Improving code maintainability using Unimal

DesignCon 2001 HP-TF2

Scope of our interest for 3 hours

- **Code maintainability: Is something missing from programmer's toolbox?**
- **Why a new macro language?**
- **Introduction to Unimal**
- **Simple Applications**
- **A sketch of an advanced application**

We are interested in taming data

- constant within a given build but
- changing during development cycle
- changing among project's twin variants
- changing from year to year in a product line environment

Unimal motivation: Focus on reuse and maintainability

1. High cost of a software bug
2. Time-to-market considerations
3. Thus, emphasis on reuse of tried-and-true code
4. Code must be easy-to-configure by a project engineer

Variations across and within “model year” variants:

1. Impede software reuse
2. May require additional code development and therefore
3. Introduce new bugs

Commonly preached solution:

Plan ahead for reuse and maintainability.

I.e., treat a project as a member of a parameterized family of projects, whether actual or envisioned.

Runtime penalties of reuse and maintainability

- In general, there is runtime overhead for planned reuse and/or maintainability
- Even if there are only scalar parameters, instantiation can be far from simple
- **Unimal** is designed to expand cases with no runtime overhead

Example: Tabulating a function

Consider a function that is rather hard to calculate in real time. For our example, let's take a simple

$$myfunc(x) = (x-1)*(x+1),\ 0 \le x \le N,$$

with integer precision. A common way of coping with this situation is to tabulate such a function.

Assume that the only parameter varying among projects is the number N+1 of equidistant points of interpolation. I.e., we want to create an array Myfunc of N+1 elements, whose x-th element is

$$(x-1)*(x+1),\ x=0,\ldots,N.$$

Requirements

The Myfunc table consists of constant elements once N is fixed.

The **`Myfunc`** array must be created automatically, otherwise it is very difficult and error-prone to maintain. The choices are:

- Calculate the array values at runtime, during system initialization
- Calculate the array values at compile time

Runtime vs. compile time (static) initialization

1. Runtime initialization requires to link in some math support, thus increasing the code size (=ROM)
2. Runtime initialization slows down system initialization
3. With runtime initialization the table ends up being in RAM even though it is constant and logically belongs to less scarce ROM

Conclusion: runtime initialization is uniformly worse than compile-time initialization.

The goal should be static initialization of the table.

Limitations of high-level languages (and many assemblers)

Our example's goal is therefore to reproduce the functionality of the C statement

```
for (t=0; t<=N; t++) Myfunc [t] = (t-1)*(t+1);
```

at compile time.

To achieve this, static initialization of Myfunc table requires a source code sprinkled with compiler directives controlling the compiler in some special ways. Namely,

- We need some kind of a repetition mechanism, or loop, to force the compiler to re-scan a piece of source code repeatedly
- We need a way to arrange an incrementing compile-time counter (t) to calculate the values of the table elements.

The availability of these facilities depends solely on the programming language used.

I don't know of any HLL with *any* of these facilities, let alone all of them.

Code maintainability: Is something missing from the programmer's toolbox? (1)

Parameter sharing across languages

- A time-critical piece of a C project coded in Assembler. How to share a constant parameter (e.g., scale) between the two languages?
- A multiple-platform project depends on configuration parameters. How to share them between, say, C++ and makefile?

Tabulated functions

- Scale, range or resolution may change. How to maintain the table without runtime penalty?

Sparse tables

- How to make them readable if significant entries are lost among "don't care" entries?
- How to chop off leading and trailing "don't care" entries in a maintainable way?

Code maintainability: Is something missing from the programmer's toolbox? (2)

Derived data tables (e.g., lookup tables)

- Can they be generated automatically, to require no maintenance?
- If they are sparse, can they be compressed?

Header dependencies in a makefile

- Can they be generated regardless of the target language?

Typical applications:

- Consumer electronics
- Automotive controllers
- Home appliances
- Most mass-produced devices
- You name it!

Why a new macro language?

Powerful macro processors do exist. Why another one?

Let's list some requirements:

- easy to understand
- simple for simple problems
- powerful enough to solve complex problems
- preserving look and feel of the target programming language

Introduction to Unimal: First example

Maintainable static initialization of tabulated function (x + 1)*(x – 1)

```
#MP Set N = 10 ;Set by project engineer

const int myfunction[] = {
#MP For x = 0, N  ;For all values of arguments
  #MP Set temp=(x+1)*(x-1)
  #MP ;"print" the value
  #mp%dtemp,
#MP Endfor
};
```

```
const int myfunction[] = {
        -1,
        0,
        3,
        8,
        15,
        24,
        35,
        48,
        63,
        80,
        99,
};
```

Illustrating:

- Unimal operators
- Unimal target language interface
- For and Set operators
- Format specifiers
- Unimal comments

Repeated scanning of the source code

The following construct is a means of arranging a compile-time loop:

#MP For <parameter>=<expression1>, <expression2>
<loop_body>
#MP Endfor

Like a C operator

for(<parameter>=<expression1>, captured=<expression2>;
<parameter> <= <captured>; <parameter>++) {<loop_body>}

Assignments

The following syntax allows assigning a value of an expression to a symbolic variable:

#MP Set <parameter> = <expression>

Important: the <expression> can use the <parameter>, so it is possible to arrange a compile-time counter, e.g.,

#MP Set S=S+1

Making reusable tabulated functions: Unimal macros

Introducing:

- Macro definition

#MP Macro <Name>
<Macro_body>
#MP Endm

- Macro invocation

#MP Expand <macro_name> (<comma_separated_list_of_formal_parameters>)

Unimal translates macro call by expanding the macro body with formal parameters (#1# … #9#) replaced by actual parameters. A special parameter #0# denotes the number of actual arguments in the invocation.

Making reusable tabulated functions: Implemetation

1. Let's wrap in a macro the implementation of the function to be tabulated

```
#MP ;myfunc (t, N)
#MP Macro myfunc
    #MP Set temp =(#1#-1)*(#1#+1)
    #mp%dtemp,
#MP Endm
```

2. Let's wrap the generator of the table in a macro

```
#MP ;FuncTable (fName, Size, Func)
#MP Macro FuncTable
  const #mp%s#1#[#mp%d#2# +1] = {
  #MP For n = 0, #2#
      #MP Expand #3#(n, #2#)
  #MP Endfor
  };
 #MP Endm
```

3. Now, we have a reusable component. E.g., the original myfunc is generated by Unimal operator

```
#MP Expand FuncTable(#@int myfunction#, 10, #@myfunc#)
```

Discussion

1. Maintenance is straightforward on two levels:
1.1. The person maintaining the project simply specifies the **N**, and
1.2. The person maintaining the algorithm component using **myfunc** function modifies the **myfunc** macro as needed.
2. These two persons may or may not be one.

Compute operator

#MP Compute <parameter> = <function>(<a>, <b>, <c>, <d>)

computes (<a>/<b>)<function>(<c>/<d>)

Supported functions:

- Usin
- Ucos
- Uasin
- Uacos
- Uatan
- Uexp
- Ulog
- Usqrt

More realistic tabulated function: sine wave

Tabulating 11 equidistant points of sin(x) with 3 decimal digits

```
#MP ;sinewave(t, N)
#MP Macro sinewave
    #MP Compute temp=Usin(1000, 1, #1#, #2#)
    #mp%dtemp./1000,
#MP Endm

#MP Expand FuncTable(#@float mysine#, 10, #@sinewave#)
```

Output for the example →

```
const float mysine [10+1] = {
        0./1000,
        309./1000,
        588./1000,
        809./1000,
        951./1000,
        1000./1000,
        951./1000,
        809./1000,
        588./1000,
        309./1000,
        0./1000,
};
```

Conditional Evaluation

Syntax:

```
#MP If <expression>
<if_body>
#MP Endif

#MP If <expression>
<if_body>
#MP Else
<else_body>
#MP Endif
```

A variant:

```
#MP Ifdef <parameter>
```

Undefining a parameter

Syntax:

```
    #MP Undef <parameter>
```

Example: tabulating log(|x|)

```
#MP ;log_abs (t, N)
#MP Macro log_abs
    #MP If #1#<0)
        #MP Set temp = -#1#
    #MP Else
        #MP Set temp = #1#
    #MP Endif
    #MP Compute temp=Usin(1000, 1, temp, #2#)
    #mp%dtemp./1000,
    #MP Undef temp ;cleanup
#MP Endm
```

Parameter sharing across languages

Introducing:

- Include operator

Syntax:

#MP Include <filename_string>

Example: We want MYVAR to have the value of 17 and to be visible to C, Assembler and make.

C-to-be:	#define MYVAR (#mp%dMyVar)
Assembler-to-be:	MYVAR SET #mp%dMyVar
make-to-be:	MYVAR = #mp%dMyVar
All:	#MP Include #@myvardef.ui#
myvardef.ui:	#MP Set MyVar = 17

- Other applications exist, e.g., to store reusable macro definitions.

Export operator

Syntax:

#MP Export (<expression>) <filename_string>

Redirects output stream to the specified file. For empty name, #@#, it redirects to default stdout.

If expression evaluates to 0, output overwrites previous file (if any); otherwise, appends to the end of it.

Application examples:

- Exporting an array size to a header file
- Exporting offsets of named entries of an array to a header file

Maintainability of sparse tables

Consider, for the example, a sparse table of 32 entries, where significant entries have values 1…4:

1 at offset 9
2 at offset 11
3 at offset 24
4 at offset 27

All other entries are “don’t care.”

Using 0 for “don’t care” entries, we can write something like this:

```
const unsigned SparseTable[ ] = {
    0,0,0,0,0,0,0,0,0,1,0,
    2,0,0,0,0,0,0,0,0,0,0,
    0,0,3,0,0,4,0,0,0,0
};
```

Unreadable, not maintainable and error-prone!

Maintainability Goals

1. Supply only significant entries of the table. Don't care about "don't care" entries.
2. Allow listing significant entries in arbitrary order.
3. Reduce maintenance to editing (adding, removing) significant entries
4. Save ROM by chopping off leading and trailing "don't care" entries of the table.

Introducing:

- Composite names
- Ifdef
- Generating target language comments

Sparse tables: first implementation in Unimal

```
#MP Set Entry9 = 1   ;value at offset 9
#MP Set Entry11 = 2  ;value at offset 11
#MP Set Entry24 = 3  ;value at offset 24
#MP Set Entry27 = 4  ;value at offset 27

const unsigned SparseTable[32] = {
#MP For Index = 0, 31
    #MP Ifdef Entry%uIndex
     #mp%dEntry%uIndex, /* Offset=#mp%uIndex */
    #MP Else
     0, /* dummy */
    #MP Endif
#MP Endfor
};
```

Goals 1 and 2 and 3 are met (sort of).

Unimal output:

```
const unsigned SparseTable[32] = {
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            1, /* Offset=9 */
            0, /* dummy */
            2, /* Offset=11 */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            3, /* Offset=24 */
            0, /* dummy */
            0, /* dummy */
            4, /* Offset=27 */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
            0, /* dummy */
};
```

Improving the appearance

Introducing:

- Composite names

<name> ::= <base_name>[<format><simple_name>[…]]
<base_name> ::= <simple_name>

Simplified printf-style format: e.g., %u, %03d

Special format (in macro body only): %s

Example

```
#MP Set mything = 23
#MP Set mything%umything = 0 ; setting mything23
#MP Set mything = 45
#MP Set mything%umything = 0 ; setting mything45
```

Cosmetic improvement (1)

For a nicer look, let's use the following macros:

```
#MP ;DefineEntry(position, value)
#MP Macro DefineEntry
    #MP Set Entry%u#1# = #2#
#MP Endm
```

```
#MP ;SparseTableEnd(size, dummy_value)
#MP Macro SparseTableEnd
   #MP For Index = 0, #1#-1
    #MP Ifdef Entry%uIndex
           #mp%uEntry%uIndex, /* Offset=#mp%uIndex */
    #MP Else
       #mp%u#2#, /* dummy */
    #MP Endif
    #MP Endfor
#MP Endm
```

Cosmetic improvement (2)

```
const unsigned SparseTable[32] = {
   #MP Expand DefineEntry(9, 1)    ;value 1 at offset 9
   #MP Expand DefineEntry(11, 2)   ;value 2 at offset 11
   #MP Expand DefineEntry(24, 3)   ;value 3 at offset 24
   #MP Expand DefineEntry(27, 4)   ;value 4 at offset 27
   #MP Expand SparseTableEnd(32, 0)
};
```

Problems:

- array size (32) is entred two times, which is hard to maintain
- cannot handle arbitrary type of entries (they must be numbers)
- too C-centric; what about other target languages?

Language-independent sparse table allocation (1)

Illustrating:

- Strings #@**<character_sequence>**#
- Manipulating loop counter
- Hiding the loop (For and Endfor) in *different* macros

Target implementation:

```
#MP ;This is a declaration for C
#MP Expand SparseTableStart(#@const mytype SparseTable[#, 32, #@]={#)
   #MP Expand DefineEntry(9, #@  Ob1,#)   ;value 1 at offset 9
   #MP Expand DefineEntry(24, #@  Ob3,#) ;value 3 at offset 24
   #MP Expand DefineEntry(11, #@  Ob2,#) ;value 2 at offset 11
   #MP Expand DefineEntry(27, #@  Ob4,#) ;value 4 at offset 27
#MP Expand SparseTableEnd(#@  Dummy,#, #@};#)
```

Language-independent sparse table allocation (2)

Algorithm:

1. Find DefineEntry macro with the smallest offset. (If none remains, fill the remainder with dummy entries. End.)
2. Allocate the entry found in step 1 and exclude it from further consideration. Go to 1.

Idea of implementation:

Use loop

#MP For Step = 1, 2

and when processing case Step==2, reset Step to 0.

```
const mytype SparseTable[32]={
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   0b1,
   dummy,
   0b2,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   dummy,
   0b3,
   dummy,
   dummy,
   0b4,
   dummy,
   dummy,
   dummy,
   dummy,
};
```

Language-independent sparse table allocation (3)

```
#MP ;SparseTableStart(LinePart1, size, LinePart2)
#MP Macro SparseTableStart
    #MP Set TempSize = #2#
    #MP Set CurrLimit = TempSize
    #MP Set NextStart = 0
#mp%s#1##mp%uTempSize#mp%s#3#
  #MPFor Step = 1, 2
#MP Endm
```

Language-independent sparse table allocation (4)

```
#MP ;DefineEntry(position, string)
#MP Macro DefineEntry
   #MP If Step == 1
      #MP ;select an entry with smallest position
      #MP Ifdef Rendered%u#1#
         #MP ;if already rendered, ignore
      #MP Else
         #MP If #1# < CurrLimit
            #MP Set CurrLimit = #1#
         #MP Endif
      #MP Endif
   #MP Endif ;step 1
   #MP If Step == 2
      #MP ;Render the entry with smallest position
      #MP If #1#==CurrLimit
#mp%s#2#
         #MP Set Rendered%u#1# = 1
         #MP Set NextStart = CurrLimit+1
      #MP Endif
   #MP Endif ;step 2
#MP Endm ;DefineEntry
```

Language-independent sparse table allocation (5)

```
#MP ;SparseTableEnd(dummy_string, LastLine)
#MP Macro SparseTableEnd
   #MP If Step==1
      #MP For Index = NextStart, CurrLimit-1
         #MP ;render enough dummy lines
#mp%s#1#
      #MP Endfor
      #MP If  !(CurrLimit < TempSize)
         #MP ;render the last line
#mp%s#2#
         #MP Set Step=10 ;this will end the loop
      #MP Endif
   #MP Endif ;step 1
   #MP If Step==2
      #MP Set Step=0  ;to become a 1 on Endfor
    #MP Set CurrLimit = TempSize
    #MP Endif
  #MPEndfor
#MP Endm
```

Removing leading and trailing zeros from sparse tables (1)

(Considering for simplicity numeric sparse tables in C)

Target implementation:

```
const unsigned SparseTable[] = {
   #MP Expand DefineEntry(9, 1)    ;value 1 at offset 9
   #MP Expand DefineEntry(11, 2)   ;value 2 at offset 11
   #MP Expand DefineEntry(24, 3)   ;value 3 at offset 24
   #MP Expand DefineEntry(27, 4)   ;value 4 at offset 27
#MP Expand SparseTableEnd(0)       ;dummy value is 0
};
```

Array size omitted since it is not known in advance anymore.

Removing leading and trailing zeros from sparse tables (2)

```
#MP ;Minimum(number, running_min)
#MP Macro Minimum
   #MPIfdef #2#
      #MP If #1#<#2#
         #MP Set #2# = #1# ;Set to current minimum
      #MP Endif
   #MPElse
      #MP Set #2# = #1# ;Minimum of one element is the element
   #MPEndif
#MP Endm

#MP ;DefineEntry(position, value)
#MP Macro DefineEntry
   #MP Set Entry%u#1# = #2#
   #MP Expand Minimum(#1#, _MinOffset)
   #MP Expand Maximum(#1#, _MaxOffset)
#MP Endm
```

Removing leading and trailing zeros from sparse tables (3)

```
#MP ;SparseTableEnd(dummy_value)
#MP Macro SparseTableEnd
   #MP For Index = _MinOffset, _MaxOffset
      #MP Ifdef Entry%uIndex
               #mp%uEntry%uIndex, /* Offset=#mp%uIndex */
      #MP Else
          #mp%u#1#, /* dummy */
      #MP Endif
   #MP Endfor
#MP Endm
```

What we did:

1. Overlapped the 9 leading “don’t care” entries with preceding code or data.
2. Overlapped the 4 trailing “don’t care” entries with subsequent code or data.

Test drive

```
const unsigned SparseTable[] = {
          1, /* Offset=9 */
    0, /* dummy */
          2, /* Offset=11 */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
          3, /* Offset=24 */
    0, /* dummy */
    0, /* dummy */
          4, /* Offset=27 */
};
```

Generating the lookup table automatically

Goal: To generate lookup tables for tables of any objects completely automatically, to require no maintenance at all.
Framework: Objects are defined by their keys and some “other arguments” as arguments.

Example in C:

```
const ob_type myobjects[] = {
    {9, myob},
    {11, yourob},
    {24, ourob},
    {27, theirob},
};
```

Observe that the sparse table from our example is also the lookup table for the table of objects **`myobjects`**. But now it contains only data that can be calculated from the **`myobjects`**, so it potentially can be hidden from the application programmer. When the **`ObTable`** changes, so does the lookup table, but it will be transparent to the user!

Implementation goal

```
#MP Expand StartTable(#@const ob_type#, #@myobjects#)
    #MP Expand DefineOb (9, #@myob#)
    #MP Expand DefineOb (11, #@yourob#)
    #MP Expand DefineOb(24, #@ourob#)
    #MP Expand DefineOb(27, #@theirob#)

#MP Expand EndTable(#@unsigned myaccess#, #@const ⇓
unsigned short#, #@mylookup#)
```

The arguments of the macros are as follows:

- The StartTable takes two string arguments: the type of the array and the name of the array.
- The DefineOb takes a numeric argument (the key) and a string argument (the rest of the entry's definition).
- The `EndTable` takes three string arguments: the type and name of the access function, the type of the lookup table and the name of the lookup table to be generated.

Implementation (1)

```
#MP ;DefineOb(key, defintion_string)
#MP Macro DefineOb
   {#mp%u#1#, #mp%s#2#},
   #MP Set Offset = Offset+1
   #MP Set Entry%u#1# = Offset
   #MP Expand Minimum(#1#, _MinOffset)
   #MP Expand Maximum(#1#, _MaxOffset)
#MP Endm
```

Implementation (2)

```
#MP Macro StartTable
   #MP Set Offset = 0
   //This is the table definition
   #MP ; render the type and the name of the table
   #MP ; as passed
   #mp%s#1# #mp%s#2# [] = {
#MP Endm
```

```
#MP Macro EndTable
   }; //End of table of objects
   //This is the lookup table
   #MP ; render the type and the name of the lookup
   #mp%s#2# #mp%s#3#[]={
   #MP Expand SparseTableEnd(0)
   };
#MP Endm
```

Test drive

```
//This is the table definition
const ob_type myobjects[] = {
 {9, myob},
 {11, yourob},
 {24, ourob},
 {27, theirob},
};
```

```
//This is the lookup table
const unsigned short mylookup[]={
           1, /* Offset=9 */
    0, /* dummy */
           2, /* Offset=11 */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
           3, /* Offset=24 */
    0, /* dummy */
    0, /* dummy */
           4, /* Offset=27 */
};
```

Summary of our achievements

1. We do not enter the lookup table manually; Unimal builds it for us automatically.
2. Item 1 not only saves us typing; it saves much more in code maintenance.
3. Automatically built lookup table is more ROM-efficient than the one entered manually.
4. The macros we came up with depend very little on the example at hand; they can be reused almost 1:1 in different circumstances.
5. Elements of the table can be entered in any order.

Further improvements – splitting the key

TableH:		Previous data or code				
0						
1						
2	Table2					
3						
4						
5		Table2				
6	Table6	0				
7		1	A1			
		2				
		3	A2	Table6		
				0	A3	
				1		
				2		
				3	A4	
Subsequent data or code						

Splitting the key into two parts

9 = 01001B → (010B, 01B) = (2,1)
11=01011B → (010B, 11B) = (2,3)
24=11000B → (110B, 00B) = (6,0)
27=11011B → (110B, 11B) = (6,3)

TableH = {0, 0, Table2, 0, 0, 0, Table6, 0}

Table2 = {0, A1, 0, A2}
Table6 = {A3, 0, 0, A4}.

Further improvements – merging separate tables

TableH:					Previous data or code
0					
1		Table2			
2	Table2	0			
3		1	A1	Table6	
4		2		0	A3
5		3	A2	1	
6	Table6			2	
7				3	A4
Subsequent data or code					

Separate tables can be intertwined so that significant entries of one table fall in the “holes” of other tables.
This can be achieved by moving tables’ origins and allocating one table at a time when possible.

Target implementation

```
#MP Set SplitDivisor = 4  ;qoutient (3 bits) will be the digest
#MP               ;remainder (2 bits) will be the lookup offset
#MP Expand StartTable(#@const ob_type#, #@myobjects#)
    #MP Expand ObjDef (9, #@myob#)
    #MP Expand ObjDef (11, #@yourob#)
    #MP Expand ObjDef(24, #@ourob#)
    #MP Expand ObjDef(27, #@theirob#)
#MP Expand EndTable(#@const unsigned mylookup#, #@unsigned myaccess#)
```

Merging the tables – a sketch of implementation

A two-pass process followed by an iterative process.

- The two-pass process allocates the table of objects and the "hash" (primary lookup) table ala simple single-level lookup table generation. In addition, min and max offsets of all (secondary) lookup tables are calculated.

  Note: care must be taken not to attempt to allocate the same entry of the hash table more than once (since "digests" do repeat).

- The iterative process:

  1. shifts the next prospective allocation position of remaining secondary tables
  2. for each secondary table checks whether it has collisions with already allocated tables
  3. if a "good" secondary table is found, it is allocated at the current shift offset from the common beginning and marked as allocated. If no more tables remain, end, otherwise, repeat the process.

Summary

Consistent solutions for high-level languages and assemblers alike are available using **Unimal** (for UNIfied MAcro Language). It handles the static initialization tasks independently of the target language.

It allows to:

- Perform complex compile-time configuration
- Reduce maintenance complexity of your code
- Put in ROM what you had to configure in runtime before.
- Reduce memory requirements of your project
- Do more sophisticated arithmetic on parameters at compile time
- Export and share parameters between different languages (e.g., between C, FORTRAN, Assembler and the make utility)