The Managed Data Structures Library: C++ API

The Managed Data Structures Project, Hewlett Packard Labs

Tuesday, August 8, 2017

Contents

A Simple MDS Example

Before getting into the gory details of programming with MDS, we will first present a very simple worked example. To get the feel for what MDS brings, we'll first sketch out how you might program it if you weren't worrying about persistence or concurrency or multi-process sharing, then we'll discuss what you might do in a non-MDS world to take those into account, and finally, we'll walk through what it would look like with MDS.

The Scenario

Imagine that you are in charge of processing sales for a retailer. The company sells a large number of products, and for each product you have to keep track of how many are in stock, how many have been sold, what the current price is, and what the total revenue for the product is.

From time to time, you need to process one of the following actions that come in from outside:

- A new product has been introduced. You are told the product SKU, its initial price, and the initial stock.
- Stock has come in on a product. You are told the product SKU and the quantity received.
- The price has been changed on a product. You are told the product SKU and the new price.
- You are asked for a price check on a product. You are told the product SKU and return the current price.
- A sale has been made. You are given a list of line items, each containing a product SKU, a quantity, and a unit price. You must confirm that there is

sufficient stock for each and that the price for each is correct. If it is, you must adjust the stock on hand and revenue for each.

These are all examples of transactional processing and should be done atomically. From time to time, you will also get requests to provide various reports that may require walking over all of the products and may take non-trivial time. It is important that the report be *consistent*. That is, everything the report says must be true as of some instant in time, as if a snapshot of all of the data had been taken. In our simple scenario, the report we will be asked to provide will contain the total revenue over all products, an array containing the top ten products by revenue, and the value of current stock. In addition to being asked for reports, you are expected to create a report once a day and stick it somewhere where it can be found, so that trends can be identified.

Doing it Without MDS

Doing this without MDS—and without worrying about persistence, concurrency, or sharing—is straightforward. You create a *Product* class that contains the SKU, number in stock, number sold, price, and revenue, and you also provide a way to look up a product by SKU and a way to walk through all of the products. Since we're not worrying about products going away, for this example, we can support walking through the products simply by having each product point to the next product in the inventory.

Assuming that all of the necessary standard include files have been included and that

```
using namespace std;
```

is in force, we'll create a *Product* class that looks like

```
class Product {
    static final unordered_map<string, Product*> by_sku;
public:
    static Product *first_product;

    const string sku;
    unsigned n_in_stock;
    unsigned n_sold;
    float price;
    float revenue;
    const Product *next_product;

    Product(const string &s, float p, unsigned n)
    : sku{s}, n_in_stock{n}, n_sold{0}, price{p}, revenue{0},
        next_product{first_product}
    {
        first_product = this;
        by_sku[s] = this;
    }

    void check(unsigned q, float p) {
        if (price != p) { throw wrong_price{sku, p, price}; }
        if (n_in_stock < q) { throw insufficient_quantity{sku, q, n_in_stock}; }
    }

    void sell(unsigned q, float p) {
        revenue += p*q;
        n_sold += q;
```

```
        n_in_stock –= q;
    }

    static bool exists(const string &sku) {
        return by_sku.count(sku) > 0;
    }

    static Product *lookup(const string &sku) {
        Product *p = by_sku[sku];
        if (p == nullptr) {  throw no_such_sku{sku}; }
        return p;
    }
};
```

The actions then become straightforward:

```
Product *new_product(const string &sku, float price, unsigned n_in_stock) {
    if (Product::exists(sku)) { throw sku_exists{sku}; }
    return new Product(sku, price, n_in_stock);
}

void stock_in(const string &sku, unsigned n) {
    Product::lookup(sku)->n_in_stock += n;
}

void new_price(const string &sku, float p) {
    Product::lookup(sku)->price = p;
}

float price_check(const string &sku) {
    return Product::lookup(sku)->price;
}

void process_sale(const vector<LineItem> &items) {
    for (const LineItem &item : items) {
        auto p = Product::lookup(item.sku);
        p->check(item.quantity, item.price);
    }
    for (const LineItem &item : items) {
        auto p = Product::lookup(item.sku);
        p->sell(item.quantity, item.price);
    }
}

Report *get_report() {
    vector<Product*> all;
    float total_revenue = 0;
    float stock_value = 0;

    for (Product *p = Product::first_product; p != nullptr; p = p->next_product) {
        total_revenue += p->revenue;
        stock_value += p->n_in_stock * p->price;
        all.push_back(p);
    }
    sort(all.begin(), all.end(),
          [](const auto &p1, const auto &p2) { return p2->price < p1-> price; });
    vector<Product *> top_ten{ all.begin(), all.size() > 10 ? all.begin()+10 : all.end() };

    return new Report(total_revenue, stock_value, top_ten);
}
```

Dealing with Persistence

That's all very simple, but it does have some drawbacks. First of all, all of the data is sitting on the local heap of our process. If the power goes out or if the process crashes, it all goes away. To avoid this, we need to make sure that the data is stored somewhere persistent, which typically means either a file system or a database.

If we opt for using files, we'll need a representation for our products that can be read from and written to disk. A straightforward approach is to use a comma-separated values (CSV) file, where each line in the file represents one product and contains (a string representation of) the product's associated data, separated by commas. Establishing the inventory becomes something like

```
ifstream in{ csv_file_path };
string line;
using tokeniter = regex_token_itrator<string::iterator>;
while (getline(in, line) {
    vector<string> fields{ tokeniter{line.begin(), line.end(), comma_re, -1} };
    string sku = fields[0];
    unsigned n_in_stock = stoul(fields[1]);
    unsigned n_sold = stoul(fields[2]);
    float price = stof(fields[3]);
    float revenue = stof(fields[4]);
    Product *p = new Product{ sku, price, n_in_stock };
    p->n_sold = n_sold;
    p->revenue = revenue;
}
```

Except that we should probably add error checking just in case there's an I/O problem or one of the lines has the wrong number of fields or one of the fields doesn't parse as a number or the same SKU is in there twice or ... And we'd better be sure that the fields we refer to are in the same order that they were when they were written. There's a lot that can go wrong, and we have to be prepared for any of it.

In addition to reading the file, we have to write it, to make sure that it's there to be read next time. We could do this during each action (making sure that we are certain that the new file has been safely written before we acknowledge that the action has been performed. But this is very expensive. So what we'll probably do is write the file out relatively infrequently and for each action, we'll append to a "journal file", which just describes the actions performed. This is more efficient (but still costly, as we have to be sure that the journal entry has made it to disk before acknowledging each action), but it means that when we read the initial state, we have to not only read the CSV file, but also the journal, and we have to replay all of the journal entries that were written since the CSV file was written, taking care that we don't perform any externally-visible side effects (like journaling!) as we do so. And with either solution, we have to be aware that writing any of these files can fail at any point, so we have to be prepared to recover from half-written files or journal entries.

The alternative is to use a database. This has advantages—persistence is now somebody else's problem—but it also has drawbacks. Using a database typically means writing code to talk to it in a completely different language (SQL) which is very different from

the C++ the programmer is used to. It also means that somebody knowledgeable needs to set up the schemas for the database tables that the program will use. And typically every interaction with the database involves communicating with another process, often on another machine, over a socket, with all of the expense that implies.

Dealing with Concurrency and Sharing

The second problem is that with the approach detailed so far, we can only safely handle one action at a time. To see why, consider what would happen if two threads each tried to process a sale of ten units of some SKU for which there are 15 in stock. It's perfectly possible for each to get through the first loop and determine that yes, in fact there are at least ten and then for both to subtract ten from the stock on hand, leading to a final stock of negative five, which is exactly what the initial check was supposed to prevent. So the program will need to make use of locks or critical sections (e.g., by use of the standard *mutex* and *lock_guard* classes) to ensure that only one thread is performing an action at a time. With careful planning, it may be possible to allow partial overlap of actions, but getting that right can be quite tricky. In any case, writing to the journal files will also need to be synchronized to ensure that the files themselves don't wind up with garbage lines containing partial lines from different threads.

And even if the appropriate synchronization methods are used to allow multiple threads to access the data at the same time, it won't be possible to actually allow multiple processes on the computer to access the data at the same time, because the actual data structures will be within the local heap of a single process.

It's at this point that most programmers throw up their hands and either decide that everything will be done within a single process or that all of the actual work will be done by using a database, with little or no local data stored on the process's local heap. This can work, as in addition to persistence, database provide a notion of *transactions* that allow a thread to perform a number of actions and be guaranteed that either it is possible to pretend that all of the actions happened at a single instant in time or nobody else can see any of the effects of the actions and the thread can try again. Programs written in this way tend to look very different from those just working in local data structures.

And databases tend to not be good at dealing with long-running computations that want to see a consistent version of the data. So the program will typically arrange for the database to make periodic snapshots, holding off all other transactions while it does so, with all long-running analytics operations working on that snapshot data. This means that it becomes impossible to ask questions about the current state of the database as opposed to the state of the database when the last snapshot was taken, and because snapshotting is typically an expensive operation, it tends to be performed infrequently and at odd hours.

Doing it with MDS

With MDS, we can get much closer to the original in-memory solution. There are some differences, but they should be easy for a C++ programmer to wrap their head around.

The MDS analogue of a C++ object is a *managed record*. Like a C++ object, managed records have associated data and methods, and like C++ classes, managed record types can be defined to form an inheritance hierarchy. Unlike C++ objects, managed records are stored in a persistent *managed space*, which is shared between multiple processes. Because of this difference, there are a few minor differences between the way you use C++ objects and the way you use managed records, notably:

- All references to managed records must be held via an *mds_ptr<R>* smart pointer class.
- To obtain an *mds_ptr* to the object performing a member function (i.e., to **this**), use the **THIS_RECORD** macro.
- To create a new record, rather than saying

  ```
  new T(args…)
  ```

  which will create an object on the process's local heap, you say

  ```
  make_mds<R>(args…)
  ```

  which creates a managed record in the managed space. This is analogous to the way you say make_shared<*T*>(…) to get a *shared_ptr<T>* or make_unique<*T*>(…) to get a *unique_ptr<T>*.

- The record classes you create (more on this below) are *proxies* to the actual managed records. As such, there can be several such proxy objects over time (or at the same time) referring to the same managed object. The == operator on *mds_ptr<R>* works as you'd want it to (returning **true** when the pointers point to proxies to the same managed record), but
  - If you have extra (non-field) data members in the record class, they should be used only as caches and they *must* be default constructable.
  - The data members that represent record fields are of a smart reference type rather than the declared type (e.g., **double** or *mds_ptr<R>*). Usually, this won't be noticeable, but it may become visible when using **auto**, and you may occasionally need to add an explicit call to read() (e.g., r->fld.read()) to make the compiler happy.

So, what does a record (proxy) class look like in MDS? We'll build it up piece by piece. In what follows, we assume that

```
#include "mds.h"
```

has been used and

```
using namespace mds;
```

is in force (in addition to the std namespace).

Deriving from mds_record

First of all, all record classes must have *mds_record* as a (direct or indirect) base class, and this derivation *must be public*.

```
class Product : public mds_record {
    …
};
```

It's okay if it derives from other classes as well, but any other direct base classes *must be default constructible*. (MDS will create instances of your class as necessary to proxy for records returned when looking up fields in managed records, elements of managed arrays, or records bound in namespaces (see below).

MDS currently only supports single inheritance between record types, so at most one direct base class can be a record class.

Declaring record fields

This is where it starts to get a little weird, so bear with us. Instead of declaring the fields straightforwardly, as we did above:

```
class Product {
    ...
public:
    …
    const string sku;
    unsigned n_in_stock;
    unsigned n_sold;
    float price;
    float revenue;
    const Product *next_product;
    …
};
```

we need to declare the record fields using MDS macros:

```
class Product : public mds_record {
    …
public:
    …
    DECLARE_CONST_FIELD(Product, mds_string, sku);
    DECLARE_FIELD(Product, unsigned, n_in_stock);
    DECLARE_FIELD(Product, unsigned, n_sold);
    DECLARE_FIELD(Product, float, price);
    DECLARE_FIELD(Product, float, revenue);
    DECLARE_CONST_FIELD(Product, Product, next_product);
    …
};
```

Each field is declared by using **DECLARE_FIELD** or **DECLARE_CONST_FIELD**, resulting in a modifiable or unmodifiable field, respectively. Both macros take three parameters:

- the record class this field is in,
- the type of the field, and
- the name of the field.

These macros expand into a static member function and a data member of a smart reference type.[1] These macros can be used in public, protected, or private sections of the class.

Once all record fields have been declared, the **RECORD_SETUP** macro must be used:

```
class Product : public mds_record {
    …
public:
    …
    RECORD_SETUP(Product, mds_record, "myco.product",
        REGISTER_FIELD(sku),
        REGISTER_FIELD(n_in_stock),
        REGISTER_FIELD(n_sold),
        REGISTER_FIELD(price),
        REGISTER_FIELD(revenue),
        REGISTER_FIELD(next_product));
    …
};
```

RECORD_SETUP takes three parameters followed by one parameter for each field. The three non-field paramers are

- the record class,
- the record class this class derives from (i.e., *mds_record* or some other user-defined record class), and
- the name by which the record type should be known to MDS.

The record type name can be any string, but must be unique among all record types in the managed heap. In this case, we prefix the class name by *"myco."* to prevent collisions with anybody else's *Product* type. Note that if the record class is a template class, each instance of the template class must have its own record type name. That is, if the record class is *Node<T>*, the third argument to **RECORD_SETUP** must be different for different values of *T*.[2]

The field arguments all take the form

- **REGISTER_FIELD**(fld) or
- **REGISTER_FIELD_AS**(fld, name),

where the field name, which defaults to the declared name of the field, must be unique across all fields in this record and its ancestors. Typically, the latter form is only necessary if multiple levels of a hierarchy have private fields with the same (C++) name.

---

[1] *record_member<R,T,Fn>* and *record_const_member<R,T,Fn>*, if you see these names in compiler error messages.

[2] One way to do this is to use typeid(*T*).name().

If there are no fields declared for a record (occasionally useful for a subclass that has the same fields but different behavior), **NO_FIELDS** is passed to **RECORD_SETUP**:

```
class SpecialProduct : public Product {
public:
    RECORD_SETUP(SpecialProduct, Product, "SpecialProduct", NO_FIELDS);
    // override virtual functions
    ...
};
```

Note that the **RECORD_SETUP** invocation *must* be in a public section of the class.

Using Record Fields

Most of the time, you can use record fields as you'd expect. If *p* is an *mds_ptr<Product>*, you can say things like

```
n = p->n_sold;
p->n_sold = n;
if (p->n_sold > 5) { … }
p->n_sold += 10;
n = ++p->n_sold;
```

Unlike normal data members (but like *std::atomic* types), the assignment and modification operations return the resulting value rather than a reference to the field, which allows you to not worry about the value changing (in another thread or process) between the time you modify and the time you read it. If, instead, you want the value before the operation, you can use, e.g.:

```
old = p->n_sold.exchange(0);
old = p->n_sold.fetch_add(10);
old = p->n_sold++;
```

As you'd expect, if the field has a record type (as next_product does in our example), the value is assigned and returned as an *mds_ptr*.

As mentioned above, there are occasionally times when the compiler gets confused because n_sold *isn't* actually and **unsigned** and next_product *isn't* actually an *mds_ptr<Product>*. For those cases, you can use read() to explicitly obtain the value and write(val) to explicitly set it. Typically, you will only need to use read() when the compiler mysteriously claims that it can't figure out how to convert your field member into something that is seems as though it really should be able to.[3]

The check and sell methods can therefore be defined exactly as they were in the non-MDS example:

```
class Product : public mds_record {
    …
public:
    …
    void check(unsigned q, float p) {
```

[3] We will add special cases when we can when these situations are brought to our attention.

```
        if (price != p) { throw wrong_price{sku, p, price}; }
        if (n_in_stock < q) { throw insufficient_quantity{sku, q, n_in_stock}; }
    }

    void sell(unsigned q, float p) {
        revenue += p*q;
        n_sold += q;
        n_in_stock –= q;
    }
    …
};
```

Adding Other Data Members

It's possible to add non-field data members to a record class, and this is done just as in any other C++ class, but it's important to remember that the class you're defining is a *proxy* to the actual record class, which is kept on the MDS managed heap. This has a couple of consequences.

First, record proxy objects may come and go at any time—the lifetime of a record class object is independent of the lifetime of the referenced managed object. The only guarantee is that the proxy object is guaranteed to stick around as long as there are any *mds_ptr*s referring to it. There may even be multiple proxy objects at the same time referring to the same managed record. (The *mds_ptr* class's operator == will count all of them as the same.)

A consequence of this is that the MDS system needs to be able to create proxy objects to refer to managed records. It does this by using a constructor that only it can access, but this constructor assumes that all non-field data members are default constructible (i.e., they have a constructor that takes no arguments).

The net result is that non-field data members can be good for caching the results of computations that can be recomputed, but that they must otherwise be used with care.

Defining Record Constructors

Record class constructors are straightforward, but they require a first argument of type **const** *rc_token* &, which must be passed up to the record's parent record class (or *mds_record*):

```
class Product : public mds_record {
public:
    Product(const rc_token &tok, const string &s, float p, unsigned n)
    : mds_record{tok}, sku{s}, n_in_stock{n}, n_sold{0}, price{p}, revenue{0},
        next_product{ … }
    {
        …
    }
    …
};
```

Calling this constructor (via make_mds<*Product*>(s,p,n)) will result in a managed record being created on the MDS heap, with the fields initialized to the provided values. (We will go into detail on the initialization of next_product in a bit.) Fields declared using **DECLARE_CONST_FIELD** can be initialized in this way but cannot be assigned to (even in the constructor).

The *rc_token* argument to the constructor is created by the make_mds<*Product*>(…) call. You just need to pass it up. ***Do not use it for anything else!*** In particular, don't try to use it to create another record class object. The value is specific to this particular invocation of make_mds().

Note that any constructor that is to be used by make_mds() must be public. Private and protected constructors can be used for delegation from other constructors.

Forward-Declaring Record Classes

When a field of a record class holds a reference to another record class, as we do in this example:

```
class Product : public mds_record {
    …
public:
    …
    DECLARE_CONST_FIELD(Product, Product, next_product);
    …
};
```

the compiler must be able to tell, at the point of declaration that the field type is, in fact a record class. If the field type record class has been fully declared, this is no problem, but if it is a class only forward-declared, e.g., by

```
class SomeRecordType;
```

or if, as in this case, it is the class being defined, you need to give the compiler some help. You do this by defining a specialization of *mds::is_record_type* for the class:

```
class Product;
namespace mds {
    template<> struct is_record_type<Product> : std::true_type{};
}
```

We hope to figure out a way to remove this requirement in the future.

Managed Strings

You may have noticed that while the non-MDS code declared the sku field as

```
const string sku;
```

using *std::string*, in the MDS record class, we have

```
DECLARE_CONST_FIELD(Product, mds_string, sku);
```

*mds_string*s are references to *managed strings* stored on the MDS managed heap and can therefore be shared between processes. Like *std::string*, *mds_string*s have begin(), end(), cbegin(), cend(), front(), back(), size(), length(), empty(),

compare(), substr(), find(), copy(), at(), and **operator**[](),[4] and mds_strings can be compared for equality and inequality and passed (via **operator**<<()) to *ostream*s.

Unlike *std::string*s, the managed strings on the MDS heap are immutable, which means that mds_string provides no operators to change the content or get a non-**const** reference to the underlying characters. When assigning to an mds_string object, what is being assigned is the reference to the managed string.

In addition to being immutable, the MDS runtime ensures that there is only one copy of each unique string in existence. This means that equality and inequality checking is very inexpensive, regardless of the length of the managed string.

Another difference from *std::string* is that the underlying character type is a 16-bit **char16_t**.[5] To extract the characters, the easiest way to construct a *std::string* (or *std::wstring*) is to use the begin() and end() iterators. A UTF-8 copy may be obtained by calling utf8_copy().

mds_string has constructors that take

- **nullptr**
- begin and end iterators
- a *T* *s and a **size_t** len for character types
- a *T* *s and the constant *mds_string::***null_terminated** for character types. This indicates a pointer to a null-terminated string.
- a **const** *T* (&chars)[N], i.e., a literal string.
- a *std::basic_string<C,T,A>*, which includes *std::string* and the like.

Note that there is no constructor that simply takes, e.g., a **char** *. You are required to add an argument that either gives the length of the string or to declare that the string is null-terminated.

For all of these constructors, a check will be made to see whether a managed string already exists for the argument's content. If not, a copy will be made. Each constructor that only takes a single argument has a corresponding assignment operator (that calls the constructor and assigns the result). Equality and comparison operators are also defined between *mds_string* and local strings of the same sort as given above. In this case, the contents of the local string are used directly (i.e., no interned *mds_string* copy is made).

---

[4] rfind(), find_first/last_[not_]of(), and **operator** +() are not yet implemented, but will be.

[5] The underlying representation may be more compact for strings containing only 8-bit characters.

The underlying compare() and find() methods are defined in terms of compare_using() and find_using() which are like compare(), but take a first argument that is a function that takes two *mds_string::char_type*s and returns a negative value if the first sorts before the second, zero if the two values sort the same, and a positive value if the first sorts after the second. Two such functions, *mds_string::*case_sensitive and *mds_string::*case_insensitive are provided, so you can say, e.g.

```
if (s.compare_using(mds_string::case_insensitive, from, to) < 0) { … }
if (s.compare_using(mds_string::case_insensitive, "SENTINEL") != 0) { … }
```

compare() is defined as compare_using() with a first argument of case_sensitive. For **char*** arguments to compare(), if the pointer is followed by a length, that many characters are used, otherwise the string is assumed to be null-terminated.

Namespaces

In the non-MDS constructor, we had

```
/** Non-MDS code **/

class Product {
    static final unordered_map<string, Product*> by_sku;
public:
    static Product *first_product;
    …
    const Product *next_product;

    Product(const string &s, float p, unsigned n)
    : …, next_product{first_product}
    {
        first_product = this;
        by_sku[s] = this;
    }
    …
};
```

Two limitations of MDS require us to do parts of this in a more circuitous manner. First, MDS does not currently have a notion of maps in the shared MDS heap.[6] And second, static data members are shared between all instances of a class, but only (given current C++ compilers) within the current process. What we want here are a by_sku map and a first_product that are shared among all processes using this record type.

We solve both of these problems by use of MDS *namespaces*:

```
class Product : public mds_record {
    static const mds_ptr<mds_namespace> by_sku;
public:
    …
    DECLARE_FIELD(Product, Product, next_product);

    static const mds_ptr<mds_namespace> data_ns;
```

[6] We hope to support them very soon.

```
    static const mds_string first_product_key;

    Product(const rc_token &tok, const string &s, float p, unsigned n)
    : …
    {
        isolated([this]{
          next_product = Product::lookup_in(data_ns, first_product_key);
          THIS_RECORD->bind_in(data_ns, first_product_key);
        });
        THIS_RECORD->bind_in(by_sku, s);
    }
    …
};

const mds_ptr<mds_namespace>
    Product::data_ns = mds_namespace::from_path("MyCo/data");
const mds_ptr<mds_namespace> Product::by_sku = (*data_ns)["SKUs"];
const mds_string Product::first_product_key = "first_product";
```

We'll talk about the **isolated** bit in a second, but first we'll walk through the rest.

MDS namespaces are like directories (or folders) in a file system, but instead of mapping names to directories and files, they map names to namespaces and managed objects. As with directories in a filesystem, namespaces are hierarchical, and they can be specified by strings (or *mds_string*s) that can represent paths, which are names separated by a delimiter (by default, a slash). So in

```
const mds_ptr<mds_namespace>
    Product::data_ns = mds_namespace::from_path("MyCo/data");
```

we establish a particular namespace that will hold our program's data. This namespace is the one named *"data"* within the namespace named *"MyCo"* within the root namespace, which is specified by an initial slash: *"/"*. Within this namespace, we will keep our shared "static" variables. Instead of creating an *unordered_map* to hold the mapping from SKU to *Product*, we will use another namespace:

```
const mds_ptr<mds_namespace> Product::by_sku = (*data_ns)["SKUs"];
```

As with the non-MDS map, we declare this in the private section of the class. The namespace we use will be the one named *"SKUs"* within the data_ns namespace we established earlier. And instead of a static *Product* * to hold the first product in the list, we will use the *Product* bound to the *"first_product"* name in the data_ns namespace.

As with managed records, namespaces are referred to by *mds_ptr*s. The paths used are of type *path*, but often *string*s or *mds_string*s can be used instead.

If ns is an *mds_ptr<mds_namespace>* and p is a *path*, you can say things like:

```
ns = mds_namespace::root();
ns = mds_namespace::current();
mds_namespace::current() = ns;
ns = mds_namespace::from_path("/some/path");
ns = mds_namespace::from_path(p, "data", "count");
ns = mds_namespace::from_absolute_path("a", "b/c");
```

```
ns2 = ns->parent();
ns2 = ns->at("a", "b/c");
ns2 = (*ns)["a/b"];
if (ns->is_root()) { … }
p2 = p.resolve("more", "components");
p = path::of("/some", p2, "path");
if (p.is_absolute()) { … }
```

mds_namespace::current() is a reference to a thread-local namespace pointer that starts out as the root namespace. When using from_path(), if the first component is an absolute path or a string that begins with *"/"*, the resolution begins with root(), otherwise it begins with current().

Unlike directories in a file system, bindings in a namespace are *typed*. You can bind any object to any name in any namespace, but when you get a value out of a namespace, you have to say what type you expect it to be. If there's no value there, you'll simply get back a default-constructed value (typically a zero, **false**, or null pointer), but if the value that's there is of an incompatible type, an *incompatible_type_ex* will be thrown. This means that if you don't get an exception, you don't have to worry about error-checking. If you expect a *Product* and get a value, you can be confident that what you got was a *Product*.

Binding and looking up values is straightforward. Our example gives examples for record classes. In the constructor, we bind the *Product* we're creating to its SKU in the by_sku namespace:

```
THIS_RECORD->bind_in(by_sku, s);
```

Records also have a bind_to_name() method that omits the first (namespace) parameter. Note that this binding will succeed even if the previous value at that name in that namespace was something other than a *Product*. Similarly, in push_and_get_first(),[7] we assert that the record we're creating is the first *Product* record:

```
THIS_RECORD->bind_in(data_ns, first_product_key);
```

To do the lookup of the old value of the first product "static", we call

```
old = Product::lookup_in(data_ns, first_product_key);
```

This does a lookup in data_ns using the first product key (*"first_product"*) and asserts that what it expects to find there is a *Product*. Every record class defines static lookup_in() and lookup_name() methods. Note that the lookup and bind methods can take any number of path components.

Note that we said above that we check to see whether the actual type is incompatible with the expected type. If we had done the above lookup and it turned out that what

[7] We'll get to the weird **isolated** call and the reason we have a separate method in just a bit.

was there was one of the *SpecialProduct* records we defined earlier, or some other derived record type, the lookup would have succeed and what was returned would have been a pointer to a proxy of the correct type.

For non-record types, a slightly different method is used. To lookup a value, use the as<*T*>() method to establish the type you expect:

```
n = (*ns)[path].as<long>();
if (ns->resolve(data_path, "done").as<bool>()) { … }
n = (*ns)[path].as<mds_string>()->size();
```

Note that you need to use as<**bool**>() to read the value as a bool. Simply saying

```
if (ns->resolve(data_path, "done")) { … }
```

will succeed if the name is bound to *anything, including* **false** *or a value of another type*.[8] It's equivalent to

```
if (ns->resolve(data_path, "done").is_bound()) { … }
```

To set a value, you have a few options

```
(*ns)[path].as<unsigned>() = 5;
(*ns)[path].bind<unsigned>(5);
(*ns)[path].bind(5);
(*ns)[path] = 5;
```

Note that the last two examples are not the same as the first two. In the last two, the type will be inferred from the argument, so they are, in fact, equivalent to

```
(*ns)[path].bind<int>(5);
```

since the literal 5 will be taken as an **int**.[9] The inferring forms should only be used to assign values when the argument will be known to the compiler to be of the correct type.

Note that there currently is no way to make an atomic modification (e.g., an increment or get-and-set) to a binding. We hope to add this soon.

With namespaces out of the way, we can now define the exists() and lookup() calls. Recall that without MDS they were defined as

```
/** Non-MDS code **/

static bool exists(const string &sku) {
```

[8] In retrospect, while this is consistent with the way **operator bool**() tends to work in the C++ standard, it's probably too error prone to be allowed to stand.

[9] And in the current implementation, a value bound as an **int** cannot be read as an **unsigned**. This may change.

```
    return by_sku.count(sku) > 0;
}

static Product *lookup(const string &sku) {
    Product *p = by_sku[sku];
    if (p == nullptr) {  throw no_such_sku{sku}; }
    return p;
}
```

With MDS, by_sku is an mds_ptr<mds_namespace> rather than an unordered_-map, so we have

```
static bool exists(const string &sku) {
    return (*by_sku)[sku].is_bound();
}

static Product *lookup(const string &sku) {
    mds_ptr<Product> p = Product::lookup_in(by_sku, sku);
    if (p == nullptr) {  throw no_such_sku{sku}; }
    return p;
}
```

Working in Isolation

This brings us to the final part of the example: setting the next_product pointer and ensuring that the *Product* being created is the head of the product list. First, let's do it naively:

```
class Product : public mds_record {
    …
public:
    Product(const rc_token &tok, const string &s, float p, unsigned n)
    : …, next_product{ Product::lookup_in(data_ns, first_product_key }
    {
        THIS_RECORD->bind_in(data_ns, first_product_key);
        …
    }
    ...
};
```

This does what we want. next_product is set to the old first *Product*, and the first *Product* in the namespace is rebound to point to this record. But there's a problem.

The namespace is shared between threads (and processes), and if we just naively ran this code, there is the possibility that in between the read of the current first *Product* and setting the first *Product* to be this one, some other thread also created a *Product*. Let's say that the first *Product* we read was P1, and we're creating P2. What we'd like when we're done is for the first *Product* list to start with P2, which points to P1. But suppose after we read P1, somebody else created P3 and added it onto the list. So now, when we do the bind_in(dataNS, first_product_key) call, the *Product* list actually contains P3 pointing to P1, and we replace it with P2 pointing to P1. What happened to P3? It's still there, but nobody iterating over the list of *Product*s will ever see it.

So, what we need is a guarantee that the reading and writing constitute a single *atomic* action. To do this, we create a new *isolation context* and do the work in it. Within an isolation context,

1. All writes to fields of managed records and slots in managed arrays and all bindings of names in namespaces are *isolated* from code working outside of the isolation context. That is, if another thread reads the field or slot or does a lookup on the name, it won't see the effect of your write.
2. When you read or write such a location, the MDS runtime guarantees that the next time you read, what you see won't be affected by subsequent writes to the location from outside of the isolation context.
3. When you *publish* the isolation context (at the end of the **isolated** block), the system checks to see whether anybody did anything outside that would *conflict* with what you did such that it isn't possible to pretend that it all happened instantaneously.
4. If there were no such conflicts, all of your changes become visible outside the isolation context as a single atomic action, as of the time you published.
5. If there were any conflicts, the isolation context you were working in, and all of the changes you made are thrown away, a new isolation context is created, and the code is run again, looping until the publication succeeds.

There's a *lot* more to isolation contexts, but that's the simplest basic mode of using them to do short atomic transactions like the one we want here. One thing to note is that isolation contexts are hierarchical, so don't think of it as "publishing to the 'real' state of the record or namespace". We're simply publishing it to whatever isolation context the constructor schema is running in, which may itself be isolated from (some or all) other threads in the system.

With isolation contexts, we can say

```
class Product : public mds_record {
public:
    …
    DECLARE_FIELD(Product, Product, next_product);
    …
    Product(const rc_token &tok, const string &s, float p, unsigned n)
    : …
    {
        isolated([this]{
            next_product = Product::lookup_in(data_ns, first_product_key);
            THIS_RECORD->bind_in(data_ns, first_product_key);
        });
        …
    }
    …
};
```

In the scenario described above, we read P1, set it as P2's next_product, and set the first *Product* to be P2, making the *Product* list P2, P1, … But we do this in an

isolation context, so nobody else can see that the first *Product* is now P2. While we're doing this, another thread sets the first *Product* to P3, so the *Product* list now reads P3, P1, … When we try to publish, assuming that the other thread successfully published its isolation context, the system disallows the publication due to the conflict at the *"first_product"* name. The MDS runtime then runs the code again in a new isolation context, and this run retrieves the first *Product* as P3, sets that as P2's next_roduct, and sets the first *Product* to be P2, making the *Product* list P2, P3, P1, … We then attempt to publish, and since there are no conflicts, the publication succeeds, and this becomes the new visible *Product* list.

But note that in order to write the initialization in this way, we had to assign to next_-product, which is something that is disallowed if it is declared using **DECLARE_-CONST_FIELD**. So in order to make this work, we had to switch to using **DECLARE_FIELD**. This is important and hard to wrap your head around. It will be tempting to try to say something like

```
/*
 * The wrong way to do it!
 */
class Product : public mds_record {
public:
    …
    DECLARE_FIELD(Product, Product, next_product);
    …
    Product(const rc_token &tok, const string &s, float p, unsigned n)
    : next_product{ push_and_get_first() }
    {
        …
    }
private:
    mds_ptr<Product> push_and_get_first() {
        mds_ptr<Product> old;
        isolated([this, &old]{
          next_product = Product::lookup_in(data_ns, first_product_key);
          THIS_RECORD->bind_in(data_ns, first_product_key);
        });
        return old;
    }
};
```

And, indeed, this is the way we had the example in the first version of this tutorial. The problem is that while the isolation context makes the setting of the first *Product* atomic with respect to reading the value that will be used to set next_product, it isn't actually atomic with respect to setting next_product itself. This allows us to use a const field, but there's a window[10] during which the first *Product* has been set, but this *Product*'s next_product is still uninitialized and will be read as a null pointer. If another thread starts iterating during that window, they will see the null pointer in next_product and

[10] And if the program crashes, potentially a very long window.

will miss all the rest of the products. So remember that all of the side effects that have to be atomic need to be *inside* the **isolated** block.

Now a word about **isolated** itself. While we've colored it as a keyword in this example, **isolated** is actually just a template function takes any function and arguments (which are passed to the function). Canonically it will be a lambda expression, as shown here, but it could also be a function or an object that defines **operator()**(). If the argument functional returns a value, for the iteration that successfully publishes, will be the value of the **isolated** form.[11]

What is a Conflict?

In MDS, conflicts have to do with *locations*, where a location may be a field of a record, an array index, or a name binding. A location is conflicted if the following two properties hold:

- The first operation on the location within the child isolation context is a read operation, which reads the value from the enclosing context. "Read" here includes modifications which require obtaining the prior value, such as increment or get-and-set (exchange()). If the first thing you do at a location is an unconditional write, you cannot get a conflict there.
- The value at the location in the enclosing context *at the time we try to publish* is different from the one we first read in the child. Typically, this means that the value in the parent changed (either due to a direct modification or to a sibling context publishing into it) after we read but before we attempted to publish.[12] Note that the MDS runtime actually tracks conflicts as they occur, so by the time you attempt to publish, the runtime already knows what conflicts are there.

If you want to read the value of a location without potentially inducing a conflict, record fields and array slots provide a peek() method that (1) doesn't count as a read for conflicts and (2) doesn't cache the value (i.e., multiple peeks can return different values if the value in the parent context changes).

---

[11] Note that if tasks (described below) are re-run to resolve conflicts, the value returned does **not** change. (If the conflict's top-level task—the one that would return the value—depends on tasks that need to be re-run, the entire function will be re-run, producing a different value.) An **isolated** block that returns a value and uses subtasks should return a C++ object whose state is updated by those tasks. Alternatively, it can not return a value, but modify its enclosing environment.

[12] If the child isolation context is a snapshot (see below), a modification to the parent after the child was created constitutes a conflict even if it happens before the read in the child, because the child's read is as of the time it was created.

A Note on Memory Management

You might have noticed that we've told you how to create an MDS record, but we haven't told you how to get rid of them. It turns out that you don't have to worry about it (much). Everything in the MDS managed space is garbage collected. This means that as long as a record is *reachable* by any process (now or in the future) starting from either some pointer a process has or by a binding in the root namespace or one of its children and following through a chain of record fields, array elements, or namespace bindings, the record will stick around. As soon as this in no longer the case, the record becomes garbage, and the garbage collector, which works behind the scenes, will shortly reclaim the memory it was using to allocate other objects in the managed space.

The garbage collector used by MDS[13] is quite efficient and fault tolerant, so you don't have to worry about processes crashing. If a process dies while in the middle of an **isolated** block, the isolation context and all its modifications will become garbage.

So for the most part, when you're done with an *mds_ptr* to a record, you can simply drop it on the floor. There are two exceptions.

First, if you put a reference to a record in a namespace and that namespace is reachable from the root namespace, the record (and everything it refers to and everything those things refer to and …) will stick around until somebody cleans up the namespace. This is the downside of file-system-like namespaces whose job it is to keep references to things even when nobody is looking at them.

Second, while MDS uses true garbage collection on the managed records, the same is not true of the record class proxy objects. Internally, *mds_ptr* holds a *std::shared_ptr*, so the proxy objects are reference counted. This isn't usually a problem, but if a record class has non-field data members and these data members induce a cycle (e.g., record A refers to record B while B refers to A), the proxy objects will never be collected (until the process exits) and they will hold onto their managed records in the managed space.[14]

Using the Methods

Now that we've defined the *Product* class, let's look at the action methods that use it. Adding a product becomes

```
mds_ptr<Product> newProduct(String sku, float price, unsigned n_in_stock) {
    mds_ptr<Product> p;
    isolated([&] {
        if (Product::lookup(sku) != nullptr) { throw sku_exists{sku}; }
        p = make_mds<Product>(sku, price, n_in_stock);
    });
    return p;
}
```

---

[13] The *Multi-Process Garbage Collector* (MPGC).

[14] If you really feel you need to do this, note that *mds_ptr<T>* has an as_shared_ptr() method that returns a *std::shared_ptr<T>* and this can be used to get a *std::weak_ptr<T>*. The reverse requires using *mds_ptr<T>*::__from_shared(sp). This API is likely to change and be replaced by an actual *mds_weak_ptr<T>*.

There are two differences from the non-MDS code. First, rather than calling **new** *Product*(…), we call make_mds<*Product*>(…). Second, we can no longer count on nobody adding a new *Product* with this SKU in between the time we look and the time we create one, so we do the check and the creation in an **isolated** block. If another *Product* with this SKU is created after we check, it will result in a conflict when we attempt to publish, and we will throw an exception on the re-run of the block.[15] Note that by doing it this way, we create an isolation context to do the check and the creation, and the constructor schema creates a second isolation context within that one to add the *Product* to the *Product* list.

Getting stock in, checking the price, and establishing a new price are unchanged:

```
void stock_in(const string &sku, unsigned n) {
    Product::lookup(sku)->n_in_stock += n;
}

void new_price(const string &sku, float p) {
    Product::lookup(sku)->price = p;
}

float price_check(const string &sku) {
    return Product::lookup(sku)->price;
}
```

Note that modifying operations are defined to be atomic themselves, so we don't have to worry that when we're performing stock_in(), somebody changed the value of n_in_stock in between the time we read the old value and the time we wrote the new value.

And processing a sale is nearly unchanged, but wraps its action in an **isolation** block:

```
void process_sale(const vector<LineItem> &items) {
    isolated([&]{
        for (const LineItem &item : items) {
            auto p = Product::lookup(item.sku);
            p->check(item.quantity, item.price);
        }
        for (const LineItem &item : items) {
            auto p = Product::lookup(item.sku);
            p->sell(item.quantity, item.price);
        }
    });
}
```

Note that a single **isolated** block wraps both **for** loops. This means that in order for the publish of the isolation context to succeed, there need to be no conflicts that would invalidate either any of the checks or any of the modifications made during the sale of

[15] Had we done the check and the binding in the isolation block of the constructor, it would have been caught there and we wouldn't need this isolation block.

an item. It also means that if there were no such conflicts, all of the products sold are updated at the same time.

Working with Snapshots

Now we turn to our analytics routine, get_report(). Here we look at every product and generate a report that gives the total revenue, the total value of the stock on hand, and the top ten products by revenue. But we want to make sure that we give a consistent report—the information should all be correct as of a single point in time, and that point should be when we ask for the report. We could do this with an **isolated** block, as we did above, walking all of the products and computing the report and then trying to publish the isolation context, hoping that nobody had changed anything while we were computing it. But since we work for the company, we don't really hope that nothing had changed as we walked the inventory, since that would mean that we hadn't sold anything. So, we'll take a different approach. What we'll do instead is compute the report in a *snapshot*:

```
mds_ptr<Report> get_report() {
    vector<mds_ptr<Report>> all;
    float total_revenue = 0;
    float stock_value = 0;

    mds_ptr<Product> p = in_read_only_snapshot(Product::get_first_product);

    for (; p != nullptr; p = p->next_product) {
        total_revenue += p->revenue;
        stock_value += p->n_in_stock * p->price;
        all.push_back(p);
    }
    sort(all.begin(), all.end(),
          [](const auto &p1, const auto &p2) { return p2->price < p1-> price; });
    vector<Product *> top_ten{ all.begin(), all.size() > 10 ? all.begin()+10 : all.end() };

    return make_mds<Report>(total_revenue, stock_value, top_ten);
}
```

(To simplify the code, from now on, we'll assume that the *Product* clas has defined

```
static mds_ptr<Product> get_first_product() {
    return Product::lookup_in(data_ns, first_product_key);
}
```

and probably a similar method to set it.)

If you compare this with the non-MDS version, you'll see that only three things have changed. First, we're using *mds_ptr<Product>* instead of *Product **. And second, the initial definition of p is obtained via a call to **in_read_only_snapshot()**:

```
mds_ptr<Product> p = in_read_only_snapshot(Product::get_first_product);
```

in_read_only_snapshot is like **isolated** in that it executes its code in a new isolation context,[16] but this isolation context is different from the one created by **isolated** in three ways. First, it's *read-only*. That means that code running in that isolation context can't modify anything in the managed space.[17] If you try to do anything other than read, the program will throw a *read_only_context_ex*. Second, since the isolation context is read-only, you can't publish it. (This is not a big deal, since there can't be any modifications to publish.) This means that code is executed only once in **in_read_only_snapshot**. And, finally, the isolation context is a *snapshot*. This means that all values read from the managed space are read *as of the time the snapshot was created*. No matter how long you work in a snapshot, the values will never change.[18]

It might be surprising that we didn't wrap **in_read_only_snapshot** around all of the code in get_report(). That would have worked, and, in this case it would have had the same effect, but doing it the way we did allows us to explore another aspect of snapshots (and isolation contexts in general). The only code that we ran in the snapshot was the code that looked up the first *Product*. That value was returned as the value of the lambda and, therefore, the value of the **in_read_only_snapshot** call and was assigned to p. But that value was the first *Product in the snapshot*. When we ask for its revenue, n_in_stock, and price, we get the values that the product had in the snapshot. And when we ask for its next_product, we get another record in the same snapshot. So even though we returned from the **in_read_only_snapshot** call, the snapshot isolation context that was created sticks around as long as there are references to it. This makes values read in snapshots things that can be kept around for a long time, and it makes it possible to have lists of values from different snapshots that can be compared with one another, for example, to discover trends.

You may be wondering about the cost of taking a snapshot. It turns out that snapshots (especially read-only snapshots) are very cheap to create in the MDS system. Nothing actually gets copied. The only overhead is that as long as a snapshot exists, the MDS system has to preserve any historical values that somebody might read in it, so it has an effect on garbage collection.

---

[16] In this example, rather than write a lambda, since the code we want to call is a single function, *Product*::get_first_product, we simply pass in the function. It will be called in the new context, and its value will be returned.

[17] It can, of course, modify variables or objects in the process's own space.

[18] You can also get non-read-only snapshots, by calling **in_detached_snapshot**. Execution in a non-read-only snapshot can modify things, but anything it hasn't modified is seen as of the time the snapshot was created. For completeness, **detached** executes in a non-snapshot child, but any modifications made to the managed space are not published, so they are only accessible through the value returned or any other *mds_ptr*s squirrelled away by the computation.

Finally, the third difference is that the report object is created in the managed space (using make_mds<*Report*>(…)) rather than on the local heap. This allows us to store it away in a way that it can be used later. For instance, we can create one every day and store them in a namespace, keyed by date, or we could create a list of them and walk the list to find the first time a particular product cracked the top ten. Note that the creation of the report object takes place in the current isolation context, not the snapshot, even though the top ten products it will contain are in the snapshot.

Managed Arrays

To illustrate working with managed arrays, we'll go one step further in the example, and show what the *Report* might look like as a managed record:

```
class Report : public mds_record {
public:
  DECLARE_CONST_FIELD(Report, float, total_revenue);
  DECLARE_CONST_FIELD(Report, float, stock_value);
  DECLARE_CONST_FIELD(Report, mds_array<Product>, top_ten_products);
  RECORD_SETUP(Report, mds_record, "myco.report",
               REGISTER_FIELD(total_revenue),
               REGISTER_FIELD(stock_value),
               REGISTER_FIELD(top_ten_products));

  Report(const rc_token& tok, float r, float s,
               const vector<mds_ptr<Product>> &top10)
    : mds_record{tok}, total_revenue{r}, stock_value{s},
      top_ten_products{top10.begin(), top10.end()}
  {
  }
};
```

The interesting part here is working with the arrays:

```
class Report : public mds_record {
public:
  …
  DECLARE_CONST_FIELD(Report, mds_array<Product>, top_ten_products);
  ...
  Report(const rc_token& tok, float r, float s,
               const vector<mds_ptr<Product>> &top10)
    : …, top_ten_products{top10.begin(), top10.end()}
  {
  }
};
```

Managed arrays are fixed-size mutable arrays that live in the managed space. They are declared as *mds_ptr<mds_array<T>>* or its alias *mds_array_ptr<T>*. Currently only arrays of primitives (e.g., *mds_array*<**double**>) and records (e.g., *mds_array<Product>*) are supported. That is, arrays of arrays are not yet supported.

Managed arrays work with isolation contexts in the same way that managed records do. For example, in

```
mds_array_ptr<long> counts = …;
isolated([&]{
    counts[to_inc]++;
    counts[to_clear] = 0;
```

```
});
```

both of the changes will become visible atomically.

To create a managed array, use one of

```
make_mds_array<T>( n )
make_mds_array<T>( from, to )
```

The first form creates an array of the given size, with default-initialized values (e.g., zero, **false**, or **nullptr**), while the second form creates an array by copying a range given a pair of iterators. (The values must be of a type that is copyable to *T*). When initializing a managed array field of a record, you can either specify an array value or use arguments suitable by make_mds_array<T>(…), which will be called to create a new array. Note that no zero-length arrays are created. If one is requested, **nullptr** is returned.

Given the array defined above in the *Record* class, you can say things like

```
mds_array_ptr<Product> a = r->top_ten_products;
for (mds_ptr<Product> p : a) { … }

p = a[5];
a[7] = p;
a[0].exchange(p);

n = a.size();
if (a.empty()) { … }

ia = make_mds_array<int>(from, to);
ia[0] *= 10;
ia[5].fetch_div(2);
for (auto i : ia) {
  i++;
}
vector<int> v(ia.begin(), ia.end());
std::fill(ia.begin(), ia.end(), 0);
iter = std::find_if(ia.begin(), ia.end(), [](int i){ return i<0; });
if (iter != ia.end()) { … }
```

As can be seen from the examples, managed arrays behave like non-resizable standard containers (e.g., like *std::array*). They provide begin() and end() methods, so they can be passed in to most standard algorithms.

When using a range-for loop as in

```
for (auto i : ia) {
  i++;
}
```

the actual type of i will be *mds_array*<**int**>*::reference*, which can be used to modify the value. (This is the same value returned by the subscript operator.) This value coerces to the actual value, so you can also say

```
for (int i : ia) {
  cout << i << endl;
}
```

Managed arrays are typically accessed directly via the *mds_array_ptr<T>*. That is, you say a.size() and a[5] rather than a->size() or (*a)[5]. The latter forms will also work, but they will cause the program to crash if the array is empty, because the underlying pointer will be null. The specialization of *mds_ptr* for *mds_array* guarantees that the right thing happens with null pointers and empty arrays.

Note that arrays are bounds checked. Any attempt to index outside of their range will result in a *std::out_of_range* exception being thrown.

As with record fields, numeric modifiers such as a[k]++ and a[k].fetch_add(2) are atomic. This, of course, only holds for a single modification. If multiple modifications need to be atomic, the code should be wrapped in an **isolated** block.

Resolving Conflicts

Now suppose that we're told that every product has to have a score associated with it. The score is a floating point number, and you need to write a function that will be passed a function for computing the score for a given product and will update the scores of all the products. The updates have to be atomic. That is, none of the products will have updated scores until they all do.

Adding the score itself is straightforward:

```
class Product : public mds_record {
    …
    DECLARE_FIELD(Product, float, score);
    …
}
```

And writing the function and making it atomic doesn't require anything we haven't seen before:

```
template<typename Fn>
void computeScores(const Fn &score_fn) {
    isolated([&]{
        for (auto p = Product::get_first_product(); p != nullptr; p = p->next_product) {
            p->score = score_fn(p);
        }
    });
}
```

We create a new isolation context, look up the first product, and walk the products. For each one, we call the provided function to get the score and set the score on the product. And then at the end, we try to publishe the changes in the isolation context. If there are no conflicts—if nobody else changed any value that the score function read—all of the scores are updated atomically. If there were conflicts, we go back to the beginning and try again.

But therein lies the problem. We're touching *every* product, and the score computation could take a non-trivial amount of time. Unless we take care to only run this at a time when nobody is selling anything, there's a very good chance that by the time we get to the end, somebody will have sold something and changed one of the products, leading to a conflict. And if we go back and try again, we're just as unlikely to get through conflict-free. So, what can we do?

Enter *tasks*. Tasks are bits of code that could be re-run if conflicts are found. When an attempt is made to publish an isolation context and conflicts are found, the system looks to see if the conflicts can be *resolved* by re-running some of the tasks that were run during the computation. The basic rule for deciding which tasks to re-run are:

1. If a task read some value (e.g., a field of a record object, a name in a namespace, or an element of an array) that had been written outside of the task's isolation context and that was later changed outside of the task's isolation context, the task is selected to be re-run.
    a. Note that if the task's isolation context is a snapshot isolation context, the notion of "later" includes "after the snapshot time but before the read occurred".
2. If a task read a value that was written by a task that was selected to be re-run, the reading task also is selected to be re-run.

There are a few other cases in which a task might be selected to be re-run, but that's the main idea. For other tasks, the presumption is that if they were re-run they would behave the way they did the first, so there's no point in actually re-running them. The goal is that if the program is partitioned into tasks, the system can chose a subset of the tasks to re-run, and this subset can be substantially smaller than the complete job.

To add tasks to our code, we simply call the work within an **as_task** function:

```
template<typename Fn>
void computeScores(const Fn &score_fn) {
    isolated([&]{
        for (auto p = Product::get_first_product(); p != nullptr; p = p->next_product) {
            as_task([&score_fn, p]) {
                p->score = score_fn(p);
            });
        }
    });
}
```

Unlike **isolated**, **as_task** only takes a single functional argument (i.e., no parameters to pass to the functional) never returns a value.[19]

[19] The logic for this restriction is that if the task needs to be re-run during conflict resolution, it might produce a different result, but the code that used the original value won't be re-run. To handle the situation in which a task should compute a value that will be used elsewhere, use

Now, when we complete iterating over our thousands of products and try to publish and the attempt fails because, say, twenty of them changed their values due to sales or price changes, only the twenty tasks that worked on those products are re-run. Re-running the tasks means re-running the functions, and that re-running takes place in the same, not-yet-published isolation context. But before any tasks are re-run, any writes made by taks to be re-run are *rolled forward*. This leaves the locations written with the value last written by a task not selected to be re-run or, if there were no such writes, to the current value in the parent isolation context. The result of this is that that when tasks are re-run, what they see is what the world would have looked like had all non-selected tasks been re-run at the time of the (failed) publication attempt. When the identified tasks have been re-run, the publication is attempted again. At this point, there may be new conflicts, but since the work took less time, there will probably be fewer of them and, therefore, fewer tasks that need to be re-run. (Note that the tasks that need to be re-run the second time might include some that didn't need to be re-run the first time.) The expectation is that after a small number of iterations, the amount of re-work needed should be small enough that there's a good chance of being able to get through without conflicts.

When tasks are re-run, the order in which this happens depends on the order in which the tasks ran initially. In particular, tasks which started first are re-run first.[20]

A Warning about Variable Capture and Tasks

You may have wondered why we used a strange capture expression, [&score_fn, p] for the lambda we passed in to **as_task**. Why didn't we simply say

```
/** Don't Do This! **/
template<typename Fn>
void computeScores(const Fn &score_fn) {
    isolated([&]{
        for (auto p = Product::get_first_product(); p != nullptr; p = p->next_product) {
            as_task([&]) {
                p->score = score_fn(p);
            });
        }
    });
}
```

and capture p as well as score_fn by reference? Let's walk through and see why this is a bad idea. First we enter the **isolated** block and run its passed function, which captures score_fn by reference. Then we go through the **for** loop, whichin the first iteration puts p on the stack. We call **as_task**, passing in a lambda that captures both

---

task::computed_val<T> objects and the *task::computed*(fn) function, which will be described below.

[20] In the current MDS library, task re-run happens in a single thread. This clearly needs to change, as it may lead to cases in which task A was selected to be re-run because it read a value written by task B, but the two tasks ran in parallel and task A happened to start before task B and will consequently be re-run in its entirety before task B.

p and score_fn by reference. This lambda is called, and it returns. p is logically removed from the stack, and the process repeats for the remaining *Product*s, with each lambda capturing a reference to where p was on the stack when its as_task was called.

Now we get to the end of the **isolated** block, and its function returns. All values of p are gone, but the functions associated with the various tasks still refer to where they were. score_fn, which is outside of the **isolated** block is still there. Now we decide that, say, the third task needs to be re-run. It's function is called, and it tries to read from the stack where it thinks that p is, reading garbage. Hilarity (or at least catastrophe) ensues.

So the rule for lambda capture is

- When passing a lambda to **as_task**, any variable declared *within* an enclosing **isolated** block's function ***must*** be captured by value.
- A variable declared *outside* of the enclosing **isolated** block's function may be captured by reference or by value. (Both make sense in different circumstances).
- A variable capture by an **isolated** block's function will typically be captured by reference.

Conflicts Outside of (Explicit) Tasks

In the code we just wrote, not all of the work in the isolation context took place within a task we explicitly created. In particular, looking up the first product and finding the next product for each product takes place outside of explicit tasks. In our example system, the next product will never change, but it's definitely possible that somebody might have added a product, which will result in the first product changing. This will result in conflict on the initial read of the first product from the namespace. While the read is in a task (there is always a prevailing task), the task isn't associated with a function that the MDS library can re-run, so the only alternative the system has is to re-run the entire isolation context work. The MDS team will be looking at ways of minimizing such situations.

Working with Arrays and Tasks

One of the most common uses of tasks is to iterate over collections (e.g., *mds_array*, *vector*, *set*) and performing an operation on each element in a task. To facilitate this, MDS provides for_each_in_tasks(fn).

In our example, if the *Products* had been stored in *mds_array<Product>* rather than a linked list, we could have simply said

```
template<typename Fn>
void computeScores(const Fn &score_fn) {
    isolated([&]{
        mds_array_ptr<Product> products = Product::products();
        for_each_in_tasks(products.begin(), products.end(),
            [&](auto &p) { p->score =  score_fn(p); });
    });
}
```

for_each_in_tasks(from, to, fn) is actually just a wrapper for

```
std::for_each(from, to, task::task_fn(fn))
```

where *task*::task_fn(fn) takes a function and returns a new function that accepts the same arguments and invokes the provided function in a new task, capturing any arguments by value,[21]

Computing Values in Tasks

At times, you may want to compute a value in a task such that whenever the task needs to be re-run, any task that read the computed value will also be selected to be re-run. To do this, you can use *task*::computed<*T*>(fn). This creates a new task as a child of the current task and computes the value based on the provided function, but what it returns is a *task::computed_val<T>* object. This object provides a get() method that returns a **const** *T* & (and a corresponding conversion operator) and guarantees that until the current isolation context is published, any task calling the get() method will depend on the task that computed the value.

The function provided to *task*::computed<*T*>() is a function that can accept a *T* and returns a value that can be converted to a *T*. The first time it is called, the value will be (a reference to a) *T*{}. On subsequent re-runs for conflict resolution, the value will be a reference to the value returned the prior time.[22] This allows the computed value to undo local state from the prior computation before recomputing.

Note that a lambda provided to *task*::computed<*T*>() must follow the same capture rules as lambdas provided to as_task().

Managing Tasks

In addition to what's been described above, there are a number of other things you can do with tasks:

```
task t = task::current();
t.depends_on(task1, task2, …);
t.depends_on_all(collection);
t.cannot_redo();
t.always_redo();
t.on_prepare_for_redo(fn);
```

current() gets the prevailing task for the current thread. depends_on(task, …) says that if any of the other tasks are selected to be re-run, this task should be as well. depends_on_all(collection) does the same, but with all of the tasks in the given collection. cannot_redo() says that this task cannot be re-run. If it needs to be re-run, the task that spawned it will be selected instead. always_redo(), conversely says that if *any* tasks in this isolation context are selected to be re-run, this one should be as well.

---

[21] Note that in this example, we can simply use [&] as our capture expression because the only variable captured by the lambda is score_fn, which is outside the **isolated** block.

[22] The intent was to also be able to pass a function that took no arguments (essentially ignoring the prior value), as this is what is desired most of the time, but we were unable to figure out a way to get it past the compiler. Similarly, it would be reasonable for compute() to infer *T*,

It is mostly to be used when the task depends on something external to the MDS library, such as a sensor reading.

on_prepare_for_redo(fn) supplies a function to be run when the task has been selected to be re-run. A task can have any number of such functions, which are run in the order they are added, with the function for different tasks run in the order the tasks will be re-run. The functions are all called *before* any locations are rolled forward. Each function can optionally take the task as a parameter and can optionally return a **bool** value. If it returns a **false** value, the entire conflict resolution is abandoned. This facility intended to be used to clean up any local data structures that might have been modified during the computation.

*Controlling **isolated** Blocks*

In addition to taking a function to run in the new context, the **isolated** function can also take several arguments that control how conflict resolution and function re-run work. Because the function passed in is often a lambda, in order to make the code lay out better, these optional arguments come *before* the function (and any arguments to it). They can come in any order.

Some of the arguments control what time of child context is created. These include enums of type *iso_ctxt::mod_type*, which say to what extent the new context will be publishable or modifiable and enums of type *iso_ctxt::view_type*, which say whether the new context is a snapshot or a normal ("live") context. The options are

- *iso_ctxt::mod_type*::**publishable**. The function will be called in the new isolation context and then an attempt will be made to publish the context. If this fails, conflict resolution will be attempted. This is the default.
- *iso_ctxt::mod_type*::**detached**. The function will be called in the new isolation context, but no attempt will be made to publish the context. In addition, conflicts will not be detected. If you don't intend to publish the side-effects of the computation, this is more efficient.
- *iso_ctxt::mod_type*::**read_only**. The function will be called in the new context, and modifications to the managed space are not allowed. Consequently, publishing will not be attempted. Note that a read-only isolation context can have detached children.
- *iso_ctxt::view_type*::**live**. Reads made in the function will read the latest value from the enclosing isolation context at all locations (fields, array slots, name bindings) that haven't been modified in the child. This is the default.
- *iso_ctxt::view_type*::**snapshot**. Reads made in the function will read the value from the enclosing isolation context *as of the time the child was created*. If a location is modified in the child, the latest value in the child will be used.

Other arguments control how long to keep attempting conflict resolution and how long to keep re-running the main **isolated** function before giving up and throwing *publish_failed*.

- resolve(cond, …) says to continue trying to re-try tasks to resolve conflicts until any of the conditions say to stop. There can be multiple resolve() options.
- rerun(cond, …) says to continue re-running the **isolated** function until any of the conditions say to stop. Note that such re-running only happens when conflict resolution fails (or is told to stop).

The conditions include

- max_tries(n) says to re-try or re-run at most n times. The literal _times (e.g., 5_times) is a shortcut for this.
- try_for(duration) says to try for the given duration (e.g., try_for(5min) means to try for five minutes). For conflict resolution, the clock is reset after each re-run of the main function.
- try_until(timepoint) says to try until the given wall-clock time.
- try_while(fn) says to try as long as the given function returns true.

Finally, an argument of report_to(report, …) allows you to pass in one or more *publish_report*s (by *shared_ptr*), which can be used to monitor the progress of the **isolated** block. A *publish_report* contains the following virtual functions, which can be overridden:

- reset() is called before the main function is called for the first time. If overridden, this should delegate to its parent.
- before_run(*iso_ctxt* &) is called before each run (first or re-run) of the main function. The argument is the new child context that will be used.
- before_resolve(**const** *pub_result* &) is called before each conflict resolution attempt. The argument can be used to obtain the set of tasks that will be re-run.
- note_success() is called when the publish is successful. It should delegate to its parent.
- note_failure() is called when the **isolated** block gives up. It should delegate to its parent.

Working with Accumulators

In the *Report* example above, we used a snapshot to compute (among other things) the total revenue and stock value:

```
mds_ptr<Product> p = in_read_only_snapshot(Product::get_first_product);

for (; p != nullptr; p = p->next_product) {
    total_revenue += p->revenue;
    stock_value += p->n_in_stock * p->price;
    …
}
```

This gives us a consistent value, but it's the value as of the time the snapshot was taken (i.e., the beginning of the computation). What if we had some *Store* object and wanted to update the revenue and stock totals atomically, so that the values would not only be consistent with one another but also be correct as of the time they are set.

We could try doing the computation within a snapshot **isolated** block:

```
isolated(iso_ctxt::view_type::snapshot, [&] {
    store->total_revenue = 0;
    store->stock_value = 0;
    for (auto p = Product::get_first_product(); p != nullptr; p = p->next_product) {
        store->total_revenue += p->revenue;
        store->stock_value += p->n_in_stock() * p->price();
    }
});
```

This would work, but if any products changed revenue, stock totals, or price as we were computing, we would have to do the whole thing again.

We could use tasks to limit the amount of rework that needed to be done:

```
isolated(iso_ctxt::view_type::snapshot, [&] {
    store->total_revenue = 0;
    store->stock_value = 0;
    for (auto p = Product::get_first_product(); p != nullptr; p = p->next_product) {
        as_task([=]{
           store->total_revenue += p->revenue;
           store->stock_value += p->n_in_stock() * p->price();
        });
    }
});
```

At first glance, this looks as though it should do what we want. Only tasks that depend on values that have been changed will be re-run. But those increments are problematic. Each one is, logically, a read followed by a write. So, if the 20th product's revenue changes, its task will need to be re-run. But its increment wrote a value that was read by the 21st task, whose increment was read by the 22nd task, and so on, so all tasks after the first with a change need to be re-run.

The way to deal with such situations is with accumulator. An accumulator is a process-local object that you can add to and read from, but which has the following properties:

- A task that reads from an accumulator is dependent on all tasks that added to it (as of the time it read).
- Tasks that add to the accumulator are not dependent on one another.

If a task that adds needs to be re-run, its original contribution is removed from the accumulator before doing so.

With accumulators, the code looks like

```
isolated(iso_ctxt::view_type::snapshot, [&] {
    accumulator<double> total_revenue;
    accumulator<double> stock_value;
    for (auto p = Product::get_first_product(); p != nullptr; p = p->next_product) {
        as_task([=]() mutable {
           total_revenue += p->revenue;
           stock_value += p->n_in_stock() * p->price();
        });
```

```
    }
    as_task([=] {
        store->total_revenue = total_revenue;
        store->stock_value = stock_value;
    });
});
```

Now, if there are n products and k of them change during execution, only those k tasks, plus the one at the end that sets the values, need to be re-run.

Note that the **mutable** keyword is necessary on the lambda that adds to the accumulators so that the value-captured accumulators can be modified. By default variables captured by value are **const**. By specifying **mutable**, they become non-**const**.

SummaryStatistics

One useful accumulator is the *summary_statistics* class. In addition to the **+=** operator, there's an add() method that takes a value and an integral weight::

```
add(double val, size_t weight = 1)
```

It has a number of questions that can be asked: [23]

```
size_t count()          // The sum of the weights added
double sum()            // The weighted sum of the values
double mean()           // The mean (weighted) value
double sum_sq()         // The sum of the (weighted) squares of the values
double mean_sq()        // The mean of the (weighted) squares of the values
double rms()            // The RMS (root mean square) of the weighted values
double variance()       // The variance of the values
double std_dev()        // The standard deviation of the values
double std_err()        // The standard error of the values
double high_95()        // The high end of the 95% confidence interval around the mean
double low_95()         // The low end of the 95% confidence interval around the mean
```

The last two are the upper and lower limits of the 95% confidence interval around the mean.

Preparing to Run

There are a few other bits of bookkeeping that you should also know about.

Dealing with Record Inheritance

As mentioned above, record types can form an inheritance hierarchy, with, e.g., a *SoftwareProduct* record deriving from the *Product* record. In addition to extra fields, *SoftwareProduct* can add instance methods or override virtual instance methods of *Product*. If the program has a reference to a record that is actually a software product record, but the reference considers it to be a *Product*, the overridden method will be called.

But there is a wrinkle: *if the runtime doesn't know about the subtype yet, the supertype will be used*. That is, if nothing has caused the system to become aware of

---

[23] I haven't yet figured out how to efficiently do max() and min() with accumulators, but I hope to.

SoftwareProduct, what will be constructed will be an instance of *Product*, and it won't have the *SoftwareProduct* specializations. So, what lets the system know about the subtype? Creating an instance of the type (or a type derived from it) is sufficient, but if you're not sure whether this will happen before you may see a record of that type as a value in a record, array, or namespace, you can call *mds_record*::force<*R,…*>(), as in

```
mds_record::force<SoftwareProduct, HardwareProduct, SomeOtherRecord>().
```

This will ensure that these record classes are known to the system and that managed records of those types will be properly identified and the correct proxies created.

Working with Threads

When working with multiple threads, a little bookkeeping needs to be done in order to ensure that a new thread executes in the correct task. There are a few ways to accomplish this.

The simplest is to use *mds_thread* in place of *std::thread*. This is a subclass of *std::thread* but it ensures that the new thread is running in the same task (and has the same current namespace) as the thread that created it. You use it just as you would *std::thread*:

```
mds_thread th(fn, args, …);
```

An alternative, which may be useful when working with a pool of threads is to wrap the function in *task*::bind_to_current, as in

```
future<unsigned> f = async(task::bind_to_current(compute_fn));
```

task::bind_to_current() returns a function that runs the given function but first establishes that the current task (as of the call to bind_to_current()) is established in the thread.[24]

Finally, if you don't care about the current isolation context and task, you can simply enure that your function calls ensure_thread_initialized() before it does pretty much anything else. This will result in the function being run in the top-level task of the global context.[25]

If you don't use one of these methods, your program will probably crash with an uncaught *thread_base_task_unset_ex*.

---

[24] To bind to an arbitrary task use, t.bind(fn).

[25] Although it's possible that in weird circumstances if you're running from a thread pool, you might wind up in the last task that was used. Maybe.

Building and Running MDS Programs

Building and running MDS programs is straightforward. MDS programs can be built using a development environment such as Eclipse or from the command line.

Things you will need:

- 64-bit Linux
- g++ 4.9.2 or 5.4.0 or later.
  - Note, MDS is known *not* to work with g++ 5.0
- pthreads
- MDS and MPGC libs and include files

Building MDS programs

Assuming that MDS and MPGC have been installed to a directory called install, you will want to compile your code with the following flags:[26]

```
–I<install> –std=c++14 –Wall –fmessage-length=0 –pthread –fPIC –mcx16
```

The resulting .o files should be linked against the following libraries:

- From install/lib:
  - libmds-cpp.a
  - libmds_core.a
  - libmpgc.a
  - libruts.a
- –lstdc++
- –lpthread

Note that MDS compilations can take a fair bit of time (it relies heavily on templates), so if you are compiling multiple source files using make, it is highly recommended to use the –j option to allow for parallel compilations.

Creating the Managed Space

Before running your program, you need to create the shared "managed space" for the MDS data structures to be created in. This is done by running the MPGC **createheap** program, e.g.,

```
<path to install>/bin/createheap 10GB
```

The size of the heap is specified as a number followed by an optional case-insensitive unit, one of KB, MB, GB, or TB (the B is optional). The number may include a decimal point, so, e.g., 1.5 TB is valid. If no unit is specified, GB is assumed. Units are binary multipliers (i.e., 1.5TB is 1.5×2^{40} bytes).

This program will create two files, by default in a directory called **heaps**. The files are

[26] At least, these are the ones we use.

- **gc_heap**: the MPGC garbage-collected heap
- **managed_heap**: a "control heap" for MPGC.

The size of the control heap is computed automatically but can be specified explicitly by the **-s** (or **--ctrl-size**) argument, which takes as a parameter the size of the control heap. This will not typically need to be changed, but may need to be raised in some circumstances.

The locations of the heap files themselves can be specified by **-f** (**--heap-path**) for the managed heap and **-c** (**--ctrl-path**) for the control heap. Note that these are the full paths to the files, not merely the names within the heaps directory.

Specifying the Heap Files

The MDS runtime (actually, the underlying MPGC garbage collector) uses three optional environment variables to specify where to look to find the heap files:

- **MPGC_HEAPS_DIR** specifies the directory to use to look for the files. Defaults to heaps.
- **MPGC_GC_HEAP** specifies the MPGC heap. Defaults to gc_heap in $(MPGC_HEAPS_DR)
- **MPGC_CONTROL_HEAP** specifies the MPGC control heap. Defaults to managed_heap in $(MPGC_HEAPS_DIR).

If the two heap files cannot be found (i.e., if createheap was not run or the environment variables do not point to the files), an error message is printed and the program terminates.

Running Your MDS Program

With the heap files created, your program can simply be run from the command line.