10 Problems with C++

(and possibly their solutions)

The C++ Computer language is extremely powerful. However, it is not without its flaws. Certain things about it are less than ideal due to its history. Unfortunately, most of these things are unlikely to be fixed due to backwards compatibility requirements. If we could though, what are the things that could be changed for the better? To improve performance, readability, and orthogonality of the language?

1. ### New Keyword

   The first thing that comes to mind is memory allocation. The current method, of using the `new` keyword is rather broken. In C, the way to dynamically allocate memory is via the `malloc()` function. Since it is a function, it is possible to override via a macro for debugging or profiling purposes:

   ```
   #define malloc(X) my_malloc(X)
   ```

   Unfortunately, the `new` keyword doesn't allow such an override. (You can use a placement new, but the syntax prevents a macro from working.)

   A second problem with the `new` keyword is that it combines two different operations in one. The first is to allocate memory, the second is to initialize it. This combination means that initializing an array is impossible to do with anything other than the default constructor.

   Similarly, destructing an array with the `delete[]` operator is inefficient. The memory allocator needs to know how large the memory region that contains the array is, so it stores the size internally somehow. The `delete[]` operator needs to know how many objects to destruct, so it also stores the array size. Finally, the user code typically needs to iterate over the array, so inevitably it also stores the array size somewhere else. This is needless duplication.

   If all backwards compatibility were thrown out, how could construction, initialization, and memory allocation be better done, (paying particular attention to orthogonality)?

   Currently, a static initialization looks like this in C++:

   ```
   class_type x(args);
   ```

   Where `class_type` is the type of some object we want to create. Note how we may pass an optional set of arguments to the constructor. However, if the arguments are not required, then the syntax is different:

   ```
   class_type x;
   ```

   not

   ```
   class_type x();
   ```

   as the second example actually declares a function called `x` returning an object of type `class_type`. This is a non-orthogonality of the syntax. It, however, can be fixed by a slight change to how C++ initializers work. Imagine a slightly different syntax:

   ```
   class_type x = class_type(x, args);
   ```

   Where the constructor is explicitly called with a reference to the object about to be built. The bad part of this syntax is that it requires slightly more typing. However, notice how if the constructor requires no arguments no syntax exception is required. Also notice how we do not need to use a constructor with the same name as the class. Any other function that takes a reference + arguments and returns that same reference will do. Without editing the source code for `class_type` it is possible to hook the creation of objects of that type:

   ```
   #define class_type(X, ...) hooked_class_type(X, __VA_ARGS__)
   ```

   where we assume C++ gains the C99 variable argument macro extension.

   So what about the `new` keyword? The current syntax looks like:

   ```
   class_type *x = new class_type(args);
   ```

   This could be modified to be

   ```
   class_type *x = class_type(new(class_type), args);
   ```

   where now, the `new` keyword is a simple macro that returns allocated memory of the size of `class_type`

```
#define new(T) ((T *) malloc(sizeof(T)))
```

Again, this is more typing, but the orthogonality gains are enormous. This just becomes a special case of initialization. Since an explicit reference parameter to the object is visible, we can use any allocator we like to create it. Overriding the new macro with another one for memory debugging is trivial if the syntax is like this. In effect, all `new` operations become a placement new.

Finally, how would arrays be allocated and constructed? At the moment, arrays must be allocated with the default constructor:

```
class_type x[n];
```

With explicit construction this would become the rather more wordy, but much more flexible:

```
class_type x[n];
for (int i = 0; i < n; i++)
{
        class_type(x[i], args);
}
```

or even

```
class_type x[n];
foreach(class_type &xi : x) class_type(xi, args);
```

using the new proposed extension to the `for` keyword in C++0X. Obviously, the constructors in such an arrangement can depend on the array index, `i`. This also could be done via placement new with current syntax, if care is taken to correctly destruct the default objects before constructing new ones on top. The resulting code would be quite a large hack though.

This leaves arrays allocated dynamically. Currently this is done via the `new[]` operator:

```
classtype *x = new class_type[n];
```

This could change into the alternate syntax

```
classtype *x = new(class_type, n);
foreach(class_type &xi : x) class_type(xi, args);
```

where a second overloaded macro for new is required:

```
#define new(T, N) ((T *) malloc(sizeof(T) * (N)))
```

Objection deletion can then be done in a similar manner.

```
delete x;
```

becomes the more explicit

```
~class_type(x);
free(x);
```

and

```
delete[] x;
```

becomes

```
for (int i = 0; i < n; i++) ~class_type(x[i]);
free(x);
```

The result is rather wordy in places. However, since under the hood only one method of memory allocation, and one method of memory freeing is used (instead of three), it is much easier for generic code to exist. Finally, with this technique of explicit construction, it is also possible for client code to use the `realloc()` function to dynamically alter array sizes. The result is much more efficient container libraries.

As an aside, with the template syntax given below, it is also possible to replace the `new` macro with a template. (Current templates do not work, as greater-than and less-than signs are not seen by the pre-processor as delineating a "function"

2. Exceptions

The second problem with C++ are exceptions. It is well known that the `goto` instruction is bad for structured programming. It breaks control flow, and can make the program structure very hard to understand if misused. Exceptions are like a hypothetical `comefrom` statement. When you catch an exception, you don't care how you've gotten to your exception handler, just that you have. Thus exceptions, if misused can just as bad as `goto` statements, if not worse for obscuring control flow.

Why were exceptions added to C++? The reason is that constructors cannot return a value. This means that if a constructor runs into a problem, such as running out of memory, there is little it can do. It needs some way of communicating its problem to the calling code. If the constructor takes an argument that is a pointer or reference to some "state" variable it can communicate by changing the state. i.e.

```
class_type(bool &failed)
{
        if (!initialize_me(this))
```

```
        {
                failed = true;
        }
        else
        {
                failed = false;
        }
}

bool failed;
class_type x(failed);

if (failed) //Houston we have a problem...
```

The problem is that arrays are constructed with the default constructor, and the default constructor takes no arguments. Thus the above trick doesn't work. A way around it is to use a global variable, but that isn't thread safe. A thread-local variable is another possibility, but those are not standardized until C++0X is released.

With explicit construction, the zero-argument count, and no-return-value requirements for a constructor evaporate. Thus the requirement for using an exception as a communication mechanism disappears. Note that exceptions roll back creation of objects by calling their destructors. Thus a destructor cannot create a temporary object, as the creation may fail. The failure would cause a second exception to be raised, terminating the program. If constructors have alternate methods of communicating failure, then destructors can be made more robust.

Also notice that although an exception can roll-back the creation of objects, it cannot roll-back time itself. If an object is created, it may communicate to another thread, send a message along a socket, or make a visible write to a file that is observable by other programs. The universe has state, and exceptions formally only work in a stateless system.

The only advantage exceptions have is that they can be much more efficient than explicit checks for problems detectable by hardware. i.e. There is no need to check your pointers for NULL if you catch the segmentation fault. If the exceptional event occurs rarely enough, then the lack of checking can be a performance win even if the exception stack-walking code path is slow.

Lets pretend we are willing to ignore all the problems, and are willing to spend the effort to write exception-safe code. The final problem is that it is currently impossible to prove that any non-trivial code is exception safe. Without proof that all exceptions are handled, it is extremely likely that latent bugs exist that can cause termination of the program. There needs to be some way to tell the compiler enough information that it can compare your expectations of the allowed exceptions in a given code fragment verses what it sees possible, and enforce equality between them at compile time.

Unfortunately, the current system, whilst allowing the a list of exceptions to be generated as an annotation to a function definition, does not enforce the existence of such lists. Also, such exception specifications are checked at run-time rather than compile time, making them rather useless for their intended purpose. These faults mean that exception specifications are rarely used in real-world code.

The new C++ standard potentially will have the `noexcept` keyword which will identify functions and blocks that cannot raise exceptions. If implemented, this will be a great help. Unfortunately, due to the poor design of dynamic allocation and the existence of `std::bad_alloc` non-trivial code will still have difficulty proving exception safety.

The real fix is to remove most of the situations that can raise exceptions, making them truly rare events. Once that is done, the burden of including compulsory exception specifications for the functions that use them would not be so large. If the compiler could check such specification lists at compile time, then it should be possible to make firm statements on the exception safeness of code.

3. Implementation of Multiple Inheritance

Implementing multiple inheritance in a compiler is a challenge. This is due to the fact that the base classes cannot overlap in memory. If only single inheritance were allowed, then the compiler could place the base class at the top of the class memory footprint. Casting from the derived class to the base class would then be a no-op. Multiple inheritance prevents this, and requires casts to alter the value of the object pointer by the offset of the base class we want to access. This pointer-shifting is typically done inline for speed.

Thus when a derived class wants to call a function defined in a base class when multiple inheritance exists, it must find the amount to shift its `*this` pointer before the call. When virtual class inheritance is used, this shift must be obtained at run-time. Most compilers obtain the shift by storing it in the vtable for the class. Thus written out explicitly, a function inherited from a virtual base class is called in a low-level way like:

```
offset = this->vtable[OFFSET_LOCATION];
base_this = &((char*) this + offset);
base_function(base_this, args);
```

Where the hidden `*this` parameter to the base class's member function has been written out explicitly.

If the function to be called is a virtual function, then it will need to be read from the vtable as well. The resulting low-level code would look something like:

```
offset = this->vtable[OFFSET_LOCATION];
base_this = &((char*) this + offset);
base_function = this->vtable[BASE_FUNC_LOCATION];
base_function(base_this, args);
```

The problem here is that the offsetting is repeated throughout the compiled code. A simple optimization which will also be extremely useful later is to defer the offset calculation of `*this` into a thunk. The advantage here is that the many additions scattered throughout the compiled code will be compressed into one location, thus saving quite a bit of space. The calling convention would then look something like

```
base_function_thunk(this, args);
```

or for a virtual function,

```
base_function_thunk = this->vtable[BASE_FUNC_LOCATION];
base_function_thunk(this, args);
```

where the thunk would look like:

```
offset = this->vtable[OFFSET_LOCATION];
base_this = &((char*) this + offset);
return base_function(base_this, args);
```

The cost of adding a thunk is less than you might think. The advantage here is that the thunk doesn't need to be called if the base offset is zero. Thus the function pointer hidden in the vtable can point directly to the member function of the base class. This saves the cost of the addition nearly all the time since multiple inheritance with more than two base classes is rare. Another trick is to notice that most of the time, only one such thunk exists, or that one offset is much more popular than the others. By placing the thunk in memory just before the function to call, the extra jump may be removed. The result is a coding size win, as many offset additions may be converted into just one.

4. Member Pointers and Member Function Pointers

The real advantage of using the thunks in the implementation of multiple inheritance is that we can now change the implementation of member pointers and member function pointers. The problem with member pointers and member function pointers is that they are not the same size as a `void *` in nearly all compilers. (The Digital Mars compiler is the notable exception.)

Having member pointers larger than normal pointers is explicitly allowed by the C++ standard as it stands today. However, it is possible as I will show to shrink them down to the size of a normal pointer. (Note that function pointers in C and C++ are also allowed to be larger than normal pointers. However, in most sane architectures function pointers and data pointers are actually the same size. We will ignore the insane itanium case.)

The reason member function pointers are so large is that they need to store information to obtain the function they want to call at run time, even if the class they are from is virtual, and even if they themselves are virtual. This means that the data for `OFFSET_LOCATION`, `BASE_FUNC_LOCATION` and/or the function location depending on whether or not the class has a vtable, somehow needs to be stored within the pointer. Since there is alot of information here, it obviously requires more space than a normal pointer to store. Some compilers take up to four times as much memory to store a member function pointer than to store a normal pointer. Most other compilers take twice as much memory, using alignment tricks to detect whether a vtable access is required or not.

To shrink the member function pointer size, we just need to implement another type of thunk. This thunk will parameterize over the function number within the vtable. Thus we have something like:

```
return this->vtable[n](this, args);
```

where a separate thunk for each possible offset `n` is required. Since most programs have classes with less than a few thousand member functions, less than a few kilobytes of thunks are required. Note how the thunk's low-level code actually will not care about the exact type of the object so long as its vtable is in the "correct" location. The calling conventions for object member functions also allow transparent lookup of the vtable without affecting the arguments. (We can use %eax or %rax for the calculation of which function to call without affecting the stack or other registers.)

Note how the above depends on the fact that the vtable references functions of the same type as the caller. Thus if we are calling into a base class, we will need to somehow include the offset calculation. This is magic of the previous section. The thunks there will fix up that for us. Thus there are four cases which can happen:

1. Compiler is able to determine at compile time which member function to call: The member function pointer is a normal function pointer.
2. Compiler needs to do a vtable lookup: Use a function pointer pointing to the vtable thunk_n parameterized by the function offset.
3. Compiler needs to do a base-class offset calculation: Use a function pointer containing the address of the offset-fixup thunk that calls our function.
4. Compiler needs both offset and function vtable lookups: Use a function pointer that points to the vtable thunk_n thunk. That will at run-time look up the vtable and call the offset-fixup thunk which will get everything to work.

Thus the most complex case requires thunk-chaining two links long, but still works with a single function pointer for the data.

What remains now is the specification for member pointers. These are things which take a pointer to an object, and will return a pointer to the required member of that object no matter the status of the inheritance tree. It turns out that these also can be represented by something the size of a single pointer.

If the inheritance tree is not virtual, then given the object, we just need to do an offset calculation to find where the member is. A thunk which does this is:

```
return &((char *)this + n);
```

where we obviously require a different thunk for each possible `n`. (But don't require anything parameterized by class type due to the low-level nature of the operation.)

If the inheritance tree is virtual, then we'll need to access the vtable in the thunk:

```
return &((char *)this + this->vtable[n]);
```

Thus a second set of thunks parameterized by `n` is required. The exact ones we need for each set is calculable at compile time.

The final difference between a member function pointer and a normal function pointer is the calling convention. (thiscall verses stdcall or cdecl) If member function pointers had the same calling convention as normal functions, then the two types of pointer could then be identical. This would require explicit mention of the hidden `&this` reference that member functions have. However, other languages such as Python do this, at it isn't much of a burden on the programmer.

5. **0 is NULL**

When C++ was created there were two different ways of writing a NULL pointer in C. The first of these was to use a zero, the second was to use zero cast to a pointer type. Once the `void` type was introduced, NULL pointers became universally defined in C as:

```
#define NULL ((void *) 0)
```

whereas C++ decided to standardize on using open-coded 0's for NULL pointers.

The reason C++ did this was because as we have seen, pointers are different sizes in C++. Thus casting between different pointer types can lose information. This fact, together with the attempt to provide strong type safety meant that the behavior of casting to and from `void *` pointers is different between the two languages.

In C, any data pointer can be assigned to a void pointer, and vice versa. No information is lost, and no cast is required. This vastly improves the language because certain functions which don't care about the type of object pointed to by the pointer can be written in a polymorphic manner. For example the return value from the `malloc()` function is a void pointer because it doesn't care what you use the memory for.

C++ is different. You cannot cast to and from a void pointer without a cast. This means that a NULL pointer cannot be implemented in the C manner. If you try to assign a NULL void pointer to some other pointer, you'll need a cast. Using zero as a keyword for NULL fixes this.

Unfortunately, using zero as a keyword for NULL has serious drawbacks. The first of these is that pointers and integers can be of different bit-depth, and a zero is usually parsed as an integer. So if you are passing a NULL pointer parameter to a variable-argument function, then you cannot use zero. The compiler will think you mean an integer, and with typical calling conventions will store an integer-sized thing on the stack. Now, if you are on a 64bit machine, then pointers are 64bits, and integers 32bits. Thus when the called function goes to read the pointer, it will find 32bits of the pointer set to zero, but the other 32bits remaining at whatever values just happened to be on the stack at the time. The resulting pointer is corrupt - and not a NULL pointer at all. Thus an explicit cast to the correct pointer type is required.

Another problem arises when you want to pass a NULL pointer to an overloaded function which also accepts integer arguments in the same location as the pointer. Using zero will default to the integer variant of the function, which may not be what was intended. Again, an explicit cast is required.

The new C++ standard attempts to rectify this problem by creating yet another keyword: "nullptr". Why didn't they call this new thing "NULL" to match decades of practice in the C universe? Also, why is this keyword required at all? As is shown above, it is possible to redefine the member pointer and member function pointer calling conventions so that they are the same size as a normal function pointer. On nearly all architectures function pointers are the same size as data pointers. Thus the reason for disallowing implicit casts to and from `void *` disappear. If such implicit casts are allowed, then the old C macro definition of NULL works just as well.

6. **Operator Overloading**

Many programmers complain about operator overloading in C++. The reason they complain is that its use tends to make code harder to understand. Thus some coding styles completely ban its use. However, mathematical object classes (such as arbitrary precision floating point numbers, matrices, octonions, integers mod Z etc.) are much easier to use when it is allowed, so a complete removal from the language would be a loss.

The real reason for the problem is easy to explain. Imagine I've created a wonderful new programming language. However, it has one catch. All function names must be a single letter long, and to make things simple, case doesn't matter. Thus all programs must deal with having a grand total of 26 functions. To get over this limitation, I allow overloading by type and perhaps argument number for these functions. Thus by selecting the correct overload, you may overcome the 26 function limitation, and write any program you wish. You wouldn't want to use such a language would you? The 26-symbol limitation remains a huge obstacle for creating easily maintainable code.

This is the problem faced by users of operators. You can overload the operators that exist... but you are not allowed to create new operators. Why not allow the creation of new operators like is allowed in other languages? What prevents me from defining the "%^%" token to be a wedge product, and the %*% token as a scalar dot product?

The reason this is disallowed currently is the tokenization done in the pre-processor. Symbols are tokenized by a maximal-munch technique that requires the pre-processor to know which characters can represent symbols, and which are operators. However, it should be possible to modify this behavior. Define three types of character: separators such as white space and brackets, symbols characters such as letters and numbers, and operator characters such as punctuation. Let a token be either a bracket, a string of symbols, or a string of punctuation characters.

This is obviously not backwards compatible, as the tokenizer currently will nicely parse things like:

```
x=!-~y;
```

To get this to correctly parse with the new implementation would need the addition of brackets or white space:

```
x = !(-(~y));
```

However, this is a rather extreme case. Most coding styles typically add whitespace around operators, and extra brackets beyond that strictly required by precedence, to improve readability. Thus in real-world code, less changes than might be expected are required.

The result is that something like the following could be legal C++

```
item @ (graph --> node) = %%value
```

7. **Template Syntax**

A big problem with the C++ language is its implementation of meta-programming. Currently, this is done with templates. During the standardization of templates, it was found by accident that they had constructed a turing-complete mini language that runs at compile time. Originally, this was a curiosity. However, with the development of the Boost library, and its subsequent standardization, template meta-programming is venturing into the main stream.

The problem with templates is that they weren't designed to be used in this way, and it shows. The resulting metalanguage is extremely verbose, hard to read and maintain. It is also functional rather than imperative, so all iteration needs to be converted into recursive type expansion. Various attempts have been made at improving it, but none have succeeded. (The latest failure is that of "concepts" which were too immature to standardize for C++0X.)

Lets try to attack this problem from another direction. Completely ignoring templates... what would be your choice of language to use for a compile-time mini language? If you are a C++ programmer, the obvious choice would be C++ itself. So the question becomes, how much does the C++ language need to change in order to allow its use as a compile-time language? The answer to that, is not that much at all.

Firstly, to make a compile-time version of C++ we'll need to distinguish between sections of code to be executed at compile time, and sections to be run at runtime. The "metacode" C++ extension attempted to do this through the use of a new keyword. However, I don't think a new keyword is required. We can make one observation, since types are abstract quantities, we can't do anything with them at runtime. (C++ isn't lisp). This means that any "function" that takes a type as one of its arguments must be run at compile time.

The result of this is that function-templates fall out of the woodwork. We just need to have the ability of declaring something as a type in a function argument. The keyword `typename` already exists, and is perfect for this purpose. So what would such code look like?

```
// Multiple possible returns?
auto max(typename T)
{
	// Compile time reflection somehow?
	if (!T::operator>(T,T))
	{
		// Some method of compile-time error support
		cerror("Type doesn't support operator>");
	}

	// Return a lambda as the "templated" function
	return [](T x, T y)->T {return x > y ? x : y;};
}

//Example usage
z1 = max(decltype(x))(x, y);
z2 = max(int)(x, y);

//Inject a specific version into the symbol table, no new keyword required.
int max(int, int) = max(int);

//Function overloading works now that the injection is done
z3 = max(x, y);
```

As can be seen there are still a few more issues to investigate. We still need some way of emulating the "Substitution failure is not an error" feature of templates. A form of compile-time type reflection could replace this. (It could also replace "concepts" as well. A concept is just a compile-time function that returns a bool saying whether or not a type has certain properties.) Another thing to note, is that the compile time version of the `if` statement needs to avoid interpreting sections of code that are not executing. This allows some sort of syntax-checking to work.

The major drawback of the above is that there is no more implicit type inference as is done with templates. The type of the `max` function is determined manually. This problem can be reduced by for example allowing symbol injection through the function assignment syntax hack used above.

What about class templates? They are much like function templates, but the compile-time "function" must return a type instead of a run-time function. The resulting syntax could look something like:

```
auto my_template(typename T, int x)
{
	// Perhaps "typename result1, result2;" instead?
	class result1;
	class result2;

	// if must work at compile time
	if (x > 0)
	{
		class result1;
		{
			// This definition would be an error if the if statement didn't protect us.
			T foo[x];

			result1(T data)
			{
				foo[0] = data;
			}
		};
	}
	else
	{
		class result2;
		{
			T foo;

			result2(T data)
			{
				foo = data;
			}
		};
	}
```

```
        // Return the correct class as the return typename.

        if (!x) return result2;

        return result1;
}

// Declare user
typename x = decltype(data);
my_template(x) user;

// Initialize
my_template(x)(&user, data);
```

The above works in a very similar way to class templates in the run-time code, except that the greater and less-than symbols get replaced by normal brackets. The use of "inner classes" also makes the symbol scope of the resulting meta-type much easier to understand. In addition, it is trivial to notice that the compile-time functions can return other things as well as types and functions. Returning plain old data, or compile-time initialized objects is easy to implement.

The only problems to remain are how to implement compile-time reflection, and a nicer way of doing function symbol injection. These can perhaps be done with new syntax that goes beyond the scope of this current article. (i.e. perhaps an "`ifexists`" keyword is required to prevent ambiguity in symbol-table checking.)

Finally, what about template overloading? The above techniques work by having static if or switch-like behavior within a meta-function to choose which run-time behavior to implement. Current templates allow a user to add functionality through simply adding a new overload. This is not immediately possible with the method shown above. However, it is trivial to add it if required. All that is needed is a user meta call-back hook:

```
//Does the user hook function exist for this meta-function?
ifexists (user_hook(this_template, type_1, type2))
{
        // Use the user overload instead
        return user_hook(this_template, type_1, type2);
}

//Continue with switching based on types to implement default behavior.
if (type_1 == int)
{
        //Do something
```

8. The Export Keyword

The `export` keyword is designed to allow a programmer to use separate compilation of templates and user code. Thus user code could include a header which declares a template but not define it. This would naively increase compilation speed as the same template definitions would not have to be parsed and recompiled for every source file. They could be compiled only once, and the linker would somehow sort out how everything would work.

Unfortunately, reality isn't so kind. The `export` keyword even though it is part of the standard, is not portable. The reason for this is that hardly any compilers support it, and the reason for that in turn is because it is much harder to implement than it seems at first glance.

The problem with the `export` keyword is that it breaks some basic assumptions about symbol lookup within a compiler. This requires a major refactoring of the compiler's implementation making the feature difficult to implement. The source of all the problems is that template instantiation requires lookups in two different symbol tables. The first is that existing when the template is defined, the second is that where it is used. This means that counter-intuitively, some file-local static variables need to be exported into the global namespace, and similarly, so do some things in anonymous name spaces.

The second problem is one of performance. As currently implemented, the `export` keyword cannot speed up compile time. The repeated work that we try so hard to remove is simply deferred until link time. This is a disaster, and removes the entire point of having this feature.

So how might the `export` keyword feature be salvaged? The first thing to fix is the symbol lookup problem. This might be achieved after moving to the meta-C++ compile-time template scheme by enforcing new symbol scope rules that prevent the problematic cases from existing. (Since we are changing how templates work, we might as well also alter that as well because there are no backwards compatibility requirements to maintain.)

How could performance be improved? This is a function of the type of meta-programming used. Templates were originally used as a more powerful form of code-massager than the C preprocessor. In this form, there is very little a compiler can do with a raw template definition. However, modern template programming uses them in much a more complex manner. For sufficiently complex meta-code it should become efficient to process them at definition time.

A sketch of a compiler implementation might be for the compiler to convert the template definitions into byte-code at compile time. This byte-code version of C++ can then be executed when required (at link time) to generate the resulting output. Memoization could be used to increase performance. (To make memoization useful, meta-functions would need to be "pure" and only depend on their arguments.) The trick is noticing that the byte-code can be optimized if the meta-functions are sufficiently complex. Thus this might provide extra performance just when we need it.

9. Classes verses Structs

In C++ currently, there is very little difference between classes and structs. In short, a struct has its members public by default, whereas a class has them private. However, both of these defaults can be overridden as required. The result is using the `struct` or `class` keyword becomes a matter of style. If something behaves more like plain-old-data, it might just be more expressive to declare it as a struct, whereas if something contains complex internal logic it probably should be declared as a class.

The problem is that the boundary between the two types of object types is fuzzy. Wouldn't it be nice to use the extra expressivity we have with the two different keywords in another way? The first thing to notice, is what exactly do we mean by "behave like plain-old-data"? Basically, such things can be freely copied about by raw memory copy operations. They have no internal state, and no compiler-hidden fields like a vtable pointer.

Objects which are not plain-old-data are quite different. They know more about themselves. They may have internal pointers or references that need to be updated on a copy. They may have virtual functions or inheritance which requires some form of vtable access. In short, they may not be freely copied by a raw memory copy and have the copy work as expected.

Why not use this difference between plain-old-data and more complex objects to define the difference between structs and classes? This is more than just semantics. If we know that structs must be raw copyable, then we can provide extra constraints on the memory layout. In practice, structs (together with unions) are used to access things like network packets, where the memory layout is defined by some standard. A struct cannot have its members move about, they must be in the order and alignment requested by the programmer. Otherwise much legacy code will not work.

Classes can be a whole different beast. A programmer tends not to care about the memory layout of a class. A class will probably have its own copy and initialization functions that take care of the layout abstraction process. So why not explicitly allow compilers to rearrange data members within a class? By sorting them in size, and packing them efficiently, memory can be saved.

In short, perhaps there should be a new difference between structs and classes. Structs have a fixed memory layout, classes have an arbitrary memory layout. This would enforce things like not using virtual inheritance with structs... but that would be good coding style anyway.

10. **Barriers**

The current proposed C++0X atomic operations extension is extremely complex as it attempts to handle all possible threading operations in a portable manner. This task is difficult because up until now, atomic operations have been highly non-portable between machine architectures. Some simply are not supported on all machines, and the differing memory models make it difficult to describe ordering constraints that are both possible and performant.

The technique chosen is to describe a set of atomic types which can then be used as arguments to various atomic operations. The operations are then in turn parametrised by type of memory model. Thus operations requesting "weak" ordering constraints can use cheaper instructions than those requesting "strong" ordering.

Most of the specification is well designed. However, as it currently is written, there remains a problem with "fence" operations. The problem with these is that they also are attached to the atomic memory locations. There are a couple of reasons why this is a bad idea. The first is that the underlying machine instructions emitted for a fence do not require one. Fences are a global operation. The reason for that is a fence synchronizes between multiple memory locations. Thus a second reason for the API being poor is that the other memory locations are being ignored in the operation. Choosing one memory location to be "primary" makes no sense. In short, the current fence operations need to be recast into global fence functions.

A second problem is that there are no thread-local fence operations. A thread of execution may want to synchronize with itself. Since processors are smart enough to get single-threaded memory accesses in program order, this type of fence translates into a compile barrier. This prevents the compiler from reordering loads and stores across it, but is invisible to the processor. Compile barriers are useful when code needs to deal with signals or interrupts. They also are useful when user-mode threading via "fibers" are used. Using compile barriers instead of full fences can yield great performance improvements since no slow atomic instructions need be emitted.

Finally, the specification neglects to mention the last remaining common threading assembly instruction. This is the use of the hyperthreading-aware `cpu_relax (rep; nop)` which tells the cpu not to use too many resources. This instruction is used inside busy-loops waiting on some value to change. There is no point for the cpu checking the memory bus more often than it can be updated, so it is better off giving its computational resources to its hyperthreading partners. (If used as an intrinsic, this function/instruction should also act as a compile barrier, since we know we are waiting on some value to change, and might as well put the compile read barrier here.)

So the C++ language has quite a few problems. However, they all seem fixable if one were willing to break source and ABI backwards compatibility. Unfortunately, reality isn't so kind, and it is unlikely that many of these changes will ever be made due to the cost of altering interfaces used for many years. Hopefully, this has provided constructive criticism, and enough detail in the solutions proposed to be useful.

Comments

Sandor Hojtsy said...

Seriously, C++ doesn't have the syntax that the "10 problems with C++" article names as "current syntax":

class_type x = new class_type(args);

Are you confusing with java, or what? I suggest fixing the error in the article.

Also how on earth would the for loop (which could not even occur where the a global variable definition could) in your suggested syntax below know how many elements are contained behind some pointer?

classtype *x = new(class_type, n);
for(class_type &xi : x) class_type(xi, args);

Thanks for reading my comments.

sfuerst said...

Oops, sorry. There were a couple of typos.

The first should have been written as:
class_type *x = new class_type(args);

(The asterix to make x a pointer type was missing.)

Also, the for loop should have been a "foreach" loop:
foreach(class_type &xi: x) class_type(xi, args)

I've altered the text to fix these silly mistakes. Thanks for noticing the problems.

infact said...

Good article.

tsc said...

Just found this site... Thanks for your articles, they're quite interesting.

This one I found somewhat stereotypical for a "good ol' C cowboy". However, it shows a deep understanding of the language, so could be fun to comment on.

1. The new operator.
1.1. You can parenthesize operator new (see 5.3.4 3).
1.2. You can legally provide definitions for a custom, global new (17.4.3.4 2).

2. Exceptions
2.1. A C program either IS a mess or reads that way for properly checking success codes returned by functions. In my humble opinion exceptions are a much superior way of handling error conditions. We shall not forget that collaborating developers communicate through source code - and it's not ONLY instructions to a machine.
2.2. Writing exception safe code is not that hard once you get used to it, really. However, in your article it seems "exception safe code" is confused with "non-throwing code" which are two very different pairs of shoes.
2.3. Of course you cannot turn back the time in a failed initialization, but this has nothing to do with how the error condition is raised (for example, there needs to be a well-defined protocol to tell a peer on the network to forget about something said before due to an error condition).
2.4. There are cases where you really need so called "n phase initialization", but "init" methods are overused. Often using an extra base class is the cleaner solution.

3. Multiple inheritance
Mentioned problem is not really an issue about the language itself but its implementation. Some notes, however:
3.1. For your optimization to pay off every function must be called in several places in the derived class.
3.2. If we have one non-polymorphic and one polymorphic base class the offset is known at compile time and the the instance pointer adjustment becomes a simple addition - nothing I'd want to infer an extra jump for.
3.3. Since you're aware of the generated code you can write the thunk as a C++ method in the derived class.

4. Member pointers
4.1. Using member pointers and member function pointers on polymorphic classes smells like a chewing gum and duct tape solution to begin with. Stick to proper OOD and don't get there.
4.2. I'd prefer adjustment of the instance pointer at the call site rather than an extra jump for non-polymorphic case of call to secondary base class.
4.3. Member pointers work best as non-type template arguments - where they can be optimized out entirely.

5. NULL
5.1. 0l in my style - and usually 64 bits on 64 bits, too.

6. Operator overloading
6.1. Operator overloading is awesome...

```
list += a,b,c; // ...especially the comma
```

6.2. It has never been C/C++ philosophy to provide a bullet-proof language that cannot be misused. I love those degrees of freedom. If I couldn't handle them I picked another language.
6.3. I agree that the rules for operator overloading are somewhat arbitrary, irregular and clumsy (e.g. pre-/postincrement or nothrow-new)

7. Templates syntax
7.1. I'm fine with templates as-is. Yes, I do write metaprograms. No, I would not want an imperative language unless I also get a compile time debugger.
7.2. Doing compile time calculations on top of the type system seems very reasonable to me. This way the two worlds are clearly separated and allows perfect control over the PoI (also the point of type erasure when building a code generation facility).
7.3. Examples very much generic programming a la D (you mentioned Digital Mars, so I guess it was the source of inspiration). Getting closer to that with "auto" in the next standard version.

9. Structs vs. Classes
9.1. I'd too welcome a portable "#pragma packed"

Well, I guess I like C++. And to me there's no doubt that it is by far the most powerful, portable low-level code generation technology available. For completeness, my list of C++ language issues (there are probably more, but those immediately pop up in my head when I think about weaknesses of C++):

1. Run time type information
1.1. No real type information except the name
1.2. Non-standardized naming
1.3. No way to selectively disable (or enable) it causing lots of bloat, especially when polymorphism and templates are in the game

2. Lack of C99 features, in particular:
2.1. Variable-length arrays
2.2. Variadic macros

3. Standard library quirks
3.1. STL-iterators and verbose <algorithm> calls
3.2. Missing template 'find' method in std::set
3.3. STL implementations overusing inlining
3.4. STL implementations ignoring allocator type members
3.5. Standard headers too coarse-grained

FUZxxl said...

I often code in Haskell, a language where operators are just functions composed whose names are composed of symbols instead of letters. This and a more systematically way for operator overloading makes it extremely easy to write useful and readable code.

sampath said...

cant C++ developers fix those errors in future, without worring about backward compatibility issue?

rs said...

This article suffers from confusion between C++98 and C++11. You seem to be aware of some C++11 features (nullptr, lambdas), but not the others which fix or mitigate most of your problems.

1. tsc has mentioned that you can override 'operator new' if you want. You can also initialize arrays: "new int[5] { 1, 2, 3, 4, 5 };".
3. Common C++ ABIs have used thunks for the *this-offset for virtual inheritance for years.
4. Data member pointers are usually the same size as ptrdiff_t. Pointers to member functions can't be dealt with as simply as you suggest, because a pointer to member function can be cast to a pointer to member of another class in the hierarchy. Thus you would need some mechanism to compute which thunk to call, at runtime, based on a thunk pointer and an offset for this.
5. Why is 'nullptr' not called 'NULL'? Well, requiring that would break a lot of code. We're in a transition period; right now, "#define NULL nullptr" is valid for a C++ implementation, but not required. The next C++ standard may make it mandatory. Implicit casts from (void*) are disallowed because they are not type safe, not for some concern about pointer width.
7. Allowing a deduced function return type, a la 'auto foo(int n) { ... }', is already approved for C++14. For compile-time errors, we already have 'static_assert' in C++11. We already have 'constexpr' to allow C++ to be used as a compile-time mini-language.

8. 'export' has been removed from the language already.
10. C++11 has fences which are not attached to any atomic operation.

ginnie hwang said...

wats dis????

said...

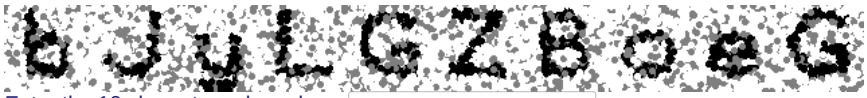
Enter your comments here

said...

Enter your comments here

said...

Enter your comments here



Enter the 10 characters above here

Enter your comments here

Name

Submit