Contributing to Simbody

=======================

Simbody is a community resource and we encourage you to contribute in whatever way you can -- for example: new code, bug fixes, test cases, and examples; documentation improvements and typo fixes; bug reports, feature requests, ideas and discussion topics; and user forum questions and answers. We appreciate contributions and our development team is collaborative and constructive -- don't be shy!

\*\*Important note:\*\* Simbody is an open source project licensed under extremely flexible terms intended to encourage use by \*anyone\*, for \*any purpose\*. When you make a contribution to the Simbody project, \*\*you are agreeing\*\* to do so under those same terms. The details are [below](#contributor-license-agreement); if you aren't comfortable with those terms, we're still friends but you shouldn't contribute.

Contents:

- [Ways to Contribute](#ways-to-contribute)

- [Submitting Pull Requests](#submitting-pull-requests-prs)

- [Coding Conventions](#coding-conventions)

- [List of Contributors](#list-of-contributors)

- [Contributor License Agreement](#contributor-license-agreement)

Ways to contribute

------------------

There are lots of ways to contribute to the Simbody project, and people with widely varying skill sets can make meaningful contributions. Please don't think your contribution has to be momentous to be appreciated. See a typo? Tell us about it or fix it! Here are some contribution ideas:

- Use Simbody and let us know how you're using it by posting to the [Simbody user forum](https://simtk.org/forums/viewforum.php?f=47).

- Ask and/or answer questions on the forum.

- File bug reports, documentation problems, feature requests, and developer discussion topics using the GitHub [Issue tracker](https://github.com/simbody/simbody/issues).

- Submit GitHub Pull Requests providing new features, examples, or bug fixes to code or documentation (see below).

- If our hard work has helped you with yours, please considering acknowledging your use of Simbody and encourage others to do so. Please cite this paper:

Michael A. Sherman, Ajay Seth, Scott L. Delp, Simbody: multibody dynamics for biomedical research, \*Procedia IUTAM\* 2:241-261 (2011) http://dx.doi.org/10.1016/j.piutam.2011.04.023

Submitting Pull Requests (PRs)

------------------------------

Please don't surprise us with big out-of-the-blue PRs. If you want to work on an existing Issue, post a comment to that effect in the Issue. That way you can engage in discussion with others to coordinate your work with theirs and avoid duplication, to see what people think of the approach you have planned, and so on. If there is not yet a relevant Issue in place, a great way to start is by creating one, and then engaging in an Issue conversation.

The main (upstream) repository (repo) for Simbody is the `simbody` repo in the `simbody` organization on GitHub; that is, https://github.com/simbody/simbody. The general idea is that you will work in your own copy of that repo on GitHub (called a \*fork\*) and then submit a PR requesting to merge a branch in your fork into a branch in the upstream repo.

### Mechanics of submitting a PR

This is a very abbreviated description of the process. If you are new to Git and Github you will need to look at some of the great GitHub tutorials, starting with GitHub Bootcamp [here](https://github.com).

Below we'll assume your GitHub account is `yourid`.

1. \*\*Create your own fork\*\* `yourid/simbody` of the `simbody/simbody` repo on GitHub. Use the `Fork` button [here](https://github.com/simbody/simbody).

2. \*\*Clone your repo\*\* `yourid/simbody` onto your local machine. (It is possible to work directly on your GitHub fork using GitHub's browser interface, but this is inadvisable except for small, safe documentation changes.)

3. \*\*Create a branch\*\* like `something-feature` for your new feature or `fix-something-issue123` for a bug fix (we're not fussy about branch names; they are just temporary).

4. \*\*Commit the new code\*\* or documentation to the `something-feature` branch.

5. \*\*Test and debug\*\* your changes locally. Be sure to build at least occasionally in Debug mode -- it will run very slowly but you get much more error checking that way.

6. \*\*Push\*\* now-debugged `something-feature` branch up to `yourid/simbody` fork on GitHub.

7. \*\*Create the PR\*\*. Go to the `simbody/simbody` repo, click Pull Requests, and create a new PR. Specify `simbody/simbody master` as the base (destination) branch and `yourid/simbody something-feature` as the head (source) branch. Provide a description and reference the corresponding Issue(s). If there are particular people whose attention you want to draw to the PR, use "at mentions" like `@someone` in your PR description.

8. \*\*Check the build status\*\*. Your PR submission will trigger our continuous integration builds on Travis (for Linux and OS-X) and AppVeyor (for Windows). GitHub provides a status message at the bottom of the PR's Conversation page allowing you to track build progress. Make sure the build succeeds on all platforms, and if not click the `Details` button and fix the problem if you can, or else ask for help.

9. \*\*Engage in discussion\*\* with Simbody maintainers who will review your changes and make comments.

10. \*\*Make changes\*\*. In most cases discussions and build problems will require you to make some changes to your submission. That is very easy to do because a PR is a \*reference\* to your branch, not a copy. So you just make the changes to the `something-feature` (or whatever) branch on your local clone, and then push those changes back to the same branch in your `yourid/simbody` fork on GitHub. The changes will immediately start building and you can return to discussing them in the same PR.

Eventually your PR will be merged (good) or closed unmerged by a Simbody maintainer, but always after an open discussion.

Coding Conventions

------------------

The coding conventions below are meant to apply to new code. If you are submitting code that includes large pieces of pre-existing open source code, that code will have its own conventions. Please \*do not\* reformat that code to use our coding conventions because (a) that is just busy work, and (b) the code is then difficult to compare with or update from the original source.

Many differences in programming technique fall into the realm of personal aesthetics (style) where one approach is not inherently better than another. It is our intent to be as accommodating as possible in this regard so that you can express yourself comfortably. However, we don't think it's a good idea to mix incompatible styles within the same or closely related source modules. That makes the software increasingly hard to read and understand over time. And it's ugly. So we ask that modifications to existing software be made in the original style of that software as much as possible, or be converted to a consistent style. We are more concerned about uniformity in the user-visible API than in internal implementation code.

Existing Simbody code does not perfectly follow these conventions and we appreciate Issues pointing out problems, and especially PRs that correct our earlier slip-ups.

- [Basic requirements](#basic-requirements)

- [Write new code in C++](#write-new-code-in-c)

- [Keep line width to 80 characters](#keep-line-width-to-80-characters)

- [Do not use tabs to indent; use 4 spaces](#do-not-use-tabs-to-indent-use-4-spaces)

- [Use English and a spell checker](#use-english-and-a-spell-checker)

- [Provide Doxygen comments](#provide-doxygen-comments)

- [Code should be `const` correct](#code-should-be-const-correct)

- [Objects should be thread safe](#objects-should-be-thread-safe)

- [Naming conventions](#naming-conventions)

- [Types: classes and structs, typedefs, enums](#types-classes-and-structs-typedefs-enums)

- [Constants](#constants)

- [Functions and methods](#functions-and-methods)

- [Variables](#variables)

- [Preprocessor macros](#preprocessor-macros)

- [Namespaces](#namespaces)

- [Header guards](#header-guards)

- [Calendar dates](#calendar-dates)

- [C++ tips and style guide](#c-tips-and-style-guide)

- [Public class members come first](#public-class-members-come-first)

- [Use anonymous namespaces](#use-anonymous-namespaces)

- [Don't waste lines on curly braces](#dont-waste-lines-on-curly-braces)

- [`throw` and `return` are not functions](#throw-and-return-are-not-functions)

- [Prefer \*pre\*-increment and \*pre\*-decrement operators](#prefer-pre-increment-and-pre-decrement-operators)

- [Place pointer and reference symbols with the type](#place-pointer-and-reference-symbols-with-the-type)

- [Avoid spaces that don't improve readability](#avoid-spaces-that-dont-improve-readability)

- [Make assignment operators safe for self-assignment](#make-assignment-operators-safe-for-self-assignment)

### Basic requirements

#### Write new code in C++

New code for Simbody should be written in C++. In Simbody 3.6 and later this can be C++11; before that it must be limited to C++03. Submissions including pre-existing open source code may be in other languages providing you can get them through our build system cleanly; we already have C and some assembly code in Simbody. However, any user-exposed API must be in C++ even if the internals are not.

#### Keep line width to 80 characters

Line widths should be no longer than \*\*80\*\* characters. The reason for this is that it permits multiple files to be laid out side-by-side during editing, which is \*really\* useful. At 80 characters you can get three windows on a modest-sized monitor, using a font that is readable even by adults.

It is best to use a "guide line" (a vertical line that marks column 80 while you edit) so that you can see where the limit is. If you are using Visual Studio, there is a very nice Editor Guidelines Extension available [here](https://visualstudiogallery.msdn.microsoft.com/da227a0b-0e31-4a11-8f6b-3a149cf2e459). If you don't have built-in guide lines available, note that the last line of the copyright block at the top of every Simbody source file is 80 characters wide.

Please don't interpret this to mean we like short lines. On the contrary it is nice to see as much code as reasonably possible on the screen at once, so don't stretch out your code vertically unnecessarily; and don't waste horizontal space where it doesn't help readability. Long comment blocks in particular should have lines as wide as possible. Just don't go over 80.

#### Do not use tabs to indent; use 4 spaces

There \*must not\* be any tabs in your code; our build system will reject them. They will look great to you, but in other viewers people will see your code as randomly formatted.

Please be sure that your code editor is set to replace tabs with four spaces. You should never allow tab characters to get into your code.

Your preferred editor almost certainly has settings to replace tabs with

spaces. For example, if you use Visual Studio, go to `Tools:Options:Text

Editor:C/C++:Tabs`, set `tab size=4` and `indent size=4`, and check the `Insert

spaces` button. In `vi` or `vim` use `set tabstop=4` and `set expandtab`.

Most editors can also help you clean up a file that already has tabs in it. In

Visual Studio, go to `Edit:Advanced:Untabify` to untabify the current

file; or `Find & Replace` with regular expressions turned on, using `\t` to

represent a tab, to untabify your entire project/solution. On UNIX in general, see the `expand` shell command.

In `vi` or `vim`, use:

```

:1,$s/\t/ /g

```

#### Use English and a spell checker

Simbody has users and contributors from around the world. However, there is much to be gained by choosing a single natural language for a project, and English is the obvious choice for us. Any submitted code must be understandable by English speakers, with English comments, error messages, and documentation. As one practical consequence, this means Simbody code can use `char` rather than `wchar\_t` (wide characters) and embedded English text to be displayed at run time is acceptable and need not be sequestered into separate files to facilitate translation. Simbody contributors thus do not need to be familiar with techniques for internationalization.

Please use correct spelling and punctuation, \*especially\* in comments that will become part of the Doxygen API documentation. It is tedious for reviewers to correct your spelling -- a spell checker is your friend and ours here. We know spelling and grammatical errors will creep in, but the fewer the better. If you are not a native English speaker, please just do your best -- we'll help.

#### Provide Doxygen comments

Some programmers think comments interfere with the pure beauty of their code; we are not among them. We would like to be able to understand your code, and especially appreciate useful things like citations to book sections or papers we can read that explain the theory. As usual though, we are much more concerned about the user-facing API than the internals. We use Doxygen to generate the API documentation from the code. We expect basic class documentation, and at least something for each publicly-visible member, using Doxygen-style comments which you can easily learn just by looking at existing code. Be sure to build the `doxygen` target (or `make doxygen`) if your code has a user-facing API and take a look at the results to make sure they are formatted well and make sense.

You can format your comments in any reasonable style (consistent within a source module, please). However, we would like to suggest that you forgo the old C-style comments where every line begins with ` \* ` (space, asterisk, space). Since comments are almost universally colorized now in every viewer, you don't need the asterisks to make them stand out. And that wastes three characters on every line out of the limited budget we allow. Consider formatting like this:

```cpp

/\*\* This is the doxygen brief description. This is the rest of the documentation

and when you get to the final line you can just wrap up on the same line. \*/

void theMethodYouAreDocumenting();

```

(The double asterisk is one way to signal a Doxygen comment.) That is compact and just as readable (when colorized) as this:

```cpp

/\*\*

\* This is the doxygen brief description. This is the rest of the documentation

\* and when you get to the final line you will feel obligated to eat up one

\* more line.

\*/

void theMethodYouAreDocumenting();

```

When you have short Doxygen comments to make about dozens of methods in a class, those two extra lines per method significantly reduce the amount of code you can squeeze onto one screen. The comments are harder to reformat also. The generated Doxygen documentation is identical either way.

#### Code should be `const` correct

One of the best features of C++ is the ability to write a method signature so that the compiler can guarantee that an argument or data member will not be modified. This is specified using the `const` keyword. A program which uses `const` wherever it is appropriate, and propagates constness throughout, is called "const correct." It is messy to take a non-const correct program and make it const correct later; that should be designed in from the start.

In addition to catching many otherwise difficult-to-find or worse, unnoticed, bugs const correctness can have a direct impact on performance. A large data structure which must not be modified can be passed by reference (i.e., by address) safely to a black-box routine that declares that argument `const`. Conscientious programmers who would otherwise copy the data to ensure its integrity do not need to do so in that case, providing a large savings in memory use and often in run time performance.

All Simbody software which is written in C++, or provides a C++ interface, must be const correct. We highly recommend this strategy for all programmers. It works.

#### Objects should be thread safe

Simbody libraries are supposed to be thread safe and new code should not violate that promise. But, that does not mean you have to write parallelized code that uses multiple threads (although you can if you want and you know how). What it does mean is that your code should not prevent \*other\* Simbody users from writing multithreaded programs that use Simbody. That is, if each of several simultaneously-executing threads allocates its own, non-shared object of one of your classes, those threads will not interfere with each other.

In practice, that means you must (a) avoid using global variables, and (b) think carefully about using static variables. Basically this means whatever you write should be wrapped up in a class, and you should use class data members for communication among the methods of your class rather than global variables.

If you are worried about thread safety, mention it in the relevant Issue or PR; we'll be happy to discuss it with you.

### Naming conventions

We do not believe it is helpful to attempt to encode type information into symbol names (for example, beginning pointer names with a `p`). Much of the need for such conventions has passed with the wide availability of IDEs offering language-sensitive code browsing and debugging, such as that provided by Visual Studio or Eclipse. We do not use name prefix characters to provide information that can easily be obtained while browsing code or debugging. We trust programmers to add appropriate conventions in their own code when those conventions are necessary locally for clarity or convenience, and to explain them in nearby comments.

We prefer consistency with existing precedent over our own conventions whenever appropriate. For example, C++ containers like `std::vector` define standard names like `const\_iterator` so if you are building a container intended to be compatible with one of those you should follow the existing precedent rather than use the Simbody convention which would have been `ConstIterator`.

#### Types: classes and structs, typedefs, enums

Type names should be nouns or noun phrases, using the `UpperCamelCase` naming convention. There should be no underscores in the names. Some examples:

```cpp

System

SimbodyMatterSubsystem

```

#### Constants

We reserve the ugly `ALL\_CAPS\_CONVENTION` for preprocessor macros both to discourage their use and to signal the type-unsafe loophole being employed. In particular, we discourage the use of preprocessor macros for constants and use a different, less violent convention for type-safe constants.

For constants defined within the language, using `enum` or `const`, use `UpperCamelCase` (same convention as for classes).

```cpp

enum Color {

Red,

Blue,

LightPink

};

static const Color AttentionColor = Red;

```

#### Functions and methods

Names should begin with a verb and use the `lowerCamelCase` convention.

```cpp

getBodyVelocity()

setDefaultLength()

```

We have some conventional starting verbs and you should use the same ones when they apply, and avoid them if your methods are doing something different:

verb | meaning

----------|---------

`get` | Return a const reference to something that has already been computed.

`set` | Change the value of some internal quantity; may cause invalidation of dependent computations.

`upd` | (update) Return a writable reference to some internal variable. May cause invalidation of dependent computations.

`find` | Perform a small calculation (e.g., find the distance between two points) and return the result without changing anything else.

`calc` | (calculate) Perform an expensive calculation and return the result. Does not cause any other changes.

`realize` | Initiate state-dependent computations and cache results internally; no result returned.

`adopt` | Take over ownership of a passed-in heap-allocated object.

#### Variables

Use generally descriptive noun phrases, expressed in `lowerCamelCase` (same as for methods).

Spell things out unless there is a good reason to abbreviate, and in that case abbreviate consistently.

```cpp

fileName

nextAvailableSlot

```

Follow other appropriate conventions in contexts where they improve readability: for example, you may prefer `x,y,z` for coordinates, `A` for a matrix, and `i,j,k` for indices.

We do not require that you give data members a distinguishing prefix. However, it is often helpful to do so in complicated classes, in which case the prefix should be `m\_`, prepended to names that otherwise follow the above convention. Do not use an initial underscore alone.

```cpp

m\_fileName

m\_nextAvailableSlot

```

Please do not use any other prefix conventions; many exist and none are widely agreed upon so they are not helpful to a mixed audience of readers.

#### Preprocessor macros

All caps, words separated by underscores. When these appear in interfaces they must be prefixed with a distinctive prefix to keep them from colliding with other symbols. Use an all-caps version of the associated name space when possible. The names of all macros from Simbody software are prefixed with `SimTK\_`.

```cpp

SimTK\_DEBUG

MYMODULE\_UGLY\_MACRO

```

#### Namespaces

Short, cryptic, low probability of having the same name as someone elses namespace. We reserve namespaces containing `SimTK` and `Simbody` (in any combination of upper/lowercase) for user-visible Simbody code.

```cpp

std::

SimTK::

```

In contexts where you can't use C++ namespaces, such as preprocessor macro names and external C functions, use a unique prefix like `SimTK\_` or `mymodule\_` in place of an actual namespace.

#### Header Guards

Header guards are preprocessor defines that surround every header file to prevent it from being included multiple times. Simbody header guards should be written like this:

```cpp

#ifndef SimTK\_MODULE\_SOME\_CLASS\_NAME\_H\_

#define SimTK\_MODULE\_SOME\_CLASS\_NAME\_H\_

// ... stuff ...

#endif // SimTK\_MODULE\_SOME\_CLASS\_NAME\_H\_

```

The initial `SimTK\_` should always be there; it is serving as a namespace to avoid collisions with other code. If you are using some other namespace, replace `SimTK\_` with yours. `MODULE` should be replaced by something defining a major grouping of code within Simbody; its purpose is to avoid collisions with other Simbody modules. Then `SOME\_CLASS\_NAME` is replaced by an uglified version of the main class defined by this header file. Some headers aren't associated with a class (like `common.h`); you can use the file name or something else descriptive instead. The final `\_H\_` is just there to keep us out of trouble.

\*\*Note:\*\* Embedded and trailing underscores (`\_`) are allowed in C++ names, but the C++ standard forbids user symbols that begin with an underscore or contain two adjacent underscores. (Those are reserved for use by the language system itself, such as for variable names inside the C++ standard header files.)

#### Calendar dates

The need for date and time stamps arises frequently enough, and causes enough trouble, that we want to state some general preferences here, although not specific requirements for any particular situation. Maybe this goes without saying, but just go ahead and use four digits for the year! Lets not go through that again. Compact date stamps such as those appearing in file names and source comments should have the form yyyymmdd, e.g. 20060322 which has the distinct advantage of being sortable, with the most significant part first. Code that formats friendly dates for user consumption should avoid ambiguous formats like 7/5/2005 (July 5 in the U.S. and May 7 in Europe). Instead, use July 5, 2005 or 5 July 2005 or 2005-May-07, for example.

For binary time stamps generated programmatically, please give careful thought to time zone independence.

### C++ tips and style guide

This section collects tips for staying out of trouble in C++, and documents some of our stylistic preferences. These are not in any particular order. Please feel free to propose more.

#### Public class members come first

Dont make people look at your dirty laundry in order to use your classes. Start with the basic constructors (and copy assignment in C++). Then put important likely-to-be-used methods first, relegating obscure bookkeeping stuff to the end.

Avoid public data members; use inline accessors instead. Even protected data members should be viewed suspiciously, especially if you expect people other than yourself to be deriving classes from yours. Occasionally this seems silly, especially for simple "plain old data" (POD) classes as described in the C++ standard. In that case you should at least put your public data members at the beginning of your class declaration so that they appear as part of the public interface rather than buried with the private stuff at the end.

#### Use anonymous namespaces

If you define classes or external functions in C++ source, even if they appear nowhere else, those names will be exported at link time and may conflict with other names. If that's intentional, make sure the names are in the `SimTK` namespace or begin with `SimTK\_`. If not, you should surround the declaration with an anonymous namespace:

```cpp

namespace {

// declarations that are private to this source file

}

```

That prevents the symbols from being exported and you can use any names for them that you want.

For functions you can achieve the same thing by declaring them `static` (which you must do if your code is in C) but anonymous namespaces in C++ are much more powerful.

#### Don't waste lines on curly braces

We do not like to see a lot of content-free lines using up vertical space in code and consequently prefer the style sometimes called "the one true brace" over conventions which attempt to align all paired braces. Here are some examples:

```cpp

if (a <= b) {

// some code

} else {

// some more

}

int myFunction() {

// function body begins here

}

class MyClass {

public:

// public members

};

```

When there is only a single statement within a control structure, there is no need for braces and we prefer that they not be used since that saves space. \*Indentation\* is the primary means for conveying code structure to human readers, so it matters a lot more that the indentation is right than where the braces are.

For small inline functions whose entire definition can be fit on one line (typical for "accessors"), we are happy to see them defined like this:

```cpp

const Thing& getSomething() const {return m\_thing;}

void setSomething(const Thing& thing) {m\_thing=thing;}

```

Many programmers think those are immoral; if that's you, feel free to use more lines. But we're glad to get these little methods over with and fully understandable with very little screen real estate.

#### Throw and return are not functions

In C++ `throw` and `return` are not functions. It is misleading to enclose their arguments in parentheses. That is, you should write `return x;` not `return(x);`. A parenthesized expression is not treated the same as a function argument list. For example `f(a,b)` and `return(a,b)` mean very different things -- the former is a 2-argument function call; the latter is an invocation of the rarely-used "comma operator".

#### \*Pre\*fer \*pre\*-increment and \*pre\*-decrement operators

Operators for both pre-increment (`++i`) and post-increment (`i++`) are available in C++. If you dont look at the result, they are logically equivalent. For simple types they are physically equivalent too. But for complicated types (like iterators), the pre-increment operators are much cheaper computationally, because they dont require separate storage for saving the previous result. Therefore you should get in the habit of using pre-increment (or pre-decrement) in all your loops:

```cpp

for (int i; i < limit; ++i) /\* <-- YES\*/

for (int i; i < limit; i++) /\* <-- NO \*/

```

This will prevent you from using the wrong operator in the expensive cases, which are not always obvious. Of course in cases where you actually need the pre- or post-value for something, you should use the appropriate operator.

#### Place pointer and reference symbols with the type

References and pointers create new types. That is `T`, `T\*`, and `T&` are three distinct types. You can tell because you can make `typedef`s like this:

```cpp

typedef T SameAsT;

typedef T\* PointerToT;

typedef T& ReferenceToT;

// and then declare

SameAsT t1, t2; // both are type T

PointerToT tptr1, tptr2; // both are type T\*

ReferenceToT tref1=a, tref2=b; // both are type T&

```

Therefore you should place the `\*` and `&` next to the type, not the variable, because logically they are part of the type. Unfortunately, the C language had a bug in its syntax which has been inherited by C++. A line like `char\* a,b;` is treated like `char\* a; char b;` rather than `char\* a; char\* b;`, but if you write `typedef char\* CharPtr;` then `CharPtr a,b;` declares both to be pointers. There is no perfect solution because the language is broken. However, there is no problem in argument lists (since each variable has to have its own type). So we recommend that you simply avoid the misleading multiple-declaration form when using pointers or references. Just use separate declarations or a `typedef`. Then always put the `\*` and `&` with the type where they belong. Here are right and wrong examples for argument lists:

```cpp

f(int I, string& name, char\* something) /\* <-- YES\*/

f(int I, string &name, char \*something) /\* <-- NO \*/

```

#### Avoid spaces that don't improve readability

Add spaces where they improve clarity, otherwise leave them out. In particular, parentheses do a fine job of surrounding `if` and `for` conditions and do not require further setting off with spaces. On the other hand, operators within those conditions are sometimes hard to spot and worth setting apart. For example, we prefer the more-compact versions below:

```cpp

if (nextItem <= minItemSoFar) /\* <-- YES\*/

if ( nextItem <= minItemSoFar ) /\* <-- NO \*/

for (int i=0; i < length; ++i) /\* <-- YES\*/

for ( int i=0; i < length; ++i ) /\* <-- NO \*/

```

You only get 80 characters per line; make them count!

#### Make assignment operators safe for self-assignment

You should let the compiler automatically generate the copy constructor and copy assignment operator for your classes whenever possible. But sometimes you have to write one. Here is the basic template for copy assignment:

```cpp

MyClass& operator=(const MyClass& source) {

if (&source != this) {

// copy stuff from source to this

}

return \*this;

}

```

A common mistake is to leave out the `if`. Since the "copy stuff" part often begins by deleting the contents of "this", a self assignment like a=a will fail without those lines; that is always supposed to work (and does for all the built-in and standard library types). Of course no one intentionally does that kind of assignment, but they occur anyway in general code since you dont always know where the source comes from.

If the "copy stuff" part consists only of assignments that work for self assignment, then you can get away without the test, but unless youve thought it through carefully you should just get in the habit of putting in the test.

List of Contributors

--------------------

This is an attempt at a complete contributor list; please submit a PR or file an Issue if you or someone else is missing, or to improve your contributions entry.

Real name | GitHub Id | Contributions/expertise

--------------------|---------------|-------------------------

Michael Sherman |@sherm1 |Lead developer; multibody dynamics

Peter Eastman |@peastman |Much early Simbody development; visualizer

Chris Dembia |@chrisdembia |Build, task space control, CMA optimizer, bug fixes & documentation

Thomas Uchida |@tkuchida |Rigid impact theory & code; documentation

Carmichael Ong |@carmichaelong |Pathname deconstruction with specified working directory

Thomas Lau |@thomasklau |Force Parallelization

Ian Stavness |@stavness |Computational geometry

Andreas Scholz |@AndreasScholz |Computational geometry

José Rivero |@j-rivero |Build, especially for Debian

Steven Peters |@scpeters |Build and VectorIterator improvements

John Hsu |@hsu |Bug fixes; iterative solver & contact theory

Nate Koenig |@nkoenig |Bug fix

Ayman Habib |@aymanhab |Bug fixes; visualization, SWIG improvements

Ajay Seth |@aseth1 |Mobilizer theory and code

Jack Wang |@jmwang |Bug fixes; visualization improvements

Tim Dorn |@twdorn |Bug fixes

Apoorva Rajagopal |@apoorvar |Xcode build fixes

Kevin Xu |@kevunix |Build fix

Antoine Falisse |@antoinefalisse|Memory issues for Matrix; smooth contact

Gil Serrancoli |@gilserrancoli |Smooth contact

Guillaume Jacquenot |@Gjacquenot |Build instructions for MinGW

Thomas Beutlich |@tbeu |Fix many typos and spelling errors

Julien Nabet |@serval2412 |Code style & safety improvements

Elena Ceseracciu |@elen4 |Improved dependency resolution

Kevin He |@kingchurch |Bug fixes

Yuri Victorovich |@yurivict |Improved support for FreeBSD

Martin Gejke |@mgejke |Bug fixes

Paul Mitiguy |@mitiguy |Rotation class; dynamics

Matthew Millard |@mjhmilla |Bicubic spline

Jack Middleton | |Numerical methods

Christopher Bruns |@cmbruns-hhmi |Molmodel

Randy Radmer | |Molmodel

Charles Schwieters | |Provided initial multibody code

Abhinandan Jain | |Underlying spatial algebra formulation

Isaac Newton | |F=ma, calculus, etc.

Jason Blum |@jjblum |Vis bug fix

Contributor License Agreement

-----------------------------

Simbody is licensed under the very permissive [Apache 2.0 license](http://www.apache.org/licenses/LICENSE-2.0). Simbody users are not required to follow our noble egalitarian principles, nor to share their profits with us, nor even to acknowledge us (though they often do). When you make a contribution in any of the ways described above, you are agreeing to allow your contribution to be used under the same terms, adding no additional restrictions to the Simbody project nor requirements on Simbody users.

Specifically, by contributing you are agreeing to the following terms:

1. The code, text, or images you submit are your original work (you own and retain the copyright) or you otherwise have the right to submit the work.

2. You grant the Simbody project, developers, and users a nonexclusive, irrevocable license to use your submission and any necessary intellectual property, under terms of the Apache 2.0 license.

3. No part of your contribution is covered by a viral ("copyleft") license like GPL or LGPL.

4. You are capable of granting these rights for the contribution.

If your contribution contains others' open source code licensed under Apache 2.0 or other non-viral license like BSD, MIT, or ZLib, it is probably fine. But be sure to mention that in the Pull Request you submit so we can discuss it.