98406Сл(ок)
С-506

Учебно-методические пособия Учебно-научного центра ОИЯИ Дубна

УНЦ-2008-32

А. В. Смирнов, А. О. Сидорин, Г. В. Трубников

ОПИСАНИЕ РАЗМЕРНЫХ ПЕРЕМЕННЫХ С ПОМОЩЬЮ КЛАССОВ НА ЯЗЫКЕ C++

Учебно-методическое пособие

2008

Ц8406 Си++(05)
С-506

Учебно-научный центр ОИЯИ

В - 20259

А. В. Смирнов, А. О. Сидорин, Г. В. Трубников

ОПИСАНИЕ РАЗМЕРНЫХ ПЕРЕМЕННЫХ С ПОМОЩЬЮ КЛАССОВ НА ЯЗЫКЕ C++

Учебно-методическое пособие

НАУЧНАЯ БИБЛИОТЕКА ОИЯИ
Дубна 2008

С50

Учебное пособие составлено доцентами базовой кафедры МИРЭА «Электроника физических установок» при УНЦ ОИЯИ А. В. Смирновым, А. О. Сидориным, Г. В. Трубниковым (ОИЯИ) и рекомендовано к изданию экспертной комиссией УНЦ ОИЯИ и редакционно-издательским советом МИРЭА.

Смирнов А. В., Сидорин А. О., Трубников Г. В.

С50 Описание размерных переменных с помощью классов на языке C++: Учебно-метод. пособие. — Дубна: ОИЯИ, 2008. — 35 с.

Учебное пособие содержит описание иерархии классов для создания физических размерных переменных на языке C++. Рассмотрены основная идея создания размерных переменных, описания классов и примеры их использования. Приведен полный текст библиотеки размерных переменных.

Пособие предназначено для студентов и аспирантов инженерных и физических специальностей, изучающих объектно-ориентированное программирование на языке C++.

Smirnov A. V., Sidorin A. O., Trubnikov G. V.

Description of dimension variables with using of C++ classes: Textbook. — Dubna: JINR, 2008. — 35 p.

The textbook contains a description of the class hierarchy for physical dimension variables which was written in the C++ language. The idea of physical variables is considered, the description of classes and their examples are presented. The full program code of the physical dimension variables is included.

The book is addressed to the students and post-graduate students of physical and engineering specializations studying the object-oriented programming in C++ language.

© Объединенный институт ядерных исследований, 2008

Предисловие

Данное пособие написано как дополнительный методический материал по курсу лекций «Объектно-ориентированное программирование», прочитанному Смирновым А.В. в 2003 – 2007 гг. студентам базовой кафедры электроники физических установок Московского государственного института радиотехники, электроники и автоматики (МИРЭА), организованной в Дубне при Учебно-научном центре Объединенного института ядерных исследований, и предназначено для студентов очной формы обучения, специальность 200600 «Электроника физических установок».

Разработка переменных нового типа – размерных переменных – подразумевает решение нескольких соподчиненных задач. В первую очередь это создание средств для разработки систем единиц измерения и разработка некоторой конкретной системы единиц. На основе разработанной системы единиц измерения необходимо разработать стандарт представления размерных переменных, определить математические операции с ними, определить операции, выполняемые с размерными переменными и другими стандартными типами переменных выбранного языка программирования. Для удобства ввода формул для моделирования физических процессов очень полезно также определить в программе основные размерные константы, такие как, например, скорость света, заряд электрона и т.п.

Представленная иерархия классов для описания размерных переменных была специально создана для программы по моделированию динамики заряженных частиц в циклических накопителях [1] и является оригинальной разработкой авторов. Использование размерных переменных позволило заметно упростить процесс разработки программы для моделирования физических процессов, а также ускорить отладку и проверку программного кода.

Пособие предназначено для студентов и аспирантов инженерных и физических специальностей, изучающих объектно-ориентированное программирование на языке C++.

Авторы выражают благодарность студентке кафедры ЭФУ МИРЭА Михайловой Е.А. за помощь в подготовке данного пособия.

Введение

Любая физическая величина X может быть представлена в следующем виде:

$$X = v \cdot u,$$

где v – числовое представление данной физической величины в принятых единицах ее измерения, а u – принятая единица измерения. В качестве v может использоваться переменная одного из стандартных в языке программирования типов, тогда как для представления единиц измерения в большинстве языков программирования специальные средства отсутствуют. (Операции с размерными переменными реализованы в некоторых специализированных средах программирования, таких, например, как MathCad или Mathematica.)

Сложность реализации размерных переменных в языках программирования связана в основном с тем, что существует большое количество различных систем единиц измерения, и для измерения одной и той же физической величины могут использоваться разные единицы измерения. Поэтому необходимо разработать стандарт представления произвольной единицы измерения, позволяющий реализовать операцию сравнения различных единиц и создать универсальный набор средств, позволяющих построить произвольную систему единиц.

Различают базовые единицы измерения, которые определяются с помощью эталонов, и производные единицы, определяемые с помощью базовых. В любой конкретной системе единиц (например, СИ или СГС) выбор величины и количества базовых единиц измерения может быть произвольным и определяется только традициями или соглашениями. Однако для измерения физических величин минимально необходимым является определение всего трех независимых базовых единиц измерения, через которые могут быть выражены все остальные единицы. Системы единиц измерения, основанные на трех базовых единицах, называются абсолютными.

Впервые абсолютная система единиц была введена в 30-х гг. XIX в. К. Гауссом, причём в качестве основных он принял за единицу длины — миллиметр, массы — миллиграмм и времени — секунду. Поэтому часто название «абсолютные системы единиц» применяют в более узком смысле по отношению к системам, построенным на трёх основных единицах — длины, массы и времени, а иногда и в ещё более узком — по отношению к СГС-системам единиц, т. е. к системам, в которых за основные единицы приняты сантиметр, грамм и секунда.

При определении производной единицы какой-либо физической величины в абсолютной системе единиц исходят из формулы, выражающей зависимость между этой величиной и другими величинами, единицы которых являются основными или выражены через основные. Например, единица измерения скорости в абсолютной системе единиц равна отношению единицы измерения длины, деленной на единицу измерения времени, в соответствии с формулой

$$V = k \cdot L / T,$$

где V – скорость, L – пройденный путь, T – время, k – коэффициент перевода единиц измерения. При этом если длину измерять в метрах, а время в секундах, то при равном единице коэффициенте k единица измерения скорости будет «метр в секунду». В абсолютных системах единиц коэффициент пропорциональности обычно и полагают равным единице. Если по каким-либо причинам нам удобнее измерять скорость в сантиметрах в секунду, то мы должны положить $k = 0{,}01$. Этот пример иллюстрирует основной недостаток абсолютных систем единиц: при равном единице коэффициенте пересчета производные единицы измерения оказываются неудобными для инженерных расчетов, а при независимом определении производных единиц в формулах появляются нефизичные коэффициенты пересчета (такие как, например, диэлектрическая проницаемость вакуума в системе СИ).

При реализации компьютерных расчетов с размерными переменными этот недостаток может быть легко преодолен, если коэффициент пересчета сделать

атрибутом самой производной единицы измерения. В этом случае любую единицу измерения U физической величины можно представить как произведение трех базовых единиц, возведенных в некоторую степень, и числового множителя, связывающего величину данной единицы измерения с величинами базовых единиц:

$$U = k \cdot u_1^i \cdot u_2^j \cdot u_3^l,$$

где $u_{1,2,3}$ – базовые единицы измерения, i, j, l – целые или дробно-рациональные числа, k – числовой множитель. Таким образом, любая единица измерения может быть представлена упорядоченным набором четырех чисел (i, j, l, k). Две разные единицы измерения считаются эквивалентными друг другу (т.е. они могут применяться для измерения одной и той же физической величины), если у них совпадают показатели степеней всех трех базовых единиц.

Для создания иерархии классов, описываемой в данном пособии, были выбраны три базовые единицы из международной системы СИ: единица измерения времени – секунда, единица измерения электрического заряда – кулон, единица измерения длины – метр. Примеры описания базовых и производных единиц измерения физических величин приведены в таблице.

Физическая величина считается безразмерной (нулевой размерности), если показатели степеней всех трех базовых единиц равны нулю. Из таблицы видно, что такие единицы измерения, как метр и сантиметр, являются эквивалентными, так как они отличаются только числовым множителем. Точно так же это относится к кельвину и джоулю.

Набор из трех базовых единиц для абсолютной системы может быть выбран различными способами. Если абсолютная система построена на основе некоторого набора базовых единиц, как, например, система единиц, представленная в таблице, то в качестве базовых единиц новой абсолютной системы может быть выбрана любая тройка единиц измерения, у которой векторы (i, j, l) являются линейно независимыми. Так, из таблицы видно, что в качестве тройки базовых единиц измерения могут быть выбраны, например, ньютон,

джоуль и ватт. Построение такой абсолютной системы единиц можно рекомендовать в качестве самостоятельного упражнения.

Пример определения единиц измерения физических величин

Степень / Единица	u_1 SECOND i	u_2 QULOUN j	u_3 METER l	Коэффициент k
с	1	0	0	1
Кл	0	1	0	1
м	0	0	1	1
см	0	0	1	0.01
А	-1	1	0	1
Ом	1	0	-1	0.11*9e-9
кг	1	2	-3	0.11*9e-9
К	0	2	-1	1.53407*e-33
Тл	0	1	-2	0.037*e-5
Н	0	2	-2	0.11*e-9
Па	0	2	-4	0.11*e-9
Гц	-1	0	0	1
В	0	1	-1	0.11*e-9
Вт	-1	2	-1	0.11*e-9
Дж	0	2	-1	0.11*e-9

Правила операций с единицами измерения приведены в следующем разделе пособия и могут быть проиллюстрированы на примере умножения. При умножении одной единицы измерения на другую показатели степеней базовых единиц складываются, а коэффициенты пересчета умножаются.

Сама размерная переменная может быть представлена в виде упорядоченного набора из пяти чисел, одно из которых определяет ее числовое представление, а четыре остальных – единицу измерения. В этом случае при

объявлении размерной переменной в программе необходимо объявить ее единицу измерения. При выполнении операции присвоения значения размерной переменной или сравнения двух размерных переменных необходимо осуществлять проверку эквивалентности размерностей. Например, если в программе объявлена переменная U, измеряемая в вольтах, и две переменные $I = 5$ А и $R = 1$ Ом, то при выполнении операции присвоения

$$U = I \cdot R$$

необходимо осуществить следующие действия. Сначала нужно перемножить единицы измерения переменных I и R, затем убедиться, что получившаяся единица измерения эквивалентна вольту. Если это не так, то выдать сообщение об ошибке. Если единицы измерения эквивалентны, то вычислить числовое представление результата измерения. И, окончательно, вычислить числовое представление переменной U с учетом коэффициентов пересчета единиц измерения.

В данном пособии реализация алгоритма операций с размерными переменными осуществляется с помощью двух основных классов. В реальной программе для поддержки работы используется значительно большее количество классов, но для отражения основной идеи достаточно описания структуры основных классов.

class Units_ – класс единиц измерения. Этот класс создает новый тип данных – единицы измерения физических величин (например, см, Н, Вт и т.д.), осуществляет вывод сообщений об ошибках, проверяет эквивалентность единиц измерения, осуществляет арифметические действия с единицами измерения (умножение, деление, возведение в степень).

class doubleU – класс, который создает новый тип данных – размерные переменные (например, 5 А, 10 В, 3 Па), проверяет соответствие размерностей при арифметических действиях, выводит сообщения об ошибках, позволяет представлять физические величины в различных единицах (например, энергию в эВ или в Дж), инициализирует константы (например, скорость света, заряд

электрона и т.д.), выполняет арифметические и математические действия, операции сравнения, возведения в степень, вычисляет тригонометрические функции и т.п.

Использование размерных переменных на базе описанной выше абсолютной системы единиц дает ряд преимуществ при разработке программ по моделированию физических процессов. Это возможность совместного использования физических величин, записанных в различных системах единиц (СИ, СГС), так как все единицы измерения имеют внутреннее представление через три базовые единицы. При программировании физических формул нет необходимости использования коэффициентов перевода из одной системы единиц в другую, так как коэффициент пересчета является атрибутом самой единицы измерения. Физические переменные сами осуществляют проверку размерностей в формулах, что заметно упрощает процесс программирования и отладку программы.

Главным недостатком такого представления является то, что любая физическая величина или константа представлена пятью переменными типа **double**. Однако это лишь незначительно увеличивает размер и время выполнения программы. При этом размерные переменные предоставляют возможность выполнять арифметические операции совместно с обычными переменными типа **double** или **int**, что позволяет переходить к обычным переменным в тех местах программы, где скорость выполнения кода наиболее критична (например, многократно вложенные циклы).

Далее рассматривается структура классов и приводятся примеры использования переменных вновь созданных типов.

Построение системы единиц измерения

class Units_ описывает единицы измерения физических величин и операции с ними. Прежде чем создать данный класс, объявим перечисляемый тип BaseEnum, который используется при определении базовых единиц: секунда, кулон, метр. Константа UNIT=3 определяет количество базовых единиц измерения, что позволит обойтись минимальным изменением программного кода, если это количество будет увеличено в будущем (например, для использования размерных переменных в экономических расчетах).

```
enum BaseEnum{SECOND, QULOUN, METER, UNIT};

class Units_
{public:
  double sQm[UNIT];
  double k;
  Units_ ();
  Units_ (double);
  Units_ (BaseEnum) {;}
  void UnitsWarning();
  void operator()(Units_);
  bool operator==(Units_);
  bool operator!=(Units_);
  bool Zero();
  Units_  operator*(Units_);
  Units_  operator/(Units_);
  Units_  operator*(double);
  Units_  operator/(double);
  Units_  operator^(double);
 };
```

Данный класс имеет два члена: массив показателей степеней базовых единиц sQm[UNIT] и коэффициент k для перевода данной единицы к базовым. Далее определены три конструктора класса. Конструктор без параметров инициализирует единицу измерения безразмерных физических величин с коэффициентом, равным единице. Элементы массива sQm при этом равны нулю:


```
Units_::Units_()
{ k = 1.;
  for (int i = 0; i < UNIT; i++)
    sQm[i] = 0;
}
```

Конструктор с параметром типа **double** используется для инициализации единицы измерения безразмерной физической величины с определенным коэффициентом:

```
Units_::Units_(double koef)
{ k = koef;
  for (int i = 0; i < UNIT; i++)
    sQm[i] = 0;
}
```

Конструктор с параметром типа BaseEnum используется для инициализации трех базовых единиц измерения:

```
void Units_::Units_(BaseEnum baseenum)
{ k = 1.;
  for (int i = 0; i < UNIT; i++)
    sQm[i] = 0;
    if (baseenum != UNIT)
      sQm[baseenum] = 1;
}
```

После конструкторов идет описание функций-членов класса. Функция UnitsWarning используется для вывода сообщений об ошибках и выводит значения массива степеней базовых единиц измерения:

```
void Units_::UnitsWarning()
{ Warning(" s^",sQm[0],", Q^",sQm[1],", m^",sQm[2]); }
```

Здесь функция Warning есть некоторая функция для вывода сообщений на экран. В зависимости от операционной системы или используемого компилятора данная функция может иметь различные варианты реализации, поэтому мы не будем приводить ее программного кода.

Функция для создания производных единиц измерения **operator()** выполняет действия, аналогичные конструктору копирования, который автоматически создается компилятором:

```
void Units_::operator()(Units_ u)
{  k = u.k;
   for (int i = 0; i < UNIT; i++)
      sQm[i] = u.sQm[i];
}
```

Для сравнения единиц измерения используется функция **operator==**, которая возвращает **true**, если единицы эквивалентны, и возвращает **false**, если не эквивалентны:

```
bool Units_::operator==(Units_ u)
{  for (int i = 0; i < UNIT; i++)
      if ( fabs(sQm[i] - u.sQm[i]) > 1e-9)
         return false;
   return true;
}
```

Данная функция сравнения используется при выполнении арифметических операций над размерными переменными в классе doubleU. Для этого можно использовать еще одну функцию сравнения, которая возвращает обратное значение сравнения:

```
bool Units_::operator!=(Units_ u)
{  return !(*this == u);  }
```

Функция проверки нулевой размерности Zero() используется в алгебраических и тригонометрических функциях в классе doubleU, таких как sin, cos, ln, log, exp и т. д., которые не определены для размерных величин:

```
bool Units_::Zero()
{  for (int i = 0; i < UNIT; i++)
      if ( fabs(sQm[i]) > 1e-9)
         return false;
   return true;
}
```


В классе определены еще несколько арифметических функций, с помощью которых создаются производные единицы измерения.

Умножение одной единицы измерения на другую:

```
Units_  Units_::operator*(Units_ u)
{  Units_  tUnit;
   tUnit.k = k * u.k;
   for (int i = 0; i < UNIT; i++)
      tUnit.sQm[i] = sQm[i] + u.sQm[i];
   return tUnit;
}
```

Деление одной единицы измерения на другую:

```
Units_  Units_::operator/(Units_ u)
{
   Units_  tUnit;
   tUnit.k = k / u.k;
   for (int i = 0; i < UNIT; i++)
      tUnit.sQm[i] = sQm[i] - u.sQm[i];
   return tUnit;
}
```

Умножение единицы измерения на константу:

```
Units_  Units_::operator*(double koef)
{  Units_  tUnit;
   tUnit.k = k * koef;
   for (int i = 0; i < UNIT; i++)
      tUnit.sQm[i] = sQm[i];
   return tUnit;
}
```

Деление единицы измерения на константу:

```
Units_  Units_::operator/(double koef)
{  Units_  tUnit;
   tUnit.k = k / koef;
   for (int i = 0; i < UNIT; i++)
      tUnit.sQm[i] = sQm[i];
   return tUnit;
}
```

Возведение единицы измерения в степень:

```
Units_  Units_::operator^(double power)
```

```
{ Units_ tUnit ;
  tUnit.k = pow(k, power);
  for (int i = 0; i < UNIT; i++)
    tUnit.sQm[i] = sQm[i] * power;
  return tUnit;
}
```

Примеры определения единиц измерения приведены в приложении. Пользователь сам может легко добавить новые единицы измерения на основе уже ранее описанных.

Представление размерных переменных

Далее приступим к созданию нового типа данных – размерных переменных, с помощью которого можно описывать в программном коде реальные физические величины. Надо отметить, что данный тип построен не только на основе одного класса, но и с помощью функций, не являющихся членами этого класса. Приведем описание основных членов и функций класса, которые обеспечивают его работу. Полный текст программы приведен в приложении.

```
class doubleU
{
 public:
  double v;
  Units_ u;

  doubleU();
  doubleU(double);
  doubleU(Units_);
  doubleU(double, Units_);
  doubleU(double, double);

  void set(double);
  void set(double, Units_);
  void set(double, double);
  void set(Units_);
  void set(doubleU);

  bool CheckUnits(doubleU, char*);
  bool CheckZero (char*);
  double operator()();
  double operator()(Units_);
  void operator=(const doubleU&);
  void operator=(double d);

  doubleU  operator+(doubleU);
  doubleU  operator-(doubleU);
  doubleU  operator-();
  doubleU  operator*(doubleU);
  doubleU  operator/(doubleU);
  doubleU  operator&(doubleU);
```

```
  doubleU  operator*(double);
  doubleU  operator/(double);
  doubleU  operator^(double);

  bool operator==(doubleU);
  bool operator!=(doubleU);
  bool operator> (doubleU);
  bool operator>=(doubleU);
  bool operator< (doubleU);
  bool operator<=(doubleU);
};

doubleU operator+(double, doubleU);
doubleU operator-(double, doubleU);
doubleU operator*(double, doubleU);
doubleU operator/(double, doubleU);
```

class doubleU имеет всего лишь две переменные: v – числовое представление физической величины и u – единицу измерения этой физической величины. Далее рассмотрим, как осуществляются операции над этими переменными.

В конструкторе без параметров значение физической величины устанавливается равным единице, а ее размерность определена в конструкторе без параметров класса Units_.

```
doubleU::doubleU()
{ v = 1; }
```

Остальные конструкторы используются для задания начальных значений физической величины и ее размерности.

```
doubleU::doubleU(double v1)
{ v = v1; }

doubleU::doubleU(Units_ u1)
{ v = 1;
  u = u1;
}
doubleU::doubleU(double v1, Units_ u1)
{ v = v1;
  u = u1;
}
doubleU::doubleU(double v1, double u1)
```



```
{ v = v1;
  u.k = u1;
}
```

Если размерность физической величины или ее величина не были заданы при инициализации с помощью конструкторов с параметрами, то это можно сделать с помощью набора следующих функций set:

```
void doubleU::set(double d)
{ v = d / u.k; }

void doubleU::set(double d, Units_ u1)
{ v = d;
  u = u1;
}
void doubleU::set(double d, double du)
{
  v = d;
  u.k = du;
}
void doubleU::set(Units_ u1)
{ u = u1; }

void doubleU::set(doubleU zvars)
{ v = zvars.v;
  u = zvars.u;
}
```

Для проверки соответствия размерности физических величин используется специальная функция, которая выводит сообщение об ошибке в случае несоответствия размерностей и сообщает размерности физических переменных, над которыми выполняются арифметические операции:

```
bool doubleU::CheckUnits(doubleU du, char* sign)
{ if (u != du.u)
  { Warning("Can not execute operator", sign);
    u.UnitsWarning();
    du.u.UnitsWarning();
//************************************************************
    return false;   //******** Debuger point ***********
//************************************************************
  }
  return true;
}
```

13-2025-9

НАУЧНО-ТЕХНИЧЕСКАЯ БИБЛИОТЕКА ОИЯИ

Строка, отмеченная комментарием Debuger point, может быть использована для установки точки отладчика в случае, когда при выполнении программы появляется данная ошибка.

Для выполнения алгебраических и тригонометрических функций над размерными переменными необходимо сделать проверку, что они являются безразмерными:

```
bool doubleU::CheckZero(char* sign)
{
  if (!u.Zero())
  { Warning("Can not execute operator", sign);
    u.UnitsWarning();
    return false;
  }
  return true;
}
```

Следующая функция используется для получения физической величины без учета ее размерности:

```
double doubleU::operator()()
{ return v; }
```

Если же необходимо получить численное значение физической величины в некоторой определенной размерности, то тогда проверяется сначала возможность преобразования к такой размерности. Иначе выдается сообщение об ошибке и приводятся значения исходной и приводимой размерностей:

```
double doubleU::operator()(Units_ u1)
{
  if (u != u1)
  { Warning("Can not execute operator(Units)");
    u.UnitsWarning();
    u1.UnitsWarning();
    return false;
  }
  return v * u.k / u1.k;
}
```

При выполнении операции присваивания сначала проверяется соответствие физических размерностей. В случае их соответствия присваиваемая

величина масштабируется согласно размерности переменной, в которую она копируется. Выполнение проверки при копировании одной переменной в другую наиболее ярко показывает преимущество использования нового типа размерных переменных.

```
void doubleU::operator=(const doubleU& zvars)
{
  CheckUnits(zvars, " = ");
  v = zvars.v * zvars.u.k / u.k;
}
```

Для совместимости с предопределенными числовыми типами реализована простая функция присвоения переменной типа **double**. В этом случае проверки на соответствие размерностей не требуется.

```
void doubleU::operator=(double d)
{ v = d; }
```

Далее следует группа арифметических операций над размерными переменными. Надо отметить, что для операций сложения и вычитания также производится проверка соответствия размерностей. Для умножения и деления такой проверки не производится. В операции деления также проверяется ошибка деления на ноль.

```
doubleU doubleU::operator+(doubleU zvars)
{ CheckUnits(zvars, " + ");
  doubleU  temp ;
  temp.u = u;
  temp.v = v + zvars.v * zvars.u.k / temp.u.k;
  return temp;
}
doubleU doubleU::operator-(doubleU zvars)
{ CheckUnits(zvars, " - ");
  doubleU  temp ;
  temp.u = u;
  temp.v = v - zvars.v * zvars.u.k / temp.u.k;
  return temp;
}
doubleU doubleU::operator*(doubleU zvars)
{ doubleU  temp ;
  temp.u = u * zvars.u;
  temp.v = v * zvars.v;
```

```
  return temp;
}
doubleU doubleU::operator/(doubleU zvars)
{ doubleU  temp ;
  temp.u = u / zvars.u;
  if (zvars.v)
    temp.v = v / zvars.v;
  else
    Warning("Division by zero");
  return temp;
}
```

Полезно также определить унарный оператор, который меняет знак размерной величины:

```
doubleU doubleU::operator-()
{ doubleU temp;
  temp.v = -v;
  temp.u = u;
  return temp;
}
```

Для арифметических операций с предопределенным типом **double** также определены операции умножения и деления. Для операции деления выполняется проверка на ошибку деления на ноль:

```
doubleU doubleU::operator*(double d)
{ doubleU  temp ;
  temp.u = u;
  temp.v = v * d;
  return temp;
}
doubleU doubleU::operator/(double d)
{ doubleU  temp ;
  temp.u = u;
  if (d)
    temp.v = v / d;
  else
    Warning("Division by zero");
  return temp;
}
```

Дополнительно определена операция возведения в степень. Надо иметь в виду, что переопределение операций не меняет приоритет их выполнения, поэтому при таком определении операции возведения в степень она не будет

иметь более высокого приоритета по сравнению с другими арифметическими операциями:

```
doubleU doubleU::operator^(double d)
{ doubleU  temp ;
  temp.u = u ^ d;
  temp.v = pow(v, d);
  return temp;
}
```

Для размерных переменных полезно определить логические операции сравнения. В них также осуществляется проверка на соответствие размерностей:

```
bool doubleU::operator==(doubleU zvars)
{ CheckUnits(zvars, " == ");
  return (v*u.k) == (zvars.v*zvars.u.k);
}
bool doubleU::operator!=(doubleU zvars)
{ CheckUnits(zvars, " != ");
  return (v*u.k) != (zvars.v*zvars.u.k);
}
```

Как уже было отмечено, новый тип размерных переменных описан не только с помощью иерархии классов, но и с помощью глобальных функций, использующих в качестве входных параметров и результата операции размерные переменные. При этом, как правило, делается проверка на то, что арифметические операции осуществляются над безразмерными величинами. Полный список всех арифметических операций приведен в приложении.

```
doubleU operator+(double d, doubleU du)
{ du.CheckZero(" (double)+ ");
  doubleU temp;
  temp.u = du.u;
  temp.v = d / du.u.k + du.v;
  return temp;
}
doubleU operator*(double d, doubleU du)
{ doubleU temp;
  temp.u = du.u;
  temp.v = d * du.v;
  return temp;
}
```

Пример программы с размерными переменными

В качестве примера приведем реализацию класса, который можно использовать в реальной программе для описания энергии релятивистских частиц. Пользователь может задать любой из вариантов определения энергии частицы: полная энергия, кинетическая энергия, импульс частицы, магнитная жесткость, релятивистские факторы γ или β. Остальные будут автоматически пересчитаны в соответствии с массой и зарядом частицы.

Сначала определим тип U_EnergyEnum с вариантами определения энергии частицы. Далее идет описание класса, использующего размерные переменные. Логическая переменная PerNucleon используется, если определена не полная энергия частицы, а энергия на один нуклон. В конструкторе класса объявим размерности членов класса.

В первой функции Set идет расчет энергетических величин частицы в зависимости от того, как она была определена во второй функции. Вторая функция Set задает, каким образом была определена энергия частицы, и вызывает первую функцию для расчета всех вариантов определения энергии частицы.

```
enum U_EnergyEnum{U_GAMMA, U_BETA, U_VELOCITY, U_ENERGY, U_KINETIC,
      U_MOMENTUM, U_REGIDITY};

class U_Energy
{public:
   static bool PerNucleon;
   doubleU Gamma, Beta, Velocity, Energy;
   doubleU Kinetic, Momentum, Rigidity, Massa;
   doubleU Z, A;
   U_Energy();
   bool Set(U_EnergyEnum);
   bool Set(U_EnergyEnum, doubleU);
};

bool U_Energy::PerNucleon = true;

U_Energy::U_Energy()
{ Velocity.set(m_/s_);
   Energy.  set(eV_);
   Kinetic. set(eV_);
   Momentum.set(eV_c_);
```



```
   Rigidity.set(T_*m_);
   Massa.   set(amu_);
}

bool U_Energy::Set(U_EnergyEnum ue)
{
   bool errors = false;
   doubleU amu;
   if (!PerNucleon) amu = A;

   switch(ue)
   {case U_GAMMA:
      if (Gamma() <= 1)
      { Warning("GAMMA should be more than unit");
        errors = true;
      } break;
    case U_BETA:
      if ((Beta() >0) && (Beta() < 1))
      { Gamma = (1 - (Beta^2))^(-0.5);
      }else
      { Warning("BETA should be less than unit and positive");
        errors = true;
      } break;
    case U_VELOCITY:
      if (( Velocity() > 0 ) && ( Velocity < U_c ))
      { Beta = Velocity / U_c;
        Gamma = (1 - (Beta^2))^(-0.5);
      }else
      { Warning("VELOCITY should be less than light speed");
        errors = true;
      } break;
    case U_ENERGY:
      Gamma = Energy / (A * Massa * (U_c^2));
      if (Gamma() <= 1)
      { Warning("Total Energy should be more than rest mass");
        errors = true;
      } break;
    case U_KINETIC:
      if (Kinetic() > 0)
      { Gamma = Kinetic / (amu * Massa * (U_c^2)) + 1;
      }else
      { Warning("Kinetic Energy should be positive");
        errors = true;
      } break;
    case U_MOMENTUM:
      if (Momentum() > 0)
      { Energy = (((Momentum*U_c)^2)+((A * Massa * (U_c^2) )^2))^0.5;
        Gamma = Energy / (A * Massa * (U_c^2));
      }else
      { Warning("MOMENTUM should be positive");
        errors = true;
      } break;
    case U_RIGIDITY:
      if (Rigidity() > 0)
      { Momentum = Rigidity * U_e / U_c;
```

```
      Energy = (((Momentum*U_c)^2)+((A*Massa * (U_c^2) )^2))^0.5;
      Gamma = Energy / (A * Massa * (U_c^2));
    }else
    { Warning("REGIDITY should be positive");
      errors = true;
    } break;
   default: Warning("Unknown zEnergyEnum parameter : ", ue);
  }
  if (!errors)
  { if (ue != U_BETA)    Beta = (1 - (Gamma^-2)) ^ 0.5;
    if (ue != U_VELOCITY) Velocity = Beta * U_c;
    if (ue != U_ENERGY)  Energy = Gamma * A * Massa * (U_c^2);
    if (ue != U_KINETIC) Kinetic = (Gamma-1)*amu*Massa*(U_c^2);
    if (ue != U_MOMENTUM)
       Momentum = (((Energy^2) - (A*Massa*(U_c^2)^2))^0.5) / U_c;
    if (ue != U_REGIDITY) Rigidity = Momentum * U_c / U_e;
  }
  return errors;
}

bool U_Energy::Set(U_EnergyEnum ue, doubleU du)
{
  switch(ue)
  {case U_GAMMA:   Gamma    = du; break;
   case U_BETA:    Beta     = du; break;
   case U_VELOCITY: Velocity = du; break;
   case U_ENERGY:  Energy   = du; break;
   case U_KINETIC: Kinetic  = du; break;
   case U_MOMENTUM: Momentum = du; break;
   case U_RIGIDITY: Rigidity = du; break;
   default: Warning("Unknown zEnergyEnum parameter : ", ue);
  }
  return Set(ue);
}
```

Литература

[1] Мешков И.Н., Пивин Р.В., Сидорин А.О., Смирнов А.В., Трубников Г.В. Численное моделирование динамики частиц в накопителях с использованием программы BETACOOL //Письма в ЭЧАЯ, т.3, №7, стр. 82-86 (2006) http://lepta.jinr.ru/betacool

Приложение. Новый тип размерных переменных

файл "doubleU.h"

```
//---------------------------------------------------------------------------
extern Units_ s_, Q_, m_, U1_;
extern Units_ cm_, barn_, H_, Hz_, min_, hour_, day_, year_;
extern Units_ e_, q_, F_, V_, A_, Ohm_, uP_, G_, T_;
extern Units_ eV_, K_, eV_c_, eV_c2_, J_, W_, kg_, g_;
extern Units_ amu_, neutron_, proton_, electron_;
extern Units_ N_, Pa_, atm_, Torr_;

extern doubleU U_0,U_1,U_hbar,U_amu,U_mn,U_mp,U_me,U_k,U_c,U_e;
extern doubleU U_Grav,U_grav,U_NA,U_pi,U_exp,U_eps0,U_mu0;
extern doubleU U_fine,U_re,U_rp,U_r1,U_R,U_Ecoup,U_lambdaC,U_muBorn;

//---------------------------------------------------------------------------
enum  BaseEnum{SECOND, QULOUN, METER, UNIT};

class Units_
{
public:
#ifdef PoWeRs
  void UnitsWarning();
  double sQm[UNIT];
#endif
  double k;
  Units_ ();
  Units_ (double);
  Units_ (BaseEnum) {;}
  void operator()(BaseEnum);
  void operator()(Units_);
  bool operator==(Units_);
  bool operator!=(Units_);
  bool Zero();
  Units_ operator*(Units_);
  Units_ operator/(Units_);
  Units_ operator*(double);
  Units_ operator/(double);
  Units_ operator^(double);
};

//---------------------------------------------------------------------------
class doubleU
{
 public:
  double v;
  Units_ u;
  doubleU();
  doubleU(double);
  doubleU(Units_);
```

```
  doubleU(double, Units_);
  doubleU(double, double);
  doubleU(BaseEnum);

  bool CheckUnits(doubleU, char*);
  bool CheckZero (char*);
  double operator()() { return v; }
  double operator()(Units_);

  void set(double);
  void set(double, Units_);
  void set(double, double);
  void set(Units_);
  void set(doubleU);

  void operator=(const doubleU&);
  void operator=(double d);

  void operator+=(doubleU);
  void operator-=(doubleU);
  void operator*=(doubleU);
  void operator/=(doubleU);
  void operator%=(doubleU);

  doubleU  operator+(doubleU);
  doubleU  operator-(doubleU);
  doubleU  operator-();
  doubleU  operator*(doubleU);
  doubleU  operator/(doubleU);

  doubleU  operator+(double);
  doubleU  operator-(double);
  doubleU  operator*(double);
  doubleU  operator/(double);
  doubleU  operator^(double);

  bool operator==(doubleU);
  bool operator!=(doubleU);
  bool operator> (doubleU);
  bool operator>=(doubleU);
  bool operator< (doubleU);
  bool operator<=(doubleU);
};

//---------------------------------------------------------

doubleU operator+(double, doubleU);
doubleU operator-(double, doubleU);
doubleU operator*(double, doubleU);
doubleU operator/(double, doubleU);

double U_Ln (doubleU);
double U_Log(doubleU);
double U_Exp(doubleU);
double U_Pow(doubleU, double);
```



```
double U_Sin(doubleU);
double U_Cos(doubleU);
double U_Tan(doubleU);
double U_Asin(doubleU);
double U_Acos(doubleU);
double U_Atan(doubleU);
double U_Atan2(doubleU, doubleU);
double ArcTan2(double, double);
double SqrtSum(double a, double b);
doubleU U_SqrtSum(doubleU, doubleU);
doubleU U_Abs(doubleU);
```

файл "doubleU.cpp"

```
//---------------------------------------------------------
double light = 2.99792458e10;

// Base units
Units_ s_ (SECOND);
Units_ Q_ (QULOUN);
Units_ m_ (METER);
Units_ U1_(UNIT);

// Some derivative units
Units_ cm_  (m_/100);                  // centimetre
Units_ barn_((cm_^2)*1e-24);           // barn
Units_ H_   (cm_*1e9);                 // Henry
Units_ Hz_  (s_^-1);                   // Herz
Units_ min_ (s_*60);                   // minute
Units_ hour_(min_*60);                 // hour
Units_ day_ (hour_*24);                // day
Units_ year_(day_*365);                // year

// Electromagnetic, perveance
Units_ e_  (Q_*1.6021893e-19);         // electron charge
Units_ q_  (Q_/3e9);                   // SGSI charge
Units_ F_  (cm_*9e11);                 // Farade
Units_ V_  (Q_/F_);                    // Volt
Units_ A_  (Q_/s_);                    // Ampere
Units_ Ohm_(V_/A_);                    // Ohm
Units_ uP_ (u_6*A_/(V_^1.5));          // Perveance
Units_ G_  (V_/cm_*300);               // Gauss
Units_ T_  (G_*1e4);                   // Tesla

// Power, temperature, momentum, massa
Units_ eV_   (e_*V_);                  // electron Volts
Units_ K_    (eV_*8.61738573e-5);      // Kelvin
Units_ eV_c2_(eV_*((s_/(cm_*light))^2)); // mass
Units_ eV_c_ (eV_*  s_/(cm_*light));   // momentum
Units_ J_    (Q_*V_);                  // Joule
Units_ W_    (J_/s_);                  // Watt
Units_ kg_   (J_*(s_/m_^2));           // kilogram
Units_ g_    (kg_/1000);               // gram
```

```
// Massa of particles
Units_ amu_     (M_6*eV_c2_ * 931.5016);          // atom mass unit
Units_ neutron_  (M_6*eV_c2_ * 939.5731);         // neutron mass
Units_ proton_  (M_6*eV_c2_ * 938.2796);          // proton mass
Units_ electron_ (M_6*eV_c2_ * 0.5110034);        // electron mass

// Force, pressure
Units_ N_  (J_/m_);                               // Newton
Units_ Pa_  (N_/(m_^2));                          // Pascal
Units_ atm_ (Pa_*1.013e5);                        // atmosphere
Units_ Torr_(atm_/760);                           // Torr

//Base constants
doubleU U_0._     ( 0,              U1_ );                    // zero
doubleU U_1._     ( 1,              U1_ );                    // unit
doubleU U_hbar._  ( 0.658212202e-15,  eV_*s_ );             // Plank constant
doubleU U_amu._    ( 1,               amu_ );               // atom mass unit
doubleU U_mn._    ( 1,              neutron_ );             // neutron mass
doubleU U_mp._    ( 1,              proton_ );              // proton mass
doubleU U_me._    ( 1,              electron_ );            // electron mass
doubleU U_k._     ( 8.61738573e-5,    eV_/K_ );             // Boltzman
doubleU U_c._     ( light/100.,       m_/s_ );              // light speed
doubleU U_e._     ( 1,              e_ );                   // electron charge
doubleU U_Grav._ (6.6725985e-8,(cm_^3)/(g_*(s_^2)));      // gravitational
doubleU U_grav._  ( 9.80665,       m_/(s_^2) );           // acceleration
doubleU U_NA._    ( 6.022136736e23,  U1_ );               // Avagadro
doubleU U_pi._    ( M_PI,           U1_ );                // pi
doubleU U_exp._   ( 2.718281828459045235,U1_ );           // exponent
doubleU U_eps0._  ( 1./(4.*U_pi()*9e+9), F_/m_ );         // permittinity
doubleU U_mu0._   (    4.*U_pi()*1e-7,  H_/m_ );          // permeability

// Derivative constants
doubleU U_fine._   ( (U_e^2) / (U_hbar * U_c) );          // fine structure
doubleU U_re._     ( (U_e^2) / (U_me*(U_c^2)) );          // electron radius
doubleU U_rp._     ( (U_e^2) / (U_mp*(U_c^2)) );          // proton radius
doubleU U_r1._     ( (U_hbar^2)/(U_me*(U_e^2)) );         // 1st Borh radius
doubleU U_R._ (U_me * (U_e^4) / ( (U_hbar^3) * 2));       // Ridberg
doubleU U_Ecoup._(U_me*(U_e^4) / ((U_hbar^2) * 2));       // coupling energy
doubleU U_lambdaC._(U_pi * U_hbar * 2 / (U_me*U_c));      // Compthon length
doubleU U_muBorn._(U_e * U_hbar / (U_me * U_c * 2));      // Borh magnethon

//-------------------------------------------------------

Units_::Units_()
{  k = 1.;
   for (int i = 0; i < UNIT; i++)
     sQm[i] = 0;
}
Units_::Units_(double koef)
{  k = koef;
   for (int i = 0; i < UNIT; i++)
     sQm[i] = 0;
}
void Units_::operator()(BaseEnum baseenum)
{  k = 1.;
```



```
   for (int i = 0; i < UNIT; i++)
     sQm[i] = 0;
   if (baseenum != UNIT)
     sQm[baseenum] = 1;
}
void Units_::operator()(Units_ u)
{  k = u.k;
   for (int i = 0; i < UNIT; i++)
     sQm[i] = u.sQm[i];
}
void Units_::UnitsWarning()
{ Warning("  s^",sQm[0],", Q^",sQm[1],", m^",sQm[2]); }

bool Units_::operator==(Units_ u)
{  for (int i = 0; i < UNIT; i++)
     if ( fabs(sQm[i] - u.sQm[i]) > accuracy )
       return false;
   Warning("! --- Comparison of dimensions is not correct --- !");
   return true;
}
bool Units_::operator!=(Units_ u)
{  return !(*this == u);  }

bool Units_::Zero()
{  for (int i = 0; i < UNIT; i++)
     if ( fabs(sQm[i]) > accuracy)
       return false;
   return true;
}
Units_  Units_::operator*(Units_ u)
{  Units_  tUnit;
   tUnit.k = k * u.k;
   for (int i = 0; i < UNIT; i++)
     tUnit.sQm[i] = sQm[i] + u.sQm[i];
   return tUnit;
}
Units_  Units_::operator/(Units_ u)
{  Units_  tUnit;
   tUnit.k = k / u.k;
   for (int i = 0; i < UNIT; i++)
     tUnit.sQm[i] = sQm[i] - u.sQm[i];
   return tUnit;
}
Units_  Units_::operator*(double koef)
{  Units_  tUnit;
   tUnit.k = k * koef;
   for (int i = 0; i < UNIT; i++)
     tUnit.sQm[i] = sQm[i];
   return tUnit;
}
Units_  Units_::operator/(double koef)
{  Units_  tUnit;
   tUnit.k = k / koef;
   for (int i = 0; i < UNIT; i++)
     tUnit.sQm[i] = sQm[i];
```

```
  return tUnit;
}

Units_  Units_::operator^(double power)
{  Units_  tUnit ;
   tUnit.k = pow(k, power);
   for (int i = 0; i < UNIT; i++)
     tUnit.sQm[i] = sQm[i] * power;
   return tUnit;
}
Units_  Units_::operator*(MicroMega mega)
{   Units_ tUnit ;
   tUnit.k = k * mega.MegaValue;
   for (int i = 0; i < UNIT; i++)
     tUnit.sQm[i] = sQm[i];
   return tUnit;
}
//---------------------------------------------------------------
doubleU::doubleU()
{  v = 1;  }

doubleU::doubleU(double v1)
{  v = v1;  }

doubleU::doubleU(Units_ u1)
{  v = 1;
   u = u1;
}
doubleU::doubleU(double v1, Units_ u1)
{  v = v1;
   u = u1;
}
doubleU::doubleU(double v1, double u1)
{  v = v1;
   u.k = u1;
}
doubleU::doubleU(BaseEnum unit) : u(unit) { ; }

bool doubleU::CheckUnits(doubleU du, char* sign)
{  if (u != du.u)
   {  Warning("Can not execute operator", sign, );
      u.UnitsWarning();
      du.u.UnitsWarning();
//*************************************************************
      return false;    // ******** Debuger point ************
//*************************************************************
   }
   return true;
}
bool doubleU::CheckZero(char* sign)
{
   if (!u.Zero())
   {  Warning("Can not execute operator", sign);
      u.UnitsWarning();
      return false;
```



```
   }
   return true;
}
double doubleU::operator()(Units_ u1)
{  if (u != u1)
   {  Warning("Can not execute operator");
      u.UnitsWarning();
      u1.UnitsWarning();
      return false;
   }
   return v * u.k / u1.k;
}

void doubleU::set(double d)
{  v = d / u.k;  }

void doubleU::set(double d, Units_ u1)
{  v = d;
   u = u1;
}
void doubleU::set(double d, double du)
{  v = d;
   u.k = du;
}
void doubleU::set(Units_ u1)
{  u = u1;  }

void doubleU::set(doubleU zvars)
{  v = zvars.v;
   u = zvars.u;
}
void doubleU::operator=(const doubleU& zvars)
{  CheckUnits(zvars, "=");
   v = zvars.v * zvars.u.k / u.k;
}
void doubleU::operator=(double d)
{  v = d;  }

void doubleU::operator+=(doubleU zvars)
{  CheckUnits(zvars, "+=");
   v += zvars.v * zvars.u.k / u.k;
}
void doubleU::operator-=(doubleU zvars)
{  CheckUnits(zvars, "-=");
   v -= zvars.v * zvars.u.k / u.k;
}
void doubleU::operator*=(doubleU zvars)
{  u = u * zvars.u;
   v *= zvars.v;
}
void doubleU::operator/=(doubleU zvars)
{  u = u / zvars.u;
   if (zvars.v)
      v /= zvars.v;
   else
```

```
    Warning("Division by zero");
}
doubleU doubleU::operator+(doubleU zvars)
{  CheckUnits(zvars, "+");
   doubleU  temp ;
   temp.u = u;
   temp.v = v + zvars.v * zvars.u.k / temp.u.k;
   return temp;
}
doubleU doubleU::operator-(doubleU zvars)
{  CheckUnits(zvars, "-");
   doubleU  temp ;
   temp.u = u;
   temp.v = v - zvars.v * zvars.u.k / temp.u.k;
   return temp;
}
doubleU doubleU::operator-()
{  doubleU temp;
   temp.v = -v;
   temp.u = u;
   return temp;
}
doubleU doubleU::operator*(doubleU zvars)
{  doubleU  temp ;
   temp.u = u * zvars.u;
   temp.v = v * zvars.v;
   return temp;
}
doubleU doubleU::operator/(doubleU zvars)
{  doubleU  temp ;
   temp.u = u / zvars.u;
   if (zvars.v)
     temp.v = v / zvars.v;
   else
     Warning("Division by zero");
   return temp;
}

doubleU doubleU::operator%(doubleU zvars)
{
        doubleU temp(*this);
   temp %= zvars;
   return temp;
}
doubleU doubleU::operator+(double d)
{  CheckZero("+(double)");
   doubleU  temp ;
   temp.u = u;
   temp.v = v + d / u.k;
   return temp;
}
doubleU doubleU::operator-(double d)
{  CheckZero("-(double)");
   doubleU  temp ;
   temp.u = u;
```



```
   temp.v = v - d / u.k;
   return temp;
}
doubleU doubleU::operator*(double d)
{  doubleU  temp ;
   temp.u = u;
   temp.v = v * d;
   return temp;
}
doubleU doubleU::operator/(double d)
{  doubleU  temp ;
   temp.u = u;
   if (d)
     temp.v = v / d;
   else
     Warning("Division by zero");
   return temp;
}
doubleU doubleU::operator^(double d)
{  doubleU  temp ;
   temp.u = u ^ d;
   temp.v = pow(v, d);
   return temp;
}
bool doubleU::operator==(doubleU zvars)
{  CheckUnits(zvars, "==");
   return (v*u.k) == (zvars.v*zvars.u.k);
}

bool doubleU::operator!=(doubleU zvars)
{  CheckUnits(zvars, "!=");
   return (v*u.k) != (zvars.v*zvars.u.k);
}
bool doubleU::operator>(doubleU zvars)
{  CheckUnits(zvars, ">");
   return (v*u.k) > (zvars.v*zvars.u.k);
}
bool doubleU::operator>=(doubleU zvars)
{  CheckUnits(zvars, ">=");
   return (v*u.k) >= (zvars.v*zvars.u.k);
}

bool doubleU::operator<(doubleU zvars)
{  CheckUnits(zvars, "<");
   return (v*u.k) < (zvars.v*zvars.u.k);
}

bool doubleU::operator<=(doubleU zvars)
{  CheckUnits(zvars, "<=");
   return (v*u.k) <= (zvars.v*zvars.u.k);
}
//-----------------------------------------------------------
doubleU operator+(double d, doubleU du)
{  du.CheckZero("(double)+");
   doubleU temp;
```

```
    temp.u = du.u;
    temp.v = d / du.u.k + du.v;
    return temp;
  }
  doubleU operator-(double d, doubleU du)
  {  du.CheckZero("(double)-");
     doubleU temp;
     temp.u = du.u;
     temp.v = d / du.u.k - du.v;
     return temp;
  }
  doubleU operator*(double d, doubleU du)
  {  doubleU temp;
     temp.u = du.u;
     temp.v = d * du.v;
     return temp;
  }
  doubleU operator/(double d, doubleU du)
  {  doubleU temp;
     temp.u = U1_ / du.u;
     temp.v = d / du.v;
     return temp;
  }
  double U_Ln(doubleU zvars)
  {  zvars.CheckZero(" U_Ln(doubleU )");
     return log(zvars(U1_));
  }
  double U_Log(doubleU zvars)
  {  zvars.CheckZero(" U_Log(doubleU )");
     return log10(zvars(U1_));
  }
  double U_Exp(doubleU zvars)
  {  zvars.CheckZero(" U_Exp(doubleU )");
     return exp(zvars(U1_));
  }
  double U_Pow(doubleU zvars, double power)
  {  zvars.CheckZero(" U_Pow(doubleU )");
     return pow(zvars(U1_), power);
  }
  double U_Sin(doubleU zvars)
  {  zvars.CheckZero(" U_Sin(doubleU )");
     return sin(zvars(U1_));
  }
  double U_Cos(doubleU zvars)
  {  zvars.CheckZero(" U_Cos(doubleU )");
     return cos(zvars(U1_));
  }
  double U_Tan(doubleU zvars)
  {  zvars.CheckZero(" U_Tan(doubleU )");
     return tan(zvars(U1_));
  }
  double U_Asin(doubleU zvars)
  {  zvars.CheckZero(" U_Asin(doubleU )");
     return asin(zvars(U1_));
  }
```



```
  double U_Acos(doubleU zvars)
  {  zvars.CheckZero(" U_Acos(doubleU )");
     return acos(zvars(U1_));
  }

  double U_Atan(doubleU zvars)
  {  zvars.CheckZero(" U_Atan(doubleU )");
     return atan(zvars(U1_));
  }
  double U_Atan2(doubleU zvars1, doubleU zvars2)
  {  zvars1.CheckUnits(zvars2, " U_Atan2(doubleU, doubleU)");
     return ArcTan2(zvars1(U1_), zvars2(U1_));
  }
  double ArcTan2(double y, double x)
  {  double angle;
     if ( x )
     {  angle =  atan(y/x);
        if (x < 0)
        {  if (y < 0) angle -=  U_pi();
           else       angle +=  U_pi();
        }
     }else
     {  if ( y )
        {  if (y < 0) angle = -U_pi()/2.;
           else       angle =  U_pi()/2.;
        }else
           angle = 0;
     }
     return angle;
  }
  double SqrtSum(double a, double b)
  {  return sqrt(a*a + b*b); }

  doubleU U_SqrtSum(doubleU a, doubleU b)
  {
     a.CheckUnits(b, " U_SqrtSum(doubleU, doubleU)");
     return (((a*a)+(b*b))^0.5);
  }
  doubleU  U_Abs(doubleU zvars)
  {  doubleU  temp;
     temp.v = fabs(zvars.v);
     temp.u = zvars.u;
     return temp;
  }
```