1ntr0 25 октября в 06:11

Классы матриц и векторов в Delphi

Программирование, ООП, Математика, Алгоритмы, Delphi

В этой статье рассматривается проектирование типов для работы с объектами линейной алгебры: векторами, матрицами, кватернионами. Показано классическое применение механизма перегрузки стандартных операций, использование приёма «Copy Write» и аннотаций.

Работая в сфере математического моделирования, мне часто приходится сталкиваться с вычислительными алгоритмами, в которы используются операции над матрицами, векторами, кватернионами. К удивлению, обнаружил, что, несмотря на возможности современной среды разработки, коллеги-программисты зачастую используют процедурный подход в решении подобных задач. Та чтобы вычислить произведение матрицы и вектора, описываются типы и функции вроде этих:

```
TVec3 = array[1..3] of Extended;
TMatrix3x3 = array[1..3, 1..3] of Extended;
function MVMult(M: TMatrix3x3; V: TVec3): TVec3;
```

Предлагается использовать объектный подход, который, в свою очередь, подразумевает совместное размещение данных и методо обработки. Посмотрим, какие возможности предоставляет Delphi для решения подобного класса задач в объектном виде. Проекти структуры объектов будем исходить из следующих требований:

- Версия среды разработки Delphi XE7
- Для числовых данных использовать тип Extended, как наиболее точный;
- Использовать динамические массивы для хранения данных, т.к. размеры векторов и матриц могут быть любыми и в отладчике удобно смотреть;
- Элементы векторов и матриц нумеруются с 1, элементы кватернионов с 0;
- Для вычислений использовать операции +, -, *, /;
- Обеспечить возможность передачи по значению и копирование векторов и матриц операцией ":=";
- Обеспечить возможность автоматизированной инициализации векторов и матриц по заранее заданным размерам.

Проектирование

Для удобства определим вспомогательные типы:

```
TAbstractVector = array of Extended;
TAbstractMatrix = array of array of Extended;
```

Теперь определим структуры кватерниона, вектора и матрицы:

```
TQuaternion = record
private
  FData: array[0..3] of Extended;
  procedure SetElement(Index: Byte; Value: Extended);
  function GetElement(Index: Byte): Extended;
public
  property Element[Index: Byte]: Extended read GetElement write SetElement; default;
end;

TVector = record
private
  FData: TAbstractVector;
  FCount: Word;
  procedure SetElement(Index: Word; Value: Extended);
```

```
  function GetElement(Index: Word): Extended;
public
  constructor Create(ElementsCount: Word);
  property Count: Word read FCount;
  property Elements[Index: Word]: Extended read GetElement write SetElement; default;
end;

TMatrix = record
private
  FData: TAbstractMatrix;
  FRowsCount: Word;
  FColsCount: Word;
  procedure SetElement(Row, Col: Word; Value: Extended);
  function GetElement(Row, Col: Word): Extended;
public
  constructor Create(RowsCount, ColsCount: Word);
  property RowCount: Word read FRowsCount;
  property ColCount: Word read FColsCount;
  property Elements[Row, Col: Word]: Extended read GetElement write SetElement; default;
end;
```

Мы используем именно *record*, т.к. перегрузка операций для конструкции *class* в Delphi не разрешена. К тому же у объектов record полезное свойство — их данные разворачиваются в памяти по месту объявления, другими словами, объект record не является ссыл на экземпляр в динамической памяти.
Однако, в нашем случае элементы векторов и матриц будут храниться в динамическом массиве, объект которого является ссылкой Поэтому будет удобно использовать явные конструкторы. Они выполняют инициализацию внутренних полей, выделяя память под требуемое число элементов:

```
constructor TVector.Create(ElementsCount: Word);
begin
  FCount := ElementsCount;
  FData := nil;
  SetLength(FData, FCount);
end;

constructor TMatrix.Create(RowsCount, ColsCount: Word);
begin
  FRowsCount := RowsCount;
  FColsCount := ColsCount;
  FData := nil;
  SetLength(FData, FRowsCount, FColsCount);
end;
```

Кватерниону на данном этапе конструктор не требуется, т.к. он хранит данные в статическом массиве и разворачивается в памяти месту своего объявления.
Для доступа к элементам здесь служат свойства-индексаторы, их удобно сделать default, чтобы опускать имя. Доступ к запрашиваемому элементу происходит после проверки его индекса на допустимые значения. Показана реализация для TVector:

```
function TVector.GetElement(Index: Word): Extended;
begin
  {$R+}
  Result := FData[Pred(Index)];
end;

procedure TVector.SetElement(Index: Word; Value: Extended);
begin
  {$R+}
  FData[Pred(Index)] := Value;
end;
```

На этом этапе, чтобы создавать наши объекты, придется использовать такой код:

```
var
  V: TVector;
. . .
V := TVector.Create(3);
```

```
V[1] := 1;
V[2] := 2;
V[3] := 3;
```

Практика показала, что полезно иметь средства, позволяющие использовать более лаконичный синтаксис для создания вектора ил матрицы. Для этого добавим дополнительные конструкторы, а также реализуем операцию неявного приведения, которая позволит перегрузить ":=".

```
TQuaternion = record
public
  . . .
  constructor Create(Q: TAbstractVector);
  class operator Implicit(V: TAbstractVector): TQuaternion;
end;

TVector = record
public
  . . .
  constructor Create(V: TAbstractVector); overload;
  class operator Implicit(V: TAbstractVector): TVector;
end;

TMatrix = record
public
  . . .
  constructor Create(M: TAbstractMatrix); overload;
  class operator Implicit(M: TAbstractMatrix): TMatrix;
end;
```

И реализация:

```
constructor TQuaternion.Create(Q: TAbstractVector);
begin
  if Length(Q) <> 4 then
    raise EMathError.Create(WRONG_SIZE);
  Move(Q[0], FData[0], SizeOf(FData));
end;

class operator TQuaternion.Implicit(V: TAbstractVector): TQuaternion;
begin
  Result.Create(V);
end;

constructor TVector.Create(V: TAbstractVector);
begin
  FCount := Length(V);
  FData := Copy(V);
end;

class operator TVector.Implicit(V: TAbstractVector): TVector;
begin
  Result.Create(V);
end;

constructor TMatrix.Create(M: TAbstractMatrix);
var
  I: Integer;
begin
  FRowsCount := Length(M);
  FColsCount := Length(M[0]);
  FData := nil;
  SetLength(FData, FRowsCount, FColsCount);
  for I := 0 to Pred(FRowsCount) do
    FData[I] := Copy(M[I]);
end;

class operator TMatrix.Implicit(M: TAbstractMatrix): TMatrix;
begin
  Result.Create(M);
end;
```

Теперь, чтобы создать и инициализировать вектор или матрицу, достаточно написать:

```
var
  V: TVector;
  M: TMatrix;
. . .
  V := [4, 5, 6];
  //
  M := [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]];
```

Перегрузка операций

Здесь, для примера, будет реализована только перегрузка операции * для умножения матрицы на вектор. Остальные операции мо
посмотреть в прикрепленном к статье файле. А полный перечень возможностей по перегрузке — здесь.

```
TMatrix = record
public
  . . .
  class operator Multiply(M: TMatrix; V: TVector): TVector;
end;

class operator TMatrix.Multiply(M: TMatrix; V: TVector): TVector;
var
  I, J: Integer;
begin
  if (M.FColsCount <> V.FCount) then
    raise EMathError.Create(WRONG_SIZE);
  Result.Create(M.FRowsCount);
  for I := 0 to M.FRowsCount - 1 do
    for J := 0 to M.FColsCount - 1 do
      Result.FData[I] := Result.FData[I] + M.FData[I, J] * V.FData[J];
end;
```

Первый аргумент метода Multiply() — матрица слева от знака *, второй аргумент — вектор-столбец, находящийся справа от знака *
Результатом произведения является новый вектор, объект которого создается в процессе вычисления. В случае несовпадения
количества столбцов матрицы и числа элементов вектора возбуждается исключение. Вот как выглядит использование этой операц
программе:

```
var
  V, VResult: TVector;
  M: TMatrix;
. . .
  VResult := M * V;
```

Удобно применять функции-обертки, чтобы конструировать анонимные вектора и матрицы из литералов массивов «налету»:

```
function TVec(V: TAbstractVector): TVector;
begin
  Result.Create(V);
end;

function TMat(M: TAbstractMatrix): TMatrix;
begin
  Result.Create(M);
end;

function TQuat(Q: TAbstractVector): TQuaternion;
begin
  Result.Create(Q);
end;
```

Использование оберток выглядит следующим образом. Показан эквивалент выражению из предыдущего примера:

```
  V := TMat([[1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]]) * TVec([4, 5, 6]);
```

Кроме стандартных операций в типы наших объектов полезно добавить специфические методы, например, транспонирование или обращение. Ниже приведен пример метода инвертирования (обращения) матрицы. Несмотря на свои размеры, он наиболее быстр всех, виденных мной (на языках высокого уровня).

```
TMatrix = record
public
  . . .
  function Inv: TMatrix;
end;

function TMatrix.Inv: TMatrix;
var
  Ipiv, Indxr, Indxc: array of Integer;
  DimMat, I, J, K, L, N, ICol, IRow: Integer;
  Big, Dum, Pivinv: Extended;
begin
  // Алгоритм Жордана.
  if (FRowsCount <> FColsCount) then
    raise EMathError.Create(NOT_QUAD);
  Result := Self;
  DimMat := FRowsCount;
  SetLength(Ipiv, DimMat);
  SetLength(Indxr, DimMat);
  SetLength(Indxc, DimMat);
  IRow := 1;
  ICol := 1;
  for I := 1 to DimMat do
  begin
    Big := 0;
    for J := 1 to DimMat do
      if (Ipiv[J - 1] <> 1) then
        for K := 1 to DimMat do
          if (Ipiv[K - 1] = 0) then
            if (Abs(Result[J, K]) >= Big) then
            begin
              Big := Abs(Result[J, K]);
              IRow := J;
              ICol := K;
            end;
    Ipiv[ICol - 1] := Ipiv[ICol - 1] + 1;
    if (IRow <> ICol) then
      for L := 1 to DimMat do
      begin
        Dum := Result[IRow, L];
        Result[IRow, L] := Result[ICol, L];
        Result[ICol, L] := Dum;
      end;
    Indxr[I - 1] := IRow;
    Indxc[I - 1] := ICol;
    if Result[ICol, ICol] = 0 then
      raise EMathError.Create(SINGULAR);
    Pivinv := 1.0 / Result[ICol, ICol];
    Result[ICol, ICol] := 1.0;
    for L := 1 to DimMat do
      Result[ICol, L] := Result[ICol, L] * Pivinv;
    for N := 1 to DimMat do
      if (N <> ICol) then
      begin
        Dum := Result[N, ICol];
        Result[N, ICol] := 0.0;
        for L := 1 to DimMat do
          Result[N, L] := Result[N, L] - Result[ICol, L] * Dum;
      end;
  end;
```

```
  for L := DimMat downto 1 do
    if (Indxr[L - 1] <> Indxc[L - 1]) then
      for K := 1 to DimMat do
      begin
        Dum := Result[K, Indxr[L - 1]];
        Result[K, Indxr[L - 1]] := Result[K, Indxc[L - 1]];
        Result[K, Indxc[L - 1]] := Dum;
      end;
end;
```

Копирование по значению

Использование динамических массивов для хранения элементов векторов и матриц приводит к тому, что при попытке копировать целиком в объекте-приёмнике (том, что слева от ":=") создается копия ссылки на этот динамический массив.
Например, попытка сохранить значение матрицы M после вычисления выражения приведет к инвертированию так же и матрицы M

```
var
  M, MStore: TMatrix;
. . .
  MStore := M;
  M := M.Inv;
```

Для того, чтобы корректно реализовать копирование по значению, используем тот факт, что по отрицательному смещению от адре первого элемента динамического массива, наряду со значением длины, хранится счетчик ссылок на этот массив. Если значение счетчика 0, то менеджер памяти этот массив освобождает. Если значение счетчика 1, это означает, что существует только одна сс на экземпляр массива в памяти.
Следовательно при копировании мы должны проанализировать значение счетчика и, если оно больше 1, то создать полноценную к массива, скопировав его в объект-приёмник *поэлементно*. Ниже представлен код функции, которая возвращает True только в том случае, когда значение счетчика ссылок переданного во входном параметре динамического массива превышает 1.

```
{$POINTERMATH ON}
function NotUnique(PArr: PCardinal): Boolean;
begin
  Result := (PArr - 2)^ > 1;
end;
```

В какой момент следует выполнять полное копирование? Это достаточно дорогая по времени операция, поэтому нет смысла выпо её при обращении к элементу вектора\матрицы на чтение. Будь у нас хоть тысяча ссылок на оригинал, если сам он не подвергаетс никаким изменениям, то все они остаются одинаковыми. Следовательно, копировать нужно только при обращении к элементу на з Для этого модифицируем методы SetElement() для векторов и матриц, добавив в начале проверку на уникальность экземпляра мас FData:

```
procedure TVector.SetElement(Index: Word; Value: Extended);
begin
  {$R+}
  CheckUnique;
  FData[Pred(Index)] := Value;
end;

procedure TVector.CheckUnique;
begin
  if NotUnique(@FData) then
    FData := Copy(FData);
end;

procedure TMatrix.SetElement(Row, Col: Word; Value: Extended);
begin
  {$R+}
  CheckUnique;
  FData[Pred(Row), Pred(Col)] := Value;
end;

procedure TMatrix.CheckUnique;
var
  I: Integer;
```

```
begin
  if NotUnique(@FData) then
    begin
      FData := Copy(FData);
      for I := 0 to Pred(FRowsCount) do
        FData[i] := Copy(FData[i]);
    end;
end;
```

Таким образом, при попытке изменить значение элемента произойдет проверка на уникальность ссылки, и, если таковая не подтвердится, будет создана поэлементная копия, в которую и будет внесено изменение.

Аннотации и автоматическая инициализация

К элементам векторов и матриц следует обращаться только после выделения для них памяти. Значения элементов хранятся в динамическом массиве, размеры которого устанавливаются в конструкторе объекта. Неявный вызов конструктора может произойт инициализации объекта, либо в процессе вычисления выражения.

```
var
  V: TVector;
  M: TMatrix;
begin
  // V[1] := 1;                  // Ошибка: объект не создан
  V := TVector.Create(4);        // Явный вызов конструктора
  M := TMatrix.Create(4, 4);     // Явный вызов конструктора
  // V := [1, 0, 0, 0];          // Неявный вызов конструктора
  // V := M * TVec([1, 0, 0, 0]); // Неявный вызов конструктора
  V[1] := 1;                     // Корректное обращение к элементу: объект создан
```

Использование неявных конструкторов может привести к ошибкам, когда, рано или поздно, будет допущено обращение к элемент несозданного объекта. По правилам хорошего тона, конструктор следует вызывать явно.
Но как быть, если векторов и матриц в нашей программе сотни и тысячи? Рассмотрим описание класса, использующего множеств векторов и матриц в качестве своих полей.

```
TMovement = record
  R: TVector;
  V: TVector;
  W: TVector;
  Color: TVector;
end;

TMovementScheme = class
private
  FMovement: array[1..100] of TMovement;
  FOrientation: TMatrix;
end;
```

Требуется разработать способ автоматизированной инициализации всех полей типа TVector и TMatrix: выделить память для вектор матриц в соответствии с нужным количествам элементов и размерами. В этом нам поможет механизм аннотаций (или атрибутов, в терминах Delphi) — средство, которое позволяет дополнять типы произвольными метаданными. Так, для каждого вектора должно б заранее известно количество его элементов, для матрицы — число строк и столбцов.
Создадим класс, инкапсулирующий данные о размерностях, по правилам создания классов атрибутов.

```
TDim = class(TCustomAttribute)
private
  FRowCount: Integer;
  FColCount: Integer;
public
  constructor Create(ARowCount: Integer; AColCount: Integer = 0); overload;
  property RowCount: Integer read FRowCount;
  property ColCount: Integer read FColCount;
end;

constructor TDim.Create(ARowCount: Integer; AColCount: Integer = 0);
begin
```

```
  FRowCount := ARowCount;
  FColCount := AColCount;
end;
```

Конструктор получает число строк и столбцов, а в случае вектора можем обойтись только числом строк. Теперь дополним определ типов из предыдущего листинга новыми аннотациями:

```
TMovement = record
  [TDim(3)] R: TVector;
  [TDim(3)] V: TVector;
  [TDim(3)] W: TVector;
  [TDim(4)] Color: TVector;
end;

TMovementScheme = class
private
  FMovement: array[1..100] of TMovement;
  [TDim(3, 3)] FOrientation: TMatrix;
end;
```

Ниже приведен код, осуществляющий инициализацию объектов типа TVector и TMatrix на основе информации, взятой из атрибутов

```
procedure Init(Obj, TypeInfoOfObj: Pointer; Offset: Integer = 0);
const
  DefaultRowCount = 3;
  DefaultColCount = 3;
  VectorTypeName = 'TVector';
  MatrixTypeName = 'TMatrix';
var
  RTTIContext: TRttiContext;
  Field : TRttiField;
  ArrFld: TRttiArrayType;
  I: Integer;
  Dim: TCustomAttribute;
  RowCount, ColCount: Integer;
  OffsetFromArray: Integer;
begin
  for Field in RTTIContext.GetType(TypeInfoOfObj).GetFields do
  begin
    if Field.FieldType <> nil then
    begin
      RowCount := DefaultRowCount;
      ColCount := DefaultColCount;
      for Dim in Field.GetAttributes do
      begin
        RowCount := (Dim as TDim).RowCount;
        ColCount := (Dim as TDim).ColCount;
      end;
      if Field.FieldType.TypeKind = tkArray then
      begin
        ArrFld := TRttiArrayType(Field.FieldType);
        if ArrFld.ElementType.TypeKind = tkRecord then
        begin
          for I := 0 to ArrFld.TotalElementCount - 1 do
          begin
            OffsetFromArray := I * ArrFld.ElementType.TypeSize;
            if ArrFld.ElementType.Name = VectorTypeName then
              PVector(Integer(Obj) +
                      Field.Offset +
                      OffsetFromArray +
                      Offset)^ := TVector.Create(RowCount)
            else if ArrFld.ElementType.Name = MatrixTypeName then
              PMatrix(Integer(Obj) +
                      Field.Offset +
                      OffsetFromArray +
                      Offset)^ := TMatrix.Create(RowCount, ColCount)
            else
              Init(Obj, ArrFld.ElementType.Handle, Field.Offset + OffsetFromArray);
          end;
        end;
```

```
      end
      else if Field.FieldType.TypeKind = tkRecord then
      begin
        if Field.FieldType.Name = VectorTypeName then
          PVector(Integer(Obj) +
                  Field.Offset +
                  Offset)^ := TVector.Create(RowCount)
        else if Field.FieldType.Name = MatrixTypeName then
          PMatrix(Integer(Obj) +
                  Field.Offset +
                  Offset)^ := TMatrix.Create(RowCount, ColCount)
        else
          Init(Obj, Field.FieldType.Handle, Field.Offset)
      end;
    end;
  end;
end;
```

Процедура Init() получает на вход адрес объекта-контейнера и его RTTI-данные. Далее происходит рекурсивный обход всех полей контейнера, и для всех встречных полей с именами типов «TVector» и «TMatrix» будут явно вызваны их конструкторы. Доработаем класс TMovementScheme с применением процедуры Init():

```
TMovementScheme = class
. . .
public
  constructor Create;
end;

constructor TMovementScheme.Create;
begin
  Init(Self, Self.ClassInfo);
end;
```

Вариант вызова Init() для произвольной записи:

```
var
  Movement: TMovement;
. . .
  Init(@Movement, TypeInfo(TMovement));
```

По умолчанию, Init() создает вектора с тремя элементами, а матрицы размером 3x3, поэтому в объявлении типов TMovement и TMovementScheme атрибуты [TDim(3)] и [TDim(3, 3)] можно опустить, оставив только [TDim(4)].

К статье прилагается файл, в котором реализация описываемых идей приведена в полном объёме.

Метки: матрицы, вектора, перегрузка операций, аннотации, copy on write

Комментарии 13

windgrace 25.10.17 в 11:21

Я Дельфи не трогал уже неизвестно сколько лет, поэтому вопрос, может быть, довольно наивный. Почему компоненты векторов и матриц хранятся в массиве? В Джаве, если бы мы перемножали такие матрицы и вектора, мы бы могли немало потерять на range checks во время доступа к компонентам. Многие библиотеки линейной алгебры обходятся просто именованными полями.

Akon32 25.10.17 в 12:04

(в delphi) range checks обычно отключаются после отладки (если нужна производительность).
Пока вижу проблему только в том, что SetLength() в конструкторе выделяет несколько блоков памяти (это может быть медленно).

1ntr0 25.10.17 в 18:27

Потери на контроле диапазонов вообще незаметны на фоне динамического выделения памяти процедурой SetLength(), которая действительно приводит к некоторому замедлению. Это плата за универсальность, за возможность работать с матрицами произвольных размеров. Альтернативой может быть размещение элементов в статическом массиве с наперед заданными границами, взятыми с больш запасом. Основной минус такого подхода — очень неудобно смотреть в отладчике на массивы 20x20, состоящие почти из одних нулей.

masai 26.10.17 в 12:10

Если нужна производительность, то, мне кажется, лучше хранить всё одним куском в памяти независимо от размеров и отдельно хранит размеры, сделать класс для каких-то базовых вещей вроде Create или извлечения элемента (это не так часто нужно, можно и с range che а сами операции выполнять с помощью какой-нибудь OpenBLAS.

Akon32 25.10.17 в 12:14

> К удивлению, обнаружил, что, несмотря на возможности современной среды разработки, коллеги-программисты зачастую используют процедурный подход в решении подобных задач.

Это потому, что старые версии delphi (для которых, вероятно, писали код ваши коллеги) не поддерживали конструкторов для record'ов, перегрузки операторов, аннотаций, вот этого всего. Создавать объект класса, а потом освобождать его (.Free()), было бы очень неудобно п использовании.

Вот это вот

```
TMatrix3x3 = array[1..3, 1..3] of Extended;
```

может размещаться на стеке, т.е. размещение выполняется очень быстро (просто изменение указателя стека), в отличие от вашего SetLeng m, n) (который ищет в куче (m+1) свободных блоков требуемого размера).
Если углубляться, доступ у "старой" версии тоже может быть чуть быстрее.

Но в целом, неплохо.

Arbane 25.10.17 в 17:30

Не думали использовать вместо Extended тип с фиксированной запятой (свой) или BCD?
И почему для записей был выбран именно вариант constructor, а не function или procedure? Ведь для записей здесь не будет разницы (всегд
есть не переопределяемый конструктор без параметров и выделение памяти осуществляется всегда). Да — всегда обновляй страницу —
заметил, что вопрос задан, но ответом не доволен.

1ntr0 25.10.17 в 18:35

Альтернативы типу Extended для себя не вижу. Для моих задач важно при отладке видеть элементы «как есть», в численном представлен
С вещественными числами работать удобно.
Конструктор использовал для наглядности — мы же «класс» проектируем :)

Akon32 26.10.17 в 09:46

Почему не double? Он вроде побыстрее должен работать (при меньшей точности, естественно).

1ntr0 26.10.17 в 22:12

Есть методы, сходимость которых обеспечивается высокой точностью вычислений. Поэтому Extended. При необходимости ускорить
вычисления ценой потери точности, можно вызвать функцию Math.SetPrecision().

Dmitri-D 25.10.17 в 18:29

векторы, матрицы и пр — это всё частные случаи контейнеров. Нельзя в 21м веке писать контейнеры не используя шаблоны.
В самом деле — вектор может сожержать цедые числа, числа с плавающией запятой, двойной точности, extended, или наконец комплексны
числа. И что на каждый тип элемента хранения писать свой новый класс и все сопутствующие алгоритмы??

1ntr0 25.10.17 в 18:30

Во-первых, шаблоны — штука полезная, но в Delphi их нет. Есть дженерики, а это совсем другая история. И в 21-ом веке пора бы их отлич
Дело в том, что над дженериками не определены арифметические операции, значит, вычислять на них нельзя.
Второй момент, матрицы — это не контейнеры, а математические объекты. Совсем разные понятия, лежащие в разных предметных обла
Контейнеры предназначены для хранения, для них определен стандартный интерфейс, куда должны входить методы «Добавить элемент
«Удалить элемент». А матрицы применяются для решения систем уравнений и при рассмотрении линейных преобразований. Матрица —
частный случай тензора, а не контейнера.

Coriolis 26.10.17 в 22:13

К сожалению никак не могу сесть на XE7, приходится пока работать на Delphi2007, поэтому с дженериками знаком поверхносно.
Разве нельзя соорудить базовый класс с а весь набор арифметических операций вынести в отдельные private классы. А этот базовый г
создании в зависимости от типа этого самого подключит нужный класс-реализацию? Для Extended свою, для целых свою и т.д.

1ntr0 26.10.17 в 22:20

При попытке использовать арифметические операции на параметризованном типе компилятор будет ругаться. Да и ни к чему это з
Серьёзно. Ценой производительности и удобства использования можно сделать всё, что угодно. Но всегда нужно исходить из
имеющихся требований к разработке. Требования приведены в начале статьи.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

+8 812 19 3

Новинка от Aftershokz — обзор новой гарнитуры с костной проводимостью Trekz Air GT

+11 1,1k 2 1