Unimal 2.0

Application note 6

Compile-time sorting of symbolic constants

Documentation revision 2.00

Techniques:

Seamless integration of Unimal with the C language.
Complex compile-time algorithm implemented jointly by Unimal and C

MacroExpressions
http://www.macroexpressions.com

Table of contents

Foreword

In this Application Note, we'll see how Unimal can seamlessly integrate with C/C++ preprocessor (and the language proper) and promote project maintainability. This note assumes that the reader is reasonably versed in the C language.

The example problem we are to work on is to sort, at compile time, several numbers given *symbolically* (i.e., via C `#define` statements). This rules out the use of Unimal's own arithmetic capabilities because Unimal being language-independent is unaware of C symbolic constants. Therefore, we can only use Unimal to generate the C source that will do the sorting we want.

While the problem we'll be solving is a simplified version of a real-world problem of configuring an embedded log file system, the main point is that **we can use Unimal to extend the capabilities of the target language**. The second point is that no project maintainability is lost since the whole operation is fully automated.

The Note will be working on the following problem:

You have a few numbers made known as constant C expressions which may involve, say, a `sizeof` operator, so they cannot be used in an `#if` statement. For instance,

```
#define SIZE_A sizeof(long)
#define SIZE_B sizeof(short)
#define SIZE_C sizeof(char[3])
```

The problem is to create, at compile time, a constant array of *unique* numbers in ascending order and terminate it with one or more markers SIZE_BIG known to be greater than all supplied numbers, where, for instance,

```
#define SIZE_BIG 999999
```

For the data we have in the example, and for a typical machine, we want to automatically generate a statement equivalent to this:

```
const size_t sorted[] = {
    SIZE_B,
    SIZE_C,
    SIZE_A,
    SIZE_BIG
};
```

Our plan is as follows.

As a first shot at the problem, we'll see what a solution in plain C could look like. Of course, we have to limit ourselves to compile-time initialization since we are to define a const object.

Then, we will analyze whether it is reasonably maintainable. We will find that there will be statements looking like complete gobbledygook that no-one should be asked to maintain.

Third, we will come up with well-maintainable Unimal code which would automatically generate the C code that we want to have but do not want to maintain.

Fourth, we will test our solution on more data to sort and discover that a typical C compiler fails to compile a perfectly good source file. We will analyze the cause of it and come up with a different solution in C.

Finally, we will implement a maintenance-free Unimal code to generate the C solution that works.

Examples for this Note are in the directory Samples\AppNotes\6.

A target C solution

A C solution we are suggesting is a compile-time bubble sort of the numbers.
On pass 1, we will find the minimum of all numbers:

Pass 1:
```
#define MIN_1_1 SIZE_A
#define MIN_1_2 MIN(MIN_1_1, SIZE_B)
#define MIN_1_3 MIN(MIN_1_2, SIZE_C)
```

where, as usual,
```
#define MIN(a, b) (((a)<(b))?(a):(b))
```

Note that `MIN_1_1`, `MIN_1_2`, `MIN_1_3` are all constant expressions and `MIN_1_3` is the minimum of all numbers.

On pass 2, we will find the minimum of all numbers not previously matched (i.e., equal to `MIN_1_3`).

Pass 2:
```
#define MATCHED_1_1 (SIZE_A==MIN_1_3)
#define MIN_2_1 ((MATCHED_1_1)?SIZE_BIG:SIZE_A)
#define MATCHED_1_2 (SIZE_B==MIN_1_3)
#define MIN_2_2 ((MATCHED_1_2)?MIN_2_1:MIN(MIN_2_1, SIZE_B))
#define MATCHED_1_3 (SIZE_C==MIN_1_3)
#define MIN_2_3 ((MATCHED_1_3)?MIN_2_2:MIN(MIN_2_2, SIZE_C))
```

This is how it works: `MATCHED_1_x` serves as an indicator of a number being already equal to the minimum. `MIN_2_1` is a terminal marker (`SIZE_BIG`) if the first number was matched or that number otherwise. `MIN_2_2` is `MIN_2_1` if the second number was matched or the smaller of that number and `MIN_2_1` otherwise. Similar goes for `MIN_2_3`. It is easy to see that

- at least one of MATCHED_1_x is non-zero
- MIN_2_3 is the second smallest number, unless all numbers are equal, in which case MIN_2_3 is SIZE_BIG

Pass 3 takes, finally, a generic shape:

```
#define MATCHED_2_1 (MATCHED_1_1 || (SIZE_A==MIN_2_3))
#define MIN_3_1 ((MATCHED_2_1)?SIZE_BIG:SIZE_A)
#define MATCHED_2_2 (MATCHED_1_2 || (SIZE_B==MIN_2_3))
#define MIN_3_2 ((MATCHED_2_2)?MIN_3_1:MIN(MIN_3_1, SIZE_B))
#define MATCHED_2_3 (MATCHED_1_3 || (SIZE_C==MIN_2_3))

#define MIN_3_3 ((MATCHED_2_3)?MIN_3_2:MIN(MIN_3_2, SIZE_C))
```

The only difference from pass 2 is that MATCHED_n_x is non-zero (true) if it was matched on any pass before, i.e., if MATCHED_(n-1)_x was true OR it was matched on the previous pass. It can be seen that

- at least two of MATCHED_1_x are true
- MIN_3_3 is the third smallest number, unless there are less than three unique supplied numbers, in which case MIN_3_3 is SIZE_BIG

Now, we are in a position to define the array of unique supplied numbers in ascending order, like

```
const size_t sorted[] = {
    MIN_1_3,
    MIN_2_3,
    MIN_3_3,
    SIZE_BIG
};
```

We can give our solution a more uniform look if we artificially define all MATCHED_0_x to be 0. Here is our solution in C:

```
//pass 1
#define MATCHED_0_1 0
#define MIN_1_1 ((MATCHED_0_1)?SIZE_BIG:SIZE_A)
#define MATCHED_0_2 0
#define MIN_1_2 ((MATCHED_0_2)?MIN_1_1:MIN(MIN_1_1, SIZE_B))
#define MATCHED_0_3 0
#define MIN_1_3 ((MATCHED_0_3)?MIN_1_2:MIN(MIN_1_2, SIZE_C))
//pass 2
#define MATCHED_1_1 (MATCHED_0_1 || (SIZE_A==MIN_1_3))
#define MIN_2_1 ((MATCHED_1_1)?SIZE_BIG:SIZE_A)
#define MATCHED_1_2 (MATCHED_0_2 || (SIZE_B==MIN_1_3))
#define MIN_2_2 ((MATCHED_1_2)?MIN_2_1:MIN(MIN_2_1, SIZE_B))
#define MATCHED_1_3 (MATCHED_0_3 || (SIZE_C==MIN_1_3))
#define MIN_2_3 ((MATCHED_1_3)?MIN_2_2:MIN(MIN_2_2, SIZE_C))
//pass 3
#define MATCHED_2_1 (MATCHED_1_1 || (SIZE_A==MIN_2_3))
#define MIN_3_1 ((MATCHED_2_1)?SIZE_BIG:SIZE_A)
#define MATCHED_2_2 (MATCHED_1_2 || (SIZE_B==MIN_2_3))
#define MIN_3_2 ((MATCHED_2_2)?MIN_3_1:MIN(MIN_3_1, SIZE_B))
```

```
#define MATCHED_2_3 (MATCHED_1_3 || (SIZE_C==MIN_2_3))
#define MIN_3_3 ((MATCHED_2_3)?MIN_3_2:MIN(MIN_3_2, SIZE_C))
//resulting array
const size_t sorted[] = {
    MIN_1_3,
    MIN_2_3,
    MIN_3_3,
    SIZE_BIG
};
```

Maintainability and a Unimal implementation

Let’s assess maintainability of the C solution of the previous section.

The first problem is that the same number, like `SIZE_A`, is entered many times, so it is very error-prone to, say, rename this symbol.

Second, the somewhat tricky yet similar definitions are very likely to be corrupted by a typo or cut-n-paste operation.

Third, and worst, removing or adding a number, like some `SIZE_D`, will affect every pass and the final definition of the array, which, incidentally, for four numbers will look like

```
const size_t sorted[] = {
    MIN_1_4,
    MIN_2_4,
    MIN_3_4,
    MIN_4_4,
    SIZE_BIG
};
```

To solve these maintainability problems, we will try to have Unimal generate the code we want. Using a generic pattern, we’ll seek a solution in a form

```
#MP Expand BeginData()
#MP Expand DefineEntry(SIZE_A)
#MP Expand DefineEntry(SIZE_B)
#MP Expand DefineEntry(SIZE_C)
#MP Expand EndData()
```

Here, the macros to be defined, `BeginData`, `DefineEntry` and `EndData`, are independent of particular symbols supplied and are reusable, and each symbolic number is entered only once. This code is easily maintainable since the macros themselves don’t need any maintenance.

(It is important to emphasize that we invented the algorithm without any Unimal; we know what code we want to obtain, but we are not satisfied with its maintainability. It is the maintainability that turned us to seek help from an advanced macro language. This is a typical situation with Unimal, or any macro language for that matter: we must know in advance how to get what we want *in any particular case*, and then use Unimal to obtain a solution in a maintainable and reusable way.)

We will design the three macros in parallel.

First, we need Unimal to make as many passes over the sequence of `DefineEntry` macro calls as the number of macro calls in the sequence. There are two implications of this:

- All `DefineEntry` calls must be within a loop. Given the shape of the solution we seek, the `For` statement must be wrapped in the `BeginData` macro, and the matching `Endfor` must be in `EndData`. All `DefineEntry` macro calls are therefore located inside the "body" of the loop. Note that it is a unique feature of Unimal to allow this split of the loop across several macros.
- In the `For` statement, we cannot specify how many times to repeat the loop: we will learn it only when we pass over all `DefineEntry` calls and reach the `EndData` macro for the first time. So, we will have to manipulate the loop counter manually to achieve the effect of a do/while loop which Unimal doesn't have.

We will use the following macro parameters as Unimal compile-time variables:

- `done` – the loop counter kept at zero while we need more passes and switched to a 2 when done
- `pass` – a 1-based counter of passes
- `count` – a counter of the `DefineEntry` macros encountered during the pass
- `num_entries` – the number of the `DefineEntry` macros encountered in the end of the (first) pass
- some miscellaneous temporary variables

One of the duties of `EndData` will be to render the actual array in the end; for clarity, let's make a separate macro for it:

```
#MP Macro RenderArray
const int sorted[] = {
#MP     For i=1, num_entries
    MIN_#mp{%di}_#mp%dnum_entries, //(1)
#MP     Endfor
    SIZE_BIG
};
#MP Endm
```

Line (1) of the macro requires an explanation. We use the alternative syntax of the target language interface: `#mp{%di}` to render **`i`** as a decimal number. If we omitted the braces, Unimal would think the name of the parameter to render ends at the word boundary, and would therefore conclude that it is **`i_`**, which would be wrong.

We can see that line (1) is re-scanned `num_entries` times. Assuming, for instance, that `num_entries` is 3, this would produce

```
    MIN_1_3,
    MIN_2_3,
    MIN_3_3,
```

as desired.

Now we are in a position to define all macros:

```
#MP Macro BeginData ;()
#MP   pass = 1 ;Set for the first pass
#MP   For done = 0, 1 ; (0a) Start loop until done is 2
#MP     count = 0 ;(1a) Initialize the number of entries
#MP Endm
```

```
#MP Macro DefineEntry ;(size)
#MP     prev_count = count
#MP     count = count + 1 ;(1b) Count the current number of entries
#MP     prev = pass - 1

#MP     If pass == 1
#define MATCHED_1_#mp%dcount 0
#MP     Else
#define MATCHED_#mp{%d pass}_#mp%dcount \
    (MATCHED_#mp{%d prev}_#mp%dcount || \
    (#mp%n#1#==MIN_#mp{%d prev}_#mp%dnum_entries))
#MP     Endif
#MP     If count == 1
#define MIN_#mp{%dpass}_#mp%dcount \
    ((MATCHED_#mp{%d pass}_#mp%dcount)?SIZE_BIG:#mp%n#1#)
#MP     Else
#define MIN_#mp{%dpass}_#mp%dcount \
    ((MATCHED_#mp{%d pass}_#mp%dcount)?\
    MIN_#mp{%dpass}_#mp%dprev_count\
    :MIN(MIN_#mp{%dpass}_#mp%dprev_count, #mp%n#1#))
#MP     Endif
#MP Endm

#MP Macro EndData ;()
#MP     If pass == 1
#MP         num_entries = count ;(1c) Capture the number of entries
#MP     Endif
#MP     If pass == num_entries
#MP         done = 2          ;(0b) Force loop termination at Endfor
#MP         Expand RenderArray()
#MP     Else
#MP         pass = pass + 1   ;Continue with the next pass
#MP         done = 0         ;(0c) Force loop continuation at Endfor
#MP     Endif
#MP   Endfor ;(0d) End of loop started in BeginData
#MP Endm
```

To see how this set of macros works, recall that we are doing bubble sorting, so we re-scan the all the elements to sort `num_entries` times, where `num_entries` is the total number of elements (SIZE_A, SIZE_B, SIZE_C in our example). At the beginning, we don't know `num_entries` yet, so we need to count them in lines (1a), (1b) (so count is a 1-based ordinal number of the currently processed element. In line (1c) the number of elements is captured.

Since we do not know the number of loop repetitions in advance, we have to control loop continuation or termination manually. Line (0a) begins a loop with the dummy counter `done`. Line (0b) forces termination of the loop in line (0d) when we have made all the passes over the list of elements (and the next line renders the sorted table). If we have not made enough passes, line (0c) forces continuation of the loop in line (0d), with the incremented `pass` number.

The #define statements are created by `DefineEntry`; this is done by careful coding of the plan we designed in the previous section.

These macros are found in sortd3.u, along with the self-test code:

```
/* ------------ self-test -------------- */
#include <stdio.h>

#define MIN(a, b) (((a)<(b))?(a):(b))
#define SIZE_BIG 999999
#define SIZE_A sizeof(long)
#define SIZE_B sizeof(short)
#define SIZE_C sizeof(char[3])

#MP Expand BeginData()
#MP Expand DefineEntry(SIZE_A)
#MP Expand DefineEntry(SIZE_B)
#MP Expand DefineEntry(SIZE_C)
#MP Expand EndData()

int main()
{
    int i;
    for(i=0; i<sizeof(sorted)/sizeof(sorted[0]); i++) {
        printf("[%d]=%d\n", i+1, sorted[i]);
    }
    return 0;
}
```

Let's run it through Unimal,

```
Unimal sortd3.u >sortd.c
```

The result is as designed (with the gobbledygook shown in a smaller font):

```
/* ------------ self-test -------------- */
#include <stdio.h>

#define MIN(a, b) (((a)<(b))?(a):(b))
#define SIZE_BIG 999999
#define SIZE_A sizeof(long)
#define SIZE_B sizeof(short)
#define SIZE_C sizeof(char[3])

#define MATCHED_1_1 0
#define MIN_1_1 \
    ((MATCHED_1_1)?SIZE_BIG:SIZE_A)
#define MATCHED_1_2 0
#define MIN_1_2 \
    ((MATCHED_1_2)?\
    MIN_1_1\
    :MIN(MIN_1_1, SIZE_B))
#define MATCHED_1_3 0
#define MIN_1_3 \
    ((MATCHED_1_3)?\
    MIN_1_2\
    :MIN(MIN_1_2, SIZE_C))

#define MATCHED_2_1 \
    (MATCHED_1_1 || \
    (SIZE_A==MIN_1_3))
#define MIN_2_1 \
    ((MATCHED_2_1)?SIZE_BIG:SIZE_A)
#define MATCHED_2_2 \
    (MATCHED_1_2 || \
    (SIZE_B==MIN_1_3))
#define MIN_2_2 \
    ((MATCHED_2_2)?\
    MIN_2_1\
    :MIN(MIN_2_1, SIZE_B))
#define MATCHED_2_3 \
    (MATCHED_1_3 || \
    (SIZE_C==MIN_1_3))
#define MIN_2_3 \
    ((MATCHED_2_3)?\
    MIN_2_2\
    :MIN(MIN_2_2, SIZE_C))

#define MATCHED_3_1 \
    (MATCHED_2_1 || \
    (SIZE_A==MIN_2_3))
#define MIN_3_1 \
    ((MATCHED_3_1)?SIZE_BIG:SIZE_A)
#define MATCHED_3_2 \
    (MATCHED_2_2 || \
    (SIZE_B==MIN_2_3))
#define MIN_3_2 \
    ((MATCHED_3_2)?\
    MIN_3_1\
    :MIN(MIN_3_1, SIZE_B))
#define MATCHED_3_3 \
    (MATCHED_2_3 || \
    (SIZE_C==MIN_2_3))
#define MIN_3_3 \
    ((MATCHED_3_3)?\
    MIN_3_2\
    :MIN(MIN_3_2, SIZE_C))
const int sorted[] = {
    MIN_1_3, //(1)
    MIN_2_3, //(1)
    MIN_3_3, //(1)
    SIZE_BIG
};

int main()
{
    int i;
    for(i=0; i<sizeof(sorted)/sizeof(sorted[0]); i++) {
        printf("[%d]=%d\n", i+1, sorted[i]);
    }
    return 0;
}
```

We can actually compile and run this file; here is an output for a typical 32-bit machine:

```
[1]=2
[2]=3
[3]=4
[4]=999999
```

It is satisfying to notice that if we change the definition of, say, SIZE_A, to, for instance,
```
#define SIZE_A sizeof(char[3])
```
so that it becomes the same as SIZE_C, the Unimal source file will expand the same (with the exception of the new SIZE_C definition), but executing the resulting C file will yield
```
[1]=2
[2]=3
[3]=999999
[4]=999999
```

That is, we indeed sort unique numbers and do it symbolically.

But it doesn't work! Why?

The first troubling sign comes if we experiment further and use the following definition of SIZE_A:
```
#define SIZE_A sizeof(unsigned char[3])
```

If we use a Microsoft compiler, cl.exe, which comes with Visual Studio (all versions tried), the Unimal output doesn't compile, reporting an error where there is none.

The problem gets more exposed if we add another size definition (as in sortd4.u):
```
#define SIZE_D sizeof(char[3])
```
and, accordingly, add the line
```
#MP Expand DefineEntry(SIZE_D)
```
to our list of elements.

The Microsoft compiler goes belly up with INTERNAL COMPILER ERROR and an invitation to contact technical support. A less casually written compiler compiles the file correctly, but takes more than twelve minutes to do so. If we add yet another symbolic element to sort, the same compiler spends hours compiling. Finally, it fails with "out of memory" error.

This is an indication that the C file we've learned to generate is for some reason extremely hard to compile. It is our intention now to find the reason and to repair our solution.

If we carefully inspect our design in the section titled *A target C solution*, we can observe that our clever C macros are defined *recursively*, via previous similarly-indexed macros. This, of course, had been our intention in response to the challenge of symbolic sorting. But a closer look reveals that the *depth* of recursion is quadratic on the number of elements to sort. That is, a fully unfolded (expanded) C macro definition would be a terrible monstrosity.

Our trouble is, as it turns out, mandated by the C standard (actually by any of 'em ISO/ANSI C standards). The standard requires (roughly speaking) that a macro be expanded at a point of reference textually, and the expanded text be searched for any macros which in turn expand textually, and so on – recursively. So our quadratic depth of recursion in the C macro definitions puts a compiler to a stress indeed. We simply cannot abuse the C macro recursion so badly. We need a different mechanism.

Such a different mechanism does exist in C; it is to use the `typedef` statement which defines a new type. The trick is to define types with appropriate sizes; the sizes of the types would hold the numbers we want, instead of C macros. The difference is that `typedef` is not a part of C preprocessor; it is a part of C proper. That is, at the time a `typedef` (of a complete type, for you pedants) is encountered, all terms in the statement must be defined, and any recursion (one type defined through another) is not unfolded. The number of types we'll have to define is still quadratic (as is the number of C macros in the present design), but there is no recursion in processing the `typedef` statements.

A better C solution, using typedef

In our original design, we defined C macros `MIN_<x>_<y>`, x-th least element among the y elements scanned so far (in pass x).

In the new design, we will define types, `typemin_<x>_<y>`, whose *size* serves the same purpose, i.e. is the x-th least element among the y elements scanned so far (in pass x).

Since for any N `sizeof(char[N])` is N, it is handy to define any type we need as an array of type char.

We will need a helper type, `contrib_<x>_<y>`, whose size is the contribution of the y-th element to the size of `typemin_<x>_<y>`. This is similar to the macros `MATCHED_<x>_<y>` of the original design: the size of is the lesser of `typemin_<x>_<y-1>` and the contribution of the y-th element; this contribution is the (symbolic) value of the element provided that it is greater than the size of `typemin_<x-1>_<num_entries>`, the (x-1)-th least unique element. For uniformity, we'll say that the contribution of a previously matched element is SIZE_BIG.

Our template of recursive definitions might therefore look like this:

```
typedef char contrib_<x>_<y>
    [(sizeof(typemin_<x-1>_<num_entries>)>=<Value of x-th element>)?
    SIZE_BIG : <Value of x-th element>];

typedef char typemin_<x>_<y>
    [MIN(sizeof(typemin_<x>_<y-1>),
    sizeof(contrib_<x>_<y>))];
```

The initial definitions to prime the recursion can be as follows:

```
typedef char contrib_1_<y>[<Value of x-th element>];

typedef contrib_<x>_1
    typemin_<x>_1;
```

Indeed, the first least (that is, minimum) value has every element contributing to its calculation.

Now, let's translate these templates to Unimal

Implementation of the new design in Unimal

The new implementation is based on the previous one; the framework remains the same and we basically drop our type definitions in place of C macro definitions (see the file `sort.u`).

The macros `BeginData` and `EndData` remain unchanged.

`RenderArray` needs a repair in line (1): the x-th least element is now the size of the type `typemin_<x>_<num_entries>`.

```
#MP Macro RenderArray
const int sorted[] = {
#MP     For i=1, num_entries
    sizeof(typemin_#mp{%di}_#mp%dnum_entries), //(1)
#MP     Endfor
    SIZE_BIG
};
#MP Endm
```

`DefineEntry` changes by dropping in our type templates instead of C macros:

```
#MP Macro DefineEntry ;(size)
#MP     prev_count = count
#MP     count = count + 1
#MP     prev = pass - 1

#MP; The contribution of 'size' minimum calc is the 'size' if it is
#MP; greater than the previous min or SIZE_BIG otherwise
#MP
#MP     If pass==1
typedef char contrib_1_#mp%dcount[#mp%n#1#];
#MP     Else
typedef char contrib_#mp{%d pass}_#mp%dcount
    [(sizeof(typemin_#mp{%d prev}_#mp%dnum_entries)>=#mp%n#1#)?
    SIZE_BIG:#mp%n#1#];
#MP     Endif
#MP
#MP; size of typemin is the running min
#MP     If count==1
typedef contrib_#mp{%d pass}_1 typemin_#mp{%dpass}_1;
#MP     Else
typedef char typemin_#mp{%dpass}_#mp%dcount
    [MIN(sizeof(typemin_#mp{%d pass}_#mp%dprev_count),
    sizeof(contrib_#mp{%d pass}_#mp%dcount))];
#MP     Endif

#MP Endm
```

Let's now run the following self-test (sort.u):

```
/* ------------ self-test -------------- */
#include <stdio.h>

#define MIN(a, b) (((a)<(b))?(a):(b))
#define SIZE_BIG 999999
#define SIZE_A sizeof(long)
#define SIZE_B sizeof(short)
#define SIZE_C sizeof(char[12])
#define SIZE_D sizeof(char[3])
#define SIZE_E sizeof(char[6])
#define SIZE_X 27
#define SIZE_Y 27

#MP Expand BeginData()
#MP Expand DefineEntry(SIZE_Y)
#MP Expand DefineEntry(SIZE_A)
#MP Expand DefineEntry(SIZE_B)
#MP Expand DefineEntry(SIZE_C)
#MP Expand DefineEntry(SIZE_D)
#MP Expand DefineEntry(SIZE_E)
#MP Expand DefineEntry(SIZE_X)
#MP Expand EndData()

int main()
{
    int i;
    for(i=0; i<sizeof(sorted)/sizeof(sorted[0]); i++) {
        printf("[%d]=%d\n", i+1, sorted[i]);
    }
    return 0;
}
```

The Unimal output is quite lengthy for seven elements in this example. It is no surprise since the new design still calls for quadratic quantity of gobbledygook:

```
/* ------------ self-test -------------- */
#include <stdio.h>

#define MIN(a, b) (((a)<(b))?(a):(b))
#define SIZE_BIG 999999
#define SIZE_A sizeof(long)
#define SIZE_B sizeof(short)
#define SIZE_C sizeof(char[12])
#define SIZE_D sizeof(char[3])
#define SIZE_E sizeof(char[6])
#define SIZE_X 27
#define SIZE_Y 27


typedef char contrib_1_1[SIZE_Y];
typedef contrib_1_1 typemin_1_1;

typedef char contrib_1_2[SIZE_A];
typedef char typemin_1_2
    [MIN(sizeof(typemin_1_1),
    sizeof(contrib_1_2))];

typedef char contrib_1_3[SIZE_B];
typedef char typemin_1_3
    [MIN(sizeof(typemin_1_2),
    sizeof(contrib_1_3))];

typedef char contrib_1_4[SIZE_C];
typedef char typemin_1_4
    [MIN(sizeof(typemin_1_3),
    sizeof(contrib_1_4))];

typedef char contrib_1_5[SIZE_D];
typedef char typemin_1_5
    [MIN(sizeof(typemin_1_4),
    sizeof(contrib_1_5))];

typedef char contrib_1_6[SIZE_E];
typedef char typemin_1_6
    [MIN(sizeof(typemin_1_5),
    sizeof(contrib_1_6))];

typedef char contrib_1_7[SIZE_X];
typedef char typemin_1_7
    [MIN(sizeof(typemin_1_6),
    sizeof(contrib_1_7))];

typedef char contrib_2_1
    [(sizeof(typemin_1_7)>=SIZE_Y)?
    SIZE_BIG:SIZE_Y];
typedef contrib_2_1 typemin_2_1;

typedef char contrib_2_2
    [(sizeof(typemin_1_7)>=SIZE_A)?
    SIZE_BIG:SIZE_A];
typedef char typemin_2_2
    [MIN(sizeof(typemin_2_1),
    sizeof(contrib_2_2))];
```

<Let’s omit a few pages>

```
typedef char contrib_7_5
    [(sizeof(typemin_6_7)>=SIZE_D)?
    SIZE_BIG:SIZE_D];
typedef char typemin_7_5
    [MIN(sizeof(typemin_7_4),
    sizeof(contrib_7_5))];

typedef char contrib_7_6
    [(sizeof(typemin_6_7)>=SIZE_E)?
    SIZE_BIG:SIZE_E];
typedef char typemin_7_6
    [MIN(sizeof(typemin_7_5),
    sizeof(contrib_7_6))];

typedef char contrib_7_7
    [(sizeof(typemin_6_7)>=SIZE_X)?
    SIZE_BIG:SIZE_X];
typedef char typemin_7_7
    [MIN(sizeof(typemin_7_6),
    sizeof(contrib_7_7))];
```

```
const int sorted[] = {
    sizeof(typemin_1_7), //(1)
    sizeof(typemin_2_7), //(1)
    sizeof(typemin_3_7), //(1)
    sizeof(typemin_4_7), //(1)
    sizeof(typemin_5_7), //(1)
    sizeof(typemin_6_7), //(1)
    sizeof(typemin_7_7), //(1)
    SIZE_BIG
};

int main()
{
    int i;
    for(i=0; i<sizeof(sorted)/sizeof(sorted[0]); i++) {
        printf("[%d]=%d\n", i+1, sorted[i]);
    }
    return 0;
}
```

This compiles instantly without any problem; here is the (correct) result of the run on a typical 32-bit machine:

```
[1]=2
[2]=3
[3]=4
[4]=6
[5]=12
[6]=27
[7]=999999
[8]=999999
```

So, eliminating quadratic-depth recursion in a quadratic number of definitions made a real difference.