Veritas Framework

NULL |

Documentation File

Ilya Doroshenko

2019

Оглавление

[Terms and Usage Conditions 2](#_Toc7724820)

[Veritas Framework 2](#_Toc7724821)

[EngineCommons.h 2](#_Toc7724822)

[Object-oriented programming within C and default constructor invocation 3](#_Toc7724823)

[Class.h, Class table, static linkage 4](#_Toc7724824)

[Declaration of class. 4](#_Toc7724825)

[Class realization, late bound methods 6](#_Toc7724826)

[Inheritance, Polymorphism and Encapsulation 7](#_Toc7724827)

[Making and object, creating an instance of a class 7](#_Toc7724828)

[Constructor 8](#_Toc7724829)

[Default containers, Exceptions, and other useful stuff 8](#_Toc7724830)



# Сердитое лицо со сплошной заливкой со сплошной заливкойTerms and Usage Conditions

1. This Framework, as well as all the products are designed to be used, specifically on Windows. No warranty is provided if used on other systems and Windows version before Windows Vista.
2. Minimal system requirements:
   1. Windows Vista or higher;
   2. Command Prompt (cmd);
3. Framework was built on MSVC-15, using ANSI-C (99) standard. No external Graphics libraries used.
4. Installation: just copy files and include .h files to your project

# Veritas Framework

## EngineCommons.h

For clearer coding I needed some packages to store objects and functions. Because C doesn’t have anything close to objects and classes in Objected oriented languages, I had to make classes myself.

For this EngineCommons.h was made:

#define \_\_rclass(x) extern const void\* x; struct x

#define class \_\_rclass(c\_class)

#define \_\_xctab(x) \_ ## x

#define \_\_rctab(x) \_\_xctab(x)

#define \_\_xconcat(x,y) x##y

#define \_\_rconcat(x,y) \_\_xconcat(x,y)

#define ctab \_\_rconcat(\_, c\_class)

#define virtual(x) \_\_rconcat( x, \_\_rctab(c\_class))

#define Destructor void\* \_\_rconcat(c\_class,\_dtor)

#define Constructor void\* \_\_rconcat(c\_class,\_ctor)

#define meth \_\_rconcat(c\_class, \_method)

#define \_private \_\_rconcat(c\_class,\_private)

#define vftb \_\_rconcat(c\_class,\_vftb)

// Type string

#define \_\_xtypestr(x) #x

#define \_\_rtypestr(x) \_\_xtypestr(x)

This region specifies all the crucial automation for mangling structures and methods to make them more appealing. For most parts they are adding “\_(class name)” at the end of each object they were called.

Construstor and Destructor fields specify automatic construction members and their prototypes.

This framework is provided with some of predefined values and some Assembly data types for faster approach. MaxInt is specified according to execution mode to exactly fit a processor register.

typedef unsigned char Byte;

typedef \_Bool bool;

typedef unsigned short Word;

typedef unsigned int DWord;

#if \_WIN64

typedef unsigned long long MaxInt;

#else

typedef unsingned int MaxInt;

#endif

#define M\_PI 3.14159265358979323846

#define true 1

#define false 0

[Укажите здесь источник.]

// Inheritance helper

#define inherits(x) struct x \_base

#define account(x) struct c\_class \*this = x

#define base (this->\_base)

// Private Handling

#define privatev(...) Byte virtual(\_\_internal\_prtb)[ sizeof( struct \_private{ \_\_VA\_ARGS\_\_ } )]

#define private (\*(struct \_private\*)(this->virtual(\_\_internal\_prtb)))

// Method Handling

#define methods(...) struct vftb { \_\_VA\_ARGS\_\_ }\*method

#define constructMethodTable(...) struct vftb meth = { \_\_VA\_ARGS\_\_ }

#define assignMethodTable(x) ((struct c\_class \*)(x))->method = &meth

// Class construction handling

#define ENDCLASSDESC const struct Class ctab = { sizeof(struct c\_class),\

.ctor = \_\_rconcat(c\_class,\_ctor),.dtor = \_\_rconcat(c\_class,\_dtor),.typestring = \_\_rtypestr(c\_class)}; \

const void\* c\_class = &ctab;

Those are macros for handling initialization of class and its proper functioning without writing too much verbose and boilerplate code.

This file must be the last included into each class description file (.h) as it erases previously defined class, it helps linker and eliminates errors. This conclusion was made, because #define can’t be defined separately, so each time developer will create class, he won’t need to erase previous instances manually.

## Object-oriented programming within C and default constructor invocation

Because C compiler has no functionality of C++ part of it such as mangling, virtuality and inheritance, as well as hidden function calls must be made manually.

File **New.h:**

// creates instance of class with launching constructor with parameters

void\* new (const void\* type, ...);

// deletes a class instance

void delete(void\* item);

unsigned int sizeOf(const void\* \_self);

const char\* stringOf(const void\* \_self);

This file declares prototypes of functions, which functionality can be seen at **Class.c:**

First one provides space for class object and invokes its constructor, which goes under keyword Constructor.

Delete does essentially opposite: calls destructor under Destructor keyword and then frees the memory object occupies. sizeOf returns size of object which is described in class table (more on that in Class.h and ENDCLASSDESC). stringOf returns a stringed class name, which can be used to output some Exception.

## Class.h, Class table, static linkage

struct Class

{

unsigned size; // size for construction

void\* (\*ctor) (void\* self, va\_list \*app); // constructor, needs pointer on self class and arguments 4 polymorf

void\* (\*dtor) (void\* self); // destructor to be able to reverse what c-tor has done

const char\* typestring;

};

This structure contains pointers to Constructor and Destructor functions, size of class structure and its typestring for stringOf().

This structure must be statically defined at the end of each class realization file (typically .c files) to declare a type which represents class and being able to call new() and delete(). All the automation of this process is done in ENDCLASSDESC macro. Constructor and destructor may be specified manually, but if Constructor and Destructor are specified ENDCLASSDESC does everything by default.

## Declaration of class.

To declare class first thing to do is defining c\_class macro, then follow it with keyword class:

StringStream.h

#pragma once

#include "EngineCommons.h"

#define c\_class StringStream

class

{

const void\* \_class;

methods(

struct c\_class\* (\*Append)(void\* self, const char\* str);

struct c\_class\* (\*AppendI)(void\* self, const long num);

char\* (\*str)(const void\* self);

char\* (\*EndStr)(void\* self);

);

privatev(

char\* inStr;

DWord len;

);

};

Here is the example of declaration everything correct:

1. #include "EngineCommons.h" // Clears all the previous instances of c\_class for linker
2. Const void\* \_class; This one is a must, because new() writes information about class inside this pointer by assigning \_class value with reference to static class table (ENDCLASSDESC)
3. Methods are all specified inside methods() macro. Essentially this creates a pointer to virtual table inside each class, so all the methods can be linked to each object.
4. Private variables are written within privatev(). Only one privatev is available for each class and no private methods are available (because those are indistinguishable from private static methods in C)

## Class realization, late bound methods

StringStream.c

struct c\_class\* \_Append\_s(void\* self, const char\* str, const DWord length)

{

}

struct c\_class\* \_Append(void\* self, const char\* str)

{

}

struct c\_class\* \_AppendI(void\* self, const long num)

{

}

char\* \_str(const void\* self)

{

}

char\* \_EndStr(void\* self)

{

}

constructMethodTable

(

.Append = \_Append,

.AppendI = \_AppendI,

.str = \_str,

.EndStr = \_EndStr

);

Constructor(void\* self, va\_list\* ap)

{

struct c\_class\* this = self;

assignMethodTable(this);

return this;

}

Destructor(void\* self)

{

struct c\_class\* this = self;

return this;

}

ENDCLASSDESC

This is an example of class realization:

1. Constructor and Destructor are better to be last, because then no forward declaration is needed.
2. To bind methods to a class a must is to create a method table first, so function pointers will fill with correct function. This is done because C does not allow function to be passed directly, but only by assigning a function pointer. Then this table is bound to the class by assignMethodTable(this);
3. Void\* self is noticeable in each method. This is because C++ implicitly passes this pointer in ecx(rcx) register. C does not, so it needs to be done explicitly. It can be however be replaced by struct c\_class\* this in the first place, but local copy of this is safer for multithreaded application. Later implementations of default classes have macro account(self), which does the same thing as direct assign.
4. Notice that number of methods in the table are less than represented in file. This is done, because those methods are private in common sense: they are invisible from code and to the child classes, but can be used inside class methods themselves.

## Inheritance, Polymorphism and Encapsulation

Encapsulation is done using private and description files, it is not full, because if you declare two methods in different classes they will collide in linker, creating an error. To overcome this virtual() macro exist. Basically, it mangles functions, but can also be used to static variables. To access private variables of class, use private. keyword.

Polymorphism is achieved through virtual() and using function pointers, because function pointer may be bound with any other function with the same prototype. See Maze3D.c for more information on this

Inheritance is a bit of a trick: in C++ child class contains a parent class, it is just pasted at compile time. None of this is available in C. So it has to be manually pasted into class description with macro inherits(), which also defines base keyword.

Inside constructor and destructor there must be an execution of base constructor. Here is a full example:

class

{

// Notice! No const void\* \_class; because parent class already has it

inherits(VeritasEngine); //here goes typename (what was in c\_class)

// something else

};

Inside .c:

Constructor(void\* self, va\_list \*ap)

{

struct c\_class \*this = ((struct Class\*)VeritasEngine)->ctor(self, ap);

////

Destructor(void\* self)

{

struct c\_class \*this = ((struct Class\*)VeritasEngine)->dtor(self);

## Making and object, creating an instance of a class

To make an instance of a class its header needs to be included, then write struct SomeClass\* scp = new(SomeClass, constructor first arg,…); Example:

#include "Maze3D.h"

#include "Exception.h"

int main(void)

{

struct Maze3D \*Game = new(Maze3D);

if (Game->\_base.method->SetupScreen(Game, 320, 180, 4, 4))

Game->\_base.method->Start(Game);

}

1. Constructor of Maze3D has no va\_arg;
2. Typedef is not possible, because name of the struct is automatically reserved for type description.
3. To access methods, use ->method->
4. In example base method call is shown, because Maze3D is a child of VeritasEngine. Verbosity of this limits usage to some extent: for example you have 6 classes inherited from each other, then it needs to be written 5 times \_base to access a method of first parent. It is possible to make usage easier using macros or rebinding (as I’ve done it in VeritasEngine), but it’s a bit of a stretch.

## Constructor

To make constructor without arguments just don’t use va\_list\*. Else call va\_arg on each input argument, the downside to this is you must remember in which order those arguments are:

Constructor(void\* self, va\_list \*ap)

{

account(self);

assignMethodTable(this);

this->Size = va\_arg(\*ap, DWord);

this->Capacity = va\_arg(\*ap, DWord);

this allows polymorphism to some extent, without loosing automatization. Downsides: it is complex and if there are no arguments those values will be probably filled with trash.

## Default containers, Exceptions, and other useful stuff

Veritas Framework provides some default containers such as Queue, BitField, BitStack. Those are primarily used to work with massive amounts of bits but can be used as regular C++ STL containers. More documentation on them in later revisions.