Deep Copy Attach and Detach

Technical Report TR-16-1
Please Send Comments and Suggestions to feedback@openacc.org

OpenACC-Standard.org

April, 2016

Contents

1 Introduction

The reader should be familiar with OpenACC 2.5. This report proposes a solution to the *deep copy* problem. It has two parts. The first is a small but important change in behavior for the current OpenACC data clauses to allow manual top-down deep copy. The second is a small set of declarative directives to define true deep copy behavior for a whole data structure. While this report is proposed as an alternative to OpenACC technical report *TR-14-1: Complex Data Management*, most of the ideas here originated in that report.

This document is intended for both OpenACC users and implementers. One of the main goals is to get feedback to refine the proposal. To promote feedback, the document includes a number of explicit questions.

1.1 Required and Desired capabilities

As an example, we look at one complex data structure from the ICON climate code, developed by the German Weather Service (DWD) and the Max Planck Institute for Meteorology (MPI-M), seen in Figures 1.1 and 1.2. In addition to what is shown here, the member `diag` contains more than 80 allocatable or pointer array members, `metrics` contains another 80 allocatable or pointer array members, and each element of the array `prog` has additional array members.

This section very briefly describes capabilities that we believe are needed to support such hierarchical, dynamically allocated data structures. In all cases, the access mechanisms to reach an object in device memory after the deep copy should be the same as to reach the corresponding original object on the host.

Member shapes A Fortran allocatable and pointer member is either a scalar object, meaning the pointee is not indexed, or an array object and includes a descriptor that gives (at a minimum) the lower bound, extent and stride for each dimension of the array. In C/C++, a pointer member is just a pointer. The compiler can't tell if the pointee is a scalar or vector object, and can't tell how big the vector is. The exceptions are in C++, where a reference is just a pointer to a scalar and the `this` pointer is similarly a pointer to a scalar object. So, for C/C++, we need a directive to provide the size for a pointer member. The size may be an expression, and the elements in the expression may (and often will) involve other members of the same struct/class, or may be constant, or may even involve global variables. We can envision cases where the size is stored in a parent struct of the struct containing the pointer member, but this could be hard to describe.

```
type t_nh_state

  !array of prognostic states at different timelevels
  type(t_nh_prog),  allocatable :: prog(:)       !< shape: (timelevels)
  type(t_var_list), allocatable :: prog_list(:)  !< shape: (timelevels)

  type(t_nh_diag)    :: diag
  type(t_var_list)   :: diag_list

  type(t_nh_ref)     :: ref
  type(t_var_list)   :: ref_list

  type(t_nh_metrics) :: metrics
  type(t_var_list)   :: metrics_list

  type(t_var_list), allocatable :: tracer_list(:) !< shape: (timelevels)

end type t_nh_state

type(t_nh_state), allocatable :: p_nh_state(:)
```

Figure 1.1: ICON data structure

Full deep copy We want an option so that when processing a data clause for the base of a hierarchical data structure, all pointer members are themselves recursively processed in the same way. This has been shown to be useful for real applications. The runtime cost can be quite high, but it simplifies program development by deferring data movement minimization to a subsequent tuning phase.

Selective members For some applications, full deep copy is not appropriate. In fact, some applications go through phases, where different subsets of the hierarchical data structure type should be moved to and from the device. To support that, we need a way to support a partial deep copy. This document proposes *named policies*, with a way to specify what behavior to take for different structures given a named policy, and to specify what policy to use at a data clause.

Selective direction The current data clauses imply a direction (`create`, `copyin`, `copyout`, etc.). We can envision cases where some parts of a deep structure are input only, or need not be initialized, independent of other parts of the structure. This could be achieved using

```
p_nh_state(:)%metrics%rayleigh_w(:)
p_nh_state(:)%metrics%rayleigh_vn(:)
p_nh_state(:)%diag%vn_ie(:,:,:)
p_nh_state(:)%diag%vt(:,:,:)
p_nh_state(:)%diag%dvn_ie_ubc(:,:)
p_nh_state(:)%diag%e_kinh(:,:,:)
p_nh_state(:)%diag%w_concorr_c(:,:,:)
p_nh_state(:)%prog(:)%vn(:,:,:)
```

Figure 1.2: ICON device members

`update` directives, but that reduces readability and modularity.

Mutable deep copy We have seen cases where a deep structure on the device is not created or copied all at once; instead, it is copied in pieces. It may be copied top-down, from the *root*, then to the children, then the grandchildren, and so on. Conversely, it may be copied bottom-up, from the *leaves* to the parents and then the grandparents. Supporting both top-down and bottom-up means the runtime must recognize when a child is created on the device and update the pointers to that child, and recognize what children are already present when a parent is created on the device. In both cases, the runtime must dynamically *attach* an object to a new parent or *attach* a new object to an existing parent, and to *detach* an object from a parent being deleted or *detach* an object being deleted from a parent. The *attach* operation must update a pointer from the parent to the child, and the *detach* operation should restore the pointer in the parent to match the value in the corresponding host pointer.

Convenient syntax We should allow the simplest possible syntax for partial deep copy, and define how and when one object gets attached to or detached from another.

Question 1.1: This document assumes that the layout of objects (struct, class, derived type) will be the same in host and device memory. Changing the layout, statically or dynamically, is another challenging and important topic. At this time, we are not considering data layout changes. In the longer term, we see three possibilities for layout control:

- Define a method to statically control data layout on both host and device. The layouts would match when compiled with OpenACC enabled, but may not be the same as when compiled without OpenACC.
- Define a method to control data layout on the device. This would require dynamically changing the layout when transferring data between the two and would complicate code generation as well.
- Define a method to control data layout dynamically on the host as well as on the device. This would move the data layout changes to the host and simplify the data transfers, but could make for hard-to-find data layout bugs.

How important is data layout control? Are there some simple but useful motivating examples?

```
float* x;
#pragma acc declare create(x)
...
void sub(int n, float* y, float a){
   x = (float*)malloc(sizeof(float)*n);
   #pragma acc enter data create(x[0:n])
   #pragma acc parallel loop copyin(y[0:n]) firstprivate(a)
   for (i=0;i<n;++i) sub2(y,i,a);
   ...
}
#pragma acc routine seq
void sub2(float* y, int i, float a){
    x[i] = y[i]*a;
}
```

Figure 2.1: Simple motivating example

2 Simple Directives and API routines for Manual Deep Copy

This chapter proposes new behavior for existing OpenACC directives and data clauses to support manual deep copy with the existing directives, by adding new *attach* and *detach* behavior.

In C and C++, the shape of a pointer is the length of the array to which it points. If it points to a scalar, the length is one. For Fortran, the shape of an allocatable or pointer array is defined by the internal Fortran array descriptor.

2.1 Scalar Pointer

The initial motivating example is shown in Figure 2.1. The `enter data create(x[0:n])` will allocate an array of `n` floats on the device. However, `x` is a global variable, and the `declare create(x)` directive creates a static, global pointer `x` in device memory as well as in system memory. The reference to `x` in the procedure `sub2` on the device will use the pointer value in the global pointer `x` in device memory. How will that pointer get filled in?

It is natural that the `enter data create` should fill in the pointer `x`, since the base pointer in host memory corresponds to a base pointer that lives in the device memory. Generalizing this, any time a data clause causes memory allocation in device memory, if the base pointer to that memory is also present in device memory, that device copy of the base pointer should get updated. This is the *attach* operation. So, when data is created on the device, it is *attached* to the corresponding pointer, if the pointer is also present on the device. If

```
typedef struct points{
    float* x;
    float* y;
    float* z;
    int n;
    float coef, direction;
}points;

void sub(int n, float* y){
    points p;
    #pragma acc data create(p)
    {
        p.n = n;
        p.x = (float*)malloc(sizeof(float)*n);
        p.y = (float*)malloc(sizeof(float)*n);
        p.z = (float*)malloc(sizeof(float)*n);
        #pragma acc update device(p.n)
        #pragma acc data copyin(p.x[0:n],p.y[0:n])
        {
            #pragma acc parallel loop
            for (i=0;i<p.n;++i) p.x[i] += p.y[i];
            ...
```

Figure 2.2: Struct definition

the base pointer is not present, the *attach* is skipped.

We will also define an API routine `acc_attach` that can be used to implement the *attach* operation directly. Many users have asked that all API routines have directive analogs, and vice versa; to that end, we will also propose an `attach` directive or data clause.

Question 2.1: We could require the programmer to insert a separate directive to perform the *attach* operation. However, we expect programmers will want the *attach* more often than not. Is this true? The cost of the *attach* will be an 8-byte update; optimizing that update may be important for performance.

Question 2.2: The *attach* operation could be a problem if we have a machine where some memory is shared between host and device, and other memory is not. Suppose, in the above example, the pointer `x` lives in shared memory but the array allocated on the device for `x[0:n]` is in device-private memory. Attaching the array to the shared pointer `x` would overwrite the host pointer value, and not attaching the array would make the device array inaccessible through the base pointer. Is this likely to be an important situation to handle?

2.2 Simple Struct

Now let's consider a C struct, used as a scalar struct, as shown in Figure 2.2.

At the first `data` construct, only the struct `p` is allocated in device memory. At the `update` construct, the value of `p.n` on the host is copied to the device. At the second `data` construct, the arrays `p.x` and `p.y` are allocated on the device, and because `p` is already present, the

```
points p;
p.n = n;
p.x = (float*)malloc(sizeof(float)*n);
p.y = (float*)malloc(sizeof(float)*n);
p.z = (float*)malloc(sizeof(float)*n);
#pragma acc data copyin(p.x[0:n],p.y[0:n])
{
    #pragma acc data create(p)
    {
     ...
```

Figure 2.3: Building the structure bottom-up.

```
points* p;
...
/* allocate array p[:] */
#pragma acc enter data copyin(p[0:n])
for(i=0; i<n; ++i){
    /* allocate p[i].x[:], p[i].y[:]  */
    #pragma acc enter data create(p[i].x[0:p[i].n],p[i].y[0:p[i].n])
}
```

Figure 2.4: Simple array-of-struct example

device pointers `p.x` and `p.y` on the device will get attached.

In this case, the directive does both the *allocate* and *attach* because the base structure `p` is already present on the device. If we inverted the data constructs, as in Figure 2.3, the *attach* would not automatically happen. In this second example, the `data copyin` of the `p.x` and `p.y` arrays would not get attached to `p` at the first data construct, because `p` is not yet present on the device. At the second `data` construct, `p` is created on the device, and at that point doing the *attach* operation would require that the implementation check each pointer in the object to see if it were present, and to *attach* if so. Supporting that requires that the implementation knows the data layout of the struct, which the compiler does but the runtime typically does not. One of the advantages of having true deep copy behavior is that the data layout will have to be exposed at runtime, so even bottom-up data creation can end up with the data objects properly attached.

Question 2.3: This second example is typically called bottom-up data structure creation, where the root of the structure is at the top. How important is the automatic *attach* for bottom-up manual deep copy?

2.3 Array of Struct

A dynamically allocated array of struct, where each struct has one or more pointers to dynamically allocated arrays, can also be supported with these simple directives. However, this can be tedious, because each element of the parent array must be handled separately; see Figure 2.4.

2.4 Detach

Just as the allocate of a child member on the device must also *attach* to a parent pointer that is present on the device, the deallocate of a child member must *detach* from the parent pointer. A *detach* operation restores the value of the pointer so it matches the value in host memory.

We are proposing an API routine `acc_detach` that would do the *detach* operation as well, and also a `detach` directive or clause.

Question 2.4: As proposed here, the *detach* operation restores the value of the host pointer. There are three other possibilities. One is to set the device pointer to NULL; this has the same overhead as restoring the value, but allows the device code to check for NULL. A second possibility is to leave the stale address in the device pointer; this avoids the overhead of setting the pointer value. A third possibility is to save the original value from before the *attach*, and restore that original value; since we expect that in most cases, the pointer on the host will not change, restoring the value from the host pointer will be safe and efficient. If the host pointer had changed, then there is already a mismatch between host and device data. Should we consider any of these other options? Would the user want to control these? If so, what should the default be? And how would you propose to spell the control of alternate options?

2.5 Capabilities

Here we evaluate the simple directives and API routines against the required and desired capabilities from Section 1.1.

- Member shapes: Each directive can specify a shape for the member, each data API call similarly gives a shape (byte count) for the member. Unfortunately, there is no way in this proposal to declare at one point the desired shape for a member.
- Full deep copy: A full deep copy could be implemented manually, but it could be quite tedious. The main reason for having true deep copy, as in the next section, is to simplify the program.
- Convenient syntax: These directives have relatively simple and straightforward syntax, a natural extension to the behavior of existing data clauses.
- Selective members: The simple directives and API calls allow (in fact, require) a user to specify which members should be copied.
- Selective direction: Similarly, the simple directives and API calls allow (require) a user to specify the data direction for each member independently.
- Mutable deep copy: As shown earlier, top-down (parent first, then child) mutable deep copy is easily implemented when the *attach* operation is part of the data clause behavior. Bottom-up (child, then parent) mutable deep copy can also be implemented, but would require extra `acc_attach` API routine calls or extra `attach` directives

after creating the parent to then *attach* the parent structure pointers to the member pointers.

2.6 Open Issues

None of the simple directives proposed in this section would fix the update problem. In the last array-of-structs example, an update of the host copy of the array `p` from the device, or an update of the device copy of `p` from the host, would either overwrite the host pointers to `p[i].x` and `p[i].y` with their device pointer values, or vice versa. There is simply no way to fix this without the data layout information needed for a true deep copy implementation.

There are still open problems when there are multiple pointers to the same block of data. Sometimes this is intrinsic in the data structure, but other times this is simply using *convenience pointers* to temporarily point to some other part of the data structure. We have some work to do here.

None of the simple directives in this section try to address the problem for a system with partially shared memory, where the device and host share some memory but not all memory.

We have not yet explored the case where the device code modifies the pointer. In that case, it may be useful to have a *reverse attach* operation, that takes the device pointer value, resolves the host address, and updates the host pointer accordingly. In general, allowing changes on both host and device can be problematic, and we need to be clear what must be supported; we probably need some experimentation to help with this.

Question 2.5: Comments and feedback on this section? Comments and feedback on the proposals in this whole chapter?

3 Directives for True Deep Copy

This chapter proposes new directives, clauses and behavior for *true* deep copy, meaning with a single data clause (`copy(a)`), a user can specify copying a structure or array of structures, where the structure has dynamically allocated member arrays, some of which might themselves be arrays of structures that have dynamically allocated member arrays, and so forth. To support this, the compiler and runtime need to know:

- Which members are pointers or dynamically allocated arrays. This is known by the compiler.
- The sizes of these arrays. Array shapes are intrinsic for Fortran allocatable and pointer arrays. We propose a `shape` directive for C and C++ pointers.
- Whether to recurse for all such members or only a subset. We propose `include` and `exclude` clauses to support the selective member capability.
- Whether to move data in each direction for all members or only subsets in each direction. We propose `in`, `out`, `inout` and `create` clauses to support the selective direction capability.
- What to do if members are already present. We propose handling mutable deep copy by processing each member pointer or dynamically allocated array individually.

3.1 Policies

This document uses the concept of *policies*. For each data type, there is an unnamed *default policy*. There may be other *named policies* as well. A policy can have several deep copy attributes, including member selection, member direction and member shapes.

This document introduces a `policy` directive. This is a declarative directive that would normally appear in the declaration of a datatype (Fortran derived type, C struct, C++ class) to define behavior for objects of that type.

3.1.1 Default policies

An open question is whether the default behavior should be shallow copy or deep copy for hierarchical data structures. There are good arguments for either choice. One argument in favor of shallow copy default is that it matches current behavior, so current programs will not change behavior. One argument in favor of deep copy default is that it is more likely to be desired, and limits the amount of new syntax a user would need to learn. This document proposes the default to be deep copy for all shaped members, as defined later. In Fortran,

this means all allocatable and pointer members would, by default, be processed for all data or update directives. The default behavior may be changed by declaring attributes for the default unnamed policy on the `policy` directive.

Question 3.1: What do you think the default should be: deep or shallow? If deep, should there be a directive or clause for a shallow copy that doesn't require adding a new `policy`? If shallow, how would you suggest specifying a deep copy? Would you want different default policies in different programs, or different parts of the same program, or for different data structures?

3.1.2 Pointer Shapes

The member shapes for Fortran allocatable arrays and array pointers is explicit in the array descriptor. For C and C++, this document proposes a `shape` clause to the `policy` directive, which specifies the shape (length) of a pointer member. The shape may be an expression, involving other struct members, constants, global variables, or even function calls.

Question 3.2: Do you think we should add a global `shape` directive for C/C++ pointers?

3.1.3 Member Selection

We propose that by default, all shaped members get processed.

To override this default, this document proposes clauses to the `policy` directive. An `exclude` clause specifies which members to exclude, and an `include` clause specifies a list of members to include (and no others). The two clauses are mutually exclusive: you can specify `exclude`, meaning include all shaped members not explicitly excluded, or specify `include`, meaning exclude all members not explicitly included, but not both. The reason for allowing either `exclude` or `include` is some data structures have many members, and we expect most users will want to either exclude most or include most members, and we want to minimize the typing.

Question 3.3: Do you have alternate name suggestions for `exclude` or `include`? We want `include` to mean *include-only*.

Question 3.4: We disallow both `include` and `exclude` in the same datatype. Is there an example where you might want both? What should it mean?

3.1.4 Direction Selection

This document proposes that when a hierarchical data structure appears in a data clause, each member is treated as appearing in the same clause. In the example in Figure 2.4, a clause `copy(p[0:n])` would implicitly also imply `copy(p[i]:x[*])`, `copy(p[i]:y[*])`

and `copy(p[i]:z[*])` for all values of `i` and where `*` is the shape defined for that member. To allow for direction selection, a `policy` directive can have `in`, `out`, `inout` and `create` clauses. These attributes affect whether data is copied from host to device, or device to host, at entry to or exit from a `data` construct, at an `enter data` or `exit data` directive, or at an `update` directive.

Question 3.5: This is a very important decision. What data movement behavior do you want or need in your hierarchical data structure?

Question 3.6: A different name than `create` might be better, since `create` really has nothing to do with data motion. We want something that means the opposite of `inout`, but `none` doesn't really give a hint as to none of what. Do you have a suggestion?

Question 3.7: There is discussion about allowing the `present` clause here, which would mean the target needs to already be present and it would be a runtime error if the data were not present. There was also discussion about a `present_or_not` or `nocreate` clause, which would translate the pointer if it is present, but leave it untranslated with no runtime error if the data is not present. Do you see these as useful?

Question 3.8: Another open question is how to traverse the hierarchical structure when the parent is already present. Should the implementation recurse to the present object and check that any subobjects are likewise present (deep present), or assume that a present object is complete? This document proposes that deep copy means *deep*, and the implementation should ensure that the entire requested structure is present on the device. This decision does impact performance.

3.1.5 Mutable Deep Copy

This document proposes handling mutable deep copy by processing each child object as if it appeared in a separate clause, including the updates of the appropriate reference counts.

Top-down Deep Copy

A top-down deep copy can be done as in the previous chapter, manually. For a true top-down deep copy, a user might want to `copyin` one data structure with one policy, then `copyin` the same data structure with a different policy that adds some new children. If we think that is important, that settles the *deep present* question raised just above. Even though the parent is present, the runtime must traverse all the children that apply to this policy to make them present as well.

```
template<class _Tp, class _Alloc> class _Vector_base{
    public:
      ...
    protected:
    _Tp* _start;
    _Tp* _finish;
    _Tp* _end_of_storage;
};
```

Figure 3.1: Prototype C++ vector class

Bottom-up Deep Copy

Bottom-up deep copy becomes natural, because the parent will find the already-present children when the runtime traverses to those children. Those children would get their present counts updated for this construct.

Question 3.9: Another question is what about a child that is present, but is excluded from the current policy? Does that pointer get translated or not?

3.1.6 Relative Pointers

In many cases, a pointer member does not point to separate data, but into data that another pointer already addresses. The C++ *vector* class is a classic example, shown in Figure 3.1. In this case, `_start` points to the actual allocated data. The `_end_of_storage` pointer points to the first word beyond the allocated data, so it should not be followed at all. The `_finish` pointer could point somewhere into the middle of allocated data, or have the same value as `_end_of_storage`. Similarly, *ragged arrays* in C are often really pointers into the middle of a long vector of data. It would be easier and faster to move the long vector all at once, then simply translate the pointers into that data. Pointer translation differs from simply treating the translated pointers as *present* data. First, pointer translation allows for a pointer to invalid data, as with `_end_of_storage` in the `vector` class above. Second, pointer translation doesn't affect the reference counts for the target data. This document proposes syntax for pointer translation in C and C++.

3.2 Policy directive

Summary The `policy` directive is used to give attributes to the unnamed default policy or to a named policy of a user-defined type. The `policy` directive may appear in the definition of a user-defined type (without a type name) to apply to members of that data type. It may appear outside the definition of a user-defined type specification by including the type name in a `type` clause, in which case it can apply to (public) members of the named data type.

Syntax In C and C++, the syntax of the `policy` directive is

`#pragma acc policy` [`(` *policy-name* `)`] [*clause-list*] *new-line*

In Fortran the syntax of the `policy` directive is:

`!$acc policy` [`(` *policy-name* `)`] [*clause-list*]

where *clause* is one of the following:

`shape(` *member-shape-list* `)`
`exclude(` *member-name-list* `)`
`include(` [<*policy-name*>]*member-name-list* `)`
`in(` [<*policy-name*>]*member-name-list* `)`
`out(` [<*policy-name*>]*member-name-list* `)`
`inout(` [<*policy-name*>]*member-name-list* `)`
`create(` [<*policy-name*>]*member-name-list* `)`
`type(` *type-specification* `)`

More than one `policy` directive may appear in a struct, class or derived type declaration, or may appear with the same struct or class type name. When two or more `policy` directives appear with no name or with the same name, they are treated as if all the clauses appear on one directive. The `shape` clause is only valid in C or C++. A pointer member of a struct or class may only appear in one `shape` clause for that data type for any named policy, or for the unnamed default policy.

3.2.1 shape clause

The `shape` clause defines the size of a pointer member, or that the pointer member should be translated relative to another member. A *member-shape* is one of

member-name`[`*shape-expression*`]`
member-name`[@`*member-name*`]`
member-name`[@]`

In the first case, the named member is declared to have a size equal to the given *shape-expression* whenever an object of this data type appears directly or indirectly in a data clause, and the named member pointer is not NULL. The *shape-expression* may involve constants, other members of this struct or class datatype, member functions of this class, or any variables visible at the point where this directive appears.

Question 3.10: How complex an expression is needed for the shape? Will it need to support class member functions?

In the second case, the named member is declared to be a pointer that is an offset from the sibling member. When the named member is translated, the translated value will have the same offset to the translated sibling member as the original value has from the original sibling member. The translated value may not point to valid device data.

In the third case, the named member is declared to be a pointer that is offset from some other, unnamed pointer that is present on the device. In that case, if the named member value is not NULL and the data at the pointer is present on the device, the translated value will be the address of the device copy of the pointer.

If the `shape` clause appears on a `policy` directive with no policy name, the shape applies to any policy for this datatype unless the same member is explicitly shaped for that policy. If the `shape` clause appears on a `policy` directive with a policy name, the shape applies only to that named policy.

Question 3.11: Should there be an explicit `policy(default)`? Should there be a `policy(*)` for clauses that apply to all policies?

3.2.2 Member Selection and Direction Clauses

For any datatype and any policy, the default behavior will be to process every shaped member in the user-defined type, and recursively in every member that is itself a user-defined type. The `include` and `exclude` clauses override the default. Fortran allocatable arrays and array pointers are always shaped. C and C++ pointers are shaped only when they appear in a `shape` clause. The default data direction for all objects is the direction specified in the data clause:

directive	**clause**	**data movement**
`data`	`copy`	in and out
`data`	`copyin`	in
`data`	`copyout`	out
`data`	`create`	none
`update`	`device`	in
`update`	`self`, `host`	out

The `in`, `out`, `inout` and `create` direction clauses override the defaults.

A *member-name* may be the name of a dynamically allocated member of the user-defined type, or the name of a member of a user-defined type that has dynamically allocated members or submembers. For all but the `exclude` clause, a *member-name* may be (in C or C++) a *member-shape*. If a *member-shape* appears in a selection or direction clause, the given shape is used for this policy only, including the unnamed policy.

For all but the `exclude` clause, a *policy-name* may appear in the clause. If so, then the members named in the clause will be processed according to the *policy-name*. If no policy

name appears, the members named in the clause will use the policy named on this directive, or the unnamed policy.

Question 3.12: There has been discussion about allowing a hierarchy of policies, where a child policy could add or modify the behavior of a parent policy. Would this be useful? Do you have a motivating examples?

exclude clause

The `exclude` clause gives a list of dynamically allocated and aggregate member names. If there is an `exclude` clause, there must be no `include` clause. Any dynamically allocated member that appears in an `exclude` clause is *excluded*. An aggregate member that appears in an `exclude` clause is likewise *excluded*. All dynamically allocated or aggregate members of an excluded member are also *excluded*. If there is an `exclude` clause, then all shaped dynamically allocated members that are not *excluded* will be processed for this data type and policy. A *member-shape* may not appear in an `exclude` clause.

Question 3.13: Is there any reason to allow both `exclude` and `include` clauses in the same policy for a datatype?

include clause

The `include` clause gives a list of dynamically allocated and aggregate member names. If there is an `include` clause for a policy, there must be no `exclude` clause. Any dynamically allocated member that appears in an `include` clause or a direction clause is *included*. An aggregate member that appears in an `include` or direction clause is likewise *included*. All dynamically allocated or aggregate members of an included member are also *included*. If there is an `include` clause for a policy, then no dynamically allocated members will be processed for this data type and policy unless they are *included*.

The *member-name* may be any of the options specified under the member selection and direction clauses.

In the example in Figure 3.2, the `include` clause specifies a dynamically allocated member (`m2`) as well as an aggregate member (`mst`), which would then include the dynamically allocated member `mst.m1`.

The reason for allowing a *member-shape* here is to avoid having to give the member twice, once in `shape` and a second time in `include`. Similarly for the direction clauses below.

Direction Clauses

If a member appears in an `in` clause, `out` clause, `inout` clause or `create` clause, then the default data movement for that member or submembers of that member will depend on the directive and data clause as follows:

```
typedef struct st1{
    long sz;
    float* m1;
}st1;
typedef struct p2{
    long len;
    float* m2;
    float* q2;
    st1 mst;
    #pragma policy include(m2,mst)
}p2;
```

Figure 3.2: `include` clause.

```
struct t1{
    float* m1;
    float* m2;
    int n;
    #pragma acc policy in(m1[n]) inout(m2[n])
} t1_t;
struct t2{
    float* x1;
    t1_t s1;
    int m;
    #pragma acc policy in(s1,x1[m])
} t2_t;
t2_t* tt;
```

Figure 3.3: Direction clauses

		default data movement			
directive	**clause**	**in**	**out**	**create**	**inout or not given**
`data`	`copy`	in	out	none	in and out
`data`	`copyin`	in	none	none	in
`data`	`copyout`	none	out	none	out
`data`	`create`	none	none	none	none
`update`	`device`	in	none	none	in
`update`	`self`, `host`	none	out	none	out

The default direction is `inout`. If a `policy` directive specifies a direction for a member of aggregate type, and that aggregate type specifies a different direction for one of its submembers, the direction specified for that submember applies.

In the example in Figure 3.3, a `data copy(tt[0:100])` would have the following behavior on the members, for `i` from 0 through 99:

```
copyin(tt[i].x1[0:tt[i].m])
copyin(tt[i].s1.m1[0:tt[i].s1.n])
copy(tt[i].s1.m2[0:tt[i].s1.n])
```

Question 3.14: This is an important point but subtle. In Figure 3.3, the policy specifies direction `in(s1)`, but the datatype for `s1` specifies direction `inout(m2)`. Which should

apply to `m2`? Is the direction specified for `s1` a default, or a limitation?

3.2.3 type clause

The `type` clause may only be used and must be used when the `policy` directive appears outside the declaration of a C struct, a C++ class or struct, or a Fortran derived type. The type specification in the `type` clause may be the typedef name of a C or C++ struct or C++ class, the struct or class name (preceded by the keyword `struct` or `class`, as appropriate), or a derived type name in Fortran. In that case, the directive is treated as if it had appeared inside the struct, class or derived type specification.

3.3 Open Issues

Polymorphic datatypes in Fortran.

Dynamic types in C++.

Virtual function members in C++: this is a significant separate issue that needs some serious thought.

We may want something between a deep update and a deep present. All the possible behaviors are being discussed.

One big difference between this proposal and technical report *TR-14-1: Complex Data Management* is the lack of a single line nested syntax, such as

```
#pragma acc enter data copyin(tt[0:n]::{s1.m2[0:s1.n]})
```