Part I. Boost.Build v2 User Manual

Table of Contents

How to use this document

If you've just found out about Boost.Build V2 and want to know if it will work for you, start with *Tutorial*. You can continue with the *User documentation*. When you're ready to try Boost.Build in practice, go to *Installation*.

If you are about to use Boost.Build on your project, or already using it and have a problem, look at *User documentation*.

If you're trying to build a project which uses Boost.Build, look at *Installation* and then read about the section called “Command line”.

If you have questions, please post them to our mailing list, and be sure to indicate in the subject line that you're asking about Boost.Build **V2**.

Installation

This section describes how to install Boost.Build from a released source distribution. All paths are given relative to the *Boost.Build v2 root directory*, which is located in the `tools/build/v2` subdirectory of a full Boost distribution.

1. Boost.Build uses Boost.Jam, an extension of the Perforce Jam portable **make** replacement. The recommended way to get Boost.Jam is to **download a prebuilt executable** from SourceForge. If a prebuilt executable is not provided for your platform or you are using Boost's sources in an unreleased state, it may be neccessary to build **bjam** from sources included in the Boost source tree.

2. To install Boost.Jam, copy the executable, called **bjam** or **bjam.exe** to a location accessible in your `PATH`. Go to the Boost.Build root directory and run **bjam `--version`**. You should see:

   ```
   Boost.Build V2 (Milestone N)
   Boost.Jam xx.xx.xx
   ```

   where N is the version of Boost.Build you're using.

3. Configure Boost.Build to recognize the build resources (such as compilers and libraries) you have installed on your system. Open the `user-config.jam` file in the Boost.Build root directory and follow the instructions there to describe your toolsets and libraries, and, if neccessary, where they are located.

4. You should now be able to go to the `example/hello/` directory and run **bjam** there. A simple application will be built. You can also play with other projects in the `example/` directory.

If you are using Boost's CVS state, be sure to rebuild **bjam** even if you have a previous version. The CVS version of Boost.Build requires the CVS version of Boost.Jam.

When **bjam** is invoked, it always needs to be able to find the Boost.Build root directory, where the interpreted source code of Boost.Build is located. There are two ways to tell **bjam** about the root directory:

- Set the environment variable `BOOST_BUILD_PATH` to the absolute path of the Boost.Build root directory.
- At the root directory of your project or in any of its parent directories, create a file called `boost-build.jam`, with a single line:

  ```
  boost-build /path/to/boost.build ;
  ```

N.B. When **bjam** is invoked from anywhere in the Boost directory tree *other than* the Boost.Build root and its subdirectories, Boost.Build v1 is used by default. To override the default and use Boost.Build v2, you have to add the `--v2` command line option to all **bjam** invocations.

Tutorial

Hello, world

The simplest project that Boost.Build can construct is stored in `example/hello/` directory. The project is described by a file called `Jamfile` that contains:

```
exe hello : hello.cpp ;
```

Even with this simple setup, you can do some interesting things. First of all, just invoking **bjam** will build the debug variant of the **hello** executable by compiling and linking `hello.cpp`. Now, to build the release variant of **hello**, invoke

```
bjam release
```

Note that debug and release variants are created in different directories, so you can switch between variants or even build multiple variants at once, without any unneccessary recompilation. Let's extend the example by adding another line to our project's `Jamfile`:

```
exe hello2 : hello.cpp ;
```

Now we can build both the debug and release variants of our project:

```
bjam debug release
```

Note that two variants of **hello2** are linked. Since we have already built both variants of **hello**, hello.cpp won't be recompiled; instead the existing object files will just be linked into the corresponding variants of **hello2**. Now let's remove all the built products:

```
bjam --clean debug release
```

It's also possible to build or clean specific targets. The following two commands, respectively, build or clean only the debug version of **hello2**.

```
bjam hello2
bjam --clean hello2
```

Properties

To portably represent aspects of target configuration such as debug and release variants, or single- and multi-threaded builds, Boost.Build uses *features* with associated *values*. For example, the "debug-symbols" feature can have a value of "on" or "off". A *property* is just a (feature, value) pair. When a user initiates a build, Boost.Build automatically translates the requested properties into appropriate command-line flags for invoking toolset components like compilers and linkers.

There are many built-in features that can be combined to produce arbitrary build configurations. The following command builds the project's "release" variant with inlining disabled and debug symbols enabled:

```
bjam release inlining=off debug-symbols=on
```

Properties on the command-line are specified with the syntax:

feature-name=feature-value

The "release" and "debug" that we've seen in **bjam** invocations are just a shorthand way to specify values of the "variant" feature. For example, the command above could also have been written this way:

```
bjam variant=release inlining=off debug-symbols=on
```

"variant" is so commonly-used that it has been given special status as an *implicit* feature — Boost.Build will deduce the its identity just from the name of one of its values.

A complete description of features can be found here.

Build Requests and Target Requirements

The set of properties specified in the command line constitute a *build request* — a description of the desired properties for building the requested targets (or, if no targets were explicitly requested, the project in the current directory). The *actual* properties used for building targets is typically a combination of the build request and properties derived from the project's `Jamfile`s. For example, the locations of `#include`d header files are normally not specified on the command-line, but described in `Jamfile`s as *target requirements* and automatically combined with the build request for those targets. Multithread-enabled compilation is another example of a typical target requirement. The `Jamfile` fragment below illustrates how these requirements might be specified.

```
exe hello
    : hello.cpp
    : <include>/home/ghost/Work/boost <threading>multi
    ;
```

When `hello` is built, the two requirements specified above will normally always be present. If the build request given on the **bjam** command-line explictly contradicts a target's requirements, the command-line usually overrides (or, in the case of "free" feautures like `<include>`[1] , augments) the target requirements.

Project Attributes

If we want the same requirements for our other target, `hello2`, we could simply duplicate them. However, as projects grow, that approach leads to a great deal of repeated boilerplate in Jamfiles. Fortunately, there's a better way. Each project (i.e. each `Jamfile`), can specify a set of *attributes*, including requirements:

```
project
    : requirements <include>/home/ghost/Work/boost <threading>multi
    ;

exe hello : hello.cpp ;
exe hello2 : hello.cpp ;
```

the section called “Feature Attributes”

The effect would be as if we specified the same requirement for both **hello** and **hello2**.

Project Hierarchies

So far we've only considered examples with one project (i.e. with one `Jamfile`). A typical large software project would be composed of sub-projects organized into a tree. The top of the tree is called the *project root*. Besides a `Jamfile`, the project root directory contains a file called `project-root.jam`. Every other `Jamfile` in the project has a single parent project, rooted in the nearest parent directory containing a `Jamfile`. For example, in the following directory layout:

```
top/
  |
  +-- Jamfile
  +-- project-root.jam
  |
  +-- src/
  |    |
  |    +-- Jamfile
  |    `-- app.cpp
  |
  `-- util/
       |
       +-- foo/
       .    |
       .    +-- Jamfile
       .    `-- bar.cpp
```

the project root is `top/`. Because there is no `Jamfile` in `top/util/`, the projects in `top/src/` and `top/util/foo/` are immediate children of the root project.

Projects inherit all attributes (such as requirements) from their parents. Inherited requirements are combined with any requirements specified by the sub-project. For example, if `top/Jamfile` has

```
<include>/home/ghost/local
```

in its requirements, then all of its sub-projects will have it in their requirements, too. Of course, any project can add additional includes. [2] More details can be found in the section on projects.

Invoking **bjam** without explicitly specifying any targets on the command-line builds the project rooted in the current directory. Building a project does not automatically cause its sub-projects to be built unless the parent project's `Jamfile` explicitly requests it. In our example, `top/Jamfile` might contain:

```
build-project src ;
```

which would cause the project in `top/src/` to be built whenever the project in `top/` is built. However, targets in `top/util/foo/` will be built only if they are needed by targets in `top/` or `top/src/`.

Libraries and Dependent Targets

TODO: need to make this section consistent with "examples-v2/libraries".

Targets that are "needed" by other targets are called *dependencies* of those other targets. The targets that need the other targets are called *dependent* targets.

To get a feeling of target dependencies, let's continue the above example and see how `src/Jamfile` can use libraries from `util/foo`. Assume util/foo/Jamfile contains:

the section called “Feature Attributes”

```
lib bar : bar.cpp ;
```

Then, to use this library in `src/Jamfile`, we can write:

```
exe app : app.cpp ../util/foo//bar ;
```

While `app.cpp` refers to a regular source file, `../util/foo//bar` is a reference to another target: a library "bar" declared in the `Jamfile` at `../util/foo`. When linking the **app** executable, the appropriate version of `bar` will be built and linked in. What do we mean by "appropriate"? For example, suppose we build "app" with:

```
bjam app optimization=full cxxflags=-w-8080
```

Which properties must be used to build `foo`? The answer is that some properties are *propagated* — Boost.Build attempts to use dependencies with the same value of propagated features. The optimization> feature is propagated, so both "app" and "foo" will be compiled with full optimization. But <cxxflags> feature is not propagated: its value will be added as-is to compiler flags for "a.cpp", but won't affect "foo". There is still a couple of problems. First, the library probably has some headers which must be used when compiling "app.cpp". We could use requirements on "app" to add those includes, but then this work will be repeated for all programs which use "foo". A better solution is to modify util/foo/Jamfilie in this way:

```
project
    : usage-requirements <include>.
    ;

lib foo : foo.cpp ;
```

Usage requirements are requirements which are applied to dependents. In this case, <include> will be applied to all targets which use "foo" — i.e. targets which have "foo" either in sources or in dependency properties. You'd need to specify usage requirements only once, and programs which use "foo" don't have to care about include paths any longer. Or course, the path will be interpreted relatively to "util/foo" and will be adjusted according to the **bjam**s invocation directory. For example, if building from project root, the final compiler's command line will contain `-Ilib/foo`.

The second problem is that we hardcode the path to library's Jamfile. Imagine it's hardcoded in 20 different places and we change the directory layout. The solution is to use project ids — symbolic names, not tied to directory layout. First, we assign a project id to Jamfile in util/foo:

```
project foo
    : usage-requirements <include>.
    ;
```

Second, we use the project id to refer to the library in src/Jamfile:

```
exe app : app.cpp /foo//bar ;
```

The "/foo//bar" syntax is used to refer to target "foo" in project with global id "/foo" (the slash is used to specify global id). This way, users of "foo" do not depend on its location, only on id, which is supposedly stable. The only thing left, it to make sure that src/Jamfile knows the project id that it uses. We add to top/Jamfile the following line:

```
use-project /foo : util/foo ;
```

Now, all projects can refer to "foo" using the symbolic name. If the library is moved somewhere, only a single line in the top-level Jamfile should be changed.

Library dependencies

The previous example was simple. Often, there are long chains of dependencies between libraries. The main application is a thin wrapper on top of library with core logic, which uses library of utility functions, which uses boost filesystem library. Expressing these dependencies is straightforward:

```
lib utils : utils.cpp /boost/filesystem//fs ;
lib core : core.cpp utils ;
exe app : app.cpp core ;
```

So, what's the reason to even mention this case? First, because it's a bit more complex that it seems. When using shared linking, libraries are build just as written, and everything will work. However, what happens with static linking? It's not possible to include another library in static library. Boost.Build solves this problem by returning back library targets which appear as sources for static libraries. In this case, if everything is built statically, the "app" target will link not only "core" library, but also "utils" and "/boost/filesystem//fs".

So, the net result is that the above code will work for both static linking and for shared linking.

Sometimes, you want all applications in some project to link to a certain library. Putting the library in sources of all targets is possible, but verbose. You can do better by using the <source> property. For example, if "/boost/filesystem//fs" should be linked to all applications in your project, you can add <source>/boost/filesystem//fs to requirements of the project, like this:

```
project
   : requirements <source>/boost/filesystem//fs
   ;
```

Static and shared libaries

While the previous section explained how to create and use libraries, it omitted one important detail. Libraries can be either *static*, which means they are included in executable files which use them, or *shared* (a.k.a. *dynamic*), which are only referred to from executables, and must be available at run time. Boost.Build can work with both types. By default, all libraries are shared. This is much more efficient in build time and space. But the need to install all libraries to some location is not always convenient, especially for debug builds. Also, if the installed shared library changes, all application which use it might start to behave differently.

Static libraries do not suffer from these problems, but considerably increase the size of application. Before describing static libraries, it's reasonable to give another, quite simple approach. If your project is built with <hardcode-dll-paths>true property, then the application will include the full paths for all shared libraries, eliminating the above problems. Unfortunately, you no longer can move shared library to a different location, which makes this option suitable only for debug builds. Further, only gcc compiler supports this option.

Building a library statically is easy. You'd need to change the value of <link> feature from it's deafault value `shared`, to `static`. So, to build everything as static libraries, you'd say

```
bjam link=static
```

on the command line. The linking mode can be fine-tuned on per-target basis.

1. Suppose your library can be only build statically. This is easily achieved using requirements:

   ```
   lib l : l.cpp : <link>static ;
   ```

2. What if library can be both static and shared, but when using it in specific executable, you want it static? Target references are here to help:

   ```
   exe important : main.cpp helpers/<link>static ;
   ```

3. What if the library is defined in some other project, which you cannot change. But still, you want static linking to that library in all cases. You can use target references everywhere:

   ```
   exe e1 : e1.cpp /other_project//bar/<link>static ;
   exe e10 : e10.cpp /other_project//bar/<link>static ;
   ```

   but that's far from being convenient. Another way is to introduce a level of indirection: create a local target, which will refer to static version of `foo`. Here's the solution:

   ```
   alias foo : /other_project//bar/<link>static ;
   exe e1 : e1.cpp foo ;
   exe e10 : e10.cpp foo ;
   ```

   Note that the alias rule is specifically used for rename a reference to a target and possibly change the properties.

Conditions and alternatives

As we've just figured out, properties can significally affect the way targets are built. The processing of the <link> feature is built in the build system, and is quite complex. But there is a couple of mechanisms which allow ordinary users to do different things depending on properties.

The first mechanism is called *conditinal requirement*. For example, you might want to set specific defines when the library is build as shared, or you have your own define to be used in release mode. Here's a piece of Jamfile.

```
lib network : network.cpp
    : <link>shared:<define>NEWORK_LIB_SHARED
      <variant>release:<define>EXTRA_FAST
    ;
```

This will have exactly the effect we wanted: whenever <link>shared is in properties, <define>NEWORK_LIB_SHARED will be in properties as well.

Sometimes different variant of a target are so different, that describing them using conditional requirements would be hard. Imagine that a library has different sources on two supported toolsets, and dummy implementation for all the other toolset. We can express this situation using *target alternatives*:

```
lib demangler : dummy_demangler.cpp ;
lib demangler : demangler_gcc.cpp : <toolset>gcc ;
lib demangler : demangler_msvc.cpp : <toolset>msvc ;
```

The proper alternative will be automatically selected.

Prebuilt targets

We've just learned how to use libraries which are created by Boost.Build. But some libraries are not. At the same time, those libraries can have different versions (release and debug, for example), that we should select depending on build properties. Prebuilt targets provide a mechanism for that. Jamfile in util/lib2 can contain:

```
lib lib2
    :
    : <file>lib2_release.a <variant>release
    ;

lib lib2
    :
    : <file>lib2_debug.a <variant>debug
    ;
```

This defines two alternatives for target "lib2", and for each one names a prebuilt file. Naturally, there are no sources. Instead, the <file> feature is used to specify the file name. Which alternative is selected depends on properties of dependents. If "app" binary should use "lib2", we can write:

```
exe app : app.cpp ../util/lib2//lib2 ;
```

If we build release version of "app", then it will be linked with "lib2_release.a", and debug version will use "lib2_debug.a". Another important kind of prebuilt targets are system libraries — more specifically, libraries which are automatically found by the compiler. E.g. gcc uses "-l" switch for that. Such libraries should be declared almost like regular ones:

```
lib zlib : : <name>z ;
```

We again don't specify any sources, but give a name which should be passed to the compiler. In this example, and for gcc compiler, the "-lz" option will be added. Paths where library should be searched can also be specified:

```
lib zlib : : <name>z <search>/opt/lib ;
```

And, of course, two variants can be used:

```
lib zlib : : <name>z <variant>release ;
lib zlib : : <name>z_d <variant>debug ;
```

Of course, you'll probably never in your life need debug version of zlib, but for other libraries this is quite reasonable.

More advanced use of prebuilt target is described in a FAQ entry.

User documentation

This section will provide the information necessary to create your own projects using Boost.Build. The information provided here is relatively high-level, and detailed reference as well as the on-line help system must be used to obtain low-level documentation (see the help option).

The Boost.Build actually consists of two parts - Boost.Jam, which is a build engine with its own interpreted language, and Boost.Build itself, implemented in Boost.Jam's language. The chain of event which happen when you type "bjam" on the command is:

1. Boost.Jam tries to find Boost.Build and loads the top-level module. The exact process is described in the section on initialization
2. Boost.Build top-level module loads user-defined configuration files, "user-config.jam" and "site-config.jam", which define available toolsets.
3. The Jamfile in the current directory is read. That in turn might cause reading of further Jamfiles. As a result, a tree of projects is created, with targets inside projects.
4. Finally, using build request specified on the command line, Boost.Build decides which targets should be built, and how. That information is passed back to Boost.Jam, which takes care of actually running commands.

So, to be able to successfully use Boost.Build, you'd need to know only three things:

- How to configure Boost.Build
- How to write Jamfiles
- How the build process works

Configuration

The Boost.Build configuration is specified in the file "user-config.jam". You can edit the one which comes with Boost.Build, or create a copy in your home directory and edit that. (See the reference for the exact search paths.) The primary function of that file is to declarate which compilers and other tools are available. The simplest syntax to configure a tool is:

```
using <tool-name> ;
```

The "using" rule is given a name of tool, and will make that tool available to Boost.Build. For example, "using gcc ;" will make available the gcc compiler.

Since nothing but tool name is specified, Boost.Build will pick some default settings -- for example will use gcc found in path, or look in some known installation locations. For ordinary users, this is quite fine. In case you have several version of a compiler, or it's located in some unusual location, or you need to tweak the configuration, you'd need to pass additional parameters to the "using" rule. Generally, for every tool module, the parameters differ, and you can obtain the documentaiton by running

```
bjam --help <tool-name>.init
```

on the command line. However, for all compilers the meaning of the first three parameters is the same: version, invocation command and options.

The "version" parameter identifies the compiler, in case you have several. It can have any form you like, but it's recommended that you use a numeric identifier, like "7.1". The "invocation command" parameter is the command which must be executed to run the compiler. This might be just compiler name, or a name with a path in it. Here are some examples.

To configure a compiler installed in non-standard location and not present in path, you can do the following:

```
using msvc : : Z:/Programs/Microsoft Visual Studio/vc98/bin/cl.exe ;
```

To configure several versions of a compiler, the following can be used.

```
using gcc : 3.3 ;
using gcc : 3.4 : g++-3.4 ;
using gcc : 3.2 : g++-3.2 ;
```

Note that in the first call to "using", the compiler found in path will be used, and there's no need to explicitly specify the command.

As shown above, both "version" and "invocation command" parameters are optional, but there's an important restriction: if you configure the same compiler more then once, you must pass the "version" parameter every time. For example, the following is not allowed:

```
using gcc ;
using gcc : 3.4 : g++-3.4 ;
```

because the first "using" does not specify the version.

The `options` parameter is used to fine-tune the configuration. All compilers allow to pass four option, intentionally similiar in spelling to builtin features: `cflags`, `cxxflags`, `compileflags` and `linkflags`. They specify additional options which will be always passed to the corresponding tools. The `cflags` option applies only to the C compiler, the `cxxflags` option applies only to the C++ compiler and the `compileflags` options applies to both. For example, to use 64 bit mode with gcc you can use:

```
using gcc : 3.4 : : <compileflags>-m64 <linkflags>-m64 ;
```

Writing Jamfiles

Overview

Jamfiles are the thing which is most important to the user, bacause they declare the targets which should be build. Jamfiles are also used for organizing targets -- each Jamfile is a separate project, which can be build independently from the other projects.

Jamfile mostly contain calls to Boost.Build functions, which do all the work, specifically:

- declare main targets

- define project properties
- do various other things

In addition to Jamfiles, Boost.Build has another user-editable file, project-root.jam, which is mostly useful to declare constants global to all the projects. It is described in more detail below.

Main targets

Main target is a user-defined named entity which can be build, for example a named executable file. Declaring a main target is usually done using one of main target functions. The user can also declare custom main target function.

Most main targets rules in Boost.Build use similiar syntax:

```
function-name main-target-name
    : sources
    : requirements
    : default-build
    : usage-requirements
    ;
```

- "main-target-name" is the name used to request the target on command line and to use it from other main targets. Main target name may contain alphanumeric characters and symbols '-' and '_';
- "sources" is the list of source files and other main targets that must be combined.
- "requirements" is the list of properties that must always be present when this main target is built.
- "default-build" is the list of properties that will be used unless some other value of the same feature is already specified.
- "usage-requirements" is the list of properties that will be propagated to all main targets that use this one, i.e. to all dependents.

Note that the actual requirements, default-build and usage-requirements attributes for a target are obtained by combining the explicitly specified one with those specified for the project where a target is declared.

Some main target rules have shorter list of parameters, and you should consult their documentation for details.

The list of sources specifies what should be processed to get the resulting targets. Most of the time, it's just a list of files. Sometimes, you'd want to use all files with the same extension as sources, in which case you can use the "glob" rule. Here are two examples:

```
exe a : a.cpp ;
exe b : [ glob *.cpp ] ;
```

Unless you specify a files with absolute path, the name is considered relative to the source directory -- which is typically the same as directory when Jamfile is located, but can be changed as described here

The list of sources can also reference other main targets. The targets in the same project can be referred by using the name, and targets in other project need to specify directory or a symbolic name of the other project. For example:

```
lib helper : helper.cpp ;
exe a : a.cpp helper ;
exe b : b.cpp ..//utils ;
exe c : c.cpp /boost/program_options//program_opions ;
```

The first exe uses the library defined in the same project. The second one uses some target (most likely library) defined by Jamfile one level higher. Finally, the third target uses some C++ Boost library, using the symbolic name to refer to it. More information about it can be found in tutorial and in target id reference.

Requirements are the properties that should always be present when building a target. Typically, they are includes and defines:

```
exe hello : hello.cpp : <include>/opt/boost <define>MY_DEBUG ;
```

In special circumstances, other properties can be used, for example if a library does not work if it's shared, or a file can't be compiled with optimization due to a compiler bug, one can use

```
lib util : util.cpp : <link>static ;
obj main : main.cpp : <optimization>off ;
```

Sometimes, requirements are necessary only for a specific compiler, or build variant. The conditional properties can be used in that case:

```
lib util : util.cpp : <toolset>msvc:<link>static ;
```

In means when whenever `<toolset>msvc` property is in build properties, the `<link>static` property will be included as well. The conditional requirements can be "chained":

```
lib util : util.cpp : <toolset>msvc:<link>static
                      <link>static:<define>STATIC_LINK ;
```

will set of static link and the `STATIC_LINK` define on the `msvc` toolset.

The default-build attribute is a set of properties which should be used if build request does not specify a value. For example:

```
exe hello : hello.cpp : : <threading>multi ;
```

would build the target in multi-threaded mode, unless the user explicitly requests single-threaded version. The difference between requirements and default-build is that requirements cannot be overriden in any way.

A target of the same name can be declared several times. In that case is declaration is called an *alternative*. When the target is build, one of the alternatives will be selected and use. Alternatives need not be defined by the same main target rule. The following is OK:

```
lib helpers : helpers.hpp ;
alias helpers : helpers.lib : <toolset>msvc ;
```

Building of the same main target can differ greatly from platform to platform. For example, you might have different list of sources for different compilers, or different options for those compilers. Two approaches to this are explained in the tutorial.

Sometimes a main target is really needed only by some other main target. For example, a rule that declares a test-suite uses a main target that represent test, but those main targets are rarely needed by themself.

It is possible to declare target inline, i.e. the "sources" parameter may include call to other main rules. For example:

```
exe hello : hello.cpp
    [ obj helpers : helpers.cpp : <optimization>off ] ;
```

Will cause "helpers.cpp" to be always compiled without optimization. It's possible to request main targets declared inline, but since they are considered local, they are renamed to "parent-main-target_name..main-target-name". In the example above, to build only helpers, one should run "bjam hello..helpers".

Projects

As mentioned before, targets are grouped into project, and each Jamfile is a separate project. Projects are useful because it allows to group related targets together, define properties common to all those targets, and assign a symbolic name to the project, allowing to easily refer to the targets in the project. Two last goals are accompished with the "project" rule.

The rule has this syntax

```
project id : <attributes> ;
```

Here, attributes is a sequence of (attribute-name, attribute-value) pairs. The list of attribute names along with its handling is also shown in the table below. For example, it is possible to write:

```
project tennis
    : requirements <threading>multi
    : default-build release
    ;
```

The possible attributes are listed below.

Project id is a short way to denote a project, as opposed to the Jamfile's pathname. It is a hierarchical path, unrelated to filesystem, such as "boost/thread". Target references make use of project ids to specify a target.

Source location specifies the directory where sources for the project are located.

Project requirements are requirements that apply to all the targets in the projects as well as all subprojects.

Default build is the build request that should be used when no build request is specified explicitly.

The default values for those attributes are given in the table below.

Table 1.

Attribute	**Name for the 'project' rule**	**Default value**	**Handling by the 'project' rule**
Project id	none	none	Assigned from the first parameter of the 'project' rule. It is assumed to denote absolute project id.
Source location	`source-location`	The location of jamfile for the project	Sets to the passed value
Requirements	`requirements`	The parent's requirements	The parent's requirements are refined with the passed requirement and the result is used as the project requirements.
Default build	`default-build`	none	Sets to the passed value
Build directory	`build-dir`	If parent has a build dir set, the value of it, joined with the relative path from parent to the current project. Otherwise, empty	Sets to the passed value, interpreted as relative to the project's location.

Additional Jamfile rules

There's a number of other helper rules which can be used in Jamfile, described in the following table.

Table 2.

Rule	**Semantic**
project	Define project attributes.
use-project	Make another project known.
build-project	Build another project when this one is built.
explicit	States that the target should be built only by explicit request.
glob	Takes a list of wildcards, and returns the list of files which match any of the wildcards.

Project root

Each project is also associated with *project root*. That's a root for a tree of projects, which specifies some global properties.

Project root for a projects is the nearest parent directory which contains a file called `project-root.jam`. That file defines certain properties which apply to all projects under project root. It can:

- configure toolsets, via call to `toolset.using`

- refer to other projects, via the `use-project` rule
- declare constants, via the `constant` and `path-constant` rules.

To facilitate declaration of simple projects, Jamfile and project-root can be merged together. To achieve this effect, the project root file should call the `project` rule. The semantic is precisely the same as if the call was made in Jamfile, except that project-root.jam will start to serve as Jamfile. The Jamfile in the directory of project-root.jam will be ignored, and project-root.jam will be able to declare main targets as usual.

Build process

When you've described your targets, you want Boost.Build to run the right tools and create the needed targets. This section will describe two things: how you specify what to build, and how the main targets are actually constructed.

The most important thing to note is that in Boost.Build, unlike other build tools, the targets you declare do not correspond to specific files. What you declare in Jamfiles is more like "metatarget". Depending on the properties that you specify on the command line, each "metatarget" will produce a set of real targets corresponding to the requested properties. It is quite possible that the same metatarget is build several times with different properties, and will, of course, produce different files.

Tip

This means that for Boost.Build, you cannot directly obtain build variant from Jamfile. There could be several variants requested by the user, and each target can be build with different properties.

Build request

The command line specifies which targets to build and with what properties. For example:

```
bjam app1 lib1//lib1 toolset=gcc variant=debug optimization=full
```

would build two targets, "app1" and "lib1//lib1" with the specified properties. You can refer to any targets, using target id and specify arbitrary properties. Some of the properties are very common, and for them the name of the property can be omitted. For example, the above can be written as:

```
bjam app1 lib1//lib1 gcc debug optimization=full
```

The complete syntax which has some additional shortcuts if described here.

Building a main target

When you request, directly or indirectly, a build of a main target with specific requirements, the following steps are made. Some brief explanation is provided, and more detailes are given in the reference.

1. Applying default build. If the default-build property of a target specifies a value of a feature which is not present in the build request, that value is added.
2. Selecting the main target alternative to use. For each alternative we look how many properties are present both in alternative's requirements, and in build request. The alternative with large number of matching properties is selected.

3. Determining "common" properties. The build request is refined with target's requirements. The conditional properties in requirements are handled as well. Finally, default values of features are added.

4. Building targets referred by the sources list and dependency properties. The list of sources and the properties can refer to other target using target references. For each reference, we take all propagated properties, refine them by explicit properties specified in the target reference, and pass the resulting properties as build request to the other target.

5. Adding the usage requirements produces when building dependencies to the "common" properties. When dependencies are built in the previous step, they return both the set of created "real" targets, and usage requirements. The usage requirements are added to the common properties and the resulting property set will be used for building the current target.

6. Building the target using generators. To convert the sources to the desired type, Boost.Build uses "generators" --- objects which correspond to tools like compilers and linkers. Each generator declares what type of targets in can produce and what type of sources it requires. Using this information, Boost.Build determines which generators must be run to produce a specific target from specific sources. When generators are run, they return the "real" targets.

7. Computing the usage requirements to be returned. The conditional properties in usage requirements are expanded and the result is returned.

Building a project

Often, user request a build of a complete project, not just one main target. In fact, invoking **bjam** without parameters builds the project defined in the current directory.

When a project is build, the build request is passed without modification to all main targets in that project. It's is possible to prevent implicit building of a target in a project with the `explicit` rule:

```
explicit hello_test ;
```

would cause the `hello_test` target to be built only if explicitly requested by the user or by some other target.

The Jamfile for a project can include a number of `build-project` rule calls, that specify additional projects to be built.

Builtin target types

Programs

Programs are created using the `exe` rule, which follows the common syntax. For example:

```
exe hello : hello.cpp some_library.lib /some_project//library
          : <threading>multi
          ;
```

This will create an executable file from the sources -- in this case, one C++ file, one library file present in the same directory, and another library which is created by Boost.Build. Generally, sources can include C and C++ files, object files and libraries. Boost.Build will automatically try to convert targets of other types.

Tip

On Windows, if an application uses dynamic libraries, and both the application and the libraries are built by Boost.Build, its not possible to immediately run the application, because the `PATH` environment variable should include the path to the libraries. It means you have to either add the paths manually, or place the application and the libraries to the same directory, for example using the stage rule.

Libraries

Libraries are created using the `lib` rule, which follows the common syntax. For example:

```
lib helpers : helpers.cpp : <include>boost : : <include>. ;
```

In the most common case, the `lib` creates a library from the specified sources. Depending on the value of <link> feature the library will be either static or shared. There are two other cases. First is when the library is installed somewhere in compiler's search paths, and should be searched by the compiler (typically, using the `-l` option). The second case is where the library is available as a prebuilt file and the full path is known.

The syntax for these case is given below:

```
lib z : : <name>z <search>/home/ghost ;
lib compress : : <file>/opt/libs/compress.a ;
```

The `name` property specifies the name which should be passed to the `-l` option, and the `file` property specifies the file location. The `search` feature specifies paths where the library should be searched. That feature can be specified several time, or can be omitted -- in which case only default compiler paths will be searched.

The difference between using the `file` feature as opposed to the `name` name feature together with the `search` feature is that `file` is more precise. A specific file will be used. On the other hand, the `search` feature only adds a library path, and the `name` feature gives the basic name of the library. The search rules are specific to the linker. For example, given these definition:

```
lib a : : <variant>release <file>/pool/release/a.so ;
lib a : : <variant>debug <file>/pool/debug/a.so ;
lib b : : <variant>release <file>/pool/release/b.so ;
lib b : : <variant>debug <file>/pool/debug/b.so ;
```

It's possible to use release version of `a` and debug version of `b`. Had we used the `name` and `search` features, the linker would always pick either release or debug versions.

For convenience, the following syntax is allowed:

```
lib z ;
lib gui db aux ;
```

and is does exactly the same as:

```
lib z : : <name>z ;
lib giu : : <name>gui ;
lib db : : <name>db ;
lib aux : : <name>aux ;
```

When a library uses another library you should put that another library in the list of sources. This will do the right thing in all cases. For portability, you should specify library dependencies even for searched and prebuilt libraries, othewise, static linking on Unix won't work. For example:

```
lib z ;
lib png : z : <name>png ;
```

Note

When a library (say, `a`), which has another library, (say, `b`) is linked dynamically, the `b` library will be incorporated in `a`. (If `b` is dynamic library as well, then `a` will only refer to it, and not include any extra code.) When the `a` library is linked statically, Boost.Build will assure that all executables which link to `a` will also link to `b`.

One feature of Boost.Build which is very important for libraries is usage requirements. For example, if you write:

```
lib helpers : helpers.cpp : : : <include>. ;
```

then compiler include path for all targets which use `helpers` will contain the directory where the target is defined.path to "helpers.cpp". So, the user need only to add `helpers` to the list of sources, and don't bother about other requirements. This allows to greatly simplify Jamfiles.

Note

If you don't want shared libraries to include all libraries which are specified in sources (especially statically linked ones), you'd need to use the following:

```
lib b : a.cpp ;
lib a : a.cpp : <use>b : : <library>b ;
```

This specifies that `a` uses `b`, and causes all executables which link to `a` also link to `b`. In this case, even for shared linking, the `a` library won't even refer to `b`.

Alias

The `alias` rule follows the common syntax. For example:

```
alias core : im reader writer ;
```

will build the sources and return the generated source targets without modification.

The `alias` rule is a convenience tool. If you often build the same group of targets at the same time, you can define the alias to save typing.

Another use of the `alias` rule is to change build properties. For example, if you always want static linking for a specific C++ Boost library, you can write the following:

```
alias boost_thread : /boost/thread//boost_thread : <link>static ;
```

and use only the `boost_thread` alias in your Jamfiles.

It is also allowed to specify usage requirements for the `alias` target. If you write the following:

```
alias header_only_library : : : :  <include>/usr/include/header_only_library ;
```

then using `header_only_library` in sources will only add an include path. Also note that when there are some sources, their usage requirements are propagated, too. For example:

```
lib lib : lib.cpp : : : <include>. ;
alias lib_alias ;
exe main : main.cpp lib_alias ;
```

will compile `main.cpp` with the additional include.

Installing

For installing the built target you should use the `stage` rule follows the common syntax. For example:

```
stage dist : hello helpers ;
```

will cause the targets `hello` and `helpers` to be moved to the `dist` directory. The directory can be changed with the `location` property:

```
stage dist : hello helpers : <location>/usr/bin ;
```

Specifying the names of all libraries to install can be boring. The `stage` allows to specify only the top-level executable targets to install, and automatically install all dependencies:

```
stage dist : hello
           : <traverse-dependencies>on <include-type>EXE
             <include-type>LIB
           ;
```

will find all targets that `hello` depends on, and install all of the which are either executables or libraries.

Testing

Boost.Build has convenient support for running unit tests. The simplest way is the `unit-test` rule, which follows the common syntax. For example:

```
unit-test helpers_test : helpers_test.cpp helpers ;
```

The `unit-test` rule behaves like the `exe` rule, but after the executable is created it is run. If the executable returns error, the build system will also return error and will try running the executable on the next invocation until it runs successfully. This behaviour ensures that you can't miss a unit test failure.

There are rules for more elaborate testing: `compile`, `compile-fail`, `run` and `run-fail`. They are more suitable for automated testing, and are not covered here yet.

Builtin features

`variant`

The feature which combines several low-level features in order to make building most common variants simple.

Allowed values:`debug`, `release`, `profile`

The value `debug` expands to

```
<optimization>off <debug-symbols>on <inlining>off <
```

The value `release` expands to

```
<optimization>speed <debug-symbols>off <inlining>fu
```

The value `profile` expands to the same as `release`, plus:

```
<profiling>on <debug-symbols>on
```

Rationale: Runtime debugging is on in debug build to suit expectations of people used various IDEs. It's assumed other folks don't have any specific expectation in this point.

`link`

Feature which controls how libraries are built.

Allowed values:`shared`, `static`

`source`

Tthe <source>X feature has the same effect on building a target as putting X in the list of sources. The feature is sometimes more convenient: you can put <source>X in the requirements for a project and it will be linked to all executables.

`library`

This feature is equivalent to the <source> feature, and exists for backward compatibility reasons.

`use`

Causes the target referenced by the value of this feature to be constructed and adds it's usage requirements to build properties. The constructed targets are not used in any other way. The primary use case is when you use some library and want it's usage requirements (such as include paths) to be applied, but don't want to link to the library.

`dll-path`

Specify an additional path where shared libraries should be searched where the executable or shared library is run. This feature only affect Unix compilers. Plase see the FAQ entry for details.

`hardcode-dll-paths`

Controls automatic generation of dll-path properties.

Allowed values:`true`, `false`. This property is specific to Unix systems. If an executable is build with `<hardcode-dll-paths>true`, the generated binary will contain the list of all the paths to the used shared libraries. As the result, the executable can be run without changing system paths to shared libraries, or installing the libraries to system paths. This is very convenient during development. Plase see the FAQ entry for details.

Differences to Boost.Build V1

While Boost.Build V2 is based on the same ideas as Boost.Build V1, some of the syntax was changed, and some new important features were added. This chapter describes most of the changes.

Configuration

In V1, there were two methods to configure a toolset. One is to set some environment variable, or use "-s" command line option to set variable inside BJam. Another method was creating new toolset module, which would set the variables and then invoke basic toolset. Neither method is necessary now, the "using" rule provides a consistent way to initialize toolset, including several versions. See section on configuraton for details.

Writing Jamfiles

Probably one of the most important differences in V2 Jamfiles is the project requirements. In V1, if several targets have the same requirements (for example, common include path), it was necessary to manually write that requirements, or use a helper rule. In V2, the common properties can be specified with the "requirements" project attribute, as documented here.

The usage requirements is also important mechanism to simplify Jamfile. If a library requires all clients to use specific includes, or macros when compiling the code which depends on the library, this information can be cleanly represented.

The difference between "lib" and "dll" targets in V1 is completely eliminated in V2. There's only one target -- "lib", which can create either static or shared library depending on the value of the <link> feature. If your target should be only build in one variant, you can add <link>shared or <link>static to requirements.

The syntax for referring to other targets was changed a bit. While in V1 one would use:

```
exe a : a.cpp <lib>../foo/bar ;
```

the V2 syntax is:

```
exe a : a.cpp ../foo//bar ;
```

Note that you don't need to specify the type of other target, but the last element should be separated to double slash, to indicate that you're referring to target "bar" in project "../foo", and not to project "../foo/bar".

Build process

The command line syntax in V2 is completely different. For example

```
bjam -sTOOLS=msvc -sBUILD=release some_target
```

now becomes:

```
bjam toolset=msvc variant=release some_target
```

or, using shortcuts, just:

```
bjam msvc release some_target
```

See the reference for complete description of the syntax.

Extender Manual

Introduction

This document explains how to extend Boost.Build to accomodate your local requirements. Let's start with quite simple, but realistic example.

Say you're writing an application which generates C++ code. If you ever did this, you know that it's not nice. Embedding large portions of C++ code in string literals is very awkward. A much better solution is:

1. Write the template of the code to be generated, leaving placeholders at the points which will change
2. Access the template in your application and replace placeholders with appropriate text.
3. Write the result.

It's quite easy to achieve. You write special verbatim files, which are just C++, except that the very first line of the file gives a name of variable that should be generated. A simple tool is created which takes verbatim file and creates a cpp file with a single char* variable, which name is taken from the first line of verbatim file, and which value is properly quoted content of the verbatim file.

Let's see what Boost.Build can do.

First off, Boost.Build has no idea about "verbatim files". So, you must register a new type. The following code does it:

```
import type ;
type.register VERBATIM : verbatim ;
```

The first parameter to 'type.register' gives the name of declared type. By convention, it's uppercase. The second parameter is suffix for this type. So, if Boost.Build sees "code.verbatim" in the list of sources, it knows that it's of type `VERBATIM`.

Lastly, you need a tool to convert verbatim files to C++. Say you've sketched such a tool in Python. Then, you have to inform Boost.Build about the tool. The Boost.Build concept which represents a tool is *generator*.

First, you say that generator 'inline-file' is able to convert VERBATIM type into C++:

```
import generators ;
generators.register-standard verbatim.inline-file : VERBATIM : CPP ;
```

Second, you must specify the commands to be run to actually perform convertion:

```
actions inline-file
{
    "./inline-file.py" $(<) $(>)
}
```

Now, we're ready to tie it all together. Put all the code above in file "verbatim.jam", add "import verbatim ;" to "project-root.jam", and it's possible to write the following in Jamfile:

```
exe codegen : codegen.cpp class_template.verbatim usage.verbatim ;
```

The verbatim files will be automatically converted into C++ and linked it.

In the subsequent sections, we will extend this example, and review all the mechanisms in detail. The complete code is available in example/customization directory.

Target types

The first thing we did in the intruduction was declaring a new target type:

```
import type ;
type.register VERBATIM : verbatim ;
```

The type is the most important property of a target. Boost.Build can automatically generate necessary build actions only because you specify the desired type (using the different main target rules), and because Boost.Build can guess the type of sources from their extensions.

The first two parameters for the `type.register` rule are the name of new type and the list of extensions associated with it. A file with an extension from the list will have the given target type. In the case where a target of the declared type is generated from other sources, the first specified extension will be used. This behaviour can be changed using the `type.set-generated-target-suffix` rule.

Something about 'main' types.

Something about base types.

Scanners

Sometimes, a file can refer to other files via some include mechanism. To make Boost.Build track dependencies to the included files, you need to provide a scanner. The primary limitation is that only one scanner can be assigned to a target type.

First, we need to declare a new class for the scanner:

```
class verbatim-scanner : common-scanner
{
    rule pattern ( )
    {
        return "//###include[ ]*\"([^\"]*)\"" ;
    }
}
```

All the complex logic is in the `common-scanner` class, and you only need to override the method which returns the regular expression to be used for scanning. The paranthethis in the regular expression indicate which part of the string is the name of the included file.

After that, we need to register our scanner class:

```
scanner.register verbatim-scanner : include ;
```

The value of the second parameter, in this case `include`, specifies which properties contain the list of paths which should be searched for the included files.

Finally, we assign the new scaner to the `VERBATIM` target type:

```
type.set-scanner VERBATIM : verbatim-scanner ;
```

That's enough for scanning include dependencies.

Tools and generators

This section will describe how Boost.Build can be extended to support new tools.

For each additional tool, a Boost.Build object called generator must be created. That object has specific types of targets which it accepts an produces. Using that information, Boost.Build is able to automatically invoke the generator. For example, if you declare a generator which takes a target of the type `D` and produces a target of the type `OBJ`, when placing a file with extention `.d` in a list of sources will cause Boost.Build to invoke your generator, and then to link the resulting object file into an application. (Of course, this requires that you specify that the `.d` extension corresponds to the `D` type.)

Each generator should be an instance of a class derived from the `generator` class. In the simplest case, you don't need to create a derived class, but simply create an instance of the `generator` class. Let's review the example we've seen in the introduction.

```
import generators ;
generators.register-standard verbatim.inline-file : VERBATIM : CPP ;
actions inline-file
{
    "./inline-file.py" $(<) $(>)
}
```

We declare a standard generator, specifying its id, the source type and the target type. When invoked, the generator will create a target of type `CPP` which will have the source target of type `VERBATIM` as the only source. But what command will be used to actually generate the file? In bjam, actions are specified using named "actions" blocks and the name of the action block should be specified when creating targets. By convention, generators use the same name of the action block as their own id. So, in above example, the "inline-file" actions block will be use to convert the source into the target.

There are two primary kinds of generators: standard and composing, which are registered with the `generators.register-standard` and the `generators.register-composing` rules, respectively. For example:

```
generators.register-standard verbatim.inline-file : VERBATIM : CPP ;
generators.register-composing mex.mex : CPP LIB : MEX ;
```

The first generators takes a *single* source of type `VERBATIM` and produces a result. The second generator takes any number of sources, which can have either the `CPP` or the `LIB` type. Composing generators are typically used for generating top-level target type. For example, the first generator invoked when building an `exe` target is a composing generator corresponding to the proper linker.

You should also know about two specific function for registering generators: `generators.register-c-compiler` and `generators.register-linker`. The first sets up header dependecy scanning for C files, and the seconds handles various complexities like searched libraries. For that reason, you should always use those functions when adding support for compilers and linkers.

(Need a note about UNIX)

Custom generator classes

The standard generators allows you to specify source and target types, action, and a set of flags. If you need anything more complex, you need to create a new generator class with your own logic. Then, you have to create an instance of that class and register it. Here's an example how you can create your own generator class:

```
class custom-generator : generator
{
    rule __init__ ( * : * )
    {
        generator.__init__ $(1) : $(2) : $(3) : $(4) : $(5) : $(6) : $(7) : $(8) :
    }
}

generators.register
  [ new custom-generator verbatim.inline-file : VERBATIM : CPP ] ;
```

This generator will work exactly like the `verbatim.inline-file` generator we've defined above, but it's possible to customize the behaviour by overriding methods of the `generator` class.

There are two methods of interest. The `run` methods is responsible for overall process - it takes a number of source targets, converts them the the right types, and creates the result. The `generated-targets` method is called when all sources are converted to the right types to actually create the result.

The `generated-target` method can be overridden when you want to add additional properties to the generated targets or use additional sources. For example (which is real), you have a tool for analysing programs, which should be given a name of executable and the list of all sources. Naturally, you don't want to list all source files manually. Here's how the `generated-target` method can find the list of sources automatically:

```
class itrace-generator : generator {
....
    rule generated-targets ( sources + : property-set : project name ? )
    {
        local leafs ;
        local temp = [ virtual-target.traverse $(sources[1]) : : include-sources ]
        for local t in $(temp)
        {
            if ! [ $(t).action ]
            {
                leafs += $(t) ;
            }
        }
        return [ generator.generated-targets $(sources) $(leafs)
          : $(property-set) : $(project) $(name) ] ;
    }
}
generators.register [ new itrace-generator nm.itrace : EXE : ITRACE ] ;
```

The `generated-targets` rule will be called with a single source target of type `EXE`. The call to the `virtual-target.traverse` will return all targets the executable depends on, and we further find files which are not produced from anything. The found targets are added to the sources.

The `run` method can be overriden to completely customize the way generator works. In particular, the conversion of sources to the desired types can be completely customized. Here's another real example. Tests for the Boost Python library usually consist of two parts: a Python program and a C++ file. The C++ file is compiled to Python extension which is loaded by the Python program. But in the likely case that both files have the same name, the created Python extension must be renamed. Otherwise, Python program will import itself, not the extension. Here's how it can be done:

```
rule run ( project name ? : property-set : sources * : multiple ? )
{
    local python ;
    for local s in $(sources)
    {
        if [ $(s).type ] = PY
        {
            python = $(s) ;
        }
    }

    local libs ;
    for local s in $(sources)
    {
        if [ type.is-derived [ $(s).type ] LIB ]
        {
            libs += $(s) ;
        }
    }

    local new-sources ;
    for local s in $(sources)
    {
        if [ type.is-derived [ $(s).type ] CPP ]
        {
            local name = [ $(s).name ] ;
            if $(name) = [ $(python).name ]
            {
                name = $(name)_ext ;
            }
            new-sources += [ generators.construct $(project) $(name) :
              PYTHON_EXTENSION : $(property-set) : $(s) $(libs) ] ;
        }
    }

    result = [ construct-result $(python) $(new-sources) : $(project) $(name)
                 : $(property-set) ] ;
}
```

First, we separate all source into python files, libraries and C++ sources. For each C++ source we create a separate Python extension by calling `generators.construct` and passing the C++ source and the libraries. At this point, we also change the extension's name, if necessary.

Features

Often, we need to control the options passed the invoked tools. This is done with features. Consider an example:

```
# Declare a new feature
import feature : feature ;
feature verbatim-options : : free ;

# Cause the value of the 'verbatim-options' feature to be
# available as 'OPTIONS' variable inside verbatim.inline-file
import toolset : flags ;
flags verbatim.inline-file OPTIONS <verbatim-options> ;

# Use the "OPTIONS" variable
actions inline-file
```

```
{
    "./inline-file.py" $(OPTIONS) $(<) $(>)
}
```

We first define a new feature. Then, the `flags` invocation says that whenever verbatin.inline-file action is run, the value of the `verbatim-options` feature will be added to the `OPTIONS` variable, an can be used inside the action body. You'd need to consult online help (--help) to find all the features of the `toolset.flags` rule.

Although you can define any set of features and interpret their values in any way, Boost.Build suggests the following coding standard for designing features.

Most features should have a fixed set of values, which is portable (tool neutral) across the class of tools they are designed to work with. The user does not have to adjust the values for a exact tool. For example, `<optimization>speed` has the same meaning for all C++ compilers and the user does not have to worry about the exact options which are passed to the compiler's command line.

Besides such portable features there are special 'raw' features which allow the user to pass any value to the command line parameters for a particular tool, if so desired. For example, the `<cxxflags>` feature allows to pass any command line options to a C++ compiler. The `<include>` feature allows to pass any value to the `-I` and the interpretation is tool-specific. (There an example of very smart usage of that feature). Of course one should always strive to use the portable features but these should still be provided as a backdoor just to make sure Boost.Build does not take away any control from the user.

Some of the reasons why portable features are better are:

- Since a portable feature have a fixed set of value, you will be able to build your project with two different settings of the feature. Boost.Build will automatically use two different directories for produced files. If you pass raw compiler options, Boost.Build assumes you know what you are doing, and would not care about what options are passed.
- Unlike "raw" features, you don't need to use specific compiler flags in Jamfile, and it will more likely work on other systems.

Steps for adding a feauture

Adding a feature requires three steps:

1. Declaring a feature. For that, the "feature.feature" rule is used. You should have to decide on the set of feature attributes:
    - if feature has several values, and significally affects build, make it "propagated", so that whole project is build with the same value by default
    - if a feature does not have a fixed list of values, it must be "free".
    - if feature is used to refer to a path, it must be "path".
    - if feature is used to refer to some target, it must be "dependency".
2. Converting the feature value into variable. To use feature in build action, it must be converted into a variable, accessible in build action. This is accomplished by "toolset.flags" rule.
3. Using the variable. The variable set in step 2 can be used in build action to form command parameters or files.

Another example

Here's an another example. Let's see how we can make a feature which refers to a target. For example, when linking dynamic libraries on windows, one sometimes needs to specify "DEF file", telling what functions should be exported. It would be nice to use this file like this:

```
        lib a : a.cpp : <def-file>a.def ;
```

Actually, this feature is already supported, but anyway...

1. Since the feature refers to a target, it must be "dependency".

   ```
   feature def-file : : free dependency ;
   ```

2. One of the toolsets which cares about DEF files is msvc. The following line should be added to it.

   ```
   flags msvc.link DEF_FILE <def-file> ;
   ```

3. Since the DEF_FILE variable is not used by the msvc.link action, we need to modify it to be:

   ```
   actions link bind DEF_FILE
   {
       $(.LD) .... /DEF:$(DEF_FILE) ....
   }
   ```

   Note the "bind DEF_FILE" part. It tells bjam that DEF_FILE refers to a file, otherwise the variable will contain internal target name, which is not likely to make sense for the linker.

   We've almost done, but should stop for a small workaround. Add the following code to msvc.jam

   ```
   rule link
   {
       DEPENDS $(<) : [ on $(<) return $(DEF_FILE) ] ;
   }
   ```

   This is needed to accomodate some bug in bjam, which hopefully will be fixed one day.

Variants and composite features.

Sometimes you want to create a shorcut for some set of features. For example, `release` is a value of the `variant` and is a shortcut for a set of features.
.

It is possible to define your build variants. For example:

```
variant crazy : <optimization>speed <inlining>off
                <debug-symbols>on <profiling>on ;
```

will define a new variant with the specified set of properties. You can also extend an existing variant:

```
variant super_release : release : <define>USE_ASM ;
```

In this case, `super_release` will expand to all properties specified by `release`, and the additional one you've specified.

You are not restricted to using the `variant` feature only. Here's example which defines a brand new feature:

```
feature parallelism : mpi fake none : composite link-incompatible ;
feature.compose <parallelism>mpi : <library>/mpi//mpi/<parallelism>none ;
feature.compose <parallelism>fake : <library>/mpi//fake/<parallelism>none ;
```

This will allow you to specify value of feature `parallelism`, which will expand to link to the necessary library.

Main target rules

The main target rule is what creates a top-level target, for example "exe" or "lib". It's quite likely that you'll want to declare your own and there are as many as three ways to do that.

The first is the simplest, but is sufficient in a number of cases. Just write a wrapper rule, which will redirect to any of the existing rules. For example, you have only one library per directory and want all cpp files in the directory to be compiled. You can achieve this effect with:

```
lib codegen : [ glob *.cpp ] ;
```

but what if you want to make it even simple. Then, you add the following definition to the project-root.jam file:

```
rule glib ( name : extra-sources * : requirements * )
{
    lib $(name) : [ glob *.cpp ] $(extra-sources) : $(requirements) ;
}
```

which would allow to reduce Jamfile to

```
glib codegen ;
```

The second approach is suitable when your target rule should just produce a target of specific type. Then, when declaring a type you should tell Boost.Build that a main target rule should be created. For example, if you create a module "obfuscate.jam" containing:

```
import type ;
type.register OBFUSCATED_CPP  : ocpp : : main ;

import generators ;
generators.register-standard obfuscate.file : CPP : OBFUSCATED_CPP ;
```

and import that module, you'll be able to use the rule "obfuscated-cpp" in Jamfiles, which will convert source to the OBFUSCATED_CPP type.

The remaining method is to declare your own main target class. The simplest example of this can be found in "build/alias.jam" file. The current V2 uses this method when transformations are relatively complex. However, we might deprecate this approach. If you find that you need to use it (that is, the first two approaches are not sufficient), please let us know by posting to the mailing list.

Toolset modules

If your extensions will be used only on one project, they can be placed in a separate `.jam` file which will be imported by your `project-root.jam`. If the extensions will be used on many projects, the users will thank you for a finishing touch.

The standard way to use a tool in Boost.Build is the `using` rule. To make it work, you module should provide an `init` rule. The rule will be called with the same parameters which were passed to the `using` rule. The set of allowed parameters is determined by you. For example, you can allow the user to specify paths, tool version, or tool options.

Here are some guidelines which help to make Boost.Build more consistent:

- The `init` rule should never fail. Even if user provided a wrong path, you should emit a warning and go on. Configuration may be shared between different machines, and wrong values on one machine can be OK on another.
- Prefer specifying command to be executed to specifying path. First of all, this gives more control: it's possible to specify

  ```
  /usr/bin/g++-snapshot
  time g++
  ```

  as the command. Second, while some tools have a logical "installation root", it better if user don't have to remember if a specific tool requires a full command or a path.
- Check for multiple initialization. A user can try to initialize the module several times. You need to check for this and decide what to do. Typically, unless you support several versions of a tool, duplicate initialization is a user error. If tool version can be specified during initialization, make sure the version is either always specified, or never specified (in which case the tool is initialied only once). For example, if you allow:

  ```
  using yfc ;
  using yfc : 3.3 ;
  using yfc : 3.4 ;
  ```

  Then it's not clear if the first initialization corresponds to version 3.3 of the tool, version 3.4 of the tool, or some other version. This can lead to building twice with the same version.
- If possible, the `init` must be callable with no parameters. In which case, it should try to autodetect all the necessary information, for example, by looking for a tool in `PATH` or in common installation locations. Often this is possible and allows the user to simply write:

  ```
  using yfc ;
  ```

- Consider using facilities in the `tools/common` module. You can take a look at how `tools/gcc.jam` uses that module in the `init` rule.

Detailed reference

General information

Initialization

bjam's first job upon startup is to load the Jam code which implements the build system. To do this, it searches for a file called "boost-build.jam", first in the invocation directory, then in its parent and so forth up to the filesystem root, and finally in the directories specified by the environment variable BOOST_BUILD_PATH. When found, the file is interpreted, and should specify the build system location by calling the boost-build rule:

```
rule boost-build ( location ? )
```

If location is a relative path, it is treated as relative to the directory of boost-build.jam. The directory specified by location and directories in BOOST_BUILD_PATH are then searched for a file called bootstrap.jam which is interpreted and is expected to bootstrap the build system. This arrangement allows the build system to work without any command-line or environment variable settings. For example, if the build system files were located in a directory "build-system/" at your project root, you might place a boost-build.jam at the project root containing:

```
boost-build build-system ;
```

In this case, running bjam anywhere in the project tree will automatically find the build system.

The default "bootstrap.jam", after loading some standard definitions, loads two files, which can be provided/customised by user: "site-config.jam" and "user-config.jam".

Locations where those files a search are summarized below:

Table 1. Search paths for configuration files

	site-config.jam	user-config.jam
Linux	/etc $HOME $BOOST_BUILD_PATH	$HOME $BOOST_BUILD_PATH
Windows	$SystemRoot $HOME $BOOST_BUILD_PATH	$HOME $BOOST_BUILD_PATH

Boost.Build comes with default versions of those files, which can serve as templates for customized versions.

Command line

The command line may contain:

- Jam options,
- Boost.Build options,
- Command line arguments

Command line arguments

Command line arguments specify targets and build request using the following rules.

- An argument which does not contain slashes or the "=" symbol is either a value of an implicit feature, or target to be built. It is taken to be value of a feature if appropriate feature exists. Otherwise, it is considered a target id. Special target name "clean" has the same effect as "--clean" option.
- An argument with either slashes or the "=" symbol specifies a number of build request elements. In the simplest form, it's just a set of properties, separated by slashes, which become a single build request element, for example:

  ```
  borland/<runtime-link>static
  ```

  More complex form is used to save typing. For example, instead of

  ```
  borland/runtime-link=static borland/runtime-link=dynamic
  ```

  one can use

  ```
  borland/runtime-link=static,dynamic
  ```

  Exactly, the conversion from argument to build request elements is performed by (1) splitting the argument at each slash, (2) converting each split part into a set of properties and (3) taking all possible combination of the property sets. Each split part should have the either the form

  `*feature-name*=*feature-value1*[","*feature-valueN*]*`

  or, in case of implicit feature

  `*feature-value1*[","*feature-valueN*;]*`

  and will be converted into property set

  ```
  <feature-name>feature-value1 .... <feature-name>feature-valueN
  ```

For example, the command line

```
target1 debug gcc/runtime-link=dynamic,static
```

would cause target called `target1` to be rebuilt in debug mode, except that for gcc, both dynamically and statically linked binaries would be created.

Command line options

All of the Boost.Build options start with the "--" prefix. They are described in the following table.

Table 2. Command line options

Option	Description
`--version`	Prints information on Boost.Build and Boost.Jam versions.
`--help`	Access to the online help system. This prints general information on how to use the help system with additional --help* options.
`--clean`	Removes everything instead of building. Unlike `clean` target in make, it is possible to clean only some targets.
`--debug`	Enables internal checks.
`--dump-projects`	Cause the project structure to be output.
`--no-error-backtrace`	Don't print backtrace on errors. Primary useful for testing.
`--ignore-config`	Do not load `site-config.jam` and `user-config.jam`

Writing Jamfiles

This section describes specific information about writing Jamfiles.

Generated headers

Usually, Boost.Build handles implicit dependendies completely automatically. For example, for C++ files, all `#include` statements are found and handled. The only aspect where user help might be needed is implicit dependency on generated files.

By default, Boost.Build handles such dependencies within one main target. For example, assume that main target "app" has two sources, "app.cpp" and "parser.y". The latter source is converted into "parser.c" and "parser.h". Then, if "app.cpp" includes "parser.h", Boost.Build will detect this dependency. Moreover, since "parser.h" will be generated into a build directory, the path to that directory will automatically added to include path.

Making this mechanism work across main target boundaries is possible, but imposes certain overhead. For that reason, if there's implicit dependency on files from other main targets, the `implicit-<dependency>` [link] feature must be used, for example:

```
lib parser : parser.y ;
exe app : app.cpp : <implicit-dependency>parser ;
```

The above example tells the build system that when scanning all sources of "app" for implicit-dependencies, it should consider targets from "parser" as potential dependencies.

Build process

The general overview of the build process was given in the user documentation. This section provides additional details, and some specific rules.

To recap, building a target with specific properties includes the following steps:

1. applying default build,
2. selecting the main target alternative to use,
3. determining "common" properties
4. building targets referred by the sources list and dependency properties
5. adding the usage requirements produces when building dependencies to the "common" properties
6. building the target using generators
7. computing the usage requirements to be returned

Alternative selection

When there are several alternatives, one of them must be selected. The process is as follows:

1. For each alternative *condition* is defined as the set of base properies in requirements. [Note: it might be better to specify the condition explicitly, as in conditional requirements].
2. An alternative is viable only if all properties in condition are present in build request.
3. If there's one viable alternative, it's choosen. Otherwise, an attempt is made to find one best alternative. An alternative a is better than another alternative b, iff set of properties in b's condition is strict subset of the set of properities of 'a's condition. If there's one viable alternative, which is better than all other, it's selected. Otherwise, an error is reported.

Determining common properties

The "common" properties is a somewhat artificial term. Those are the intermediate property set from which both the build request for dependencies and properties for building the target are derived.

Since default build and alternatives are already handled, we have only two inputs: build requests and requirements. Here are the rules about common properties.

1. Non-free feature can have only one value
2. A non-conditional property in requirement in always present in common properties.
3. A property in build request is present in common properties, unless (2) tells otherwise.
4. If either build request, or requirements (non-conditional or conditional) include an expandable property (either composite, or property with specified subfeature value), the behaviour is equivalent to explicitly adding all expanded properties to build request or requirements.

5. If requirements include a conditional property, and condiiton of this property is true in context of common properties, then the conditional property should be in common properties as well.

6. If no value for a feature is given by other rules here, it has default value in common properties.

Those rules are declarative, they don't specify how to compute the common properties. However, they provide enough information for the user. The important point is the handling of conditional requirements. The condition can be satisfied either by property in build request, by non-conditional requirements, or even by another conditional property. For example, the following example works as expected:

```
exe a : a.cpp
      : <toolset>gcc:<variant>release
        <variant>release:<define>FOO ;
```

Features and properties

A *feature* is a normalized (toolset-independent) aspect of a build configuration, such as whether inlining is enabled. Feature names may not contain the '>' character.

Each feature in a build configuration has one or more associated *value*s. Feature values for non-free features may not contain the '<', '`:`', or '=' characters. Feature values for free features may not contain the '<' character.

A *property* is a (feature,value) pair, expressed as <feature>value.

A *subfeature* is a feature which only exists in the presence of its parent feature, and whose identity can be derived (in the context of its parent) from its value. A subfeature's parent can never be another subfeature. Thus, features and their subfeatures form a two-level hierarchy.

A *value-string* for a feature **F** is a string of the form `-value-subvalue1-subvalue2`...`subvalueN`, where `value` is a legal value for **F** and `subvalue1`...`subvalueN` are legal values of some of **F**'s subfeatures. For example, the properties `<toolset>gcc <toolset-version>3.0.1` can be expressed more conscisely using a value-string, as `<toolset>gcc-3.0.1`.

A *property set* is a set of properties (i.e. a collection without duplicates), for instance: `<toolset>gcc <runtime-link>static`.

A *property path* is a property set whose elements have been joined into a single string separated by slashes. A property path representation of the previous example would be `run-<toolset>gcc/<time-link>static`.

A *build specification* is a property set which fully describes the set of features used to build a target.

Property Validity

For free features, all values are valid. For all other features, the valid values are explicitly specified, and the build system will report an error for the use of an invalid feature-value. Subproperty validity may be restricted so that certain values are valid only in the presence of certain other subproperties. For example, it is possible to specify that the `<gcc-target>mingw` property is only valid in the presence of `<gcc-version>2.95.2`.

Feature Attributes

Each feature has a collection of zero or more of the following attributes. Feature attributes are low-level descriptions of how the build system should interpret a feature's values when they appear in a build request. We also refer to the attributes of properties, so that an *incidental* property, for example, is one whose feature has the *incidental* attribute.

- *incidental*

  Incidental features are assumed not to affect build products at all. As a consequence, the build system may use the same file for targets whose build specification differs only in incidental features. A feature which controls a compiler's warning level is one example of a likely incidental feature.

  Non-incidental features are assumed to affect build products, so the files for targets whose build specification differs in non-incidental features are placed in different directories as described in "target paths" below. [where?]

- *propagated*

  Features of this kind are propagated to dependencies. That is, if a main target is built using a propagated property, the build systems attempts to use the same property when building any of its dependencies as part of that main target. For instance, when an optimized exectuable is requested, one usually wants it to be linked with optimized libraries. Thus, the `<optimization>` feature is propagated.

- *free*

  Most features have a finite set of allowed values, and can only take on a single value from that set in a given build specification. Free features, on the other hand, can have several values at a time and each value can be an arbitrary string. For example, it is possible to have several preprocessor symbols defined simultaneously:

  ```
  <define>NDEBUG=1 <define>HAS_CONFIG_H=1
  ```

- *optional*

  An optional feature is a feature which is not required to appear in a build specification. Every non-optional non-free feature has a default value which is used when a value for the feature is not otherwise specified, either in a target's requirements or in the user's build request. [A feature's default value is given by the first value listed in the feature's declaration. -- move this elsewhere - dwa]

- *symmetric*

  A symmetric feature's default value is not automatically included in build variants. Normally a feature only generates a subvariant directory when its value differs from the value specified by the build variant, leading to an assymmetric subvariant directory structure for certain values of the feature. A symmetric feature, when relevant to the toolset, always generates a corresponding subvariant directory.

- *path*

  The value of a path feature specifies a path. The path is treated as relative to the directory of Jamfile where path feature is used and is translated appropriately by the build system when the build is invoked from a different directory

- *implicit*

  Values of implicit features alone identify the feature. For example, a user is not required to write "<toolset>gcc", but can simply write "gcc". Implicit feature names also don't appear in variant paths, although the values do. Thus: bin/gcc/... as opposed to bin/toolset-gcc/.... There should typically be only a few such features, to avoid possible name clashes.

- *composite*

  Composite features actually correspond to groups of properties. For example, a build variant is a composite feature. When generating targets from a set of build properties, composite features are recursively expanded and *added* to the build property set, so rules can find them if neccessary. Non-composite non-free features override components of composite features in a build property set.

- *dependency*

  The value of dependency feature if a target reference. When used for building of a main target, the value of dependency feature is treated as additional dependency.

  For example, dependency features allow to state that library A depends on library B. As the result, whenever an application will link to A, it will also link to B. Specifying B as dependency of A is different from adding B to the sources of A.

Features which are neither free nor incidental are called *base* features.

Feature Declaration

The low-level feature declaration interface is the `feature` rule from the `feature` module:

```
rule feature ( name : allowed-values * : attributes * )
```

A feature's allowed-values may be extended with the `feature.extend` rule.

Build Variants

A build variant, or (simply variant) is a special kind of composite feature which automatically incorporates the default values of features that . Typically you'll want at least two separate variants: one for debugging, and one for your release code. [Volodya says: "Yea, we'd need to mention that it's a composite feature and describe how they are declared, in pacticular that default values of non-optional features are incorporated into build variant automagically. Also, do we wan't some variant inheritance/extension/templates. I don't remember how it works in V1, so can't document this for V2.". Will clean up soon -DWA]

Property refinement

When a target with certain properties is requested, and that target requires some set of properties, it is needed to find the set of properties to use for building. This process is called *property refinement* and is performed by these rules

1. If original properties and required properties are not link-compatible, refinement fails.
2. Each property in the required set is added to the original property set
3. If the original property set includes property with a different value of non free feature, that property is removed.

Conditional properties

Sometime it's desirable to apply certain requirements only for a specific combination of other properties. For example, one of compilers that you use issues a pointless warning that you want to suppress by passing a command line option to it. You would not want to pass that option to other compilers. Conditional properties allow you to do just that. Their syntax is:

```
        property ( "," property ) * ":" property
```

For example, the problem above would be solved by:

```
exe hello : hello.cpp : <toolset>yfc:<cxxflags>-disable-pointless-warning ;
```

Target identifiers and references

Target identifier is used to denote a target. The syntax is:

```
target-id -> (project-id | target-name | file-name )
              | (project-id | directory-name) "//" target-name
project-id -> path
target-name -> path
file-name -> path
directory-name -> path
```

This grammar allows some elements to be recognized as either

- project id (at this point, all project ids start with slash).
- name of target declared in current Jamfile (note that target names may include slash).
- a regular file, denoted by absolute name or name relative to project's sources location.

To determine the real meaning a check is made if project-id by the specified name exists, and then if main target of that name exists. For example, valid target ids might be:

```
a                                    -- target in current project
lib/b.cpp                            -- regular file
/boost/thread                        -- project "/boost/thread"
/home/ghost/build/lr_library//parser -- target in specific project
```

Rationale: Target is separated from project by special separator (not just slash), because:

- It emphasises that projects and targets are different things.
- It allows to have main target names with slashes.

Target reference is used to specify a source target, and may additionally specify desired properties for that target. It has this syntax:

```
target-reference -> target-id [ "/" requested-properties ]
```

```
requested-properties -> property-path
```

For example,

```
        exe compiler : compiler.cpp libs/cmdline/<optimization>space ;
```

would cause the version of `cmdline` library, optimized for space, to be linked in even if the `compiler` executable is build with optimization for speed.

Generators

Warning

The information is this section is likely to be outdated and misleading.

To construct a main target with given properties from sources, it is required to create a dependency graph for that main target, which will also include actions to be run. The algorithm for creating the dependency graph is described here.

The fundamental concept is *generator*. If encapsulates the notion of build tool and is capable to converting a set of input targets into a set of output targets, with some properties. Generator matches a build tool as closely as possible: it works only when the tool can work with requested properties (for example, msvc compiler can't work when requested toolset is gcc), and should produce exactly the same targets as the tool (for example, if Borland's linker produces additional files with debug information, generator should also).

Given a set of generators, the fundamental operation is to construct a target of a given type, with given properties, from a set of targets. That operation is performed by rule `generators.construct` and the used algorithm is described below.

Selecting and ranking viable generators

Each generator, in addition to target types that it can produce, have attribute that affects its applicability in particular sitiation. Those attributes are:

1. Required properties, which are properties absolutely necessary for the generator to work. For example, generator encapsulating the gcc compiler would have <toolset>gcc as required property.
2. Optional properties, which increase the generators suitability for a particual build.

Generator's required and optional properties may not include either free or incidental properties. (Allowing this would greatly complicate caching targets).

When trying to construct a target, the first step is to select all possible generators for the requested target type, which required properties are a subset of requested properties. Generators which were already selected up the call stack are excluded. In addition, if any composing generators were selected up the call stack, all other composing generators are ignored (TODO: define composing generators). The found generators are assigned a rank, which is the number of optional properties present in requested properties. Finally, generators with highest rank are selected for futher processing.

Running generators

When generators are selected, each is run to produce a list of created targets. This list might include targets which are not of requested types, because generators create the same targets as some tool, and tool's behaviour is fixed. (Note: should specify that in some cases we actually want extra targets). If generator fails, it returns an empty list. Generator is free to call 'construct' again, to convert sources to the types it can handle. It also can pass modified properties to 'construct'. However, a generator is not allowed to modify any propagated properties, otherwise when actually consuming properties we might discover that the set of propagated properties is different from what was used for building sources.

For all targets which are not of requested types, we try to convert them to requested type, using a second call to `construct`. This is done in order to support transformation sequences where single source file expands to several later. See this message for details.

Selecting dependency graph

After all generators are run, it is necessary to decide which of successfull invocation will be taken as final result. At the moment, this is not done. Instead, it is checked whether all successfull generator invocation returned the same target list. Error is issued otherwise.

Property adjustment

Because target location is determined by the build system, it is sometimes necessary to adjust properties, in order to not break actions. For example, if there's an action which generates a header, say "a_parser.h", and a source file "a.cpp" which includes that file, we must make everything work as if a_parser.h is generated in the same directory where it would be generated without any subvariants.

Correct property adjustment can be done only after all targets are created, so the approach taken is:

1. When dependency graph is constructed, each action can be assigned a rule for property adjustment.
2. When virtual target is actualized, that rule is run and return the final set of properties. At this stage it can use information of all created virtual targets.

In case of quoted includes, no adjustment can give 100% correct results. If target dirs are not changed by build system, quoted includes are searched in "." and then in include path, while angle includes are searched only in include path. When target dirs are changed, we'd want to make quoted includes to be search in "." then in additional dirs and then in the include path and make angle includes be searched in include path, probably with additional paths added at some position. Unless, include path already has "." as the first element, this is not possible. So, either generated headers should not be included with quotes, or first element of include path should be ".", which essentially erases the difference between quoted and angle includes. **Note:** the only way to get "." as include path into compiler command line is via verbatim compiler option. In all other case, Boost.Build will convert "." into directory where it occurs.

Transformations cache

Under certain conditions, an attempt is made to cache results of transformation search. First, the sources are replaced with targets with special name and the found target list is stored. Later, when properties, requested type, and source type are the same, the store target list is retrieved and cloned, with appropriate change in names.

Frequently Asked Questions

I'm getting "Duplicate name of actual target" error. What does it mean?

The most likely case is that you're trying to compile the same file twice, with almost the same, but differing properties. For example:

```
exe a : a.cpp : <include>/usr/local/include ;
exe b : a.cpp ;
```

The above snippet requires two different compilations of 'a.cpp', which differ only in 'include' property. Since the 'include' property is free, Boost.Build can't generate two objects files into different directories. On the other hand, it's dangerous to compile the file only once -- maybe you really want to compile with different includes.

To solve this issue, you need to decide if file should be compiled once or twice.

1. Two compile file only once, make sure that properties are the same:

   ```
   exe a : a.cpp : <include>/usr/local/include ;
   exe b : a.cpp : <include>/usr/local/include ;
   ```

2. If changing the properties is not desirable, for example if 'a' and 'b' target have other sources which need specific properties, separate 'a.cpp' into it's own target:

   ```
   obj a_obj : a.cpp : <include>/usr/local/include ;
   exe a : a_obj ;
   ```

3. To compile file twice, you can make the object file local to the main target:

   ```
         exe a : [ obj a_obj : a.cpp ] : <include>/usr/local/include ;
         exe b : [ obj a_obj : a.cpp ] ;
   ```

A good question is why Boost.Build can't use some of the above approaches automatically. The problem is that such magic would require additional implementation complexities and would only help in half of the cases, while in other half we'd be silently doing the wrong thing. It's simpler and safe to ask user to clarify his intention in such cases.

Accessing environment variables

Many users would like to use environment variables in Jamfiles, for example, to control location of external libraries. In many cases you better declare those external libraries in the site-config.jam file, as documented in the recipes section. However, if the users already have the environment variables set up, it's not convenient to ask them to set up site-config.jam files as well, and using environment variables might be reasonable.

In Boost.Build V2, each Jamfile is a separate namespace, and the variables defined in environment is imported into the global namespace. Therefore, to access environment variable from Jamfile, you'd need the following code:

```
import modules ;
local SOME_LIBRARY_PATH = [ modules.peek : SOME_LIBRARY_PATH ] ;
exe a : a.cpp : <include>$(SOME_LIBRARY_PATH) ;
```

How to control properties order?

For internal reasons, Boost.Build sorts all the properties alphabetically. This means that if you write:

```
exe a : a.cpp : <include>b <include>a ;
```

then the command line with first mention the "a" include directory, and then "b", even though they are specified in the opposite order. In most cases, the user doesn't care. But sometimes the order of includes, or other properties, is important. For example, if one uses both the C++ Boost library and the "boost-sandbox" (libraries in development), then include path for boost-sandbox must come first, because some headers may override ones in C++ Boost. For such cases, a special syntax is provided:

```
exe a : a.cpp : <include>a&&b ;
```

The && symbols separate values of an property, and specify that the order of the values should be preserved. You are advised to use this feature only when the order of properties really matters, and not as a convenient shortcut. Using it everywhere might negatively affect performance.

How to control the library order on Unix?

On the Unix-like operating systems, the order in which static libraries are specified when invoking the linker is important, because by default, the linker uses one pass though the libraries list. Passing the libraries in the incorrect order will lead to a link error. Further, this behaviour is often used to make one library override symbols from another. So, sometimes it's necessary to force specific order of libraries.

Boost.Build tries to automatically compute the right order. The primary rule is that if library a "uses" library b, then library a will appear on the command line before library b. Library a is considered to use b is b is present either in the sources of a or in its requirements. To explicitly specify the use relationship one can use the <use> feature. For example, both of the following lines will cause a to appear before b on the command line:

```
lib a : a.cpp b ;
lib a : a.cpp : <use>b ;
```

The same approach works for searched libraries, too:

```
lib z ;
lib png : : <use>z ;
exe viewer : viewer png z ;
```

Can I get output of external program as a variable in a Jamfile?

From time to time users ask how to run an external program and save the result in Jamfile variable, something like:

```
local gtk_includes = [ RUN_COMMAND gtk-config ] ;
```

Unfortunately, this is not possible at the moment. However, if the result of command invocation is to be used in a command to some tool, and you're working on Unix, the following workaround is possible.

```
 alias gtk+-2.0 : : : :
         <cflags>"`pkg-config --cflags gtk+-2.0`"
         <inkflags>"`pkg-config --libs gtk+-2.0`"
     ;
```

If you use the "gtk+-2.0" target in sources, then the properties specified above will be added to the build properties and eventually will appear in the command line. Unix command line shell processes the backticks quoting by running the tool and using its output -- which is what's desired in that case. Thanks to Daniel James for sharing this approach.

How to get the project-root location?

You might want to use the location of the project-root in your Jamfiles. To do it, you'd need to declare path constant in your project-root.jam:

```
path-constant TOP : . ;
```

After that, the `TOP` variable can be used in every Jamfile.

How to change compilation flags for one file?

If one file must be compiled with special options, you need to explicitly declare an `obj` target for that file and then use that target in your `exe` or `lib` target:

```
exe a : a.cpp b ;
obj b : b.cpp : <optimization>off ;
```

Of course you can use other properties, for example to specify specific compiler options:

```
exe a : a.cpp b ;
obj b : b.cpp : <cflags>-g ;
```

You can also use conditional properties for finer control:

```
exe a : a.cpp b ;
obj b : b.cpp : <variant>release:<optimization>off ;
```

Why are the `dll-path` and `hardcode-`

`dll-paths` properties useful?

(This entry is specific to Unix system.)Before answering the questions, let's recall a few points about shared libraries. Shared libraries can be used by several applications, or other libraries, without phisycally including the library in the application. This can greatly decrease the total size of applications. It's also possible to upgrade a shared library when the application is already installed. Finally, shared linking can be faster.

However, the shared library must be found when the application is started. The dynamic linker will search in a system-defined list of paths, load the library and resolve the symbols. Which means that you should either change the system-defined list, given by the `LD_LIBRARY_PATH` environment variable, or install the libraries to a system location. This can be inconvenient when developing, since the libraries are not yet ready to be installed, and cluttering system paths is undesirable. Luckily, on Unix there's another way.

An executable can include a list of additional library paths, which will be searched before system paths. This is excellent for development, because the build system knows the paths to all libraries and can include them in executables. That's done when the `hardcode-dll-paths` feature has the `true` value, which is the default. When the executables should be installed, the story is different.

Obviously, installed executable should not hardcode paths to your development tree. (The `stage` rule explicitly disables the `hardcode-dll-paths` feature for that reason.) However, you can use the `dll-path` feature to add explicit paths manually. For example:

```
stage installed : application : <dll-path>/usr/lib/snake
                                <location>/usr/bin ;
```

will allow the application to find libraries placed to `/usr/lib/snake`.

If you install libraries to a nonstandard location and add an explicit path, you get more control over libraries which will be used. A library of the same name in a system location will not be inadvertently used. If you install libraries to a system location and do not add any paths, the system administrator will have more control. Each library can be individually upgraded, and all applications will use the new library.

Which approach is best depends on your situation. If the libraries are relatively standalone and can be used by third party applications, they should be installed in the system location. If you have lots of libraries which can be used only by our application, it makes sense to install it to a nonstandard directory and add an explicit path, like the example above shows. Please also note that guidelines for different systems differ in this respect. The Debian guidelines prohibit any additional search paths, and Solaris guidelines suggest that they should always be used.

Targets in site-config.jam

It is desirable to declare standard libraries available on a given system. Putting target declaration in Jamfile is not really good, since locations of the libraries can vary. The solution is to put the following to site-config.jam.

```
import project ;
project.initialize $(__name__) ;
project site-config ;
lib zlib : : <name>z ;
```

The second line allows this module to act as project. The third line gives id to this project — it really has no location and cannot be used otherwise. The fourth line just declares a target. Now, one can write:

```
exe hello : hello.cpp /site-config//zlib ;
```

in any Jamfile.

Appendix A. Boost.Build v2 architecture

This document is work-in progress. Don't expect much from it yet.

Overview

The Boost.Build code is structured in four different components: "kernel", "util", "build" and "tools". The first two are relatively uninteresting, so we'll focus on the remaining pair. The "build" component provides classes necessary to declare targets, determine which properties should be used for their building, and for creating the dependency graph. The "tools" component provides user-visible functionality. It mostly allows to declare specific kind of main targets, and declare avaiable tools, which are then used when creating the dependency graph.

The build layer

The build layer has just four main parts -- abstract targets, virtual targets, generators and properties. The abstract targets, represented by the "abstract-target" class, correspond to main targets -- which, as you recall, can produce different files depending on properties. Virtual targets, represented by the 'virtual-target' class correspond to real files. The abstract-target class has a method 'generate', which is given a set of properties and produces virtual targets for those properties.

There are several classes derived from "abstract-target". The "main-target" class represents top-level main target, the "project-target" acts like container for all main targets, and "basic-target" class is a base class for all further target types.

Since each main target can have several alternatives, all top-level target objects are just containers, referring to "real" main target classes. The type is that container is "main-target". For example, given:

```
alias a ;
lib a : a.cpp : <toolset>gcc ;
```

we would have one-top level instance of "main-target-class", which will contain one instance of "alias-target-class" and one instance of "lib-target-class". The "generate" method of "main-target" decides which of the alternative should be used, and call "generate" on the corresponding instance.

Each alternative is a instance of a class derived from "basic-target". The "basic-target.generate" does several things that are always should be done:

- Determines what properties should be used for building the target. This includes looking at requested properties, requirements, and usage requirements of all sources.
- Builds all sources
- Computes the usage requirements which should be passes back.

For the real work of constructing virtual target, a new method "construct" is called.

The "construct" method can be implemented in any way by classes derived from "basic-target", but one specific derived class plays the central role -- "typed-target". That class holds the desired type of file to be produces, and calls the generators modules to do the job.

Generators are Boost.Build abstractions for a tool. For example, one can register a generator which converts target of type CPP into target of type OBJ. When run with on a virtual target with CPP type, the generator will construct the virtual target of type OBJ. The "generators" module implements an algorithm, which given a list of sources, the desired type and a list of properties, find all the generators which can perform the conversion.

The virtual targets which are produces by the main targets form a graph. Targets which are produces from other ones refer to an instance of "action" class, which in turn refers to action's sources, which can further refer to actions. The sources, which are not produces from anything, don't refer to any actions.

When all virtual targets are produced, they are "actualized". This means that the real file names are computed, and the commands that should be run are generated. This is done by "virtual-target.actualize" and "action.actualize" methods. The first is conceptually simple, while the second need additional explanation. The commands in bjam are generated in two-stage process. First, a rule with the appropriate name (for example "gcc.compile") is called and is given the names of targets. The rule sets some variables, like "OPTIONS". After that, the command string is taken, and variable are substitutes, so use of OPTIONS inside the command string become the real compile options.

Boost.Build added a third stage to simplify things. It's now possible to automatically convert properties to appropriate assignments to variables. For example, <debug-symbols>on would add "-g" to the OPTIONS variable, without requiring to manually add this logic to gcc.compile. This functionality is part of the "toolset" module.

When target paths are computed and the commands are set, Boost.Build just gives control to bjam, which controls the execution of commands.

The tools layer

Write me!

Targets

NOTE: THIS SECTION IS NOT EXPECTED TO BE READ! There are two user-visible kinds of targets in Boost.Build. First are "abstract" — they correspond to things declared by user, for example, projects and executable files. The primary thing about abstract target is that it's possible to request them to be build with a particular values of some properties. Each combination of properties may possible yield different set of real file, so abstract target do not have a direct correspondence with files.

File targets, on the contary, are associated with concrete files. Dependency graphs for abstract targets with specific properties are constructed from file targets. User has no was to create file targets, however it can specify rules that detect file type for sources, and also rules for transforming between file targets of different types. That information is used in constructing dependency graph, as desribed in the "next section". [link?] **Note:**File targets are not the same as targets in Jam sense; the latter are created from file targets at the latest possible moment. **Note:**"File target" is a proposed name for what we call virtual targets. It it more understandable by users, but has one problem: virtual targets can potentially be "phony", and not correspond to any file.

Dependency scanning

Dependency scanning is the process of finding implicit dependencies, like "#include" statements in C++. The requirements for right dependency scanning mechanism are:

- Support for different scanning algorithms. C++ and XML have quite different syntax for includes and rules for looking up included files.
- Ability to scan the same file several times. For example, single C++ file can be compiled with different include paths.
- Proper detection of dependencies on generated files.
- Proper detection of dependencies from generated file.

Support for different scanning algorithms

Different scanning algorithm are encapsulated by objects called "scanners". Please see the documentation for "scanner" module for more details.

Ability to scan the same file several times

As said above, it's possible to compile a C++ file twice, with different include paths. Therefore, include dependencies for those compilations can be different. The problem is that bjam does not allow several scans of the same target.

The solution in Boost.Build is straigtforward. When a virtual target is converted to bjam target (via `virtual-target.actualize` method), we specify the scanner object to be used. The actualize method will create different bjam targets for different scanners.

All targets with specific scanner are made dependent on target without scanner, which target is always created. This is done in case the target is updated. The updating action will be associated with target without scanner, but if sources for that action are touched, all targets — with scanner and without should be considered outdated.

For example, assume that "a.cpp" is compiled by two compilers with different include path. It's also copied into some install location. In turn, it's produced from "a.verbatim". The dependency graph will look like:

```
a.o (<toolset>gcc)  <--(compile)-- a.cpp (scanner1) ----+
a.o (<toolset>msvc) <--(compile)-- a.cpp (scanner2) ----|
a.cpp (installed copy)    <--(copy) ---------------------- a.cpp (no scanner)
                                                                 ^
                                                                 |
                       a.verbose --------------------------------+
```

Proper detection of dependencies on generated files.

This requirement breaks down to the following ones.

1. If when compiling "a.cpp" there's include of "a.h", the "dir" directory is in include path, and a target called "a.h" will be generated to "dir", then bjam should discover the include, and create "a.h" before compiling "a.cpp".
2. Since almost always Boost.Build generates targets to a "bin" directory, it should be supported as well. I.e. in the scanario above, Jamfile in "dir" might create a main target, which generates "a.h". The file will be generated to "dir/bin" directory, but we still have to recornize the dependency.

The first requirement means that when determining what "a.h" means, when found in "a.cpp", we have to iterate over all directories in include paths, checking for each one:

1. If there's file "a.h" in that directory, or
2. If there's a target called "a.h", which will be generated to that directory.

Classic Jam has built-in facilities for point (1) above, but that's not enough. It's hard to implement the right semantic without builtin support. For example, we could try to check if there's targer called "a.h" somewhere in dependency graph, and add a dependency to it. The problem is that without search in include path, the semantic may be incorrect. For example, one can have an action which generated some "dummy" header, for system which don't have the native one. Naturally, we don't want to depend on that generated header on platforms where native one is included.

There are two design choices for builtin support. Suppose we have files a.cpp and b.cpp, and each one includes header.h, generated by some action. Dependency graph created by classic jam would look like:

```
a.cpp -----> <scanner1>header.h  [search path: d1, d2, d3]


                  <d2>header.h  --------> header.y
                  [generated in d2]

b.cpp -----> <scanner2>header.h [ search path: d1, d2, d4]
```

In this case, Jam thinks all header.h target are not realated. The right dependency graph might be:

```
a.cpp ----
          \
           \
            >---->  <d2>header.h  --------> header.y
           /       [generated in d2]
          /
b.cpp ----
```

or

```
a.cpp -----> <scanner1>header.h  [search path: d1, d2, d3]
                          |
                       (includes)
                          V
                  <d2>header.h  --------> header.y
                  [generated in d2]
                          ^
                      (includes)
                          |
b.cpp -----> <scanner2>header.h [ search path: d1, d2, d4]
```

The first alternative was used for some time. The problem however is: what include paths should be used when scanning header.h? The second alternative was suggested by Matt Armstrong. It has similiar effect: add targets which depend on <scanner1>header.h will also depend on <d2>header.h. But now we have two different target with two different scanners, and those targets can be scanned independently. The problem of first alternative is avoided, so the second alternative is implemented now.

The second sub-requirements is that targets generated to "bin" directory are handled as well. Boost.Build implements semi-automatic approach. When compiling C++ files the process is:

1. The main target to which compiled file belongs is found.
2. All other main targets that the found one depends on are found. Those include main target which are used as sources, or present as values of "dependency" features.
3. All directories where files belonging to those main target will be generated are added to the include path.

After this is done, dependencies are found by the approach explained previously.

Note that if a target uses generated headers from other main target, that main target should be explicitly specified as dependency property. It would be better to lift this requirement, but it seems not very problematic in practice.

For target types other than C++, adding of include paths must be implemented anew.

Proper detection of dependencies from generated files

Suppose file "a.cpp" includes "a.h" and both are generated by some action. Note that classic jam has two stages. In first stage dependency graph graph is build and actions which should be run are determined. In second stage the actions are executed. Initially, neither file exists, so the include is not found. As the result, jam might attempt to compile a.cpp before creating a.h, and compilation will fail.

The solution in Boost.Jam is to perform additional dependency scans after targets are updated. This break separation between build stages in jam — which some people consider a good thing — but I'm not aware of any better solution.

In order to understand the rest of this section, you better read some details about jam dependency scanning, available at this link.

Whenever a target is updated, Boost.Jam rescans it for includes. Consider this graph, created before any actions are run.

```
A -------> C ----> C.pro
     /
B --/         C-includes   ---> D
```

Both A and B have dependency on C and C-includes (the latter dependency is not shown). Say during building we've tried to create A, then tried to create C and successfully created C.

In that case, the set of includes in C might well have changed. We do not bother to detect precisely which includes were added or removed. Instead we create another internal node C-includes-2. Then we determine what actions should be run to update the target. In fact this mean that we perform logic of first stage while already executing stage.

After actions for C-includes-2 are determined, we add C-includes-2 to the list of A's dependents, and stage 2 proceeds as usual. Unfortunately, we can't do the same with target B, since when it's not visited, C target does not know B depends on it. So, we add a flag to C which tells and it was rescanned. When visiting B target, the flag is notices and C-includes-2 will be added to the list of B's dependencies.

Note also that internal nodes are sometimes updated too. Consider this dependency graph:

```
a.o ---> a.cpp
            a.cpp-includes -->  a.h (scanned)
                                   a.h-includes ------> a.h (generated)
```

```
                                                           |
                                                           |
        a.pro <-------------------------------------------+
```

Here, out handling of generated headers come into play. Say that a.h exists but is out of date with respect to "a.pro", then "a.h (generated)" and "a.h-includes" will be marking for updating, but "a.h (scanned)" won't be marked. We have to rescan "a.h" file after it's created, but since "a.h (generated)" has no scanner associated with it, it's only possible to rescan "a.h" after "a.h-includes" target was updated.

Tbe above consideration lead to decision that we'll rescan a target whenever it's updated, no matter if this target is internal or not.

Warning

The remainder of this document is not indended to be read at all. This will be rearranged in future.

File targets

As described above, file targets corresponds to files that Boost.Build manages. User's may be concerned about file targets in three ways: when declaring file target types, when declaring transformations between types, and when determining where file target will be placed. File targets can also be connected with actions, that determine how the target is created. Both file targets and actions are implemented in the `virtual-target` module.

Types

A file target can be given a file, which determines what transformations can be applied to the file. The `type.register` rule declares new types. File type can also be assigned a scanner, which is used to find implicit dependencies. See "dependency scanning" [link?] below.

Target paths

To distinguish targets build with different properties, they are put in different directories. Rules for determining target paths are given below:

1. All targets are placed under directory corresponding to the project where they are defined.
2. Each non free, non incidental property cause an additional element to be added to the target path. That element has the form `<feature-name>-<feature-value>` for ordinary features and `<feature-value>` for implicit ones. [Note about composite features].
3. If the set of free, non incidental properties is different from the set of free, non incidental properties for the project in which the main target that uses the target is defined, a part of the form `main_target-<name>` is added to the target path. **Note:**It would be nice to completely track free features also, but this appears to be complex and not extremely needed.

For example, we might have these paths:

```
debug/optimization-off
debug/main-target-a
```