PB_Event_Menu,** и если равно, тогда тестируем какой именно пункт меню был вызван с помощью команды EventMenu() . Все тесты в моей программе были сделаны с помощью оператора SELECT.

Подпункты меню

До сих пор я продемонстрировал, как создать стандартное меню, но вы также можете создать так называемое SUBMENU(подпункт меню). Данное меню реализовано в меню ПУСК Microsoft Windows. Получается так сказать древовидная структура. События для этих меню отслеживаются так же как и для основного меню. Вот пример:

```
Enumeration
   #WINDOW_MAIN
   #MENU_MAIN
   #MENU_CHILD
   #MENU_QUIT
   #MENU_ABOUT
EndEnumeration

Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, 300, 200, "Menu Example", #FLAGS)
  If CreateMenu(#MENU_MAIN, WindowID(#WINDOW_MAIN))
    MenuTitle("File")
    OpenSubMenu("Parent Menu Item")
      MenuItem(#MENU_CHILD, "Sub Menu Item")
    CloseSubMenu()
    MenuBar()
    MenuItem(#MENU_QUIT, "Quit")
    MenuTitle("Help")
    MenuItem(#MENU_ABOUT, "About...")
    Repeat
      Event.l = WaitWindowEvent()
      Select Event
        Case #PB_Event_Menu
          Select EventMenu()
            Case #MENU_CHILD
              Debug "Sub menu item selected."
            Case #MENU_QUIT
              Quit = #True
            Case #MENU_ABOUT
              MessageRequester("About", "This is where you describe your program.")
          EndSelect
      EndSelect
    Until Event = #PB_Event_CloseWindow Or Quit = #True
  EndIf
EndIf
End
```

Подпункт меню создается с помощью команды OpenSubMenu() после команды MenuTitle(). Это объясняется тем, подпункт меню, связан с заголовком меню так же, как и любой другой пункт меню. Команда OpenSubMenu() принимает всего один параметр,

которым является строка. Она отображается как и пункт меню, но со стрелкой(сноской). После того, как мы определили команду OpenSubMenu(), мы можем добавлять пункты с помощью MenuItem() ниже этой команды, но выше команды CloseSubMenu(), которая закрывает блок SUBMENU. Дальнейшее добавление пунктов вне блока, будут обыкновенными без сносок. Кроме того если нам надо в SUBMENU создать еще одно SUBMENU, то это реализуется точно так же: в блок записывается еще один дополнительный блок со своими названиями.

Отделение пунктов меню

Если вы внимательнее посмотрите на приведенный выше пример, то увидите, что я добавил еще одну новую команду. Эта команда есть не что иное, как графический бар или разделитель меню. Название этой команды MenuBar(). Он не принимает никаких параметров и используется там, где вы хотите разместить разделитель в текущем меню. Обычно он разделяет пункт "Quit" от остальных пунктов меню, что я и сделал в прошлом примере.

Сочетание гаджетов и меню в одной программе

Включение меню и гаджетов в одной программе очень просто. Во-первых, вы открываете окно и создайте меню. Затем вы создаете список гаджетов. Дальше связываете события гаджетов и меню с нужными действиями. Вот пример кода:

```
Enumeration
 #WINDOW_MAIN
 #MENU_MAIN
 #MENU_QUIT
 #MENU_ABOUT
 #TEXT_INPUT
 #STRING_INPUT
 #LIST_INPUT
 #BUTTON_INTERACT
 #BUTTON_CLOSE
EndEnumeration
Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, 300, 222, "Interaction", #FLAGS)
  If CreateMenu(#MENU_MAIN, WindowID(#WINDOW_MAIN))
    MenuTitle("File")
    MenuItem(#MENU_QUIT, "Quit")
    MenuTitle("Help")
    MenuItem(#MENU_ABOUT, "About...")
     If CreateGadgetList(WindowID(#WINDOW_MAIN))
       TextGadget(#TEXT_INPUT, 10, 10, 280, 20, "Enter text here:")
       StringGadget(#STRING_INPUT, 10, 30, 280, 20, "")
       ListViewGadget(#LIST_INPUT, 10, 60, 280, 100)
       ButtonGadget(#BUTTON_INTERACT, 10, 170, 120, 20, "Enter text")
       ButtonGadget(#BUTTON_CLOSE, 190, 170, 100, 20, "Close window")
       SetActiveGadget(#STRING_INPUT)
       Repeat
         Event.l = WaitWindowEvent()
         Select Event
```

```
          Case #PB_Event_Menu
            Select EventMenu()
              Case #MENU_QUIT
                Quit = #True
              Case #MENU_ABOUT
                MessageRequester("About", "This is your program description.")
            EndSelect
          Case #PB_Event_Gadget
            Select EventGadget()
              Case #BUTTON_INTERACT
                AddGadgetItem(#LIST_INPUT, -1, GetGadgetText(#STRING_INPUT))
                SetGadgetText(#STRING_INPUT, "")
                SetActiveGadget(#STRING_INPUT)
              Case #BUTTON_CLOSE
                Quit = #True
            EndSelect
        EndSelect
      Until Event = #PB_Event_CloseWindow Or Quit = #True
    EndIf
  EndIf
EndIf
End
```

Для обработки событий пунктов меню или гаджетов, желательно использовать константы событий, а тестировать все с помощью условных операторов:

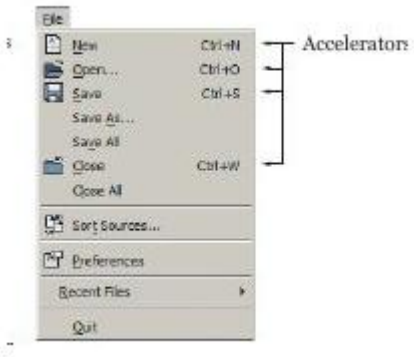
```
...
Select Event
  Case #PB_Event_Menu
    Select EventMenu()
      ...
    ;Menu events are handled here
      ...
    EndSelect
      ...
  Case #PB_Event_Gadget
    Select EventGadget()
      ...
  ;Gadget events are handled here
      ...
    EndSelect
EndSelect
...
```

Если событие вызывается из пункта меню, то есть событие будет равно #PB_Event_Menu, то далее определяется точный пункт меню с помощью команды EventMenu(). Если событие срабатывает от гаджета, то есть событие равно #PB_Event_Gadget, то точный гаджет устанавливается с помощью команды EventGadget(). Это позволяет обрабатывать все сразу и меню и гаджеты в одной программе, с помощью условного оператора SELECT.

Почему я не могу поставить иконки в меню?

Вы вероятно заметили, что в меню PureBasic IDE расположены иконки. Но нет встроенных команд для расположения иконок в меню. PureBasic является кросс-платформенным языком программирования, а значит его код, должен поддерживаться на любой из поддерживаемых платформ. Таким образом, все поддерживаемые команды должны достичь такого же желаемого эффекта на каждой платформе. Иконки в меню были расценены как слишком трудными для поддержки кросс-платформенности, поэтому они были исключены из окончательного набора команд. Иконки были включены в PureBasic IDE с использованием API каждой платформы (например, WinAPI в случае с Microsoft Windows). Чтобы узнать больше о WinAPI, см. главу 13.

Сочетания клавиш в меню



Большинство программ поддерживают горячие клавиши для выбора пунктов меню. Еще их называют клавишами быстрого вызова, нажав на которые, можно вызвать меню, не заходя в него. Вы можете вставить их в программу с легкостью. Для демонстрации этого мы должны создать окно с меню. В этом меню мы определим наши пункты меню, но на этот раз мы будем определять их вместе с акселераторами(надписями сочетаний клавиш). Принято писать акселераторы по правому краю меню, при том не забывая отделять их от названия пункта пробелами. Все это делается для правильного визуального восприятия:

```
Enumeration
   #WIN_MAIN
   #MENU_MAIN
   #M_QUIT
   #M_ABOUT
EndEnumeration
Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Menu Example", #FLAGS)
   If CreateMenu(#MENU_MAIN, WindowID(#WIN_MAIN))
      MenuTitle("File")
      MenuItem(#M_QUIT, "Quit" + #TAB$ + "Ctrl+Q")
      MenuTitle("Help")
      MenuItem(#M_ABOUT, "About..." + #TAB$ + "Ctrl+A")

AddKeyboardShortcut(#WIN_MAIN,#PB_Shortcut_Control|#PB_Shortcut_Q,#M_QUIT)
```

```
AddKeyboardShortcut(#WIN_MAIN,#PB_Shortcut_Control|#PB_Shortcut_A,#M_ABOU
T)
      Repeat
        Event.l = WaitWindowEvent()
        Select Event
          Case #PB_Event_Menu
            Select EventMenu()
              Case #M_QUIT
                Quit = #True
              Case #M_ABOUT
                MessageRequester("About", "This is where you describe your program.")
            EndSelect
        EndSelect
      Until Event = #PB_Event_CloseWindow Or Quit = #True
   EndIf
EndIf
End
```

Если вы запустите пример, вы увидите, что в пункты меню вставлены акселераторы. Они разделяются от меню с помощью символов табуляции:

```
...
MenuItem(#M_QUIT, "Quit" + #TAB$ + "Ctrl+Q")
...
```

Эта команда, как вы знаете, имеет два параметра, но второй параметр строка состоит из трех частей. Первая часть пункта меню строка, собственно название Quit. Затем мы объединяем с помощью константы табуляции #TAB$ строки "Quit" и "Ctrl+Q". Эта константа имеет значение 9 в ASCII, то есть запись #TAB$ равноценна записи CHR(9) . Чтобы добавить сочетания клавиш акселераторов, мы должны использовать команду AddKeyboardShortcut():

```
...
AddKeyboardShortcut(#WIN_MAIN,#PB_Shortcut_Control|#PB_Shortcut_Q,#M_QUIT)
AddKeyboardShortcut(#WIN_MAIN,#PB_Shortcut_Control|#PB_Shortcut_A,#M_ABOU
T)
...
```

Эта команда имеет три параметра, первый номер PB нашего окна, с которым связана горячая клавиша, в данном случае это #WIN_MAIN. Вторым параметром является значение, которое представляет собой фактическую комбинацию клавиш. Эта величина состоит из встроенных констант, которые представляют различные комбинации клавиш на клавиатуре. В первой строке в приведенном выше фрагменте, я объединил константы #PB_Shortcut_Control и #PB_Shortcut_Q, чтобы указать, что я хочу получить горячую клавишу Ctrl + Q на клавиатуре. Эти константы комбинируются обычным способом, с помощью побитового оператора | (or). Все доступные константы для сочетаний горячих клавиш перечислены в справке PureBasic для команды AddKeyboardShortcut() (Helpfile:Reference Manual->General Libraries->Window->AddKeyboardShortcut). Третий параметр PB, это номер пункта меню, который вы хотите ассоциировать с горячей клавишей. В первой строке в приведенном выше фрагменте, я использовал #M_QUIT. Это номер одного из пунктов меню. Это означает, что после нажатия на клавиатуре Ctrl+Q,

будет вызвано меню #M_QUIT, которое связано с событием присваивания переменной Quit значения #TRUE. Что в свою очередь вызовет выход из цикла и завершение программы. На следующей строке кода, я повторил эту команду, но с другими параметрами, чтобы предоставить клавишу быстрого вызова для пункта меню с названием About и номером #M_ABOUT. Эти события сочетаний клавиш обрабатываются точно так же, как если бы они были спровоцированы из выбора пункта меню и не существует абсолютно никакой разницы для обработки их в основном цикле, как вы могли видеть из прошлого примера.

Сочетания клавиш без меню

Используя те же сочетания клавиш, можно создать горячие клавиши и без меню. Все что вам нужно сделать, это использовать команду AddKeyboardShortcut() для создания горячей клавиши, связанную с вашим окном, и вместо номера пункта меню в третий параметр, записать к примеру любой идентификационный номер . Вот пример:

```
#WIN_MAIN = 1
#SC_EVENT = 2
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Hello World", #FLAGS)
  AddKeyboardShortcut(#WIN_MAIN,#PB_Shortcut_Control|#PB_Shortcut_Z,
#SC_EVENT)
    Repeat
     Event.l = WaitWindowEvent()
     Select Event
       Case #PB_Event_Menu
          Select EventMenu()
            Case #SC_EVENT
              Debug "The shortcut was pressed"
         EndSelect
    EndSelect
  Until Event = #PB_Event_CloseWindow
EndIf
End
```

Приведенный выше пример работает даже при отсутствии меню. При нажатии на сочетание клавиш Ctrl+Z выводится строка в окно вывода отладки. Этот пример работает, потому что команда AddKeyboardShortcut() вызывает меню с помощью его номера, который обрабатывается в основном цикле. То есть мы как бы подменили вызов меню нашим действием(вывод строки). Я думаю,что вам эти знания могут пригодится, особенно когда нужно будет добавить горячие клавиши в свои программы, даже если в вашей программе не будет меню.

Включение графики в вашу программу

Иногда, может понадобится в ваших программах включать определенные изображения, и PureBasic делает это очень просто. В следующем разделе я покажу вам, как добавить изображения на свой интерфейсы двумя способами. Первый способ: вы можете загрузить из внешнего источника, например, изображение может быть в той же директории программы. Второй способ: можно вставлять изображения напрямую в вашу программу из ресурсов исполняемого файла. Оба способа дают нужный результат, но все же от вас зависит, как вы хотите загрузить изображение. Оба этих метода требуют ImageGadget,

размещенный на вашем окне для отображения изображения. Он находится на вашем окне так же, как и любой другой гаджет, и соответствуют тем же правилам.

Загрузка изображений в Вашей программе

Этот метод загрузки изображений для использования в вашей интерфейса зависит от внешних изображений. В отличие от вложенных изображений, этот метод требует от вас распространять используемые изображения, вместе с исполняемым файлом. Вот пример:

```
Enumeration
   #WIN_MAIN
   #IMAGE_FILE
   #IMAGE_DISPLAY
   #BUTTON_CLOSE
EndEnumeration
Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Image Example", #FLAGS)
   If CreateGadgetList(WindowID(#WIN_MAIN))
      If LoadImage(#IMAGE_FILE, "image.bmp")
         ImageGadget(#IMAGE_DISPLAY, 10, 10, 280, 150, ImageID(#IMAGE_FILE))
         ButtonGadget(#BUTTON_CLOSE, 100, 170, 100, 20, "Close window")
         Repeat
            Event.l = WaitWindowEvent()
            Select Event
               Case #PB_Event_Gadget
                  Select EventGadget()
                     Case #BUTTON_CLOSE
                        Quit = #True
                  EndSelect
            EndSelect
         Until Event = #PB_Event_CloseWindow Or Quit = #True
      EndIf
   EndIf
EndIf
End
```

Для этого примера для правильной работы, вам необходим файл изображения 280x150 в формате .bmp в той же папке, что и исходный код или скомпилированный файл EXE. После этого, в действии, программа будет выглядеть примерно так (см. ниже). В этом примере я использовал команду LoadImage() для загрузки изображения с жесткого диска в память:

```
...
LoadImage(#IMAGE_FILE, "image.bmp")
...
```

Если вы посмотрите на команду LoadImage() в справке (Helpfile:Reference Manual->General Libraries->Image->LoadImage) вы можете увидеть в синтаксисе этой команды, что она принимает три параметра, первый номер PB, который будет связан с загруженным изображением. Второй имя и путь к файлу на жестком диске. Если прописывать только имя файла, то программа будет считать, что файл находится в той же папке, что и

исполняемый файл. Я не использовал третий дополнительный параметр в данном примере. Как обычно, я проверяю возвращаемое значение этой команды с помощью команды IF, дабы убедиться, что файл изображения загружен, прежде чем продолжить работу остальной части программы. После того, как изображения были загружены корректно, вы можете поместить его в ImageGadget(). Пример определения ImageGadget() в списке гаджетов:

```
...
ImageGadget(#IMAGE_DISPLAY, 10, 10, 280, 150, ImageID(#IMAGE_FILE))
...
```

Первым параметром ImageGadget() является число PB(идентификатор гаджета). Следующие четыре параметра связаны с расположением и размерами изображения. В пятом параметре мы указываем идентификатор ОС этого изображения. Поскольку мы загрузили подходящее изображение, мы уже можем использовать команду ImageID() для получения ОС идентификатора. В приведенном выше коде я использовал команду ImageID(# image_file) для получения идентификатора ОС. В свою очередь #image_file стал действенным, после того как мы загрузили изображение с помощью команды LoadImage().

Это способ загрузки изображений для отображения их в ImageGadget() является простым способом, но вы должны помнить, что, когда используете этот способ, вы всегда должны распространять изображения, используемые вместе с файлом программы. Если вы хотите, чтобы ваши рисунки были в ресурсах вашего файла программы , читайте дальше.

Вложенное изображение в вашей программе

Предыдущий метод загрузки изображений хорош, но иногда необходимо создать исполняемый файл со встроенными изображениями в нем.Этот метод не потребует внешних носителей для вашего изображения, поскольку файл будет в ресурсах файла EXE. Этот метод использует разделы секции ресурса файла, в него вы можете вставлять бинарные файлы и использовать при необходимости. Вот пример:

```
Enumeration
  #WIN_MAIN
  #IMAGE_MEMORY
  #IMAGE_DISPLAY
  #BUTTON_CLOSE
EndEnumeration

Global Quit.b = #False
```

```
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Image Example", #FLAGS)
  If CreateGadgetList(WindowID(#WIN_MAIN))
    If CatchImage(#IMAGE_MEMORY, ?Image)
      ImageGadget(#IMAGE_DISPLAY, 10, 10, 280, 150, ImageID(#IMAGE_MEMORY))
      ButtonGadget(#BUTTON_CLOSE, 100, 170, 100, 20, "Close window")
      Repeat
        Event.l = WaitWindowEvent()
        Select Event
          Case #PB_Event_Gadget
            Select EventGadget()
              Case #BUTTON_CLOSE
                Quit = #True
            EndSelect
        EndSelect
      Until Event = #PB_Event_CloseWindow Or Quit = #True
    EndIf
  EndIf
EndIf
End

DataSection
Image:
IncludeBinary "image.bmp"
EndDataSection
```

Этот пример очень похож на предыдущий, за исключением, того что это изображение вкладывается в раздел данных при компиляции. После запуска этой программы, изображение считывается из раздела данных , а не с жесткого диска. Если вы посмотрите на пример, вы можете увидеть раздел данных в нижней части исходного кода выглядит так:

```
...
DataSection
Image:
IncludeBinary "image.bmp"
EndDataSection
```

Команды DataSection и EndDataSection являются началом и концом раздела. Вы можете вставлять бинарные файлы в его пределах его с помощью команды IncludeBinary. Эта команда не использует каких-либо скобок, она просто использует строку, в которой записано имя вставляемого файла.

В моем случае я использую для вложения файл под названием image.bmp. Вы также заметили, что я использовал метку перед командой IncludeBinary. Этот знак позволяет вашей программе найти начало этого рисунка в памяти, когда он загружен с этой программой. После того как изображение было вставлено, мы можем легко использовать его в основном коде. Мы больше не используем команду LoadImage(), чтобы загрузить изображение в память, поскольку она загружается из ресурсов программы. Но загрузить из ресурсов все же надо, и для этого я использую команду CatchImage():

```
...
CatchImage(#IMAGE_MEMORY, ?Image)
...
```

Если вы посмотрите на команду CatchImage() в справке (Helpfile:Reference Manual->General Libraries->Image->CatchImage) вы можете увидеть что команда принимает три параметра: Первый номер PB, с которым будет связан образ изображения. Вторым параметром является адрес памяти, где этот образ находится. Если вы помните, я использовал метку в разделе данных, и мне при загрузке изображения нужен адрес памяти. Делается это с помощью знака вопроса перед названием метки, когда я использую его в качестве параметра. Этот знак нужен для того чтобы вернуть адрес памяти нашей метки. Более подробно про указатели и метки в главе 13 (указатели). Я не использовал третий параметр в этом примере, потому что он не является обязательным. Изображения загружаются с помощью команды CatchImage() точно так же, как при помощи оператора LoadImage().Ну а далее мы опять используем ImageGadget и заполняем его параметры таким же образом, как и прежде:

```
...
ImageGadget(#IMAGE_DISPLAY, 10, 10, 280, 150, ImageID(#IMAGE_MEMORY))
...
```

После запуска данного примера она будет выглядеть таким же образом. Это потому, что это та же программа, но теперь она больше не нуждается в внешнем файле image.bmp.

Какие графические форматы можно использовать?

В последних примерах, использовались изображения в формате Bitmap (*. BMP) , но вы не ограничены в этом одном формате.

Стандартно можно загружать и отображать Bitmap (*. BMP) и Icon (*. ICO) форматы файлов, но иногда нужно использовать другие форматы. Для этого можно использовать декодеры. Использование декодеров чрезвычайно просто в использовании, вы просто используете команду установки декодера для нужного формата в начале вашего исходного кода и с тех пор в вашей программе, все команды связанные с загрузкой изображений в этом формате выполняются без каких либо ограничений. Более подробную информацию о декодерах вы можете найти в справке (Helpfile:Reference Manual->General Libraries->ImagePlugin). Вот команды декодеров:

UseJPEGImageDecoder() - поддерживаются (*.jpg/*.jpeg) форматы

UsePNGImageDecoder() - поддерживаеся (*.png) формат

UseTIFFImageDecoder() - поддерживаются (*.tif/*.tiff) форматы

UseTGAImageDecoder() - поддерживается (*.tga) формат

Как отмечалось ранее, использование этих декодер команд просто. Возьмем последний пример вложения изображений и добавим в него поддержку формата Jpeg для того, чтобы мы могли вставлять Jpeg файл вместо файла Bitmap:

```
UseJPEGImageDecoder()


Enumeration
  #WIN_MAIN
  #IMAGE_MEMORY
  #IMAGE_DISPLAY
  #BUTTON_CLOSE
EndEnumeration


Global Quit.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WIN_MAIN, 0, 0, 300, 200, "Image Example", #FLAGS)
  If CreateGadgetList(WindowID(#WIN_MAIN))
    If CatchImage(#IMAGE_MEMORY, ?Image)
      ImageGadget(#IMAGE_DISPLAY, 10, 10, 280, 150, ImageID(#IMAGE_MEMORY))
      ButtonGadget(#BUTTON_CLOSE, 100, 170, 100, 20, "Close window")
      Repeat
        Event.l = WaitWindowEvent()
        Select Event
          Case #PB_Event_Gadget
            Select EventGadget()
              Case #BUTTON_CLOSE
                Quit = #True
            EndSelect
        EndSelect
    Until Event = #PB_Event_CloseWindow Or Quit = #True
  EndIf
 EndIf
EndIf
End


DataSection
  Image:
  IncludeBinary "image.jpg"
EndDataSection
```

Это прошлый пример, только сделана поддержка формата JPEG. Я вставил в исходный код декодер первой строкой. Эта программа сейчас вкладывает изображение в формате Jpeg вместо изображения в формате .bmp. Все остальные декодеры используются таким же образом. Достаточно добавить команду декодер в верхней части исходного кода, чтобы добавить поддержку этих дополнительных форматов файлов.

Первый взгляд на визуальный редактор

В комплекте с PureBasic идет мощный визуальный конструктор, чтобы вы могли построить интерфейс визуально и динамически. Это означает, то, что вы можете разместить ваши гаджеты в окне визуально, создать свой интерфейс, а код за вас

сгенерируется автоматически в реальном времени. Использование таких инструментов, поможет сэкономить много времени при проектировании интерфейсов, регулируя вид и поведение интерфейса, просматривая немедленные результаты. Вы также можете сохранить проект на случай, если вам нужно будет что-то добавить или предоставить дополнительные функции в вашей программе. Существует одно ограничение в настоящее время в отношении загрузки существующих источников, код которых написан не в визуальном конструкторе, он может показать неправильно результат. Это происходит потому, визуальный конструктор генерирует код, определенным образом и все рукописные отклоненные от его формата тексты, будет принимать не как должно.

Почему ты не говорил об нем раньше?

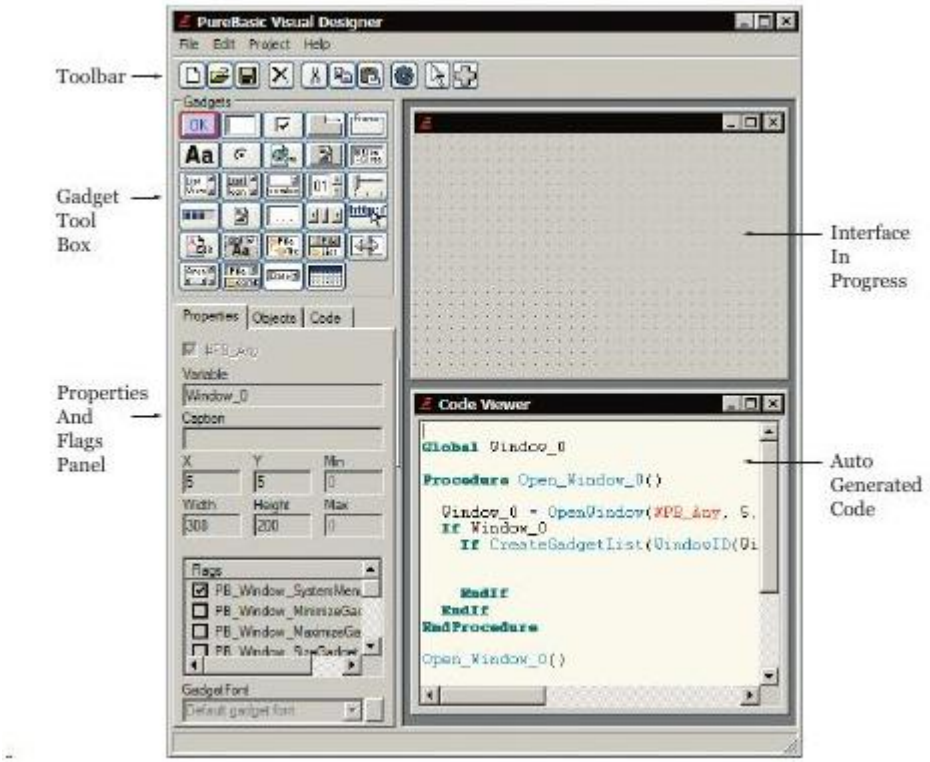
Я думал, что для Вас будет очень важно изучить, как работают в PureBasic устройства, такие как меню и события, чтобы дать вам более твердое основание понимание кода, выполненного в визуальном конструкторе.

Главное окно

При запуске Visual Designer будет выглядеть см ниже. Главное окно содержит все инструменты, для того чтобы вам легко было создать свой интерфейс. Визуальный конструктор состоит из меню в верхней панели инструментов, и под ним. В левой части экрана находится панель инструментов, содержащие множество гаджетов и панели Свойства. В центре будет текущий интерфейс разработки и ниже автоматически сгенерированный код.

Использование Visual Designer

Если вы посмотрите на рисунок ниже, вы увидите интерфейс, состоящий пока лишь из пустого окна, которое в свою очередь покрыто маленькими точками. Это сетка, она нужна для удобного размещения гаджетов на окне. Сетка не отображается в готовой программе.



После того как вы поставили нужные размеры окна(они регулируются) ,вы можете начать размещение гаджетов на нем. Для этого на панели с гаджетами нужно выделить необходимый гаджет , а после нарисовать его на окне мышкой прямо по точкам. После того как вы установили гаджет, вы увидите, что он окружен большими синими точками. Это так называемые ручки, позволяющие изменять размер. Кроме того вы можете перемещать гаджет в любое место. При добавлении гаджетов в интерфейсе можно заметить, что код генерируется автоматически в нижней части экрана. После того как все гаджеты размещены и у вас все готово, вы можете сохранить ваш проект. Позднее вы всегда можете его открыть в визуальном редакторе и отредактировать. Ну и конечно открыв его в IDE продолжить разработку кода, ведь для этого мы его и создавали.

За и против визуальный конструктор

Хотя визуальный конструктор это отличный инструмент для быстрой и простой разработки интерфейса, в нем сгенерированный код не может быть изменен. Таким образом, создавая в нем вы должны будете придерживаться его встроенного формата. Конечно не проблема, если вы являетесь опытным программистом в PureBasic, вы легко потом подправите в IDE код так как вам надо, но и для начинающего пользователя, экспортированный код может показаться довольно сложным. Главным козырем визуального конструктора является скорость. Вы можете размещать гаджеты так как вам надо, до тех пор пока не будет достигнут желаемый результат. И уж конечно это намного быстрее, чем если бы вы создавали это в IDE. Некоторые люди говорят, что лучше проект составлять в редакторе IDE, другие говорят что незачем тратить время на составление кода для интерфейса. Ну а вы решите сами за себя, что вам удобнее и практичнее.

Узнать больше о визуальном конструкторе

Визуальный конструктор поставляется в комплекте с обширной справкой, которая охватывает всю работу с ним. Для просмотра справки нажмите клавишу F1 при

включенном визуальном конструкторе и справка загрузится. Я рекомендую прочитать ее полностью по крайней мере один раз. Новые версии визуального конструктора, наверняка более развитые, так что если есть возможность, то лучше обновить с сайта PureBasic визуальный дизайнер. В приложении указаны все необходимые и полезные интернет-ссылки.

Графика и звук

Программы, использующие графику и звук чаще всего являются компьютерными играми и интерактивными приложениями. При программировании такого, вам нужны инструменты, необходимые для реализации ваших идей. PureBasic предлагает простые и в то же время мощные и полнофункциональные команды, которые могут создать качественные игры, демосцены, скринсейверы и другие интерактивные программы. В этом разделе я покажу вам основы использования графики и звука в ваших программах. Сначала рассказ пойдет о 2D командах. Они нужны для рисования простых форм, линий и текста. И конечно я продемонстрирую, как сохранять эти образы. Затем я покажу вам, как сделать полный размер графического экрана. Кроме того, я познакомлю вас со спрайтами и как использовать их для создания графических эффектов. После этого мы перейдем на 3D-графику, где я представлю вам имеющийся 3D-движок и покажу вам основную информацию об использовании 3D-графики в PureBasic. Этот движок называется OGRE и стоит на третьем месте по созданию качественной и профессиональной графики и эффектов. В последней главе данного раздела, пойдет речь о звуке. Я покажу как загружать и воспроизводить звуки в форматах WAV, Tracker модули, MP3 и CD Audio. Главы, содержащиеся в этом разделе, не являются полным руководством по достижению каждого графического эффекта и уж тем более не дадут вам полных знаний для написания серьезных игр. Эти главы для того, чтобы познакомить вас с библиотеками команд, которые дадут вам плацдарм и основу. Но даже этого хватит, чтобы вы могли поэкспериментировать. Надеюсь, этот раздел вдохновит вас на создание демосцен или небольшой игры.

9

2D графика

В этой главе я собираюсь объяснить, как рисовать 2D графику с помощью PureBasic. Термин 2D графика охватывает довольно большую тему. Когда я рассказываю о 2D-графике в этой книге, я имею в виду простую, двумерную графику, которую можно изобразить на экране в GUI, изображении в ОЗУ или даже принтере. Эта графика может содержать различные типы сложности, от одного пикселя, вплоть до большого многопиксельного цветного изображения. Вся 2D графика двумерная, это можно понять по названию. Есть еще 3D-графика, которая отображает 3D модели, и имеет трехмерный вид. Используя 2D-графику, можно добиться весьма впечатляющих визуальных эффектов, и как следствие, множество игр , заставок и демосцен, были созданы с помощью PureBasic. Надеюсь, вы сможете обучиться и данные примеры из этой главы, станут началом для создания великолепной графики в ваших программах.

2D команды для рисования

PureBasic содержит команды, которые позволяют использовать стандартные формы и цвета в ваших программах. Эти простые 2D команды для рисования описываются в главе *2D Drawing* справки по PureBasic (Helpfile: ReferenceManual->General Libraries->2DDrawing). Эти команды являются полезными для рисования простых фигур и текста в разные цвета и даже есть пара команд, которые рисуют ранее созданные или загруженные изображения.

Способы рисования

Встроенные 2D команды можно использовать для рисования таких элементов как: графического оконного интерфейса пользователя, зарисовок экрана, спрайта, только что созданного образа в памяти, текстуры 3D модели или прямо на принтер. Этих шести методов достаточно, чтобы охватывать все, связанное с 2D и в то же время использовать их одинаково просто. Вам достаточно раз указать, где вы хотите выводить рисунок(1 из шести методов, описанных выше), а дальше с помощью команд рисования выводить нужное в эту область. Все шесть применительных методов, используют приблизительно один и тот же синтаксис.

Рисование по окну

В этом первом примере я покажу вам, как нарисовать 2D-графику непосредственно на окно. Вы, наверное, никогда не будет использовать такой способ, но это поможет понять и изучить синтаксис. Посмотрите на этот кусок кода:

```
#WINDOW_MAIN = 1
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, 200, 240, "Window Drawing", #FLAGS)
   If StartDrawing(WindowOutput(#WINDOW_MAIN))
      Box(15, 15, 75, 75, RGB(255, 0, 0))
      Circle(140, 125, 45, RGB(35, 158, 70))
      ;The next 2D drawing commands draw a triangle
      LineXY(62, 140, 112, 220, RGB(0, 0, 255))
      LineXY(112, 220, 12, 220, RGB(0, 0, 255))
      LineXY(12, 220, 62, 140, RGB(0, 0, 255))
      FillArea(62, 180, RGB(0, 0, 255), RGB(0, 0, 255))
   StopDrawing()
EndIf
   Repeat
      Event.l = WaitWindowEvent()
   Until Event = #PB_Event_CloseWindow
EndIf
End
```

Если вы запустите приведенный выше пример, можете увидеть окно, подобное изображенному ниже на рисунке. В нем отображены: красный квадрат, зеленый круг и синий треугольник. Эти формы отображаются в окне без всяких гаджетов. Для того, чтобы вы в полной мере поняли код выше, я должен подробно объяснить синтаксис команд из прошлого примера.

Во-первых, вы увидите, я использовал новую команду с названием StartDrawing(). Эта команда сообщает PureBasic о начале рисования, используя 2D графику. А так же определяет каким методом рисовать. Если вы посмотрите в справку (Helpfile:Reference Manual->General Libraries->2DDrawing->StartDrawing) , вы увидите, что эта команда может принимать в качестве параметра одну из шести команд, которая отвечает за метод вывода рисунка. Эти команды:

WindowOutput()

Используется для рисования на окнах PureBasic. При этом методе необходимо указать идентификатор окна.

ScreenOutput()

Используется для рисования непосредственно на экран PureBasic.

SpriteOutput()

Используется для рисования на спрайтах. При этом методе необходимо указать идентификатор спрайта.

ImageOutput()

Используется для рисования на загруженных или вновь созданных изображениях в памяти. При этом методе необходимо указать идентификатор изображения.

PrinterOutput()

Используется для рисования непосредственно в принтер.

TextureOutput()

Используется для рисования на текстуры 3D модели. При этом методе необходимо указать идентификатор текстуры.

В моем примере я я дал указание рисовать на окне:

```
...
StartDrawing(WindowOutput(#WINDOW_MAIN))
...
```

Как только я настроил вывод рисунка с помощью команды WindowOutput, я могу начать рисовать с помощью различных 2D команд. Теперь все команды записанные в блок *StarDrawing(WindowOutput())-----StopDrawing* будут выводить рисунок на окно. Обязательно при использовании команды StartDrawing(), после инструкций рисования, необходимо завершить блок командой StopDrawing. Это позволит компилятору PureBasic отказаться от использования 2D команд рисования, и продолжать работать с остальной частью программы. Это необходимо делать, иначе возникнут ошибки в программе. Внутри блока, я использовал четыре 2D команды для рисования фигур на окне: Box() Circle() LineXY() FillArea(). Все эти 2D команды подробно описаны в справке (Helpfile:Reference Manual->General Libraries->2DDrawing). Почти все 2D команды имеют сходные параметры. Этими параметрами чаще всего выступают: начальная позиция, размеры и цвет. Если мы возьмем к примеру команду Box(), то первые два параметра это начальная позиция прямоугольника на окне. Третий и четвертый параметры ширины и высоты прямоугольника. Пятый параметр это его цвет.

Работа с цветами

Цвета в PureBasic задаются с помощью 24-битного значения цветности . Эти значения являются различными сочетаниями красного, зеленого и синего цветов. Для получения 24 бит значения цвета, можно использовать команду RGB(). Эта команда получает три параметра, которые указывают значения красного, зеленого и синего цветов по отдельности. Каждый из этих показателей имеет диапазон значений от 0 до 255. С их помощью возможно получить более 16,7 млн 24-битного значения цветности. Вот как я использовал команду в прошлом примере:

```
...
Box(15, 15, 75, 75, RGB(255, 0, 0))
...
```

Здесь, я использовал команду RGB() встроенную в качестве параметра в команду Box(). Если внимательно присмотреться, можно увидеть, что я указал максимальное значение для параметра красный, и нулевое значение для зеленого и синего цветов. Это означает, что RGB() команда возвращает 24 битное значение красного цвета не смешанного с двумя другими цветами. Таким же способом и для треугольника:

```
 ...
LineXY(62, 140, 112, 220, RGB(0, 0, 255))
LineXY(112, 220, 12, 220, RGB(0, 0, 255))
LineXY(12, 220, 62, 140, RGB(0, 0, 255))
FillArea(62, 180, RGB(0, 0, 255), RGB(0, 0, 255))
...
```

Эти четыре строчки составляют синий треугольник из прошлого примера. Первые три являются командами LineXY() которые рисуют три стороны треугольника. Последняя команда FillArea() берет отправную точку в центре треугольника, и наполняет область тем же цветом, что и стороны. Вы можете видеть цвет же во всех этих четырех командах рисования. Первый и второй параметры команды RGB() представляющие красный и зеленый, а третий параметр представляет синий. В данном примере видно, что последний параметр равен 255, а первые два 0,значит отобразится синий цвет без примесей. Зеленый кружок из примера, немного отличается тем, что я использую все три цвета, чтобы достичь желаемого зеленого цвета, например:

```
...
Circle(140, 125, 45, RGB(35, 158, 70))
...
```

Здесь я использовал сочетание красного, зеленого и синего, чтобы добиться зеленого цвета. Использование цветов довольно просто в PureBasic, поэтому у новичков не должно возникнуть проблем с пониманием. Использование команды RGB() может вернуть практически любой цвет, различаемый человеческим глазом.

Команда RGB() может устанавливать нужный цвет из красного, зеленого и синего параметров 24 битного значения. Может случится так, что вам надо будет вернуть используемый цвет в числовой эквивалент. Для этого можно воспользоваться командами Red() Green() Blue() Вот пример:

```
ColorValue.l = RGB(35, 158, 70)
Debug "The 24 bit color value is made up of: "
Debug "Red: " + Str(Red(ColorValue))
Debug "Green: " + Str(Green(ColorValue))
Debug "Blue: " + Str(Blue(ColorValue))
```

Каждая из команд Red() Green() Blue() возвращает соответствующее значение компонента цвета. Первым параметром в команде RGB() идет число 35. Если я захочу получить значение красного компонента позднее, я попросту воспользуюсь командой Red() . С другими цветами поступаем аналогично.

Почему нет предпочтений для рисования на окнах в Windows

Из-за внутренней обработки событий, что происходит в Microsoft Windows, рисунок нанесенный непосредственно на окно при переходе между окнами, либо свертыванием окна, попросту затирается. Конечно можно заставить графику перерисовываться, но для этого нужны знания API. Более удобный способ отображения графики, это создавать новый образ в памяти, рисовать в нем, а затем поместить это изображение с помощью гаджета в вашем окне. Использование гаджета (ImageGadget()) гарантирует, что нужное изображение автоматически перерисовывается. Из-за этого на изображение не влияют такие действия, как свертывания окна, переключения между окнами и т д.

Рисование на изображениях

Работа с новыми изображениями в PureBasic легка, так как есть целая библиотека написанная исключительно для создания и манипулирования этими образами. В следующем примере, я создам похожую программу с треугольником, прямоугольником и кругом, но все же у этих примеров есть разница. Я собираюсь создать новый образ, используя команду CreateImage() для рисования цветных форм на него, а потом отправить конечный результат в ImageGadget(). Это поможет избежать проблем с обновлением окна. Вы можете больше узнать об ImageGadget() в справке (Helpfile:Reference Manual->General Libraries->Image). А вот пример:

```
Enumeration
#WINDOW_MAIN
#IMAGE_GADGET
#IMAGE_MAIN
EndEnumeration
```

```
If CreateImage(#IMAGE_MAIN, 180, 220)
   If StartDrawing(ImageOutput(#IMAGE_MAIN))
      ;Because a new image has a Black background, draw a white one instead:
      Box(0, 0, 180, 220, RGB(255, 255, 255))
      ;Now, continue drawing the shapes:
      Box(5, 5, 75, 75, RGB(255, 0, 0))
      Circle(130, 115, 45, RGB(35, 158, 70))
      LineXY(52, 130, 102, 210, RGB(0, 0, 255))
      LineXY(102, 210, 2, 210, RGB(0, 0, 255))
      LineXY(2, 210, 52, 130, RGB(0, 0, 255))
      FillArea(52, 170, RGB(0, 0, 255), RGB(0, 0, 255))
      StopDrawing()
   EndIf
EndIf
#FLAGS=#PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, 200, 240, "Drawing On A New Image",
#FLAGS)
   If CreateGadgetList(WindowID(#WINDOW_MAIN))
      ImageGadget(#IMAGE_GADGET, 10, 10, 180, 220, ImageID(#IMAGE_MAIN))
      Repeat
         Event.l=WaitWindowEvent()
      Until Event=#PB_Event_CloseWindow
   EndIf
EndIf
End
```

Основная часть этого кода должна быть понятна, потому что единственное различие с кодом из последнего примера, это то что мы здесь создаем новое изображение, используя команду CreateImage() и рисуем на нем, а не на окне. Команда CreateImage() принимает четыре параметра. Первый это идентификатор нового изображения. Второй и третий ширина и высота (в пикселях) нового образа, и четвертый параметр глубина цветности. Этот параметр не является обязательным. Если четвертый параметр не используется, как в моем примере, то он использует ту же самую глубину цвета, которая используется для рабочего стола. После создания нового образа, я командую программе начать рисовать на нем с помощью этой строки:

```
...
StartDrawing(ImageOutput(#IMAGE_MAIN))
...
```

Здесь, используя команду StartDrawing(), я указал метод рисования ImageOutput(), который будет выводить рисование на изображение с идентификатором # IMAGE_MAIN. В этом способе нет ограничения на рисование только нового созданного изображения. Вы можете так же легко загрузить любое фото в программу и используя идентификатор изображения, начать рисовать на нем.

После того как изображение было создано и выбран метод рисования, можно непосредственно рисовать на нем. Поскольку все вновь созданные изображения в PureBasic имеют черный фон, первой операцией рисования я решил сделать изменение цвета фона на белый. Я делаю это с помощью команды **Box(0, 0, 180, 220, RGB(255, 255,**

255))**, которая покрывает черное изображение белым фоном. После этого, я нарисовал квадрат, круг и треугольник так же, как в прошлом примере. Когда я закончил все необходимые операции рисования, я вызываю команду StopDrawing(). Чтобы отобразить это изображение правильно и избежать проблем при перерисовках(обновлении рисунка) я использую ImageGadget. Это позволяет операционной системе заботиться о вашем изображении при свертывании, переключении фокуса окон, перемещении окна за грань экрана и т.д.

О глубине изображения.

Каждый пиксель на дисплее компьютера или цифрового изображения описывается двоичным числом, как в принципе и любая информация на компьютере. Большее количество бит (или двоичных цифр) используемое для описания одного пикселя, позволяет этому пикселю, иметь более широкую цветовую гамму. Количество бит, описывающих отдельный пиксель в том или ином цифровом изображении или компьютерного монитора, называется глубиной цвета. Используются: 1 бит, 8 бит, 16 бит, 24 бит, 32 бит. Изображения 1 бит может описать только черно-белые(2 цвета) изображения. В зависимости от разрядности этого бита(0 или 1) будет черный или белый цвет. Пиксели с глубиной 32 бит способны отображать больше цветов, чем человеческий глаз может распознать, поэтому этот формат регулярно используется в изображениях для цифровых кинофильмов, цифровых фотографий, реалистичных компьютерных игр и т.д. Современные компьютеры в настоящее время, как правило, используют только 32 бит на пиксель.

Изменение режима рисования

В следующем примере я покажу вам, как изменить режим рисования некоторых команд, включая текст. Использовав команду DrawingMode() вы можете переключиться на режим команды для набросков форм, рисовать текст с прозрачным фоном, или даже смешивать(XOR) пиксели рисунка с пикселями фона. Вот пример с режимом рисования:

```
Enumeration
#WINDOW_MAIN
#IMAGE_GADGET
#IMAGE_MAIN
#FONT_MAIN
EndEnumeration


Global ImageWidth=401
Global ImageHeight=201
Global XPos.l, YPos.l, Width.l, Height.l, Red.l, Green.l, Blue.l
Global Text.s="PureBasic - 2D Drawing Example"
Procedure.l MyRandom(Maximum.l)
  Repeat
    Number.l=Random(Maximum)
  Until (Number % 10)=0
  ProcedureReturn Number
EndProcedure
If CreateImage(#IMAGE_MAIN, ImageWidth, ImageHeight)
  If StartDrawing(ImageOutput(#IMAGE_MAIN))
    For x.l=0 To 1500
```

```
      XPos.l=MyRandom(ImageWidth)+1
      YPos.l=MyRandom(ImageHeight)+1
      Width.l=(MyRandom(100)-1)+10
      Height.l=(MyRandom(100)-1)+10
      Red.l=Random(255)
      Green.l=Random(255)
      Blue.l=Random(255)
      Box(XPos, YPos, Width, Height, RGB(Red, Green, Blue))
      DrawingMode(#PB_2DDrawing_Outlined)
      Box(XPos-1, YPos-1, Width+2, Height+2, RGB(0, 0, 0))
      DrawingMode(#PB_2DDrawing_Default)
    Next x
      LineXY(ImageWidth-1, 0, ImageWidth-1, ImageHeight, RGB(0, 0, 0))
      LineXY(0, ImageHeight-1, ImageWidth, ImageHeight-1, RGB(0, 0, 0))
      Box(10, 10, 230, 30, RGB(90, 105, 134))
      DrawingMode(#PB_2DDrawing_Outlined)
      Box(10, 10, 231, 31, RGB(0, 0, 0))
      DrawingMode(#PB_2DDrawing_Transparent)
      DrawText(21, 18, Text, RGB(0, 0, 0))
      DrawText(19, 18, Text, RGB(0, 0, 0))
      DrawText(21, 16, Text, RGB(0, 0, 0))
      DrawText(19, 16, Text, RGB(0, 0, 0))
      DrawText(20, 17, Text, RGB(255, 255, 255))
    StopDrawing()
  EndIf
EndIf
#FLAGS=#PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_MAIN, 0, 0, ImageWidth+20, ImageHeight+20, "Abstract",
#FLAGS)
  If CreateGadgetList(WindowID(#WINDOW_MAIN))
    ImageGadget(#IMAGE_GADGET, 10, 10, ImageWidth, ImageHeight,
ImageID(#IMAGE_MAIN))
  EndIf
  Repeat
    Event.l=WaitWindowEvent()
  Until Event=#PB_Event_CloseWindow
EndIf
End
```

Этот пример, в основном расширенная версия прошлого. В нем я так же создал новый образ, и в этом образе нарисовал нужное. Кол-во форм, созданных с помощью BOX() здесь большое, и появление их на экране случайно, то есть случайны их размеры, цвет и нахождение на нашем образе. В каждом прямоугольнике я рисовал, переключая режим рисования, при этом намечая черный контур с такими же размерами сверху. Это делает прямоугольники выделяющимися. Мало того, что каждый прямоугольник имеет случайный цвет, но каждый из них так же имеет черный контур. Для переключения режимов во время рисования, вы можете использовать команду DrawingMode() вместе с одной из нескольких констант, которые используются в качестве параметра. Вот эти константы, а так же возможности, связанные с ними:

#PB_2DDrawing_Default

Этот режим по умолчанию, текст отображается с цветом фона и графические формы заполнены.

#PB_2DDrawing_Outlined

Этот режим позволяет делать наброски форм, для таких команд, как Box() Circle() и Ellipse()

#PB_2DDrawing_Transparent

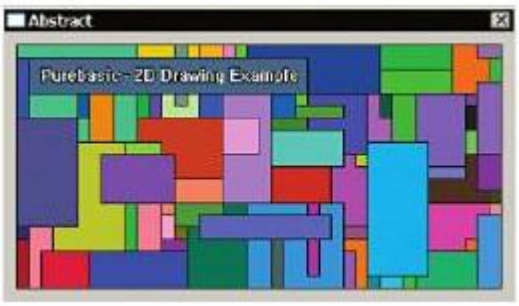
Устанавливает фон текста прозрачным, поэтому и используется только для команды DrawText()

#PB_2DDrawing_XOr

Это режим XOR, который означает, что вся графика будет преобразована(Поксорена) с текущим фоном.

Использовать эти команды очень просто. Если вы хотите изменить рисунок, нужно вызвать команду StartDrawing() с константами по своему выбору. Эти константы могут быть объединены с помощью побитового оператора OR. Взглянув на код, вы можете увидеть, что я много раз включал режимы.

После запуска этого примера вы увидите окно, очень похожее на рисунок ниже. В нем показываются красочные прямоугольники, в комплекте с черными очертаниями и белым текстом в левом верхнем углу.



Я использовал режим рисования #PB_2DDrawing_Transparent, когда я выводил текст на изображении, получая текст с прозрачным фоном, вместо сплошного цвета. Чтобы рисовать текст, я использовал команду DrawText(). Она имеет пять параметров: x и y позиция на выходе, String рисуемая строка, и два цветовых параметра(цвет текста и цвет фона текста). Поскольку я использовал прозрачный фон, пятый параметр игнорируется. Черные контуры текста на рисунке были сделаны с помощью таких же команд DrawText(), но со смещением в координатах. Режим # PB_2DDrawing_Outlined не поддерживает текст, поэтому были применены четыре дополнительные команды для контура текста.

Рисование текста

Рисование текста на выходе достигается с помощью команды **DrawText()** (Helpfile:Reference Manual->General Libraries->2DDrawing->DrawText). Если вы посмотрите на последний пример, вы можете увидеть, что я использовал эту команду

несколько раз. Все ее параметры я описал чуть выше. Скажу лишь, если два последние параметра(цвет текста и цвет фона, являющимися необязательными) не используются, то информация берется из значения по умолчанию, которые могут меняться с помощью команд FrontColor() и BackColor(). Эти цветовые параметры 24 битного значения цвета, легко, ставятся с помощью команды RGB().

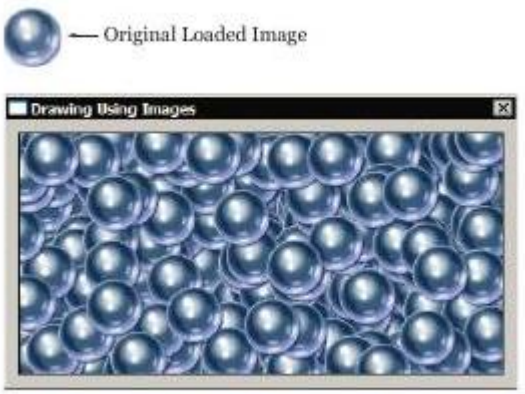
Рисуем с помощью изображений

В последних примерах, я показал, как рисовать простые формы и линии, используя встроенные 2D команды рисования, но на этом 2D графика не ограничивается. В следующем примере я покажу вам, как можно взять изображение и нарисовать его на другой только что созданный образ, в любой позиции, какой хотите:

```
Enumeration
  #WINDOW_MAIN
  #IMAGE_GADGET
  #IMAGE_SMALL
  #IMAGE_MAIN
  #FONT_MAIN
EndEnumeration

Global ImageWidth=400
Global ImageHeight=200
Global XPos.l, YPos.l, LoadedImageWidth.l, LoadedImageHeight.l
Global File.s
Global RequesterText.s="Choose an image"
Global DefaultFile.s=""
Global Pattern.s="Bitmap (*.bmp)|*.bmp|Icon (*.ico)|*.ico"
File=OpenFileRequester(RequesterText, DefaultFile, Pattern, 0)
If File
  LoadImage(#IMAGE_SMALL, File)
  LoadedImageWidth=ImageWidth(#IMAGE_SMALL)
  LoadedImageHeight=ImageHeight(#IMAGE_SMALL)
  If CreateImage(#IMAGE_MAIN, ImageWidth, ImageHeight)
    If StartDrawing(ImageOutput(#IMAGE_MAIN))
      Box(0, 0, ImageWidth, ImageHeight, RGB(255, 255, 255))
      For x.l=1 To 1000
        XPos=Random(ImageWidth)-(ImageWidth(#IMAGE_SMALL)/2)
        YPos=Random(ImageHeight)-(ImageHeight(#IMAGE_SMALL)/2)
        DrawImage(ImageID(#IMAGE_SMALL), XPos, YPos)
      Next x
      DrawingMode(#PB_2DDrawing_Outlined)
      Box(0, 0, ImageWidth, ImageHeight, RGB(0, 0, 0))
      StopDrawing()
    EndIf
  EndIf
  #TEXT="Drawing Using Images"
  #FLAGS=#PB_Window_SystemMenu | #PB_Window_ScreenCentered

  If OpenWindow(#WINDOW_MAIN, 0, 0, ImageWidth+20, ImageHeight+20, #TEXT,
#FLAGS)
    If CreateGadgetList(WindowID(#WINDOW_MAIN))
```

```
      ImageGadget(#IMAGE_GADGET, 10, 10, ImageWidth, ImageHeight,
ImageID(#IMAGE_MAIN))
   EndIf
   Repeat
      Event.l=WaitWindowEvent()
   Until Event=#PB_Event_CloseWindow
  EndIf
End
EndIf
```

В приведенном выше коде, я даю пользователю возможность, с помощью команды OpenFileRequester() открыть изображение в форматах Bitmap или Icon. После этого запроса, при правильном возвращении имени файла изображения, я загружаю файл, с помощью команды LoadImage(). После того как файл будет загружен, я могу создать новый образ и нарисовать на него изображения множество раз. Делается это с помощью команды DrawImage(). Полученное изображение выводится на окне, используя ImageGadget(). Если вы посмотрите на рисунок ниже, вы можете видеть, как он выглядит после того, как загрузили небольшой значок, который затем случайно рисуется во всем созданном образе и отображается.



Если вы хотите рисовать загруженные изображения на вновь созданный образ, вы используете команду DrawImage(). Эта команда принимает пять параметров, которые несложно понять. Первый это идентификатор ОС изображения, которое нужно для рисования. Устанавливается с помощью команды ImageID(). Второй и третий параметры задают горизонтальное и вертикальное положение для рисования изображения в пикселях. Четвертый и пятый параметры являются необязательными, и я не использовал их здесь, но это ширина и высота рисуемого изображения в пикселях. Последними параметрами вы пользуетесь, если вам нужно изменить размер изображения динамически перед рисованием. В моем примере, я большинство параметров делал случайными с помощью команды Random(). И как результат: случайное заполнение образа тысячами экземпляров загруженного изображения. Скорее всего вы в первый раз, увидели команду OpenFileRequester(), поэтому я поясню. Эта команда открывает стандартный запрос для выбора файла пользователем. После того как файл был выбран, команда OpenFileRequester() возвращает его имя в виде строки. Это может быть продемонстрировано с помощью небольшого фрагмента кода:

```
Global File.s
Global RequesterText.s = "Choose an image"
Global DefaultFile.s = ""
Global Pattern.s = "Bitmap (*.bmp)|*.bmp|Icon (*.ico)|*.ico"
File = OpenFileRequester(RequesterText, DefaultFile, Pattern, 0)
Debug File
```

Если вы посмотрите в файл справки(Helpfile:Reference Manual->GeneralLibraries->Requester->OpenFileRequester) по команде OpenFileRequester(), то увидите, что эта команда имеет четыре параметра. Первым из них является текст, отображаемый в заголовке окна. Второй параметр, это ожидаемое имя файла при запросе. Данный параметр можно указывать как **""(двойные кавычки).** В этом случае ничего отображено в имени ожидаемого файла не будет. Третий параметр представляет собой шаблон для выбранных файлов, который позволяют программисту определить, какие файлы должны или не должны быть отображены для отбора в запрос. Четвертый параметр определяет, тип файла из шаблона по номеру индекса. В нашем фрагменте, будет показан Bitmap, поскольку он стоит первым в шаблоне, а индексный файл шаблона начинается с 0.

Файл шаблона поначалу выглядит довольно сложным, но его не трудно понять. Типы файлов разбиты на куски в строке шаблона, и разделяются с помощью побитового оператора. Части каждого расширения работают в паре, для того чтобы иметь возможность указать строку, описывающую расширение файла, а затем собственно само расширение. Например, в приведенном выше фрагменте, картина выглядит следующим образом:

```
"Bitmap (*.bmp)|*.bmp | Icon (*.ico)|*.ico"
```

Если мы разъединим эти куски с помощью пробелов в качестве разделителей, мы можем более отчетливо увидеть, что указано:

```
Bitmap (*.bmp)        *.bmp        Icon (*.ico)        *.ico
```

Первый кусок это строка, сообщающая пользователю какой тип файла нужен для выбора, в данном случае это все BMP файлы. Вторая часть это непосредственно расширение, которое задает нужный тип файла для выбора в окне(остальные отсеиваются). Помните, звездочка используется здесь для обозначения строки с любым именем. Третий кусок это часть другого расширения для отображения пользователю и четвертый как и второй указывает непосредственно само расширение. Во втором случае это все иконки. Если вам нужно задать множество расширений для конкретного типа файла, можно использовать точку с запятой для разделения, вот так:

```
"JPEG |*.jpg ;*.jpeg"
```

Это будет отображать любые файлы с расширениями .JPG или .JPEG. Вы должны помнить, что если вы хотите загрузить несколько видов изображений, кроме растровых изображений или иконок, вам придется использовать другой декодер изображения, как описано в главе 9.

Рисование изображения с альфа-каналом

Иногда вам может понадобиться нарисовать изображение, которое содержит альфа-канал, для сохранения теней или, может быть, прозрачных частей изображения. В прошлом

рисунке с использованием значка, рисуемом в созданном образе, PureBasic сохраняет стандартную информацию об альфа-канале, которая находится внутри этой иконки. Но что произойдет, если вам нужно нарисовать из Tiff или Png формата изображения и сохранить его альфа-канал? Для этого используется команда DrawAlphaImage(). Эта команда будет сохранять найденный в графическом формате альфа-канал. Команда DrawAlphaImage() используется точно так же, как и команда DrawImage(), с той лишь разницей в том, что DrawAlphaImage() имеет три параметра, поэтому не позволяет динамических изменений размера изображения, во время рисования. Вот пример того, как команда может быть использована:

```
...
DrawAlphaImage(ImageID(#IMAGE_PNG), XPos, YPos)
...
```

Команда выше, будет отображать формат изображения Png и сохранять его альфа-канал, рисуя его над любым фоном. И конечно следует всегда помнить, что при загрузке изображений, кроме Bitmap или Icon форматов, вы должны использовать соответствующую команду декодер, как описано в главе 9.

Сохранение изображений

Как только вы создали образ в вашей программе, вы можете сохранить его на диск. В PureBasic решить эту задачу легко с помощью команды SaveImage() (Helpfile:Reference Manual->GeneralLibraries->Image->SaveImage). Эта команда сохранит изображение, имея его идентификатор. Команда SaveImage() принимает четыре параметра. Первый это идентификатор изображения, которое вы хотите сохранить. Второй это имя файла, которое вы хотите закрепить за изображением. Третий параметр является необязательным, и все таки он определяет формат изображения, в который этот образ должен быть сохранен. Четвертый параметр это дополнительное значение для выбранного формата(качество сохраненного изображения) Если мы хотим сохранить изображение, созданное в последнем примере, мы можем добавить следующую строку кода в конец кода примера . После того как программа закончит, она сохранит наш образ:

```
...
SaveImage(#IMAGE_MAIN, "Image.bmp")
...
```

По умолчанию, PureBasic будет сохранять в формат BMP(Bitmap) при использовании этой команды без последних двух дополнительных параметров. Файлу надо задавать правильное расширение (*. BMP), при сохранении.

Сохранение изображений в других форматах

Вы можете сохранять изображения в другие графические форматы, указав одну из трех встроенных констант в качестве третьего параметра:

#PB_ImagePlugin_BMP

Сохранить изображение в формате Bitmap. Это значение по умолчанию, и указывать константу нет необходимости.

#PB_ImagePlugin_JPEG

Сохранить изображение в формате Jpeg. (Для правильной работы вначале должна вызываться команда UseJPEGImageEncoder())

#PB_ImagePlugin_PNG

Сохранить изображение в формате PNG. (Для правильной работы вначале должна вызываться команда UsePNGImageEncoder())

При сохранении изображений с использованием #PB_ImagePlugin_JPEG или #PB_ImagePlugin_PNG надо поставить соответствующую команду кодировщик в верхней части исходного кода, прежде чем использовать SaveImage(). Кодеры достаточно вызвать один раз, чтобы добавить необходимые функции по всей программе. Вот они:

UseJPEGImageEncoder()

Этот кодер добавляет поддержку для изображений JPEG (*. JPG / *. JPEG) форматов.

UsePNGImageEncoder()

Этот кодер добавляет поддержку изображений PNG (*. PNG) формата.

Используя UseJPEGImageEncoder() и тем самым добавляя поддержку Jpeg, вы можете при желании использовать четвертый параметр для команды SaveImage(). Этот параметр указывает значение сжатия сохраненного изображения. Это единственный тип изображения, который в настоящее время поддерживает этот четвертый параметр.

Вот несколько примеров использования команды SaveImage():

```
SaveImage(#IMAGE_MAIN, "Image.bmp")
```

Этот первый пример сохраняет изображение под названием Image.bmp (по умолчанию 24bit формате Bitmap). Обратите внимание, что кодер не нужен потому, что PureBasic поддерживает изображения в формате Bitmap в качестве стандарта.

```
UseJPEGImageEncoder()
...
SaveImage(#IMAGE_MAIN, "Image.jpg", #PB_ImagePlugin_JPEG)
```

Второй пример сохраняет изображение с названием Image.jpg (в формате JPEG используя для сжатия значение по умолчанию 7 , потому что мы не указали его сами).

```
UseJPEGImageEncoder()
...
SaveImage(#IMAGE_MAIN, "Image.jpg", #PB_ImagePlugin_JPEG, 10)
```

Третий пример сохраняет изображение с названием Image.jpg (в формате JPEG используя для сжатия наибольшее значение 10).

```
UsePNGImageEncoder()
...
SaveImage(#IMAGE_MAIN, "Image.png", #PB_ImagePlugin_PNG)
```

Это четвертый пример сохранения изображения с названием Image.png (сохраняет в формате PNG).

Понимание экрана

Если вы когда-нибудь захотите создать свою собственную игру или написать свою собственную заставку с использованием PureBasic, то вы всегда должны начинать с открытия экрана. Экран этот попросту чисто графическая среда, созданная с одной целью, для отображения графики: таких как выход 2D рисунка, загруженных изображений, загруженных спрайтов, загруженных 3D моделей и миров.

Что такое Sprite(спрайт)?

Спрайты изначально использовались при специальном аппаратном ускорении изображения, и применялись для создания меняющейся 2D-графики в компьютерных играх. Так как вычислительная мощность компьютеров увеличилась на протяжении многих лет, специальное оборудование, которое использовалось для перемещения и использования этих изображений стало не нужным. Тем не менее, даже сегодня, название **Спрайт** до сих пор остается в употреблении для описания 2D изображений нарисованных на экране для создания игр и тому подобного. Сегодня спрайт можно охарактеризовать как небольшую графическую (обычно содержащую прозрачный фон) картинку, которая может быть размещена и нарисована независимо от экрана, чтобы имитировать анимацию или для предоставления статической графики.

Экраны, как правило, открыты полностью на всю ширину и высоту экрана, но если есть необходимость, можно также открыть в созданном окне. Это называется оконный режим. Только один тип экрана может быть открыт в одно время, либо полный экран, либо экран с оконным режимом. Причина, по которой экраны предпочитают для отображения графики (а не просто вывод рисунка на окна) для создания игр и т.п., является то, что экраны быстры, ... очень быстры! На каждой платформе в PureBasic, доступны ее экраны и оптимизированы для максимально возможной производительности, вне зависимости от операционной системы.

Открытие первого экрана

Чтобы создавать и открывать экран в PureBasic нужно следовать основному шаблонному коду. Экран и обработка спрайтов очень тесно связаны, так что вы всегда должны инициализировать механизм спрайта до открытия экрана. Необходимо, чтобы все было правильно инициализировано и после этого экран будет готов к операции рисования. После того, как механизм спрайта был инициализирован и экран открыт, необходимо, чтобы главный цикл обеспечивал осуществление этой программы, так же как и у графического интерфейса программы. Код ниже показывает скелет программы **экран** со всеми решающими частями: инициализация механизма спрайта, открытие экрана, создание основного цикла и инициализация команд клавиатуры, для того, чтобы обеспечить закрытие программы.

```
Global Quit.b=#False
;Simple error checking procedure
Procedure HandleError(Result.l, Text.s)
 If Result=0
```

```
    MessageRequester("Error", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure
HandleError(InitSprite(), "InitSprite() command failed.")
HandleError(InitKeyboard(), "InitKeyboard() command failed.")
HandleError(OpenScreen(1024, 768, 32, "Fullscreen"), "Could not open screen.")
Repeat
  ClearScreen(RGB(0, 0, 0))
  ;Drawing operations go here
  FlipBuffers(2)
  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Quit=#True
  EndIf
Until Quit=#True
End
```

Если вы посмотрите в раздел справки (Helpfile:ReferenceManual->2D Games Libraries->Sprite & Screen) вы сможете найти более подробную информацию обо всех новых командах, используемых в этом примере. В этом примере на самом деле ничего не происходит, кроме открытого пустого экрана. Для выхода из программы достаточно нажать Esc на клавиатуре компьютера. Этот пример является скелетом всех игр, созданных для экрана, так что остановимся здесь для детального изучения. Для начала я создал простую процедуру проверки ошибок, как описано в главе 8. Она отлавливает ошибки при установке команд: InitSprite() InitKeyboard() OpenScreen() Если какая то команда вернет ноль, произойдет выход из программы. При возникновении сбоя по какой-либо причине, всегда лучше закрыть программу и информировать пользователя о проблеме. В противном случае, если ваша программа будет продолжаться, есть риск крупной аварии. Команды, необходимые для инициализации механизма спрайта и клавиатуры InitSprite() и InitKeyboard(). Помните, что нам нужно инициализировать механизм спрайта, прежде чем открывать экран. Команда OpenScreen() принимает четыре параметра, это ширина, высота, глубина цвета экрана, а также текст(его подпись). Эта подпись будет отображаться в панели задач, если экран будет минимизирован.

Использовать клавиатуру?

В этих примерах я использовал несколько клавиатурных команд для инициализации клавиатуры и обнаружения клавиш. Все описания можно найти в справке (Helpfile:Reference Manual->GeneralLibraries->Keyboard). Это очень маленькая и простая в использовании библиотека команд, так что вы должны легко ее разобрать.

Команда InitKeyboard() (инициализация клавиатуры) должна вызываться перед любой другой командой клавиатуры. Команда ExamineKeyboard () обновляет состояние клавиатуры. Команды KeyboardReleased() и KeyboardPushed() возвращают истину, если данная клавиша отжата, либо нажата соответственно. Эти команды задаются со встроенными константами, отвечающими за клавиши. Полный список найдете в справке.

Параметры ширина, высота и глубина цвета очень важны, так как они определяют, какой размер и глубина цвета будут у экрана. Независимо от компьютера перед запуском программы следует убедиться, что компьютер в состоянии поддерживать нужный размер и глубину цвета. Значения ширины и высоты вместе известны как разрешение, и это

должно быть поддержано полностью за счет видеокарты и монитора, иначе команда OpenScreen() вернет ошибку. В этом примере я использовал значения, которые поддерживают большинство компьютеров, так что не должно быть никаких проблем. Рисунок ниже показывает, разрешения экрана и глубину цветопередачи, которые должны работать практически на любом современном компьютере. Вы знает уже наверно, что компьютерные игры с большим разрешением экрана, имеют более четкое отображение, но поскольку приходится рисовать больше пикселей, это может замедлять программу. Кроме того чем выше глубина цветопередачи, тем более реалистичная графика может быть отображена.

Common Screen Resolutions

Width	Height	Bit Depths
640	480	8, 16 & 32
800	600	8, 16 & 32
1024	768	8, 16 & 32

Конечно, лучше перед использованием программы проверить способность компьютера, использовать эти значения. Вы можете получить список различных разрешений и глубину цветопередачи, которые может использовать компьютер, запустив следующий код:

```
InitSprite()
If ExamineScreenModes()
  While NextScreenMode()
    Width.l=ScreenModeWidth()
    Height.l=ScreenModeHeight()
    BitDepth.l=ScreenModeDepth()
    Debug Str(Width)+" x "+Str(Height)+" x "+Str(BitDepth)
  Wend
EndIf
```

Опять же в этом фрагменте мы инициализируем механизм спрайта , прежде чем использовать команду ExamineScreenModes(). Теперь все правильно, и мы можем использовать основные команды экрана. Далее я использую цикл While-Wend, в котором с помощью команд ScreenModeWidth() ScreenModeHeight() ScreenModeDepth(), получаю каждый поддерживаемый экран и вывожу информацию в отладочное окно. Это простой пример для понимания, больше читайте в справке (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen).

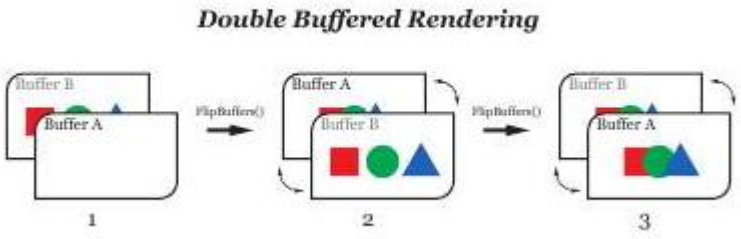
Двойной буферизированный рендеринг

Использование отображения графики необходимо связывать с основным циклом, чтобы обеспечить осуществление этой программы. Рисование графики, тестирование пользовательского ввода и т.д., должно следовать с использованием шаблона экрана. Вот основной цикл взят из скелета программы **Экран,** описанной выше:

```
...
Repeat
  ClearScreen(0)
  ;Drawing operations go here
```

```
  FlipBuffers(2)
  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Quit = #True
  EndIf
Until Quit = #True
...
```

На первый взгляд кажется нормальный основной цикл, как и в любой другой программе, но есть несколько различий. Есть две команды очень важные, для правильной организации графики экрана ClearScreen() и FlipBuffers(). Прежде чем я расскажу о них более подробно, я должен пояснить о двойной буферизации. Двойная буферизация используется экраном PureBasic для избежания некачественного отображения графики и мерцающего дисплея. Поскольку компьютерные мониторы постоянно перерисовывают экран монитора обычно шестьдесят-семьдесят (иногда больше) раз в секунду, трудно вносить изменения на экране, такие как перемещение или показ новой графики, прежде завершения имеющейся. Это вызывает результаты поврежденных графических эффектов и других плохих ненужных визуальных эффектов. Если вы хотите избежать этой проблемы, нужно очищать экран каждый раз перед перерисовкой экрана, он будет убирать ненужную графику, но появится мерцание экрана. Использование двойной буферизации PureBasic решает эти проблемы. Когда экран открыт, автоматически назначаются два видео-буфера в памяти, точно такого же размера, как открытый экран. Когда вы рисуете графику на экране, используя команды для графики, на самом деле вы это делаете в заднем буфере, в то время как передний буфер отображает на экране. Когда монитор закончил показывать передний буфер и операции с рисунком закончены, заполнив задний буфер, они меняются местами. Теперь передний буфер становится назад, а задний будет отображаться. Тот буфер, который опять задан на рисование, перед этим полностью очищается стандартной процедурой для очистки буфера, используя один цвет (обычно черный), прежде чем начать рисовать на ней еще раз. Рисунок ниже показывает эту технику в действии. Показ трех циклов буферов:



1) Открытие экрана и начинается рисование в заднем буфере. (Передний буфер отображается на экране)

2) Буферы поменялись. Передний буфер теперь сзади очищается и перерисовывает обновленную графику. (Задний буфер отображается на экране)

3) Буферы опять поменялись местами

На этой диаграмме вы могли увидеть, что при запуске программы (обозначается шаг 1), передний буфер (Buffer A) не имеет графики и ничего не отображается на экране. Когда мы рисуем, мы делаем это на задний буфер, обозначенный(Buffer B). После того, как все нарисовано, мы можем вызвать команду FlipBuffers() и буферы поменяются местами. Это показано в шаге 2. Теперь (Buffer B) отображается на экране, а мы продолжаем рисовать в переднем буфере. Опять же, когда все операции с рисунком проделаны, мы можем перевернуть буферы местами еще раз, используя команду FlipBuffers() и мы оказываемся на шаге 3, и так можно проделывать циклически. При создании иллюзии движения, мы должны будем очищать и перерисовать нужный рисунок, прежде использования FlipBuffers(). Для удаления старой графики мы используем команду ClearScreen(), она принимает один параметр: 24 битное значение цвета, для очистки экрана. Это можно получить, используя команду RGB(). Команда FlipBuffers() (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen->FlipBuffers) также имеет необязательный параметр и может принимать следующие значения:

0: Отключение вертикальной синхронизации монитора

1: Включение вертикальной синхронизации монитора(по умолчанию)

2: Включение вертикальной синхронизации монитора, использование режима экономии процессора (только в полноэкранном режиме).

Если 0 в качестве параметра, то FlipBuffers() меняет местами буферы так быстро, как от него зависит для достижения наивысшей частоты кадров. Единственным негативом для этого является то, что частота обновления монитора может быть не достаточно быстра, чтобы показывать новую графику в реальном времени, так что некоторые визуальные разрывы могут произойти, поскольку монитор все же пытается идти в ногу с перевертыванием буферов.

Если 1 используется в качестве параметра, команда FlipBuffers() перевертывает буферы в полной синхронизации с частотой обновления монитора, чтобы вся графика рисовалась правильно и согласовываясь с монитором. Единственным недостатком является то, что частота кадров не может превышать за частоту обновления монитора. Это режим по умолчанию, если нет параметра.

Использование 2 в качестве параметра, будет иметь тот же эффект, как использование 1 , но будет энергосберегающий режим процессора, не дающий процессору работать на сто процентов, кроме того другие программы не имеют возможности выполняться.

Рисование на экран

Вот пример с двойной буфферизацией и анимацией созданного рисунка, обновление графики с помощью команды FlipBuffers() (выход из этой программы, нажатие Esc на клавиатуре)

```
#IMAGE_MAIN=1
;Set the width, height and bit depth of the screen
;Abbreviated variables are used here due to page width constraints :(
Global ScrW.l=1024
Global ScrH.l=768
Global ScrD.l=32
Global Quit.b=#False
```

```
XOrigin.f=(ScrW/2)-64 : YOrigin.f=(ScrH/2)-64
;Simple error checking procedure


Procedure HandleError(Result.l, Text.s)
   If Result=0
      MessageRequester("Error", Text, #PB_MessageRequester_Ok)
      End
   EndIf
EndProcedure


;Initialize environment
HandleError(InitSprite(), "InitSprite() command failed.")
HandleError(InitKeyboard(), "InitKeyboard() command failed.")
HandleError(OpenScreen(ScrW, ScrH, ScrD, "Blobs"), "Could not open screen.")
SetFrameRate(60)
;Create an image
If CreateImage(#IMAGE_MAIN, 128, 128)
   If StartDrawing(ImageOutput(#IMAGE_MAIN))
      For x.l=255 To 0 Step-1
         Circle(64, 64, x/4, RGB(0, 0, 255-x))
      Next x
      StopDrawing()
   EndIf
EndIf
;Convert Degrees to Radians


Procedure.f DegToRad(Angle.f)
   ProcedureReturn Angle.f*#PI/180
EndProcedure


;Main loop
Repeat
   ClearScreen(RGB(0, 0, 0))
   Angle.f+2.0
   Radius.f=((ScrH/2)-100)*Sin(DegToRad(Angle))
   StartDrawing(ScreenOutput())
      For x.l=0 To 359 Step 45
         XPos.f=XOrigin+(Radius*Cos(DegToRad(Angle+x)))
         YPos.f=YOrigin+(Radius*Sin(DegToRad(Angle+x)))
         DrawImage(ImageID(#IMAGE_MAIN), XPos, YPos)
      Next x
   StopDrawing()
   FlipBuffers()
   ExamineKeyboard()
   If KeyboardReleased(#PB_Key_Escape)
      Quit=#True
   EndIf
```

```
Until Quit=#True
End
```

В этом примере я создал новый образ, используя команду CreateImage(), а потом вывожу этот образ на экран, используя блок StartDrawing(). Хотя этот код выглядит немного сложнее, особенно при расчетах X и У значений, этот пример является чисто, чтобы продемонстрировать переворот буферов.

Вы можете увидеть в основном цикле, что первое, что я делаю, очищаю экран с помощью команды ClearScreen(). Это дает мне возможность начать на четком буфере, так чтобы ни одна старая графика, не осталась от предыдущей операции рисования. После этого я использую математику, чтобы вычислить новые координаты моих пятен на рисунке и использую цикл для их рисования. После того как они нарисованы, они находятся на заднем буфере, так что я должен использовать команду FlipBuffers(), чтобы отобразить их на экране. И так далее, очистка, оформление и отражение, а также между каждым сальто, я могу изменять положение графики.

Вы могли заметить еще одну новую команду, которую я использовал в этом примере, это SetFrameRate() (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen->SetFrameRate). Эта команда имеет один параметр, который определяет какое количество раз FlipBuffers() может быть выполнен в секунду. Это создание стандартной частоты кадров на других компьютерах, которые могли бы запустить этот код. Данный пример ограничивает обновление для отображения графики до шестидесяти раз в секунду. Вся анимация произведенная на компьютерах, очень похожа на мультфильмы или кинофильмы. Ничего фактически не движется на экране, это лишь демонстрация слайдов различных изображений (буфера), в которых графика находится в немного различных позициях. Поскольку все это случается очень быстро (шестьдесят раз в секунду), нам кажется, что изображения начинают двигаться.

Простое звездное небо

Это один из эффектов для игр или демосцен, каждый программист в принципе знает, как рисовать подобное. Звездное небо, примерно такое как это, было использовано в сотнях игр и демосцен, поскольку здесь есть эффект перемещения в пространстве. Это эффект, который оживляет сотни пикселей на экране, используя разные оттенки цвета, создает иллюзию глубины и движения. Существуют различные пути программирования этого эффекта, а вот мой пример:

```
#APP_NAME="Stars v1.0"
#NUMBER_OF_STARS=10000
;Set the width, height and bit depth of the screen
;Abbreviated variables are used here due to page width constraints :(
Global ScrW.l=1024
Global ScrH.l=768
Global ScrD.l=32
Global Quit.b=#False


Structure STAR
   xPos.f
   yPos.f
   xStep.f
```

```
    Color.l
EndStructure


Global Dim Stars.STAR(#NUMBER_OF_STARS)
;Simple error checking procedure


Procedure HandleError(Result.l, Text.s)
    If Result=0
        MessageRequester("Error", Text, #PB_MessageRequester_Ok)
        End
    EndIf
EndProcedure


;init stars


Procedure InitializeStars()
    For x=0 To #NUMBER_OF_STARS
        Stars(x) \xPos=Random(ScrW-1)
        Stars(x) \yPos=Random(ScrH-1)
        If x<#NUMBER_OF_STARS/3
            Stars(x) \xStep=(Random(10)/100)+0.2
            Stars(x) \Color=RGB(40, 40, 40)
        ElseIf x>=#NUMBER_OF_STARS/3 And x<(#NUMBER_OF_STARS/3)*2
            Stars(x) \xStep=(Random(10)/100)+0.6
            Stars(x) \Color=RGB(100, 100, 100)
        Else
            Stars(x) \xStep=(Random(10)/100)+1.2
            Stars(x) \Color=RGB(255, 255, 255)
        EndIf
    Next x
EndProcedure


;move stars on the 'x' axis


Procedure MoveStarsX()
    For x=0 To #NUMBER_OF_STARS
        Stars(x) \xPos-Stars(x) \xStep
        If Stars(x) \xPos<0
            Stars(x) \xPos=ScrW-1
            Stars(x) \yPos=Random(ScrH-1)
        EndIf
    Next x
EndProcedure


;Initialize environment
```

```
HandleError(InitSprite(), "InitSprite() command failed.")
HandleError(InitKeyboard(), "InitKeyboard() command failed.")
HandleError(OpenScreen(ScrW, ScrH, ScrD, #APP_NAME), "Could not open screen.")
SetFrameRate(60)
InitializeStars()

Repeat
   ClearScreen(RGB(0, 0, 0))
   StartDrawing(ScreenOutput())
      For x=0 To #NUMBER_OF_STARS
         Plot(Stars(x) \xPos, Stars(x) \yPos, Stars(x) \Color)
      Next x
      DrawingMode(#PB_2DDrawing_Transparent)
      DrawText(20, 20, #APP_NAME, #White)
      DrawText(20, 40, Str(#NUMBER_OF_STARS)+" Animated stars", #White)
      DrawText(20, 60, "Screen Resolution: "+Str(ScrW)+" x "+Str(ScrH), #White)
      DrawText(20, 80, "Screen Bit depth: "+Str(ScrD)+"bit", #White)
   StopDrawing()
   FlipBuffers()
   MoveStarsX()
   ExamineKeyboard()
   If KeyboardReleased(#PB_Key_Escape)
      Quit=1
   EndIf
Until Quit=1
End
```

Этот пример использует 2D команду рисования Plot(), для рисования точек (Helpfile:Reference Manual->General Libraries->2DDrawing->Plot). Эта команда использует три параметра, из которых третий не является обязательным. Этими параметрами являются X и У расположение, и третий цвет пикселя. Если последний параметр не используется, то эта команда использует цвет по умолчанию, который устанавливается с помощью команды FrontColor() . В приведенном выше примере, я использовал структуру, чтобы сохранить все сведения об одном пикселе вместе. Затем создается массив переменных использующий эту структуру. Каждый из элементов массива описывает положение, цвет и величину шага всех отдельных пикселей. Затем я с помощью цикла прохожу по массиву, получая каждый рисуемый мной пиксель на экране, используя информацию о точках (положение, цвет и т.д.), содержащихся внутри каждого структурированного элемента массива. После того как рисунок завершен, я переварачиваю буферы и обновляю позиции пикселей в массиве, используя соответствующие значения шага. Как только это будет сделано, я очищаю экран и перерисовываю, ... и так далее. Используя такое положение вещей в коде, делает код понятным, запоминающимся и позволяет его легко обновлять позднее.

Открытие экрана в окне

Иногда вы хотели бы открыть экран в окне, особенно если вы хотите, чтобы игра или демо были оконными. Вы можете сделать это, используя команду OpenWindowedScreen(). Чтобы открыть экран в окне, вы должны сначала создать окно, а затем обрабатывать события из этого окна в главном цикле, а также рисунок. Вот пример использования оконного экрана:

```
#WINDOW_MAIN=1
#IMAGE_MAIN=1
;Set the width, height and bit depth of the screen
;Abbreviated variables are used here due to page width constraints :(
Global ScrW.l=800
Global ScrH.l=600
Global ScrD.l=32
Global Quit.b=#False
Global XOrigin.f=(ScrW/2)-64
Global YOrigin.f=(ScrH/2)-64
;Simple error checking procedure


Procedure HandleError(Result.l, Text.s)
   If Result=0
      MessageRequester("Error", Text, #PB_MessageRequester_Ok)
      End
   EndIf
EndProcedure


;Convert Degrees to Radians


Procedure.f DegToRad(Angle.f)
   ProcedureReturn Angle.f*#PI/180
EndProcedure


;Initialize environment
HandleError(InitSprite(), "InitSprite() command failed.")
HandleError(InitKeyboard(), "InitKeyboard() command failed.")
#FLAGS=#PB_Window_SystemMenu | #PB_Window_ScreenCentered


If OpenWindow(#WINDOW_MAIN, 0, 0, ScrW, ScrH, "Windowed Screen", #FLAGS)
   If OpenWindowedScreen(WindowID(#WINDOW_MAIN), 0, 0, ScrW, ScrH, 0, 0, 0)
      SetFrameRate(60)
      ;Create an image
      If CreateImage(#IMAGE_MAIN, 128, 128)
         If StartDrawing(ImageOutput(#IMAGE_MAIN))
            For x.l=255 To 0 Step-1
               Circle(64, 64, x/4, RGB(255-x, 0, 0))
            Next x
            StopDrawing()
         EndIf
      EndIf
      ;Main loop
      Repeat
         Event.l=WindowEvent()
         ClearScreen(RGB(0, 0, 0))
         Angle.f+2.0
```

```
        Radius.f=((ScrH/2)-100)*Sin(DegToRad(Angle))
        StartDrawing(ScreenOutput())
          For x.l=0 To 359 Step 45
            XPos.f=XOrigin+(Radius*Cos(DegToRad(Angle+x)))
            YPos.f=YOrigin+(Radius*Sin(DegToRad(Angle+x)))
            DrawImage(ImageID(#IMAGE_MAIN), XPos, YPos)
          Next x
        StopDrawing()
        FlipBuffers()
        ExamineKeyboard()
        If KeyboardReleased(#PB_Key_Escape)
          Quit=#True
        EndIf
      Until Event=#PB_Event_CloseWindow Or Quit=#True
    EndIf
  EndIf
  End
```

Этот пример должен быть для вас понятным, потому что большую часть этого кода вы видели раньше. Основным отличием является команда OpenWindowedScreen(), которая принимает восемь параметров! Первым параметром является идентификатор ОС окна, в котором графика должна быть открыта. Второй и третий параметры X и У позиция нового экрана на окне. Четвертый и пятый параметры ширина и высота нового экрана. Шестой параметр автоматический размер или auto-stretch. Если этот параметр установлен в 0, то нет автоматического изменения размера, но если он установлен в 1 , то экран автоматически изменяет размер самого себя до максимального размера окна. Это позволит, не обращать внимания на параметры четыре и пять. Это означает, что даже если, мы будем изменять размер родительского окна, экран будет автоматически изменять свой размер, чтобы заполнить всегда всю область окна. Седьмой и восьмой параметры можно определять, когда auto-stretch включен. Они будут тянуть автоматическую калибровку экрана немного назад, с правой стороны или снизу, соответственно, оставив место для строки состояния или других устройств в окно. Кроме того, очень важно понимать при использовании экрана в окне: нужно правильно вести обработку событий и здесь как раз подойдет команда WindowEvent(). Если вы помните из главы 9, эта команда не ждет пока событие произойдет, а она всегда пытается обнаружить и вернуть любые события, которые требуют обработки. Использование этой команды является обязательным. Если мы будем использовать команду WaitWindowEvent(), то это остановит переключения буферов для любого промежутка времени.

Спрайты

Спрайты в PureBasic это образы, которые можно отобразить в любое место на экране. В отличие от изображений, спрайты имеют свой собственный набор команд, оптимизированы по скорости и могут быть использованы для достижения многих специальных эффектов. Все команды обыкновенных спрайтов вы можете почитать и изучить в справке (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen), но существуют еще так называемые 3D спрайты, о которых можно почитать (Helpfile:Reference Manual->2DGames Libraries->Sprite3D). Для ознакомления со спрайтами , советую изучить оба раздела.

Разница между простыми спрайтами и 3D спрайтами.

В PureBasic есть два разных типов спрайтов. Первый, это обыкновенный спрайт из загруженного изображения, манипулировать которым нужно, используя стандартную библиотеку спрайтов. Второй тип почти тоже самое, за исключением того, что PureBasic использует небольшой 3D движок для отображения 3D спрайтов. Кроме того, команды, используемые для рисования и управления этим типом спрайта, позволяет программисту достигнуть графических эффектов, которые невозможны с простыми типами спрайтов. Эти эффекты включают масштабирование в реальном времени, трехмерное преобразование и плавное сопряжение спрайта. 3D-движок, который выполняет преобразования и отображения 3D-спрайтов, это не движок OGRE, о чем говорится в следующей главе. Это небольшой 3D-движок, специально созданный для манипуляций с данным типом спрайта.

Использование обычных спрайтов

Для создания спрайтов, используемых в вашей программе, вы можете либо загрузить готовый с помощью команды LoadSprite( ) (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen->LoadSprite), либо создать новый, используя команду CreateSprite() (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen->CreateSprite).

Чтобы загрузить существующие изображения, которые будут использоваться в качестве спрайтов, надо использовать команду LoadSprite(). Она очень похожа на команду LoadImage() и так же принимает три параметра. Первый идентификатор(номер PB), который будет закреплен за данным спрайтом. Второй, это имя файла, которое хотите загрузить, помня, конечно, про использование декодеров изображения в случае необходимости. Третий, это режим спрайта, который определяет, как это спрайт будет использоваться, подробнее об этом чуть позже. Для создания своего собственного спрайта, надо использовать 2D команды рисования, но перед этим его надо создать с помощью команды CreateSprite(). Будучи очень похожа на команду CreateImage(), команда CreateSprite() имеет четыре параметра. Первый параметр, это идентификатор, который закрепится за этим спрайтом. Второй и третий параметры это ширина и высота в пикселях нового спрайта, а четвертый параметр это режим спрайта. Эти две команды создают новый спрайт и оба имеют необязательный параметр-режим спрайта. Этот режим определяет формат внутреннего спрайта, для правильного использования других команд, с ним связанных. Этот формат обычно определяет встроенная константа, рисунок ниже показывает эти константы для каждого из режимов, а также описание каждого режима. В зависимости от команд применяемых в последствии для спрайта, вы должны использовать правильный формат спрайтов, который устанавливается при создании или загрузки спрайта.

Режимы спрайтов

Используемые константы	Описание режима
Нет	Режим по умолчанию. Спрайт загружается в видеобуфер (если возможно).
#PB_Sprite_Memory	Спрайт загружается в нормальную оперативную память, а не в видеобуфер, для использования с командой StartSpecialFX()
#PB_Sprite_Alpha	У спрайта должна быть 8 битовая градация серого цвета для использования с командами DisplayAlphaSprite() или DisplayShadowSprite(). При использовании команды

	StartSpecialFX() в данном режиме, необходимо указать так же константу #PB_Sprite_Memory
#PB_Sprite_Texture	Спрайт создается с поддержкой 3D, так что вы можете создавать 3D спрайт, используя команду CreateSprite3D()
#PB_Sprite_AlphaBlending	Спрайт создается с поддержкой альфа-канала. Загруженные изображения должны содержать альфа-канал. В настоящее время для PureBasic альфа-канал поддерживается только в форматах Png и Tiff. (Если вы собираетесь преобразовать эти спрайты в 3D-спрайты,нужно будет так же указать константу PB_Sprite_Texture)

Режимы могут быть объеденены как обычно с помощью побитового оператора |

Вот несколько примеров того, как правильно определить режим спрайта для каждой команды дисплея. Я так же дал пример того, как создать и загрузить спрайт с правильными настройками режима, поэтому он(спрайт) будет отображаться правильно при использовании команд.

DisplaySprite() & DisplayTransparentSprite()

```
;Default format
CreateSprite(#PB_NUMBER, Width.l, Height.l)
LoadSprite(#PB_NUMBER, "Image.bmp")
```

DisplaySprite3D()

```
;Without an alpha channel
CreateSprite(#PB_NUMBER, Width.l, Height.l, #PB_Sprite_Texture )
LoadSprite(#PB_NUMBER, "Image.bmp", #PB_Sprite_Texture )
;With an alpha channel
;NOTE: You can’t create sprites with an alpha channel in PureBasic yet.
LoadSprite(#PB_NUMBER,"Image.bmp",#PB_Sprite_Texture|#PB_Sprite_AlphaBlendin
g)
```

DisplayTranslucentSprite()

```
;Load in normal RAM for processing by the 'StartSpecialFX()' command
CreateSprite(#PB_NUMBER, Width.l, Height.l, #PB_Sprite_Memory)
LoadSprite(#PB_NUMBER, "Image.bmp", #PB_Sprite_Memory)
```

DisplayAlphaSprite()’, ‘DisplayShadowSprite()’ & ‘DisplaySolidSprite()

```
;Load in normal RAM for processing by the 'StartSpecialFX()' command
;and specify as an alpha type sprite
CreateSprite(#PB_NUMBER, Width.l, Height.l, #PB_Sprite_Memory|#PB_Sprite_Alpha)
LoadSprite(#PB_NUMBER, "Image.bmp", #PB_Sprite_Memory | #PB_Sprite_Alpha)
```

Эти примеры должны дать вам хорошее представление о том, как загружать и создавать свои собственные спрайты для каждой отображенной команды.

Вы можете рисовать на спрайтах тоже.

После того как вы создали или загрузили спрайт, вы можете использовать его. Так же как и с изображениями, это делается с помощью стандартной команды StartDrawing() с параметром SpriteOutput(). Это позволяет использовать 2D команды на поверхность спрайта. Вы можете даже нарисовать один спрайт на другой. Для этого вы должны переключать спрайт из заднего буфера на ваш целевой спрайт. Это достигается с помощью команды UseBuffer(). Эта команда имеет один параметр-идентификатор того спрайта, на который вы хотите рисовать. После подключения рисования с помощью UseBuffer(), все команды рисования применяются для рисования на спрайт. Когда вы все нарисовали на спрайте, и вам нужно вернуть возможность рисовать на основном экране, примените опять команду UseBuffer(), только на этот раз с параметром #PB_Default.

Команды спрайта SpecialFX

В таблице с режимами, я уже упоминал о спрайтах SpecialFX. Это обыкновенные спрайты, но работают гораздо быстрее, если вставлять их в блок команд StartSpecialFX().....StopSpecialFX(). Все спрайты, которые используют спецэффекты, способны производить красивую графику, но поскольку все это происходит внутри основной оперативной памяти компьютера, эти спрайты не так быстры, как 3D спрайты. Вот небольшой фрагмент текста, который показывает использование этих команд:

```
...
StartSpecialFX()
  DisplayAlphaSprite(#SPRITE_NUMBER, 64, 64)
  DisplayRGBFilter(100, 100, 100, 100, 0, 255, 0)
  DisplayTranslucentSprite(#SPRITE_NUMBER, 128, 128, 128)
  ;etc...
StopSpecialFX()
...
```

Как вы можете видеть, команда StartSpecialFX() начинает блок, а команда StopSpecialFX() заканчивает. Все команды, связанные со спецэффектами должны идти внутрь этого блока. Это сделано для увеличения скорости рендеринга спецэффектов, в противном случае, без StartSpecialFX() производительность отображения будет очень низкая. Если вы используете эти команды, очень важно понимать, что все команды графики должны быть в этом блоке. В противном случае, если вы используете другие команды графики до блока спецэффектов, они будут рисовать в заднем буфере. Если вы используете команду ClearScreen() для очистки буфера перед рисованием, вы должны включить ее в блок StartSpecialFX() тоже. Еще один важный момент, желательно использовать только один блок StartSpecialFX() в любом основном цикле, так как это увеличивает производительность.

Команды, которые используют блок StartSpecialFX():

```
DisplayAlphaSprite()
DisplaySolidSprite()
```

```
DisplayShadowSprite()
DisplayRGBFilter()
DisplayTranslucentSprite()
```

Вы можете прочитать об этих командах более подробно в справке (Helpfile:Reference Manual->2D Games Libraries->Sprite & Screen).

Отображение простых спрайтов

Простые спрайты лежат в основе двумерных игр в PureBasic. Этот тип спрайта был использован снова и снова, раз за разом, производя "холодные" изображения во многих самых популярных играх. Вот простой пример создания новых спрайтов и отображения их на экране с помощью стандартной команды DisplayTransparentSprite():

```
#SPRITE_MAIN = 1
#NUMBER_OF_BALLS = 500
;Set the width, height and bit depth of the screen
;Abbreviated variables are used here due to page width constraints :(
Global ScrW.l = 1024
Global ScrH.l = 768
Global ScrD.l = 32
Global Quit.b = #False

Structure BALL
   x.f
   y.f
   XOrigin.l
   YOrigin.l
   Radius.l
   Angle.f
   Speed.f
EndStructure


Global Dim Balls.BALL(#NUMBER_OF_BALLS)
;Simple error checking procedure


Procedure HandleError(Result.l, Text.s)
   If Result = 0
      MessageRequester("Error", Text, #PB_MessageRequester_Ok)
      End
   EndIf
EndProcedure


;Convert Degrees to Radians


Procedure.f DegToRad(Angle.f)
   ProcedureReturn Angle.f * #PI / 180
EndProcedure
```

```
;Initialize all ball data


Procedure InitialiseBalls()
   For x.l = 0 To #NUMBER_OF_BALLS
      Balls(x)\XOrigin = Random(ScrW) - 32
      Balls(x)\YOrigin = Random(ScrH) - 32
      Balls(x)\Radius = Random(190) + 10
      Balls(x)\Angle = Random(360)
      Balls(x)\Speed = Random(2) + 1
   Next x
EndProcedure


;Initialize environment
HandleError(InitSprite(), "InitSprite() command failed.")
HandleError(InitKeyboard(), "InitKeyboard() command failed.")
HandleError(OpenScreen(ScrW, ScrH, ScrD, "Blobs"), "Could not open screen.")
SetFrameRate(60)
;Create an image
Global Offset.f = 32
If CreateSprite(#SPRITE_MAIN, 64, 64)
   If StartDrawing(SpriteOutput(#SPRITE_MAIN))
      Box(0, 0, 64, 64, RGB(255, 255, 255))
      For x.l = 220 To 1 Step -1
         Offset + 0.025
         Circle(Offset, 64 - Offset, x / 8, RGB(0, 255 - x, 0))
      Next x
      StopDrawing()
   EndIf
EndIf
TransparentSpriteColor(#SPRITE_MAIN, RGB(255, 255, 255))
InitialiseBalls()
;Main loop
Repeat
   ClearScreen(RGB(56, 76, 104))
   For x.l = 0 To #NUMBER_OF_BALLS
      Balls(x)\x=Balls(x)\XOrigin+(Balls(x)\Radius*Cos(DegToRad(Balls(x)\Angle)))
      Balls(x)\y=Balls(x)\YOrigin+(Balls(x)\Radius*Sin(DegToRad(Balls(x)\Angle)))
      Balls(x)\Angle + Balls(x)\Speed
      DisplayTransparentSprite(#SPRITE_MAIN, Balls(x)\x, Balls(x)\y)
   Next x
   FlipBuffers()
   ExamineKeyboard()
   If KeyboardReleased(#PB_Key_Escape)
      Quit = #True
   EndIf
Until Quit = #True
End
```

Команда DisplayTransparentSprite() позволяет отображать на экране спрайты. Она очень похожа на команду DisplaySprite(), но при выводе спрайтов с помощью DisplayTransparentSprit(), она выбирает один цвет на изображении и относит его к прозрачному цвету. Это позволяет не показывать всю площадь спрайта. В этом примере я создал новый спрайт, а затем тут же заполнил его белым цветом с использованием команды Box(). После этого я нарисовал затененную зеленую область с помощью команды Circle() в цикле. В итоге зеленая область на белом фоне. После команд рисования, я пометил все белые пиксели в новом спрайте как прозрачные, используя команду TransparentSpriteColor(). Эта команда имеет два параметра, первый идентификатор спрайта, а второй цвет, который мы хотим пометить прозрачным. Как только цвет взят для следующего использования, команда DisplayTransparentSprite() показывает спрайт, минус прозрачный цвет. После запуска примера, вы должны увидеть экран, заполненный зелеными сферами, без отображения белого фона. Это отличный способ показать спрайты с прозрачностью. Вы также заметили в этом примере, что для того, чтобы отобразить нормальный спрайт не нужны блоки команд: StartSpecialFX() или StartDrawing() и т.д. Вы можете использовать команды отображения обычного спрайта сами по себе. До тех пор, пока движок спрайта инициализируется и экран открыт, вы можете использовать стандартные команды для отображения спрайтов. Стандартные команды показывающие спрайты :

DisplaySprite()
DisplayTransparentSprite()

Использование 3D спрайтов

Каждый 3D спрайт это 2D поверхность, состоящая из двух многоугольников. Эти многоугольники рисуются с помощью небольшого 3D-движка и могут быть преобразованы в 3D, но каждый 3D спрайт в конечном счете используется на 2D на экране. Непонятно? Хорошо, давайте продолжим. 3D спрайт в PureBasic на самом деле простой спрайт, для которого оказывается поддержка 3D с помощью малого 3D движка, отображающего его. Чтобы использовать любой 3D спрайт в вашей программе, вы должны инициализировать 3D движок перед использованием любых связанных команд. Это делается с помощью команды InitSprite3D(). Эта команда, однако вызывается после обязательной команды InitSprite(). Каждый 3D спрайт начинает "жизнь" как нормальный спрайт. Превращение его в 3D происходит в два этапа. Первый это при загрузке устанавливается режим PB_Sprite_Texture#. Вторым этапом спрайт с помощью команды CreateSprite3D() как раз и превращается в 3D спрайт. Эта команда имеет два параметра. Первый это идентификатор нового 3D спрайта, который за ним закрепится. Второй параметр собственно тот 2D спрайт, из которого надо его создать. Вот как происходит преобразование простого спрайта в 3D спрайт:

```
LoadSprite(#NORMAL_SPRITE, "Image.bmp", #PB_Sprite_Texture)
CreateSprite3D(#SPRITE_3D, #NORMAL_SPRITE)
```

После того как 3D спрайт был создан, мы можем показывать это на экране, используя команду DisplaySprite3D(). Чтобы было все более или менее понятно, покажу на примере отображения и манипулирования 3D спрайта:

```
UsePNGImageDecoder()

Enumeration
  #SPRITE_2D
```

```
  #SPRITE_3D
EndEnumeration

#NUMBER_OF_FLOWERS = 150
;Set the width, height and bit depth of the screen
;Abbreviated variables are used here due to page width constraints :(
Global ScrW.l = 1024
Global ScrH.l = 768
Global ScrD.l = 32
;Other global variables
Global Quit.b = #False
Global XOrigin.l = ScrW / 2
Global YOrigin.l = ScrH / 2

Structure FLOWER
  XPos.f
  YPos.f
  Width.f
  Height.f
  Angle.f
  Radius.f
  RadiusStep.f
EndStructure

Global Dim Flowers.FLOWER(#NUMBER_OF_FLOWERS)
;Simple error checking procedure

Procedure HandleError(Result.l, Text.s)
  If Result = 0
    MessageRequester("Error", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Convert Degrees to Radians

Procedure.f DegToRad(Angle.f)
  ProcedureReturn Angle.f * #PI / 180
EndProcedure

;Initialize all flowers

Procedure InitialiseAllFlowers()
  For x.l = 0 To #NUMBER_OF_FLOWERS
    Flowers(x)\Width = 0
    Flowers(x)\Height = 0
    Flowers(x)\Angle = Random(360)
    Flowers(x)\Radius = 1.0
    Flowers(x)\RadiusStep = (Random(30) / 10) + 1.0
  Next x
EndProcedure
```

```
;Reset a flower

Procedure ResetFlower(Index.l)
  Flowers(Index)\Width = 0
  Flowers(Index)\Height = 0
  Flowers(Index)\Angle = Random(360)
  Flowers(Index)\Radius = 1.0
  Flowers(Index)\RadiusStep = (Random(30) / 10) + 1.0
  ProcedureReturn
EndProcedure

;Initialize environment

HandleError(InitSprite(), "InitSprite() command failed.")
HandleError(InitSprite3D(), "InitSprite3D() command failed.")
HandleError(InitKeyboard(), "InitKeyboard() command failed.")
HandleError(OpenScreen(ScrW, ScrH, ScrD, "Flowers"), "Could not open screen.")
SetFrameRate(60)
Sprite3DQuality(1)
;Load sprite
LoadSprite(#SPRITE_2D,"Flower.png",#PB_Sprite_Texture|#PB_Sprite_AlphaBlending)
CreateSprite3D(#SPRITE_3D, #SPRITE_2D)
InitialiseAllFlowers()
;Main loop

Repeat
  ClearScreen(RGB(200, 100, 100))
  HandleError(Start3D(), "Start3D() command failed.")
  For x.l = 0 To #NUMBER_OF_FLOWERS
    Flowers(x)\Width + 1.5
    Flowers(x)\Height + 1.5
    Flowers(x)\Angle + 1.0
    If Flowers(x)\Width > 512.0 Or Flowers(x)\Height> 512.0
       Flowers(x)\Width = 512.0
       Flowers(x)\Height = 512.0
    EndIf
    If Flowers(x)\Radius > ScrW
       ResetFlower(x)
    EndIf
    Flowers(x)\Radius + Flowers(x)\RadiusStep
    Flowers(x)\XPos=XOrigin+(Flowers(x)\Radius*Cos(DegToRad(Flowers(x)\Angle)))
    Flowers(x)\YPos=YOrigin+(Flowers(x)\Radius*Sin(DegToRad(Flowers(x)\Angle)))
    Flowers(x)\XPos - Flowers(x)\Radius / 3.5
    Flowers(x)\YPos - Flowers(x)\Radius / 3.5
    ZoomSprite3D(#SPRITE_3D, Flowers(x)\Width, Flowers(x)\Height)
    RotateSprite3D(#SPRITE_3D, Flowers(x)\Angle, 0)
    DisplaySprite3D(#SPRITE_3D, Flowers(x)\XPos, Flowers(x)\YPos)
  Next x
  Stop3D()
  FlipBuffers()
  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
```

```
    Quit = #True
  EndIf
Until Quit = #True
End
```

Чтобы иметь возможность использовать 3D спрайт в моем примере, я инициализировал два движка, один за другим с помощью команд InitSprite() и InitSprite3D(). Это крайне важно перед использованием любых команд, связанных с 3D спрайтами. После этого, я использовал команду LoadSprite() для загрузки изображения формата PNG. Это изображение с именем "Flower.png", используется с альфа-каналом для создания прозрачного фона. Загрузка изображения в формате "Png" с альфа-каналом, требует определения правильного режима для команды LoadSprite(). Если вы посмотрите на пример кода, я определил режим: # PB_Sprite_Texture | # PB_Sprite_AlphaBlending. Это сообщает компилятору,что я хочу использовать этот спрайт, как 3D спрайт, содержащий альфа-канал. После того, как он загружен и правильно определен, я могу создать 3D спрайт из него при помощи команды CreatSprite3D(), при этом альфа-канал сохраняется. После того, как спрайт был создан, настало время обратить его на экране. Это достигается с помощью команды DisplaySprite3D()внутри блока Start3D()...Stop3D() . Это выглядит примерно так:

```
...
Start3D()
DisplaySprite3D(#SPRITE_3D, x, y, Alpha)
Stop3D()
...
```

Если вы посмотрите на прошлый пример, вы увидите что блок Start3D() проверяется с помощью процедуры малых ошибок. Если все правильно инициализировано, мы можем продолжить и сделать спрайты. Если он запускается неправильно, мы не должны делать никаких 3D спрайтов, во избежание серьезной аварии программы. Команда Stop3D()используется для завершения блока команд 3D спрайта, которые в нем находятся .. Для показа 3D спрайта я использую команду DisplaySprite3D(), которая принимает четыре параметра. Первый идентификатор 3D спрайта. Второй и третий X и У координаты. Четвертый параметр это альфа значение спрайта, определяет прозрачность спрайта на экране. Это целое число в диапазоне от 0, которое является полностью прозрачным, до 255, которое полностью непрозрачное. В примере я также использовал команды: ZoomSprite3D() и RotateSprite3D() для изменения размера и поворота 3D спрайта. Эти команды несложно понять, и прочесть о них более подробно, можно в справке (Helpfile:Reference Manual->2D GamesLibraries->Sprite3D).

Качество 3D спрайтов

При использовании 3D спрайтов в вашей программе, есть возможность переключать качество спрайтов. Это делается с помощью команды Sprite3DQuality(). Эта команда имеет один параметр, которым является режим рендеринга всех 3D спрайтов в вашей программе. Параметр представляет собой числовое значение:

0: Нет фильтрации (быстро, но очень неровно при масштабировании и наклоне)

1: Билинейная фильтрация (медленно, но смешивает пиксели для более качественного просмотра при масштабировании и наклоне)

В моем примере я использовал команду Sprite3DQuality(1), которая позволит билинейную фильтрацию и дает спрайтам хороший ровный взгляд, когда я меняю размеры и их ротацию. Когда эта команда не используется (по умолчанию настройки качества для программ 0), получается отсутствие фильтрации, в результате чего 3D спрайты выглядят не так гладко, при манипуляциях с ними.

10

Sound

На определенном этапе при программировании, вам может понадобиться звуки, которые играли в программе. Это может быть звуковым перезвон, который позволяет пользователю, что задача завершена, или это может быть звуковые эффекты в игре. Однако вы используете звук, вы должны знать, как загружать и воспроизводить звуковые файлы частности. В данной главе объясняется, как загружать различные различные звуковые файлы, и приводятся примеры о том, как играть эти звуковые файлы из программы.

Wave файл

Wave файлы один из наиболее распространенных форматов звука на персональных компьютерах, в связи с их создания объединить усилия между Microsoft и IBM . Wave файлов, которые обычно имеют расширение файла *. WAV, являются родной формат звука используется всеми компьютерами. Хотя этот формат не имеет встроенной сжатия звуковых данных, это до сих пор используется для повседневных целей.

Следующий пример загружает звуковой файл называется Intro.wav и играет ее:

```
#SOUND_FILE = 1
 If InitSound()
    LoadSound(#SOUND_FILE, "Intro.wav")
    PlaySound(#SOUND_FILE)
    StartTime.l = ElapsedMilliseconds()
    Repeat
     Delay(1)
    Until ElapsedMilliseconds() > StartTime + 8000
 End
EndIf
```

Этот пример, вероятно, не будут использоваться в реальной программе, но она показывает, правильные шаги, необходимые для воспроизведения звуковой файл. Во-первых, мы должны инициализировать звуковой среды, используя InitSound() команды. Это правильно инициализирует все аппаратные средства и программное обеспечение, необходимое для воспроизведения файла. Если это возвращает истину, мы знаем, что он инициализирован должным образом, и мы можем продолжить. Там нет смысла продолжать со звуком, если звук инициализации сбой, компьютер, вероятно, не звуковой картой.

После инициализации сделано и проверено мы загружаем звуковой файл, используя LoadSound() команды (Helpfile: Справочное руководство -> 2D игры Библиотеки -> Звук -> LoadSound). Эта функция загружает в память звуковой ждала нас, чтобы воспроизвести его. LoadSound() команда принимает два параметра, первый номер PB, которые вы хотите быть связано с этой звуковой файл, и второй это строка, содержащая имя файла звука, который вы хотите загрузить.

Когда звуковой файл был загружен вы можете играть в любое время в вашей программе, используя PlaySound() команду (Helpfile: Справочное руководство->2D Games Libraries->Sound->PlaySound).
Эта команда имеет один параметр, который является число PB в звуковой файл, вы хотите играть.

Если вы внимательно посмотрите на пример файла волны вы увидите, я использовал довольно сложную петлю на конце программы. Это, чтобы остановить этот пример программы выхода слишком рано. При этом выход программы, все звуки, в настоящее время играет им будет остановиться и выгружается из памяти. Я не хочу, чтобы это произошло сразу, так что я закодированы этот цикл, чтобы дать программе 8 секунд, чтобы воспроизвести файл, а затем выйти. Вы, вероятно, никогда не увидите это в реальной программе, поскольку она, вероятно, основной цикл, чтобы сохранить программе живой , а звук проигрывается.

Вложенный Wave файл

Иногда в своих программах вы можете не загружать внешние файлы Wav, но файлы на самом деле содержится в вашей программе, с тем все это можно распространять в виде одного файла. Это может быть сделано путем внедрения вашей звуковой файл в DataSection, аналогичные вложения изображений, как описано в главе 9 (включая графику в программе). Существует одна разница однако, вместо CatchImage() команды, для загрузки файлов из DataSection, мы используем CatchSound() команду, вот пример:

```
#SOUND_FILE = 1
 If InitSound()
   CatchSound(#SOUND_FILE, ?SoundFile)
   PlaySound(#SOUND_FILE)
   StartTime.l = ElapsedMilliseconds()
   Repeat
     Delay(1)
    Until ElapsedMilliseconds() > StartTime + 8000
    End
 EndIf
```

```
DataSection
 SoundFile:
 IncludeBinary "Intro.wav"
EndDataSection
```

Звуковой файл внедрен в DataSection с помощью IncludeBinary команды точно так же, как и изображение, и метка звуковой волны файла знаменует начало файла в памяти. Чтобы загрузить этот звук из DataSection при запуске программы, мы используем CatchSound() команду
(Helpfile: Справочное руководство по-> 2D игры Библиотеки-> Звук->CatchSound).
Эта команда имеет два параметра, первый номер PB, который будут связан с этим звуком и второе, это адрес в памяти, где этот звук файл должен быть загружен. Этот адрес будет этикетке адрес, поскольку это означает, где звуковой файл в памяти. Чтобы получить адрес используется знак вопроса напротив его названия, например ?SoundFile. Мы получили пути Wav файла , вы можете использовать его, как и любой другой файл Wav, в данном случае, я играю звук с помощью PlaySound() команды. Многие Wav файлы могут быть внедрены в программу, нужно просто дать всем им уникальные идентификаторы..

Изменение звука в реальном времени

Использование звуковой команды в библиотеке звука PureBasic, можно сделать изменения громкости, звук баланса и скорость воспроизведения. Том увеличивает или уменьшает громкость звука, панорамирования означает переход звук из одного громкоговорителя в другой, как правило, с левой или правой и изменение частоты, по сути ускорить, замедлить воспроизведение звукового файла во время воспроизведения . Для демонстрации этих эффектов, я создал программу небольшой звуковой проигрыватель, который использует изменение громкости, панорамирования и скорость воспроизведения. Попробуйте его, открыть, загрузить звуковой файл, нажмите клавишу воспроизведения и ручки трекбара.

```
Enumeration
  #WINDOW_ROOT
  #SOUND_FILE
  #TEXT_FILE
  #BUTTON_CHOOSE_FILE
  #TEXT_VOLUME
  #TRACKBAR_VOLUME
  #TEXT_PAN
  #TRACKBAR_PAN
  #TEXT_FREQUENCY
  #TRACKBAR_FREQUENCY
  #BUTTON_PLAY_FILE
  #BUTTON_STOP_FILE
EndEnumeration

Global FileName.s = ""

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#WINDOW_ROOT, 0, 0, 500, 250, "Sound Player", #FLAGS)
 If CreateGadgetList(WindowID(#WINDOW_ROOT))
```

```
TextGadget(#TEXT_FILE, 10, 10, 480, 20, "", #PB_Text_Border)
ButtonGadget(#BUTTON_CHOOSE_FILE, 10, 40, 150, 20, "Choose Wave File...")

TextGadget(#TEXT_VOLUME, 10, 70, 480, 20, "Volume")
TrackBarGadget(#TRACKBAR_VOLUME, 10, 90, 480, 20, 0, 100)
SetGadgetState(#TRACKBAR_VOLUME, 100)

TextGadget(#TEXT_PAN, 10, 120, 480, 20, "Pan")
TrackBarGadget(#TRACKBAR_PAN, 10, 140, 480, 20, 0, 200)
SetGadgetState(#TRACKBAR_PAN, 100)

TextGadget(#TEXT_FREQUENCY, 10, 170, 480, 20, "Frequency")
TrackBarGadget(#TRACKBAR_FREQUENCY, 10, 190, 480, 20, 100, 10000)
SetGadgetState(#TRACKBAR_FREQUENCY, 4400)

ButtonGadget(#BUTTON_PLAY_FILE, 10, 220, 100, 20, "Play File")
ButtonGadget(#BUTTON_STOP_FILE, 130, 220, 100, 20, "Stop Sound")

If InitSound()
 Repeat
 Event.l = WaitWindowEvent()
 Select Event
  Case #PB_Event_Gadget
   Select EventGadget()
    Case #BUTTON_CHOOSE_FILE
     FileName=OpenFileRequester("Choose","","Wave File (*.wav)|*.wav",0)
     If filename <> ""
      SetGadgetText(#TEXT_FILE, GetFilePart(FileName))
      LoadSound(#SOUND_FILE, filename)
     EndIf

    Case #TRACKBAR_VOLUME
     If filename <> ""
      SoundVolume(#SOUND_FILE, GetGadgetState(#TRACKBAR_VOLUME))
    EndIf

   Case #TRACKBAR_PAN
     If filename <> ""
      SoundPan(#SOUND_FILE, GetGadgetState(#TRACKBAR_PAN) - 100)
     EndIf

   Case #TRACKBAR_FREQUENCY
     If filename <> ""
      SoundFrequency(#SOUND_FILE,GetGadgetState(#TRACKBAR_FREQUENCY) * 10)
     EndIf

   Case #BUTTON_PLAY_FILE
     If filename <> ""
      PlaySound(#SOUND_FILE)
     EndIf

    Case #BUTTON_STOP_FILE
```

```
      If filename <> ""
        StopSound(#SOUND_FILE)
      EndIf
    EndSelect
   EndSelect
  Until Event = #PB_Event_CloseWindow
 EndIf
EndIf
End
```

Этот пример представляет три новые звуковые команды:

‘SoundVolume()’
Эта команда используется для управления громкости загруженного звука. Это не меняет оригинальный звуковой файл в любом случае, она лишь меняет громкость звука, когда она воспроизводится. Для выполнения этой команды вы должны передать её двумя параметрами, первый номер PB от звука, который вы хотите изменить И второе: уровень громкости который вы хотите изменить. Уровень громкости - число в диапазоне от 0 и до 100% изменяет от 0 до максимальной громкости.

‘SoundPan()’
Эта команда баланса звука левого и правого каналов. Чтобы использовать эту команду нужно передать два параметра. Первый номер PB от звука, который вы хотите для панорамирования и вторым параметром является значение баланса. Это значение баланса число которых колеблется от -100 до 100. Если вы используете значение \-100, то звук полностью играет из левой колонки. Если вы используете значение 100, то звук полностью играет из правой колонки.

‘SoundFrequency()’
Эта команда изменяет частоту звука для воспроизведения. Частота звуковой волны, измеряется в герцах и объясняется как количество колебаний в секунду. Например, вся музыка хранится на компакт-диске оцифровывается в размере 44,1 кГц (килогерц). Это означает, что сигнал содержащий звуковой информации на CD читается 44100 раз в секунду. Это необходимо в рамках резолюции для кодирования даже малейшего изменения звука. Если звук кодируется на 44,1 кГц, и вы используете эту команду, чтобы изменить свою частоту 22050 Гц, то, как только этот звук воспроизводится, она будет воспроизводить на половине скорости оригинала. Чтобы использовать эту команду в PureBasic Чтобы изменить частоту загруженных звук, вы передаете его двумя параметрами. Первый номер PB от звука, который вы хотите изменить и второй номер, который выражает новую частоту в Гц. Это второй параметр должен быть между 1000 и до 100000.

Чтобы узнать больше о командах, которые манипулируют файлов волны, см. звуковой библиотеки в файл помощи PureBasic
(Helpfile:Reference Manual->2D Games Libraries->Sound).

Модуль Файлы

Эти типы файлов используются форматы, что представляет музыку с использованием цифровых моделей. Внутри они хранят несколько страниц узорной музыки данных в форме, аналогичной, что и таблицы. Эти модели содержат сведения номера, номера инструмента и контроллера сообщения, которые говорят программа чтения файла, когда для воспроизведения нот, используя то, что образцы и как долго. Модуль файлов также проведет список, который определяет порядок, в котором Для воспроизведения модели.

Количество записей, которые могут звучать одновременно зависит от того, сколько треков Есть в шаблон. Ранняя программ, которые были доступны позволяет пользователям создавать свои собственные модули, используя четыре трека. Самым большим преимуществом модулей над стандартных звуковых файлов является то, что модули включают собственные сэмплы и должны звучать же от одного игрока к другому.

Модуль файлы часто называют трекер модулем и искусство составления модулей называется слежения, просто потому, что первая программа, которая позволяет пользователям создавать модули было называется Soundtracker. Эта программа, хотя изначально плохо получил, был выпущен в общественное достояние, и был клонирован много раз лучше спортивных характеристик и разные названия, такие как Noisetracker или ProTracker , эти стали очень популярны особенно в Commodore Amiga игры и демо-создателей. Программы, которые могут создавать файлы модулей сегодня вместе известны как трекеры.

Модуль файлов может быть много различных расширений файлов, потому что они приходят в разных форматах, эти расширения обычно показывают в программе создателя файла. Модуль типы файлов, которые поддерживаются PureBasic являются:

FastTracker (‘*.xm’)
Scream Tracker (‘*.s3m’)
Protracker(‘*.mod’)
Impulse Tracker (‘*.it’)

Эти различные типы модулей рассматриваются таким же образом, при погрузке и играть в вашей программе PureBasic. Вот пример, показывающий, как загружать и играть в любую из перечисленных выше типов модулей:

```
#MODULE_FILE = 1
 If InitSound()
  ;If InitModule()
   LoadModule(#MODULE_FILE, "Eighth.mod")
    PlayModule(#MODULE_FILE)
    StartTime.l = ElapsedMilliseconds()
    Repeat
      Delay(1)
    Until ElapsedMilliseconds() > StartTime + 15000
    StopModule(#MODULE_FILE)
   End
  ;EndIf
EndIf
```

Во-первых, нам необходимо инициализировать здоровой окружающей среды, как

например волны файл с помощью InitSound() команды. Далее, нам необходимо инициализировать модуль возможности воспроизведения PureBasic используя функцию InitModule() команды. Обе эти команды должны быть проверены для обеспечения того, чтобы оба инициализируется правильно.

После того как среда создана можно загрузить модуль с помощью LoadModule() команды (Helpfile: Справочное руководство по->2D игры Библиотеки-> модули-> LoadModule). Эта команда имеет два параметра, первый номер PB, что вы хотите, связаны с этим модулем И второе, это имя файла модуля должен быть загружен.

После его загрузки можно воспроизвести этот модуль в любое время, используя PlayModule() команды
(Helpfile: Справочное руководство по-> 2D игры Библиотеки-> модули->PlayModule).
Так же, как PlaySound() команду, на этот раз принимает один параметр, который является число PB модуля вы хотите играть.
Чтобы остановить модуль из игры вы можете использовать StopModule() команды.

Недостатки использования модулей

Есть две большие минусы использования модулей в своих программах PureBasic. Во-первых, не существует простой способ внедрения и загрузить из памяти все модули, которые должны быть воспроизведены. Это означает, что вы должны распространять все используемые файлы модулей вместе с исполняемым файлом. Существуют, однако, пути сохранения модуля в исполняемый файл и последующей записи на диск перед погрузкой и играть, но это немного громоздким. Второй недостаток в том, что вы должны распространять файл Midas11.dll вместе с вашей программой. Эта динамика связана библиотека загружается по команде InitModule() и будет ошибкой, если он не находит его. Вы можете думать, что это не так плох, но лицензии связанных с использованием этого Мидас библиотеке запрещает использовать его в коммерческих целях.
Эта лицензия также запрещает Midas11.dll файлы, который включен в пакет PureBasic, поэтому вам придется скачать ее для себя от Audio System веб-сайт Housemarque.

Чтобы узнать о других команд, которые могут использоваться для управления файлами модуля, см. модуль библиотеки в файл помощи PureBasic
(Helpfile:Reference Manual->2D Games Libraries->Module).

Mp3

MP3 файлы быстро становится самой популярной звуковой формат всех времен, в основном благодаря тому, что формат MP3 является стандартом де-факто для практически всех скачиваемой музыки в Интернете.

MP3 файлы обрабатываются немного по-разному в PureBasic А для того, Чтобы играть в них мы должны использовать команды из библиотеки ‘Movie’ library (Helpfile:Reference Manual->General Libraries->Movie).
Это может показаться немного странным, использование видео команд проигрывать MP3-файлы, но видео команды способны гораздо больше, чем просто воспроизведение фильмов. Видео библиотека предоставляет удобный способ для загрузки и играть почти во всех средствах массовой информации, которые кодек, установленный на компьютере.

Вы можете не только играть видео форматы с помощью этих команд, но вы можете также воспроизводить аудио форматы.

Вот список наиболее популярных форматов файлов, которые библиотека видео может играть, В зависимости от установленных кодеков Вы можете даже играть больше форматов, чем это показано в этом списке:

Movie Files:
- Audio Video Interleave (‘*.avi’)
- MPEG Video (‘*.mpg’)

Audio Files:
- Midi Files (‘*.mid’)
- MP3 Files (‘*.mp3’)
- Ogg Vorbis (‘*.ogg’)
- Wave Files (‘*.wav’)

Может показаться, что Библиотека Видео , подойдёт для всех ваших потребностей играть аудио и видео. И некоторые люди даже попросил, чтобы она была переименована в библиотеку Media, но надо помнить одну вещь , что если что-то и играет на компьютере, оно может не загружается и не играть на компьютере другого человека. Это потому, что могут быть различные плагины или кодеки установлены, (или нет) на других компьютерах. Однако, приведенный выше список, как представляется, довольно стандартным для большинства машин.

В следующем примере я использовал видео команды, чтобы создать простой MP3-плеер и обеспечить простой том, и пан контроля. Она не может соперничать WinAmp, но это дает вам представление о том, как легко медиаплееры создать в PureBasic.

```
Enumeration
  #WINDOW_ROOT
  #SOUND_FILE
  #TEXT_FILE
  #BUTTON_CHOOSE_FILE
  #TEXT_VOLUME
  #TRACKBAR_VOLUME
  #TEXT_PAN
  #TRACKBAR_PAN
  #BUTTON_PLAY_FILE
  #BUTTON_PAUSE_FILE
  #BUTTON_STOP_FILE
EndEnumeration

Global FileName.s = ""
Global FilePaused.b = #False
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#WINDOW_ROOT, 0, 0, 500, 215, "MP3 Player", #FLAGS)
 If CreateGadgetList(WindowID(#WINDOW_ROOT))
  TextGadget(#TEXT_FILE, 10, 10, 480, 20, "", #PB_Text_Border)

  ButtonGadget(#BUTTON_CHOOSE_FILE, 10, 40, 150, 20, "Choose MP3 File...")
```

```
TextGadget(#TEXT_VOLUME, 10, 70, 480, 20, "Volume")
TrackBarGadget(#TRACKBAR_VOLUME, 10, 90, 480, 20, 0, 100)
SetGadgetState(#TRACKBAR_VOLUME, 100)

TextGadget(#TEXT_PAN, 10, 120, 480, 20, "Pan")
TrackBarGadget(#TRACKBAR_PAN, 10, 140, 480, 20, 0, 200)
SetGadgetState(#TRACKBAR_PAN, 100)

ButtonGadget(#BUTTON_PLAY_FILE, 10, 180, 100, 20, "Play")
ButtonGadget(#BUTTON_PAUSE_FILE, 130, 180, 100, 20, "Pause")
ButtonGadget(#BUTTON_STOP_FILE, 250, 180, 100, 20, "Stop")

 If InitMovie()
  Repeat
   Event.l = WaitWindowEvent()
    Select Event
     Case #PB_Event_Gadget
      Select EventGadget()

       Case #BUTTON_CHOOSE_FILE
        FileName=OpenFileRequester("Choose","","MP3 File (*.mp3)|*.mp3",0)
         If filename <> ""
          SetGadgetText(#TEXT_FILE, GetFilePart(FileName))
          LoadMovie(#SOUND_FILE, filename)
         EndIf

       Case #TRACKBAR_VOLUME, #TRACKBAR_PAN
        If filename <> ""
         Volume.l = GetGadgetState(#TRACKBAR_VOLUME)
         Balance.l = GetGadgetState(#TRACKBAR_PAN) - 100
         MovieAudio(#SOUND_FILE, Volume, Balance)
        EndIf

       Case #BUTTON_PLAY_FILE
        If filename <> ""
         PlayMovie(#SOUND_FILE, #Null)
         FilePaused = #False
         SetGadgetText(#BUTTON_PAUSE_FILE, "Pause")
        EndIf

       Case #BUTTON_PAUSE_FILE
        If filename <> ""
         If FilePaused = #False
          PauseMovie(#SOUND_FILE)
          FilePaused = #True
          SetGadgetText(#BUTTON_PAUSE_FILE, "Resume")
         Else
          ResumeMovie(#SOUND_FILE)
          FilePaused = #False
          SetGadgetText(#BUTTON_PAUSE_FILE, "Pause")
         EndIf
        EndIf
```

```
        Case #BUTTON_STOP_FILE
         If filename <> ""
          StopMovie(#SOUND_FILE)
          FilePaused = #False
          SetGadgetText(#BUTTON_PAUSE_FILE, "Pause")
         EndIf
       EndSelect
      EndSelect
    Until Event = #PB_Event_CloseWindow
  EndIf
 EndIf
EndIf
End
```

Глядя на этот небольшой пример, вы можете увидеть, что для использования команды видео, вы должны инициализировать природной среды, как волны файлов и модулей. Для инициализации фильм используемые вами команды InitMovie() команды. После этого был назван Вы можете использовать другие команды библиотеки видео. Как и другие команды инициализации она должна быть проверена и, если он сломается, вы не сможете продолжать использовать фильм команд.

Для загрузки видео (или в данном случае MP3-файл) мы используем LoadMovie() команды (Helpfile: Справочное руководство по-> Общие библиотеки-> Movie-> LoadMovie). Эта команда имеет два параметра. Первый номер PB, что вы хотите, связанных с СМИ о загружать и вторым параметром является строка, содержащая имя файла реальных средах для загрузки файлов.

После того как средства массовой информации был загружен мы можем воспроизвести его при помощи оператора PlayMovie() команды
(Helpfile: Справочное руководство по->Общие библиотеки-> Movie-> PlayMovie).
Эта команда имеет два параметра, чтобы иметь возможность поддерживать фильмы, а также аудио информации. Первым параметром является номер PB средств массовой информации вы хотите играть, а второй параметр является идентификатором OS из окна. Этот идентификатор ОС При воспроизведении фильма, потому что это окно, где кино показывать будут оказаны. Если вы воспроизведения файла, который состоит только аудио-данных (например, MP3), то вы можете использовать встроенные в постоянном #Null Как идентификатор OS, это не затем связать любое окно в воспроизведении файл:

```
...
PlayMovie(#SOUND_FILE, #Null)
...
```

Кроме того, в примере я использовал PauseMovie() и ResumeMovie() команды, они просты в использовании, они оба принимают один параметр, который PB количество средств массовой информации вы хотите, чтобы приостановить или возобновить.

Чтобы вы могли остановить воспроизведение файлов, я также использовал StopMovie() команды в этом примере. Опять же, это просто один использовать, как вам нужно всего лишь передать его одному параметру. Это число PB средств массовой информации вы хотите остановить воспроизведение.

Даже если это скелет медиа плеер и он поддерживает только MP3, было бы тривиальная задача, чтобы перевести этот код, чтобы сделать свой собственный медиа-проигрыватель способный решать все из перечисленных форматов. Почему бы вам не попробовать?

CD Audio

Воспроизведение аудио CD является хорошим способом обеспечить высокое качество музыки Для любой игры или приложения, которые необходимо музыки. Есть много инструментов, доступных в Интернете, для создания и записи своего музыку на CD. Использование CD Предоставлять музыка это замечательная идея, потому что играть в них требует очень мало системных ресурсов и качества является фантастической.

Вот пример, который использует AudioCD библиотека PureBasic, чтобы создать очень простой проигрыватель компакт-дисков:

```
Enumeration
  #WINDOW_ROOT
  #BUTTON_PREVIOUS
  #BUTTON_PLAY
  #BUTTON_STOP
  #BUTTON_NEXT
  #BUTTON_EJECT
  #TEXT_STATUS
  #PROGRESS_SONG
  #LIST_TRACKS
EndEnumeration

;Global variables, etc.
Global NumberOfTracks.l
Global CurrentTrack.l
;Convert seconds into a String containing minutes
Procedure.s ConvertToMin(Seconds.l)
 ProcedureReturn Str(Seconds / 60) + ":" + Str(Seconds % 60)
EndProcedure
;Set the current track
Procedure UpdateStatusText(Track.l)
 If NumberOfTracks > 0
  TrackLength.l = AudioCDTrackLength(Track)
  TrackLengthString.s = ConvertToMin(TrackLength)
  TrackTimings.s = " (" + TrackLengthString + ")"
  SetGadgetText(#TEXT_STATUS, "Track: " + Str(Track) + TrackTimings)
  SetGadgetState(#PROGRESS_SONG, 0)
   If AudioCDStatus() > 0
    TimeElapsed.l = AudioCDTrackSeconds()
    TrackTimings.s=" ("+ConvertToMin(TimeElapsed)+" / "+TrackLengthString+")"
    SetGadgetText(#TEXT_STATUS, "Track: " + Str(Track) + TrackTimings)
    Progress.f = (100 / TrackLength) * TimeElapsed
    SetGadgetState(#PROGRESS_SONG, Progress)
   EndIf
  SetGadgetState(#LIST_TRACKS, Track - 1)
```

```
  Else
   SetGadgetText(#TEXT_STATUS, "Please insert an Audio CD")
 EndIf
EndProcedure

;Move to next track
Procedure NextTrack()
 If CurrentTrack < NumberOfTracks
   CurrentTrack + 1
   UpdateStatusText(CurrentTrack)
    If AudioCDStatus() > 0
      PlayAudioCD(CurrentTrack, NumberOfTracks)
    EndIf
 EndIf
EndProcedure

;Move to previous track
Procedure PreviousTrack()
  If CurrentTrack > 1
   CurrentTrack - 1
   UpdateStatusText(CurrentTrack)
    If AudioCDStatus() > 0
      PlayAudioCD(CurrentTrack, NumberOfTracks)
   EndIf
 EndIf
EndProcedure

;Populate the list to show all tracks on a disc
Procedure PopulateTrackListing()
 ClearGadgetItemList(#LIST_TRACKS)
 NumberOfTracks = AudioCDTracks()
  If NumberOfTracks > 0
   For x.l = 1 To NumberOfTracks
   TrackLength.s = ConvertToMin(AudioCDTrackLength(x))
   AddGadgetItem(#LIST_TRACKS, -1, "Track "+Str(x)+" ("+TrackLength+")")
    Next x
    If CurrentTrack = 0
      CurrentTrack = 1
    EndIf
  Else
   CurrentTrack = 0
  EndIf
EndProcedure

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#WINDOW_ROOT, 0, 0, 320, 250, "CD Player", #FLAGS)
If CreateGadgetList(WindowID(#WINDOW_ROOT))
ButtonGadget(#BUTTON_PREVIOUS, 10, 10, 60, 20, "Previous")
ButtonGadget(#BUTTON_PLAY, 70, 10, 60, 20, "Play")
ButtonGadget(#BUTTON_STOP, 130, 10, 60, 20, "Stop")
ButtonGadget(#BUTTON_NEXT, 190, 10, 60, 20, "Next")
```

```
ButtonGadget(#BUTTON_EJECT, 250, 10, 60, 20, "Eject")
TextGadget(#TEXT_STATUS, 10, 40, 300, 20, "", #PB_Text_Center)
ProgressBarGadget(#PROGRESS_SONG,10,65,300,10,0,100,#PB_ProgressBar_Smooth)
ListViewGadget(#LIST_TRACKS, 10, 90, 300, 150)

If InitAudioCD()
  PopulateTrackListing()
  StartTime.l = ElapsedMilliseconds()
Repeat
 Event.l = WindowEvent()
  Select Event
   Case #PB_Event_Gadget
    Select EventGadget()
     Case #BUTTON_PREVIOUS
       PreviousTrack()
     Case #BUTTON_PLAY
      If NumberOfTracks > 0
        PlayAudioCD(CurrentTrack, NumberOfTracks)
      EndIf
     Case #BUTTON_STOP
       StopAudioCD()
     Case #BUTTON_NEXT
       NextTrack()
     Case #BUTTON_EJECT
       EjectAudioCD(#True)
       PopulateTrackListing()
     Case #LIST_TRACKS
      If EventType() = #PB_EventType_LeftDoubleClick
        CurrentTrack = GetGadgetState(#LIST_TRACKS) + 1
        UpdateStatusText(CurrentTrack)
        PlayAudioCD(CurrentTrack, NumberOfTracks)
      EndIf
    EndSelect
  EndSelect

CurrentTime.l = ElapsedMilliseconds()
  If CurrentTime > StartTime + 1000
    PopulateTrackListing()
    UpdateStatusText(CurrentTrack)
    StartTime.l = ElapsedMilliseconds()
  EndIf
   Delay(1)
Until Event = #PB_Event_CloseWindow
 StopAudioCD()
EndIf
EndIf
EndIf
End
```

Этот пример очень простой CD-плеер, который обеспечивает минимум функций по контролю и воспроизведению по списку. О бо всей AudioCD библиотеке команд, которые могут быть использованы, читайте более подробно в файле помощи PureBasic

(Helpfile: Справочное руководство по-> Общие библиотеки-> AudioCD).

Чтобы использовать AudioCD команды для начала вы должны инициализировать необходимые ресурсы, чтобы иметь возможность играть CD. Для этого мы используем InitAudioCD () команды.
Эта команда должна быть проверена, чтобы определить, если она была успешной или нет. Заодно Возвращаемое значение этой команды является количество CD дисков, подключенных к компьютеру и из которых доступны для использования в целях воспроизведения музыки. Если возвращаемое значение равно 0, значит компьютер либо не имеет CD привода или еще одна проблема была выявлена, который препятствует работе компакт-дисков с музыкой.
Если возвращаемое значение выше 0,то все должно быть хорошо.

После того как аудио CD была инициализирована, вы можете использовать все другие команды, доступные в AudioCD библиотеке...

Вот список тех команд, которые я использовал в своем примере проигрыватель компакт-дисков:

'PlayAudioCD()'
Эта команда используется для воспроизведения трека на текущем CD, который вставляется в дисковод. Эта команда имеет два параметра. Первый трек Чтобы начать игру. А второй трек для остановки воспроизведения после окончания. Так, например, если я назвал команду наподобие следующей:

```
PlayAudioCD(1, 3)
```

Трек 1 начнет игру, а затем продолжить играть трек 2 и далее. Остановить после проигрывания трека 3.

'StopAudioCD()'
Эта команда может быть использован для остановки воспроизведение любого трека компакт-диска.

'EjectAudioCD()'
Эта команда откроет или закроет лоток на CD диске в зависимости от того, какой параметр передается ей. Эта команда принимает только один параметр, и если это параметр 1 лоток привода откроется, остановка воспроизведения и извлечения CD. Если параметр 0 привод лоток закроется, загрузкой CD, который помещен в нем.

'AudioCDStatus()'
Эта команда не имеет параметров, но возвращает результат, который дает в реальном времени статус CD в приводе. Если вы вызываете эту команду и возвращает 1, то диск не готов, это означает, что диск не содержит ни CD или что лоток дисковода в настоящее время закрыт. Если эта команда возвращает значение 0 , значит CD был обнаружен правильно и находится в приводе CD, но не играет. Если эта команда возвращает значение, которое выше 0, значит компакт-дисков играет CD дорожки. Число, которое было фактически вернулись, это трек номер, который воспроизводится

‘AudioCDTracks()’
Эта команда не принимает никаких параметров и при вызове, возвращает количество треков на CD загружается, которые доступны для воспроизведения.

‘AudioCDTrackLength()’
Эта команда требует один параметр И это номер трека из загруженного диска. Имея эту информацию, команда вернет длину дорожки в секундах.

‘AudioCDTrackSeconds()’
Эта команда не принимает никаких параметров. При вызове, она вернет количество времени, в секундах с начала проигрывания трека CD.

Если вы посмотрите внимательнее к моему примеру проигрыватель компакт-дисков, я использовал команду WindowEvent () вместо WaitWindowEvent (). Это означает, что основной цикл будет работать всегда, независимо от обнаружено событие или нет. Мне нужна чтобы программа делала обновления статуса текста и графического пользовательского интерфейса CD. Если бы я не использовать эту команду, статус текст будет обновляться только если событие было обнаружено, что может быть или нет.

11
Работа с памятью

Современные операционные системы не позволяют обращаться к любому участку оперативной памяти. Это в частности вызвано многозадачностью системы, т. е. чтобы одна программа, умышленно или по ошибке не могла исказить данные другой программы. Для доступа к памяти, сначала нужно запросить у операционной системы память, требуемого размера. Для работы с памятью в PureBasic есть ряд функций, описание которых можно найти в разделе **Memory** справки.
Итак, для того, чтобы получить доступ к памяти, её сначала нужно "заказать" у операционной системы, затем выполнить требуемые действия, после чего нужно обязательно освободить память, что-бы избежать её "утечек". Под словом "утечка" имеется в виду использование памяти программой, после того, как она становится не нужной для дальнейших операций. Этого следует всячески избегать.

Вот пример работы с памятью.

```
*MemoryID = AllocateMemory(50) ; Запрашиваем у ОС 50 байт памяти
 If *MemoryID
    ; В цикле производится запись в память
    For i=0 To 9
      PokeC(*MemoryID+i, i)
    Next i

    ; В цикле производится чтение из памяти
    For i=0 To 9
     Debug PeekC(*MemoryID+i)
```

```
    Next i
    ; Запись строки в память
    PokeS(*MemoryID, "Работа с памятью")

    ; Чтение строки из памяти
    Debug PeekS(*MemoryID)

    FreeMemory(*MemoryID)  ; Освобождаем память, т. к. она больше не нужна
 Else
    MessageRequester("", "Не удалось выделить память!")
 EndIf
```

С помощью функции AllocateMemory у операционной системы запрашивается требуемое число байт, в данном случае 50. В случае успеха, функция возвращает указатель на первую ячейку памяти. Если по той или иной причине, не удалось выделить память, то функция вернет число **0**. Оператор **If** проверяет чтобы в переменной-указателе было число не равное нулю. В цикле **For - Next** происходит запись 10 байт в первые 10 ячеек выделенной памяти, а затем, в следующем цикле, данные из памяти считываются и отображаются в отладочном окне. Потом в память записывается строка текста, после чего считывается от-туда.
Функция **FreeMemory** освобождает память, т. к. для работы программы она больше не нужна.

В следующем примере показано как можно прочитать всю информацию из файла и пометить её в память, а затем, скопировать из памяти в строковую переменную и отобразить данные в текстовом редакторе.

```
File.s=OpenFileRequester("","","Текстовые файлы (txt)|*.txt",0)
If File<>""
  If ReadFile(0, File) ; Открытие файла
    Size=Lof(0) ; Определение размера файла в байтах
    *mem=AllocateMemory(Size) ; Запрашиваем у ОС число байт, равное размеру
файла
    If *mem
      ReadData(0, *mem, Size) ; Копирование данных из файла в память
      CloseFile(0) ; Закрытие файла
      Text.s=PeekS(*mem) ; Копируем данные из памяти в строковую переменную
      FreeMemory(*mem) ; Освобождаем память
    Else
      MessageRequester("", "Не удалось получить запрашиваемую память")
      End ; Завершение работы программы
    EndIf
  Else
    MessageRequester("", "Не удалось открыть файл")
    End ; Завершение работы программы
  EndIf
EndIf
; Открываем окно
OpenWindow(1,0,0,400,400,"Работа с памятью", #PB_Window_MinimizeGadget |
#PB_Window_ScreenCentered)
CreateGadgetList(WindowID(1))
EditorGadget(1,2,2,396,396) ; Создаём текстовый редактор
SetGadgetText(1,Text) ; Помещаем текст из строковой переменной в редактор

Repeat ; Главный цикл программы
 Event=WaitWindowEvent() ; Получаем идентификатор события в программе
Until Event=#PB_Event_CloseWindow  ; Прерываем цикл если нужно закрыть окно
```

Жёлтым цветом выделены строки, в котрых происходит работа с памятью.
В начале программы с помощью функции OpenFileRequester создаётся стандартное окно выбора открываемого файла, в котром нужно указать путь с текстовому файлу, имеющему расширение TXT. Функция возвращает полный путь к выбранному файлу, который будет записан в строковую переименую **File**. Далее открывается выбранный файл и определяется его размер - количество байт в файле. Затем с помощью функции AllocateMemory выделяется размер памяти, равный размеру файла и с помощью функции ReadData данные из файла копируются в память. Так как файл нам больше не нужен (его копия хранится в памяти), то его закрываем с помощью функции CloseFile. Далее с помощью функции PeekS считываются данные из памяти и помещаются в строковую переменную с именем **Text**. Так как данные были скопированы в переменную и память нам больше не нужна, то её освобождаем при помощи функции FreeMemory.
Далее открывается окно и создаётся текстовый редактор, в который помещается текст из переменной **Text**. В итоге мы видим текст из выбранного файла.

12
Указатели

Обычно указатели хранят адрес в памяти того или иного объекта (переменной, структуры, процедуры и т. д.), т. е. указывают на этот объект. Не скажу что работать с указателями приходится очень часто, но иногда возникает такая необходимость. Для хранения адреса памяти существует специальный тип переменной. В начале такой переменной находится символ ***** (звёздочка). То есть это `*pointer` переменная-указатель на область памяти.
PureBasic позволяет получать адреса, переменных, массивов, структур, связанных списков, процедур и меток.
Адрес метки можно получить подставив перед её именем символ **?**, адреса остальных объектов можно получить поставив перед их именем символ **@**.
Следующий пример показывает как можно получить адрес переменной и работать с ней как с памятью.

```
var=100 ; Запись числа 100 в переменную
*pointer=@var ; Поучение адреса переменной
Debug PeekC(*pointer) ; Чтение данных из переменной зная её адрес в памяти
```

В первой строке происходит запись числа в переменную. Узнаём адрес в памяти где хранятся данные этой переменной, поставив перед её именем "собаку" - символ **@**. После чего, с помощью функции PeekC происходит чтение данных из памяти, т. е. из переменной **var**.

Помните, в статье Работа с памятью был приведён пример программы, которая считываем

текст из файла и отображает его в текстовом редакторе. Там данные сначала копировались из файла в память, а затем из памяти в строковую переменную. Этот пример можно упростить, исключив работу с памятью, точнее можно использовать строковую переменную как память. Тогда данные будут копироваться из файла сразу в переменную. Пример будет выглядеть так:

```
File.s=OpenFileRequester("","","Текстовые файлы (txt)|*.txt",0)
If File<>""
  If ReadFile(0, File) ; Открытие файла
    Size=Lof(0) ; Определение размера файла в байтах
    Text.s=Space(Size) ; Заполнение строковой переменной пробелами
    ReadData(0, @Text, Size) ; Копирование данных из файла в переменную
    CloseFile(0) ; Закрытие файла
  Else
    MessageRequester("", "Не удалось открыть файл")
    End ; Завершение работы программы
  EndIf
EndIf
; Открываем окно
OpenWindow(1,0,0,400,400,"Работа с памятью", #PB_Window_MinimizeGadget |
#PB_Window_ScreenCentered)
CreateGadgetList(WindowID(1))
EditorGadget(1,2,2,396,396) ; Создаём текстовый редактор
SetGadgetText(1,Text) ; Помещаем текст из строковой переменной в редактор

Repeat ; Главный цикл программы
 Event=WaitWindowEvent() ; Получаем идентификатор события в программе
Until Event=#PB_Event_CloseWindow  ; Прерываем цикл если нужно закрыть окно
```

Перед чтением теста из файла, в строковую переменную **Text**, с помощью функции Space записывается число пробелов, равное размеру файла. Это нужно чтобы зарезервировать в переменной требуемое число байт. Далее, при помощи функции ReadData производится копирование данных из файла в переменную. Функция как и положено получает указатель на память, только это память, занимаемая переменной.

С помощью указателей можно возвращать из процедуры более одного результата и при этом не использовать глобальные переменные. Следующий пример демонстрирует это.

```
Procedure Test(*var1, *var2)
 PokeL(*var1, 12345678) ; Запись числа в память
 PokeS(*var2, "Строка текста") ; Запись текста в память
EndProcedure

var.l=0
Text.s=Space(100) ; Резервируем данные в строковой переменной, путём записи
100 пробелов
Test(@var, @Text) ; Вызываем процедуру и передаём ей указатели на переменные
var и Text
; Отображаем в отладочном окне данные из переменных.
Debug var
Debug Text
```

При инициализации переменных, в **var** записывается число 0, а в строковую переменную **Text**, записывается 100 пробелов, т. е. резервируется память. Далее вызывается процедура Test, которой передаются указатели на эти переменные. В процедуре производится запись данных в эти переменные, причём с ними производится работа как с памятью. И в конце программы, с помощью оператора **Debug**, в отладочном окне отображается данные из

этих переменных.

Но гораздо удобнее работать со структурами, т. к. не нужно использовать функции доступа к памяти.

```
Structure Proba
  x.l
  y.l
  Text.s
EndStructure

test.Proba ; Объявление структуры

Procedure Test(*var.Proba)
 *var\x=1
 *var\y=2
 *var\Text="Текст"
EndProcedure

Test(@test) ; Вызываем процедуру и передаём ей указатель на структуру
; Отображаем в отладочном окне данные из структуры
Debug test\x
Debug test\y
Debug test\Text
```

В следующем примере показано, как можно вызывать процедуру зная её адрес в памяти.

```
Procedure Message(Title.s, Message.s)
 MessageRequester(Title, Message)
EndProcedure
; Узнаём адрес процедуры и помещаем его в переменную-указатель
*ponter=@Message()
; Вызываем процедуру зная её адрес в памяти
CallFunctionFast(*ponter, "Заголовок", "Текст")
```

Возможность получать адрес метки, позволяет сохранять различные данные в теле исполняемого файла используя так называемую ДатаСекцию. Это позволяет хранить, скажем один или несколько файлов, требуемых для работы программы. Чаще всего это значки, различные рисунки, музыка и т. д. Можно даже хранить динамическую библиотекиу подпрограмм (исполняемый файл с расширением DLL), которую можно запускать прямо из памяти.

Следующий пример показывает как можно поместить значок в исполняемый файл и использовать его в программе.

```
; Открываем окно
OpenWindow(1,0,0,80,80,"",#PB_Window_MinimizeGadget |
#PB_Window_ScreenCentered)
CreateGadgetList(WindowID(1)) ; Создаём новый список гаджетов

  CatchImage(1,?Metka1, ?Metka2-?Metka1) ; Загружаем рисунок из памяти
  ImageGadget(2,40,20,32,32, ImageID(1) ) ; Отображаем его

Repeat ; Начало главного цикла Repeat-Until
 Event=WaitWindowEvent() ; Получаем текущий идентификатор события
; Прерываем цикл при попытке закрыть окно (щелчёк по крестику в заголовке
окна)
```

```
Until Event=#PB_Event_CloseWindow
End ; Завершаем работу программы

DataSection
 Metka1:
  IncludeBinary "Значок.ico"
 Metka2:
EndDataSection
```

На этапе компиляции, с помощью оператора **IncludeBinary** в исполняемый файл записывается файл с именем **Значок.ico**, находящийся в одной папке с исходным текстом. Метки перед и после этого оператора позволяют узнать место в памяти, где хранится файл, в нашем случае значок, который в функции CatchImage загружается, а затем отображается в окне. Первый аргумент этой функции - идентификатор создаваемого рисунка, второй аргумент - адрес памяти начала рисунка, который получаем с помощью символа **?** перед именем метки. Третий аргумент - размер памяти в байтах, занимаемый рисунком. Чтобы его получить, просто отнимаем адрес второй метки от адреса первой метки.

```
; Открываем окно

OpenWindow(1,0,0,80,80,"",#PB_Window_MinimizeGadget|#PB_Window_ScreenCentered)

CreateGadgetList(WindowID(1)) ; Создаём новый список гаджетов

 CatchImage(1,?Metka1, ?Metka2-?Metka1) ; Загружаем рисунок из памяти

 ImageGadget(2,40,20,32,32, ImageID(1) ) ; Отображаем его

 Repeat ; Начало главного цикла Repeat-Until

 Event=WaitWindowEvent() ; Получаем текущий идентификатор события

Until Event=#PB_Event_CloseWindow ; Прерываем цикл при попытке закрыть окно (щелчёк по крестику в заголовке окна)

End ; Завершаем работу программы

DataSection

 Metka1:

 IncludeBinary "Значок.ico"

 Metka2:

EndDataSection
```

13
Массивы

Массивы предназначены для объединения множества однотипных переменных.

Доступ к переменных массива осуществляется через индекс (номер в массиве). Массив объявляется с помощью оператора **Dim**.

```
Dim  Array.l( 10 )
```

Вот так объявляется массив, состоящий из 11 переменных типа **Long** и имеющий имя **Array**. У вас может появится вопрос, почему 11 переменных, а не десять как задано в скобках? Дело в том, что нумерация начинается с нуля и поэтому получается что на самом деле в массиве на одну переменную больше чем задано.
Для доступа к переменной массива, нужно в скобках указать её индекс (номер) и прочитать либо записать информацию.

```
Dim Array.l(10) ; Создание массива
Array(2)=100 ; Запись в массив
x=Array(2) ; Чтение из массива
Debug x ; Отображаем число из переменной "x" в отладочном окне
```

В этом примере создаётся массив с именем Array, а затем в переменную с номером 2, записывается число 100. После чего производится чтение из этой же переменной. Результат помещается в переменную с именем **X**.
В полной мере ощутить приемущество массива перед отдельными переменными можно в циклах, когда нужно при каждом выполнении кода получать доступ к требуемой переменной.

```
Dim Array.l(100)
For i=1 To 100
  Array(i)=i
Next i
```

В этом примере создаётся массив с 101 переменной типа **Long**, а затем в цикле в этот массив записываются данные, в нашем случае число из переменной **i**
Выше переведены примеры одномерных массивов, но при необходимости можно использовать многомерный массив. Для этого, при объявлении нужно задать размерность много мерного массива.
Например в этом примере создаётся 3-ёх мерный массив

```
Dim Array.l(10, 4, 8) ; Создание 3-ёх мерного массива
Array(2, 1, 1)=100 ; Запись в массив
x=Array(2, 1, 1) ; Чтение из массива
Debug x ; Отображаем число из переменной "x" в отладочном окне
```

При необходимости, можно изменить размерность массива (количество переменных в массиве) после того как массив был создан. Это выполняется с помощью оператора **ReDim**.

```
Dim Array.l(10) ; Создание массива
; Здесь находится код программы
; Вдруг выяснилось что 11 переменных это мало, надо как минимум 21
ReDim Array(20) ; Было 11 переменных в массиве, а стало 21.
```

Структуры

В отличие от массива, переменные в структуре могут иметь разный тип.
Вот пример создания структуры

```
Structure Proba
  x.l
  y.l
  Text.s
EndStructure
```

Операторы **Structure** и **EndStructure** определяют начало и конец структуры. Между ними расположены переменные структуры Первые две переменные имеют тип **Long**, а последняя, тип **String**. Структуре присвоено имя **Proba** которое будет использоваться при её объявлении.
Следующий пример содержит работу со структурой

```
Structure Proba
  x.l
  y.l
  Text.s
EndStructure

test.Proba ; Объявление структуры

; Запись в структуру
test\x=10
test\y=20
test\Text="Строка текста"

; Чтение из структуры и отображение данных в отладочном окне
Debug test\x
Debug test\y
Debug test\Text
```

В структурах допускаются вызовы других структур и статические (размер которых после объявления нельзя изменить) массивы переменных. Учтите, в структуре индексация массива начинается с нуля и чило элементов равно заданному. Другими словами если задан массив, равный 100, то к нему можно обращаться по индексам 0...99

```
Structure Proba
  x.l[100]  ; Массив переменных
  z.POINT ; Объявление системной структуры с именем POINT
EndStructure

text.Proba ; Объявление структуры

; Запись в структуру
For i= 0 To 99
  text\x[i]=i
Next i

text\z\x=1
text\z\y=2

; Чтение из структуры и отображение данных в отладочном окне
Debug "Чтение из массива"
For i= 0 To 99
 Debug text\x[i]
```

```
Next i

Debug ""
Debug "Чтение из вложенной структуры POINT"

Debug text\z\x
Debug text\z\y
```

Кроме того, есть возможность добавить структуру в обычному массиву. Для этого в место типа переменных массива указываем имя структуры.

```
Structure Proba
  x.l
  y.l
  Text.s
EndStructure

; Создание массива с прикреплённой к нему структурой
Dim Array.Proba(2)

; Запись в структуру массива
Array(1)\x=1
Array(1)\y=2
Array(1)\Text="Текст"

; Чтение из структуры массива и отображение данных в отладочном окне
Debug Array(1)\x
Debug Array(1)\y
Debug Array(1)\Text
```

Динамически связанные списки

В динамически связанных списках в отличие от массивов, за переменными чётко не закреплены определённые индексы (номера, позиции). Кроме того, есть механизм, позволяющий раздвигать переменные и вклинивать тода одну или несколько переменных или наоборот, уничтожать несколько переменных, сдвинув остальные переменные. Всё это позволяет динамически изменять данные в списке.
Список создаётся с помощью оператора **NewList**.

```
NewList MyList.l()
```

Созданный список пока пустой, т. е. не содержащий элементов. Элементы добавляются с помощью функции **AddElement**. Созданный элемент становится текущим и именно с ним будет происходить работа. Если нужно обратится к определенному элементу, то его сначала нужно сделать текущим с помощью функции **SelectElement**. Получить номер текущего элемента можно с помощью функции **ListIndex**.
Для того чтобы просмотреть содержимое всего списка, можно использовать специальный цикл, расчитанный только для работы со связанными списками. Этот цикл организуется с помощью операторов **ForEach** и **Next**.

```
NewList MyList.l() ; Создание списка

AddElement(MyList()) ; Добавляем элемент
MyList()=1
AddElement(MyList()) ; Добавляем элемент
```

```
MyList()=2
AddElement(MyList()) ; Добавляем элемент
MyList()=3
AddElement(MyList()) ; Добавляем элемент
MyList()=4

; В цикле просматриваем содержимое списка
ForEach MyList()
  Debug MyList()
Next
```

В этом примере, в список добавляется четыре элемента. Как только элемент добавлен, он становится текущим, с которым на данный момент можно работать. В цикле **ForEach** и **Next** происходит последовательная активация всех элементов, т. е. элементы последовательно начиная с нулевого, становятся активными.

В место тпа переменных списка, можно указать имя существующей структуры и тогда каждый элемент списка будет содержать до такой структуре.

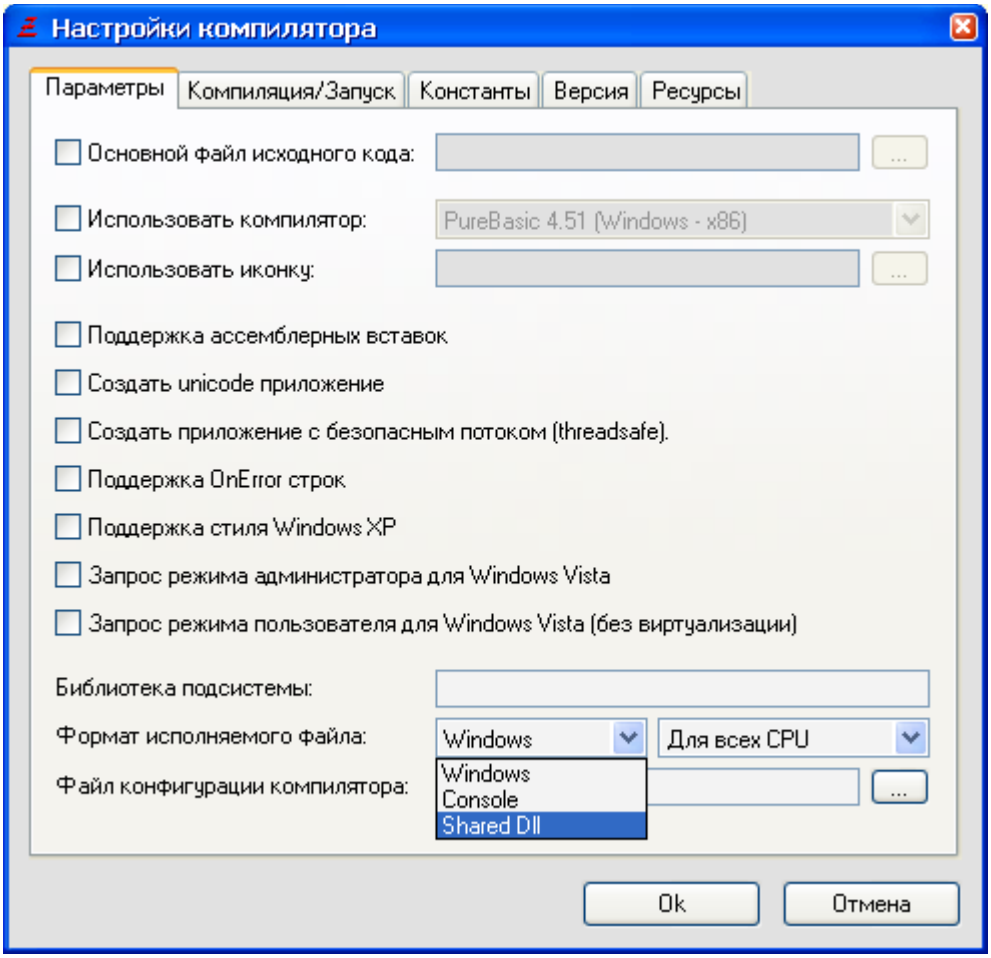
```
Structure Proba
  x.l
  y.l
  Text.s
EndStructure

NewList MyList.Proba() ; Создание списка

AddElement(MyList()) ; Добавляем элемент
; Запись в структуру нулевого элемента списка
MyList()\x=1
MyList()\y=2
MyList()\Text="Текст"

; Чтение из структуры нулевого элемента списка и отображение данных в
отладочном окне
Debug MyList()\x
Debug MyList()\y
Debug MyList()\Text
```

14
Создание динамической библиотеки подпрограмм и работа с ней

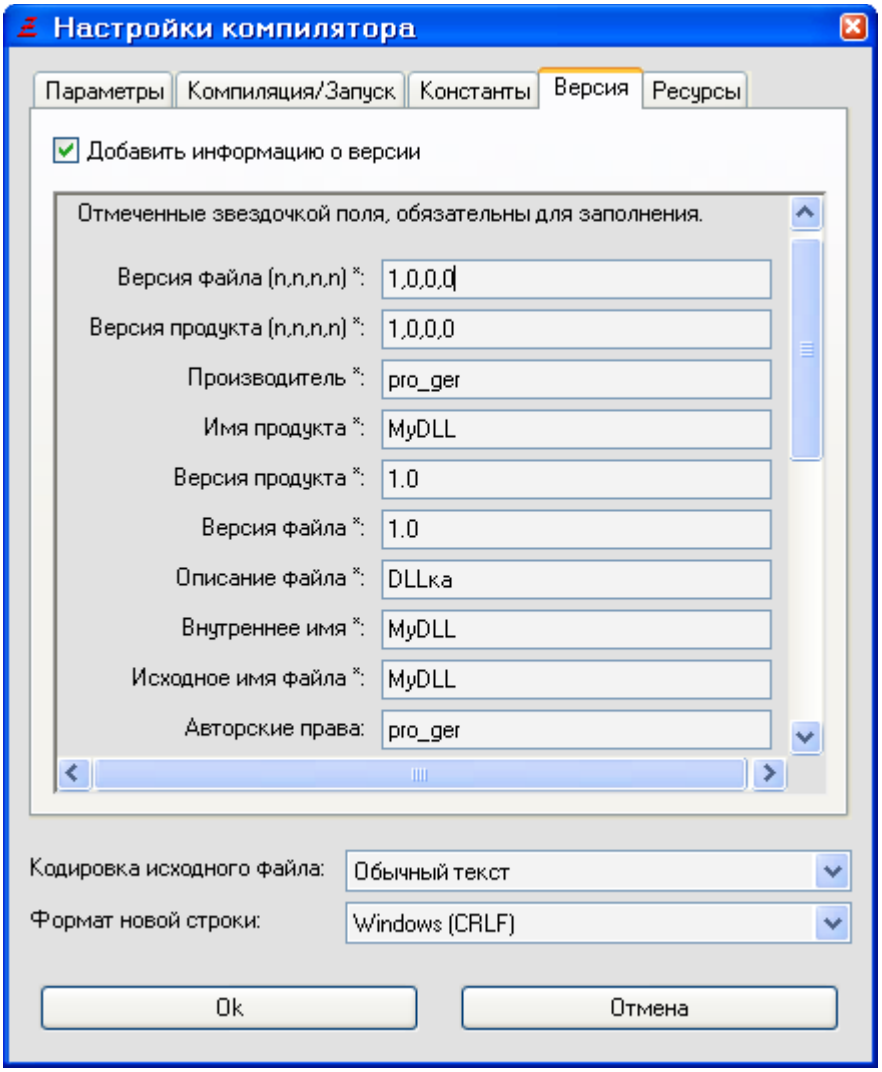
Я сейчас не буду объяснять что такое динамическая библиотека подпрограмм и для чего они нужна, так как это выходит за рамки данной статьи.
Перейду сразу к делу (практике).
В PureBasic исходник динамической библиотеки подпрограмм представляет из себя обычные процедуры, но в место оператора **Procedure** используется оператор **ProcedureDLL** или оператор **ProcedureCDLL** если динамическая библиотека подпрограмм должна быть типа **cdecl.**
Вконце этого оператора может следовать суффикс, обозначающий тип переменной, которую вернет процедура с помощью оператора **ProcedureReturn**. Например, если нужно чобы процедура возвращала строку текста, т. е. строковую переменную или константу, то следует указать суффикс **.s**, т. е. **ProcedureDLL.s**

А если требуется вернуть из процедуры дробное число, занимаемое 4 байта в памяти компьютера, то следует использовать суффикс **.f** являющийся сокращением от слова Float.
Более подробно о типах переменных. 2
Если не указан суффикс (как в примерах ниже), то будет использован тип переменной заданный по умолчанию. В PureBasic до версии 4.30 это был тип Long, а в последующих версиях, тип Integer.

Итак, запускаем среду PureBasic (это обязательно должна быть полная версия, а не демо-версия).
Перед нами будет пустой проект - страница в редакторе кода.
В первую очередь нужно сохранить эту страницу в виде файла. Имя файла и его расположение на диске может быть любым.
Далее открываем свойства проекта выбрав в меню **Компилятор** пункт **Настройки компилятора** и на закладке Параметры в выпадающем списке Формат исполняемого файла выбираем пункт **Shared Dll**



Если захотите, можете ещё указать дополнительные данные о файле, такие как его версия, данные об авторе и т. д. Но это не обязательно



Теперь всё готово для ввода кода динамической библиотеки подпрограмм.
Для примера используем простейший код.
Код:

```
ProcedureDLL Plus(a, b)
 ProcedureReturn a+b
EndProcedure


ProcedureDLL Message(Title.s, Message.s)
 MessageRequester(Title, Message)
EndProcedure
```

В этом примере две процедуры. Первая просто складывает числа из переменных **a** и **b** и возвращает результат сложения.
Вторая процедура отображает окно с сообщением.

Для создания из этого кода динамической библиотеки подпрограмм, нужно в меню **Компилятор** выбрать пункт **Создать приложение**
При этом будет создано несколько файлов, но нас сейчас интересует файл с расширением DLL, это и есть динамическая библиотека подпрограмм.
Теперь задействуем её в нашей программе.

Код:

```
If OpenLibrary(1, "MyDLL.dll")=0 ; Открываем динамическую библиотеку
подпрограмм
 MessageRequester("", "Отсутствует файл MyDLL.dll")
 End ; Завершение работы программы
EndIf

a=10 ; Запись данных в переменные
b=20
Result=CallCFunction(1, "Plus", a, b) ; Вызов функции Plus
String.s="a+b="+Str(Result)
CallCFunction(1, "Message", @"Сообщение", @String) ; Вызов функции Message

CloseLibrary(1) ; Закрытие динамической библиотеки подпрограмм
```

Функция OpenLibrary открывает (загружает в память, если это первый вызов) библиотеку MyDLL.dll, которая должна быть в одной папке этой программой.
Если не получилось открыть, то появится сообщение об этом и работа программы завершится.
В случае успешного открытия, в переменные с именами **a** и **b** будут записаны числа (их значения могут быль любыми).
Далее при помощи функции CallCFunction вызывается функция с именем Plus из динамической библиотеки подпрограмм. Ей передаются данные из переменных **a** и **b**, а возвращаемый результат помещается в переменную **Result**
В следующей строке в строковую переменную **String** будет записан текст "a+b=" и текущее значение переменной **Result**, преобразованное в строковой вид функцией Str
После этого вызывается функция Message из динамической библиотеки подпрограмм, которой передаётся текущее содержимое строковой переменной **String**. Эта функция отобразит окно с текстом из переменной **String**.
Когда же это окно будет закрыто, функция CloseLibrary закроет динамическую библиотеку подпрограмм.
Так как дальше кода нет, работа программы завершится.

Скачать архив со всеми файлами можно http://pure-basic.narod.ru/docs/MyDLL.rar .

PS.
Здесь был приведён пример создания и работы с динамической библиотекой подпрограмм в операционной системе семейства Windows, но все выше написанное можно отнести и к созданию динамической библиотекой подпрограмм для операционной системы Linux, но нужно использовать компилятор для этой платформы и учесть, что у динамической библиотеки подпрограмм будет расширение **SO**
Эти же примеры будут работать в MacOS X и AmigaOS.

Работа с динамической библиотекой подпрограмм, находящейся в памяти

Иногда, желательно чтобы программа состояла только из одного исполняемого файла.
А что делать, если для работы программы нужны другие файлы, например, динамические библиотеки подпрограмм (DLLки)?
Если их немного и они имеют небольшой размер, то вполне реально разместить их в теле исполняемого файла.

Для примера используем программу, требующую для своей работы файл *inpout32.dll*. Это драйвер доступа к портам компа.

Программа определяет текущую температуру процессора и системной платы, а также текущую загрузку процессора.

Код программы

Код:

```
#ADR_REG = $295 ;Это адреса регистров системной платы, с помощью
которых можно узнать температуру
#DATA_REG = $296

*HModule=LoadLibraryM(?DLL) ; Загрузка динамической библиотеки
из тела исполняемого файла
*Inp32_address=GetProcAddressM(*HModule, "Inp32") ; Узнаём адрес
функции "Inp32" DLLки
*Out32_address=GetProcAddressM(*HModule, "Out32") ; Узнаём адрес
функции "Out32" DLLки

Procedure Termo(z) ; Эта процедура работает в параллельном
потоке
Shared *Inp32_address, *Out32_address

 Repeat ; Начало "безконечного" цикла Repeat ForEver
 CallFunctionFast(*Out32_address, #ADR_REG, $2B) ; Получает текущую
температуру процессора
 x=CallFunctionFast(*Inp32_address, #DATA_REG)
 SetGadgetItemText(0,0,StrU(x, #PB_Byte)+" °C",1) ; отображаем её в таблице

 CallFunctionFast(*Out32_address, #ADR_REG, $29) ; Получает текущую
температуру системной платы
 x=CallFunctionFast(*Inp32_address, #DATA_REG)
 SetGadgetItemText(0,1,StrU(x, #PB_Byte)+" °C",1) ; отображаем её в таблице

 SetGadgetText(3, Str(CpuUsage())+" %" ) ; Получаем и отображаем текущую
загрузку процессора

 Delay(1000) ; Пауза, равная 1 секунде.

 ForEver
EndProcedure


OpenWindow(0,0,0,274,100,"TermoControl",#PB_Window_MinimizeGadget|#PB_Window_
Invisible|#PB_Window_ScreenCentered)

 ListIconGadget(0,2,2,270,70,"Имя",120,#PB_ListIcon_GridLines) ; Таблица
 SetGadgetFont(0,LoadFont(0,"MS Sans Serif",10) ) ; Шрифт используемый в
таблице
 AddGadgetColumn(0, 1, "Температура", 140) ; Добавление колонки в таблицу
 AddGadgetItem(0, 0, "Процессор") ; Добавление строк в таблицу
 AddGadgetItem(0, 1, "Мат. плата")

 TextGadget(2,10, 80,140,16,"Процессор загружен на ")
 TextGadget(3,150, 80,50,16,"") ; Здесь будет отображаться текущая загрузка
процессора

 CreateThread(@Termo(), 0) ; Запуск кода процедуры "Termo" в параллельном
потоке
```

```
 HideWindow(0,0) ; Отображение окна

 Repeat ; Главный цикл программы
 Event=WaitWindowEvent()
 Until Event=#PB_Event_CloseWindow

 FreeLibraryM(*HModule) ; При завершении работы проги, выгружаем DLLку из
памяти
 End

DataSection ; Добавление файла inpout32.dll в секцию кода, исполняемого файла
 DLL:
 IncludeBinary "inpout32.dll"
EndDataSection
```

Для компиляции программы нужна специальная библиотека с дополнительными функциями для PureBasic, именуемая PBOSL Найти её можно http://pure-basic.narod.ru/libs.html

В начале программы, в константы #ADR_REG и #DATA_REG записываются адреса регистров, из которых можно прочитать текущие температуры.
Далее с помощью функции LoadLibraryM из библиотеки PBOSL регистрируем DLLку, находящуюся в теле исполняемого файла и получаем её начальный адрес в памяти.
Потом с помощью функции GetProcAddressM определяем адреса функций DLLки - "Inp32" и "Out32".
Зная эти адреса, можно без проблем работать с функциями DLLки

В процедуре **Termo** производится периодическое считывание температуры и отображение её в таблице. Код данной процедуры выполняется в параллельном потоке и можно так сказать, "живёт" сам по себе.

Вот скрин проги.

PS.
Этот метод определения температуры работает только при условии что в компе интеловский процессор и системная плата поддерживает данную функцию.

Архив с программой TermoControl и её исходным текстом, можно скачать http://pure-basic.narod.ru/docs/TermoControl.rar

15

Пишем электронную книгу или добавление других файлов в исполняемый файл

Сейчас мы поговорим о возможности добавления в скомпилированный исполняемый файл других файлов без использования секции ресурсов.
Файлы будут располагаться непосредственно в секции кода исполняемого файла.
Это значит что они будут загружаться в память при запуске программы.

В PureBasic, для размещения файлов в секции кода, существуют операторы **DataSection**, **IncludeBinary** и **EndDataSection**
Синтаксис такой:
Код:

```
DataSection
 Metka_1:
 IncludeBinary "Путь к файлу"
 Metka_2:
EndDataSection
```

После оператора **IncludeBinary** в кавычках указывается абсолютный или относительный путь к требуемому файлу.
Метки с именами Metka_1 и Metka_2 расположеные перед и после оператора **IncludeBinary** позволяют получить содержимое файла. Для этого достаточно узнать адрес метки в памяти.

Для примера, создадим небольшую электронную книгу из этих html файлов. Для этого понадобится создать окно и разместить там два веб браузера, один для странички навигации, а второй для страничек учебника. Кроме того, нам понадобится перехватывать клики по ссылкам в первом браузере, чтобы во первых, предотвратить загрузку данных в браузер по этим ссылкам, а во вторых чтобы узнать имя загружаемой странички и самим ещё загрузить из кодовой секции исполняемого файла.

Код программы приведён ниже

Код:

```
; Пример электронного учебника.
; Компилятор PureBasic

Procedure NavigationCallback(Gadget, Url$) ; В этой процедуре перехватываем
клики в браузере-навигаторе.
 If Url$<>""
 Url$=LCase(Url$)
 pos=FindString(Url$, "about:blank",1)
 If pos>0
 Url$=Right(Url$,Len(Url$)-pos-10)
 EndIf

 Select Url$ ; Узнаём какую страничку загрузить из памяти в браузер
 Case "ch1.html"
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch1, ?ch1End-?ch1))
 Case "ch2.html"
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch2, ?ch2End-?ch2))
 Case "ch3.html"
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch3, ?ch3End-?ch3))
```

```
 Case "ch4.html"
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch4, ?ch4End-?ch4))
 Case "ch5.html"
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch5, ?ch5End-?ch5))
 Case "ch6.html"
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch6, ?ch6End-?ch6))
 EndSelect

 EndIf
 ProcedureReturn 0 ; Запрет открытия ссылки браузером
EndProcedure

 ; Открываем окно
 If OpenWindow(0, 0, 0, 800, 500, "WebGadget",
#PB_Window_MinimizeGadget|#PB_Window_ScreenCentered|#PB_Window_Invisible)
 WebGadget(0, 10, 10, 150, 480, "") ; Браузер-навигатор
 SetGadgetItemText(0,#PB_Web_HtmlCode,PeekS(?h, ?hEnd-?h)) ; Загрузка из
памяти странички навигации
 SetGadgetAttribute(0,#PB_Web_NavigationCallback,@NavigationCallback()) ;
Установка процедуры-перехватчика кликов по ссылкам
 WebGadget(1, 180, 10, 780-180, 480, "") ; Браузер, отображающий выбираемые
странички
 SetGadgetItemText(1,#PB_Web_HtmlCode,PeekS(?ch1, ?ch1End-?ch1)) ; Загрузка
из памяти начальной странички
 HideWindow(0,0) ; Отображение окна

 Repeat ; Это главный цикл прогаммы - обработчик событий. Мы отслеживаем
только закрытие окна
 Until WaitWindowEvent() = #PB_Event_CloseWindow

 EndIf
End ; Типа всё, программа завершает работу


 ; Вот этот код при компиляции загружает файлы в исполняемый файл.
 ; Доступ к файлам как к памяти получаем с помощью меток
 DataSection
 h:
 IncludeBinary "h.html"
 hEnd:

 ch1:
 IncludeBinary "ch1.html"
 ch1End:

 ch2:
 IncludeBinary "ch2.html"
 ch2End:

 ch3:
 IncludeBinary "ch3.html"
 ch3End:

 ch4:
 IncludeBinary "ch4.html"
 ch4End:

 ch5:
 IncludeBinary "ch5.html"
 ch5End:

 ch6:
 IncludeBinary "ch6.html"
```

```
ch6End:
EndDataSection
```

Теперь разберёмся как этот код работает.
Процедура **NavigationCallback** вызывается при клике по ссылке в браузере-навигаторе и ей передаётся адрес ссылки через переменную строковую **Url$**. В процедуре анализируется адрес ссылки и загружается из памяти соответствующий Html файл.
Например, файл ch1.html загружает в браузер такая строка
Код:

```
SetGadgetItemText(1, #PB_Web_HtmlCode, PeekS(?ch1, ?ch1End-?ch1) )
```

С помощью функции PeekS производится загрузка текста из памяти (html код это текст). Адрес памяти и длина памяти вычисляется с помощью меток перед и после оператора **IncludeBinary**, о чём писалось выше. Для получения адреса метки, перед её именем нужно поставить символ **?**.
Подробнее об этом можно узнать http://pure-basic.narod.ru/docs/point.html
Функция SetGadgetItemText загружает текст в браузер в формате html, что задаётся с помощью константы-флага #PB_Web_HtmlCode.

Фактически программа начинает выполнятся с функции OpenWindow, открывающей окно. После открытия окна, создаётся браузер с идентификатором 0 и в него по умолчанию загружается код html странички ch1.html.
После этого, процедура **NavigationCallback** регистрируется как обработчик кликов этого браузера.

Далее создаётся ещё один браузер (ему присваивается идентификатор 1), в котором будут отображаться странички учебника.
Далее следует код
Код:

```
Repeat ; Это главный цикл программы - обработчик событий. Мы отслеживаем
только закрытие окна
 Until WaitWindowEvent() = #PB_Event_CloseWindow
```

являющийся главным циклом программы, в котором обрабаотывается только одно событие, закрытие окна.

В конце исходного текста, находятся операторы **DataSection**, **IncludeBinary** и **EndDataSection** загружающие html-файлы в исполняемый файл программы на этапе компиляции.

Скачать все требуемые файлы http://pure-basic.narod.ru/docs/e_book.rar

16

Разработка кроссплатформенного приложения

В этой статье будет рассмотрено создание программы, работающей в Windows и Linux. Скорее всего она будет работать на платформах MasOS X и AmigaOS, но проверить это не получилось из-за отсутствия этих операционнок в моём компе.
Не нужно думать что один и тот же скомпилированный исполняемый файл получится

запустить на разных платформах - это фантастика!
Кроссплатформенность существует только на уровне исходного текста программы и для создания исполняемого файла для требуемой платформы, нужно воспользоваться компилятором для этой платформы. Демонстрационные версии компиляторов для разных платформ, можно скачать здесь http://www.purebasic.com/download.php
Собственно кроссплатформенность достигается (точнее существует без изрядной правки исходника) благодаря функциям среды, описание которых можно найти в справке, поставляемой с дистрибутивом PureBasic или в он-лайн справке http://www.purebasic.com/documentation/index.html
Примерно половина этих функций - немного облагороженные API функции соответствующей платформы. Но фишка в том, что эти функции среды, компиляторы для разных платформ, преобразуют в различный код. Вот и получается, что один и тот же исходник можно скомпилировать под разные платформы.
Но бывают случаи, когда для каждой платформы нужны выполнить свой код.
Есть два варианта, использовать несколько исходных текстов для разных платформ, что не всегда удобно, или использовать условную компиляцию.
В последнем случае используются операторы **CompilerIf**, **CompilerElse** и **CompilerEndIf**
Код:
```
 CompilerIf #PB_Compiler_OS = #PB_OS_Windows
 здесь должен быть код специально для Windows
 CompilerElse
 здесь должен быть код для других платформ
 CompilerEndIf
```

Эти операторы действуют только на этапе компиляции, добавляя в исполняемый файл только требуемый код.

Есть ещё операторы **CompilerSelect**, **CompilerCase**, **CompilerDefault**, **CompilerEndSelect**.

Код:
```
CompilerSelect #PB_Compiler_OS
 CompilerCase #PB_OS_Windows
 здесь должен быть код специально для Windows
 CompilerCase #PB_OS_Linux
 здесь должен быть код специально для Linux
 CompilerCase #PB_OS_MacOS
 здесь должен быть код специально для MacOS
 CompilerDefault
 здесь должен быть код для остальных платформ
CompilerEndSelect
```

В качестве примера кроссплатформенной программы был использован несложный текстовый редактор, в котором для упрощения кода исключены некоторые функции, например меню "Правка".

Исходный текст редактора
Код:
```
 ; Эта процедура открывает файл, указанный в переменной File и загружает
данные из него в редактор.
Procedure LoadFile(File.s)
 If ReadFile(0, File) ; Открытие файла.
 FileSize=Lof(0) ; Определение размера файла в байтах.
 FormatFile=ReadStringFormat(0) ; Определение кодировки файла (Ascii, UTF8
или Unicode).
 *mem=AllocateMemory(FileSize+1) ; Выделение памяти на 1 байт больше размера
файла.
```

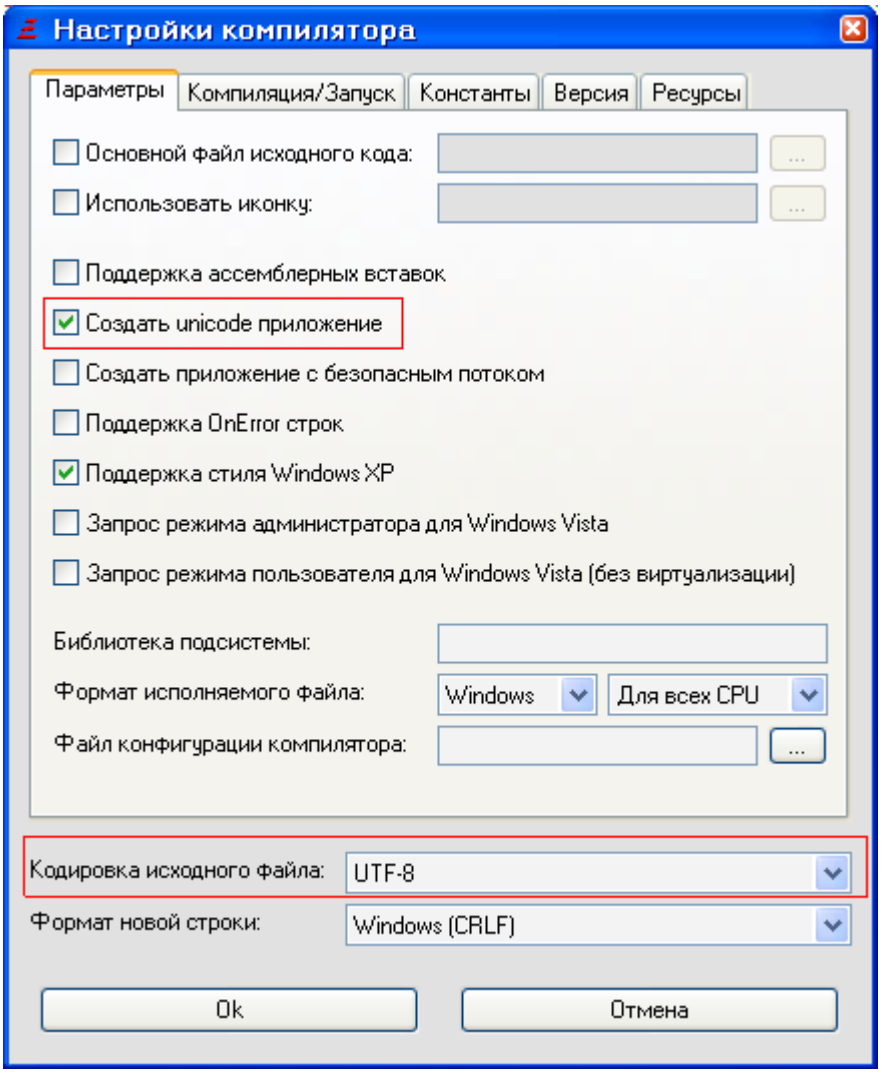
```
 ReadData(0, *mem, FileSize) ; Копирование данных из файла в память.
 SetGadgetText(1, PeekS(*mem,FileSize, FormatFile)) ; Преобразование
кодировки и загрузка текста в редактор.
 FreeMemory(*mem) ; Освобождение памяти.
 CloseFile(0) ; Закрытие файла.
 StatusBarText(1,0,File) ; Отображение пути к файлу в строке состояния.
 Else
 MessageRequester("Ошибка", "Не удалось открыть файл")
 EndIf
EndProcedure

 ; Эта процедура сохраняет данные из редактора в файл.
Procedure SaveFile()
 ; Создание стандартного окна сохранения файла.
 File.s=SaveFileRequester("Сохраняем файл","","Текстовые файлы
(*.txt)|*.txt|Все файлы (*.*)|*.*",0)
 If File<>""
 If GetExtensionPart(File)="" And SelectedFilePattern()=0 ; Если не заданно
расширение файла
 File=File+".txt" ; добавляем его
 EndIf
 If CreateFile(0,File) ; Создание файла на диске
 Text.s=GetGadgetText(1) ; Копирование текста из редактора в строковую
переменную Text
 CompilerIf #PB_Compiler_OS = #PB_OS_Linux ; Это код для операционки Linux
 WriteStringFormat(0, #PB_Unicode) ; Запись в файл специальной метки,
сообщающей что кодировка Unicode
 FormatFile=#PB_Unicode
 CompilerElse ; Это код для операционок Windows, MacOS и AmigaOS
 FormatFile=#PB_Ascii
 CompilerEndIf
 WriteString(0, Text, FormatFile) ; Запись текста в файл в требуемой
кодировке
 CloseFile(0) ; Закрытие файла.
 Else
 MessageRequester("Ошибка", "Не удалось сохранить файл")
 EndIf
 EndIf
EndProcedure

ProgPath.s=GetPathPart(ProgramFilename()) ; Получаем путь к исполняемому
файлу

Gosub LoadSetting ; Вызов подпрограммы, читающей настройки программы из файла
SettingEdit.ini

If Window_X<2 : Window_X=100 : EndIf
If Window_Y<2 : Window_Y=100 : EndIf

 ; Открытие главного окна
OpenWindow(1,Window_X,Window_Y,Window_Width,Window_Height,"Текстовый
редактор",#PB_Window_MinimizeGadget|#PB_Window_MaximizeGadget|#PB_Window_Size
Gadget|#PB_Window_Invisible)
 SmartWindowRefresh(1,1) ; Активация функции, уменьшающей мерцания окна при
изменении его размеров

 If CreateMenu(0,WindowID(1)) ; Создание главного меню
 MenuTitle("Файл")
 MenuItem(1,"Открыть"+Chr(9)+"Ctrl+O")
 MenuItem(2,"Сохранить"+Chr(9)+"Ctrl+S")
 MenuBar()
 MenuItem(3,"Выход")
 MenuTitle("Формат")
 MenuItem(4,"Шрифт")
```

```
 MenuHeight=MenuHeight() ; Высота меню
 EndIf

 If CreateToolBar(1,WindowID(1)) ; Создание панели инструментов
 ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
 ToolBarToolTip(1,1,"Открыть файл")
 ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
 ToolBarToolTip(1,2,"Сохранить файл")
 ToolBarHeight=ToolBarHeight(1) ; Высота панели инструментов
 EndIf

 ; Регистрация "горячих клавиш", дублирующих меню
 AddKeyboardShortcut(1, #PB_Shortcut_Control | #PB_Shortcut_O, 1)
 AddKeyboardShortcut(1, #PB_Shortcut_Control | #PB_Shortcut_S, 2)


 If CreateStatusBar(1,WindowID(1)) ; Создание строки состояния
 AddStatusBarField(#PB_Ignore) ; Создание одного раздела на всю строку
состояния
 StatusBarHeight = StatusBarHeight(1) ; Высота строки состояния
 EndIf

 CompilerIf #PB_Compiler_OS = #PB_OS_Linux ; Код для Linux
 EditorX=2 ; Верхняя позиция редактора в окне
 CompilerElse
 EditorX=ToolBarHeight+2 ; Верхней позиция редактора смещена вниз на высоту
панели инструментов
 CompilerEndIf
 ; Собственно редактор текста
 EditorGadget(1,1,EditorX, Window_Width-2, Window_Height-StatusBarHeight-
MenuHeight-ToolBarHeight-4)
 SetGadgetColor(1,#PB_Gadget_FrontColor, $950121) ; Цвет текста
 SetGadgetColor(1,#PB_Gadget_BackColor, RGB(237, 255, 246)) ; Фон

 LoadFont(1, FontName.s, FontSize, FontStyle) ; Шрифт для редактора
 SetGadgetFont(1,FontID(1))

 File.s=ProgramParameter() ; Командная строка переданная программе при старте
 If File<>"" And FileSize(File)>=0 ; Передан путь к файлу
 LoadFile(File) ; Загружаем файл
 EndIf

 ; Активация функций, позволяющих открывать файл просто перетащив его в окно
программы
 EnableWindowDrop(1, #PB_Drop_Files, #PB_Drag_Link)
 EnableGadgetDrop(1, #PB_Drop_Files, #PB_Drag_Link)

 HideWindow(1,0) ; Отображение окна

 Repeat ; Главный цикл Repeat - Until

 Event=WaitWindowEvent() ; Получаем идентификатор события в программе

 If Event=#PB_Event_SizeWindow ; Размер окна изменился
 ; Изменение размеров редактора
 ResizeGadget(1, #PB_Ignore, #PB_Ignore, WindowWidth(1)-2, WindowHeight(1)-
StatusBarHeight-MenuHeight-ToolBarHeight-2)

 ElseIf Event=#PB_Event_WindowDrop Or Event=#PB_Event_GadgetDrop ; На окно
перетащали файл
 If EventDropAction()=#PB_Drag_Link
 File=EventDropFiles() ; Получаем путь к файлу
 If FileSize(File)>=0 ; Файл существует
 LoadFile(File) ; Загружаем файл
```

```
 EndIf
 EndIf

 ElseIf Event=#PB_Event_Menu ; Событие в меню
 Menu=EventMenu() ; Определение выбранного пункта меню
 Select Menu
 Case 1 ; Пункт "Открыть"
 File.s=OpenFileRequester("Открываем файл","","Текстовые файлы
(*.txt)|*.txt|Все файлы (*.*)|*.*",0)
 If File<>""
 LoadFile(File)
 EndIf

 Case 2 ; Пункт "Сохранить"
 SaveFile()

 Case 3 ; Пункт "Выход"
 Break

 Case 4 ; Пункт "Шрифт"
 If FontRequester(FontName.s, FontSize,0,0, FontStyle)
 FontName=SelectedFontName()
 FontSize=SelectedFontSize()
 FontStyle=SelectedFontStyle()
 LoadFont(1, FontName, FontSize, FontStyle)
 SetGadgetFont(1,FontID(1))
 EndIf

 EndSelect
 EndIf

 Until Event=#PB_Event_CloseWindow ; Прерывание главного цикла при закрытии
окна
 Gosub SaveSetting ; Вызов подпрограммы, сохраняющей настройки программы ф
файле SettingEdit.ini
 End

 LoadSetting: ; Подпрограмма читающая из файла SettingEdit.ini настройки
программы
 OpenPreferences(ProgPath+"SettingEdit.ini")
 PreferenceGroup("Window")
 Window_X=ReadPreferenceLong("Window_X", 200)
 Window_Y=ReadPreferenceLong("Window_Y", 200)
 Window_Width=ReadPreferenceLong("Window_Width", 500)
 Window_Height=ReadPreferenceLong("Window_Height", 400)
 PreferenceGroup("Editor")
 CompilerIf #PB_Compiler_OS = #PB_OS_Windows
 FontName.s=ReadPreferenceString("FontName","Lucida Console")
 CompilerElse
 FontName.s=ReadPreferenceString("FontName","Liberation Mono")
 CompilerEndIf
 FontSize=ReadPreferenceLong("FontSize", 10)
 FontStyle=ReadPreferenceLong("FontStyle", 0)
 ClosePreferences()
Return


SaveSetting: ; Подпрограмма сохраняющая в файле SettingEdit.ini настройки
программы
 If CreatePreferences(ProgPath+"SettingEdit.ini")
 PreferenceGroup("Window")
 WritePreferenceLong("Window_X", WindowX(1))
 WritePreferenceLong("Window_Y", WindowY(1))
 WritePreferenceLong("Window_Width", WindowWidth(1))
```

```
 WritePreferenceLong("Window_Height", WindowHeight(1))
 PreferenceGroup("Editor")
 WritePreferenceString("FontName", FontName)
 WritePreferenceLong("FontSize", FontSize)
 WritePreferenceLong("FontStyle", FontStyle)
 ClosePreferences()
 EndIf
Return
```

Значит, запускаем PureBasic. При запуске будет создан новый проект, который надо сохранить на диске под любым именем. Далее нужно немного изменить свойства проекта. Для этого в меню Компилятор выбираем пункт Настройки компилятора и в открывшемся окне отмечаем пункт Создать unicode приложение и в выпадающем списке Кодировка исходного файла выбираем пункт UTF-8.
Это необходимо для поддержки кириллицы в Linux, а для Windows это не имеет существенного значения, разве что программа перестанет запускаться в Win9x.



После этого можно скопировать исходный текст в редактор кода PureBasic и произвести компиляцию.

Теперь рассмотрим как устроена программа и как она работает.
В начале программы есть две процедуры с именами **LoadFile** и **SaveFile**. Несмотря на то, что они находятся в начале программы, их код будет выполнен лишь при непосредственном вызове процедур!

Процедура **LoadFile** открывает файл и загружает его в редактор. Путь к файлу передаются через аргумент процедуры - строковую переменную **File**.
Функция ReadFile пытается открыть файл. В случае успешного открытия файла, функция Lof возвращает размер файла в байтах, а функция ReadStringFormat кодировку файла. Программа сама распознаёт и поддерживает кодировки Ascii, UTF8 и Unicode.
Далее функция AllocateMemory выделяет участок памяти, размером, но 1 байт большим, размера файла.
Затем с помощью функции ReadData копируется содержимое файла в эту память.
Далее функция PeekS считывает из памяти текст в требуемой кодировке (её ранее получили с помощью ReadStringFormat) и передаёт текст функции SetGadgetText которая загружает этот текст в редактор. После этого память освобождается, файл закрывается и работа процедуры завершается.

В процедуре **SaveFile** производится сохранение текста файле. В первую очередь с помощью функции SaveFileRequester создаётся стандартное окно сохранения файла (правда этот стандарт меняется в каждой версии операционки, а про разные платформы я вообще молчу). Если место сохранения файла выбрано, то в строковой переменной **File** будет полный путь к файлу, включая его имя и расширение. Следующие несколько строк проверяют есть ли у выбранного файла расширение, если нет, то к имени файла добавляется расширение txt. После этого функция CreateFile пытается создать пустой файл. В случае успеха, с помощью функции GetGadgetText считывается весь текст из редактора и помещается в строковую переменную **Text**. Далее следуют операторы условной компиляции, с помощью которых задаётся кодировка сохраняемого файла. Для Linux будет кодировка Unicode, а для остальных платформ, в том числе и Windows будет кодировка Ascii. Функция WriteString сохраняет текст в файле как одну большую строку в заданной кодировке. Далее файл закрывается и после этого работа процедуры завершается.

Далее находится код, который начнёт исполняться сразу после запуска программы.
В первую очередь с помощью строки **ProgPath.s=GetPathPart(ProgramFilename())** определяется путь к запущенному исполняемому файлу. Этот путь нужен для работы с файлом настроек SettingEdit.ini, который находится в одной папке с исполняемым файлом.
Далее с помощью оператора **Gosub** вызывается подпрограмма, начинающаяся с метки LoadSetting, которая открывает файл с настройками и загружает их.
Далее создаётся невидимое окно (за невидимость отвечает флаг #PB_Window_Invisible в функции OpenWindow).
Затем создаётся меню, панель инструментов, регистрируются "горячие клавиши", создаётся строка состояния.
После этого создаётся RTF редактор при помощи функции EditorGadget. Функция SetGadgetColor задаёт цвет текста и фона редактора.
Далее код
Код:

```
File.s=ProgramParameter() ; Командная строка переданная программе при старте
If File<>"" And FileSize(File)>=0 ; Передан путь к файлу
LoadFile(File) ; Загружаем файл
EndIf
```

проверяет наличие пути к файлу в командной строке. Если есть путь к файлу, то вызывается процедура **LoadFile**, загружающая файл.

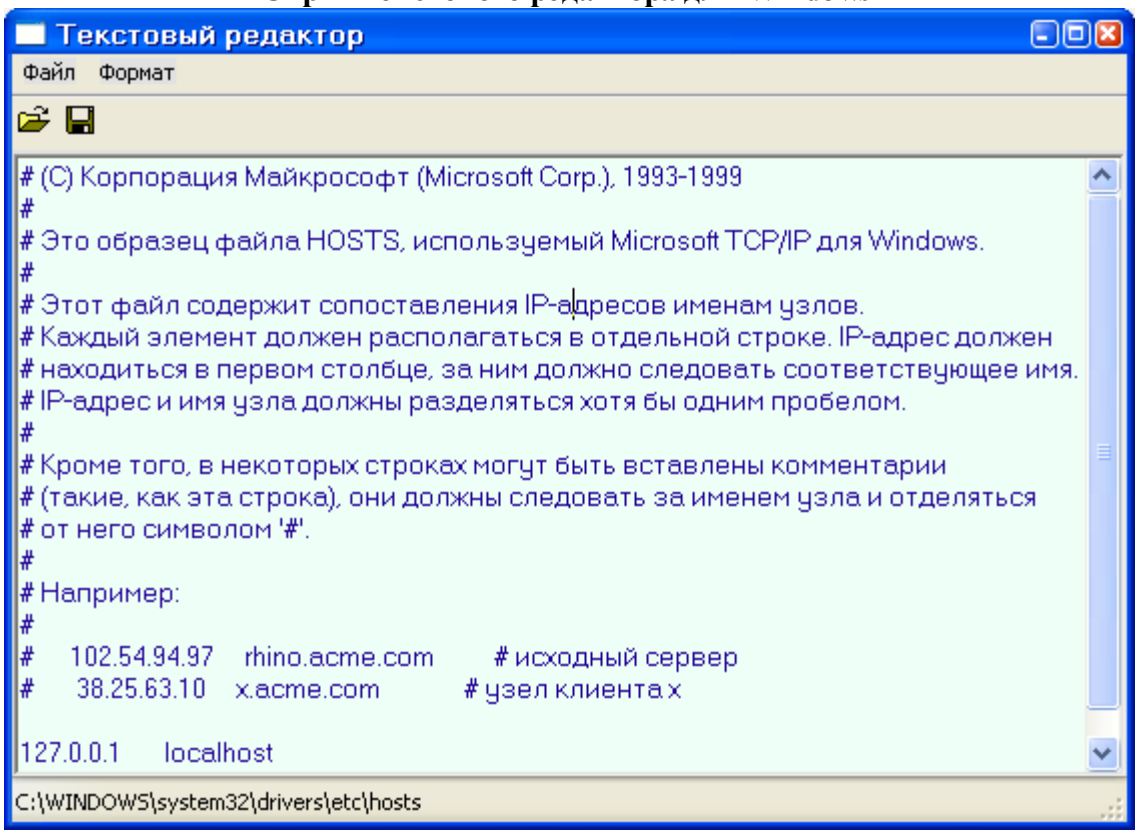
Код
Код:
```
EnableWindowDrop(1, #PB_Drop_Files, #PB_Drag_Link)
EnableGadgetDrop(1, #PB_Drop_Files, #PB_Drag_Link)
```

активирует опцию открытия файла простым перетаскиванием его в окно программы.
Функция HideWindow отображает окно.

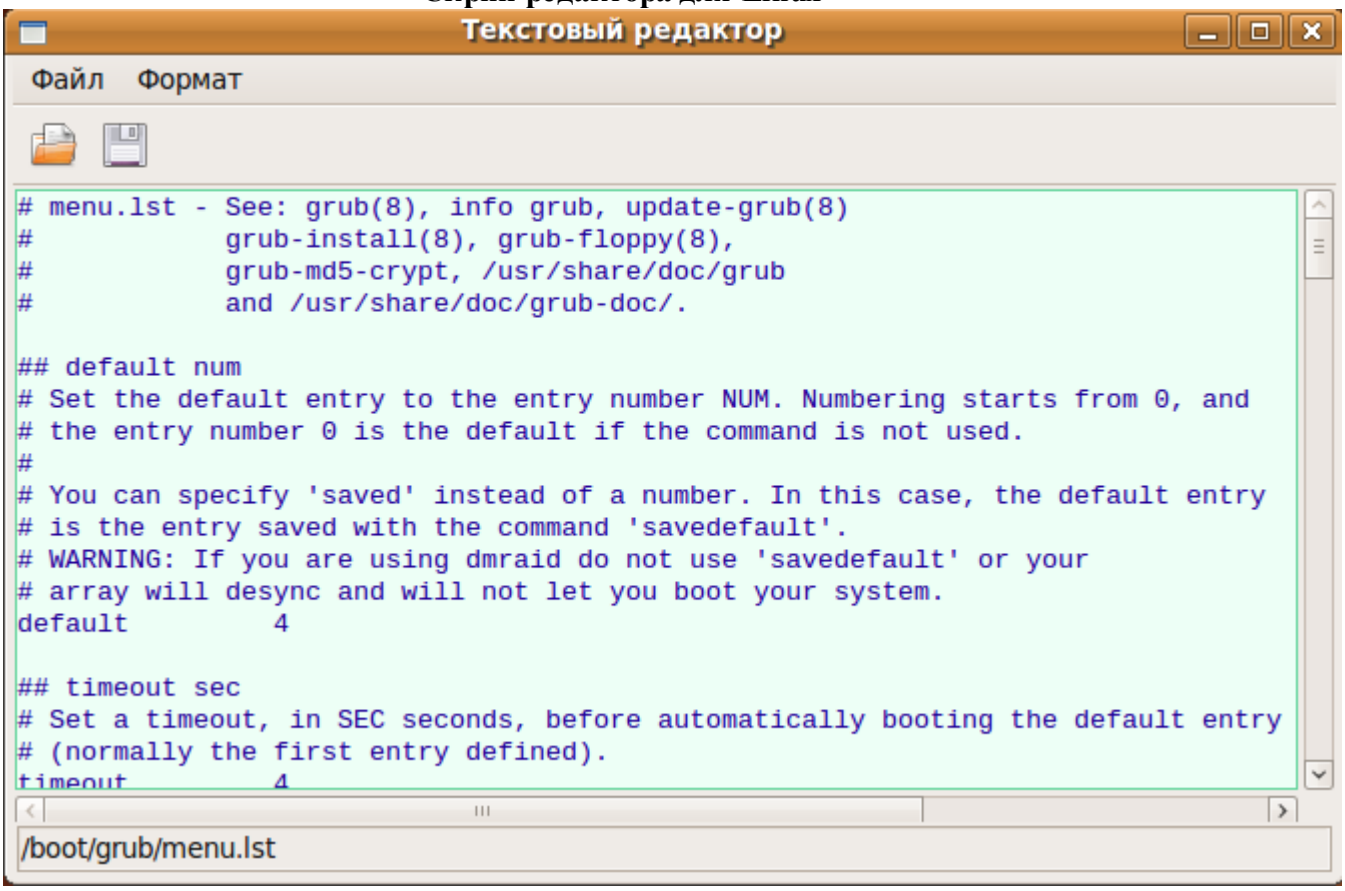
Далее находится главный цикл программы созданный операторами **Repeat** и **Until**, код которого будет выполнятся большую часть времени работы программы. Выполнение кода этого цикла прервётся при событии закрытия окна.
В этом цикле происходит обработка событий программы. Функция WaitWindowEvent возвращает идентификатор события, который помещается в переменную **Event**. Далее с помощью операторов **If**, **ElseIf** и **EndIf** анализируются идентификаторы событий и выполняются требуемые действия.
Если изменился размер окна, то появится событие #PB_Event_SizeWindow что приведёт к выполнению функции ResizeGadget, изменяющей размеры редактора.
Событие #PB_Event_WindowDrop или #PB_Event_GadgetDrop произойдёт если перетащить файл на окно программы и "бросить" его там. Эти события приведут к выполнению этого кода
Код:
```
If EventDropAction()=#PB_Drag_Link
File=EventDropFiles() ; Получаем путь к файлу
If FileSize(File)>=0 ; Файл существует
LoadFile(File) ; Загружаем файл
EndIf
EndIf
```

При клике по пункту меню, произойдёт событие #PB_Event_Menu
При закрытии окна произойдёт событие #PB_Event_CloseWindow и условие в операторе **Until** будет выполнено, а это значит что главный цикл программы будет прерван. Вызов подпрограммы SaveSetting, сохранит текущие настройки программы в файле SettingEdit.ini, а затем, оператор **End** завершит работу программы. При этом освобождаются все ресурсы программы, в т. ч. закрывается окно, так что нет необходимости программно его закрывать.

Скрин текстового редактора для Windows



Скрин редактора для Linux



Скачать исходный текст редактора и скомпилированные исполняемые файлы для Windows и Linux можно http://pure-basic.narod.ru/docs/PB_Editor.rar

Все материалы взяты с сайта http://pure-basic.narod.ru**.**

Собрал всё в кучу что было для начинающих.

Может кому то и понадобится, а потом и появится

нормальная литература по теме.

Все вопросы и информация , просьба, на rostik01@mail.ru

Rostik