**Concepts and terminology**

### **Workspace**

A workspace is a directory on your filesystem that contains the source files for the software you want to build, Each workspace directory has a text file named WORKSPACE which may be empty, or may contain references to [external dependencies](https://docs.bazel.build/versions/4.0.0/external.html) required to build the outputs. Directories containing a file called WORKSPACE are considered the root of a workspace. Bazel also supports WORKSPACE.bazel file as an alias of WORKSPACE file. If both files exist, WORKSPACE.bazel will take the priority.

### **Repositories**

Code is organized in repositories. The directory containing the WORKSPACE file is the root of the main repository, also called @. Other, (external) repositories are defined in the WORKSPACE file using workspace rules. As external repositories are repositories themselves, they often contain a WORKSPACE file as well. However, these additional WORKSPACE files are ignored by Bazel.

### **Packages**

The primary unit of code organization in a repository is the *package*. A package is a collection of related files and a specification of the dependencies among them. A package is defined as a directory containing a file named BUILD or BUILD.bazel, residing beneath the top-level directory in the workspace. A package includes all files in its directory, plus all subdirectories beneath it, except those which themselves contain a BUILD file.

src/my/app/BUILD

src/my/app/app.cc

src/my/app/data/input.txt

src/my/app/tests/BUILD

src/my/app/tests/test.cc

Here my/app & my/app/tests are packages while my/app/data is a subdirectory of my/app

### **Targets**

The elements of a package are called *targets*. Most targets are one of two principal kinds, *files* and *rules*. Additionally, there is another kind of target, [package groups](https://docs.bazel.build/versions/4.0.0/be/functions.html#package_group), but they are far less numerous.

Files are further divided into two kinds. *Source files* are usually written by the efforts of people, and checked in to the repository. *Generated files*, sometimes called derived files, are not checked in, but are generated by the build tool from source files according to specific rules.

The second kind of target is the *rule*. A rule specifies the relationship between a set of input and a set of output files, including the necessary steps to derive the outputs from the inputs. The outputs of a rule are always generated files. The inputs to a rule may be source files, but they may be generated files also; consequently, outputs of one rule may be the inputs to another.

The inputs to a rule may also include *other rules*. a C++ library rule A might have another C++ library rule B for an input. The effect of this dependency is that B's header files are available to A during compilation, B's symbols are available to A during linking, and B's runtime data is available to A during execution.

An invariant of all rules is that the files generated by a rule always belong to the same package as the rule itself; it is not possible to generate files into another package.

### **Labels**

All targets belong to exactly one package. The name of a target is called its label, and a typical label in canonical form looks like this:

@myrepo//my/app/main:app\_binary

In the typical case that a label refers to the same repository it occurs in, the repository name may be left out. So, inside @myrepo this label is usually written as

//my/app/main:app\_binary

Each label has two parts, a package name (my/app/main) and a target name (app\_binary). Every label uniquely identifies a target.

Labels sometimes appear in other forms; when the colon is omitted, the target name is assumed to be the same as the last component of the package name, so these two labels are equivalent:

//my/app

//my/app:app

my/app is the package containing the label//my/app

within the BUILD file for package my/app (i.e. //my/app:BUILD), the following "relative" labels are all equivalent:

//my/app:app

//my/app

:app

app

### **Lexical specification of a label**

#### **Target names, //...:**target-name

Target names must be composed entirely of characters drawn from the set a–z, A–Z, 0–9, and the punctuation symbols !%-@^\_` "#$&'()\*-+,;<=>?[]{|}~/.

to refer to files in other packages; use //packagename:filename

Filenames must be relative pathnames in normal form, which means they must neither start nor end with a slash (e.g. /foo and foo/ are forbidden) nor contain multiple consecutive slashes as path separators (e.g. foo//bar). Similarly, up-level references (..) and current-directory references (./) are forbidden.

#### **Package names, //**package-name**:...**

The name of a package is the name of the directory containing its BUILD file, relative to the top-level directory of the source tree. For example: my/app. Package names must be composed entirely of characters drawn from the set A-Z, a–z, 0–9, '/', '-', '.', and '\_', and cannot start with a slash.

### **Rules**

Every rule has a name, specified by the name attribute, of type string. The name must be a syntactically valid target name.

cc\_binary(

name = "my\_app",

srcs = ["my\_app.cc"],

deps = [

"//absl/base",

"//absl/strings",

],

)

Every rule has a set of *attributes*; the applicable attributes for a given rule, and the significance and semantics of each attribute are a function of the rule's class. Each attribute has a name and a type. Some of the common types an attribute can have are integer, label, list of labels, string, list of strings, output label, list of output labels. Not all attributes need to be specified in every rule. Attributes thus form a dictionary from keys (names) to optional, typed values.

The srcs attribute present in many rules has type "list of labels"; its value, if present, is a list of labels, each being the name of a target that is an input to this rule.

The outs attribute present in many rules has type "list of output labels"; the outputs of a rule belong to the same package as the rule itself, output labels cannot include a package component; they must be in one of the "relative" forms. a rule *is depended on by* its outs.

## **BUILD files**

Every package contains a BUILD file, which is a short program. They are interpreted as a sequential list of statements. most BUILD files consist only of declarations of build rules, and the relative order of these statements is immaterial; all that matters is which rules were declared, and with what values, by the time package evaluation completes.

### **Loading an extension**

Bazel extensions are files ending in .bzl. Use the load statement to import a symbol from an extension.

load("//foo/bar:file.bzl", "some\_library")

This code will load the file foo/bar/file.bzl and add the some\_library symbol to the environment. This can be used to load new rules, functions or constants

## **Types of build rule**

The majority of build rules come in families, grouped together by language. For example, cc\_binary, cc\_library and cc\_test are the build rules for C++ binaries, libraries, and tests, respectively.

\*\_binary rules build executable programs in a given language. After a build, the executable will reside in the build tool's binary output tree at the corresponding name for the rule's label.

\*\_test rules are a specialization of a \*\_binary rule, used for automated testing. Tests are simply programs that return zero on success.

\*\_library rules specify separately-compiled modules in the given programming language. Libraries can depend on other libraries, and binaries and tests can depend on libraries, with the expected separate-compilation behavior.

## **Dependencies**

A target A *depends upon* a target B if B is needed by A at build or execution time. The *depends upon* relation induces a [Directed Acyclic Graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph) (DAG) over targets, and we call this a dependency graph.

# C / C++ Rules

## **cc\_binary**

cc\_binary([name](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.name), [deps](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.deps), [srcs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.srcs), [data](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.data), [additional\_linker\_inputs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.additional_linker_inputs), [args](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#binary.args), [compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.compatible_with), [copts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts), [defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.defines), [deprecation](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.deprecation), [distribs](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.distribs), [env](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#binary.env), [exec\_compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.exec_compatible_with), [exec\_properties](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.exec_properties), [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.features), [includes](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.includes), [licenses](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.licenses), [linkopts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.linkopts), [linkshared](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.linkshared), [linkstatic](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.linkstatic), [local\_defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.local_defines), [malloc](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.malloc), [nocopts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.nocopts), [output\_licenses](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#binary.output_licenses), [restricted\_to](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.restricted_to), [stamp](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.stamp), [tags](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.tags), [target\_compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.target_compatible_with), [testonly](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.testonly), [toolchains](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.toolchains), [visibility](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.visibility), [win\_def\_file](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.win_def_file))

| **Attributes** | |
| --- | --- |
| name | [Name](https://docs.bazel.build/versions/4.0.0/build-ref.html#name); required  A unique name for this target. |
| deps | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of other libraries to be linked in to the binary target.  These can be cc\_library or objc\_library targets. |
| srcs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of C and C++ files that are processed to create the target. These are C/C++ source and header files, either non-generated (normal source code) or generated.  All .cc, .c, and .cpp files will be compiled. These might be generated files: if a named file is in the outs of some other rule, this rule will automatically depend on that other rule.  A .h file will not be compiled, but will be available for inclusion by sources in this rule. Both .cc and .h files can directly include headers listed in these srcs or in the hdrs of any rule listed in the deps argument.  All #included files must be mentioned in the srcs attribute of this rule, or in the hdrs attribute of referenced cc\_library()s. The recommended style is for headers associated with a library to be listed in that library's hdrs attribute, and any remaining headers associated with this rule's sources to be listed in srcs. See ["Header inclusion checking"](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#hdrs) for a more detailed description.  If a rule's name is in the srcs, then this rule automatically depends on that one. If the named rule's outs are C or C++ source files, they are compiled into this rule; if they are library files, they are linked in.  Permitted srcs file types:   * C and C++ source files: .c, .cc, .cpp, .cxx, .c++, .C * C and C++ header files: .h, .hh, .hpp, .hxx, .inc, .inl, .H * Assembler with C preprocessor: .S * Archive: .a, .pic.a * "Always link" library: .lo, .pic.lo * Shared library, versioned or unversioned: .so, .so.*version* * Object file: .o, .pic.o   ...and any rules that produce those files. Different extensions denote different programming languages in accordance with gcc convention. |
| additional\_linker\_inputs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  Pass these files to the C++ linker command.  For example, compiled Windows .res files can be provided here to be embedded in the binary target. |
| copts | List of strings; optional  Add these options to the C++ compilation command. Subject to ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization).  Each string in this attribute is added in the given order to COPTS before compiling the binary target. The flags take effect only for compiling this target, not its dependencies, so be careful about header files included elsewhere. All paths should be relative to the workspace, not to the current package.  If the package declares the [feature](https://docs.bazel.build/versions/4.0.0/be/functions.html#package.features) no\_copts\_tokenization, Bourne shell tokenization applies only to strings that consist of a single "Make" variable. |
| defines | List of strings; optional  List of defines to add to the compile line. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization). Each string, which must consist of a single Bourne shell token, is prepended with -D and added to the compile command line to this target, as well as to every rule that depends on it. Be very careful, since this may have far-reaching effects. When in doubt, add define values to [local\_defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.local_defines) instead. |
| includes | List of strings; optional  List of include dirs to be added to the compile line.  Subject to ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution. Each string is prepended with -isystem and added to COPTS. Unlike [COPTS](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts), these flags are added for this rule and every rule that depends on it. (Note: not the rules it depends upon!) Be very careful, since this may have far-reaching effects. When in doubt, add "-I" flags to [COPTS](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts) instead.  Headers must be added to srcs or hdrs, otherwise they will not be available to dependent rules when compilation is sandboxed (the default). |
| linkopts | List of strings; optional  Add these flags to the C++ linker command. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution, [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization) and [label expansion](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#label-expansion). Each string in this attribute is added to LINKOPTS before linking the binary target.  Each element of this list that does not start with $ or - is assumed to be the label of a target in deps. The list of files generated by that target is appended to the linker options. An error is reported if the label is invalid, or is not declared in deps. |
| linkshared | Boolean; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes); default is False  Create a shared library. To enable this attribute, include linkshared=True in your rule. By default this option is off. If you enable it, you must name your binary lib*foo*.so (or whatever is the naming convention of libraries on the target platform) for some sensible value of *foo*.  The presence of this flag means that linking occurs with the -shared flag to gcc, and the resulting shared library is suitable for loading into for example a Java program. However, for build purposes it will never be linked into the dependent binary, as it is assumed that shared libraries built with a [cc\_binary](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary) rule are only loaded manually by other programs, so it should not be considered a substitute for the [cc\_library](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library) rule. For sake of scalability we recommend avoiding this approach altogether and simply letting java\_library depend on cc\_library rules instead.  If you specify both linkopts=['-static'] and linkshared=True, you get a single completely self-contained unit. If you specify both linkstatic=1 and linkshared=True, you get a single, mostly self-contained unit. |
| linkstatic | Boolean; optional; default is True  For [cc\_binary](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary) and [cc\_test](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test): link the binary in static mode. For cc\_library.linkstatic: see below.  By default this option is on for cc\_binary and off for the rest.  If enabled and this is a binary or test, this option tells the build tool to link in .a's instead of .so's for user libraries whenever possible. Some system libraries may still be linked dynamically, as are libraries for which there is no static library. So the resulting executable will still be dynamically linked, hence only *mostly* static.  There are really three different ways to link an executable:   * STATIC with fully\_static\_link feature, in which everything is linked statically; e.g. "gcc -static foo.o libbar.a libbaz.a -lm". This mode is enabled by specifying fully\_static\_link in the [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#features) attribute. * STATIC, in which all user libraries are linked statically (if a static version is available), but where system libraries (excluding C/C++ runtime libraries) are linked dynamically, e.g. "gcc foo.o libfoo.a libbaz.a -lm". This mode is enabled by specifying linkstatic=True. * DYNAMIC, in which all libraries are linked dynamically (if a dynamic version is available), e.g. "gcc foo.o libfoo.so libbaz.so -lm". This mode is enabled by specifying linkstatic=False.   The linkstatic attribute has a different meaning if used on a [cc\_library()](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library) rule. For a C++ library, linkstatic=True indicates that only static linking is allowed, so no .so will be produced. linkstatic=False does not prevent static libraries from being created. The attribute is meant to control the creation of dynamic libraries.  If linkstatic=False, then the build tool will create symlinks to depended-upon shared libraries in the \*.runfiles area. |
| local\_defines | List of strings; optional  List of defines to add to the compile line. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization). Each string, which must consist of a single Bourne shell token, is prepended with -D and added to the compile command line for this target, but not to its dependents. |
| malloc | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; default is @bazel\_tools//tools/cpp:malloc  Override the default dependency on malloc.  By default, C++ binaries are linked against //tools/cpp:malloc, which is an empty library so the binary ends up using libc malloc. This label must refer to a cc\_library. If compilation is for a non-C++ rule, this option has no effect. The value of this attribute is ignored if linkshared=True is specified. |
| nocopts | String; optional  Remove matching options from the C++ compilation command. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution. The value of this attribute is interpreted as a regular expression. Any preexisting COPTS that match this regular expression (including values explicitly specified in the rule's [copts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts) attribute) will be removed from COPTS for purposes of compiling this rule. This attribute should rarely be needed. |
| stamp | Integer; optional; default is -1  Enable link stamping. Whether to encode build information into the binary. Possible values:   * stamp = 1: Stamp the build information into the binary. Stamped binaries are only rebuilt when their dependencies change. Use this if there are tests that depend on the build information. * stamp = 0: Always replace build information by constant values. This gives good build result caching. * stamp = -1: Embedding of build information is controlled by the [--[no]stamp](https://docs.bazel.build/versions/4.0.0/user-manual.html#flag--stamp) flag. |
| win\_def\_file | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The Windows DEF file to be passed to linker.  This attribute should only be used when Windows is the target platform. It can be used to [export symbols](https://msdn.microsoft.com/en-us/library/d91k01sh.aspx) during linking a shared library. |

## **cc\_import**

cc\_import([name](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.name), [data](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.data), [hdrs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.hdrs), [alwayslink](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.alwayslink), [compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.compatible_with), [deprecation](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.deprecation), [distribs](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.distribs), [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.features), [interface\_library](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.interface_library), [licenses](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.licenses), [restricted\_to](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.restricted_to), [shared\_library](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.shared_library), [static\_library](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.static_library), [system\_provided](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_import.system_provided), [tags](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.tags), [testonly](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.testonly), [visibility](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.visibility))

cc\_import rules allows users to import precompiled C/C++ libraries.

The following are the typical use cases:  
1. Linking a static library

cc\_import(

name = "mylib",

hdrs = ["mylib.h"],

static\_library = "libmylib.a",

# If alwayslink is turned on,

# libmylib.a will be forcely linked into any binary that depends on it.

# alwayslink = 1,

)

2. Linking a shared library (Unix)

cc\_import(

name = "mylib",

hdrs = ["mylib.h"],

shared\_library = "libmylib.so",

)

3. Linking a shared library with interface library (Windows)

cc\_import(

name = "mylib",

hdrs = ["mylib.h"],

# mylib.lib is a import library for mylib.dll which will be passed to linker

interface\_library = "mylib.lib",

# mylib.dll will be available for runtime

shared\_library = "mylib.dll",

)

4. Linking a shared library with system\_provided=True (Windows)

cc\_import(

name = "mylib",

hdrs = ["mylib.h"],

# mylib.lib is an import library for mylib.dll which will be passed to linker

interface\_library = "mylib.lib",

# mylib.dll is provided by system environment, for example it can be found in PATH.

# This indicates that Bazel is not responsible for making mylib.dll available.

system\_provided = 1,

)

5. Linking to static or shared library  
On Unix:

cc\_import(

name = "mylib",

hdrs = ["mylib.h"],

static\_library = "libmylib.a",

shared\_library = "libmylib.so",

)

# first will link to libmylib.a

cc\_binary(

name = "first",

srcs = ["first.cc"],

deps = [":mylib"],

linkstatic = 1, # default value

)

# second will link to libmylib.so

cc\_binary(

name = "second",

srcs = ["second.cc"],

deps = [":mylib"],

linkstatic = 0,

)

On Windows:

cc\_import(

name = "mylib",

hdrs = ["mylib.h"],

static\_library = "libmylib.lib", # A normal static library

interface\_library = "mylib.lib", # An import library for mylib.dll

shared\_library = "mylib.dll",

)

# first will link to libmylib.lib

cc\_binary(

name = "first",

srcs = ["first.cc"],

deps = [":mylib"],

linkstatic = 1, # default value

)

# second will link to mylib.dll through mylib.lib

cc\_binary(

name = "second",

srcs = ["second.cc"],

deps = [":mylib"],

linkstatic = 0,

)

| **Attributes** | |
| --- | --- |
| name | [Name](https://docs.bazel.build/versions/4.0.0/build-ref.html#name); required  A unique name for this target. |
| hdrs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of header files published by this precompiled library to be directly included by sources in dependent rules. |
| alwayslink | Boolean; optional; default is False  If 1, any binary that depends (directly or indirectly) on this C++ precompiled library will link in all the object files archived in the static library, even if some contain no symbols referenced by the binary. This is useful if your code isn't explicitly called by code in the binary, e.g., if your code registers to receive some callback provided by some service.  If alwayslink doesn't work with VS 2017 on Windows, that is due to a [known issue](https://github.com/bazelbuild/bazel/issues/3949), please upgrade your VS 2017 to the latest version. |
| interface\_library | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  A single interface library for linking the shared library.  Permitted file types: .ifso, .tbd, .lib, .so or .dylib |
| shared\_library | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  A single precompiled shared library. Bazel ensures it is available to the binary that depends on it during runtime.  Permitted file types: .so, .dll or .dylib |
| static\_library | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  A single precompiled static library.  Permitted file types: .a, .pic.a or .lib |
| system\_provided | Boolean; optional; default is False  If 1, it indicates the shared library required at runtime is provided by the system. In this case, interface\_library should be specified and shared\_library should be empty. |

## **cc\_library**

cc\_library([name](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.name), [deps](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.deps), [srcs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.srcs), [data](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.data), [hdrs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.hdrs), [alwayslink](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.alwayslink), [compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.compatible_with), [copts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.copts), [defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.defines), [deprecation](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.deprecation), [distribs](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.distribs), [exec\_compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.exec_compatible_with), [exec\_properties](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.exec_properties), [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.features), [include\_prefix](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.include_prefix), [includes](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.includes), [licenses](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.licenses), [linkopts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.linkopts), [linkstamp](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.linkstamp), [linkstatic](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.linkstatic), [local\_defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.local_defines), [nocopts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.nocopts), [restricted\_to](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.restricted_to), [strip\_include\_prefix](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.strip_include_prefix), [tags](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.tags), [target\_compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.target_compatible_with), [testonly](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.testonly), [textual\_hdrs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.textual_hdrs), [toolchains](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.toolchains), [visibility](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.visibility), [win\_def\_file](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library.win_def_file))

cc\_binary(

name = "foo",

srcs = [

"foo.cc",

"foo.h",

],

deps = [":bar"],

)

cc\_library(

name = "bar",

srcs = [

"bar.cc",

"bar-impl.h",

],

hdrs = ["bar.h"],

deps = [":baz"],

)

cc\_library(

name = "baz",

srcs = [

"baz.cc",

"baz-impl.h",

],

hdrs = ["baz.h"],

)

The allowed direct inclusions in this example are listed in the table below. For example foo.cc is allowed to directly include foo.h and bar.h, but not baz.h.

| **Including file** | **Allowed inclusions** |
| --- | --- |
| foo.h | bar.h |
| foo.cc | foo.h bar.h |
| bar.h | bar-impl.h baz.h |
| bar-impl.h | bar.h baz.h |
| bar.cc | bar.h bar-impl.h baz.h |
| baz.h | baz-impl.h |
| baz-impl.h | baz.h |
| baz.cc | baz.h baz-impl.h |
| **Attributes** | |
| name | [Name](https://docs.bazel.build/versions/4.0.0/build-ref.html#name); required  A unique name for this target. |
| deps | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of other libraries to be linked in to the binary target.  These can be cc\_library or objc\_library targets. |
| srcs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of C and C++ files that are processed to create the target. These are C/C++ source and header files, either non-generated (normal source code) or generated.  All .cc, .c, and .cpp files will be compiled. These might be generated files: if a named file is in the outs of some other rule, this rule will automatically depend on that other rule.  A .h file will not be compiled, but will be available for inclusion by sources in this rule. Both .cc and .h files can directly include headers listed in these srcs or in the hdrs of any rule listed in the deps argument.  All #included files must be mentioned in the srcs attribute of this rule, or in the hdrs attribute of referenced cc\_library()s. The recommended style is for headers associated with a library to be listed in that library's hdrs attribute, and any remaining headers associated with this rule's sources to be listed in srcs. See ["Header inclusion checking"](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#hdrs) for a more detailed description.  If a rule's name is in the srcs, then this rule automatically depends on that one. If the named rule's outs are C or C++ source files, they are compiled into this rule; if they are library files, they are linked in.  Permitted srcs file types:   * C and C++ source files: .c, .cc, .cpp, .cxx, .c++, .C * C and C++ header files: .h, .hh, .hpp, .hxx, .inc, .inl, .H * Assembler with C preprocessor: .S * Archive: .a, .pic.a * "Always link" library: .lo, .pic.lo * Shared library, versioned or unversioned: .so, .so.*version* * Object file: .o, .pic.o   ...and any rules that produce those files. Different extensions denote different programming languages in accordance with gcc convention. |
| hdrs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of header files published by this library to be directly included by sources in dependent rules.  This is the strongly preferred location for declaring header files that describe the interface for the library. These headers will be made available for inclusion by sources in this rule or in dependent rules. Headers not meant to be included by a client of this library should be listed in the srcs attribute instead, even if they are included by a published header. See ["Header inclusion checking"](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#hdrs) for a more detailed description. |
| alwayslink | Boolean; optional; default is False  If 1, any binary that depends (directly or indirectly) on this C++ library will link in all the object files for the files listed in srcs, even if some contain no symbols referenced by the binary. This is useful if your code isn't explicitly called by code in the binary, e.g., if your code registers to receive some callback provided by some service.  If alwayslink doesn't work with VS 2017 on Windows, that is due to a [known issue](https://github.com/bazelbuild/bazel/issues/3949), please upgrade your VS 2017 to the latest version. |
| copts | List of strings; optional  Add these options to the C++ compilation command. Subject to ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization).  Each string in this attribute is added in the given order to COPTS before compiling the binary target. The flags take effect only for compiling this target, not its dependencies, so be careful about header files included elsewhere. All paths should be relative to the workspace, not to the current package.  If the package declares the [feature](https://docs.bazel.build/versions/4.0.0/be/functions.html#package.features) no\_copts\_tokenization, Bourne shell tokenization applies only to strings that consist of a single "Make" variable. |
| defines | List of strings; optional  List of defines to add to the compile line. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization). Each string, which must consist of a single Bourne shell token, is prepended with -D and added to the compile command line to this target, as well as to every rule that depends on it. Be very careful, since this may have far-reaching effects. When in doubt, add define values to [local\_defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.local_defines) instead. |
| include\_prefix | String; optional  The prefix to add to the paths of the headers of this rule.  When set, the headers in the hdrs attribute of this rule are accessible at is the value of this attribute prepended to their repository-relative path.  The prefix in the strip\_include\_prefix attribute is removed before this prefix is added. |
| includes | List of strings; optional  List of include dirs to be added to the compile line.  Subject to ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution. Each string is prepended with -isystem and added to COPTS. Unlike [COPTS](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts), these flags are added for this rule and every rule that depends on it. (Note: not the rules it depends upon!) Be very careful, since this may have far-reaching effects. When in doubt, add "-I" flags to [COPTS](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts) instead.  Headers must be added to srcs or hdrs, otherwise they will not be available to dependent rules when compilation is sandboxed (the default). |
| linkopts | List of strings; optional  Add these flags to the C++ linker command. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution, [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization) and [label expansion](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#label-expansion). Each string in this attribute is added to LINKOPTS before linking the binary target.  Each element of this list that does not start with $ or - is assumed to be the label of a target in deps. The list of files generated by that target is appended to the linker options. An error is reported if the label is invalid, or is not declared in deps. |
| linkstamp | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  Simultaneously compiles and links the specified C++ source file into the final binary. This trickery is required to introduce timestamp information into binaries; if we compiled the source file to an object file in the usual way, the timestamp would be incorrect. A linkstamp compilation may not include any particular set of compiler flags and so should not depend on any particular header, compiler option, or other build variable. *This option should only be needed in the base package.* |
| linkstatic | Boolean; optional; default is False  For [cc\_binary](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary) and [cc\_test](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test): link the binary in static mode. For cc\_library.linkstatic: see below.  By default this option is on for cc\_binary and off for the rest.  If enabled and this is a binary or test, this option tells the build tool to link in .a's instead of .so's for user libraries whenever possible. Some system libraries may still be linked dynamically, as are libraries for which there is no static library. So the resulting executable will still be dynamically linked, hence only *mostly* static.  There are really three different ways to link an executable:   * STATIC with fully\_static\_link feature, in which everything is linked statically; e.g. "gcc -static foo.o libbar.a libbaz.a -lm". This mode is enabled by specifying fully\_static\_link in the [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#features) attribute. * STATIC, in which all user libraries are linked statically (if a static version is available), but where system libraries (excluding C/C++ runtime libraries) are linked dynamically, e.g. "gcc foo.o libfoo.a libbaz.a -lm". This mode is enabled by specifying linkstatic=True. * DYNAMIC, in which all libraries are linked dynamically (if a dynamic version is available), e.g. "gcc foo.o libfoo.so libbaz.so -lm". This mode is enabled by specifying linkstatic=False.   The linkstatic attribute has a different meaning if used on a [cc\_library()](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library) rule. For a C++ library, linkstatic=True indicates that only static linking is allowed, so no .so will be produced. linkstatic=False does not prevent static libraries from being created. The attribute is meant to control the creation of dynamic libraries.  If linkstatic=False, then the build tool will create symlinks to depended-upon shared libraries in the \*.runfiles area. |
| local\_defines | List of strings; optional  List of defines to add to the compile line. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization). Each string, which must consist of a single Bourne shell token, is prepended with -D and added to the compile command line for this target, but not to its dependents. |
| nocopts | String; optional  Remove matching options from the C++ compilation command. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution. The value of this attribute is interpreted as a regular expression. Any preexisting COPTS that match this regular expression (including values explicitly specified in the rule's [copts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts) attribute) will be removed from COPTS for purposes of compiling this rule. This attribute should rarely be needed. |
| strip\_include\_prefix | String; optional  The prefix to strip from the paths of the headers of this rule.  When set, the headers in the hdrs attribute of this rule are accessible at their path with this prefix cut off.  If it's a relative path, it's taken as a package-relative one. If it's an absolute one, it's understood as a repository-relative path.  The prefix in the include\_prefix attribute is added after this prefix is stripped. |
| textual\_hdrs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of header files published by this library to be textually included by sources in dependent rules.  This is the location for declaring header files that cannot be compiled on their own; that is, they always need to be textually included by other source files to build valid code. |
| win\_def\_file | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The Windows DEF file to be passed to linker.  This attribute should only be used when Windows is the target platform. It can be used to [export symbols](https://msdn.microsoft.com/en-us/library/d91k01sh.aspx) during linking a shared library. |

## **cc\_test**

cc\_test([name](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.name), [deps](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.deps), [srcs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.srcs), [data](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.data), [additional\_linker\_inputs](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.additional_linker_inputs), [args](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.args), [compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.compatible_with), [copts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.copts), [defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.defines), [deprecation](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.deprecation), [distribs](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.distribs), [env](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.env), [env\_inherit](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.env_inherit), [exec\_compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.exec_compatible_with), [exec\_properties](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.exec_properties), [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.features), [flaky](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.flaky), [includes](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.includes), [licenses](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.licenses), [linkopts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.linkopts), [linkstatic](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.linkstatic), [local](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.local), [local\_defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.local_defines), [malloc](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.malloc), [nocopts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.nocopts), [restricted\_to](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.restricted_to), [shard\_count](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.shard_count), [size](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.size), [stamp](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.stamp), [tags](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.tags), [target\_compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.target_compatible_with), [testonly](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.testonly), [timeout](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.timeout), [toolchains](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.toolchains), [visibility](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.visibility), [win\_def\_file](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test.win_def_file))

### **Arguments**

| **Attributes** | |
| --- | --- |
| name | [Name](https://docs.bazel.build/versions/4.0.0/build-ref.html#name); required  A unique name for this target. |
| deps | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of other libraries to be linked in to the binary target.  These can be cc\_library or objc\_library targets. |
| srcs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of C and C++ files that are processed to create the target. These are C/C++ source and header files, either non-generated (normal source code) or generated.  All .cc, .c, and .cpp files will be compiled. These might be generated files: if a named file is in the outs of some other rule, this rule will automatically depend on that other rule.  A .h file will not be compiled, but will be available for inclusion by sources in this rule. Both .cc and .h files can directly include headers listed in these srcs or in the hdrs of any rule listed in the deps argument.  All #included files must be mentioned in the srcs attribute of this rule, or in the hdrs attribute of referenced cc\_library()s. The recommended style is for headers associated with a library to be listed in that library's hdrs attribute, and any remaining headers associated with this rule's sources to be listed in srcs. See ["Header inclusion checking"](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#hdrs) for a more detailed description.  If a rule's name is in the srcs, then this rule automatically depends on that one. If the named rule's outs are C or C++ source files, they are compiled into this rule; if they are library files, they are linked in.  Permitted srcs file types:   * C and C++ source files: .c, .cc, .cpp, .cxx, .c++, .C * C and C++ header files: .h, .hh, .hpp, .hxx, .inc, .inl, .H * Assembler with C preprocessor: .S * Archive: .a, .pic.a * "Always link" library: .lo, .pic.lo * Shared library, versioned or unversioned: .so, .so.*version* * Object file: .o, .pic.o   ...and any rules that produce those files. Different extensions denote different programming languages in accordance with gcc convention. |
| additional\_linker\_inputs | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  Pass these files to the C++ linker command.  For example, compiled Windows .res files can be provided here to be embedded in the binary target. |
| copts | List of strings; optional  Add these options to the C++ compilation command. Subject to ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization).  Each string in this attribute is added in the given order to COPTS before compiling the binary target. The flags take effect only for compiling this target, not its dependencies, so be careful about header files included elsewhere. All paths should be relative to the workspace, not to the current package.  If the package declares the [feature](https://docs.bazel.build/versions/4.0.0/be/functions.html#package.features) no\_copts\_tokenization, Bourne shell tokenization applies only to strings that consist of a single "Make" variable. |
| defines | List of strings; optional  List of defines to add to the compile line. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization). Each string, which must consist of a single Bourne shell token, is prepended with -D and added to the compile command line to this target, as well as to every rule that depends on it. Be very careful, since this may have far-reaching effects. When in doubt, add define values to [local\_defines](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.local_defines) instead. |
| includes | List of strings; optional  List of include dirs to be added to the compile line.  Subject to ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution. Each string is prepended with -isystem and added to COPTS. Unlike [COPTS](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts), these flags are added for this rule and every rule that depends on it. (Note: not the rules it depends upon!) Be very careful, since this may have far-reaching effects. When in doubt, add "-I" flags to [COPTS](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts) instead.  Headers must be added to srcs or hdrs, otherwise they will not be available to dependent rules when compilation is sandboxed (the default). |
| linkopts | List of strings; optional  Add these flags to the C++ linker command. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution, [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization) and [label expansion](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#label-expansion). Each string in this attribute is added to LINKOPTS before linking the binary target.  Each element of this list that does not start with $ or - is assumed to be the label of a target in deps. The list of files generated by that target is appended to the linker options. An error is reported if the label is invalid, or is not declared in deps. |
| linkstatic | Boolean; optional; default is False  For [cc\_binary](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary) and [cc\_test](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_test): link the binary in static mode. For cc\_library.linkstatic: see below.  By default this option is on for cc\_binary and off for the rest.  If enabled and this is a binary or test, this option tells the build tool to link in .a's instead of .so's for user libraries whenever possible. Some system libraries may still be linked dynamically, as are libraries for which there is no static library. So the resulting executable will still be dynamically linked, hence only *mostly* static.  There are really three different ways to link an executable:   * STATIC with fully\_static\_link feature, in which everything is linked statically; e.g. "gcc -static foo.o libbar.a libbaz.a -lm". This mode is enabled by specifying fully\_static\_link in the [features](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#features) attribute. * STATIC, in which all user libraries are linked statically (if a static version is available), but where system libraries (excluding C/C++ runtime libraries) are linked dynamically, e.g. "gcc foo.o libfoo.a libbaz.a -lm". This mode is enabled by specifying linkstatic=True. * DYNAMIC, in which all libraries are linked dynamically (if a dynamic version is available), e.g. "gcc foo.o libfoo.so libbaz.so -lm". This mode is enabled by specifying linkstatic=False.   The linkstatic attribute has a different meaning if used on a [cc\_library()](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_library) rule. For a C++ library, linkstatic=True indicates that only static linking is allowed, so no .so will be produced. linkstatic=False does not prevent static libraries from being created. The attribute is meant to control the creation of dynamic libraries.  If linkstatic=False, then the build tool will create symlinks to depended-upon shared libraries in the \*.runfiles area. |
| local\_defines | List of strings; optional  List of defines to add to the compile line. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization). Each string, which must consist of a single Bourne shell token, is prepended with -D and added to the compile command line for this target, but not to its dependents. |
| malloc | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; default is @bazel\_tools//tools/cpp:malloc  Override the default dependency on malloc.  By default, C++ binaries are linked against //tools/cpp:malloc, which is an empty library so the binary ends up using libc malloc. This label must refer to a cc\_library. If compilation is for a non-C++ rule, this option has no effect. The value of this attribute is ignored if linkshared=True is specified. |
| nocopts | String; optional  Remove matching options from the C++ compilation command. Subject to ["Make" variable](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution. The value of this attribute is interpreted as a regular expression. Any preexisting COPTS that match this regular expression (including values explicitly specified in the rule's [copts](https://docs.bazel.build/versions/4.0.0/be/c-cpp.html#cc_binary.copts) attribute) will be removed from COPTS for purposes of compiling this rule. This attribute should rarely be needed. |
| stamp | Integer; optional; default is 0  Enable link stamping. Whether to encode build information into the binary. Possible values:   * stamp = 1: Stamp the build information into the binary. Stamped binaries are only rebuilt when their dependencies change. Use this if there are tests that depend on the build information. * stamp = 0: Always replace build information by constant values. This gives good build result caching. * stamp = -1: Embedding of build information is controlled by the [--[no]stamp](https://docs.bazel.build/versions/4.0.0/user-manual.html#flag--stamp) flag. |
| win\_def\_file | [Label](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The Windows DEF file to be passed to linker.  This attribute should only be used when Windows is the target platform. It can be used to [export symbols](https://msdn.microsoft.com/en-us/library/d91k01sh.aspx) during linking a shared library. |

# Common definitions

## **Attributes common to all build rules**

| **Attribute** | **Description** |
| --- | --- |
| data | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The list of files needed by this rule at runtime.  Targets named in the data attribute will appear in the \*.runfiles area of this rule, if it has one. This may include data files needed by a binary or library, or other programs needed by it. See the [data dependencies](https://docs.bazel.build/versions/4.0.0/build-ref.html#data) section for more information about how to depend on and use data files.  Almost all rules permit a data attribute, but where this attribute is not allowed, this fact is documented under the specific rule. |
| visibility | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; default [default\_visibility](https://docs.bazel.build/versions/4.0.0/be/functions.html#package.default_visibility) from [package](https://docs.bazel.build/versions/4.0.0/be/functions.html#package) if specified, or //visibility:private otherwise; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  The visibility attribute on a rule controls whether the rule can be used by other packages. See the documentation for [visibility](https://docs.bazel.build/versions/4.0.0/visibility.html). |
| toolchains | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  The set of targets whose [Make variables](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) the rule is allowed to access. These rules are either rules that provide the TemplateVariableInfo provider or special targets for toolchain types built into Bazel. These include:   * @bazel\_tools//tools/cpp:current\_cc\_toolchain * @bazel\_tools//tools/cpp:current\_java\_runtime   Note that this is distinct from the concept of [toolchain resolution](https://docs.bazel.build/versions/4.0.0/toolchains.html#toolchain-resolution) that is used by rule implementations for platform-dependent configuration. You cannot use this attribute to determine which specific cc\_toolchain or java\_toolchain a target will use. |
| deps | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  A list of dependencies of this rule.  The precise semantics of what it means for this rule to depend on another using deps are specific to the kind of this rule, and the rule-specific documentation below goes into more detail. At a minimum, though, the targets named via deps will appear in the \*.runfiles area of this rule, if it has one.  Most often, a deps dependency is used to allow one module to use symbols defined in another module written in the same programming language and separately compiled. Cross-language dependencies are also permitted in many cases: for example, a java\_library rule may depend on C++ code in a cc\_library rule, by declaring the latter in the deps attribute. See the definition of [dependencies](https://docs.bazel.build/versions/4.0.0/build-ref.html#deps) for more information.  Almost all rules permit a deps attribute, but where this attribute is not allowed, this fact is documented under the specific rule. |
| deprecation | String; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  An explanatory warning message associated with this rule. Typically this is used to notify users that a rule has become obsolete, or has become superseded by another rule, is private to a package, or is perhaps considered harmful for some reason. It is a good idea to include some reference (like a webpage, a bug number or example migration CLs) so that one can easily find out what changes are required to avoid the message. If there is a new target that can be used as a drop in replacement, it is a good idea to just migrate all users of the old target.  This attribute has no effect on the way things are built, but it may affect a build tool's diagnostic output. The build tool issues a warning when a rule with a deprecation attribute is depended upon by another rule.  Intra-package dependencies are exempt from this warning, so that, for example, building the tests of a deprecated rule does not encounter a warning.  If a deprecated rule depends on another deprecated rule, no warning message is issued.  Once people have stopped using it, the package can be removed. |
| tags | List of arbitrary text tags. Tags may be any valid string; default is the empty list; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  *Tags* can be used on any rule. *Tags* on test and test\_suite rules are useful for categorizing the tests. *Tags* on non-test rules are used to control sandboxed execution of genrules and [Starlark](https://docs.bazel.build/versions/4.0.0/skylark/index.html) actions, and for parsing by humans and/or external tools.  Bazel modifies the behavior of its sandboxing code if it finds the following keywords in the tags attribute of any test rule or genrule, or the keys of execution\_requirements for any Starlark action.   * no-sandbox keyword results in the action or test never being sandboxed; it can still be cached or run remotely - use no-cache or no-remote to prevent either or both of those. * no-cache keyword results in the action or test never being cached (remotely or locally) * no-remote-cache keyword results in the action or test never being cached remotely (but it may be cached locally; it may also be executed remotely). Note: for the purposes of this tag, the disk-cache is considered a local cache, whereas the http and gRPC caches are considered remote. If a combined cache is specified (i.e. a cache with local and remote components), it's treated as a remote cache and disabled entirely. * no-remote-exec keyword results in the action or test never being executed remotely (but it may be cached remotely). * no-remote keyword prevents the action or test from being executed remotely or cached remotely. This is equivalent to using both no-remote-cache and no-remote-exec. * local keyword precludes the action or test from being remotely cached, remotely executed, or run inside the sandbox. For genrules and tests, marking the rule with the local = True attribute has the same effect. * requires-network keyword allows access to the external network from inside the sandbox. This tag only has an effect if sandboxing is enabled. * block-network keyword blocks access to the external network from inside the sandbox. In this case, only communication with localhost is allowed. This tag only has an effect if sandboxing is enabled. * requires-fakeroot runs the test or action as uid and gid 0 (i.e., the root user). This is only supported on Linux. This tag takes precedence over the --sandbox\_fake\_username command-line option.   *Tags* on tests are generally used to annotate a test's role in your debug and release process. Typically, tags are most useful for C++ and Python tests, which lack any runtime annotation ability. The use of tags and size elements gives flexibility in assembling suites of tests based around codebase check-in policy.  Bazel modifies test running behavior if it finds the following keywords in the tags attribute of the test rule:   * exclusive will force the test to be run in the "exclusive" mode, ensuring that no other tests are running at the same time. Such tests will be executed in serial fashion after all build activity and non-exclusive tests have been completed. Remote execution is disabled for such tests because Bazel doesn't have control over what's running on a remote machine. * manual keyword will exclude the target from expansion of target pattern wildcards (..., :\*, :all, etc.) and test\_suite rules which do not list the test explicitly when computing the set of top-level targets to build/run for the build, test, and coverage commands. It does not affect target wildcard or test suite expansion in other contexts, including the query command. Note that manual does not imply that a target should not be built/run automatically by continuous build/test systems. For example, it may be desirable to exclude a target from bazel test ... because it requires specific Bazel flags, but still have it included in properly-configured presubmit or continuous test runs. * external keyword will force test to be unconditionally executed (regardless of --cache\_test\_results value).   See [Tag Conventions](https://docs.bazel.build/versions/4.0.0/test-encyclopedia.html#tag-conventions) in the Test Encyclopedia for more conventions on tags attached to test rules. |
| testonly | Boolean; optional; default False except as noted; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  If True, only testonly targets (such as tests) can depend on this target.  Equivalently, a rule that is not testonly is not allowed to depend on any rule that is testonly.  Tests (\*\_test rules) and test suites ([test\_suite](https://docs.bazel.build/versions/4.0.0/be/general.html#test_suite) rules) are testonly by default.  This attribute is intended to mean that the target should not be contained in binaries that are released to production.  Because testonly is enforced at build time, not run time, and propagates virally through the dependency tree, it should be applied judiciously. For example, stubs and fakes that are useful for unit tests may also be useful for integration tests involving the same binaries that will be released to production, and therefore should probably not be marked testonly. Conversely, rules that are dangerous to even link in, perhaps because they unconditionally override normal behavior, should definitely be marked testonly. |
| features | List of *features*. Default is the empty list.  A feature is string tag that can be enabled or disabled on a target. The meaning of a feature depends on the rule itself.  This features attribute is combined with the [package](https://docs.bazel.build/versions/4.0.0/be/$%7Blinkpackage%7D) level features attribute. For example, if the features ["a", "b"] are enabled on the package level, and a rule features attribute contains ["-a", "c"], the features enabled for the rule will be "b" and "c". [See example](https://github.com/bazelbuild/examples/blob/master/rules/features/BUILD). |
| licenses | List of strings; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  A list of license-type strings to be used for this particular build rule. This is part of a deprecated licensing API that Bazel no longer uses. Don't use this. |
| compatible\_with | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  The list of environments this rule can be built for, in addition to default-supported environments.  This is part of Bazel's soft-launched constraint system, which lets users declare which rules can and cannot depend on each other. For example, externally deployable binaries shouldn't depend on libraries with company-secret code. See [ConstraintSemantics](https://github.com/bazelbuild/bazel/blob/master/src/main/java/com/google/devtools/build/lib/analysis/constraints/ConstraintSemantics.java#L46) for details. |
| exec\_properties | Dictionary of strings. Default is an empty dictionary.  A dictionary of strings that will be added to the exec\_properties of a platform selected for this target. See exec\_properties of the [platform](https://docs.bazel.build/versions/4.0.0/be/platform.html) rule.  If a key is present in both the platform and target-level properties, the value will be taken from the target. |
| distribs | List of strings; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  A list of distribution-method strings to be used for this particular build rule. This is part of a deprecated licensing API that Bazel no longer uses. Don't use this. |
| target\_compatible\_with | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; default is the empty list  A list of [constraint\_value](https://docs.bazel.build/versions/4.0.0/be/platform.html#constraint_value)s that must be present in the target platform for this target to be considered "compatible". This is in addition to any constraints already set by the rule type. If the target platform does not satisfy all listed constraints then the target is considered "incompatible". Incompatible targets are skipped for building and testing when the target pattern is expanded (e.g. `//...`, `:all`). When explicitly specified on the command line, incompatible targets cause Bazel to print an error and cause a build or test failure.  Targets that transitively depend on incompatible targets are themselves considered incompatible. They are also skipped for building and testing.  An empty list (which is the default) signifies that the target is compatible with all platforms.  See the [Platforms](https://docs.bazel.build/versions/4.0.0/platforms.html#skipping-incompatible-targets) page for more information about incompatible target skipping. |
| exec\_compatible\_with | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  A list of [constraint\_values](https://docs.bazel.build/versions/4.0.0/be/platform.html#constraint_value) that must be present in the execution platform for this target. This is in addition to any constraints already set by the rule type. Constraints are used to restrict the list of available execution platforms, see the description of [toolchain resolution](https://docs.bazel.build/versions/4.0.0/toolchains.html#toolchain-resolution) for details. |
| restricted\_to | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  The list of environments this rule can be built for, *instead* of default-supported environments.  This is part of Bazel's soft-launched constraint system. See [compatible\_with](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.compatible_with) for details. |

## **Attributes common to all test rules (\*\_test)**

| **Attribute** | **Description** |
| --- | --- |
| args | List of strings; optional; subject to [$(location)](https://docs.bazel.build/versions/4.0.0/be/make-variables.html#location) and ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution, and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization)  Add these arguments to the --test\_arg when executed by bazel test.  These arguments are passed before the --test\_arg values specified on the bazel test command line. |
| size | String "enormous", "large" "medium" or "small", default is "medium"; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  How "heavy" the test is.  A classification of the test's "heaviness": how much time/resources it needs to run.  Unit tests are considered "small", integration tests "medium", and end-to-end tests "large" or "enormous". Bazel uses the size to determine a default timeout (which can be overridden using the timeout attribute) and the amount of resources that have to be acquired for the test to run. Test sizes correspond to the following resources and default timeouts:   |  |  |  |  | | --- | --- | --- | --- | | **Size** | **RAM (in MB)** | **CPU (in CPU cores)** | **Default timeout** | | small | 20 | 1 | short (1 minute) | | medium | 100 | 1 | moderate (5 minutes) | | large | 300 | 1 | long (15 minutes) | | enormous | 800 | 1 | eternal (60 minutes) |   The environment variable [TEST\_SIZE](https://docs.bazel.build/versions/4.0.0/test-encyclopedia.html#initial-conditions) will be set to the value of this attribute when spawning the test. |
| flaky | Boolean; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  Marks test as flaky.  If set, executes the test up to 3 times before being declared as failed. By default this attribute is set to 0 and test is considered to be stable. Note, that use of this attribute is generally discouraged - we do prefer all tests to be stable. |
| shard\_count | Non-negative integer less than or equal to 50; optional  Specifies the number of parallel shards to use to run the test.  This value will override any heuristics used to determine the number of parallel shards with which to run the test. Note that for some test rules, this parameter may be required to enable sharding in the first place. Also see --test\_sharding\_strategy.  If test sharding is enabled, the environment variable [TEST\_TOTAL\_SHARDS](https://docs.bazel.build/versions/4.0.0/test-encyclopedia.html#initial-conditions) will be set to this value when spawning the test.  Sharding requires the test runner to support the test sharding protocol. If it does not, then it will most likely run every test in every shard, which is not what you want.  See [Test Sharding](https://docs.bazel.build/versions/4.0.0/test-encyclopedia.html#test-sharding) in the Test Encyclopedia for details on sharding. |
| env\_inherit | List of strings; optional  Specifies additional environment variables to inherit from the external environment when the test is executed by bazel test.  This attribute only applies to native rules, like cc\_test, py\_test, and sh\_test. It does not apply to Starlark-defined test rules. |
| env | Dictionary of strings; optional; values are subject to [$(location)](https://docs.bazel.build/versions/4.0.0/be/make-variables.html#location) and ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution  Specifies additional environment variables to set when the test is executed by bazel test.  This attribute only applies to native rules, like cc\_test, py\_test, and sh\_test. It does not apply to Starlark-defined test rules. |
| timeout | String "short", "moderate", "long", "eternal" (with the default derived from the test's size attribute); [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  How long the test is expected to run before returning.  While a test's size attribute controls resource estimation, a test's timeout may be set independently. If not explicitly specified, the timeout is based on the [test's size](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#test.size). The test timeout can be overridden with the --test\_timeout flag, e.g. for running under certain conditions which are known to be slow. Test timeout values correspond to the following time periods:   |  |  | | --- | --- | | **Timeout Value** | **Time Period** | | short | 1 minute | | moderate | 5 minutes | | long | 15 minutes | | eternal | 60 minutes |   For times other than the above, the test timeout can be overridden with the --test\_timeout bazel flag, e.g. for manually running under conditions which are known to be slow. The --test\_timeout values are in seconds. For example --test\_timeout=120 will set the test timeout to two minutes.  The environment variable [TEST\_TIMEOUT](https://docs.bazel.build/versions/4.0.0/test-encyclopedia.html#initial-conditions) will be set to the test timeout (in seconds) when spawning the test. |
| local | Boolean; optional; [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  Forces the test to be run locally, without sandboxing.  By default this attribute is set to 0 and the default testing strategy is used. This is equivalent to providing "local" as a tag (tags=["local"]). |

## **Attributes common to all binary rules (\*\_binary)**

| **Attribute** | **Description** |
| --- | --- |
| args | List of strings; optional; subject to [$(location)](https://docs.bazel.build/versions/4.0.0/be/make-variables.html#location) and ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution, and [Bourne shell tokenization](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#sh-tokenization); [nonconfigurable](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#configurable-attributes)  Command line arguments that bazel will pass to the target when it is executed either by the run command or as a test. These arguments are passed before the ones that are specified on the bazel run or bazel test command line.  *NOTE: The arguments are not passed when you run the target outside of bazel (for example, by manually executing the binary in bazel-bin/).*  Most binary rules permit an args attribute, but where this attribute is not allowed, this fact is documented under the specific rule. |
| env | Dictionary of strings; optional; values are subject to [$(location)](https://docs.bazel.build/versions/4.0.0/be/make-variables.html#location) and ["Make variable"](https://docs.bazel.build/versions/4.0.0/be/make-variables.html) substitution  Specifies additional environment variables to set when the target is executed by bazel run.  This attribute only applies to native rules, like cc\_binary, py\_binary, and sh\_binary. It does not apply to Starlark-defined executable rules.  *NOTE: The environment variables are not set when you run the target outside of bazel (for example, by manually executing the binary in bazel-bin/).* |
| output\_licenses | List of strings; optional  The licenses of the output files that this binary generates. This is part of a deprecated licensing API that Bazel no longer uses. Don't use this. |

# Functions

## **package**

package(default\_deprecation, default\_testonly, default\_visibility, features)

This function declares metadata that applies to every subsequent rule in the package. It is used at most once within a package (BUILD file).

The package() function should be called right after all the load() statements at the top of the file, before any rule.

### **Arguments**

| **Attribute** | **Description** |
| --- | --- |
| default\_visibility | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  The default visibility of the rules in this package.  Every rule in this package has the visibility specified in this attribute, unless otherwise specified in the visibility attribute of the rule. For detailed information about the syntax of this attribute, see the documentation of [visibility](https://docs.bazel.build/versions/4.0.0/visibility.html). The package default visibility does not apply to [exports\_files](https://docs.bazel.build/versions/4.0.0/be/functions.html#exports_files), which is public by default. |
| default\_deprecation | String; optional  Sets the default [deprecation](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.deprecation) message for all rules in this package. |
| default\_testonly | Boolean; optional; default is 0 except as noted  Sets the default [testonly](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.testonly) property for all rules in this package.  In packages under javatests the default value is 1. |
| features | List strings; optional  Sets various flags that affect the semantics of this BUILD file.  This feature is mainly used by the people working on the build system to tag packages that need some kind of special handling. Do not use this unless explicitly requested by someone working on the build system. |

The declaration below declares that the rules in this package are visible only to members of package group //foo:target. Individual visibility declarations on a rule, if present, override this specification.

package(default\_visibility = ["//foo:target"])

## **package\_group**

package\_group(name, packages, includes)

This function defines a set of packages and assigns a label to the group. The label can be referenced in visibility attributes.

Package groups are used for visibility control. You can grant access to a rule to one or more package groups, every rule in the entire source tree, or only to rules declared in the same package. For more detailed description of the visibility system, see the [visibility](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.visibility) attribute.

### **Arguments**

| **Attribute** | **Description** |
| --- | --- |
| name | [Name](https://docs.bazel.build/versions/4.0.0/build-ref.html#name); required  A unique name for this target. |
| packages | List of [Package](https://docs.bazel.build/versions/4.0.0/build-ref.html#packages); optional  A complete enumeration of packages in this group.  Packages should be referred to using their full names, starting with a double slash. For example, //foo/bar/main is a valid element of this list.  You can also specify wildcards: the specification //foo/... specifies every package under //foo, including //foo itself.  Package specifications can be prefixed with - to indicate negation: the specification -//foo/bar/... excludes all packages under //foo/bar that would otherwise have been matched by the package patterns in the current package\_group. When used together with includes, the set of packages for each package group is computed and then the results are unioned: negative patterns in one package group do not affect the result of included package groups.  If this attribute is missing, the package group itself will contain no packages (but it can still include other package groups). |
| includes | List of [labels](https://docs.bazel.build/versions/4.0.0/build-ref.html#labels); optional  Other package groups that are included in this one.  The labels in this attribute must refer to other package groups. Packages in referenced package groups are taken to be part of this package group. This is transitive, that is, if package group a contains package group b, and b contains package group c, every package in c will also be a member of a. |

## **exports\_files**

exports\_files([*label*, ...], visibility, licenses)

exports\_files() specifies a list of files belonging to this package that are exported to other packages.

The BUILD file for a package may only refer directly to source files belonging to another package if they are explicitly exported with an exports\_files() statement. Read more about [visibility of files](https://docs.bazel.build/versions/4.0.0/visibility.html#visibility-of-a-file).

As a legacy behaviour, also files mentioned as input to a rule are exported with the default visibility until the flag [--incompatible\_no\_implicit\_file\_export](https://github.com/bazelbuild/bazel/issues/10225) is flipped. However, this behavior should not be relied upon and actively migrated away from.

### **Arguments**

The argument is a list of names of files within the current package. A visibility declaration can also be specified; in this case, the files will be visible to the targets specified. If no visibility is specified, the files will be visible to every package, even if a package default visibility was specified in the [package](https://docs.bazel.build/versions/4.0.0/be/functions.html#package) function. The [licenses](https://docs.bazel.build/versions/4.0.0/be/common-definitions.html#common.licenses) can also be specified.

# Globals

Objects, functions and modules registered in the global environment.

## **all**

[bool](https://docs.bazel.build/versions/4.0.0/skylark/lib/bool.html) all(elements)

Returns true if all elements evaluate to True or if the collection is empty. Elements are converted to boolean using the [bool](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#bool) function.

all(["hello", 3, True]) == True

all([-1, 0, 1]) == False

## **any**

[bool](https://docs.bazel.build/versions/4.0.0/skylark/lib/bool.html) any(elements)

Returns true if at least one element evaluates to True. Elements are converted to boolean using the [bool](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#bool) function.

any([-1, 0, 1]) == True

any([False, 0, ""]) == False

## **bool**

[bool](https://docs.bazel.build/versions/4.0.0/skylark/lib/bool.html) bool(x=False)

Constructor for the bool type. It returns False if the object is None, False, an empty string (""), the number 0, or an empty collection (e.g. (), []). Otherwise, it returns True.

## **dict**

[dict](https://docs.bazel.build/versions/4.0.0/skylark/lib/dict.html) dict(pairs=[], \*\*kwargs)

Creates a [dictionary](https://docs.bazel.build/versions/4.0.0/skylark/lib/dict.html) from an optional positional argument and an optional set of keyword arguments. In the case where the same key is given multiple times, the last value will be used. Entries supplied via keyword arguments are considered to come after entries supplied via the positional argument.

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| pairs | ; default = [] A dict, or an iterable whose elements are each of length 2 (key, value). |
| kwargs | ; required Dictionary of additional entries. |

## **dir**

[list](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) dir(x)

Returns a list of strings: the names of the attributes and methods of the parameter object.

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The object to check. |

## **enumerate**

[list](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) enumerate(list, start=0)

Returns a list of pairs (two-element tuples), with the index (int) and the item from the input sequence.

enumerate([24, 21, 84]) == [(0, 24), (1, 21), (2, 84)]

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| list | ; required input sequence. |
| start | ; default = 0 start index. |

## **analysis\_test\_transition**

[transition](https://docs.bazel.build/versions/4.0.0/skylark/lib/transition.html) analysis\_test\_transition(settings)

Creates a configuration transition to be applied on an analysis-test rule's dependencies. This transition may only be applied on attributes of rules with analysis\_test = True.

| **Parameter** | **Description** |
| --- | --- |
| settings | ; required A dictionary containing information about configuration settings which should be set by this configuration transition. Keys are build setting labels and values are their new post-transition values. All other settings are unchanged. Use this to declare specific configuration settings that an analysis test requires to be set in order to pass. |

## **fail**

[None](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#None) fail(msg=None, attr=None, \*args)

Causes execution to fail with an error.

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| msg | ; default = None Deprecated: use positional arguments instead. This argument acts like an implicit leading positional argument. |
| attr | [string](https://docs.bazel.build/versions/4.0.0/skylark/lib/string.html); or [None](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#None); default = None Deprecated. Causes an optional prefix containing this string to be added to the error message. |
| args | ; required A list of values, formatted with str and joined with spaces, that appear in the error message. |

## **float**

[float](https://docs.bazel.build/versions/4.0.0/skylark/lib/float.html) float(x=unbound)

Returns x as a float value.

* If x is already a float, float returns it unchanged.
* If x is a bool, float returns 1.0 for True and 0.0 for False.
* If x is an int, float returns the nearest finite floating-point value to x, or an error if the magnitude is too large.
* If x is a string, it must be a valid floating-point literal, or be equal (ignoring case) to NaN, Inf, or Infinity, optionally preceded by a + or - sign.

Any other value causes an error. With no argument, float() returns 0.0.

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; default = unbound The value to convert. |

## **hasattr**

[bool](https://docs.bazel.build/versions/4.0.0/skylark/lib/bool.html) hasattr(x, name)

Returns True if the object x has an attribute or method of the given name, otherwise False. Example:

hasattr(ctx.attr, "myattr")

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The object to check. |
| name | ; required The name of the attribute. |

## **int**

[int](https://docs.bazel.build/versions/4.0.0/skylark/lib/int.html) int(x, base=unbound)

Returns x as an int value.

* If x is already an int, int returns it unchanged.
* If x is a bool, int returns 1 for True and 0 for False.
* If x is a string, it must have the format <sign><prefix><digits>. <sign> is either "+", "-", or empty (interpreted as positive). <digits> are a sequence of digits from 0 up to base - 1, where the letters a-z (or equivalently, A-Z) are used as digits for 10-35. In the case where base is 2/8/16, <prefix> is optional and may be 0b/0o/0x (or equivalently, 0B/0O/0X) respectively; if the base is any other value besides these bases or the special value 0, the prefix must be empty. In the case where base is 0, the string is interpreted as an integer literal, in the sense that one of the bases 2/8/10/16 is chosen depending on which prefix if any is used. If base is 0, no prefix is used, and there is more than one digit, the leading digit cannot be 0; this is to avoid confusion between octal and decimal. The magnitude of the number represented by the string must be within the allowed range for the int type.
* If x is a float, int returns the integer value of the float, rounding towards zero. It is an error if x is non-finite (NaN or infinity).

This function fails if x is any other type, or if the value is a string not satisfying the above format. Unlike Python's int function, this function does not allow zero arguments, and does not allow extraneous whitespace for string arguments.

Examples:

int("123") == 123

int("-123") == -123

int("+123") == 123

int("FF", 16) == 255

int("0xFF", 16) == 255

int("10", 0) == 10

int("-0x10", 0) == -16

int("-0x10", 0) == -16

int("123.456") == 123

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The string to convert. |
| base | ; default = unbound The base used to interpret a string value; defaults to 10. Must be between 2 and 36 (inclusive), or 0 to detect the base as if x were an integer literal. This parameter must not be supplied if the value is not a string. |

## **len**

[int](https://docs.bazel.build/versions/4.0.0/skylark/lib/int.html) len(x)

Returns the length of a string, sequence (such as a list or tuple), dict, or other iterable.

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The value whose length to report. |

## **list**

[list](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) list(x=[])

Returns a new list with the same elements as the given iterable value.

list([1, 2]) == [1, 2]

list((2, 3, 2)) == [2, 3, 2]

list({5: "a", 2: "b", 4: "c"}) == [5, 2, 4]

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; default = [] The object to convert. |

## **max**

unknown max(\*args)

Returns the largest one of all given arguments. If only one argument is provided, it must be a non-empty iterable.It is an error if elements are not comparable (for example int with string), or if no arguments are given.

max(2, 5, 4) == 5

max([5, 6, 3]) == 6

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| args | ; required The elements to be checked. |

## **min**

unknown min(\*args)

Returns the smallest one of all given arguments. If only one argument is provided, it must be a non-empty iterable. It is an error if elements are not comparable (for example int with string), or if no arguments are given.

min(2, 5, 4) == 2

min([5, 6, 3]) == 3

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| args | ; required The elements to be checked. |

## **print**

[None](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#None) print(sep=" ", \*args)

Prints args as debug output. It will be prefixed with the string "DEBUG" and the location (file and line number) of this call. The exact way in which the arguments are converted to strings is unspecified and may change at any time. In particular, it may be different from (and more detailed than) the formatting done by [str()](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#str) and [repr()](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#repr).

Using print in production code is discouraged due to the spam it creates for users. For deprecations, prefer a hard error using [fail()](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#fail) whenever possible.

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| sep | ; default = " " The separator string between the objects, default is space (" "). |
| args | ; required The objects to print. |

## **range**

[sequence](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) range(start\_or\_stop, stop\_or\_none=None, step=1)

Creates a list where items go from start to stop, using a step increment. If a single argument is provided, items will range from 0 to that element.

range(4) == [0, 1, 2, 3]

range(3, 9, 2) == [3, 5, 7]

range(3, 0, -1) == [3, 2, 1]

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| start\_or\_stop | ; required Value of the start element if stop is provided, otherwise value of stop and the actual start is 0 |
| stop\_or\_none | [int](https://docs.bazel.build/versions/4.0.0/skylark/lib/int.html); or [None](https://docs.bazel.build/versions/4.0.0/skylark/lib/globals.html#None); default = None optional index of the first item *not* to be included in the resulting list; generation of the list stops before stop is reached. |
| step | ; default = 1 The increment (default is 1). It may be negative. |

## **repr**

[string](https://docs.bazel.build/versions/4.0.0/skylark/lib/string.html) repr(x)

Converts any object to a string representation. This is useful for debugging.

repr("ab") == '"ab"'

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The object to convert. |

## **reversed**

[list](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) reversed(sequence)

Returns a new, unfrozen list that contains the elements of the original iterable sequence in reversed order.

reversed([3, 5, 4]) == [4, 5, 3]

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| sequence | ; required The iterable sequence (e.g. list) to be reversed. |

## **sorted**

[list](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) sorted(iterable, \*, key=None, reverse=False)

Returns a new sorted list containing all the elements of the supplied iterable sequence. An error may occur if any pair of elements x, y may not be compared using x < y. The elements are sorted into ascending order, unless the reverse argument is True, in which case the order is descending. Sorting is stable: elements that compare equal retain their original relative order.

sorted([3, 5, 4]) == [3, 4, 5]

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| iterable | ; required The iterable sequence to sort. |
| key | ; default = None An optional function applied to each element before comparison. |
| reverse | ; default = False Return results in descending order. |

## **str**

[string](https://docs.bazel.build/versions/4.0.0/skylark/lib/string.html) str(x)

Converts any object to string. This is useful for debugging.

str("ab") == "ab"

str(8) == "8"

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The object to convert. |

## **tuple**

[tuple](https://docs.bazel.build/versions/4.0.0/skylark/lib/tuple.html) tuple(x=())

Returns a tuple with the same elements as the given iterable value.

tuple([1, 2]) == (1, 2)

tuple((2, 3, 2)) == (2, 3, 2)

tuple({5: "a", 2: "b", 4: "c"}) == (5, 2, 4)

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; default = () The object to convert. |

## **type**

[string](https://docs.bazel.build/versions/4.0.0/skylark/lib/string.html) type(x)

Returns the type name of its argument. This is useful for debugging and type-checking. Examples:

type(2) == "int"

type([1]) == "list"

type(struct(a = 2)) == "struct"

This function might change in the future. To write Python-compatible code and be future-proof, use it only to compare return values:

if type(x) == type([]): # if x is a list

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| x | ; required The object to check type of. |

## **zip**

[list](https://docs.bazel.build/versions/4.0.0/skylark/lib/list.html) zip(\*args)

Returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The list has the size of the shortest input. With a single iterable argument, it returns a list of 1-tuples. With no arguments, it returns an empty list. Examples:

zip() # == []

zip([1, 2]) # == [(1,), (2,)]

zip([1, 2], [3, 4]) # == [(1, 3), (2, 4)]

zip([1, 2], [3, 4, 5]) # == [(1, 3), (2, 4)]

### **Parameters**

| **Parameter** | **Description** |
| --- | --- |
| args | ; required lists to zip. |