This is the to do list for the loader

# Add file archiving

## Refactor *Token\_Stream* and create *the Token class*

Currently *Token\_Stream* is responsible for reading chars out of files echoing them, and breaking them into tokens. We now want to increase the complexity of the echoing process to include file archiving of run files. Lest this class becomes too complex you should separate the (echoing functions) from the tokenizing mechanism.

We will farther simplify Token\_Stream by moving its indent management into a class of its own and creating Token\_Stream by inheriting from this class.

Indent\_Management will be used to track the number of indents.

### Token

In order to allow more advanced mechanisms to generate high quality trace files it will be necessary to move the echoing functionality to the output end of the token stream system. Thus it will be necessary to pass currently discarded information up through the token stream system. To do this we will need a new class called Token which will not only contain the data that is currently passed up through the Token\_Stream system but also the characters that came before it.

I have gone ahead and implemented the Token class but you still have to make the system use it.

First of all it will have to be included in all the Stream classes.

Then all the get\_next functions will have to be modified to use it.

The initial get\_next function Token\_Stream::get\_next() will have to create it from scratch. Its state machine will have to be reworked to put all the proceeding “white space” characters into the format field of the Token and all the actual token characters into the data field.

The output generating mechanisms in this class can be commented out but you may want to keep them around as we will be moving them to another class.

The overloads of *Decoded\_Stream::get\_datum* will all have to be modified a little to take the data field of the Token.

All these function should be converted to functions that do nothing :

open\_dump\_file this one should have its argument changed to:

bool open\_dump\_file(const string &file\_name, int i = 0)

close\_dump\_file

set\_output\_file

highlight\_on

highlight\_off

This function should be set to do nothing

report\_error

This function should be kept

set\_echo

and you should add a complement

get\_echo that returns the flag

## Converting Streams into templates

To add other features into the token stream system we will need to add more classes into the inheritance hierarchy so we should modify the stream classes to make them easier to reconfigure.

Token\_Stream will remain unchanged for this process as will Decoded\_Stream however we will make the intermediate streams into templates which specify there parents as templated arguments.

For instance:

*class Single\_Push\_Back\_Stream: public Echo\_Stream*

will become:

*template<class PARENT>*

*class Single\_Push\_Back\_Stream: public PARENT*

This will require that the contents of these classes’ .cpp files be moved in to the .h files.

Additionally it will require that Decoded\_Stream specifies the entire stack of stream classes it wants to use.

The declaration for Decoded\_Stream will end up looking something like this:

class Decoded\_Stream: public Single\_Push\_Back\_Stream<Echo\_Stream<Include\_Stream<Comment\_Stream< Token\_Stream>>>>

This lets us add or remove components quite easily without having to make modifications throughout the code. Additionally it makes it possible to reuse components like Single\_Push\_Back\_Stream which we may need to do.

Some #includes should be removed from these classes as they no longer need to know who their parents are and this might help us catch errors when references to parents are not correctly changed.

## Modifications to comment stream

Comment stream must be modified so that when a comment is detected it is put into the format portion of the token it eventually passes on.

## Echo\_Stream

Echo\_Stream will only need its parent turned into a template value.

The function of this class is only to intercept echo directives and to toggle the echo control flag accordingly.

## Add a new classes to echo data

These classes all implement fairly simple get\_next functions that just pass the results of the get\_next function of their parents, but also echo it out to some other destination.

### Stdout\_Stream\_Echo

* This class sends the tokens out to the standard output.
* It also implements report\_error in a similar manner

### Html\_Stream\_Echo

* This class sends the tokens out to an html file
* It also implments report\_error, highlight\_on and highlight\_off
* This class implements set\_output\_file(HTML\_Dump \*newstream, int i = 0);
  + If i != 0 then when it opens the file it inserts i between the body of the file name and the extension
  + perhaps it would be easiest if it we assumed that the string: newstream did not have the extension, so and you should add this extension after you insert the number
  + finally the set\_output\_file should call the parents overload of set\_output\_file incromenting the i value before calling it.

This lets us use Html\_Stream\_Echo multiple times within the a stream stack so we can get observe the stream after points.

The implementation of set\_output\_file will look something like this

bool set\_output\_file(HTML\_Dump \*newstream, int i = 0);

{

dump\_file = new HTML\_Dump;

if(i == 0)

{

dump\_file->open(file\_name.c\_str()+ “.html”);

}

else

{

dump\_file->open(file\_name.c\_str()+ i +“.html”); // you may need to convert something

}

bool opened = dump\_file->is\_open();

return = opened && PARENT::set\_output\_file(newstream, i+1);

// call the parents implementation giving a different i value so that other copies of Html\_Stream\_Echo write to different files

}

The idea is that if you put this class in the stack more than once it will write to different files. This should aid you in debugging the stream system.

### Archive\_Stream\_Echo

* This class sends its tokens to an archive copy of the file.
* It has a function called open\_Archive (const sting &new\_file\_name ,const string & header\_comment)
  + When this is called the file is opened and a comment listing the old file name is written
* The file is closed when the object is freed

This class also maintains an internal variable called last token. This is used to delay the writing of paths out to the archive file until the next token is requested. If the user of this class determines that the token it just got was a path it calls a function called rewrite\_Last\_in\_Archive(string). This causes the data portion of the delayed token to be replaced.

The algorithm for the get next function looks something like this:

get\_next()

{

send old\_token to archive file if file is open

old\_token = PARENT::get\_next();

return old\_token;

}

rewrite\_Last\_in\_Archive(const string &s)

{

old\_token.set\_data(s);

}

## Run\_File\_Archive\_Director

The *Run\_File\_Archive\_Director* class manages file names for the archiving of run files.

1. It devises names for the archive files so that they don’t overwrite one another if two different files in different places are used that have the same name. EG *foo/stuff.txt* and *bar/stuff.txt* -> *stuff\_1.txt* and *stuff\_2.txt* rather than just one of them being stored in stuff.txt in the archive.
2. It insures that run files are only archived once potentially saving some amount of disk access.
3. And it keeps track of the translation between archive file names and source file names to aid in the altering of file names when they occur within run files themselves, such as after a *#include* statement.

*void set\_Destination\_Directory(const File\_Path* *&destination\_file\_path)*

This function lets one set where the data is to be stored. A directory with the given path must be created before the names generated by this class can be used.

*string get\_Destination(const string & source\_file)*

This function returns the name where a file should be stored. If the file has not been seen before it gives it a new name and adds the translation from source file to destination path to its translation table.

*bool is\_New\_File(const File\_Path & source\_file) const*

This function returns true if the input has not been presented to the object before.

This class will be used in 6 different places.

1. To make a copy of the root run file. That is to say the run file that is initially given to the loader.
2. To make copies of files which are included into this initial run file.
3. To copy files which are specified in the run file but are not opened by the loader mechanism
4. To rewrite the file path of the initial run file given in the archived command line.
5. To rewrite the paths of files that are specified in #include statements.
6. To rewrite the paths of files that are not managed by the loader.

Generally the a function will need to do both a file copy and a name rewrite at the same time. So it will first call *is\_New\_File* to see if the file should be copied and then it will call *get\_Destination*  to generate the destination for the file if it is needed and the name that it should use in the rewrite. We will talk more about the use of this class in the subsection below.

In the current design: it is assumed that the file archiving process will begin or at least its name will be saved before the file’s name is rewritten in other files. Based on this assumption the *Run\_File\_Archive\_Director* class does not maintain any explicit mechanism to indicate whether a file has been copied or not. Instead all it knows is whether a file path has been seen before. If the logic that uses this class cannot guarantee that, just because the *Run\_File\_Archive\_Director* has seen the file means that at has been copied, or it is being copied (or at least the relevant paths have been stored for latter copying), then the *Run\_File\_Archive\_Director* class may have to be modified to keep track of whether a file has been copied. Hopefully we can live without this feature. It will make it necessary to gather the path translations necessary to create the command line archive after the loader has completed.

To perform these functions this class will have to maintain a number of private data structures and functions.

* It will need to store the destination path.
* And it will need some kind of look up structure or structures to make the path translations and check for other conditions.
* It will also need a counter of some kind to generate new file names

The *get\_Destination* function contains really the only algorithm in this class.

Static int i = 1;

if(*source\_file\_path* is in the database)

return database entry

while(*source\_file\_path.name* is in archive directory)

add “\_” + i to the end of the destination archive

i++

add the new mapping to the data structures.

The while loop insures that archived source files are not over written when two files with the same name are archived. For instance consider what would happen if we had two included files both called *stuff.txt* . On resides in the directory foo while the other is stored in bar. Without this mechanism one would be copied and then the over would be written over it. With this mechanism however when the second file is about to be copied it detects the second file name in its database and rewrites the second files name as *stuff\_1.txt.* Now if there is a third file called *stuff.txt* it will be rewritten as *stuff\_2.txt* . A subsequent file named *stuff\_1.txt* will be rewritten as *stuff\_1\_3.txt*.

The table below illustrates and explains how a sequence of files are renamed by the algorithm will rename files. The fact that it is a sequence is very important.

|  |  |  |  |
| --- | --- | --- | --- |
| Initial file name | Resulting file name | Reason | Value of the static i |
| Stuff | Stuff | It is unique and does not need to have its name changed to be added to the archive | 1 |
| Stuff | Stuff\_1 | A file named *Stuff* is already in the archive directory so append the value of *i* to it and increment *i* | 2 |
| Stuff | Stuff\_2 | A file named *Stuff* is already in the archive directory so append the value of *i* to it and increment *i* | 3 |
| thing | thing | It is unique and does not need to have its name changed to be added to the archive | 3 |
| Stuff | Stuff\_3 | A file named *Stuff* is already in the archive directory so append the value of *i* to it and increment *i* | 4 |
| thing | Thing\_4 | A file named *Thing* is already in the archive directory so append the value of *i* to it and increment *i* | 5 |
| Stuff\_1 | Stuff\_1\_5 | A file named *Stuff\_1* is already in the archive directory so append the value of *i* to it and increment *i* | 6 |
| Stuff\_2 | Stuff\_2\_6 | A file named *Stuff\_2* is already in the archive directory so append the value of *i* to it and increment *i* | 7 |
| Stuff\_1 | Stuff\_1\_7 | A file named *Stuff\_1* is already in the archive directory so append the value of *i* to it and increment *i* | 8 |
| Stuff\_8\_9 | Stuff\_8\_9 | It is unique and does not need to have its name changed to be added to the archive | 8 |
| Stuff\_8 | Stuff\_8 | It is unique and does not need to have its name changed to be added to the archive | 8 |
| Stuff | Stuff\_8\_9\_10 | A file named *Stuff* is already in the archive directory so append the value of *i* to it and increment *i.*  But on the next iteration of the while loop it finds that the file *Stuff\_8* is already in the archive directory so it appends the value of *i* to it and increment *i.* this still does not work so it repeats the process until it gets a unique name. | 11 |

The algorithm is somewhat simplistic but it is adequate for our purposes.

## Modifications to Include\_Stream

Include\_Stream will need to be modified in a number of ways.

Will need its parent turned into a template value.

All CCP code brought into the .h file and the cpp file removed

As before Include\_Stream will maintain a pointer to a secondary Include\_Stream and it will create it when it encounters the #include directive and then proceed to draw data from it.

It will also need to instruct the secondary how to echo its tokens. It should do this as follows:

1. Make a copy of the original Include\_Stream.

So we would write *\*secondary = \*this* or something to this effect. It is uncertain if this will work or not. The goal is to be able to copy the state of whatever classes are put into the inheritance list of Include\_Stream without having to know exactly which data elements it contains. Some experimentation may be needed here.

1. Then you need to get the secondary to open up a new source file
2. Finally you need to figure out where the archive file if any is but not set this if the file has already been archived. The Run\_File\_Archive\_Director will provide this information.

You also need to provide an overload of rewrite\_Last\_in\_Archive which will alternately call the same function of the parent class or the secondary.

Include\_Stream::get\_Next will also have to call rewrite\_Last\_in\_Archive when it encounters the #include directive. When a path name is changed in a file a comment should be written after it to specify where it came from.

Also when a file loader controlled run file is archived a comment should be placed at its start specifying where it came from.

## Modifications to the Loader system

* A new class of *Loader\_Link*s will have to be created.
* We will call this class ***File\_Path\_Loader\_Link***.
* The constructor of this will take a pointer to STL string.
* The *load\_s* for this class will read the string much as Native\_Loader\_Link does except that it will use the *Run\_File\_Archive\_Director* and new calls in the Token\_Stream dynasty to rename paths in the run files. Additionally it will need to copy these files it self.
* File\_Path\_Loader\_Link will also need a pointer to the *Run\_File\_Archive\_Director*  and a function to set this variable.
* The class *Loadable* will then need a new registration function that takes the tag and a pointer to the string where the file name is supposed to go.
* *Loadable* will keep the master copy of the *Run\_File\_Archive\_Director*  object in a private static variable.
* *Loadable* will need a function to initialize this variable from a destination path
* *Loadable* will also give this object the source directory when it is first called so there will also have to be some mechanism to prevent this from being set more than once.
* A pointer to this variable will also have to be passed down to the *Token\_Stream* dynasty so that proper archives are created.
* *Loadable* will need to pass pointers to this object to *File\_Path\_Loader\_Link***s** so that
* If the destination is not provided to *Loadable* then it should work normally but without creating an archive of any of the run files.
* A function will have to be provided so that the calling class can get the name of the file where archived run files are stored. Rather than just giving it access to the primary run file archive directory this function should be able to take any file name that is involved with the run file set. If a the path name is new then the function should not add it to the list of translations but rather return an empty string. static *File\_Path* *Loadable::get\_Archive\_Path(const File\_Path &source).* This function will be used to write an archive version of the command line.

## Modifications to specific loaders

* In the hand full of cases where the file paths are loaded the new registration function will have to be called.

For example the path to the bata database is currently registered as string. This must be changed so that this variable is registered as a path.

* Also the name of the archive path will have to be passed into *Loadable.* For initial testing this can just be hardwired.

That is to say we need to have some means to specify where everything is specified when we test the code. As far as the loadable class is concerned this is simply passed to it. Other documents describe how this value is set and stored.

# Make function names camel case

# Add header comments

# Parameter default checking

## Implement loader files

## Check loader link code

## Use in load functions

# Parameter range checking

# Poly loader

# FlashBack

Flashback is the process by which the loader takes the registration information given to it but instead of using this to generate a GUI interface. At the time of writing the value and priority of having a GUI has not yet been determined so the flash back mechanism may or may not actually be needed in the end.

If Flashback is implemented one will make a call to put the loader into a particular mode and specify an output file. Then one will call the loader of the root of the primary loader function. When in this mode this function will rather than loading a scenario write details of what the load file is expected to contain out to some kind of a file that contains the appearance and behavior of GUI based run file editor.

This GUI control file may be in a fully fledged source code that can be compiled or interpreted by an existing system or it may be a proprietary code that can only be read by a special program we will also have to create. Some kind of web browser based solution may be the best option.

At the time of writing limit checking, default setting and other checks of the run file are done in the loader that is written for each object. In the future we hope to migrate much of this functionality to the loadable class. Under this regime the loader that is written for an object will specify things like the limits of a value as well as its default setting when it registers the variable with the loadable class mechanism. During a normal loading operation the loadable class will use this information to check the input parameters. If the loadable class is in flashback mode (WORK IN PROGRESS)