Introduction to Class-C

by Milan Toth

Preface

Class-C

What? Another Object-Oriented C language? Why do we need another?

C is a really simple and minimalistic language, that's why we love it. But C++ and Objective-C became language monsters with over a 100 keywords and confusing features, and more confusing libraries. Just because you need classes and objects, you don't have to overcomplicate things.

Minimalism

The idea behind Class-C is very simple : take your existing C code, put global variables and functions in a block and global variables become member variables, functions become methods. Class is ready.

Speed

Explicit method calls are translated to direct c function calls. Dynamic casting is just a few cpu cycles.

Class Merging

Instead of hierarchical inheritance, merge classes freely in a flat structure. If you have overridden a method, but for some reason you need the original one, you can call the original method on your object with an explicit call.

Productivity

The Class-C compiler generates header files automatically, you can concentrate on the implementation. With CLObject as a base class, memory management becomes really simple, just retain/release ownership to objects.

Philosophy

Class-C encourages openness and simplicity. Clean, small, open and well documented code is the safe and reliable code.

License

Creator

Class-C is developed by Milan Toth
milgra@milgra.com
http://milgra.com

QuickLook

Okay, enough talk, how does it look like? In comparison with other languages?

1. Class declaration

objective-c	c++	Class-C

BigPerson.h
 	
`@interface BigPerson : Person`
`{`
`NSString* name;`
`}`
`- ( void ) sayHello;`
`@end`
 	
BigPerson.m
 	
`#import "BigPerson.h"`
 	
`@implementation BigPerson`
 	
`-( void ) sayHello`
`{`
`printf( "Hello" );`
`}`
	BigPerson.h
	
	`class BigPerson : public Person`
	`{`
	`public :`
	`void sayHello( );`
	`private :`
	`string name;`
	`}`
	
	BigPerson.m
	
	`#include "BigPerson.h"`
	
	`BigPerson::sayHello( )`
	`{`
	`printf( "Hello" );`
	`}`
		BigPerson.clc
		
		`BigPerson:Person`
		`{`
		`CLString* name;`
		`void sayHello( )`
		`{`
		`printf( "Hello" );`
		`}`
		`}`

2. Allocation/Instantiation/Deallocation

objective-c	c++	Class-C

`NSString* name = [ [ NSString alloc ] initWithCString : "milan" ];`
 	
`[ name release ];`
	`string* name = new string( );`
	
	`name.setCString( "milan" );`
	
	`free( name );`
		`CLString* name = CLString:alloc( );`
		
		`name.initWithCString("milan");`
		`name.release( );`

3. Polymorphism

objective-c	c++	Class-C

`@protocol definitions`
 	
`@interface BigPerson:Person < PTalk >`
	`classes with virtual functions`
	
	`class BigPerson : public Person`
	`: public ITalk`
		`class merging`
		
		`BigPerson:Person:ITalk`

5. Instance method call

objective-c	c++	Class-C

`[ instance method : arg ];`	`string.method( arg );`	`string.method( arg );`
		`string:method( arg );`
		`CLString:method( string, arg );`

4. Static/Class method call

objective-c	c++	Class-C
`[ NSString method ]`	`[ stringClass method ]`	`CLString:method( NULL );`

QuickStart

Class-C compiler is a source-to-source compiler in its actual state, it produces C source files.

1. Compile the compiler

Download clcc.c, compile it with your favorite c compiler. With gcc it looks like this :

```
gcc clcc.c –o clcc
```

The binary "`clcc`" appeared in the actual folder. Now you have a working Class-C compiler on your system. You might want to put it under a default system path to make it accessible from anywhere.

2. Compile your first class

Download FirstClass.clc to the same folder, and compile it with clcc :

```
clcc FirstClass
```

Two files with the default names clcsrc.h and clcsrc.c appeared in the folder, these are the compiled c sources of FirstClass.clc.

3. Include the compiled c source in a c file, use FirstClass, and create a binary.

Download main.c to the same folder, and compile your sources with your favorite c compiler. With gcc :

```
gcc main.c clcsrc.c –o firstprog
```

main.c inlcudes clcsrc.h and instantiates FirstClass, calls one of its methods, cleans up and exits. Note that main.c uses bridged Class-C calls, these expression looks different in Class-C, it's just for the quickstart.

4. Run your first program

On Unix-like systems, type

```
./firstprog
```

The output should be :

```
Hello, this is FirstClass!!!
```

Language Elements

Class declaration

Files containing a Class-C class must have the extension ".clc". Only one class per file is allowed.
In a Class-C class file the top-level block contains the class declaration, the class name is before the opening brace.

[*Example : FirstClass.clc*

```
FirstClass
{

}
```

---end example]

Inheritance

You can attach base classes to your class with the explicit accessor ":". Base class count is not limited, but member variable names must be unique.

[*Example : FirstClass.clc*

```
FirstClass:OtherClass:CLObject
{

}
```

---end example]

Polymorphism

Because of multiple inheritance, Class-C doesn't have a special interface/protocol definition, just use an other class containing the desired methods.

[*Example : FirstClass.clc*

```
FirstClass:OtherClass:CLObject:IClickable
{

}
```

---end example]

Member variables

Variables declared in the top-level block of a class file are member variables.

[*Example : FirstClass.clc*

```
FirstClass
{
      int counter;
}
```

---end example]

Member functions

Functions defined in the top-level block of a class file are member functions.

[*Example : FirstClass.clc*

```
FirstClass
{
      int setCounter( int theCount )
      {
            counter = theCount;
      }
}
```

---end example]

The self pointer

In a class definition, you can use the built-in "self" pointer to point to the current object.

[*Example :*

```
void callOtherObject( )
{
      otherObject.doSomethingWithMe( self );
}
```

---end example]

The allocator function

To allocate an object, you have to use an explicit call to the "alloc" member function of the wanted class.
If you use CLObject as a base class, then you must initalize the object immediately, to set retain count to 1.

[*Example :*

```
CLString* string = CLString:alloc( );
string.init( );
```

---end example]

Deallocating objects

To deallocate a standalone Class-C object, you use the `free_object( )` function, but before that you might want to call the destructor of your object manually. (Unmanaged objects)
If you use CLObject as a base class, you have to use the `release( )` member function instead, which, if retain count of the object is 0, calls the destructor and frees memory in one step. (Managed objects)

[*Example :*

```
simpleObject.destruct( );     // calls destructor on non-CLObject
free_object( simpleObject ); // frees memory for non-CLObject

string.release( );            // calls destructor and frees memory
```

---end example]

Constructor methods

In Class-C constructor/initializer methods are just user-defined custom methods. Just be sure to initialize all member variables (and possibly base classes) in them to avoid unwanted behaviour. The return type can be anything.

[*Example :*

```
	void initInACustomWay( )
	{
		CLObject:init( self );
		myCArray = malloc( sizeof( int ) * 10 );
	}
```

---end example]

Destructor methods

In Class-C destructor methods are just user-defined custom methods. If you use CLObject as a base class, then you have to use the "destruct" member function as the destructor function. You have to override that function and place your destructor code there. "release" member function will call "destruct" member function when retain count is 0.

[*Example :*

```
	void destruct( )
	{
		free( myCArray );
		CLObject:destruct( self );
	}
```

---end example]

Member variable access

Access member variables with the dot "." accessor.

[*Example :*

```
	myObject.text;
```

---end example]

Within the scope of a class, you don't have to use the "self" accessor to refer to a local member variable.

[*Example :*

```
	int myVariable;
	void myFunc( )
	{
		myVariable = 5;
	}
```

---end example]

Member function calls

There are four ways to call a member function in Class-C, and it is very important to understand the difference between them to write efficient code.

Local Member Function Calls

Within the scope of a class, you can call local member functions just like you call a global c function, without the "self" accessor.

[*Example :*

```
void myFunc1( )
{

}
void myFunc2( )
{
      myFunc1( );
}
```

---end example]

They are translated to direct c function calls, and no casting is happening during the call - they are high speed calls.

Normal member function calls

They are the general member function calls, you have to use the dot accessor "." before the method name. They are cast-safe, a call to casted instance's member function is identical to the call to the original instance's member function.

[*Example :*

```
string.appendCString( "something" );
```

---end example]

At least three structure is accessed during a normal member function call - they are medium speed calls.

Class-Explicit calls

In a class-explicit call you define the class of the method explicitly. You have to use the explicit accessor ":" before the method name. The first parameter of the call have to be an object or NULL, and the function uses that object as the member variable container. The only exception is the allocator call, it needs no first parameter.

[*Example :*

```
CLString:appendCString( string , "something" );
```

---end example]

Use class-explicit calls when you want to call member functions of a class directly from anywhere, or when you want to access the original version of an overridden member function. The most common usage is the base class initializer from class initializer.

[*Example : FirstClass.clc*
```
	void init( )
	{
		CLObject:init( self );
	}
```
---end example]

They are translated to direct c function calls, but the first parameter is always casted to the declared class. They are medium speed calls.

Instance-Explicit calls

In an instance-explicit call the class of the instance defines the class of the method explicitly. You have to use the explicit accessor ":" before the method name.

[*Example :*
```
	string:appendCString( "something" );
```
---end example]

If string's type is CLString, then the above call is identical to :

[*Example :*
```
	CLString:appendCString( string , "something" );
```
---end example]

but no type casting is applied to the first parameter.

They are translated to direct c function calls, and no casting is happening during the call - they are high speed calls.

Comparison of member function calls

As you can see, local member function calls and instance-explicit calls are the fastest calls, they have the speed of pure c function calls, the other two are a little bit slower, but they are still very fast. You can write your full program without instance-explicit calls, but if you want to make high-performance code you might want to consider using instance-explicit calls where ever possible.

But be careful, instance-explicit calls are not cast-safe, and they can cause a lot of headache if not used properly! Consider the following setup :

[*Example :*
```
	CLCustomString* custom = CLCustomString:alloc( );
	custom.init( );
	CLString* string = ( CLString* ) custom;	// casts to CLString

	// calls CLCustomString:appendCString method
	string.appendCString( "something" );
```

```
      // calls CLString:appendCString method
      string:appendCString( "something" );
```
---end example]

Dynamic casting

Class type casting has the same syntax as normal type casting.

[*Example :*

```
CLString* string = ( CLString* ) custom;    // cast to CLString
```

---end example]

Class Name and Class Id of an instance

It can be useful to know the original class and class id of an instance. To get it :

[*Example :*

```
struct Instance* original = object._components[0];
char* className  = original->_class->className;
void* classId    = original->_class->classId;
```

---end example]

className is a c string, classId is a word-length number, the memory address of the class descriptor structure.

For-each iteration in Core Library containers

The two Core Library containers, CLDataList and CLObjectList can be iterated through with a java-like iteration syntax, but only with their true types

[*Example :*

```
for ( CLObject* object : objectList )
{

      CLString* string = ( CLString* ) object;

}

for ( void* data : dataList )
{

      long number = ( long ) data;

}
```

---end example]

The Core Library

The Core Library contains CLObject which should be the base class of all Class-C classes, and a few classes that became the bare minimum for me to a faster development. Feel free to expand and modify them for your own needs, and if you think you put in something really necessary, feel free to send me a pull request on github.

CLObject

Class containing methods and variables for reference counted memory management and for standard literal description.

CLObject
`unsigned long` **`retainCount;`**
the retain count of the object
`void` **`init();`**
initializer, set retain count to 1
`void` **`destruct();`**
destructor, empty function, should be overridden
`void` **`retain();`**
increases retain count with 1
`void` **`release();`**
decreases retain count with 1, destructs and deallocates object if needed
`void` **`describe();`**
prints a description of the object to the standard output

CLThread

Encapsulates a POSIX thread and mutex

CLThread:CLObject
`char` **`alive;`**
A flag indicating the thread state.
`pthread_t` **`thread;`**
The POSIX thread
`pthread_mutex_t` **`mutex;`**
The POSIX mutex.
`void` **`init`**`();`
initializer, sets retain count to 1, initializes mutex
`void` **`destruct`**`();`
destructor
`void` **`start`**`();`
starts execution
`void` **`run`**`();`
empty function, should be overridden with the executable code

CLDataList

CLDataList is a linked list containing general data. CLDataList knows nothing about the data, so memory management of the data has to be done manually outside of the data list.

CLDataList:CLObject
`struct CLLink*` **`head;`**
starting link element
`struct CLLink*` **`last;`**
ending link element
`unsigned long` **`length;`**
length of list
`void` **`init`**`();`
initializer, sets retain count to one, length to 0, head and last to NULL
`void` **`destruct`**`();`
destructor
`void` **`addData`**`( void* theData );`
adds data

`void addDatasInDataList( CLDataList* theDataList );`
adds all data from a data list
`void removeData( void* theData );`
removes data
`void* removeDataAtIndex( unsigned long theIndex );`
removes data at given index
`void removeAllDatas();`
resets list
`void* dataAtIndex( unsigned long theIndex );`
returns data at given index
`long long indexOfData( void* theData );`
returns index of given data
`void* firstData( )`
returns data at index 0
`void* lastData( )`
returns data at last index

CLObjectList

CLObjectList is a linked list containing objects with CLObject base classes. CLObjectList retains/releases objects on addition/removal, makes memory management easier, but because it accepts casted objects only, and retain/releases continuously, it is slower than CLDataList.

`CLObjectList:CLObject`
`struct CLLink* head;`
starting link element
`struct CLLink* last;`
ending link element
`unsigned long length;`
length of list
`void init();`
initializer, sets retain count to one, length to 0, head and last to NULL
`void destruct();`
destructor
`void addObject( CLObject* theObject );`

adds object to the list
`void addObjectsInObjectList( CLObjectList* theObjectList );`
adds all objects from object list to the list
`void removeObject( CLObject* theObject );`
removes object from the list
`void removeObjectAtIndex( long long theIndex );`
removes object at given index
`void removeAllObjects();`
resets list
`CLObject* objectAtIndex( unsigned long theIndex );`
returns object at given index
`long long indexOfObject( CLObject* theObject );`
returns index of given object
`CLObject* firstObject( );`
returns object at index 0
`CLObject* lastObject( );`
returns object at last object

CLString

CLString is a linked list containing characters. It has direct file read/save functionality, basic path management, and basic string functions.

CLString:CLObject
`struct CLChar* head;`
starting character
`struct CLChar* last;`
last character
`unsigned long length;`
length of string
`void init();`
initializer, sets retain count to one, length to 0, head and last to NULL
`void initWithString( CLString* theString);`
initializer, same as init but fills up string with theString

`void initWithCString(char* theString);`
initializer, same as init but fills up string with theString
`void destruct();`
destructor
`void appendCharacter( char theCharacter );`
appends one character to the end of the string
`void appendString( CLString* theString );`
appends string to the end of the string
`void appendCString( char* theString );`
appends c string to the end of the string
`void removeAllCharacters();`
resets string
`void readFile( CLString* thePath );`
reads up given file into the string
`void readFilePointer( FILE* thePointer);`
reads up file contents into the string
`void writeToFile( CLString* thePath );`
writes string into given file
`void writeToFilePointer(FILE* thePointer);`
writes string into given file
`CLString* stringWithLastPathComponent( );`
returns last path component of string in a new CLString
`CLString* stringByRemovingExtension();`
returns string without the extension in a new CLString
`CLString* stringByRemovingLastPathComponent();`
returns string without the last path component in a new CLString
`long long indexOfString( CLString* theString );`
returns index of given string. result is -1 if string is not found.
`char equals( CLString* theString );`
compares given string with the string. result is 1 is two string is identical, 0 if not
`char* cString();`
returns the c string representation of string. the result has to be deallocated.

CLStringDataList

This class is an utility class for operations involving CLString and CLDataList instances. Instantiation is not necessary, class method's don't need member variables.

`CLStringDataList:CLObject`
`long long` **`indexOfString`** `( CLString* theString , CLDataList* theDataList )`
returns index of given string in the given data list. result is -1 if string is not found
`CLDataList*` **`splitStringByCharacter`**`( CLString* theString , char theCharacter );`
Splits the given string at given characters, to a CLDataList

CLStringObjectList

This class is an utility class for operations involving CLString and CLObjectList instances. Instantiation is not necessary, class method's don't need member variables.

`CLStringObjectList:CLObject`
`long long` **`indexOfString`**`( CLString* theString , CLObjectList* theObjectList );`
returns index of given string in the given data list. result is -1 if string is not found
`void` **`removeString`**`( CLString* theString , CLObjectList* theObjectList );`
removes string from the object list
`void` **`removeStrings`**`( CLObjectList* theStringList , CLObjectList* theObjectList );`
removes all strings in given object list from the second object list
`void` **`addStringAsUnique`**`( CLString* theString , CLObjectList* theObjectList );`
adds given string as an unique string to the object list
`void` **`addStringsAsUnique`**`( CLObjectList* theStringList , CLObjectList* theObjectList );`
adds all strings from the object list as unique strings to the object list
`CLObjectList *` **`splitStringByCharacter`**`( CLString* theString , char theCharacter );`
Splits the given string at given characters, to an object list

Using Class-C code in C/C++/Objective-C

Just include the compiled Class-C source header - clcsrc.h by default, and use the bridge functions to do whatever you want with the help of the mapping table below :

Class-C code	C code
`CLString* string;`	`struct CLString* string;`
`CLString:alloc( );`	`CLString_alloc( );`
`CLString:init( string );`	`CLString_init( string );`
`CLString:release( string );`	`CLString_release( string );`
`string.release( )`	`CLString_release( string );`

Do as less as possible in this pseudo-Class-C-code, you better put your whole control logic in Class-C classes, and do only the allocation-instantiation-deallocation of the main control logic in C/C++/Objective-C.

How to develop in Class-C?

You can develop Class-C on every platform with a C compiler.

1. Put the compiler under one of your PATH locations

On UNIX-like systems just copy it under /usr/bin or /usr/local/bin

2. Use your favorite c code editor/IDE

You can go on in an old school way, with a plain text editor and command line compiling, or you can use your favorite IDE with C syntax highlighting. If you IDE supports build phases and script running, add Class-C file compiling as the first phase. If build phases aren't supported, you have to create a compile script first, or do the compiling manually before building.

3**. Always check compiler output**

The compiler output is your best friend, always check it before compiling the final project. You will see which classes are not found, you will get possibly unincluded but needed classes, and so on.

4**. Debug your code**

Your IDE/Terminal will show the problem in the compiled C source, and you have to find it in the Class-C source. The method name in the compiled source tells you which Class-C class and what method contains the problematic line.

The compiler

You add the classes you want to compile without switches. Included classes will be recursively added to the compile chain, so it is enough to add the top-level classes only.

Other switches :

-p : define a class lookup path
-o : define the output file prefix, two files will be created with this prefix, one ".c" and one ".h"

Example :

1. Compiling the dynamics engine with all core lib paths, dynamics.c and dynamics.h as output

```
clcc Compiler -o /Users/milgra/CodeBase/Dynamics/dynamics -p /Users/
milgra/CodeBase/Dynamics/ -p /Users/milgra/CodeBase/Library/CoreLib/ -p /
Users/milgra/CodeBase/Library/MathLib/ -p /Users/milgra/CodeBase/Library/
PhysicsLib/ -p /Users/milgra/CodeBase/Library/TextLib/ -p /Users/milgra/
CodeBase/Library/GraphicsLib/ -p /Users/milgra/CodeBase/Library/
GraphicsLib/Primitives/
```

Programs written in Class-C

Class-C Compiler

The Class-C compiler was written in a declarative manner : no member variables used with the exception of the Main class, everything runs in its own scope.

DynamicsX

Simple dynamics engine with an openGL visualizer.

Cerebral Cortex for iOS, OSX

My nerve-wrecking ambient/reflex game, it was selected for the Best New Games section in the Mac App Store.

Future of Class-C

I find the source-to-source compiler quite handy, because I cannot compete with the dozens of special c compilers for embedded systems, with clang and others, and in addition, with the logic in c it is very easy to write multi-platform code.

Best practices

One-statement rule

For the sake of clarity, use only one statement per line. Don't overload lines with stuff, no function calls as parameters, and so on.

Standaloneness

Classes, especially core classes must be standalone, they have to work without other classes (with the exception of CLObject of course).

Utility classes

If you need multiple high-level classes for a behaviour, create a utility class (CLStringObjectList for example), and do things in class level explicit methods.

Setter importance

For object members, always write a setter method which deals with null values and releases/ retains old and new values properly, it is a MUST for trouble-free memory management and development.

Storage Classes instead of Maps/Dictionaries

Maps and dictionaries are not really good things - they are too generic. If you want to store a large amount of key-value pairs, use a database, in case of a moderate amount of pairs, create a storage class. If you really really need a map, then you have to write your own container class.

Memory management

If you don't like memory management, you don't like programming

Hatred and Love!!!

Hate hidden things. Hate when you can't see - down to assembly level - what a function does, hate closed frameworks and pre-compiled libs and generic frameworks. Hate plugins. Hate dependencies. Hate unnecessary things.

Love simple, clean, open standalone projects. Love clarity, love simplicity.

Philosophy

Class-C encourages openness. Don't hide your code in compiled libraries, don't hide your comments, let the coders understand what's going on down to assembly level. That's the only way to write safe, reliable code. And that's the best way to teach others and to learn.

Hopefully, Class-C syntax also helps coders to write simple, clean, balanced, symmetric, beautiful classes which are a pleasure to look at and read.

Try to avoid big and general purpose frameworks and libraries in the hope of quick deployment, try to create your own small, minimalistic frameworks instead. Understand how the processor works, always count the possible cpu cycles, whatever you write.