Alignment Proposal

Doc No: SC22/WG21/N1546
J16/03-0129

Date: 2002-05-02 to 2003-11-17

Project: JTC1.22.32

Reply to: Attila (Farkas) Fehér

Address: LM Ericsson Oy Ab
Hirsalantie 1
Jorvas 02420

Phone: +358 40 507 8729 (mobile)

Fax: +

Email: attila.f.feher@ericsson.com
whitewolf@mailbox.hu

Alignment Proposal

1 Executive summary

Document status: first draft, update 3. (Nonofficial comments has been removed)

One-liner: This proposal extends the core language and the standard library with variable alignment related features.

Problems targeted:

- Allow most efficient fixed capacity-dynamic size containers
- Allow specially aligned variables/buffers for hardware related programming
- Allow heterogeneous containers runtime/optional elements

Related problems not solved:

- Class-type "packing"
- Requesting specially aligned memory from memory allocators (`new`, `malloc`)

Proposed changes:

- New: *alignment-specifier* to declarations (type based and value based)
- New: `align_of` operator to retrieve alignment-value for a type (like `sizeof` for size)
- New: standard function for pointers for proper alignment runtime

Major open questions:

- Should weakening of alignment be enabled, if yes how?
- Where do we require diagnostics? Review.
- Is it possible to find syntax native to both C and C++ while still being readable?
- Prefer readability or less keyword? (separate keyword for type and number based alignment)
- Fix poor terminology

2 The Problems

Dynamic memory allocation might be expensive. Not only do the general-purpose allocators need precious CPU cycles to do their work, but also many of them scale badly to multithreaded environments. Once we have alignment specifications (and the rest of the alignment support I am going to introduce) it becomes possible to allocate one chunk of memory instead of many (creating a “flat” type) or even to allocate a container entirely on the stack.

Special alignment requirements cannot be portably specified today. With the proposed features it will be possible to create close-to-hardware variables and still keep the syntax portable.

2.1 Fixed capacity, dynamic size containers

Creation of fixed capacity, dynamic size templated sequential containers in today’s standard C++ language is not possible with reasonable effort and with only the minimum required overhead. This also means that such constructs cannot be created in a fully portable and memory-efficient way.

The need for such containers arises in cases when we want to avoid dynamic memory allocation. This can be due to runtime efficiency reasons or in case we need to use “flat” data layout.

Today’s solution to the problem is offered by template metaprogramming. One problem with template metaprogramming is that it is intrusive: the contained type T must be built in a way, that its members can be iterated or otherwise examined. This immediately rules out possible non-standard fundamental types, since they are not class types and we (the designers of the container) have no idea that they even exist, so we cannot specialize for them.

The other problem with template metaprogramming is code readability. The code has to be overcomplicated today, just to add otherwise simple – and inside the compiler existing – support for alignment specifications.

Without language support for alignment specifications the designer can only aim at partially efficient solutions. The first element of the container is either aligned for all possible types or for all types inside the type allocated and both of these waste memory. We can do better than this best guess strategy – with language support.

2.2 Support for special, aligned types

Programmers dealing with special alignment requirements today have to use non-standard extensions to achieve their goal. The problem with non-standard extensions is that they force programmers to revert to macro magic or duplication of code to make them portable (between compilers). This kind of special alignment requirement shows up mostly in embedded and close-to-the-hardware code (like drivers).

Question: *This case makes the picture more complicated. Containers are always of class type. Aligned variables on the other hand might be fundamental types. So the preparation has to take it into account that the rules might make it to C one day. So syntax should be “natural” in both C and C++. And while C++ begs for template syntax here, but C does not support such features.*

2.3 Conclusion

The extensions I propose add **support for generic programming, library building** and **systems programming,** since they enable performance without excess memory usage. They also **remove the**

embarrassment of not being able to make the most optimal code – with the least effort necessary. These are 4 out of the 7 motivators for change.

A possible future role of the features is the creation of standardized heterogeneous containers. This might involve a vector like reallocation scheme for a buffer containing the different types.

3 Status

The status of this document is first draft. Hence, at the moment it is rather a study with questions open than a full-blown change proposal with exact wording for the standard.

The document contains 3 major features that might be added, and variations on those features. It attempts to list all possible and meaningful alignment-related scenarios and to ask all the open questions.

4 The Proposal

4.1 Basic Cases

4.1.1 The *alignment-specifier*

The *alignment-specifier* can be used in variable declarations just like a *storage-class-specifier*. It can be used portably to specify the alignment requirement for the variable being declared.

> In general alignment requirement for a type can be strengthened, weakened or unchanged. Open question is whether it makes any sense to weaken the alignment requirements. The only place where it seems to be useful is the creation of "packed" class types. Since on many platforms accessing a misaligned variable is very costly and "packing" is not directly related to alignment I propose not to allow weakening.

The required alignment can be specified by using another type or by an integral number. The former is called *type-based* while the latter is called *value-based alignment-specifier*. See subsections (4.1.1.1, 4.1.1.2) for examples. This takes the following grammatical form:

alignment-specifier:
    type-based-alignment-specifier
    value-based-alignment-specifier

The *alignment-specifier* does not become part of the type (just like the *storage-class-specifier*).

Listing 1)

```
// The alignment-specifier does not change the type:
template <typename T> tfunc( T const &t) { ... }

void func( int  const &i):
void func( long const &i):

// ...

int align_as<long> aligned_var;

func(aligned_var); // Calls funct(int const &)
tfunc(aligned_var); // Instantiates tfunc<int>( int const &)
```

If the *alignment-specifier* would weaken the alignment requirement of the type, the program is ill-formed and diagnostics are required. The alignment requirement relations of related types have to be defined in the standard, just like as requirements for size relations are. The relations of alignment requirements between unrelated types are implementation defined, but every type has to have stronger or equal alignment requirements than the char type(s). This ensures that char (unsigned char) buffers can always be aligned.

Listing 2)

```
// The following line will emit diagnostics and the program becomes ill formed
int align_as<char> aligned_var;

// The following line must be OK for any imaginable type T:
char align_as<T> aligned_buffer[sizeof T];
```

See additional requirements for *type-based* aligned variables in the subsection.

The *alignment-specifier* will cause the given variable to be aligned according to the specification. See the subsections 4.1.1.1 and 4.1.1.2 about *type-based* and *value-based alignment-specifier* for the rules.

Listing 3)

```
// Make a buffer to store 50 variables of a type as an array:
unsigned char align_as<T> buffer[50*sizeof(T)];

// Make a type T in the first element:
T *firstT = new (buffer) T;  // Safe, buffer is well-aligned

// Make a T in the 42th element of the array
T *raw = static_cast<T*>(buffer);
T *theAnswer = new (raw+42-1) T;
```

The *alignment-specifier* can be applied to any variable declaration including member variables of class types. It cannot be used on declarations of function arguments or catch arguments.

Listing 4)

```
// The following lines are OK
unsigned char align_as<T> buffer[50*sizeof(T)];

template <class T, std::size_t S> class vectorRudiment {
  char align_as<T> buffer_[S*sizeof(T)];
};

if (bool align_as<double>b(1)) {  // Makes no sense, but allowed and will be aligned
  // ...
}

// The following lines are ill-formed
void func( int align_as<double> i); // aligment-specification in parameter-declaration
try {
  doSomething();
} catch (int align_as<double> &i) { // aligment-specification in exception-declaration
  // Handle exception
}
```

The *alignment-specifier* is optional in declarations of variables using the extern *storage-class-specifier*. This gives the opportunity to hide this special requirement from the users of library code.

Listing 5)

```
// Somewhere in a header file I use
extern char force[];

// Somewhere in a library implementation far far away...
class Force : DarkSide {
  Jedi *good; // etc.
};
char align_as<Force> force[sizeof(Force)];
```

It is not supported to create a type with changed alignment requirements, since the *alignment-specifier* does not become part of the type. On the other hand it is possible to create a class type with aligned member variable(s).

Listing 6)

```
// Wrong attempt:
typedef double align_with<0x10000> hwDoubleVector;  // Error!
Void clear(hwDoubleVector &toClear, unsigned size);

// Using C++, why not make a class?
template class <std::size_t S> hwDoubleVector {
 double align_with<0x10000> vec_[S];
 // etc.
};
```

4.1.1.1 Alignment by type

The *type-based-alignment-specifier* takes the form of:

type-based-alignment-specifier:
`align_as`<*type-id*>

The type used in a *type-based-alignment-specifier* has to be complete.

The variable will be well aligned for the given *type-id*. If the architecture does not require strict alignment the *type-based-alignment-specifier* will use the optimal alignment of the given *type-id*.

A variable declared this way must contain enough space to store at least one instance of the type used in the *alignment-specifier*. Otherwise the program is ill-formed and diagnostics are required. It makes no sense, so it is better caught at compile time. If it is necessary to make such a declaration *value-based alignment-specifier* can be used.

Listing 7)

```
// This is not gonna work
char align_as<T> buff[sizeof(T)/2];

// But this will
char align_with<align_of(T)> buff[sizeof(T)/2];
```

4.1.1.2 Alignment by alignment-value

The *value-based-alignment-specifier* takes the form of:

value-based-alignment-specifier:
`align_with`<*alignment-value*>

alignment-value:
constant-expression

The *constant-expression* shall be integral and representable using the type `std::size_t`.

The value can be one returned by the `align_of<T>` expression, zero, one or a value defined to be valid by the implementation.

- The value of zero does not change the alignment.
- The alignment-value 1 represents the alignment requirements of the type `char`
- A value returned by the `align_of<T>` expression represents the alignment requirements of type `T`.
- The effect of any other value is implementation defined. Implementations are encouraged to define this value based on number of bytes – if it makes sense on the given platform.

Listing 8)
```
// Aligning the buffer using a number:
template <std::size_t A, std::size_t S> class dyn_array_allocator {
  ...
  char align_with<A> buff_[S];
};
```

Using any value not defined here or by the implementation renders the program ill-formed. Diagnostics is required.

Listing 9)
```
// Aligning the buffer using a number not known by the implementation:
char align_with<11> buff_[S];  // Error!
```

This alignment style is added for completeness and to create the possibility for implementations to enable special alignment for the purposes of systems programming (hardware programming).

4.1.2 Getting the alignment-value

The `align_of` operator retrieves – during compile time - the alignment-value associated with a type. It is an *unary-expression* and takes the following form:

unary-expression:
    `align_of(`*type-id*`)`

Its value is an integral *constant-expression* of type `std::size_t`.

The return value is zero, one or any other – implementation defined – value.

The return value represents the required or – if there is none – the optimal alignment for the given type.

Listing 10)
```
// Getting alignment value of a type for later use where the type is unknown
const std::ptrdiff_t intAlign = align_of<int>;

// somewhere, in a Galaxy far, far away:
char align_with<intAlign> buff_[SomeConst];

// Or in the dark and secret chambers of some template metaprogramming genius:
template <typename T> struct magic {
  enum { value = align_of<T> };
};
```

***Question:** do polymorph types in a hierarchy have the same alignment by the nature of the art? If not – which would surprise me a lot – we will need runtime-evaluated form as well.*

4.1.3 Runtime pointer alignment

Aligning a pointer value means the process of increasing it to the closest value, which is well aligned for a given type T or for a given alignment-value.

The function introduced here is to be in the memory standard header.

The runtime pointer alignment function is based on value-based alignment to avoid template code bloat. Type based alignment can be achieved by using the `sizeof` and the `align_of` operators together in order to specify the alignment value and the required buffer size for the function.

The `std::align` function aligns a void pointer within a given buffer. It also checks if the aligned pointer and the object(s) it is supposed to point to will fit into the buffer. The function takes the from:

```
void *std::align( std::size_t align_val,
                  void *&ptr, std::ptrdiff_t &space,
                  std::size_t size) throw();
```

Aligns `ptr` using value-based alignment based on `align_val`, to form an aligned `size` bytes buffer.

Parameters:

• `align_val`	alignment-value as described in 4.1.1.2 above (*)
• `ptr`	the pointer to be aligned (**)
• `space`	the number of bytes allowed to be used (**)
• `size`	the number of bytes intended to be used after the pointer is aligned

(*) The value of zero and one means no alignment is done.

(**) Also set on successful alignment.

The return value of the function is a null pointer if the aligned pointer itself or the `size` bytes of buffer would not fit into the `space` bytes available to use at the address give in the `ptr` pointer. Otherwise the aligned pointer is returned.

The adjustment increases the value of `ptr` – if needed – to make it address a memory area properly or optimally aligned for the given type T and returns that pointer value if it is valid.

Upon success (not returning NULL) the function updates the `ptr` and the `space` arguments. The pointer is set to point right after the created buffer. The `space` is decreased by the number of bytes used up when moving `ptr`.

The function deliberately does not return a pointer to T, since there is no T there yet, until constructed or initialized.

Listing 11)

```
char *ptr;  std::ptrdiff_t space;
// Want to make 4 doubles there
void *alignedPtr = std::align( align_of(double),
                               ptr, space,
                               4*sizeof(double));
/* ptr == static_cast<char *>(alignedPtr)+size;
   std::ptrdiff_t tmp = static_cast<char*>(ptr)-static_cast<char*>(orig_ptr);
   space == orig_space - (tmp + size);                                   */
if (!alignedPtr) {
   // realocate, throw, abort, scream...
}
double *dblPtr = static_cast<double*>(alignedPtr); // Safe to do it
dblPtr[0] = 42.00; dblPtr[1] = 3.14; dblPtr[2] = 2.71; dblPtr[3] = 0.00;

// Then 10 of class type T
alignedPtr = std::align( align_of(T),
                         ptr, spaceleft,
                         10*sizeof(T));
if (!alignedPtr) {
   // realocate, throw, abort, scream...
}
```

The function does not check the validity of its arguments. Calling the function with invalid arguments results in undefined behavior.[1]

4.1.4 Generic, orientating examples

Please see the examples inside the descriptions above and below. *If someone can help me to rearrange the text to be still understandable and have the examples here I will do it.*

4.2 Advanced Cases

There are no cases which could be called advanced.

5 Interactions and Implementability

5.1 Interactions

The proposed features have a loose connection to the rest of the language. Hence they do not require any change in those, and they can be considered independently. However the power of portable alignment features can best be used together with templates.

5.1.1 Effects on syntax of the language

The extensions affect two major syntactic elements of the language: variable declaration and definition (*decl-specifier)* and expressions (*unary-expression*).

[1] The pointer is invalid, if space is not really present, if the alignment value is invalid

5.1.1.1 Effects on declaration (*decl-specifier)*

There is a new declaration specifier to be added, the *alignment-specifier*. Changes to the grammar are shown in boldface.

decl-specifier:
> *storage-class-specifier*
> ***alignment-specifier***
> *type-specifier*
> *function-specifier*
> `friend`
> `typedef`

alignment-specifier:
> ***type-based-alignment-specifier***
> ***value-based-alignment-specifier***

type-based-alignment-specifier:
> **`align_as`<type-id>**

value-based-alignment-specifier:
> **`align_with`<*alignment-value*>**

alignment-value:
> ***constant-expression***

5.1.1.2 Effects on expressions (*unary-expression)*

There is a new unary expression to add for the `align_of` operator. Changes to the grammar are shown in boldface.

unary-expression:
> *postfix-expression*
> ++*castexpression*
> −−*castexpression*
> *unary-operator*
> *cast-expression*
> `sizeof` *unaryexpression*
> `sizeof(`*typeid* `)`
> **`align_of(`*typeid*`)`**
> *new-expression*
> *delete-expression*

5.1.2 Effects on the semantics of the language

5.1.3 Effects on the type system

5.1.3.1 Not a type

Alignment may affect the "placement" of the variable it is applied to, but does not change its size and does not create a new type.

In declarations of variables – which are not also definitions – the *alignment-specifier* can be omitted, as long as it is present in the definition.

See section 4.1.1 above for examples.

5.1.3.2 Effects on class types

If an *alignment-specifier* is applied to a non-static member variable declaration in a class declaration, this alignment specification becomes part of the class type. It may change the layout, the size and the alignment requirements of the class type compared to one without that alignment specifier.

Listing 12)

```
class FirstClassSeat     : public AirCraftSeat {
   Seat seat_;
};
class TouristClassSeat   : public AirCraftSeat {
   Seat align_as<SardineCan> seat_;
};

// sizeof FirstClassSeat >= sizeof TouristClassSeat
```

If an *alignment-specifier* is applied to a static member variable declaration it does not change the layout, size or the alignment requirements for the objects of such class. The alignment specifier can be omitted from the declaration of the static member, as long as it is present in its definition.

Listing 13)

```
class A {
   static char buff[sizeof(double)*42]; // In the header no alignment-specification
};

// This is OK, the compiler only needs to know alignment when it creates the variable
char align_as<double> A::buff[sizeof(double)*42]; // Implementation
```

5.1.4 Strength of alignment

The strength of alignment is to be defined similarly to how sizes of types are defined in C++. The type `char` has the weakest alignment requirements of all, followed by short and so on. It is implementation defined (as for sizes) what is the connection between the alignment of a floating point and an integer type etc.

A type can have stronger or in other words stricter alignment requirements than another type. Then this other type has weaker alignment requirements.

5.1.5 Supporting arguments

There is no existing feature in the language to specify alignments for a variable. Only suboptimal workarounds can be achieved using existing language features. Most of them use either heavy template machinery or ignore the possibility of implementation specific fundamental types.

5.1.5.1 Fixed capacity, dynamic size containers

In today's language implementation of a fixed capacity, dynamic size container requires use of some magic tricks, and it will still not be an optimal solution. Please see the Andrei Alexandrescu article in CUJ[1] for one possible solution.

Any external (library like) solution to the alignment problems of such containers will either:

- be intrusive to the contained class (template metaprogramming used to built it and provide means of finding out the types in it) or
- waste memory (use an alignment which is suitable for all known fundamental types) or
- both

Furthermore all library based solutions fail to address the situation of implementation specific fundamental types. While it is arguable that a standard should not be concerned with those unknown types, it is also important not to rule them out.

Listing 14)

```
// The guts of an "array" class
template <class T, std::size_t S> class array {
  char align_as<T> buff_[S*sizeof(T)];
 public:
  // Operators, constructors and member-functions in the style of std::vector
  // Those, who might have thrown std::bad_alloc will throw std::out_of_space
};

// The use of such a container on the stack in some function
array<double, 42> dblArray;
// Imagine a complex operation adding numbers
   dblArray.push_back(somethingICalculated);

// Accessing elements is just like a vector
   double number = dblArray[12] / 42.0;

// While this array introduces a limit on the maximum number of elements
// it does prevent buffer overruns by wrapping a class around that array

// This kind of array type can be even made fully dynamic if VLAs are ever to be
// introduced into C++
```

Why do we need the alignment here? The construct is aimed at eliminating the need for dynamic memory allocation (runtime efficiency) while still ensuring minimal code execution (elements of the array are not constructed until requested) and minimal requirements on the type contained (no requirement for a default constructor).

If VLAs (Variable Length Arrays) will be introduced into C++, a similar construct can be done but with the size also dynamic – at creation. With such a type one could write functions to calculate the median of a collection without the need for dynamic memory allocation.

[1] URI: http://www.cuj.com/documents/s=7982/cujcexp2006alexandr/alexandr.htm

5.1.5.2 Optional elements

Optional elements of a class nowadays have to be made using dynamic memory allocation to avoid the requirement for a default constructor. In this case dynamic memory allocation is not necessarily needed. We know that there will be one and only one of this element – or zero. There is not much “dynamic” in it. The allocation is only required to get a properly aligned buffer. (In addition to this the type of this optional member might have a small size, which puts further burden on the allocation system.)

We need a buffer, well aligned for the type, but we do not want to have it initialized:

Listing 15)

```
// A preliminary attempt for a generic optional element
template <class T> class optional {
  bool exists_;
  char align_as<T> buff_[sizeof(T)];

  T &elem() { return *static_cast<T*>(buff_); }
  T const &elem() const { return *static_cast<T*>(buff_); }
  void set(T const &e) { if (exists_) elem()=e; else construct_(e); }

  void construct_( T const &e) { new (buff_) T(e); exists_ = true; }

  void destruct_() { if (exists_) elem().~T(); exists_ = false; }
  }
 public:
  optional() : exists_(false) {};
  explicit optional( T const &e) { construct_(e);};
  ~optional() { destruct_(); }

  // Assignment operator, copy assignment operator, copy constructor etc.
  T &operator =( T const &e) { set(e); return elem(); }
  optional(optional const &o) {
    if (o.exists_) construct_(o.elem());
    else exists_ = false;
  }
  optional &operator =( optional const &o) {
    if (o.exists_) set(e);
    else destruct_();
    return *this;
  }

  // The rest (access to T, acces to existence info)
  operator void *() const { return exists?buff_:0; } // For bool as streams

  T *operator ->() { if (exists) return &elem(); throw bad_access; }
  T const *operator ->() const { if (exists) return &elem(); throw bad_access; }

};
```

5.1.5.3 Special alignments for special hardware

Today this kind of variable can only be created using non-portable constructs.

Listing 16)

```
// Here only the alignment number may need to change when porting
double align_with<SOMETHING_FROM_HW_MANUFACTURER> hwVector[1024];
```

5.1.6 Language integration

Please see previous examples to examine how the proposed feature works together with existing features.

5.1.7 Effects on legacy code

There are now known effects to legacy code. The proposed additions do not change the meaning of existing code.

5.2 Implementability

As much as I know about compiler implementations I feel that most of the core issues needed to implement of this proposal are present in some form in all compilers.

If compiler intermediate code contains still types and not only objects of no type (addresses and sizes) the implementers need to add support for expressing different alignment requirements.

Most of the change must come into parsing and code generation. For non-member variables the code generator needs to generate code to align the variable properly according to the specifier. For member variables the class types layout needs to be generated according to the specified alignment requirements.

For the alignment operator compiler implementers need to uncover and document their internal alignment values as well as change them to byte based, if this is possible.

6 *Open questions, tasks*

6.1 Type for the alignment value

Should we introduce a new type for this value and the `align_of(T)` expression? It seems that `std::ptrdiff_t` is just about right for the task, but it leaves me with the feeling of the probability of coupling two absolutely unrelated things.

6.2 Implement alignment loss calculation

When aligning there will be some bytes lost. In case we have a data structure with many optional elements and the possible valid combinations of those are known we could just calculate the maximum required memory and “allocate” it at compile time. However we cannot just use sizes, since the alignment will cause some more memory usage. Neither in the current language nor in this proposal an operator exists to get this (maximum) alignment loss at compile time.

I have left such an operator out of the first draft proposal since it would require a new keyword (see my concerns later) and we can possibly achieve it without an operator.

The question is can we define alignment values as number of bytes for **all** possible platforms we know and can foresee? If yes, the imaginary `align_loss(T)` can be written as `align_of(T)-1`, since this is the maximum number of bytes that can be lost due to alignment.

If the alignment value cannot be defined in terms of bytes (like `sizeof` is), we will need to add an operator returning the maximum alignment loss in bytes to be able to create small buffer for flat types described above.

6.3 Value range of the alignment value

Should we enable negative values for alignment for the implementation defined value range? It seems it would not make sense at all, on the other hand to rule something out I feel I would need to find an absolute argument against it, not the lack of arguments to support it.

6.4 Fix terminology

Being new to standardization I am sure I have made some rather nasty mistakes in the language of this proposal.

6.5 Weakening of alignment requirements

Should we rather enable it in the standard? If yes, should we define it or leave it to the implementation?

6.6 Syntax, semantics, diagnostics

Review/fix what is ill-formed, where should we require diagnostics.

6.7 Find common syntax

Is it possible (or necessary) to find a common syntax for C and C++ implementation of the alignment features?

C++ begs for template syntax but the C grammar has no idea of that.

The runtime pointer alignment, the value based variable alignment (with `align_of`) and alignment of members of structures seems to be possible candidates for a C extension. I have tried to keep those C compatible, with the exception of the template-like syntax of the alignment specifier.

Keeping C and C++ close is a hot topic today, so I believe it is important to look at every proposal from this perspective. The compromise to be made in this proposal is to completely drop template like syntax in favor of the `sizeof` function like syntax. The drawback of this is that the syntax will not "harmonize with" the rest of the C++ language using types (except the `sizeof` operator)

6.8 Reduce the number of keywords while keeping reasonable readability

This proposal in its first draft state contains three new keywords: `align_of`, `align_as`, `align_with`.

I proposed three keywords provide readability and to avoid overloading of a keyword. For example looking at an alignment specification with its two separate keywords one does not need to track down the magic (template argument) identifier S to find out if it is a type or a number.

6.9 Finalize proposed names

Should the alignment of operator be `align_of` (and avoid collision with existing implementations) or should it be `alignof`? What shall be the final name of the runtime pointer adjusting function? See also "6.8 Reduce the number of keywords while keeping reasonable readability".

6.10 Fully implement the examples

The static-capacity dynamic-size vector, the optional element class, and a runtime built structure.

6.11 Make it shorter

I have a birth defect of writing too long, talking too much. IMO this proposal should be much shorter, I just don't yet see how.