2006

Myth- an extension to C

Greg Rowe

Myth - An extension to C

Greg Rowe

Revision History

Revision 0.1 January 17th, 2005
Initial revision.
Revision 0.2 February 3rd, 2005
Some updates based on comments from Dr. Schreiner.
Revision 0.3 February 15th, 2005
Added references to existing work.
Revision 0.4 February 16th, 2005
Minor updates.
Revision 0.5 March 8th, 2005
Changes based on feedback from Dr. Heliotis.
Revision 1.0 March 24th, 2005
Misc updates and inclusion of grammar.
Revision 1.1 March 26th, 2005
Removed full grammar from appendix, corrected grammar mistakes, revised coverage of tuples.

Abstract

Myth is a programming language that is an extension of C. Myth adds modules, tuples, and bits sets. These features are very helpful in an embedded environment. Many features such as full object support and exceptions are avoided in an effort to keep the language small. Myth will be implemented as a pre-processor that emits C source code.

1. Introduction

Myth is an extension to C. Myth adds modules, interfaces, tuples, and bit sets. The added features are a great benefit to programmers but the language remains small. Myth will be implemented as a preprocessor that emits C source code.

2. Motivation

2.1. Why extend C?

Despite being a relatively low level language C is a very popular programming language for software development for a wide range of applications. C has become a very popular choice in embedded environments partly because of the ease of low level hardware access. C is also very popular in open

source software particularly because, when used carefully, C is extremely portable. No language is perfect, and C could use a few additions to make programming easier.

2.2. Tuples

Myth adds a new data type to C - the tuple. A tuple is very similar to a struct but has one major distinction. A tuple is never named. This frees the programmer from having to explicitly declare a struct. This is particularly useful when returning values from a function. However, tuples do not replace structs. A struct is better suited when there are large numbers of items and where naming each element, and the structure itself, is helpful.

Often when writing programs a function will need to return multiple values. A typical function will return a single result and an error code. It is common in C that functions return an error code and use pointers to return other results to a caller. This is ugly because the arguments to a function no longer serve as just input. It would be better if all arguments were input and multiple return values could be made.

Returning multiple values in C is possible using structs. The technique is rarely used in practice because it is tedious and time consuming to declare the struct. It would be better if multiple values could be returned from a function with little prior delcaration.

C++ and Java both offer exceptions which can be a powerful mechanism for error handling. The implementation for exceptions can be too expensive for many environments. Tuple returns offer a good balance between power and efficiency for error handling.

Error handling is only one example were tuple returns are beneficial. Any situation where two or more values should be returned from a function can benefit from tuple returns. When a function requires a very large number of values to be returned a programmer will probably choose to use a struct (or a module if using the Myth extensions) where the cost of declaration out weighs the benefit of named fields.

S-Lang and Limbo are two languages that offer tuple returns. Spar/Java is an extension to Java that adds a tuple data type. All three languages have similar syntax that makes using tuples easy. Reeuwik and Sips have found that their tuple syntax made their some programs easier to read [ReeuwijkSips2002]. Myth offers a similar syntax.

2.3. Bitsets

When working at the bit level in C it is common to see large headers that use the C pre-processor to define large numbers of bit masks. This style of declaring bit level masks, while popular, is tedious and difficult to maintain. Often dependencies exist between masks and changing one mask requires changing all of the masks. It would be beneficial to have a method of declaring and grouping bit masks. Enums can’t be used for this because enums are a sequential series of integers. The Myth addition of bitsets makes declaring, organizing, and maintaining large numbers of bit masks easier.

C offers bit level access using bit fields in structs. The bitset extension provided in Myth is not closely related to this feature in C. Bitsets in Myth are intended to help with the declaration of constant data. A bit field in C on the other hand deals more with runtime level access.

2.4. Modules and Interfaces

Most programmers would agree that writing modular software is a desirable goal. Language level support to promote modular software would be a good addition to C. Interfaces and modules, as implemented in Myth, promote writing modular code and offer a valuable abstraction with out the overhead of dynamic binding.

Almost all languages have support for modules. Java has a "package" system. Pascal has "units." Object oriented languages such as C++ and Java offer modules in the form of classes as well. C on the other hand has little language support for modules. Using the "static" keyword one can limit the scope of functions and data to a file but beyond file scoping there is no notion of modules in C. The goal of module support in Myth is to promote writing of modular code, not to enforce scoping rules. Module support in Myth in many ways is more simple than module systems in other languages.

3. Partial Myth Grammar

In this section a partial grammar is presented. This grammar is for a grl parser. These grammar rules are to be added to Jeff Lee’s C grammar [Leej].

3.1. Tuples

Example 1. Tuple assignment

```
tuple_arg
: TUPLE ’(’ argument_expr_list ’)’
;

tuple_id
: TUPLE ’(’ identifier_list ’)’
;

assignment_expr
: conditional_expr
| unary_expr assignment_operator assignment_expr
| tuple_id assignment_operator assignment_expr
| tuple_id assignment_operator tuple_arg
| unary_expr assignment_operator tuple_arg
;
```

Example 2. Tuple return

```
jump_statement
: GOTO IDENTIFIER ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN expr ';'
| RETURN tuple_arg ';'
;
```

Example 3. New Tuple Type

```
type_specifier
: CHAR
| SHORT
| INT
| LONG
| SIGNED
| UNSIGNED
| FLOAT
| DOUBLE
| CONST
| VOLATILE
| VOID
| struct_or_union_specifier
| enum_specifier
| TYPE_NAME
| TUPLE '(' parameter_list ')'
;
```

Example 4. Tuple assignment

```
assignment_expr
: conditional_expr
| unary_expr assignment_operator assignment_expr
| tuple_id assignment_operator assignment_expr
| tuple_id assignment_operator tuple_arg
;
```

3.2. Bitsets

Example 5. Bitset declaration

```
bitset_decl
: BITSET IDENTIFIER '{' bitset_identifiers '}'
```

```
;

bitset_identifiers
: bitset_identifier
| bitset_identifier  bitset_identifiers
;

bitset_identifier
: IDENTIFIER ';'
| IDENTIFIER ':' range_list ';'
;

range_list
: range_item
| range_item ',' range_list
;

range_item
: CONSTANT RANGE_SEPERATOR CONSTANT
;
```

3.3. Modules and Interfaces

Example 6. Module grammar

```
module_decl
: MODULE IDENTIFIER '{' module_body '}'
| MODULE IDENTIFIER IMPLEMENTS identifier_list '{' module_body '}'
;

module_body
: self_decl
| self_decl module_function_decls
| module_function_decls self_decl module_function_decls
| module_function_decls self_decl
;

self_decl
: SELF '{' '}'
| SELF '{' struct_declaration_list '}'
;

module_function_decls
: module_function_decl
| module_function_decl module_function_decls
;

module_function_decl
: declarator ';'
| declaration_specifiers declarator ';'
```

```
;
```

Example 7. Interface grammar

```
interface_decl
: INTERFACE IDENTIFIER '{' module_function_decls '}'
;
```

4. Manual

Myth is an extension of C. All C syntax is allowed. Myth reserves the following keywords: interface, module, bitset, self, tuple, and any struct beginning with 'tuple'.

4.1. Interfaces and Modules

In Myth an interface is a set of functions. Only function declarations are legal within an interface decalaration. A function declaration within a module or interface is allowed to have storage specifiers such as static or extern but these specifiers are ignored. They are left in as part of myth to potentially be used for some unknown future purpose. Modules implement zero or more sets of interfaces and have a state. The state of the module is declared in the 'self' definition within a module. Mythpp will insure that all advertised interfaces are implemented within a module. Mythpp will check the names of the functions but not the types. The types are not checked because function overloading is not legal in C or in Myth. Thus, if the types are incorrect the backend C compiler will catch the mistake.

The motivation behind this feature is to promote the writing of modular code. It also offers the programmer the valuable abstraction of interfaces. Using interfaces it can make writing reusable algorithms easier. Traditional inheritance that allows for dynamic binding is intentionally avoided. Traditional inheritance systems are easy to abuse and all too often are.

All data declared in the 'self' section of a module is implicitly available to all module functions. That is to say that the self pointer is passed into each function. The programmer must explicitly dereference the self pointer when accessing module data.

Example 8. Myth Interface Example

```
interface interface1
{
       int func1();
}
```

```
interface interface2
{
	int func2();
	int func3();
}

module impl2 implements interface2
{
	int func2(){}
	int func3(){}
}

module impl1 implements interface1, interface2
{
	self
	{
		int someData;
		impl2 impl2Instance;
	}


	int func1()
	{
		/* do something here */
	}

	int func2()
	{
		return impl2Instance->func2();
	}

	int func3(){}
}


int algorithm(interface1 intf)
{
	return intf->func1();
}

int main(int argc, char **argv)
{
	impl1 theImpl;
	algorithm(theImpl);
}
```

Example 9. Interface code output

```
typedef struct interface1{
       int (*func1)(void *self);
} interface1_t;

typedef struct interface2{
       int (*func2)(void *self);
       int (*func3)(void *self);
} interface2_t;


typedef struct impl2_self {
} impl2_self_t;
typedef struct impl2{
       impl2_self_t self;
       interface2_t interface2_funcs;
} impl2_t;

typedef struct impl1_self {
       int someData;
       impl2_t impl2Instance;
} impl1_self_t;
typedef struct impl1{
       interface1_self_t self;
       interface1_t interface1_funcs;
       interface2_t interface2_funcs;
} impl1_t;


int impl1_func1(impl1_self_t *self)
{
       /* do something here */
}
int  impl1_func2(impl1_self_t *self)
{
       return self->impl2Instance->func2((void*)self);
}
int  impl1_func3(impl1_self_t *self){}

int impl2_func2(impl2_self_t *self){}
int impl2_func3(impl2_self_t *self){}


int algorithm(void *self, interface1_t *intf){
       return intf->func2(self);
}

int main(int argc, char **argv){
       impl1_t theImpl;
       theImpl.interface1_funcs = {impl1_func1};
       theImpl.interface2_funcs = {impl1_func2, impl1_func3};
```

```
        algorithm((void*)&theImpl.self,
                &theImpl.interface1_funcs);
 }
```

4.2. Tuple Returns

Example 10. Tuple Return Myth Example

```
 tuple(int, int) function(int arg1, int arg2){
        return tuple(0, arg2+2);
 }


 int main(int argc, char **argv){
        int i = 0;
        int j = 5;
        int k1;
        int err;
        tuple(err, k1) = function(i,j);
        if(err){
                /* handle the error */
        }
 }
```

Example 11. Tuple Return Output

```
struct tuple_int_int {
        int arg0;
        int arg1;
}
struct tuple_int_int function(int arg1, int arg2){
        struct tuple_int_int tuple0;

        tuple0.arg0 = 0;
        tuple0.arg1 = arg2+2;
        return tuple0;
 }

 int main(int argc, char **argv){
        struct tuple_int_int tuple1;
        int i = 0;
        int j = 5;
        int k1;
        int err;
        tuple1 = function(i, j);
        err = tuple1.arg0;
```

```
        k1 = tuple1.arg1;
        if(err){
                /* handle the error */
        }
}
```

4.3. Bit Sets

Example 12. Myth Bit Set Example

```
bitset myBitSet {
        flag1;
        flag2;
}

myBitSet mfs;
mfs |= myBitSet_flag1;

biset setWithRange {
        highByte:       31..23;
        lowByte:         0..7;
        nonContiguous:  31..23, 0..7;
}

setWithRange anotherSet;
```

Example 13. Bit Set Output

```
typedef enum {
        myBitSet_flag1 = (1 << 1),
        myBitSet_flag2 = (1 <<  2)
} myBitSet_t;

myFlagSet_t mfs;
mfs = 0;
mfs |= myBitSet_flag1;

typedef enum {
        setWithRange_highByte = 0xff000000,
        setWithRange_lowByte = 0x000000ff,
        setWithRange_nonContiguous = 0xff0000ff
} setWithRange_t;

setWithRange_t anotherSet;
```

```
anotherSet = 0;
```

5. Design and Implementation

Myth will be implemented as a preprocessor. That is to say that the input to mythpp will be myth source code and the output will be C source code. Output will be strict C to allow for use on a wide range of platforms.

Development will be on the GNU/Linux operating system. GCC will be the primary compiler that output is tested with. No GCC specific options will be used.

Mythpp will run in 3 major phases. The first phase is to scan the input for myth specific extensions. The purpose of this phase is to collect symbols that will require special output to be inserted. Structures will be filled in with the information needed to generate output during this phase. However, during this first phase the compiler will not know where to generate output.

In the second phase the source code will be scanned a second time. During this scan the correct locations to output code will be located. In the third and final phase output will be generated. The compilation of the output source code will be left to the programmer to complete via tools such as make.

Mythpp will be implemented in C. Glib 2.0 will be used for common data structures. The Boehm garbage collector will be used to avoid the tediousness of memory management. Flex will be used to generate the lexical analyzer. Bison will be used to generate a parser. The lexical specification as well as the grammar specification will be based on a public domain grammar of C that appeared on usenet in the late 1980s [Leej].

6. Deliverables

Deliverables

Technical Report

: A report detailing what was learned while implementing mythpp.

Example programs

: A few example Myth programs including at least one non-trivial example program.

Project archive

This archive will contain all project related material. This includes all source code for mythpp, all support scripts, all example files, and all documentation.

User manual

A user manual for Myth will be provided in the project archive. The user manual will be written with a target audience of experienced C programmers wishing to try the extensions provided by the myth pre-processer (mythpp).

7. Schedule

Schedule

Proposal completion

This proposal will be complete and signed off on by April 2005. I will register for the Masters Project for the Spring quarter of 2005.

Mythpp program design

The design of mythpp will be created. It will be a relatively informal design document. This design document will be complete by April, 2005.

Test cases written

Automated test cases will be written by March 1st, 2005. This will be complete to the extent possible at the time. Other test cases will be written during development.

Phase 1 - Extension Declarations

The first phase in mythpp will be complete by March 24th, 2005. This phase is the initial source code scan that finds Myth declarations. I expect that the design will need some rework during this phase.

Phase 2 - Extension Use

The second phase in mythpp will be complete by mid April, 2005. This phase is the phase where use of Myth extensions are found.

Phase 3 - Output

The third and final phase in mythpp is to generate output. This phase will be complete by the end of April, 2005.

Technical Report

A final report will be written. This report will detail the successes or failures that were encountered while implementing mythpp and using the language. This will be complete by early May, 2005.

Defense

This project will have a defense no later than May 20th, 2005.

8. Current Status

Currently this project has this proposal and the design that is listed in this document. Quick and very dirty prototype code written in Perl has been completed as a proof of concept for all language features.

More formal work has begun on mythpp. A complete grammar for mythpp has been created and tested. It appears to be working.

Annotated References

[ReeuwijkSips2002] *Adding tuples to java: a study in lightweight data structures*, C. van Reeuwijk and H. Sips, ACM Java Grande/ISCOPE, 2002, 185-191.

[Leej] *Ansi C grammar and lexical specification by Jeff Lee*.

ftp://ftp.uu.net/usenet/net.sources/ansi.c.grammar.Z