MUAECL2 User's Manual ver. 2.0.1_alpha

MUA Group(Modification Ultraman Alliance)

2018 年 8 月 25 日

目录

1 关于

MUAECL 是 MUA 制作组制作的 ecl 编译器，可以将一种比较类似 C 的语言——MUAECL 语言的 mecl 文本文件，编译成 ecl 文件，并且可以被 MUA 引擎，以及 th15 的引擎所识别运行。

2 安装与运行

下载后解压至任意文件夹即可。解压后的文件包括：

文件名	用途
MUAECL2.exe	程序本体
action.csv	跳转表
ins.ini	ins 对照表
default.ini	default.ecl 中包含的子程序名
include.ini	预定义预处理指令

这些文件需要被放在同一文件夹下。

本程序使用命令行方式运行。在 cmd 中运行 “`MUAECL2.x.x.exe -h`”（实际 exe 名随版本号而不同）或 “`MUAECL2.x.x.exe --help`” 即可查看命令行帮助。

命令行参数		用途
-h	–help	显示命令行帮助
-c	–compile	将 MUAECL2 源代码编译成 ecl 文件
-p	–preprocess	预处理 MUAECL2 源代码，但不编译
-i <filename>	–input-file <filename>	输入文件名
-o <filename>	–output-file <filename>	输出文件名
-n	-no-preprocess	在编译时不进行预处理

命令行应用示例：

```
MUAECL2 -h
MUAECL2 -c -i st01.mecl -o st01.ecl
MUAECL2 -p -i st01.mecl -o st01p.mecl
MUAECL2 -c -n -i st01p.mecl -o st01.ecl
```

3 MUAECL 语言

MUAECL 语言是一种类似 C 的，描述 MUA 引擎中诸如子弹发射、敌人出场等信息的脚本。

3.1 预处理指令

预处理时，MUAECL 首先检测每行行尾的反斜杠，对于行尾有反斜杠的行，将反斜杠删除，并将下一行的文本移至本行。

MUAECL 的预处理指令由# 开始，共有以下几种：

s 指令（s 代表 substitute） `#s` *deliminator pattern deliminator string deliminator name*

作用：将后续文本中每行内的 *pattern* 的所有匹配（识别正则表达式）替换成 *string*，直到 *name* 相同的`ends` 指令出现为止。若没有对应的 ends 指令，则替换直到文件尾。

示例：

```
#s/abc/def/name
#s ((?:a|b|c)*)d(e?) $2$1 mysubs1
```

1. *deliminator* 为一个任意非标识符字符`[^a-zA-Z0-9_]`。
2. *pattern* 与 *string* 内的特殊字符依照ECMAScript规则（即 C++ 标准库 <reg_ex> 中默认的语法规则）替换
3. *pattern* 与 *string* 按照全部字面意思读取，包括空格与制表符，但不包括换行符。
4. *pattern* 与 *string* 内不能包含该非标识符字符，也不能跨行匹配或替换为多行文本。
5. *name* 不得包含非标识符字符。
6. 若有两个 *name* 相同则报错

无名 s 指令 `#s` *deliminator pattern deliminator string deliminator*

作用：将后续文本中 *pattern* 的所有匹配替换成 *string*，直到文件尾。

ends 指令 `#ends` *deliminator name*

作用：结束名为 *name* 的替换。*deliminator* 与 *name* 的要求与`s` 指令相同。

define 指令 `#define` *space identifier space string*

作用：与 c 相同，将 *identifier* 关键词替换成 *string*。

示例：`#define abc def`

会将文本中`abc + abcd` 替换成`def + abcd`。

1. 相当于匹配 “`\b`*identifier*`\b`”，并且 *name* 定义为 *identifier* 的`s` 指令。
2. *identifier* 不得包含非标识符字符。

函数式 define 指令 `#define` *space identifier{paraname1, paraname2, ...} space string*

作用：与 c 类似，将带参数的 *identifier* 关键词替换成 *string*，与 c 不同的是，对于 *string* 中 *paraname* 的处理，只替换字符串不识别标识符。**所以请一定要注意各个 paraname 不要有相互包含的字符串，而且也不要使用 string 中包含的。**

示例：`#define abc{d,e,f} (d+e+f+def)`

会将文本中`abc{1,3,5}` 替换成`(1+3+5+135)`。

1. 相当于匹配 “`\b`*identifier*`\b`”，并且 *name* 定义为 *identifier* 的 s 指令。
2. *identifier* 与 *paraname* 不得包含非标识符字符。

3. 以示例为例，等价于：将 *string* 中 *paraname1* 替换成'$01'，将 *paraname2* 替换成'$02'，以此类推后：

```
#s \babc\{\s*(.*?\S)\s*,\s*(.*?\S)\s*,\s*(.*?\S)\s*\} ($01+$02+$03+$01$02$03) abc
```

undef 指令 `#undef` *space name*

结束 *name* 的后续匹配。等价于`#ends` *name*。

ecli 包含 `#ecli` *space eclfile*

anim 包含 `#anim` *space anmfile*

为文件添加 ecl 文件与 anm 文件的包含。

示例：`#ecli default.ecl`

预处理不改变后续文本的行号。

3.2 预定义宏变量

mecl 包含几个预定义宏变量：

宏名	作用
__FILE__	展开成当前文件名
__DATE__	展开成翻译到当前关键字时的日期
__TIME__	展开成翻译到当前关键字时的时间
__LINE__	展开成行号
pi	展开成 3.14159265

4 基本概念

4.1 注释

mecl 允许两种注释：C 风格注释或 C++ 风格注释：

```
1. /* some content */
2. // some content \n
```

与 C 不同的是，注释的处理在预处理替换步骤之后，也即：1. 注释范围内的预处理指令会被处理；2. 预处理替换后组合而成的注释符号会被识别。

4.2 标识符

与 C 相同，一个标识符是一个由数字，下划线，小写和大写拉丁字母组成的任意长度的序列，且不可以数字起始。标识符区分大小写。

有效的标识符示例：`shedarshian`,`Temp0_float`,`_name_`

4.2.1 全局变量

mecl 语言中包含若干全局变量，诸如自机坐标、内置随机数等，列表可在 ins.ini 中查询。若标识符与某一全局变量名相同，则会被识别成该全局变量。

4.3 类型

mecl 语言包含如下几种类型：

void 类型，int 类型，float 类型，string 类型，point 类型。（目前版本中对于 string 类型**变量**尚无支持）

mecl 中 int 与 float 类型占据 4 字节。

称算术类型包含 int，float，point，基本算术类型包含 int，float

4.3.1 隐式类型转换

在需要时（如重载决议发生时，或 if、for 表达式内），会检测是否需要隐式类型转换。

隐式类型转换包括：左值到右值转换，整数到浮点数转换，以及 void 类型左值到任意类型左值转换。

5 子程序

mecl 由多个子程序（`sub`）组成：*suffix* `sub` *subname* `(` *subvars* `)` *suffix* `{` *subcontents* `}`

suffix 为可选的`no_overload` 关键字。参数列表为*参数声明*的逗号分隔列表，每一项包含一个类型名以及一个变量名。这些具名形参均可在函数体内被访问。函数体 *subcontents* 内为一条条的语句。

示例：

```
sub MainSub01(float x) no_overload
{
        //some content
}
no_overload sub Main()
{
        //some content
}
sub Boss_at1(int num, float angle) {
        //some content
}
```

子程序之间允许同名（**重载**），只要参数数或类型不同。调用重载的函数时遵循重载决议（后述）。

默认情况下，编译器输出 ecl 文件时会使用输入参数类型名装饰 sub 名。`no_overload` 关键字指示编译器在输出至 ecl 文件时不装饰 sub 名，同时禁止了对此函数的重载。

子程序名为全局，也即允许**后声明先调用**。

6 表达式

表达式是运算符和它们的运算数的序列，它指定一项计算。

6.1 类型与值类型

每个表达式有一个类型，和一个值类型。值类型为左值或右值。左值指示其求值确定一个对象个体，右值指示其求值没有结果对象。

下列表达式是左值表达式：

1. 变量名a。
2. 赋值及复合赋值表达式，如a = b。
3. 间接寻址表达式，如*p。

左值可以取地址&(i = 1)，左值可以作为赋值运算符的左操作数。

下列表达式是右值表达式：

1. 字面量，如42。
2. 函数调用表达式，如g(1)。
3. 算术表达式、逻辑表达式、比较表达式，如a + 1、c && d 等。
4. 取地址表达式，如&a。
5. 转型表达式，如(int)f。

右值不能被取地址，右值不能作为赋值运算符的左操作数。

6.2 初等表达式

初等表达式包括字面量与标识符。

整数字面量 整数字面量包含以下几种：

1. 十进制字面量，由1-9 的数位后随零或多个0-9 的数位组成；
2. 八进制字面量，由0 后随多个0-7 的数位组成；
3. 十六进制字面量，由0x 后随多个0-9 或A-F 或a-f 组成。

例如下列变量被初始化到相同值：

```
int d = 42;
int o = 052;
int x = 0x2a;
int X = 0x2A;
```

浮点数字面量 浮点数字面量为一个包含且仅包含一个小数点的十进制数，后随一个可选的f 组成的字面量。如：2.34,.1f,1.。

字符串字面量 字符串字面量为用一对双引号"" 围起来的字符序列。反斜杠可以用于 escape，在单个反斜杠后的字符以本义被录入，可用于在字符串中添加双引号字符本身。字符串字面量拥有 string 类型。

预定义常量 预定义常量在 ins.ini 中定义，被转义成等效的整数字面量或浮点数字面量。支持的预定义常量将在对应的 instruction 处列出。

标识符 标识符可以用于函数名，或者指代一个变量。

6.3 运算符

运算符有以下几类：

赋值	算术	逻辑	比较	成员访问	其他
a = b a += b a -= b a *= b a /= b a %= b a &&= b a ||= b a &= b a |= b a ^= b	-a a + b a - b a * b a / b a % b a & b a | b a ^ b	!a a && b a || b a and b a or b	a == b a != b a < b a <= b a > b a >= b	a.b a[b] *a &a	id(...) a1 : a2 : a3 : a4 (type)a (a)

内建运算符有几种内建的重载，重载决议规则决定调用哪个重载。

6.3.1 算术运算符

一元负运算符 返回运算数的相反数。形式为：-expr。

对于算术类型A，下列函数签名参与重载决议：

```
A operator-(A)
```

加减运算符 返回运算数之和或差。形式为：expr + expr，expr - expr。

对于算术类型A，下列函数签名参与重载决议：

```
A operator+(A, A)
A operator-(A, A)
string operator+(string, string)
```

乘法性运算符 返回运算数之积、商或余数。对于整数之商，向零截断小数部分。形式为：`expr * expr`，`expr / expr`，`expr % expr`

对于基本算术类型`I`，下列函数签名参与重载决议：

```
I operator*(I, I)
I operator/(I, I)
int operator%(int, int)
point operator*(float, point)
point operator*(point, float)
point operator/(point, float)
```

按位逻辑运算符 返回运算数之按位与、按位或、按位异或。形式为：`expr & expr`，`expr | expr`，`expr ^ expr`。

下列函数签名参与重载决议：

```
int operator&(int, int)
int operator|(int, int)
int operator^(int, int)
```

6.3.2 逻辑运算符

逻辑运算符返回运算数之逻辑与、逻辑或、逻辑非。此处以 int 代替布尔值，非 0 为真，0 为假。返回值以 1 代表真，0 代表假。形式为：`expr && expr`，`expr || expr`，`expr and expr`，`expr or expr`，`! expr`。提供两种版本的与、或运算符：符号形式短路求值，单词形式不短路求值。短路求值即：对于`&&`，若第一运算数为假则不求值第二运算数；对于`||`，若第一运算数为真则不求值第二运算数。

下列函数签名参与重载决议：

```
int operator&&(int, int)
int operator||(int, int)
int operator and(int, int)
int operator or(int, int)
int operator!(int)
```

6.3.3 比较运算符

比较运算符返回运算数间是否相等以及大小关系。形式为：`expr == expr`，`expr != expr`，`expr < expr`，`expr <= expr`，`expr > expr`，`expr >= expr`。

对于全部四种类型之一`T`（int, float, point, string），下列函数签名参与重载决议：

```
int operator==(T, T)
int operator!=(T, T)
int operator<(T, T)
int operator<=(T, T)
```

```
int operator>(T, T)
int operator>=(T, T)
```

6.3.4 赋值运算符

赋值运算符的右操作数允许直接使用冒号分隔表达式，其余运算符则不允许。赋值运算符期待可修改左值为其左运算数，右值表达式为其右运算数。

简单赋值运算符 简单赋值运算符以右操作数的值修改左操作数。形式为：`expr = exprf`。赋值运算符返回左操作数的左值。

对于全部四种类型`T`（int, float, point, string），下列函数签名参与重载决议：

```
T& operator=(T&, T)
```

复合赋值运算符 复合赋值运算符以左操作数与右操作数运算后的值修改左操作数，`E1 op= E2` 的行为与表达式`E1 = E1 op E2` 行为相同，除了只求值表达式`E1` 一次。形式为：`expr += exprf`，`expr -= exprf`,`expr *= exprf`,`expr /= exprf`,`expr %= exprf`,`expr &&= exprf`,`expr ||= exprf`，`expr &= exprf`，`expr |= exprf`，`expr ^= exprf`。其中逻辑与赋值运算符、逻辑或赋值运算符有短路求值。

对于算术类型`A`、基本算术类型`I`、全部四种类型之一`T`（int, float, point, string），下列函数签名参与重载决议：

```
T& operator+=(T&, T)
A& operator-=(A&, A)
I& operator*=(I&, I)
point& operator*=(point&, float)
I& operator/=(I&, I)
point& operator/=(point&, float)
int& operator%=(int&, int)
int& operator&&=(int&, int)
int& operator||=(int&, int)
int& operator&=(int&, int)
int& operator|=(int&, int)
int& operator^=(int&, int)
```

6.3.5 成员访问运算符

间接寻址运算符 间接寻址运算符取出指针指向的操作数。形式为：`*expr`。由于当前版本无指针类型，以 int 代替指针，故间接寻址运算符返回 void 左值对象，需使用类型转换表达式指定取出对象的类型。特别的是，此 void 左值对象可以隐式转换到任意类型，故可以利用此特性靠上下文推断出取出的对象类型。而不能推断出类型的表达式则视为错误。更多信息请参考重载决议。

下列函数签名参与重载决议：

```
void& operator*(int)
```

取地址运算符 取地址运算符获得指向该对象的指针。形式为：`&expr`。由于当前版本无指针类型，故取地址运算符返回 int 类型。

对于全部四种类型之一`T`（int, float, point, string）下列函数签名参与重载决议：

```
int operator&(T&)
```

成员运算符 `expr.expr`，当前版本未加载其作用。

数组下标运算符 `expr[expr]`，当前版本未加载其作用。

6.3.6 其他运算符

函数调用运算符 形式为：`id(exprf, exprf, ...)`。`id` 为标识函数名的标识符，可以为用户定义的 sub 名或是内建的 ins 名或 mode 名。函数传入的参数列表可以为冒号分隔表达式或者普通表达式。用户定义的 sub 名可以重载，重载决议规则决定调用哪个重载。需注意的是，几乎所有内建的 ins 与 mode 均返回 void。

当前版本所有函数返回 void 右值。

冒号分隔表达式 冒号分隔表达式是一种简便的难度 switch，接受对应 ENHL 四种难度的四个参数，形式为`expr : expr : expr : expr`。只有赋值表达式的右侧与函数调用支持冒号分隔表达式。

对于全部四种类型之一`T`（int, float, point, string）下列函数签名参与重载决议：

```
T operator:(T, T, T, T)
```

C 风格转型表达式 转型表达式可以将一种类型转换为另一种类型。形式为：`(type)expr`。转型表达式无转换类型限制（???）。

括号表达式 括号表达式用于指定运算顺序。形式为：`(expr)`。

6.4 运算符优先级

下表列出 mecl 运算符的优先级和结合性。运算符从顶到底以降序列出。

优先级	运算符	描述	结合性
1	()	括号	左到右
2	a.b	取成员	左到右
	a[b]	取下标	
3	!	逻辑非	右到左
	*a &a	寻址与取址	
	-a	一元负	
4	a*b a/b a%b	乘法除法余数	左到右
5	a+b a-b	加法减法	左到右
6	a<b a<=b	小于小于等于	左到右
	a>b a>=b	大于大于等于	
7	a==b a!=b	等于不等于	左到右
8	a&b	按位与	左到右
9	a^b	按位异或	
10	a|b	按位或	
11	a and b	非短路逻辑与	
12	a or b	非短路逻辑或	
13	a&&b	短路逻辑与	
14	a||b	短路逻辑或	
15	= += -= *= /=	所有赋值	右到左
	%= &&= ||= &= ^= |=		
16	a1:a2:a3:a4	冒号分隔	-

6.5 重载决议

选取一个函数或运算符的重载时，使用如下规则：

对该函数或运算符的所有重载判断输入参数个数是否相等，以及能否将每个输入参数类型隐式转换至目标类型。之后在所有可以转换到的重载函数集中选取最优的。最优定义为：有高优先级的某转换的隐式转换优于无该转换的。隐式转换的优先级由高到低为：

`void&`→`point`, `void&`→`string`, `void&`→`float`, `void&`→`int`, `int`→`float`, `lvalue`→`rvalue`.

6.6 字面量优化

编译器会对于字面量运算在编译时进行运算，优化后存入 ecl 脚本。如`a*(1./2)` 在 ecl 文件中会变成`a*0.5`。

7 语句

mecl 的每条语句（*stmt*）有以下几种类型：

7.1 表达式语句

跟随分号的表达式是语句。

expr ;

示例：

```
a += 1;                  //赋值语句
wait(20);                  //ins 调用
MainSub00();                  //函数调用
a;                 //弃值表达式
```

7.2 语句块

{ *stmts* }

可以使用大括号括起多条语句，将其合并成一条。

7.3 变量声明

变量声明语句声明（并可选的初始化）一个或多个变量。形式为：`type` *vdecl* ;

其中 *vdecl* 为一个或多个下列项的逗号分隔列表：`id` 或`id =` *inif*。

inif 可以为有或无冒号分隔的初始化列表或者表达式。

mecl 的每条变量声明均为子程序局域，也即只要在同一个子程序中，允许**后声明先调用**。同样无局部作用域的概念。

示例：

```
int i;
float a = 1., b;
int j = 123 * 456 : 234 * 567 : 345 * 678 : 456 * 789;
```

7.3.1 初始化列表

{ *expr1, expr2,...* }

初始化列表是用来初始化复杂算术类型（如 point）的手段。示例：

```
point p = {1., 2.};
point p1 = {1., 2.}:{3., 4.}:{5., 6.}:{7., 8.};
```

7.4 选择语句

选择语句在数个控制流中选择一个。形式为：

```
if ( condition ) true_stmt
if ( condition ) true_stmt else false_stmt
```

若 *condition* 为非 0 则执行 *true_stmt*（常为复合语句），否则执行 *false_stmt*。

condition 需可以隐式转换至 int 右值。

在第二形式的`if` 语句（包含`else` 者）中，若 *true_stmt* 亦是`if` 语句，则内层`if` 语句必须也含有`else` 部分。换言之，嵌套`if` 语句中，`else` 关联到最近的尚未有`else` 的`if`。

若通过goto 进入 *true_stmt*，则不执行 *false_stmt*。

示例：

```
if (i == 1) {
        et_shape(0, ET_small, BLUE16);
        et_shoot(0);
        wait(30);
}
else if (i == 2)
        if (j == 2)
                wait(20);
        else                        //此 else 与 j == 2 的 if 对应
                wait(30);
```

7.5 循环语句

循环语句重复执行一些代码。形式为：

```
while ( condition ) stmt
do stmt while ( condition ) ;
loop ( times ) stmt
```

在循环语句中可以使用 break 语句和 continue 语句进行流程控制。

7.5.1 while 循环

while 循环重复执行 *stmt*，直到 *condition* 变为 0。

condition 需可以隐式转换至 int 右值。

示例：

```
float s = 10.;
while (s >= 0) {
        et_dir(0, RanDEG, 0.);
        et_speed(0, s, 0.);
        et_shoot(0);
        wait(1);
        s -= 0.1;
}
while (1)
        wait(1000);                         //这是死循环
```

7.5.2 do-while 循环

do-while 循环类似 while 循环，但至少执行一次 *stmt*。

示例：

```
float c = 1, s = 0;
do {
        s += c;
        c /= 2;
} while (c > 0.00001);
```

7.5.3 loop 循环

loop 循环执行 *times* 次 *stmt*。

times 需可以隐式转换至 int 右值。

示例：

```
loop (100) {
        et_shoot(1);
        wait(1);
}
```

7.6 标号语句

标号语句建立一个标号，以供 goto 语句跳转。形式为：

```
id :
```

7.7 跳转语句

跳转语句无条件地转移控制流。形式为：

```
break ;
continue ;
goto id ;
```

7.7.1 break 语句

break 语句退出外围的 while、do-while、loop 语句。

break 语句不能用于跳出多重嵌套循环。这种情况可以使用 goto 语句。

示例：

```
while (1) {
        et_shoot(0);
        wait(30);
        if (Yplayer <= 120.)
                break;
}
et_shoot(1);
```

7.7.2 continue 语句

continue 语句跳过外围的 while、do-while、loop 语句中的剩余部分，继续下一次循环。如同用 goto 跳转到循环体尾。

示例：

```
loop (1000) {
        et_shoot(0);
        wait(10);
        if (Y_absol <= 224.)
                continue;
        et_shoot(1);
}
```

7.7.3 goto 语句

goto 语句跳转至指定的标签处。goto 语句必须在与它所用的标签相同的函数中，它可出现于标签前后。

7.8 thread 语句

thread 语句开启一个线程，与后续程序同步进行。形式为：

```
thread id ( exprf, exprf, ... ) ;
```

示例：

```
thread BossCard1_at1(1., 20, pi/20, RED32);
thread BossCard1_at1(2., 20, pi/20, GREEN32);
thread BossCard1_at1(1.5, 10, pi/10, CYAN32);
while (1) wait(1000);
```

7.9 rawins 语句

rawins 语句使用数据直接构建 ecl 内容。形式为：

```
__rawins {
        insdata ;
        insdata ;
        ...
}
```

每条 *insdata* 为：

{ *id difficulty_mask pop_count param_count param_mask, param1 param2 ...* }