The Debugger Annex

---------------------------------------------------------------------

The interactive development environments (IDE's), complete with source-level interactive debuggers have surely made the development of computer programs quicker and easier!

But there is one kind of error that these interactive debuggers will NOT catch: erroneous operations on numbers that represent quantities and/or places, e.g. 245 calories, 30 mph, 21 inches, or 72 degrees Fahrenheit.

Given the existing compilers and interactive debuggers, there doesn't seem to be any way to tell the system what the numbers represent; and that's the problem, but with this package there is. And hence the name Debugger Annex.

Using the object-oriented capability of C++ one can code statements like the following:

munit\_quantity my\_weight = 197.3 \* lbs;

munit\_quantity airplane\_speed = 675.0 \* mph;

munit\_quantity Concorde\_speed = mach \* 2.03;

munit\_quantity package\_net\_weight = lb / 4.0; // i.e. a quarter of a pound

munit\_quantity package\_nt\_weight = lb \* 0.25; // i.e. a quarter of a pound – same thing

munit\_quantity vanilla\_1\_amount = 5.2 \* grams; // weight measure

munit\_quantity vanilla\_2\_amount = 1.5 \* tsp; // volume measure

munit\_quantity waiting\_time = 20 \* time\_minutes;

munit\_quantity Xu = vanilla\_1\_amount + vanilla\_2\_amount; // This is an error.

It is an error, because it attempts to add unlike quantities. Should this package have a table of volume/weight densities of every known powder and liquid? That is a rhetorical question, of course.

munit\_quantity distance\_driven = 50.47 \* miles + 40.52 \* km;

The two quantities added here are not unlike – different units but the same unit type. When a conversion factor is put in, this addition result will be 121.743592 km, 75.64796071 miles, 4,793,054.79 inches, or however you choose to format it. See below, on how the munit\_readout function determines form and format.

In short, the commutative and associative rules apply to the units as well as to numbers. But only like quantities can be added and subtracted. Do you remember working the problems in your physics courses? In the Implementation section of this document you’ll find more explanation of this modern algebra.

Angle is an important unit of measure. And this package makes a special provision for trigonometric and inverse trigonometric functions.

Examples:

angle\_45 = 45 \* trig\_degrees; angle\_33g = (100.0 / 3.0) \* gradients;

double aorth\_45\_d\_sin = sin(angle\_45); // value: 0.5 \* sqrt(2)

double aorth\_45\_d\_tan = tan(angle\_45); // value: 1

double aorth\_33\_g\_sin = sin(angle\_33g); // value: 0.5

double aorth\_33\_g\_cos = cos(angle\_33g); // value: sqrt(3) / 2

double aorth\_33\_g\_tan = tan(angle\_33g); // value: sqrt(3) / 3

munit\_quantity xxiz = Atan(sqrt(3.0) / 3.0); // value: 60 degrees

munit\_quantity xxia = Asin(sqrt(2.0) / 2.0); // value: 45 degrees

Notice the capital “A” in the names of the inverse trigonometric functions.

The name trig\_degrees is used in this package to distinguish between temperature degrees and trigonometric degrees. And the names trig\_minutes and trig\_seconds are used because it’s important to distinguish time minutes and seconds from trigonometric minutes and seconds.

There are multiplicative combinations of quantities. For example:

Velocity = distance / time.

Acceleration = velocity / time. (Can your car go from zero to sixty in three seconds?)

force = mass \* acceleration,

energy = force \* distance,

voltage = current \* resistance, and power = current \* voltage.

With capacitance and inductance, the multiplicative combinations get complicated.

Aside from physics and engineering, a job takes a certain number of man-hours;

and on hand \* days is an important quantity in inventory control applications.

Places are something different from quantities.

A pointer in C/C++ programming denotes a place in a computer's memory.

A date, with or without an accompanying time of day, is a place in time.

A temperature, in Fahrenheit or Celcius is a place.

A pair of coordinates is a pair of two places: longitude and latitude,

both being angles.

There are only three “regular” arithmetic operations involving places:

1. A place + a quantity = a place. Example: 4/12/2018 + 3 days = 4/15/2018

2. A place - a quantity = a place. Example: 4/15/2018 – 3 days = 4/12/2018

3. A place - a place = a quantity. Example: 4/15/2018 – 4/12/2018 = 3 days

and two ‘tentative place’ operations for the sake of “algebraic freedom”.

4. A place + a place = a tentative place.

5. An integer \* a place (repeated addition)

The units have to be in agreement, as in the addition and subtraction of quantities.

tentative place example: 6/20/1018 + 4/15/2018 – 4/12/2018 = 6/23/2018,

This is equivalent to 6/20/2018 + (4/15/2018 – 4/12/2018), which is a place + a quantity.

Same thing for 2 \* 4/15/2018 – 4/12/2018

or 4/15/2018 \* 2 - 4/12/2018

This is equivalent to 4/15/2018 + 4/15/2018`- 4/12/2018. repeated addition

or 4/15/2018 + (4/15/2018 – 4/12/2018), which is a place + a quantity.

Tentative, here means not really valid but could be valid in an expression.

2 \* 4/15/2018 or 6/20/2018 + 4/15/2018 would be tentatively valid.

The tentative place consists of a ‘multiplicity’ and a quantity.

Let P1 = <M1 , Q1> and P2 = <M2 , Q2>. Then P1+P2 = <M1 + M2 , Q1 + Q2>

And P1-P2 = <M1 – M2 , Q1 - Q2>

simple enough, right? Any tentative place with a multiplitity of 1 is a valid place.

Any “ “ “ “ “ “ 0 is a valid quantity.

I just happen to believe that there is something desirable about preserving the commutative and associative rules, even if it means that X is not a valid expression (but tentative), but X minus P is valid.

--- – - --- – - --- – - --- – - --- – - --- – - --- – - --- – - --- – - --- – - --- – -

Coding examples:

munit\_place comfortable\_room\_temperature = Fahrenheit(72);

munit\_place healthy\_body\_temperature = Centigrade(37);

munit\_place dangerous\_body\_temperature = healthy\_body\_temperature + 6.4 \* Rankine\_deg;

/\* This would be 105°F – the kind of fever that can result in serious brain damage. Whatever the cause, the patient must be packed in ice to bring the fever down. \*/

yrmo\_date D\_Day = make\_yrmo\_date\_from\_string("6/6/1944"); // string to date

munit\_place \_D\_Day\_m = make\_munit\_date\_from\_string("6/6/1944");

yrmo\_date important\_anniversary = D\_Day + 50 \* years;

/\* around this 50th anniversary of D Day, several TV channels were continuously

on about memories of World War II. I kind of enjoyed the interviews with people like the Andrews Sisters and Vera Lynn. \*/

munit\_place \_D\_Day = make\_munit\_date\_from\_yrmo(D\_Day);

munit\_place an\_important\_anniversary = make\_munit\_date\_from\_yrmo(important\_anniversary);

Some of the above date arithmetic operations and conversion to/from a displayable string involve conversion from munit\_place to yrmo\_date and/or vice versa. There is no getting away from that. But this package makes it transparent to the user.

In the spreadsheet packages I have used, there are good date arithmetic facilities. But I haven’t yet seen any date arithmetic facilities that can be included in procedure-oriented programming language code, except in this package.

– - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – -

To read out a munit\_quantity, (i.e. convert it to a displayable string) use the munit\_readout function, like this:

string reading\_out\_m = munit\_readout(distance\_driven, mile, 9);

string value is "75.6479607 miles"

string reading\_out\_k = munit\_readout(distance\_driven, km, 9);

string value is "121.743592 km" without the quotes.

The third parameter, which was nine in the above two examples, defaults to six.

munit\_quantity Sunday\_distance\_driven = 50.0 \* miles + 40.0 \* km;

string Sunday\_reading\_out\_k = munit\_readout(Sunday\_distance\_driven, km);

string value is "120.467 km"

If fewer than the designated number of significant figures is sufficient for the exact value, the trailing zeroes are eliminated.

string Sunday\_reading\_out\_k9 = munit\_readout(Sunday\_distance\_driven, km, 9);

string value is "120.4672 km" – not “120.467200 km”

Places can be read out in the same way.

string temp\_hly = Fahrenheit\_readout(healthy\_body\_temperature);

// string value: “98.6 deg\_F”

string temp\_dng = Fahrenheit\_readout(dangerous\_body\_temperature);

// string value: “105 deg\_F”

string temp\_rcm = Centigrade\_readout(comfortable\_room\_temperature, 9);

// string value: “22.2222222 deg\_C” Nine significant figures were called for.

string temp\_out = Centigrade\_readout(comfortable\_room\_temperature + 23 \* Rankine\_deg);

// string value: “95 deg\_C” uncomfortable outdoor temperature

Some of the unit names in the second parameter of munit\_readout generate “split string” results. Examples:

munit\_quantity xxiat = 52.0 \* trig\_degrees + 5.0 \* trig\_seconds + 78.0 \* trig\_minutes;

string aoXXSt = munit\_readout(xxiat, trig\_seconds);

// string value: "53 degrees 18 minutes 5 seconds”

munit\_quantity xxiau = 1410.0 \* ft;

string aoXXSu = munit\_readout(xxiau, inches);

// string value: "117 ft 6 inches"

munit\_quantity xxiauv = 10677.5 \* ft;

string aoXXSm = munit\_readout(xxiauv, inches);

// string value: "2 miles 117 ft 6 inches"

The above, on these four pages is all you need to know, to use this package right out of the box.

In so doing, keep three things in mind:

**1. munit\_quantities.cpp** is a sample program meant to show what this package does.

2. In your main program, put **#include "munit\_QP\_standard.h"** as the first line after all the other required #include statements.

3. Don’t put in **int main(int argc, char \*argv[])** or the opening curly brace in it. These are included in the **munit\_QP\_standard.h**. Still, don’t forget the closing curly brace and return statement.

Read on, if you need to do some tweaking, like adding/deleting units or changing unit names.

Maybe you might want 53°18’5” instead of 53 degrees 18 minutes 5 seconds?

What about 10:18 instead of 10 hours 18 minutes?

Non-ASCII Unicode characters, such as the degree symbol, could come out looking funny on some screens and/or printers. And 10:18 could mean 10 minutes 18 seconds, depending on context. And what about more weight units, like grains or scruples? What about users who want to change almost all the unit names to fit a language other than English?

For this kind of tweaking, one has to edit **munit\_vectordefs.h** and maybe **split\_measure.h**. The non-English speakers would need to edit the names of the months and the days of the week in **days\_and\_months.h**.

Caution:

If you plan to add units (munit quantities to use as the second operand of munit\_readout statements) a matching quovink statement must previously be executed, in order for it to be valid.

Example:

in munit\_vectordefs.h Line 71:

munit\_quantity mile = 63360.0 \* inch; quovink(mile, miles);

on page 3 here: string reading\_out\_m = munit\_readout(distance\_driven, mile, 9);

The first operand of quovink matches the second operant of munit\_readout. The quovink automatically gets executed before the munit\_readout, because the former is in munit\_ vectordefs.h.

Two more provisos:

1. The first operand of quovink must match a previously-defined munit\_quantity.

2. The first operand of munit\_readout must be a scalar multiple of the second operand.

For an explanation of how quovink works, see page 6 here.

The munit\_vectordefs.h file

----------------------------------------------------------------------------------------------

The changes mentioned at the bottom of page 4 here are done by editing the **munit\_vectordefs.h**. **split\_measure.h** has to be edited only when putting in/editing the readout results of split units — split strings like 17’5” or 17 feet 5 inches.

Here is a look at the different parts of **munit\_vectordefs.h** .

Lines 30 through 50: the Constructor that is used in defining the “fundamental” units

In the Implementation, we’ll see how the constructor works.

The comment lines 34 and 35 are meant to emphasize the fact that the two fundamental units of time and angle are defined in other files of this package; and those files must not be edited by the user.

Lines 62 through 76: Scalar multiples of the Constructor-defined fundamental units

are defined.

With mm defined as the fundamental unit of length, some other units like mile, meter, and km are scalar multiples of mm (millimeter). Notice that the “unconventional” unit furlong is defined as an eighth of a mile.

Notice that these statements are a lot like the statements on the first page of this file; statement like “waiting time = 20 days“ and “hour = 60 minutes” are similar, because both are scalar multiples of the fundamental unit time\_second (defined in munit\_preliminaries.h).

Add some more scalar multiples of the fundamental units here, if you want to. Maybe you might want to define some “unorthodox” units, like grains or scruples?

Line 105 to end of file:

the units derived through multiplicative combinations of fundamental units

and, of course scalar multiples of these multiplicative combinations

Anyone who has had an elementary course in physics knows a few things like velocity = distance / time, acceleration = velocity / time, force = mass \* acceleration and energy = force \* distance.

Aside from physics and engineering, On Hand \* days is an important quantity in inbentory applications. And a job takes a certain number of worker hours. It kinda stands to reason that a job can be done by three people in one-third of the time it takes for one person to do it!

quovink and munit\_readout

----------------------------------------------------------------------------------------------

general form: quovink(runit, op2);

and: munit\_readout(subject, runit, sigdigits);

As stated in Page 4 here, runit (first operand of quovink and second operand of munit\_readout) have to agree with each other.

op2 is a string literal that is not enclosed in quotes, and there are some limitations on the charactres it can contain. For an explanation of how this works, read up on the C/C++ preprocessor’s stringizing operator.

Example: quovink(trig\_degree, degrees);

string rd\_angle = munit\_readout(67.5 \* gradients, trig\_degrees);

If op2 does not begin with a numeral, then subject / runit is computed, the result is converted to a string; and the second operand of quovink is concatenated to it.

In this example, subject / runit = 67.5 / (10/9) = 60.75 and op2 = “degrees”.

So, the munit\_readout would return the string “60.75 degrees”.

If op2 does begin with a numeral, control passes to the line in split\_measure.h that begins with “case xxxx”, where xxxx is the number specified by op2.

Example: in vectordefs: munit\_quantity inch = 25.4 \* mm; quovink(inch, 15257);

on page 3 here: munit\_quantity xxiauv = 10677.5 \* ft;

string aoXXSm = munit\_readout(xxiauv, inches);

and on to split\_measure.h.

– - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – - – – -

split\_measure.h

– - –- – - –- – - –- – - –- – - –- – - –- – - –-

must execute previously: quovink(runit, op2);

must form the subject.

then the readout stmt: munit\_readout(subject, runit, sigdigits);

and in split\_measure.h: usually the line:

case op2: splitting nstrings, strings, numbers end\_splitting;

with nstrings = 1 + the number of parts.

strings: one for each part – (nstrings – 1) strings

numbers: one for each part – (nstrings – 2) numbers

Example: in vectordefs: munit\_quantity inch = 25.4 \* mm; quovink(inch, 15257);

on page 3 here: mnit\_quantity xxiauv = 10677.5 \* ft;

string aoXXSm = munit\_readout(xxiauv, inches);

and in split\_measure.h:

case 15257: splitting 4, "", " miles ", " ft ", " inches", 5280, 12 end\_spliting;

In this example, op2 == 15257. So, control passes to the line in cplit\_measure.h that begins with case 15257.

nstrings = 1 + the number of parts. In this example nstrings = 4

In this example, the number subject / runit is to be split into three

parts: miles, feet, and inches, e.g. 2 miles 117 ft 6 inches.

strings: “miles”, “ft”, “inches” – the strings that represent the parts

three strings, since nstrings = 4

Notice the null string before, in the split\_measure line. See below for am example of where there is a non-null string in this position.

Numbers: the limiting numbers of each part

two numbers, since nstrings = 4.

In this example, 5280, 12 – in keeping with the fact that there are twelve inches in one ft and 5280 ft in a mile.

date arithmetic

– - –- – - –- – - –- – - –- – - –- – - –- – - –-

In the last part of the munit\_system.cpp file, it shows munit\_place dates and yrmo dates being formed and read out.

You might, just for the neck of it, want to check them against a spreadsheet.

Even though date arithmetic is available in spreadsheets, there doesn’t seem to be any such facility that one can put in C/C++ code; but with this package, now there is.

My date arithmetic is not quite accurate for years before the Gregorian calendar was universally adopted, but the spreadsheets’ date arithmetic isn’t either.