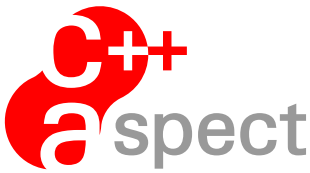

Documentation:
AC++ Compiler Manual

The AspectC++ Developers

Version 2.5

January 16, 2026

Contents

1 Introduction

The program ac++ is a compiler for the AspectC++ programming language. It is implemented as a preprocessor that transforms AspectC++ code into ordinary C++ code. During this transformation aspect code, which is defined by aspects, is woven statically into the component code. Aspects are a special AspectC++ language element, which can be used to implement crosscutting concerns in separate modules. Aspect definitions have to be implemented in special "aspect header files", which normally have the filename extension ".ah". After the code transformation the output of ac++ can be compiled to executable code with ordinary C++ compilers like GNU g++, clang++, or Microsoft VisualC++.

More details about the features of the AspectC++ language can be found in the *First Steps* manual, the *Quick Reference Sheet*, and the *Language Reference*. Everything is available on the AspectC++ homepage http://www.aspectc.org, which is also a source for updates of this manual.

The compiler's source code is freely available from the project's web site and covered by the GPL. For your convenience there are also binary versions of the open source implementation available.

This document focuses on how the ac++ compiler works and how it is used. The following sections are structured as follows: Section 2 describes how to get and install the compiler. It is followed by Section 3, which describes the two transformation modes of ac++ and the meaning of the command line arguments. Section 4 is dedicated to ac++-specific "pragmas" that extend the core AspectC++ language. Platform-specific notes are given in Section 5. It describes the specifics of the ac++ ports and which non-standard features of the back-end C++ compiler are supported. Section 6 lists some known problems, common pitfalls, and unimplemented language features.

2 Download and Installation

Binaries of ac++ for various platforms are available for free download from the AspectC++ homepage (see Section 5 for the list and state of the ac++ ports). The versioning scheme is shown in table 1 on the following page.

Scheme	Example	Kind of Release/Meaning
<version>.<release>	0.7	A regular release 0.7.
<version>.<release>.<fix-no>	0.7.3	Bug fix release number 3 of 0.7
<version>.<release>pre<no>	0.8pre1	Pre-Release number 1 for 0.8

Table 1: Versioning scheme

Besides the archive file with the compiler there is a README file and a CHANGELOG file available for each release. The README file explains the necessary steps for the installation, while the CHANGELOG documents the changes in the corresponding release as well as the history of changes.

The following subsections explain how the current version of the `ac++` compiler is unpacked, installed, and configured. This process depends on the development platform. Skip to the appropriate part from here.

2.1 Linux and MacOS X

The Linux and MacOS X installation procedures are very similar, because all of them belong to the UNIX system family. The `ac++` compiler and the example code is provided in a `gzip`-ed `tar` archive (tgz file). It can be unpacked with the following command in any directory:

```
tar xzvf <tar-file-name>
```

The command creates a directory aspectc++-<version>, which contains the `ac++` binary, the `ag++` front-end, the example code, and everything else that is needed to run the examples like a Makefile. To transform the examples (in the `examples` directory) simply execute `make` in the installation directory. Each example is then transformed from AspectC++ code into C++ code by weaving aspects and saved in `examples/<name>-out`. To run the example, enter the created directory, call `make` and start the executable.

The `Makefile`, which is used to compile the examples uses the command `ag++`, which is a wrapper for calling `ac++`, `g++`, and for the generation of the parser configuration file, which is needed for `ac++`. A separate manual for `ag++` is available from the AspectC++ web site.

2.2 Windows

The Windows port of `ac++` supports the freely available Cygwin/GNU `g++` and MinGW `g++` compiler as back-end compilers.

The installation of `ac++` in an environment with GNU `g++` and `make` is similar to the UNIX-like installation described in Section 2.1. Additionally refer to Section 5.2.3 on page 22, which provides some specific information about path names in the Cygwin environment.

The following procedure outlines the installation for non-g++ windows command line compilers: The `ac++` compiler and the examples are provided in a ZIP archive. Unpack `ac++` in a directory of your choice, for instance into `C:\AC`. The next step is to create a parser configuration file that describes predefined macros and standard include file paths of your back-end compiler. The files `pumabc55.cfg` and `pumavc7.cfg` can be taken as examples. An automatic generation of the config file as under UNIX systems is not available at the moment in the free `ac++` version.

The examples directory contains various examples that show how to write aspects in AspectC++. You can use the `examples.bat` batch file to weave all the examples at once. After this step the transformed example files can compiled.

3 Invocation

3.1 Modes

The `ac++` compiler supports two major transformation modes:

3.1.1 Whole ProgramTransformation (WPT)

WPT mode was the first transformation mode of `ac++`. However, it is not obsolete, because it may be useful in many cases. In this mode `ac++` transforms all files in a project directory tree (or set of directories) and saves the result in a different directory tree. For each translation unit and header file a new file is generated in the target tree with the same name. If further transformations of the source code have to be done, either with `ac++` or other tools, this is the mode to choose. Even comments and whitespace remain untouched.

The compiler performs a simple dependency check in WPT mode. A translation unit is recompiled if either the translation unit itself or any header file of the project

has been changed. This is not very precise but makes sure that after changing an aspect header file all translation units are recompiled.

3.1.2 Single Translation Unit (STU)

In the STU mode `ac++` must be called once for each translation unit like a normal C++ compiler. This makes it easier to integrate `ac++` into Makefiles or IDEs. As `ac++` can't save manipulated header files in this mode, because the unchanged header files are needed for the next translation units, all #include directives in the translation unit that refer to header files of the project directory tree are expanded and, thus, saved together with the manipulated translation unit. The resulting files can be fed directly into a C++ compiler. They do not depend on any other files of the project anymore.

In the STU mode the user is responsible for checking the dependencies of changed files and for calling the right `ac++` to transform all translation units that depend on a changed file. The general dependency rule is that a translation unit depends on every header file that is directly or indirectly included and every aspect header that might affect the translation unit (normally all!) and the files they depend on. If you are using `g++` and `make`, checking of this rule can be automized:

```
g++ -E -I<some-path> -MM <trans-unit> -include "*.ah"
```

This call of the `g++` preprocessor prints a makefile dependency rule, which is suitable to determine when `ac++` must be run to rebuild a tranlsation unit.

3.2 Weaving in Library Code

A C++ library consists of header files that have to be included by the client code and an archive file that contains the object code. If the library is implemented in AspectC++ and the client code should not be compiled with `ac++` it is necessary to generate manipulated header files. In the WPT mode this is done anyway. In the STU a directory tree with all manipulated headers can be generated with the `-i` option (see 3.4.7 on page 12).

3.3 The Project Repository

The `ac++` weaver internally creates a translation unit model, which contains a description of all name and code join points as well as the weaving plan, while it processes a C++ input file. By using the command line option `-r` (or

`--repository`) it is possible to save this model in a file called the "project repository". If the project repository already exists, ac++ will *merge* its translation unit model into the existing project repository. Eventually, after translation of all C++ input files of the project, the repository will contain a description of *all* potential and affected join points in the project. When an input file is modified and re-translated, `ac++` will update the repository accordingly.

The project model can be used for various purposes, e.g. as input for join point visualization tools. It can also be used for checking pointcut expressions or even their interactive development. This is supported by the `-x` (or `--expr`) command line option, which can be used to evaluate a given pointcut expression by matching it against the join points in the the repository. The internal structure of the project repository might be subject to future changes.

3.4 Options

Table 2 on the next page summarizes the platform-independent options supported by `ac++`. Platform specific options will be explained in Section 5. All options can either be passed as command line arguments or by the configuration file[1], which is referenced by the environment variable PUMA_CONFIG (see Section 2). '–' in any of the columns WPT or STU means that this option has no meaning in the corresponding translation mode.

The upper part of the table lists `ac++`-specific options, while the options in the lower part are widely-known from other compilers like `g++`.

3.4.1 `-p|--path <arg>`

This option defines the name of a project directory tree `<arg>`. The option can be used more than once if several directories belong to the project. At least one `-p` options is always needed when `ac++` has to transform code, even in STU mode.

3.4.2 `-d|--dest <arg>`

With `-d` a target directory for saving is selected. It corresponds to the last `-p` option. For example, if two directories belong to a project they would be described in STU mode with

```
-p dir1 -p dir2
```

[1] In the current ac++ version some of these options are not allowed in the config file, namely all between `-v` and `--no_problem...`.

Option	WPT	STU	Description
-p|--path <arg>	X	X	Defines a project directory
-e|--extension <arg>	X	–	Filename extension of translation units
-v|--verbose <arg>	X	X	Level of verbosity (0-9)
-c|--compile <arg>	–	X	Name of the input file
-o|--output <arg>	–	X	Name of the output file
-g|--generate	–	X	Generate link-once code
-i|--include_files	–	X	Generate manipulated header files
-a|--aspect_header <arg>	X	X	Name of aspect header file or 0
-r|--repository <arg>	X	X	Name of the project repository
-x|--expr <arg>	–	–	Match a pointcut expression (arg) agaist the project repository
--gen_deps <arg>	X	X	Generate aspect dependency file based on pragmas in aspect headers
--config <arg>	X	X	Parser configuration file
-k|--keywords	X	X	Allow AspectC++ keywords in normal project files
--introduction_depth <arg>	X	X	Set the maximum depth for nested introductions
--no_line	X	X	Disable generation of #line directives
--gen_size_type <arg>	X	X	use a specific string as size_t
--warn...	X	X	enable a weaver warning that is suppressed by default
--no_warn...	X	X	suppress a specific weaver warning
--problem...	X	X	enable back-end compiler problem workaround (see 5.2)
--no_problem...	X	X	disable back-end compiler problem workaround
--builtin_operators	X	X	Support advice on built-in operator calls
--data_joinpoints	X	X	Support data-based join points, e.g. get(), set(), ...
--attributes	X	X	Support C++11-attribute-based join points
--no_attributes	X	X	Disable support user-defined attributes
-I <arg>	X	X	Include file search path
-D <name>[=<value>]	X	X	Macro definitions
-U <name>	X	X	Undefine a macro
--include <arg>	X	X	Forced include

Table 2: ac++ Compiler Option Summary

and in WPT with two source/target pairs:

```
-p source1 -d target1 -p source2 -d target2
```

In STU mode `-d` makes only sense in combination with `-i` to generate header files for a library (see 3.4.7 on the following page).

3.4.3 `-e|--extension <arg>`

In WPT mode `ac++` searches in all project directories for translation units to transform. Translation units are identified by their filename extension. The default is "cc", which means that all files ending with ".cc" are handled. By using the option `-e cpp` or `-e cxx` you can select other frequently used filename extensions. The option can be used more than once, but only the last one is effective.

In WPT mode `ac++` generates a file called `ac_gen.<extension>`. This extension is also taken from the `-e` option, if one is provided.

3.4.4 `-v|--verbose [<arg>]`

The compiler can print message on the standard output device, which describe what it is currently doing. These message can be printed with different levels of details. You can select this level with the parameter `<arg>`. The range is from `0`, which means no output, to `9`, which means all details. The option `-v0` is the same as having no `-v` option at all. `-v` without `<arg>` is the same as `-v3`.

The `-v` option can be used more than once but only the last one is effective.

3.4.5 `-c|--compile <arg>`

The `-c` option is used to select an input file for `ac++` in the STU mode. Using it more than once is possible, but only one is effective. There are no restrictions on the filename extension. `ac++` expects that the file contains AspectC++ source code.

3.4.6 `-o|--output <arg>`

With the -o option one can select the name of the output file, i.e. the name of the target of the code transformation, in STU mode. If this option is not used, the default output filename is `ac.out`. Note that the output filename is *not* derived from the input file name as it is done by other compilers.

3.4.7 -i|--include_files

The -i option has to be used if the source code of the project should be compiled into a library and ac++ should run in STU mode (see 3.2 on page 8). When a translation unit is transformed by using -c and -o in STU mode no manipulated header files are generated. All include files are expanded within the generated source code. This is fully sufficient if the translation units will then be compiled and linked directly. However, if a library should be provided the client needs a library file (an archive) *and* manipulated header files. These can be generated with -i. The generation results in a directory tree with the same structure as the input directory tree specified by -p exhibits. Use the -d option to select the target directory name(s).

Note that at the moment only and all files with the extension .h are considered to be include files. This is rather inflexible and will be improved in future releases.

3.4.8 -a|--aspect_header <arg>

By default ac++ searches all files with the filename extension .ah in the project directory tree(s) and allows all aspects defined in these files to affect the current translation unit. If you are looking for a simple mechanism to deactivate aspects at compile-time, or if .ah does not conform to your local conventions, or if not all aspects should affect all translation units (be careful! See 6.1 on page 22), the -a option might help.

The option may be used more than once and each of them selects one aspect header that has to be considered for the current translation unit in STU mode or all translation units in WPT mode. If no aspect header should be considered use -a0.

3.4.9 -r|--repository <arg>

The “project repository” is an XML-based description of global information about an AspectC++ development project that is compiled with ac++. It fulfills two purposes:

1. It is a vehicle to transport information from one compiler run to another
2. It might be used by integrated development environments to visualize the join points where aspects affect the component code.[2]

[2] In fact, the AspectC++ Development Tools for Eclipse (ACDT) already use the repository

The `-r` option is used to define the name of the project repository file. However, this is an experimental feature. The file format might be subject to future changes. The uniqueness of join point IDs is only guaranteed if the project is compiled with a project repository. If a file with the given name does not exist, `ac++` will create a new repository file. If the file exists, but is empty or does not contain valid data, `ac++` terminates with an error message. A warning messages will be printed if the version of the weaver, which created the project repository, differs from the current `ac++` version.

3.4.10 -x|--expr <arg>

This option is used to match a pointcut expression, given as argument <arg>, against a project repository file. The project repository filename has to be provided with the `-r` option. For example the following command prints all nested class known in the "PragmaOnceObserver" test program in the AspectC++ development tree.

```
prompt> ac++ -x '"%::%"' -r PragmaOnceObserver/repo.acp
ObserverPattern.ah:12: Class "ObserverPattern::Subject"
ObserverPattern.ah:13: Class "ObserverPattern::Observer"
```

This example illustrates the mechanism with a more complicated pointcut expression:

```
prompt> ac++ -x 'call("%") && within("% main()")' -r ...
main.cc:29: Call "void ObserverPattern::addObserver( ...
main.cc:32: Call "void ObserverPattern::addObserver( ...
main.cc:34: Call "void ClockTimer::Tick()"
main.cc:37: Call "void ClockTimer::Tick()"
```

Note that pointcut expression contain quotes (‟). Make sure that quotes are not removed by the command shell. On Linux systems it is a convenient solution to enclose the pointcut expression in single quotes, e.g. `'"%"'`.

3.4.11 --gen_deps <arg>

With this command line option ac++ analyzes all "`#pragma acxx affect` <filename-pattern>" in aspect headers and generates an aspect dependency file named <arg> suitable for use with `ag++`. This can speed up incremental build times dra-

to visualize matched join points. See the ACDT homepage http://acdt.aspectc.org/ for information on the ACDT project.

matically. For more information refer to Section 4 on the ac++ pragmas as well as the *Ag++ Manual*.

3.4.12 `--config <arg>`

Besides setting the environment variable `PUMA_CONFIG` this options can be used to set the path to the parser configuration file.

3.4.13 `-k|--keywords`

By default the AspectC++ keywords `aspect`, `pointcut`, `advice`, `slice` and `attribute` are only treated as keywords in aspect header files. If they are used in normal project files, `ac++` interprets them as normal identifiers. By this design decision aspects can be woven into legacy code even if the code uses the AspectC++ keywords as normal identifiers.

If the AspectC++ keywords should be interpreted as keywords in normal project files as well, the command line option `-k` or `--keywords` has to be used.

In files that do *not* belong to the project, e.g. standard library header files, the AspectC++ keywords are always regarded as normal identifiers, even if `-k` or `--keywords` is used.

If any of the AspectC++ keywords is generated by a macro, the classification as keyword or identifier is based on the file in which the macro expansion takes place. It does not depend on the location of the macro definition.

3.4.14 `--introduction_depth <arg>`

AspectC++ introductions may affect introduced code. This is called a “nested introduction”. In order to avoid problems with infinitely nested introductions, `ac++` checks the “depth” of a nested introduction and does not allow a depth that exceeds the given maximum `<arg>`. The default value for `<arg>` is 10.

3.4.15 `--no_line`

When `ac++` manipulates files, e.g. by inserting generated code, it also inserts `#line` directives. Inserting these directives can be disabled with the `--no_line` option. Normally, `#line` directives are only generated by C preprocessors. The directives are important for back-end compiler error messages and source code debuggers. Without the `#line` generation these numbers correspond to the lines

Warning Name	Condition
`deprecated`	a deprecated syntax is being used
`macro`	macro-generated code would have to be transformed

Table 3: `ac++` Warnings

in the generated code, while they correspond to the source code written by the programmer otherwise.

3.4.16 `--gen_size_type <arg>`

`ac++` generates a `new` operator, which has `size_t` in its argument type list. As the generated code shall not include the respective header file (to avoid portability problems), the weaver normally generates the name of the right type. However, in case of cross-compilation the type on the target platform might differ. Then it is possible to provide a string with this option, which is directly used in the constructor's argument list.

3.4.17 `--problem...`

An option like this is used to enable a back-end compiler-specific code generation workaround. This is sometimes needed, because the C++ compilers differ in their degree of standard conformance. For details about the workarounds needed for each back-end refer to Section 5.2.

3.4.18 `--no_problem...`

This option can be used to disable a back-end compiler-specific code generation workaround which is enabled by default.

3.4.19 `--warn_...`

With this option the weaver is instructed to print specific warnings that are otherwise suppressed. Table 3 lists the names of warnings currently supported by the weaver.

3.4.20 `--no_warn_...`

The warnings listed in table 3 can be suppressed with `--no_warn_`<*Name*>.

3.4.21 --builtin_operators

This option is needed if you want to use the pointcut function builtin(). For more information on this feature consult the language reference manual.

3.4.22 --data_joinpoints

This option is needed if you want to use the pointcut functions get(), set(), and ref(). For more information on this feature consult the language reference manual.

3.4.23 --attributes/--no_attributes

This option is needed if you want to use attributes in C++11-Style (e.g. [[noreturn]]). It is enabled by default and can be disabled with `--no_attributes`. For more information on this feature consult the language reference.

3.4.24 -I <arg>

The option `-I` adds the directory `<arg>` to the list of directories to be searched for header files. It can be used more than once. The compiler `ac++` needs to know all directories, where header files for the current translation unit might be located.

In case of system headers there are often a lot of these directories. To make the setup of `ac++` more convenient we provide the `ag++ --gen_config` command. The command calls the `g++` compiler to get all these paths. Users of non-supported back-end compilers have to find out this list on their own.

3.4.25 -D <name>[=<value>]

With `-D` a preprocessor macro `<name>` will be defined. Without the optional value assignment the macro will get the value `1`. The option can be used more than once.

In most cases your source code expects some standard macros to be defined like `win32`, `linux`, or `i386`. And even if your code doesn't use them directly, they are often required to be set correctly by system header files. Thus, for the `ac++` parser a correct set of these macros has to be defined. For `g++` users we provide a command called `ag++` that calls the compiler to get the list of these macros. Users of non-supported back-end compilers have to find out this list on their own.

3.4.26 -U <name>

This option can be used to undefine a previously defined macro.

3.4.27 -include <arg>

The `-include` option can be used to include a file `<arg>` into the compiled translation unit(s) even though there is no explicit `#include` directive given in the source code. If multiple `-include` options are given on the command line, the files are included in the same order (from left to right). If you use the option in STU mode make sure that the back-end compiler is not forced to include the same files again (read details in 6.1.2 on page 25).

3.5 Examples

- `ac++`
  Displays all options with a short description.
- `ac++ -I examples/Trace -p examples/Trace -d examples/Trace-out`
  Transforms the complete project from directory “examples/Trace” into the directory “examples/Trace-out”. This is the whole program transformation (WPT) mode, which also performs a simple dependency check.

The following examples describe the compiler like interface (STU Mode). All dependency handling has to be done by the user.

- `ac++ -c main.cc -p.`
  Transforms only the translation unit `main.cc`. The default name for the output file is `ac.out`.
- `ac++ -c main.cc -o main.acc -p.`
  Transforms the file `main.cc` into the new file `main.acc`.
- `ac++ -c main.cc -o main.acc -p. -a trace.ah`
  Transforms the file `main.cc` into the new file `main.acc` with the aspect located in `trace.ah`.
- `ac++ -i -v9 -p. -d includes`
  Creates the manipulated project header files and stores them into the directory `includes`. ATTENTION: This works only once, because the `includes` directory is located inside the project directory tree and the aspect header files exists twice then.

4 Pragmas

Pragmas are a means for programmers to give the compiler additional information that cannot be expressed in the core AspectC++ language.

4.1 Specifying Aspect Dependencies

Aspects that are defined in aspect header files can affect various join points throughout the whole development project. This is a powerful feature, but has two negative consequences:

1. Changes in aspect headers could affect any translation unit and, thus, by default, trigger a time-consuming full rebuild.
2. It is not unlikely to accidentally weave aspect code at join points in files that shouldn't be affected.

To mitigate both problems `ac++` provides a pragma that can be used to declare the set of files that are *intended* to be affected by the concrete aspects defined within the aspect header in which the pragma is located. Here is an example:

```
#ifndef TRACE_FOO_CALLS_AH
#define TRACE_FOO_CALLS_AH

#pragma acxx affect "classA.*"
#pragma acxx affect "classB.*"
aspect TraceFooCalls {
  advice call("% foo()") : before() {...}
};

#endif // TRACE_FOO_CALLS_AH
```

The two pragmas state that only files that match the filename pattern `"classA.*"` *or* `"classB.*"` should be affected by the aspect `TraceFooCalls`. If `foo()` is called in any other file an error message is printed and the compilation stops. As `ac++` enforces this, you can also make use of the pragma for handling aspect dependencies more efficiently. Using the command line option `-gen_deps` (see Section 3.4.11) an aspect dependency file can be generated and passed to ag++. The exact format of filename patterns is described in the *Ag++ Manual*.

4.2 Filtering Join Points Based on Filenames

The C++ language elements for expression the program structure, such as classes and namespaces, are independent of the file structure of the project. Therefore, the AspectC++ core language also doesn't support the file concept when expression crosscutting concerns. However, in some situations, for example when dealing with C-style projects, the file structure might be the only practical way for distinguishing join points. In such case the following pragma might be useful:

```
#pragma acxx filter "main.cc"
advice call("% printf(...)") : before() {...}
```

The `#pragma acxx filter` in the first line makes sure that the advice in the second line will only affect join points that are lexically located in `"main.cc"` even if many calls to `printf()` are scattered over the whole code base. Instead of `"main.cc"` a filename pattern can be specified. The pattern syntax is described in the *Ag++ manual*. If multiple `#pragma acxx filter` are used in front of advice, they all specify valid join point locations. Each filter pragma will only affect the next advice in the source code and is "forgotten" afterwards.

Filtering join points can be used to avoid error messages triggered by #pragma acxx affect (see Section 4.1).

5 Platform Notes

5.1 Ports

The `ac++` compiler was originally developed on RedHat Linux systems.Today most of the development is still done under Linux (Debian and OpenSuse), but Windows has become a second development platform. This means that the Windows and Linux ports are the most tested. The MacOS X ports were compiled, because they were demanded by users, but they are far less tested than our development platform ports.

5.1.1 Linux

The `ac++` binary was tested on...

- Debian 3.0, . . . , 12.12 and various Ubuntu versions up to 25.10. Note that Debian and Ubuntu packages of AspectC++ are integrated into the distributions. They can be easily installed with `apt-get install aspectc++`.
- OpenSuse 8.2, ..., 15.2

5.1.2 Windows

Windows systems have different filename conventions than UNIX systems. Although `ac++` was originally developed on Linux and does not use or need the Cygwin environment, path names are allowed to contain '`\`' characters and drive names like '`C:`'. The UNIX filename delimiters '`/`' are also accepted.

Warning: `ac++` is not well tested on Windows. Use it only with code that is compatible with `g++` or `clang++`. We recommend to use the "Windows Subsystem for Linux" with the Linux binary or a Linux-compatible shell as provided by MSYS2 with the Windows binary.

5.1.3 MacOS

A MacOS port of the AspectC++ toolchain is available for MacOS/x86_64 as well as MacOS/arm. Note that the binaries provided at `www.aspect.org` are not verified by Apple. Therefore, MacOS asks you to trust the source of `ag++` and `ac++` before you can run them. We are very sorry for this inconvenience, but we are not willing to pay an annual fee to Apple.

We recommend to check out the AspectC++ source code and build the executable programs `ac++` and `ag++` locally. Thereby, they get a locally valid signature.

5.2 Back-End Compiler Support

The C++ compiler that should be used to compile the output of `ac++` (back-end compiler) plays a very important role for `ac++`, because compilers normally come with header files, which `ac++` must be able to parse. None of the back-end compilers listed here has totally standard-conforming header files, which makes it very hard for `ac++` to parse all this code.

The current implementation of `ac++` is based on the C++ parser of the LLVM/-Clang project. Clang aims to be compatible with `g++` and Visual C++, but there are always subtle differences. Development and testing is normally done on Linux with `g++`. The MacOS version uses the `g++` command as well, but on MacOS

`g++` is alias for `clang++`. As a consequence parsing code of a `g++` or `clang++` project normally works well. Visual C++ compatibility hasn't been tested for years. So don't expect too much!

In the following we document compiler-specific problems.

5.2.1 GNU g++

There are a lot of GNU g++ specific C++ extensions as well as several builtin functions and types. To enable all these extensions the option `--gnu` has to be used. If a configuration file is generated with `ag++ --gen_config`, this option will be automatically inserted.

Compilers from the g++ family do not support explicit template specialization in a non-namespace scope. However, this feature is needed by ac++ in the code generation process. A workaround for this problem is automatically enabled when you use the `--gnu` *<version>* option. To explicitly enable or disable the workaround use `--problem_spec_scope` or `--no_problem_spec_scope`.

5.2.2 Tricore/GNU g++

Unfortunately, the freely available variant of LLVM/Clang, i.e. the parser of `ac++`, does not support the Infineon AURIX Tricore platform. As `ac++` doesn't need machine code generation for this platform, we were able to find a workaround so that Tricore projects can be compiled with `ac++`/`ag++` with only minimal limitations. If the parser configuration file contains the option `--target tricore` or `--target tricore-elf`, the internal Clang parser is configured with the name of a very similar target architecture. Incompatibilities in the size and alignment values of built-in data types are fixed so that the front end and the back end have the same view on data structure layouts.

Warning: A known problem is that Clang checks whether CPU register names in inline assembler code are valid for the respective platform. It is not possible to fix this register list without having to patch the Clang parser. Therefore, we decided to skip all token sequences of the form "`asm volatile`$_{opt}$`(...)`" completely during the code analysis. This avoids any error messages related to unknown register names. However, if the inline assembler code statement contains C++ code with a function call or other kind of AspectC++ join point, it will not be seen by `ac++` as well. In most use cases for `asm` statements it only contains assembler code and register names.

5.2.3 Cygwin/GNU g++

The `ac++` compiler can also be used with the Cygwin/GNU g++ compiler under Windows. Note, that `ac++` itself is not a Cygwin application and, thus, does not support Cygwin-specific path names like `/home/olaf`, which is relative to the cygwin installation directory. If you generate your parser configuration file automatically with `ag++ --gen_config` the contained include paths will automatically be converted from Cygwin paths names to Windows path names using the `cygpath` command. However, be careful when you set the `PUMA_CONFIG` environment variable or when you pass any other path name to `ac++`. Furthermore, `ac++` and `ag++` don't support Cygwin file links. This might also cause compilation problems. A known problem is that in some Cygwin versions `g++` itself is a link to `g++-`*<version>*. This means that it will not be found by `ag++`. It helps to provide the proper compiler name with the `--c_compiler g++-`*<version>* option.

5.2.4 MS VC++

Warning: Using ac++ in combination with MS VC++ is problematic. It hasn't been tested for quite some time.

6 Problems & Workarounds

6.1 Common Pitfalls

6.1.1 Include Cycles

In versions prior to 1.0pre1 include cycles could occur in many situations and workarounds could not always be found. In version 1.0pre1 include cycles can only occur in the case of aspect code with introductions. Advice for code join points cannot produce cycles.

The reason for the remaining possible cycles is that `ac++` generates `#include <aspect-header>` in every file that contains the definition of a target class of an introduction. Without this generation pattern definitions from the aspect header would not be accessible by introduced code. However, if the aspect header directly or indirectly includes the target file, there is a cycle, which might cause parse errors.

Figure 1 illustrates the include cycle problem by giving an example. Here an aspect `Problem` uses the type `Target` and therefore includes `Target.h`. At the

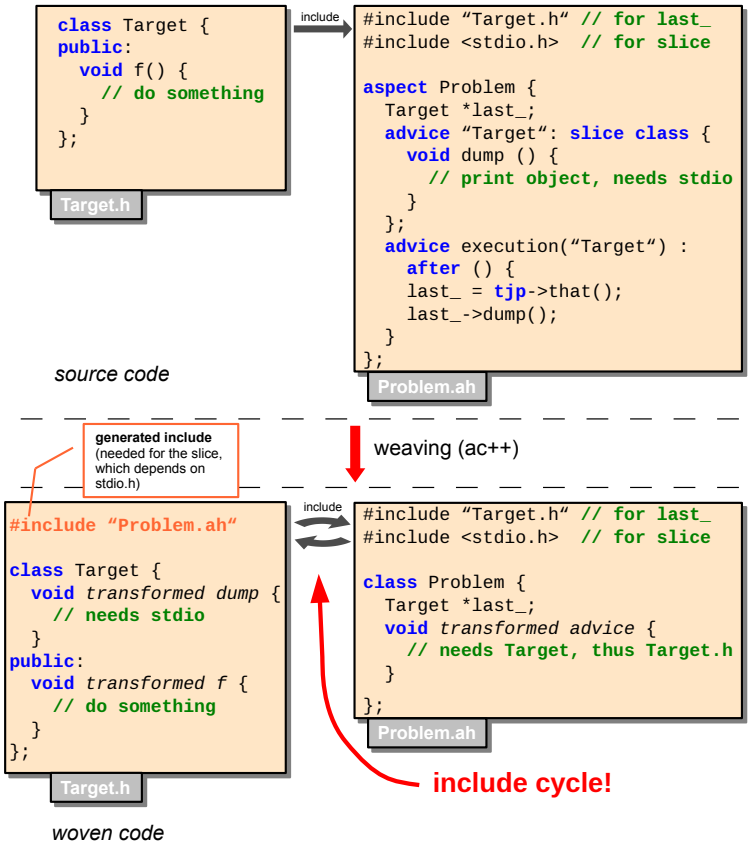

Figure 1: Include cycle problem

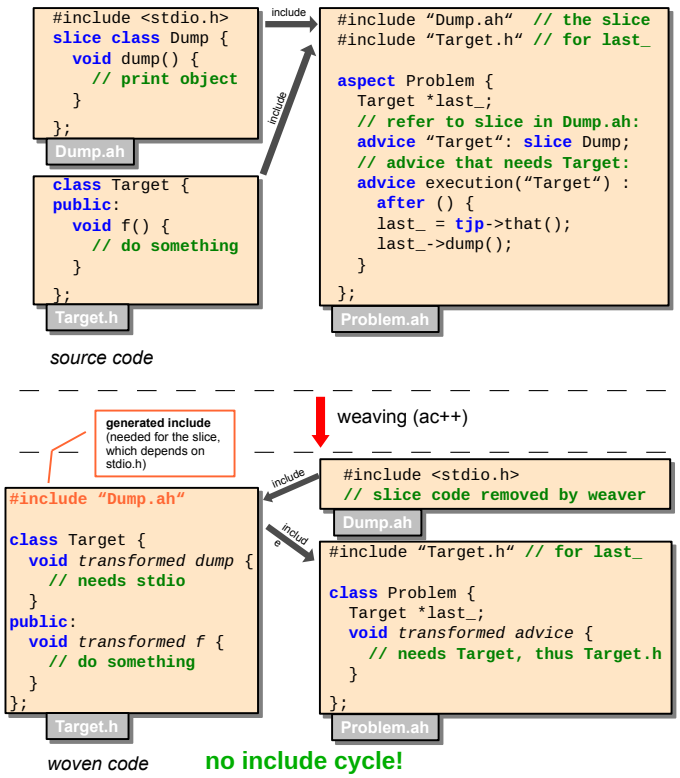

Figure 2: Include cycle avoidance

same time the aspect introduces a slice into the class `Target`. As the slice might depend on definitions or `#include`s in `Problem.ah`, the weaver generates the `#include "Problem.ah"` in `Target.h`. This causes the include cycle. Include guards (which should always be used!) avoid duplicate definitions, but do not solve the problem. It might still be the case that the parser complains about undefined types.

To avoid these cycles introductions can always be separated from the aspect by means of slices. Slice declarations and slice member definitions can be located in arbitrary aspect header files. The aspect weaver will only include these aspect headers in the target classes' header/implementation files and thereby avoid the cycle. For a slice reference within an advice declaration even a forward declaration of the slice is sufficient.

Figure 2 shows how the include cycle from the example in Figure 1 can be avoided. Here the function `dump` is implemented in a separate slice class `Dump` that is stored in an aspect header file `Dump.ah`. The implementation of `Dump` could rely on defintions and `#include`s in `Dump.ah` (`stdio.h` in this example), but not on defintions in `Problem.ah`. Therefore, the aspect weaver generates

`#include "Dump.ah"` and not `#include "Problem.ah"` in `Target.h`.

Note that in the case of **non-inline** introductions the #include directive is generated in the file that contains the “link-once element” of the target class, which is never a header file. You can, for example, exploit this feature to produce cyclic class relationships. The included file will be the aspect header file that contains the definition of the non-inline slice member.

6.1.2 Duplicate Forced Includes in STU Mode

In the Single Translation Unit (STU) mode `ac++` handles forced includes (see `-include` option in Section 3.4.27 on page 17) in the following way:

internal includes: If the included file is part of your project, the file content will be expanded in the compiled translation unit.

external includes: If the included file is *not* part of the project, ac++ generates an #include directive with the absolute path name of the file.

In both cases the back-end compiler should not be forced to include the same file again. For example, `g++` users should not use the `-include` option with `ac++` and with `g++`, because otherwise symbols might be defined twice.

6.1.3 Compiling libraries

There are certain restrictions on the code structure if `ac++` should generate transformed header files for an aspect-oriented C++ *library*. For instance, all header files need "include guards" and should not depend on the context by which they are included. Furthermore all headers must have the extension `.h`.

Furthermore, users have to be careful *not* to generate transformed headers into the project directory tree (`-p` option). Otherwise, the *next* compilation of the library is likely to fail, because `ac++` would search aspect header files in the generated directory tree.

6.1.4 Project structure

The AspectC++ weaver `ac++` expects that "projects" do not overlap, have no cyclic dependencies, and can be described by a list of directory names (`-p` option). AspectC++ needs the notion of a "project" in order to restrict the set files that are affected by the aspects. Sometimes big applications are organized in multiple

projects within the same workspace and have arbitrary dependencies (include relations) to each other. For these applications selecting the `-p` option is sometimes difficult. Often treating the whole workspace as one AspectC++ project is the best solution.

6.1.5 The `--aspect_header` option

This option can be used to avoid that `ac++` automatically searches all aspect header files in the project directory tree. It is to be made sure by the user that each aspect header file is included *only once*. If no aspect headers should be taken into account, the option "`-a 0`" has to be used.

6.2 Unimplemented Features

6.2.1 Multi-Threading Support

C++ has no integrated thread model like Java. Therefore, the woven AspectC++ code cannot rely on any available thread synchronization mechanism. As a result the implementation of the `cflow` pointcut functions is currently *not thread-safe*.

We are urgently investigating how thread synchronization and thread local storage can be integrated into AspectC++.

6.2.2 C++ 11/14/...

`ac++` does not fully support the latest language features introduced with the C++ 11 standard. These features, such as lambda functions or move constructors, are accepted by the underlying parser (the Clang framework, which is used in `ac++` 2.x), but the weaver can't deal with these features, yet. Therefore, it is safe to use C++ 11 code as long as the code affected by advice conforms to C++ 2003.

6.2.3 Parse Errors

If `ac++` stops processing because of parse errors this might be due to an incompatibility or missing feature in the underlying C++ parser.

In the case that the error is found in your own code, i.e. code you are able to modify, you could use the following workaround:

```
#ifdef __acweaving
// ... simplified version of the code for ac++
#else
// ... original code
#endif
```

Even if your own AspectC++ code contains only harmless C++ code you might experience parsing problems due to header files from libraries which your application code includes, especially in the case of template libraries. In this situation it might help to copy the file with the parse error into a different directory. Then you have to change the code in this file to avoid the error message by simplifying it. The final step is to extend the `puma.config` file by a "`-I <path>`" entry for the directory where you placed the copy. As the result `ac++` will now parse the simplified version while the original file is untouched and used while the C++ compiler runs.

6.2.4 Templates

Currently `ac++` is able to parse a lot of the (really highly complicated) C++ templates, but weaving is restricted to non-templated code only. That means you can not weave in templates or even affect calls to template functions or members of template classes. However, template instances can be matched by match expressions in the pointcut language and calls to members of class templates or template functions can be affected by advice.

6.2.5 Macros

In versions prior to 1.0 the weaver was not able at all to transform code that was generated by macro expansion. It simply printed a warning and continued without transforming the code. To turn this warning off the command line option `--no_warn_macro` could be used (see Table 3 on page 15).

The current solution is to expand a macro whenever it is affected by aspects and do the weaving afterwards. While this works fine for most cases, problems may occur if the macro definition used by the (cross-)compiler differs from the one used by ac++. Future releases will thus distinguish between macros definitions that belong to the project and can safely be expanded and macros that were defined outside the project.

6.2.6 Unimplemented Language Elements

`cflow` does not yet support exposure of context information.

`base` only works as expected if all classes that should be matched by the pointcut function's argument are known in the translation unit. Therefore, the aspect header file has to contain the right set of include directives.

6.2.7 Support for Plain C Code

Currently `ac++` generates C++ code, which cannot be compiled by a C compiler. As for many hardware platforms in the embedded domain no C++ compiler is available we are actively looking for a solution.

6.2.8 Support for C++ language extensions

The parser fully supports the language features of C++ 11 and more recent standards (see 6.2.2), but the AspectC++ language and the code transformation patterns have not been extended, yet. Make sure that match expressions don't match functions with signatures that contain elements that did not exist in C++ 03. Since version 2.3 `constexpr` functions are not matched automatically.

6.2.9 Constructor/Destructor Generation

If advice for construction/destruction joinpoints is given and no constructor/destructor is defined explicitly, `ac++` will generate it. However, currently `ac++` assumes that the copy constructor has one argument of type "`const <Classname>&`". This leads to problems if the implicitly declared copy constructor has an argument of type "`<Classname>&`". Therefore, you should not define construction/destruction advice for classes with this copy constructor signature.

Further problems appear when an a class instance is aggregate initialized. The automatic constructor generation can break this ability.

6.2.10 Functions with variable argument lists

There is no support for execution advice on functions with variable argument lists. A warning will be issued. There is no portable way to generate a wrapper function for this class of functions.

6.2.11 Restrictions on calling `tjp->proceed()`

Due to a problem with result object construction/destruction of intercepted functions, the `tjp->proceed()` function may only be called *once* during around advice.

6.2.12 Advice on advice

Join points within advice code are not matched by pointcut expressions.

6.2.13 JoinPoint-API for slices

There is a joinpoint-API for slices introduced into a target class. It provides static type information about the target's baseclasses and members as well as dynamic information, such as a pointer to each member. However, the baseclass part of the slice may not access the JoinPoint-API. This is partly natural, as, for instance, member types might depend on the introduced baseclass. Yet, not even the target's classname is available. Future versions might make this possible.

The identifier 'JoinPoint' is only to be used to access the joinpoint-API. Even though it would conceptually make sense to allow, for instance, a local variable to be called 'JoinPoint', it is not supported, yet.

7 Code Transformation

This section documents some internals of the `ac++` weaver implementation and describes .acc file content.

7.1 Typical Patterns in Woven Source Code (.acc) Files

7.1.1 Namespace AC

Woven .acc code begins with the declaration of the AC namespace. The first AC declaration consists of typedefs, templates and helper functions that may be useful for your aspects. They might also not be used at all. For that purpose you might ignore it for now and use it to look up if some contents appear in your aspects later on. Below the AC declaration, your aspects will be declared and your woven code begins.

```
namespace AC {
  ...
  #Typedefs, Templates & Functions usable by Aspects
  ...
}
```

7.1.2 Include Expansion

Includes that are part of the project directory tree will be expanded into .acc code. Below you can see how an example header file “hello.h” is included into a .acc file. The content is copied and the include line is commented out. Notice that `#include <iostream>` remains as a regular include. That is because `<iostream>` is not part of our project directory tree. The project directory tree is by default the current working directory or can be specified with parameters `-p /--path` (See 3.4.1). Double includes will be ignored by `ac++`.

```
// commented out by ac++: #include "hello.h"
#line 1 "example/hello.h"
#ifndef __HELLO_H__
#define __HELLO_H__

#include <iostream>
```

```
void hello(){
  std::cout << "Hello" << std::endl;
}
#endif
#line 94 "example/main.cc"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

7.1.3 #line Directives

For debugging and compiling purposes, the woven source code contains `#line` directives. For example the line `#line 10 main.cc` will signal the compiler that the content below it is from (and starts at) line 10 of the original main.cc file. That way compiler error messages are able to link to the original project files and a debugger can refer to the original source files instead of the woven files.

```
#line 1 "example/hello.h"
#ifndef __HELLO_H__
#define __HELLO_H__

#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
#endif
```

7.2 Aspects in Woven Code

7.2.1 Aspects Become Classes

In woven .acc code, aspects are converted to classes. Advice will be translated as public methods that are named in the following schema: `__a0_after()`. *a0* meaning the advice index (a0 is the first advice in its class / aspect) and *after* describing what kind of advice is used. See below for a simple example of aspect translation in woven code.

```
aspect World {
  advice execution("void hello()") : after() {
    //print "World" after execution of the 'hello()' function
    std::cout << "World" << std::endl;
  }
};
```

```
class World {
  public: void __a0_after() {
    std::cout << "World" << std::endl;
  }
  ...
}
```

7.2.2 Singleton Pattern and Invoke Function

By default Aspects are instantiated using the singleton design pattern. They provide an `aspectof()` function to obtain a pointer to the singleton object. For each advice, an invoke function is generated that accesses the singleton and calls the advice (see `invoke_World_World__a0_after`). Generally, a lot of transformed code functions will be inline functions, preventing call and return instructions to affect code runtime. Note that the example below also shows an `aspectOf()` getter. `aspectOf()` was the default getter in AspectJ and is now deprecated in AspectC++.

```
public:
  static World *aspectof () {
    static World __instance;
    return &__instance;
  }
  static World *aspectOf () {
    return aspectof ();
  }
```

```
namespace AC {
  __attribute((always_inline)) inline
```

```
    void invoke_World_World__a0_after ()
  {
    ::World::aspectof()->__a0_after ();
  }
}
```

7.3 Basic Code Transformation Patterns

In this section, you will find simplified examples of the code transformation patterns of predefined pointcut functions. There may be some exceptions to these examples but they should be helpful in understanding how aspects are woven and how aspects and C++ code interact with each other. Code that is not important for understanding the weaving process is cut out of the examples. Code is also reformatted for better readability. These examples are organized in the following schema:

```
Listing with Aspect to be applied
```

```
Listing with Example Code
```

```
Listing with Transformed / Woven Code
```

7.3.1 Execution

```
aspect World {
  advice execution("void hello()") : after() {
    //print "World" after execution of 'void hello()'
    std::cout << "World" << std::endl;
  }
};
```

```
void hello(){
  std::cout << "Hello" << std::endl;
}

int main(){
```

```
  hello();
  return 0;
}
```

```
class World{
  public: void __a0_after() {
    std::cout << "World" << std::endl;
  }
};

inline void __exec_old_hello(){
  std::cout << "Hello" << std::endl;
}

void hello(){
  ::__exec_old_hello(); // print "Hello"
  AC::invoke_World_World__a0_after (); // print "World"
}

int main(){
  hello();
  return 0;
}
```

When an execution advice is applied, the executed function is replaced by a wrapper function that calls the advice at the specified point and also calls the original function, which is put into a new separate inline function (__exec_old_*function_name*).

7.3.2 Call

```
aspect Hello {
  advice call("void world()") : before() {
    // print "Hello" before void world() is called
    std::cout << "Hello" << std::endl;
  }
};
```

```
void world(){
  std::cout << "World" << std::endl;
}

int main(){
  world();
  return 0;
}
```

```
class Hello{
  public: void __a0_before() {
    std::cout << "Hello" << std::endl;
  }
};

void world(){
  std::cout << "World" << std::endl;
}

inline void __Call__Z4mainv_0_0 (){
  AC::invoke_Hello_Hello__a0_before (); // print "Hello"
  ::world(); // print "World"
}

int main(){
  __Call__Z4mainv_0_0 ( );
  return 0;
}
```

In comparison to execution advice, call advice do not replace the executed function with a wrapper function, but rather wrap it with a new inline function when it is called. The wrapper function also calls the advice at specified position.

7.3.3 Construction

```
aspect Hello {
  advice construction("World") : after() {
    std::cout << "- Hello World" << std::endl;
  } // printed when World instance is constructed
};
```

```
class World{ // Class with simple constructor
  public: World(){std::cout << "New World created";}
};

int main(){
  World A = World(); //Construct "World" instance
  return 0;
}
```

```
class Hello{
  public: void __a0_after() {
  std::cout << "- Hello World" << std::endl;
  }
};

class World{
  public: World(){
    this->__exec_old_C1(); // Call old constructor
    AC::invoke_Hello_Hello__a0_after (); // Call advice
  }
  inline void __exec_old_C1(){std::cout << "New World created";}
};

int main(){
  World A = World(); // Construct temporary World instance
  return 0;
}
```

Each constructor is transformed into a wrapper function that calls the advice. Old constructor content is copied into new inline functions (*__exec_old_C1* for first constructor, *_C2* for second constructor, etc.) and also called from the wrapper function. Note that if a construction advice exist, wrapper functions for the default constructor and the copy constructor will be generated by `ac++` even though they were not explicitly declared.

7.3.4 Destruction

```
aspect Goodbye {
  advice destruction("World") : after() {
    //print "Goodbye World" after destructing a World instance
    std::cout << "- Goodbye World" << std::endl;
  }
};
```

```
class World{
  public: ~World(){std::cout << "Cleanup";} // Example destructor
};

int main(){
  World A = World(); // World instance
  return 0;
}
```

```
class Goodbye{
  public: void __a0_after() {
    std::cout << "- Goodbye World" << std::endl;
  }
};

class World{
  public: inline ~World () {
    this->__exec_old_D1(); // Call old destructor
    AC::invoke_Goodbye_Goodbye__a0_after (); // Call advice
  }
  // Old destructor
  inline void __exec_old_D1(){std::cout << "Cleanup";}
};

int main(){
  World A = World(); // World instance
  return 0;
} // World gets destructed at end of main
```

Destruction advice are transformed similar to construction advice. All Destructors are transformed to wrapper functions that call the advice and old destructor content is put into new inline functions (for example *__exec_old_D1* for the first

destructor, *_D2* for second destructor, etc.).

7.3.5 Slice

```
aspect Example {
  advice "Laptop": slice class SL { // Add to Laptop
    private: int ram;
    public: int get_ram(){return ram;}
  };
  advice "Laptop": slice class : public ShopItem{}; // Add base class
};
```

```
class ShopItem{int price;};
class Laptop{int threads;};

int main(){
  Laptop abc = Laptop();
  return 0;
}
```

```
// Aspect class empty besides default content (aspectof getter)
class Example {};
class ShopItem{int price;};

// New base class, new variable, new getter
class Laptop:  public ShopItem{
  private: int threads;
  private: typedef Laptop SL; // Typedef for slice name SL
  private : int ram ;
  public : int get_ram ( ) { return ram ; }
};

int main(){
  Laptop abc = Laptop();
  return 0;
}
```

Transformation of slice introductions is pretty straightforward. The class `Example` for the aspect is left almost empty, a `typedef` in the sliced class is created for our aspect slice name. Changes in the sliced class are added straight into the

class (`ram` and `get_ram`) and `ShopItem` is now a base class of `Laptop`.

7.3.6 Get

```
aspect Hello {
  advice get("char %") : before() {
    //print "Hello World" before a char is get
    std::cout << "Hello World";
  }
};
```

```
char exclamation = '!';
int main(){
  // get and print exclamation mark
  std::cout << exclamation << std::endl;
  return 0;
}
```

```
class Hello {
  public: inline void __a0_before() {
    std::cout << "Hello World";
  }
};

char exclamation = '!';

template <typename TResult, typename TEntity>
  inline TResult __Get__Z4mainv_0_0 (TEntity &ent, AC::RT<TResult>)
{
  TResult __result_buffer;
  AC::invoke_Hello_Hello__a0_before (); //print Hello World
  __result_buffer = ::exclamation;
  return (TResult &)__result_buffer;
}

int main(){
  std::cout <<
  __Get__Z4mainv_0_0<  > (exclamation, __AC_TYPEOF((exclamation)) )
  << std::endl;
  return 0;
```

```
}
```

Where our specified variable would be get, a wrapper function is called instead. It gets the specified variable, buffers it, calls the advice at the specified location and returns the buffered value as inline function.

7.3.7 Ref

```
aspect Hello {
  advice ref("char %") : before() {
    std::cout << "Hello World";
  } //print "Hello World" before a char is referenced
};
```

```
char exclamation = '!';

int main(){
  char * exclamation_mark = &exclamation;
  std::cout << *exclamation_mark << std::endl;
  return 0;
}
```

```
class Hello {
  public: inline void __a0_before() {
    std::cout << "Hello World";
  }
};

template <typename TResult, typename TEntity>
  inline TResult __Ref__Z4mainv_1_0 (TEntity &ent, AC::RT<TResult>)
{
  TResult __result_buffer;
  AC::invoke_Hello_Hello__a0_before (); // call advice
  __result_buffer = &( ::exclamation ); // buffers reference
  return (TResult &)__result_buffer; // returns buffered reference
}

char exclamation = '!';
```

```
int main(){
  char * exclamation_mark = __Ref__Z4mainv_1_0<  >
    (exclamation, __AC_TYPEOF((&exclamation)) );
  std::cout << *exclamation_mark << std::endl;
  return 0;
}
```

When the specified reference is created, a wrapper function is created instead. In it the advice is called, the reference is created, it is buffered and that buffer is returned.

7.3.8 Set

```
aspect Example {
  advice set("int %") : after() {
    std::cout << "Finished" << std::endl;
  } //print "Finished" after setting an int
};
```

```
int i;
int main(){
  i = 42; //setting int
  return 0;
}
```

```
class Example {
  public: inline void __a0_after() {
    std::cout << "Finished" << std::endl;
  }
};

template <typename TArg0, typename TEntity>
  inline void __Set__Z4mainv_1_0 (TEntity &ent, TArg0 arg0)
{
  ::i = (arg0); // Setting value
  AC::invoke_Example_Example__a0_after (); // Calling advice
}

template <typename TArg0, typename TArg1, typename TResult>
```

```
  inline TResult &__Builtin__Z4mainv_0_0
  (TArg0 arg0, TArg1 arg1, AC::RT<TResult>)
{
  TResult *__result_buffer;
  __Set__Z4mainv_1_0< int > (arg0, arg1 ); // Calling set function
  __result_buffer = &arg0;
  return *(TResult *&)__result_buffer;
}

int i;
int main(){
  __Builtin__Z4mainv_0_0< int &, int >
    (i , 42, __AC_TYPEOF((i = 42)) );
  return 0;
}
```

The set advice transformation is a bit more complicated. The set wrapper function is called from a builtin wrapper function. Wherever the specified variable is set, a builtin operator function is called, that calls the set wrapper function. In there the advice is called and the variable is set.

7.3.9 Exposing Context Information with the Join Point API

Via the join point API, context information such as parameters of a captured function can be passed to the advice. When the join point API is used, the advice is implemented as a template function. The type of context information is defined when calling the advice. Inside the `execution` wrapper function, join point type and join point data are generated and passed to the advice via an adapted invoke function. Some elements are left out of this example to keep it readable.

```
aspect Example {
  advice execution("void give_int(%)") : before() {
    std::cout << *tjp->arg<0>() << std::endl;
  } //print first parameter of give_int()
};
```

```
void give_int(int i){
  std::cout <<  "Got the Int!" << std::endl;
}
```

```
int main(){
  give_int(42);
  return 0;
}
```

```
class Example {
  public:
    template <typename JoinPoint> void __a0_before(JoinPoint *tjp) {
      std::cout << *tjp->template arg<0>() << std::endl;
  }
};

namespace AC { // different invoke function passing JoinPoint Type
  template <class JoinPoint> __attribute((always_inline))
    inline void invoke_Example_Example__a0_before (JoinPoint *tjp)
  { ::Example::aspectof()->__a0_before (tjp); }
}

inline void __exec_old_give_int(int i){
  std::cout <<  "Got the Int!" << std::endl;
}

void give_int(int i)
{
  typedef TJP__Z8give_inti_0< void, void, void, void (int),
    AC::TL< int, AC::TLE > > __TJP;
  __TJP tjp;
  tjp._args[0] = (void*)&i;
  AC::invoke_Example_Example__a0_before<__TJP> (&tjp);
  ::__exec_old_give_int(i);
}

int main(){
  give_int(42);
  return 0;
}
```

7.3.10 Exposing Context Information with Pointcut Functions

```
aspect Example {
  advice execution("void give_int(%)")
    && args(wert) : before(int wert)
  {
    std::cout << wert << std::endl;
  }
};
```

```
void give_int(int i){
  std::cout <<  "Got the Int!" << std::endl;
}

int main(){
  give_int(42);
  return 0;
}
```

```
class Example{
  public: void __a0_before(int wert) {
    std::cout << wert << std::endl;
  }
};

// Invoke function typecasting JoinPoint to advice parameter
namespace AC {
  template <class JoinPoint> __attribute((always_inline)) inline
    void invoke_Example_Example__a0_before (JoinPoint *tjp)
  {
    typedef typename JoinPoint::Binding_Example_Example\
      __a0_before::template Arg<0> Arg0;
    ::Example::aspectof()->__a0_before ((int)Arg0::val (tjp));
  }
}

inline void __exec_old_give_int(int i){
  std::cout <<  "Got the Int!" << std::endl;
}

void give_int(int i){ // Wrapper function
```

```
  typedef TJP__Z8give_inti_0< void, void, void, void (int),
    AC::TL< int, AC::TLE > > __TJP;
  __TJP tjp;
  tjp._args[0] = (void*)&i;
  AC::invoke_Example_Example__a0_before<__TJP> (&tjp); // Advice
  ::__exec_old_give_int(i); // Calling old code
}

int main(){
  give_int(42);
  return 0;
}
```

Using pointcut functions makes it possible to achieve similar results to using the join point API. In this example, `args()`[3] is used to pass function parameters to the advice body. For that, `__TJP` is defined as in the join point API. Since the advice code is left clean of any `tjp` references, `__TJP` info is passed to the invoke function, which in turn typesets `__TJP` info into expected parameters for the advice call.

7.4 Inclusion of Aspect Header Files

The weaver has to guarantee that aspect header files are only compiled in a translation unit if they are affecting the shadows of code join point that are located within the translation unit. If an aspect header has to be included because of this reason, the same check has to be performed again, because the aspect header might contain code join points that are affected by other aspects.

In order to implement this behavior a forward declaration of the advice invocation function is generated and a macro `__ac_need_<`*mangled_ah_filename*`>` is defined in each file that contains a join point shadow, which is affected by an aspect that is defined in an aspect header whose mangled files name is *<mangled_ah_filename>*. Multiple inclusions shall be avoided. Therefore, another macro `__ac_have_<`*mangled_ah_filename*`>` is set wherever an aspect header is included by generated code. The following code is an example that shows the code which is generated at the end of each translation unit for each known aspect header of the project:

[3] See AspectC++ Language Reference for other pointcut context interactions

```
#ifdef __ac_need_<mangled_ah_1>
#ifndef __ac_have_<mangled_ah_1>
#define __ac_have_<mangled_ah_1>
#include "ah_1"
#endif
// other ah files that are needed if ah_1 is needed
#ifndef __ac_have_<mangled_ah_4>
#define __ac_have_<mangled_ah_4>
#include "ah_4"
#endif
#endif // __ac_need_<mangled_ah_1>
```

This code transformation pattern might result in multiple `#include` directives for the same aspect header files. This is correct, as there might be cyclic dependencies between the aspect headers.