DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 1 : Dynamic Object Creation

2. Re: Object creation and destruction. Aug 2, PM (in response to) sorry, i mean calling a dispose() method on an internal frame or 'nullying.

Initialization tasks often must be performed on new objects before they are used. Common initialization tasks include opening files, connecting to databases, and reading values of registry keys. Visual Basic controls the initialization of new objects using procedures called constructors special methods that allow control over initialization. After an object leaves scope, it is released by the common language runtime CLR. Visual Basic controls the release of system resources using procedures called destructors. Together, constructors and destructors support the creation of robust and predictable class libraries. Using Constructors and Destructors Constructors and destructors control the creation and destruction of objects. Sub New The Sub New constructor can run only once when a class is created. It cannot be called explicitly anywhere other than in the first line of code of another constructor from either the same class or from a derived class. Furthermore, the code in the Sub New method always runs before any other code in a class. Visual Basic and later versions implicitly create a Sub New constructor at run time if you do not explicitly define a Sub New procedure for a class. To create a constructor for a class, create a procedure named Sub New anywhere in the class definition. To create a parameterized constructor, specify the names and data types of arguments to Sub New just as you would specify arguments for any other procedure, as in the following code: Sub New ByVal s As String, i As Integer When you define a class derived from another class, the first line of a constructor must be a call to the constructor of the base class, unless the base class has an accessible constructor that takes no parameters. A call to the base class that contains the above constructor, for example, would be MyBase. New is optional, and the Visual Basic runtime calls it implicitly. Sub New can accept arguments when called as a parameterized constructor. The Finalize method can contain code that needs to execute just before an object is destroyed, such as code for closing files and saving state information. There is a slight performance penalty for executing Sub Finalize, so you should define a Sub Finalize method only when you need to release objects explicitly. Note The garbage collector in the CLR does not and cannot dispose of unmanaged objects, objects that the operating system executes directly, outside the CLR environment. This is because different unmanaged objects must be disposed of in different ways. That information is not directly associated with the unmanaged object; it must be found in the documentation for the object. A class that uses unmanaged objects must dispose of them in its Finalize method. The Finalize destructor is a protected method that can be called only from the class it belongs to, or from derived classes. Visual Basic and later versions allow for a second kind of destructor, Dispose , which can be explicitly called at any time to immediately release resources. Note A Finalize destructor should not throw exceptions, because they cannot be handled by the application and can cause the application to terminate. How New and Finalize Methods Work in a Class Hierarchy Whenever an instance of a class is created, the common language runtime CLR attempts to execute a procedure named New, if it exists in that object. New is a type of procedure called a constructor that is used to initialize new objects before any other code in an object executes. A New constructor can be used to open files, connect to databases, initialize variables, and take care of any other tasks that need to be done before an object can be used. When an instance of a derived class is created, the Sub New constructor of the base class executes first, followed by constructors in derived classes. This happens because the first line of code in a Sub New constructor uses the syntax MyBase. New to call the constructor of the class immediately above itself in the class hierarchy. The Sub New constructor is then called for each class in the class hierarchy until the constructor for the base class is reached. At that point, the code in the constructor for the base class executes, followed by the code in each constructor in all derived classes and the code in the most derived classes is executed last. When an object is no longer needed, the CLR calls the Finalize method for that object before freeing its memory. The Finalize method is called a destructor because it performs cleanup tasks, such as saving state information, closing files and connections to databases, and other tasks that must be done before releasing the object. IDisposable Interface Class instances often control resources not managed by the CLR,

such as Windows handles and database connections. These resources must be disposed of in the Finalize method of the class, so that they will be released when the object is destroyed by the garbage collector. However, the garbage collector destroys objects only when the CLR requires more free memory. This means that the resources may not be released until long after the object goes out of scope. To supplement garbage collection, your classes can provide a mechanism to actively manage system resources if they implement the IDisposable interface. IDisposable has one method, Dispose , which clients should call when they finish using an object. You can use the Dispose method to immediately release resources and perform tasks such as closing files and database connections. Unlike the Finalize destructor, the Dispose method is not called automatically. Clients of a class must explicitly call Dispose when you want to immediately release resources. Implementing IDisposable A class that implements the IDisposable interface should include these sections of code: A field for keeping track of whether the object has been disposed: This method should be called by the Dispose and Finalize methods of the base class: Dispose Dispose True GC. Finalize End Sub Deriving from a Class that Implements IDisposable A class that derives from a base class that implements the IDisposable interface does not need to override any of the base methods unless it uses additional resources that need to be disposed. Garbage Collection and the Finalize Destructor The. NET Framework uses the reference-tracing garbage collection system to periodically release unused resources. Although both systems perform the same function automatically, there are a few important differences. The CLR periodically destroys objects when the system determines that such objects are no longer needed. Objects are released more quickly when system resources are in short supply, and less frequently otherwise. The delay between when an object loses scope and when the CLR releases it means that, unlike with objects in Visual Basic 6. In such a situation, objects are said to have non-deterministic lifetime. In most cases, non-deterministic lifetime does not change how you write applications, as long as you remember that the Finalize destructor may not immediately execute when an object loses scope. Another difference between the garbage-collection systems involves the use of Nothing. To take advantage of reference counting in Visual Basic 6. In later versions of Visual Basic, while there may be cases in which this procedure is still valuable, performing it never causes the referenced object to release its resources immediately. The only time you should set a variable to Nothing is when its lifetime is long relative to the time the garbage collector takes to detect orphaned objects.

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 2 : TypeScript Â· TypeScript

Synchronize threads' creation and destruction of (static) object. You may take a look at the boost implementation for some idea. - user2k5 Aug 2 Destruction.

This creates a reference to the object. For most object types, the object will only contain its default state when it is first bound to the context; until it is bound, attempting to use it will fail. All functions of this type have the same signature: This allows you to create multiple objects with one call. An object name is always a GLuint. These names are not pointers, nor should you assume that they are. They are references, numbers that identify an object. They can be any bit unsigned integer except 0. The object number 0 is reserved for special use cases; see below for details. In OpenGL versions before 3. The user could just decide that "3" is a valid object name, bind it to the context to create its default state, and then start using it like an object. The implementation would then have to accept that and create the object behind the scenes when you first start using it. In core GL 3. Once you are finished with an object, you should delete it. These functions have this signature: Any values that are not valid objects or are object 0 will be silently ignored. When OpenGL objects are deleted, their names are no longer considered valid. Deletion unbinding When an object is deleted, if the object is bound to the current context and note this only applies to the current context , then the object will be unbound from all binding to the context. This affects "binding", not "attachment". Objects are bound to the context, whereas attachment refers to one object referencing another. Attachments are not severed due to this call. Furthermore, if the object is attached to any container object , and that object is itself bound to the current context, the object will be unattached from the container object. If it is attached to an object that is not bound to the current context, then the attachment is not broken. Some objects can be bound to the context in unusual ways. These ways include, but are not limited to: Textures bound as images. This does not unbind it from the indexed target. These "bindings" are reset to their default state. Again, recall that this only takes place for the current OpenGL context. If an object is still "in use" after it is deleted, then the object will remain alive within the OpenGL implementation. An object is "in use" if: It is bound to a context. This is not necessarily the current one, since deleting it will automatically unbind it from the context that caused the deletion. Though remember the caveat above about non-standard binding points for GL 4. It is attached to a container object. So if a Texture is attached to a Framebuffer Object that is not bound to the context, the FBO will still be functional after deleting the texture. Only when the FBO is either deleted or a new texture attachment replaces the old will the texture finally be fully deleted. This means that any functions that change the state governed by that object will simply change the state within the object, thus preserving that state. Binding a newly generated object name will create new state for that object. In some cases, the target to which it is first bound see below will affect properties of the newly created state for the object. Different objects have different binding functions. They do share a naming convention and general parameters: Some objects can be bound to multiple targets in the context, while others can only be bound to a single specific place. For example, a buffer object can be bound as an array buffer, index buffer, pixel buffer, transform buffer, or various other possibilities. Different targets have separate bindings. So you can bind a buffer object as an vertex attribute array, and a different buffer object as an index buffer. If an object is bound to a location where another object is already bound, the previously bound object will be unbound. However, some objects treat it in different ways. For most object types, object 0 is much like the NULL pointer: If 0 is bound for such object types, then attempts to use that bound object for rendering purposes will fail. For some objects, object 0 represents a kind of "default object". Textures have a default texture. However, the default texture is very complex to use; texture object 0 technically represents multiple default textures. Furthermore, default textures cannot be used in many ways that regular texture objects can. For example, you cannot attach images of them to FBOs. Therefore, you are strongly encouraged to think of texture 0 as a non-existent texture. For Framebuffers , object 0 represents the Default Framebuffer. It has similar state compared to a proper Framebuffer Object , but it has a very different set of images with its own image names. Even so, image property query functions do work on object 0, as do general framebuffer interface functions. With the exception of Framebuffer Objects , you should treat object 0

as a non-functional object. Even if an object type has a valid object 0, you should treat it as if it did not.

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 3 : profiling - JAVA method for Class/Object creation destruction - Stack Overflow

Among the dispatch routines that a kernel-mode driver can supply, there are a few related to the creation and destruction of the file objects. This article discusses the use of the file object from the driver's perspective and the related dispatch routines.

Overview[edit] While the basic idea of object lifetime is simple â€” an object is created, used, then destroyed â€” details vary substantially between languages, and within implementations of a given language, and is intimately tied to how memory management is implemented. Further, many fine distinctions are drawn between the steps, and between language-level concepts and implementation-level concepts. Terminology is relatively standard, but which steps correspond to a given term varies significantly between languages. This varies by language, and within language varies with the memory allocation of an object; object lifetime may be distinct from variable lifetime. Objects with static memory allocation , notably objects stored in static variables , and classes modules if classes or modules are themselves objects, and statically allocated , have a subtle non-determinism in many languages: In garbage-collected languages, objects are generally dynamically allocated on the heap even if they are initially bound to an automatic variable, unlike automatic variables with primitive values, which are typically automatically allocated on the stack or in a register. This allows the object to be returned from a function "escape" without being destroyed. However, in some cases a compiler optimization is possible, namely performing escape analysis and proving that escape is not possible, and thus the object can be allocated on the stack; this is significant in Java. A complex case is the use of an object pool , where objects may be created ahead of time or reused, and thus apparent creation and destruction may not correspond to actual creation and destruction of an object, only re initialization for creation and finalization for destruction. In this case both creation and destruction may be nondeterministic. Steps[edit] Object creation can be broken down into two operations: These are implementation-level concepts, roughly analogous to the distinction between declaration and initialization or definition of a variable, though these later are language-level distinctions. For an object that is tied to a variable, declaration may be compiled to memory allocation reserving space for the object , and definition to initialization assigning values , but declarations may also be for compiler use only such as name resolution , not directly corresponding to compiled code. Analogously, object destruction can be broken down into two operations, in the opposite order: These do not have analogous language-level concepts for variables: Together these yield four implementation-level steps: Initialization is very commonly programmer-specified in class-based languages , while in strict prototype-based languages initialization is automatically done by copying. Allocation is more rarely specified, and deallocation generally cannot be specified. Status during creation and destruction[edit] An important subtlety is the status of an object during creation or destruction, and handling cases where errors occur or exceptions are raised, such as if creation or destruction fail. Thus during initialization and finalization an object is alive, but may not be in a consistent state â€” ensuring class invariants is a key part of initialization â€” and the period from when initialization completes to when finalization starts is when the object is both alive and expected to be in a consistent state. If creation or destruction fail, error reporting often by raising an exception can be complicated: The opposite issue â€” incoming exceptions, not outgoing exceptions â€” is whether creation or destruction should behave differently if they occur during exception handling, when different behavior may be desired. Another subtlety is when creation and destruction happen for static variables , whose lifespan coincides with the run time of the program â€” do creation and destruction happen during regular program execution, or in special phases before and after regular execution â€” and how objects are destroyed at program termination, when the program may not be in a usual or consistent state. This is particularly an issue for garbage-collected languages, as they may have a lot of garbage at program termination. Class-based programming[edit] In class-based programming, object creation is also known as instantiation creating an instance of a class , and creation and destruction can be controlled via methods known as a constructor and destructor , or an initializer and finalizer. Creation and destruction are thus also known as construction and destruction, and when these methods are called an object is said to be constructed or

destructed not "destroyed" â€” respectively, initialized or finalized when those methods are called. A key distinction is that constructors are class methods, as there is no object class instance available until the object is created, but the other methods destructors, initializers, and finalizers are instance methods, as an object has been created. Further, constructors and initializers may take arguments, while destructors and finalizers generally do not, as they are usually called implicitly. The steps during finalization vary significantly depending on memory management: Thus in automatic reference counting, programmer-specified finalizers are often short or absent, but significant work may still be done, while in tracing garbage collectors finalization is often unnecessary. Resource management[edit] In languages where objects have deterministic lifetimes, object lifetime can also be used for resource management , notably via the Resource Acquisition Is Initialization RAII idiom: In languages where objects have non-deterministic lifetimes, notably due to garbage collection, resources are managed in other ways, notably the dispose pattern: Using object lifetime for resource management ties memory management to resource management, and thus is not generally used in garbage-collected languages, as it would either constrain the garbage collector requiring immediate finalization or result in possibly long-lasting resource leaks , due to finalization being deferred. Object creation[edit] In typical case, the process is as follows: When the object in question is not derived from a class, but from a prototype instead, the size of an object is usually that of the internal data structure a hash for instance that holds its slots. For example, in multi-inheritance , which initializing code should be called first is a difficult question to answer. However, superclass constructors should be called before subclass constructors. It is a complex problem to create each object as an element of an array. Handling exceptions in the midst of creation of an object is particularly problematic because usually the implementation of throwing exceptions relies on valid object states. For instance, there is no way to allocate a new space for an exception object when the allocation of an object failed before that due to a lack of free space on the memory. Due to this, implementations of OO languages should provide mechanisms to allow raising exceptions even when there is short supply of resources, and programmers or the type system should ensure that their code is exception-safe. Propagating an exception is more likely to free resources than to allocate them. But in object oriented programming, object construction may fail, because constructing an object should establish the class invariants , which are often not valid for every combination of constructor arguments. Thus, constructors can raise exceptions. The abstract factory pattern is a way to decouple a particular implementation of an object from code for the creation of such an object. Creation methods[edit] The way to create objects varies across languages. In some class-based languages, a special method known as a constructor , is responsible for validating the state of an object. Just like ordinary methods, constructors can be overloaded in order to make it so that an object can be created with different attributes specified. Also, the constructor is the only place to set the state of immutable objects [Wrong clarification needed]. Other programming languages, such as Objective-C , have class methods, which can include constructor-type methods, but are not restricted to merely instantiating objects. This can be problematic if the programmer wants to provide two constructors with the same argument types, e. However, if there is sufficient memory or a program has a short run time, object destruction may not occur, memory simply being deallocated at process termination. In some cases object destruction simply consists of deallocating the memory, particularly in garbage-collected languages, or if the "object" is actually a plain old data structure. In other cases some work is performed prior to deallocation, particularly destroying member objects in manual memory management , or deleting references from the object to other objects to decrement reference counts in reference counting. This may be automatic, or a special destruction method may be called on the object. In garbage collecting languages, objects may be destroyed when they can no longer be reached by the running code. In class-based GCed languages, the analog of destructors are finalizers , which are called before an object is garbage-collected. Example of such languages include Java , Python , and Ruby. Destroying an object will cause any references to the object to become invalid, and in manual memory management any existing references become dangling references. In garbage collection both tracing garbage collection and reference counting , objects are only destroyed when there are no references to them, but finalization may create new references to the object, and to prevent dangling references, object resurrection occurs so the references remain valid. Integer ; constructor

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

CreateCopy APoint:

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 4 : Data Objects and Data Sources: Creation and Destruction

Tracing destruction is more difficult - you can't hook into the finalize method because it is not mandatory for an object to provide one. Right now you would need to run BTrace in unsafe mode and provide your own logic using eg.

Destroying data sources Creating Data Objects Data objects are used by the destination application â€" either the client or the server. A data object in the destination application is one end of a connection between the source application and the destination application. A data object in the destination application is used to access and interact with the data in the data source. There are two common situations where a data object is needed. The first situation is when data is dropped in your application using drag and drop. The second situation is when Paste or Paste Special is chosen from the Edit menu. In a drag-and-drop situation, you do not need to create a data object. A pointer to an existing data object will be passed to your OnDrop function. This data object is created by the framework as part of the drag-and-drop operation and will also be destroyed by it. This is not always the case when pasting is done by another method. For more information, see Destroying Data Objects. If the application is performing a paste or paste special operation, you should create a COleDataObject object and call its AttachClipboard member function. This associates the data object with the data on the Clipboard. You can then use this data object in your paste function. Destroying Data Objects If you follow the scheme described in Creating Data Objects , destroying data objects is a trivial aspect of data transfers. The data object that was created in your paste function will be destroyed by MFC when your paste function returns. If you follow another method of handling paste operations, make sure the data object is destroyed after your paste operation is complete. Until the data object is destroyed, it will be impossible for any application to successfully copy data to the Clipboard. Creating Data Sources Data sources are used by the source of the data transfer, which can be either the client or the server side of the data transfer. A data source in the source application is one end of a connection between the source application and the destination application. A data object in the destination application is used to interact with the data in the data source. Data sources are created when an application needs to copy data to the Clipboard. A typical scenario runs like this: The user selects some data. The user chooses Copy or Cut from the Edit menu or begins a drag-and-drop operation. The selected data is inserted into the data source by calling one of the functions in the COleDataSource:: The application calls the SetClipboard member function or the DoDragDrop member function if this is a drag-and-drop operation belonging to the object created in step 3. These sample programs provide a good example of how to implement these concepts. One other situation in which you might want to create a COleDataSource object occurs if you are modifying the default behavior of a drag-and-drop operation. For more information, see the Drag and Drop: Destroying Data Sources Data sources must be destroyed by the application currently responsible for them.

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 5 : INFO: Creation and Destruction of File Objects - ç¨‹åº•å›-

Every object, whether it is a custom object, a dynamic array or an array of objects, is created and deleted in MQL5-program in its particular way. Often, some objects are part of other objects, and the order of object deleting at deinitialization becomes especially important.

Dynamic Object Creation Sometimes you know the exact quantity, type, and lifetime of the objects in your program. How many planes will an air-traffic system need to handle? How many shapes will a CAD system use? How many nodes will there be in a network? If you could do that, you might: Accidentally do something to the object before you initialize it, expecting the right thing to happen. Hand it the wrong-sized object. And of course, even if you did everything correctly, anyone who modifies your program is prone to the same errors. The answer is by bringing dynamic object creation into the core of the language. However, if you have an operator to perform the combined act of dynamic storage allocation and initialization and another operator to perform the combined act of cleanup and releasing storage, the compiler can still guarantee that constructors and destructors will be called for all objects. Storage is allocated for the object. The constructor is called to initialize that storage. By now you should believe that step two always happens. Step one, however, can occur in several ways, or at alternate times: Storage can be allocated before the program begins, in the static storage area. This storage exists for the life of the program. Storage can be created on the stack whenever a particular execution point is reached an opening brace. That storage is released automatically at the complementary execution point the closing brace. These stack-allocation operations are built into the instruction set of the processor and are very efficient. Storage can be allocated from a pool of memory called the heap also known as the free store. This is called dynamic memory allocation. To allocate this memory, a function is called at runtime; this means you can decide at any time that you want some memory and how much you need. Often these three regions are placed in a single contiguous piece of physical memory: However, there are no rules. The stack may be in a special place, and the heap may be implemented by making calls for chunks of memory from the operating system. As a programmer, these things are normally shielded from you, so all you need to think about is that the memory is there when you call for it. These functions are pragmatic but primitive and require understanding and care on the part of the programmer. But the worst problem is this line: The problem here is that the user now has the option to forget to perform the initialization before the object is used, thus reintroducing a major source of bugs. When you create an object with new using a new-expression , it allocates enough storage on the heap to hold the object and calls the constructor for that storage. You can create a new-expression using any constructor available for the class. If the constructor has no arguments, you write the new-expression without the constructor argument list: Just as a new-expression returns a pointer to the object, a delete-expression requires the address of an object. For this reason, people often recommend setting a pointer to zero immediately after you delete it, to prevent deleting it twice. Deleting an object more than once is definitely a bad thing to do, and will cause problems. A simple example This example shows that initialization takes place: Note, however, that even though the function is declared as a friend , it is defined as an inline! Memory manager overhead When you create automatic objects on the stack, the size of the objects and their lifetime is built right into the generated code, because the compiler knows the exact type, quantity, and scope. Creating objects on the heap involves additional overhead, both in time and in space. This code may actually be part of malloc. The pool is searched for a block of memory large enough to satisfy the request. This is done by checking a map or directory of some sort that shows which blocks are currently in use and which are available. The way all this is implemented can vary widely. You can also read the library source code, if you have it the GNU C sources are always available. Early examples redesigned Using new and delete, the Stash example introduced previously in this book can be rewritten using all the features discussed in the book so far. Examining the new code will also give you a useful review of the topics. The reason this is not possible is because, in an attempt to be generic, they hold void pointers. Object int sz, char c: So you get a very quiet memory leak. If you have a memory leak in your program, search through all the delete statements and check the type of pointer being deleted. Cleanup responsibility with pointers To make the Stash and Stack

containers flexible able to hold any type of object , they will hold void pointers. The other memory leak issue has to do with making sure that delete is actually called for each object pointer held in the container. The user must be responsible for cleaning up the objects. And when you fetch a pointer back from the container, how will you know where its object has been allocated? Thus, you must be sure that objects stored in the following versions of Stash and Stack are made only on the heap, either through careful programming or by creating classes that can only be built on the heap. For a Stash of pointers, however, another approach is needed. Stash for pointers This new version of the Stash class, called PStash, holds pointers to objects that exist by themselves on the heap, whereas the old Stash in earlier chapters copied the objects by value into the Stash container. The destructor deletes the storage where the void pointers are held rather than attempting to delete what they point at which, as previously noted, will release their storage and not call the destructors because a void pointer has no type information. Here are the member function definitions: However, the fact that the function calls are more concise than the loops may help prevent coding errors. To put the responsibility of object cleanup squarely on the shoulders of the client programmer, there are two ways to access the pointers in the PStash: In the first case, note the line: During printing, the value returned by PStash:: The second test opens the source code file and reads it one line at a time into another PStash. When fetching the pointers, you see the expression: There must be a default constructor , except for aggregate initialization on the stack see Chapter 6 , because a constructor with no arguments must be called for every object. Because you wrote the code, you know that fp is actually the starting address of an array, so it makes sense to select array elements using an expression like fp[3]. But what happens when you destroy the array? The destructor will be called for the MyType object pointed to by the given address, and then the storage will be released. The proper amount of storage will still be released, however, because it is allocated in one big chunk, and the size of the whole chunk is stashed somewhere by the allocation routine. The solution requires you to give the compiler the information that this is actually the starting address of an array. This is accomplished with the following syntax: This is actually an improved syntax from the earlier form, which you may still occasionally see in old code: The additional overhead of letting the compiler handle it was very low, and it was considered better to specify the number of objects in one place instead of two. It makes more sense to define it as a constant, so any attempt to modify the pointer will be flagged as an error. Running out of storage What happens when the operator new cannot find a contiguous block of storage large enough to hold the desired object? A special function called the new-handler is called. Or rather, a pointer to a function is checked, and if the pointer is nonzero, then the function it points to is called. The default behavior for the new-handler is to throw an exception, a subject covered in Volume 2. You replace the new-handler by including new. The while loop will keep allocating int objects and throwing away their return addresses until the free store is exhausted. At the very next call to new, no storage can be allocated, so the new-handler will be called. The behavior of the new-handler is tied to operator new , so if you overload operator new covered in the next section the new-handler will not be called by default. Of course, you can write more sophisticated new-handlers, even one to try to reclaim memory commonly known as a garbage collector. This is not a job for the novice programmer. First, storage is allocated using the operator new , then the constructor is called. In a delete-expression , the destructor is called, then storage is deallocated using the operator delete. The constructor and destructor calls are never under your control otherwise you might accidentally subvert them , but you can change the storage allocation functions operator new and operator delete. The memory allocation system used by new and delete is designed for general-purpose use. The most common reason to change the allocator is efficiency: You might be creating and destroying so many objects of a particular class that it has become a speed bottleneck. Another issue is heap fragmentation. That is, the storage might be available, but because of fragmentation no piece is big enough to satisfy your needs. By creating your own allocator for a particular class, you can ensure this never happens. In embedded and real-time systems, a program may have to run for a very long time with restricted resources. The compiler will simply call your new instead of the default version to allocate storage, then call the constructor for that storage. So, although the compiler allocates storage and calls the constructor when it sees new, all you can change when you overload new is the storage allocation portion. When you overload operator new , you also replace the behavior when it runs out of

memory, so you must decide what to do in your operator new: Overloading new and delete is like overloading any other operator.

Chapter 6 : The Order of Object Creation and Destruction in MQL5 - MQL5 Articles

In a typical small GUI application, the programmer does not care about object destruction. GUI objects simply live until the end of the program. In multi window applications it might be useful to dispose a window object when the window gets closed.

A handle to the file object is returned to the caller if the call succeeds. Among the dispatch routines that a kernel-mode driver can supply, there are a few related to the creation and destruction of the file objects. It may not contain the valid file object pointer intended for the driver. Like other objects in Windows NT, each file object header has an open handle count and a reference count. The open handle count represents the number of handles that have been opened to the file object. The reference count represents the number of pointers to the object that are being used in the system. Each time the handle is duplicated or inherited, the open handle count is incremented. Each CloseHandle call decrements the count. The reference count is decremented when each operation is completed. Drivers may also reference or derefence fileobjects by calling Ob De referenceObject. In other words, the open handle count for the file object goes to 0. Of course, IRPs belonging to other file objects should not be canceled. Also, if an outstanding IRP is completed immediately, the driver does not have to cancel it. A driver with long-term IRPs should implement both. A cancel routine is called when the system attempts to cancel a specific IRP. If the thread terminates with some requests pending, the system tries to cancel the pending IRPs that belong to the terminating thread. As explained earlier, this happens when the last handle to a file object is closed. In this case, even if there are pending IRPs associated with the file object, the system does not try to cancel them. As discussed earlier, the reference count is different from the open handle count. The reference count can be a non-zero value when the open handle count goes to 0.

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 7 : Class: Object (Ruby)

Creating and Deleting Objects Contents Up Previous Next Slide Annotated slide Aggregated slides Subject index Program index Exercise index. Our goal in this section is to obtain an overall understanding of object creation and deletion, in particular in relation to the dimension of explicit/implicit creation and deletion.

Nil means the two values could not be compared. This method must always be called with an explicit receiver, as class is also a reserved word in Ruby. See also the discussion under Object dup. The method parameter can be a Proc, a Method or an UnboundMethod object. If a block is specified, it is used as the method body. This method may have class-specific behavior. While clone is used to duplicate an object, including its internal state, dup typically uses the class of the descendant object to create the new instance. When using dup , any modules that the object has been extended with will not be copied. If a block is given, it will be used to calculate the size of the enumerator without the need to iterate it see Enumerator size. Here is such an example, with parameter passing and a sizing block: Typically, this method is overridden in descendant classes to provide class-specific meaning. This is used by Hash to test members for equality. For objects of class Object, eql? Subclasses normally continue this tradition by aliasing eql? A RuntimeError will be raised if modification is attempted. There is no way to unfreeze a frozen object. See also Object frozen?. This method returns self. Objects of the following classes are always frozen: Fixnum , Bignum , Float , Symbol. This function must have the property that a. The hash value is used along with eql? Any hash value that exceeds the capacity of a Fixnum will be truncated before being used. The hash value for an object may not be identical across invocations or implementations of Ruby. If you need a stable identifier across Ruby invocations and implementations you will need to generate one with a custom method. User defined classes should override this method to provide a better representation of obj. When overriding this method, it should return a string whose encoding is compatible with the default external encoding. String arguments are converted to symbols. The part of the variable name should be included for regular instance variables. Throws a NameError exception if the supplied symbol is not valid as an instance variable name. The variable does not have to exist prior to this call. If the instance variable name is passed as a string, that string is converted to a symbol. Note that simply defining an accessor does not create the corresponding instance variable. This is the case for immediate values and frozen string literals. Immediate values are not passed by reference but are passed by value: If the all parameter is set to false, only those methods in the receiver will be listed.

Chapter 8 : Object Creation Under Windows NT4 and Windows : Page 2

By bringing dynamic object creation into the core of the language with new and delete, you can create objects on the heap as easily as making them on the stack. In addition, you get a great deal of flexibility.

Creating Data Objects Data objects are used by the destination application â€" either the client or the server. A data object in the destination application is one end of a connection between the source application and the destination application. A data object in the destination application is used to access and interact with the data in the data source. There are two common situations where a data object is needed. The first situation is when data is dropped in your application using drag and drop. The second situation is when Paste or Paste Special is chosen from the Edit menu. In a drag-and-drop situation, you do not need to create a data object. A pointer to an existing data object will be passed to your OnDrop function. This data object is created by the framework as part of the drag-and-drop operation and will also be destroyed by it. This is not always the case when pasting is done by another method. For more information, see Destroying Data Objects. If the application is performing a paste or paste special operation, you should create a COleDataObject object and call its AttachClipboard member function. This associates the data object with the data on the Clipboard. You can then use this data object in your paste function. Destroying Data Objects If you follow the scheme described in Creating Data Objects , destroying data objects is a trivial aspect of data transfers. The data object that was created in your paste function will be destroyed by MFC when your paste function returns. If you follow another method of handling paste operations, make sure the data object is destroyed after your paste operation is complete. Until the data object is destroyed, it will be impossible for any application to successfully copy data to the Clipboard. Creating Data Sources Data sources are used by the source of the data transfer, which can be either the client or the server side of the data transfer. A data source in the source application is one end of a connection between the source application and the destination application. A data object in the destination application is used to interact with the data in the data source. Data sources are created when an application needs to copy data to the Clipboard. A typical scenario runs like this: The user selects some data. The user chooses Copy or Cut from the Edit menu or begins a drag-and-drop operation. The selected data is inserted into the data source by calling one of the functions in the COleDataSource:: The application calls the SetClipboard member function or the DoDragDrop member function if this is a drag-and-drop operation belonging to the object created in step 3. These sample programs provide a good example of how to implement these concepts. One other situation in which you might want to create a COleDataSource object occurs if you are modifying the default behavior of a drag-and-drop operation. For more information, see the Drag and Drop:

DOWNLOAD PDF 2.2 OBJECT CREATION AND DESTRUCTION

Chapter 9 : Object Lifetime: How Objects Are Created and Destroyed (Visual Basic) | Microsoft Docs

The new home for Visual Studio documentation is Visual Studio Documentation on www.nxgvision.com. The latest version of this topic can be found at Data Objects and Data Sources: Creation and Destruction.

The default value of numeric types is zero, the default value of bool is false, and the default char value is the null character. The default value of a struct type is aggregated by the default values of the fields of the struct. Instance variables fields in classes are, however. This is confusing, and it may easily lead to errors if you forget the exact rules of the language. Initializers are static code, and from static code you cannot refer to the current object, and you cannot refer to other instance variables. You should write one or more constructors of every class, and you should explicitly initialize all instance variables in your constructors. By following this rule you do not have to care about default values. So how do we recognize constructors? The answer is given in first two bullet points above: A constructor has the same name as the surrounding class, and it specifies no return type. Overloading takes place if we have two constructors or methods of the same name. Overloaded constructors are distinguished by different types of parameters. Overload resolution takes place at compile time. It means that a constructor used in new C The special delegation mentioned in bullet point four is illustrated by the difference between Program In the latter, the two first constructors activate the third constructor. The third constructor in Program As already stressed, I recommend that you always supply at least one constructor in the classes you program. In that case, there will be no parameterless default constructor available to you. You can always, however, program a parameterless constructor yourself. The philosophy is that if you have started to program constructors in your class, you should finish the job. The three constructors reflect different ways to initialize a new bank account. They provide convenience to clients of the BankAccount class. Just count the lines and compare. Make sure to program your constructors like in Program We also show and emphasize the constructors in the Die class, which we meet in Program Below, in Program Notice that the second Die constructor creates a new Random object. It is typical that a constructor in a class instantiates a number of other classes, which again may instantiate other classes, etc. A copy constructor can be recognized by the fact that it takes a parameter of the same type as the class to which it belongs. Object copying is an intricate matter, because we will have to decide if the referred object should be copied too shallow copying, deep copying, or something in between, see more details in Section It is sometimes useful to have a constructor that creates an identical copy of an existing object In Program Notice that the Random object is shared between the original Die and the copy of the Die. This is shallow copying. The use of copy constructors is particularly useful when we deal with mutable objects Objects are mutable if their state can be changed after the constructor has been called. It is often necessary to copy a mutable object. Because of aliasing, an object may be referred from several different places. If the object is mutable, all these places will observe a change, and this is not always what we want. Therefore, we can protect against this by copying certain objects. The observation from above is illustrated by means of an example - privacy leak - in Section Class instances are objects. Class variables static fields do not belong to any object. They belong to the class as such, but they can be used from instances of the class as well. Class variables can be useful even in the case where no instances of the class will ever be made. Therefore we will need other means than constructors to initialize class variables in C. Initialization of a class variable of type T takes place at class load time Via the default value of type T Via the static field initializers Via a static constructor Initialization of class variable static fields v.