**Montana Tech Software Development Standard**

**MTM Algorithm Language Framework for Functions/Methods**Version 5.6  
Dec 2, 2015

A. Frank Ackerman

*Software Engineering  
Montana Tech*

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comment |
| 4.6 | 01/26/11 | Frank Ackerman | Added Hold |
| 4.7 | 02/05/11 | Frank Ackerman | Added Exchange |
| 4.8 | 03/27/11 | Frank Ackerman | Minor edits. Student input page. Remove example |
| 4.9 | 04/03/11 | Frank Ackerman | Add Elide and Negate |
| 4.10 | 05/01/11 | Frank Ackerman | Expand Reset to functions w/ static |
| 4.11 | 08/31/11 | Frank Ackerman | Add Throw |
| 4.12 | 09/09/11 | Frank Ackerman (from A. Hoff) | Add Parse |
| 4.13 | 10/01/11 | Frank Ackerman | Organize by major construct |
| 4.14 | 12/14/11 | Frank Ackerman | Add Run |
| 4.15 | 01/22/12 | Frank Ackerman | Add Truncate |
| 4.16 | 09/24/12 | Frank Ackerman | Add Call |
| 4.17 | 10/05/12 | Frank Ackerman | Add Start |
| 4.18 | 11/1/12 | Frank Ackerman | Edits to Display and Obtain |
| 4.19 | 11/5/12 | Frank Ackerman | Add ReadIn for stdin |
| 4.20 | 11/28/12 | Frank Ackerman | Add Setup and use of functions |
| 4.21 | 12/16/12 | Frank Ackerman | Add WriteLine and functions |
| 4.22 | 12/23/12 | Frank Ackerman | Restrict Obtain to using a prompt |
| 4.23 | 02/02/13 | Frank Ackerman | Added missing semicolons |
| 5.0 | 05/04/13 | Frank Ackerman | Section overview to *Introduction,* Add Terminate C string |
| 5.1 | 10/12/13 | Frank Ackerman | Exec for C |
| 5.2 | 03/28/14 | Frank Ackerman | Section 4 wording and Catch |
| 5.3 | 10/13/14 | Frank Ackerman | Added example |
| 5.4 | 01/25/15 | Frank Ackerman | Minor changes |
| 5.5 | 08/19/15 | Frank Ackerman | Change title |

**Montana Tech Software Engineering Students:**

These Montana Tech Method software engineering standards encapsulate Dr. Ackerman’s decades of experience in the software industry, the IEEE software engineering standards, and many suggestions from various texts. They have gone through many revisions and additions over the last several years. They are part of your software engineering studies so that (1) you may have the experience of developing software to a standard (which you may find you need to do if you take a job that requires high reliability software), and so that (2) you will have the experience of developing high quality software. You are also invited to participate in the continuing evolution of these standards by studying them critically and making suggestions for their improvement and correction.

Purpose

The purpose of this document is to define the *MTM Algorithm Language for Functions/Methods*. This language is used throughout the Montana Tech Software Engineering courses to describe the detail design of functions or methods for procedural languages.

Introduction

The *MTM Algorithm Language for Functions/Methods* provides SE students with an easy-to-use and easy-to-understand method for unambiguously describing the design of functions and methods.

It should be possible for a knowledgeable person to "execute" a program design. That is, while specific details may not be spelled out, all the important logical program constructs should be readily apparent. For example, if the program will use a loop the design should use one of the loop constructs given below. The loop initial and continuing conditions should be clearly stated.

It should be possible to clearly align the design statements with the code and a sample run. In general, blocks of the design will correspond one-to-one with blocks of code.

The design elements defined in this standard are divided into the following categories:

### **5. Sequential Constructs**

**6. Selection Constructs**

**7. Repetition Constructs**

**8. Other Constructs**

Application

This standard applies to program designs for procedural and object-oriented modules and functions written for programming exercises and projects throughout the Montana Tech software engineering curriculum.

Standard

An MTM function/method design is a sequence of design elements. Each element consists of a sequence of design language constructs. Except for the initial, data definition element, each construct in the element begins with one of the keywords (initial letter capitalized and bolded) from the list below and ends with a semicolon or a right brace on a separate next line with a type comment (as shown in the table below). Each element begins with a design element label of the form A*..* that is attached to the first construct in the element.

An initial, data definition element (A00) may explicitly declare user data types and also the definition of key variables. Type and declarations are written as code in the target language.[[1]](#footnote-1)

The initial data definition element may be preceded by an optional Algorithm Notes section to assist the reader in understanding the design.

As much as possible, design elements should be label in the sequence A00, A01, A02, .... . As the design develops and new elements need to be inserted, these labels may be extended by adding a suffix of a, b, c, ...

The design list of design constructs given in the next section is the heart of this standard. To facilitate clear design intent, to make a design abstractly executable, and to permit the construction of correctness arguments, only the listed constructs may be used. The listed constructs have been (and continue to be) developed) using the following considerations:

* the list includes all of the control structures used in any of the procedural coding done at Montana Tech
* the list includes all of the data operations used in any of the procedural coding done at Montana Tech
* the semantics of each data operation be clearly understood by Montana Tech software engineering students and faculty and there should be minimal semantic overlap between the listed data operations
* within the above constraints, the list of design constructs should be as short as possible.

The phrases between the keywords and the optional comment are not explicitly defined but should clearly describe the intention or meaning that the code will implement, as should the optional comment following terminating (and labeled) right braces. Braces that will appear in the program should usually appear in the design. Program objects and object attributes, where these are used in the program, should be explicitly referenced by name. Since design text may be part of code, which almost always uses a single fixed width font, ordinary English words should not be used to name objects. Use compound names that clearly reference the object or attribute. For example, use WordCnt instead of count to reference a word count. See the MTM coding standards for more information on naming program objects and attributes.

# Sequential Constructs

|  |  |
| --- | --- |
| **Accumulate** *text*; | to describe a += accumulation operation |
| **Add *text;*** | to add a numeric quantity |
| **Advance *walkListPtr;*** | use at the bottom a Walk construction |
| **Allocate** *text*; | explicitly allocate dynamically managed memory |
| **Append** *text*; | to add string/item to the end of a string or a list |
| **Assert(***condition***);** | assert condition |
| **Calculate** *text*;  **Compute** *text*; | to describe a calculation that is more complex than incrementing or decrementing a object or attribute. |
| **Call** *function [*with *parameters]* | to invoke a function |
| **Catch** *exception(s)* | starting a catch block |
| **Close** *text*; | to close a file |
| **Construct** *something*; | to describe creating a new object |
| **Convert** *from one form to another***;** | usually to describe changing a type or converting a value, e.g., miles to kilometers |
| **Decrement** varName *by amount*; | to describe that a value is being decremented (the amount is 1 if not stated)  This construction should always explicitly reference an object or attribute |
| **Define** *text;* | to introduce identifiers outside of D00 |
| **Delete** *text*; | to remove something from a string or a list |
| **Demonstrate** *text* | usually only for a demonstration program |
| **Dialog to display** *tex;t* | to use a dialog box to display an output |
| **Dialog to obtain** *text;* | to use a dialog box to obtain user input |
| **Display** *text*; | to display something on stdout |
| **Exec** *text*; | In Unix – replace a running process |
| **Elide** *text;* | applies to elide part of a string, usually a prefix or suffix |
| **Exchange** *text;* | exchange the values of two objects |
| **Generate** *something***;** | usually to describe generating code in a scripting language |
| **Hold** *text***;** | used to keep console screen up when running programs in Visual Studio |
| **Increment** varName *by amount*; | to describe that a value is being incremented (the amount is 1 if not stated.  This construction should always explicitly reference an object or attribute |
| **Initialize** *text* | Set value of object for first time. |
| **Instantiate** *object;* | to introduce an object of a defined class |
| **Link** *text;* | to explicitly manipulate pointers to place an item into a linked structure |
| **Negate;** | applies to a numeric object:  objVar = -objVar |
| **Obtain** *text*; | To give a prompt on stdout and obtain data from the user on stdin |
| **Open** *text*; | to open a file |
| **Peek** *text***;** | obtain next character of an instream without taking it out of the stream. |
| **Parse** *text***;** | break a string or body of text into parts |
| **Prepend** *tex*t**;** | to add string/item to the beginning of a string or a list |
| **Process** *tex*t [**;**|**:**] | perform a defined or easily understood steps; usually terminated with a colon and followed by subordinate steps |
| **Putback** *text;* | put a character in at the head of an instream |
| **Read** *text*; | to describe the operation of reading data from an opened file |
| **Readin** *text*; | to describe the operation of just reading data from stdin without first prompting |
| **Release** *text*; | explicitly release dynamically allocated memory |
| **Reset** *varName* to *text*;  or  **Reset** *funcName for* *text*; | to change the value of an object or attribute back to something it was before  or  call a function to reset static variables |
| **Return** *[object value];*  or  **Return** *to invoker* | to return from a function;  or  to exit a program, return 0 for a normal exit and 1 for an error exit. |
| **Rotate** *text*; | to describe the operation of moving an item from one end of a linear arrangement to another, e.g. moving the left-most character of a string to become the right-most. |
| **Run** *text*; | Run is a C# Application method. It is used to give control to a class derived from Form. |
| **Seed** the random number generator; | for the code:  srand(time(0)); |
| **Set** varName to *text*; | to change the value of an object or attribute |
| **Setup** *text;* | when setting the size or initial contents of a container |
| **Show** *text* | when using a "show" method in Java |
| **Terminate** *text* | when adding ‘\0’ to create a C string. |
| **Truncate** *text* | to shorted a string by eliding characters from its left or right end |
| **Try** *text;* | Starting a try block |
| **Unlink** *text*; | to explicitly link manipulate pointers to take an item out of a linked structure |
| **Update** *text*; | to describe making a modification to the values of an object – generally to reflect a certain condition |
| **WriteLine** *text*; | to correspond with a common C# Console method |

# Selection Constructs

|  |  |
| --- | --- |
| **Call** funcName(); | to show an explicit function call |
| **Case** *casevalue***:** | to mark a entry in a switch statement |
| **Catch** *text;* | to start a catch block |
| **Default:** | the default case for a switch |
| **If** (*condition*) {  *body of if*  **}//If** *optional comment* | for code that is to be executed only if *condition* is true. |
|  |  |
| **If** (*condition*) {  *body of if*  **}//If** *optional comment*  **Else if** (*condition*) {  *body of elseif*  **}//Else if** *optional comment*  …  **Else {**  *body of else*  **}//Else** *optional comment* | to describe a complex multi-way branch |
| **Switch** (*condition*) {  **Case** *value-1*:  statement;  …  **Default**:  statement  …  **}//Switch** | to describe a simple multi-way branch |
| **Throw** *text;* | Throw an exception. |
| **Try** *text;* | Starting a try block |

# Repetition Constructs

|  |  |
| --- | --- |
| **Backup *walkListPtr;*** | use at the bottom a Walk construction |
| **Break;** | for code break |
| **Continue;** | for code continue |
| **Do** **{**  *body of dowhile loop*  **} Until** (*condition*); | to describe a loop that will be executed once before *condition* is checked |
| **For** (*condition*) {  *body of for*  **}//For** *optional comment* | to describe loops that iterate across a sequence of items |
| **Walk** *list with* ***walkListPtr*** {  *body of loop* **}//Walk** | to describe a while loop to walk a linked list. |
| **While** (*condition*) {  *body of loop* **}//While** *optional comment* | to describe a loop that will not be executed even once if *condition* is not meet  For an loop in which the exit condition is given by a break in the body of the loop, *condition*, is forever. |

# Other Constructs

|  |  |
| --- | --- |
| **Start *thread;*** | Start a thread executing |

# Use of Functions

All of the constructs described above apply to a single function. It is important for the student to understand that function calls (for example, the Call construct above) should be used for all substantial and logically coherent operations.

# Example

Problem statement: Any positive integer in [1, 2B] can be the start of a 3n+1 sequence. The sequence is constructed by halving the previous value if it is even or multiplying it by 3 and adding 1. Such a sequence is guaranteed to eventually generate the value 1. (If the starting number is 1, the sequence length is 1; if the starting number is 2, the sequence length is 2, as the sequence will be: 1, 1). A algorithm (using the constructs defined above) for obtaining a positive integer from a user (this is assumed) and computing the length of its 3n+1 sequence is given below. An example of execution of a program constructed to this design is:

Positive integer to start> 5

Length of the 3n+1 sequence starting at 5 is 6

The sequence in this example is 5, 16, 8, 4, 2, 1.

An algorithm that gives this result is:

A01 **Obtain** positive integer startInt from user:   
 “Positive integer to start> “

**Set** seqLngth to 1;

**Set** preSeqVlu to startInt;

A02 **While** (True) {

A03 **If** (prevSeqVlu equals 1) {

A04 **Display**:

The length of the 3n+1 sequence starting at <startInt> is <seqLngth>

**Break**;

}//If at end of sequence

A05 **If** (prevSeqVlu is even) {

Set nxtSeqVlu to (prevSeqVul / 2);

}//If

A06 **Else** {

Set nxtSeqVlu to (3\*prevSeqVlu) + 1;

}//Else

A07 **Increment** seqLngth;

**Set** prevSeqVlu to nxtSeqVlu;

}//While computing sequence values

1. When the design appears in a function heading comment D00 should not be used. Instead types and variables referenced in the design should appear at the beginning of the function stub (or complete code). [↑](#footnote-ref-1)