# NOTE 196
# Notes on using the g++ compiler to compile AIPS++ code

Ralph Marson, NRAO

1 May 1996

A postscript version of this note is available. *(updated October 15, 2006) with assistance from Mark Calabretta, Mark Wieringa, Shelby Yang, Brian Glendenning & Wim Brouw*

## 1 Motivation

The main reason I had for using the g++ compiler was that the template instantiation bug in the Sun native compiler was really getting to me. It meant for me that the time to compile nearly identical pieces of code could vary from $\sim$ 40s to $\sim$ 10mins, depending on whether the compiler (for unknown reasons) decided to re-instantiate everything or not. Sun is aware of this bug and a recent patch has alleviated this problem somewhat.

In g++ the template instantiation is done by hand (with assistance from the aips++ system) so this problem cannot occur. On a line by line basis the g++ compiler is slower than the Sun native compiler. In practice (my experience, and in the time taken for system rebuilds) it appears to be faster, because it does not do excessive recompilations.

Because of these problems with the standard Sun compiler and to aid porting of AIPS++ to other architectures, g++ has become the standard C++ compiler for AIPS++.

This note describes details on compiling AIPS++ code that are specific to using g++ and the "do it yourself" template instantiation mechanism that is used to specify which templates are created.

Code compiled and optimised under g++ is in my experience about 30% slower than code compiled and optimised under the Sun native compiler. Currently we are investigating using the using the "do it yourself" template instantiation mechanism in conjunction with the latest release of the Sun compiler and other compilers.

## 2 g++ system setup

These notes assume you have the g++ compilers installed and aips++ setup to use it. I know nothing about this and you should contact (in order) either, Pat Murphy or Brian Glendenning (bglenden@nrao.edu) on how to do this. The g++ compiler is now available at all of the consortium sites, and version 2.7.2 has been extensively used at Socorro. These notes relate to that version.

## 3 Switching to g++

Most sites are setup to have the gnu compiler as the default compiler.

If not you can switch to to the g++ compiler using:

```
aipsinit gnu
```

This assumes that you have already run aipsinit to get the normal AIPS++ environment variables set.

## 4 Setting up your workspace

Ensure that the following directories exist (otherwise create them)

```
${MYAIPS2ROOT}/sun4sol_gnu
${MYAIPS2ROOT}/sun4sol_gnu/aux
${MYAIPS2ROOT}/sun4sol_gnu/bin
${MYAIPS2ROOT}/sun4sol_gnu/lib
${MYAIPS2ROOT}/sun4sol_gnu/bindbg
${MYAIPS2ROOT}/sun4sol_gnu/libdbg
```

where `$MYAIPS2ROOT` is the root of your workspace. (eg. `$HOME/aips++`).

At Socorro where the code is on one partition and the executables are on another (which is not backed up), you should substitute for `$MYAIPS2ROOT` the path `/tarzan2/$USERNAME`, and create a symbolic link pointing from `/tarzan/$USERNAME/sun4sol_gnu` to `/tarzan2/$USERNAME/sun4sol_gnu`

## 5 Compiling

Compiling should be done in the normal way (ie. using gmake in the source directory), and with all the normal flags (eg. OPT=1). Using gmake, in my experience, with the OPT=1 flag results in faster compilation as well as optimised code. But the fastest way to recompile your code is to use the OPTLIB=1 option. This links your code (compiled with debug flags) against the optimised libraries, and compiles about 2-3 times faster than not using any flags. It has the added advantage that you can use the debugger to step through your code (but not the system code). Using gmake without any flags also results in executables that are about ten times bigger than the optimised version (ie. they are huge).

Debugging should be done using gdb (use ddd or xxgdb for a graphical interface), rather than the normal system debuggers (ie. dbx and friends). Note that the commands differ from the standard ones (e.g. no 'stop' but rather 'break'). Help can be obtained by typing 'help' or 'help subject', while information is available with the 'info' command (see 'help info').

If gmake dumps core it may be because you are using a very old version that exists in `/opt/local1/bin` (Socorro only).

## 6 Upcasting Bug

The g++ compiler cannot do a few things the Sun native compiler can. In particular if a global function with templated arguments of some base class is called with a class derived from the base class it will not find a match. (i.e. it cannot do upcasts)

This problem is particularly endemic in the Arrays module and for these classes a particular solution has been devised. The Array class (and hence derived classes) have an upcasting member function called `arrayCast()` (or `ac()` for short).

For example in the aips/Arrays/ArrayMath.h class there is a function

```
T max(const Array<T> &a)
```

which will not compile when called with a vector as an argument. ie.

```
Vector<Float> vec(4);
cout << max(vec) << endl; // error flagged on this line!
```

but will compile if rewritten as:

```
cout << max(vec.arrayCast()) << endl;
```

This solution gets particularly useful when doing arithmetic on Vectors because g++ will complain about:

```
Vector<Float> v1(4), v2(4);
v1 += v2;
```

And this should be changed to

```
v1.arrayCast() += v2.arrayCast(); // make the cast explicit
or
v1.ac() += v2.ac(); // emphasize the arithmetic not the cast
```

This problem has also been recently seen in the Lattice classes and a corresponding set of member functions called `latticeCast()`, or `lc()` for short, have been written.

On rare occasions you may need to selectively do an upcast when a templated argument can be either a builtin data type (Int/Float etc.) or an Array (Vector/Matrix etc.). For these occasions the functions `at_c()` & `at_cc()` exist. These global functions only upcast Arrays and leave the builtin data types alone. They are defined in the Array, Vector, Matrix & Cube classes[1], and in aips.h for the builtin data types (and in String.h also).

The difference between the two functions is that `at_c()` is takes a non-constant input arguement while `at_cc()` takes a constant input arguement. Two function names are used rather than overloading just one because of what we think is a bug in the g++ compiler.

An example of this function in use in is the Measures module where a Quantum has standard templates of `Double` and `Vector<Double>`. Then the floor function is defined in QMath.cc as:

```
template <class Qtype>
Quantum<Qtype> floor(const Quantum<Qtype> &left) {
    Qtype tmp = left.getValue();
    Qtype ret;
    at_c(ret) = floor(at_c(tmp));
    return (Quantum<Qtype>(ret,left));
}
```

If you see this problem in other class hierarchies, you can do an explicit type cast but are instead urged to create an upcasting member function in the base class. The advantage of using member functions, rather than

[1] you need to look at the .h files to see the definition as they do not show up in the .html files

explicit casts, is that if g++ is ever able to do upcasts in the future it will then be easier to search for the upcasting functions and remove them. Please tell me if you do this, so I can add it to the documentation.

It is expected that this bug will disappear in g++ version 2.8.0 (private communication between Brian and one of the g++ developers).

# 7 Typedef Bug

The g++ compiler sometimes complains it cannot find a type definition even when it has definitely been included. For example I found that it was complaining that `Vector<Int>` was undefined even though `aips/Arrays/Vector.h` was included. To solve this add the line:

```
typedef Vector<Int> gpp_vector_int;
```

near the top of the .cc file. the type `gpp_vector_int` is arbitrary but this seems like a good convention (Brian's convention).

This appears to occur when a specific templated types used in the context of a general template definition. eg.

```
template <class T> const Lattice<Bool> & PagedImage<T>::mask() const
```

will trigger this bug because requiring a typedef for `Lattice<Bool>`

I enclose this change and others that are specific to the gnu compiler in ifdefs's to avoid getting other compilers off side. eg.

```
#ifdef __GNUG__
   typedef Vector<Int> gpp_vector_int;
#endif
```

It is not necessary to do this when using the array upcasting functions (arrayCast() etc.) described in section 6.

The error messages that indicate you are being affected by this bug can be quite obscure and some are listed in section 11

# 8 Do it yourself templates (creating a templates file)

When linking the aips++ system will have a large number of template instantiations already done for you. However there may be some template instantiations you will have to create for yourself. This is done with a personal template instantiation list. To do this create a file called `templates`. This should be in the directory containing the test program for the class you are currently working on (eg., `.../code/trial/implement/MyModule/test/templates`) or in the applications directory (eg., `.../code/trial/apps/MyApp/templates`). gmake will see this file and "knows" how to create the templates you have specified. You should *never* create a templates file in a code module directory (eg. `.../code/trial/implement/MyModule/templates` as the templates in this file will overwrite the system templates (when you check the templates file in).

The format of the template entries is described in detail in Chapter 9 of the AIPS++ system manual (the mkinst section). I will only outline the basics here.

The templates file should contain entries like:

```
1010 trial/Arrays/Convolver.cc
     template class Convolver<Float>
1000 aips/Arrays/ArrayMath.cc aips/Arrays/Array.h
     template Double min(Array<Double> const &)
```

The entries are:

**Identification number:** The exact number is arbitrary but it should be a unique number for each .cc file, and should be (by convention) a 4 digit number. An value of four zeros will be converted to a unique number by "reident" (described below).

**template file(s):** The .cc file containing the template. If using templates like `myClass< Array<Float> >` you should also include a .h file containing the definitions of the template arguments. In this example `trial/Utilities/myClass.cc aips/Arrays/Array.h`

**template:** A keyword separating the filenames from the required template. This usually goes in a separate line although it is not manditory to do this.

**required template:** This states which specific template is required to be instantiated. It can either be a class or a global function. An example of each is shown above.

See the system template instantiation list in:
`.../code/{aips,trial}/implement/_ReposFiller/templates` for further examples and a listing of the system generated templates. These templates are the ones necessary to link the aips++ applications.

Templates that are used by the test programs should not be put in the system file (unless there is an overriding reason), but stored in separate templates files that exist in the test directories. This keeps the aips++ libraries at a manageable size and avoids excessive link times.

If your templates file is getting long and unmanageable there is a utility called `reident` to sort/uniq, re-sequence and "canonicalise"the templates file.

In the default mode[2] reident will do the following:

- Adjust the identification number on the template definitions so that they are unique. It will also add identification numbers if they are missing and replace the identification numbers for all templates below 1000.
- Convert builtin types to AIPS++ standard definitions. eg., float will be converted to Float.
- Remove unnecessary white space.
- Warn if the const keywords are incorrectly positioned
- Warn if global functions do not have a return type
- Check that `#ifdef` and `#else` and `#endif` are properly sequenced.
- Comment out bad templates entries.

Its output is always to the file called templates (and input and output *can* be the same file) so I usually just type

```
reident
```

as the default input file is also called templates and I have already appended my changes tothis file.

reident can also merge multiple template files together so that

```
reident templates templates.additions
```

[2]There is an old mode enabled using the `-z` option which adds identification numbers starting at 1000 for each .cc file and increments by 10 to allow sufficient space for inserting new entries. You should rebuild your object library from scratch after running reident using this mode

will merge both input files into the output file.

The shell comment character # can be used to comment entries in your templates file and the preprocessor directives #if, #if defined(symbol), #if defined(symbol)!, #ifdef, #ifndef, #else, #endif can also be used.

Each entry in the templates file gets compiled into an object file called myClass_Ident.o where the myClass is the name of the first .cc file in the template line and Ident is the indentification number. Normally this object file contains only one instantiation. But often templates instaitions are required in groups. For example to use a CountedPtr<SkyCompRep> requires you instantiate four other classes, namely: CountedConstPtr<SkyCompRep>, PtrRep<SkyCompRep>, SimpleCountedConstPtr<SkyCompRep> & SimpleCountedPtr<SkyCompRep>. In this case it is better to group all five instantiations in one object file. This is done by adding multiple template lines as shown below.

```
1000 aips/Utilities/CountedPtr.cc trial/ComponentModels/SkyCompRep.h
     template class CountedConstPtr<SkyCompRep>
     template class CountedPtr<SkyCompRep>
     template class PtrRep<SkyCompRep>
     template class SimpleCountedConstPtr<SkyCompRep>
     template class SimpleCountedPtr<SkyCompRep>
```

## 9 Demangling the template names

When linking the names of the missing class/function instantiations may be printed in a mangled form. To demangle them save the g++ output in a file and use /usr/local/gnu/bin/c++filt.
An example (Using csh) is:

```
gmake test.cc >&! g++.out
/usr/local/gnu/bin/c++filt < g++.out
```

or using sh

```
gmake test.cc 2>&1 | /usr/local/gnu/bin/c++filt | tee missing-templates
```

The c++filt command will translate the following mangled name

```
getArray__Ct11MaskedArray1Z14PointComponen
```

to

```
MaskedArray<PointComponent>::getArray(void)
```

and because the getArray function is a member function of the MaskedArray class we need to instantiate the whole class (MaskedArray<PointComponent>)

## 10 Checking in your templates

When code is checked in, the templates entries should be added to the relevant system/test template instantiation file:

```
.../code/{aips,trial}/implement/\_ReposFiller/templates
```

if the templates are associated with an application or

```
.../code/{aips,trial}/implement/yourModule/test/templates
```

if the templates are associated with a test program. [3]

Be very careful when adding your templates to the system file as any duplication of the indent. number between your added templates and the already defined ones will cause the next system rebuild to fail (a sure way to become popular). Mark Calabretta suggests:

"When you add idents to the system templates file you should always run 'reident' before checking it back in. A good strategy is to give your new entries a 0000 ident."

A question that is often asked is "when do I put a template in the aips package and when do I put it in the trial package?". If any template required for an application uses code (.cc or .h files) only from the aips package it should go in the `aips/implement/_Reposfiller/template` file. If it uses anything from the trial package it should go in the `trial/implement/_Reposfiller/template` template file. This avoids the problem of having undefined symbols when some purely aips templates are defined in the trial library (at the expense of having a larger than necessary aips library).

Pure aips templates that are needed for test programs in the aips package should still be in `aips/implement/yourModule/test/templates`. The aips test programs should not rely on anything in trial, they are not linked against the trial libraries, and should not use any templates from the trial package.

Any templates (pure aips or involving trial) that are needed for test programs in the trial package should still be in the `trial/implement/yourModule/test/templates` file. This may mean that you may have to use the `EXTRA_PGMRLIBS` flag when compiling these test programs, although we will attempt to automate this.

You should also ensure that the templates you are adding to the system are not defined alsewhere. Duplicate templates lead to link time errors on many systems (eg. Dec Alpha). The policy for checking for duplicates is:

[3] If you are modifying an existing class and you find you need to add additional templates to the test program it is likely that these same templates will be used by other programs using this class. So you will need to go and find them and if they are applications then just put the templates in the system (_Reposfiller) templates file, otherwise you will need to modify the templates file of other test programs that use the modified class

- If checking new templates into `aips/implement/_Reposfiller/templates` then you *must* ensure that the same template is not duplicated in any of the following template files:
  - `aips/implement/anyModule/test/templates` (remove templates from test if any found)
  - `trial/implement/anyModule/test/templates` (remove templates from test if any found)
- If checking new templates into `aips/implement/anyModule/test/templates` then you *must* ensure that the same template is not duplicated in
  - `aips/implement/_Reposfiller/templates`. (template in test should not be necessary)
- If checking new templates into `trial/implement/_Reposfiller/templates` you need to check
  - `trial/implement/anyModule/test/templates` (remove templates from test if any found)
- If checking new templates into `trial/implement/anyModule/test/templates` then you need to check:
  - `aips/implement/_Reposfiller/templates` (template in test should not be necessary)
  - `trial/implement/_Reposfiller/templates` (template in test should not be necessary)

Brian Glendenning has a tool that can automatically check for template duplicates. It is checked into the system and information on how to use it can be obtained by typing `duplicates -help`

## 11 Error Messages

The gnu compiler can generate many different error messages that are in fact caused by the problems described in section 6 and 7. Here I intend to list the error messages reported for each of the problems as an aid to programmers. Please forward new error messages to me (rmarson@nrao.edu).

### 11.1 Upcasting Bug (section 6)

- type unification failed for function template
- too few arguments for function

The upcasting bug has also been known to cause an exception to be thrown. This occured in the following piece of code:

```
template <class Qtype>
ostream& operator<< (ostream &os, const Quantum<Qtype> &ku) {
    os << ku.qVal;
    return os;
}
```

where ku.qVal could be a `Double` or a `Vector<Double>` (but the `operator<<` function is defined for Arrays not Vectors). The compiler did not detect that there was an upcasting problem and examination with a debugger revealed it was getting into an infinite constructor loop! Further investigation revealed that the compiler did not find a function match using an upcast but was able to find another unintended match, that was further resolved to the original function!

### 11.2 Missing Typedef Bug (section 7)

- invalid use of undefined type
- parse error at null character

### 11.3 Other Gotcha's

You will probably have problems linking when you modify a class and then create a new set of templates (all in your private workspace), if the same templates also exist in the system library. This is because the newly modified templates may not be required until after your private library has been searched and the linker is using the system library. By this stage it will not know to use the templates in your personal library and instead pickup the system ones. The solution to this problem is.

- Wait for a system update. This may be overnight or take a week.
- Copy the system library into your own workspace prior to compiling the templates.