

## 1.1 Limitations of g++

- Limitations on input source code: 240 nesting levels with the parser stacksize (YYSTACKSIZE) set to 500 (the default), and requires around 16.4k swap space per nesting level. The parser needs about 2.09 \* number of nesting levels worth of stackspace.
- I suspect there are other uses of `pushdecl_class_level` that do not call `set_identifier_type_value` in tandem with the call to `pushdecl_class_level`. It would seem to be an omission.
- For two argument delete, the second argument is always calculated by “`virtual_size =`” in the source. It currently has a problem, in that object size is not calculated by the virtual destructor and passed back for the second parameter to delete. Destructors need to return a value just like constructors. ANSI C++ Jun 5 92 wp 12.5.6  
The second argument is magically deleted in `build_method_call`, if it is not used. It needs to be deleted for global operator delete also.
- Visibility checking in general is unimplemented, there are a few cases where it is implemented. `grok_enum_decls` should be used in more places to do visibility checking, but this is only the tip of a bigger problem.
- `volatile` is not implemented in general.
- Pointers to members are only minimally supported, and there are places where the grammar doesn’t even properly accept them yet.

## 1.2 Routines

This section describes some of the routines used in the C++ front-end.

`build_vtable` and `prepare_fresh_vtable` is used only within the `cp-class.c` file, and only in `finish_struct` and `modify_vtable_entries`.

`build_vtable`, `prepare_fresh_vtable`, and `finish_struct` are the only routines that set `DECL_VPARENT`.

`finish_struct` can steal the virtual function table from parents, this prohibits related\_vslot from working. When `finish_struct` steals, we know that

```
get_binfo (DECL_FIELD_CONTEXT (CLASSTYPE_VFIELD (t)), t, 0)
```

will get the related binfo.

`layout_basetypes` does something with the VIRTUALS.

Supposedly (according to Tiemann) most of the breadth first searching done, like in `get_base_distance` and in `get_binfo` was not because of any design decision. I have since found out the at least one part of the compiler needs the notion of depth first binfo searching, I am going to try and convert the whole thing, it should just work. The term left-most refers to the depth first left-most node. It uses `MAIN_VARIANT == type` as the condition to get left-most, because the things that have `BINFO_OFFSETs` of zero are shared and will have themselves as their own `MAIN_VARIANTs`. The non-shared right ones, are copies of the left-most one, hence if it is its own `MAIN_VARIANT`, we know it IS a left-most one, if it is not, it is a non-left-most one.

`get_base_distance`’s path and distance matters in its use in:

- `prepare_fresh_vtable` (the code is probably wrong)

- `init_vfields` Depends upon distance probably in a safe way, `build_offset_ref` might use partial paths to do further lookups, `hack_identifier` is probably not properly checking visibility.
- `get_first_matching_virtual` probably should check for `get_base_distance` returning -2.
- `resolve_offset_ref` should be called in a more deterministic manner. Right now, it is called in some random contexts, like for arguments at `build_method_call` time, `default_conversion` time, `convert_arguments` time, `build_unary_op` time, `build_c_cast` time, `build_modify_expr` time, `convert_for_assignment` time, and `convert_for_initialization` time.

But, there are still more contexts it needs to be called in, one was the ever simple:

```
if (obj.*pmi != 7)
    ...
```

Seems that the problems were due to the fact that `TREE_TYPE` of the `OFFSET_REF` was not a `OFFSET_TYPE`, but rather the type of the referent (like `INTEGER_TYPE`). This problem was fixed by changing `default_conversion` to check `TREE_CODE (x)`, instead of only checking `TREE_CODE (TREE_TYPE (x))` to see if it was `OFFSET_TYPE`.

### 1.3 Implementation Specifics

- Explicit Initialization

The global list `current_member_init_list` contains the list of mem-initializers specified in a constructor declaration. For example:

```
foo::foo() : a(1), b(2) {}
```

will initialize ‘a’ with 1 and ‘b’ with 2. `expand_member_init` places each initialization (a with 1) on the global list. Then, when the `fndecl` is being processed, `emit_base_init` runs down the list, initializing them. It used to be the case that g++ first ran down `current_member_init_list`, then ran down the list of members initializing the ones that weren’t explicitly initialized. Things were rewritten to perform the initializations in order of declaration in the class. So, for the above example, ‘a’ and ‘b’ will be initialized in the order that they were declared:

```
class foo { public: int b; int a; foo (); };
```

Thus, ‘b’ will be initialized with 2 first, then ‘a’ will be initialized with 1, regardless of how they’re listed in the mem-initializer.

- Argument Matching

In early 1993, the argument matching scheme in GNU C++ changed significantly. The original code was completely replaced with a new method that will, hopefully, be easier to understand and make fixing specific cases much easier.

The ‘`-fansi-overloading`’ option is used to enable the new code; at some point in the future, it will become the default behavior of the compiler.

The file `cp-call.c` contains all of the new work, in the functions `rank_for_overload`, `compute_harshness`, `compute_conversion_costs`, and `ideal_candidate`.

Instead of using obscure numerical values, the quality of an argument match is now represented by clear, individual codes. The new data structure `struct harshness` (it used to be an `unsigned` number) contains:

- a. the ‘code’ field, to signify what was involved in matching two arguments;
- b. the ‘distance’ field, used in situations where inheritance decides which function should be called (one is “closer” than another);
- c. and the ‘int\_penalty’ field, used by some codes as a tie-breaker.

The ‘code’ field is a number with a given bit set for each type of code, OR’d together. The new codes are:

- `EVIL_CODE` The argument was not a permissible match.
- `CONST_CODE` Currently, this is only used by `compute_conversion_costs`, to distinguish when a non-`const` member function is called from a `const` member function.
- `ELLIPSIS_CODE` A match against an ellipsis ‘...’ is considered worse than all others.
- `USER_CODE` Used for a match involving a user-defined conversion.
- `STD_CODE` A match involving a standard conversion.
- `PROMO_CODE` A match involving an integral promotion. For these, the `int_penalty` field is used to handle the ARM’s rule (XXX cite) that a smaller `unsigned` type should promote to a `int`, not to an `unsigned int`.
- `QUAL_CODE` Used to mark use of qualifiers like `const` and `volatile`.
- `TRIVIAL_CODE` Used for trivial conversions. The ‘int\_penalty’ field is used by `convert_harshness` to communicate further penalty information back to `build_overload_call_real` when deciding which function should be call.

The functions `convert_to_aggr` and `build_method_call` use `compute_conversion_costs` to rate each argument’s suitability for a given candidate function (that’s how we get the list of candidates for `ideal_candidate`).

## 1.4 Glossary

**binfo**            The main data structure in the compiler used to represent the inheritance relationships between classes. The data in the `binfo` can be accessed by the `BINFO_` accessor macros.

**vtable**

virtual function table

The virtual function table holds information used in virtual function dispatching. In the compiler, they are usually referred to as `vtables`, or `vtbls`. The first index is not used in the normal way, I believe it is probably used for the virtual destructor.

**vfield**

`vfields` can be thought of as the base information needed to build `vtables`. For every `vtable` that exists for a class, there is a `vfield`. See also `vtable` and virtual function table pointer. When a type is used as a base class to another type, the virtual function table for the derived class can be based upon the `vtable` for the base class, just extended to include the additional virtual methods declared in the derived class.

virtual function table pointer

These are `FIELD_DECLs` that are pointer types that point to vtables. See also `vtable` and `vfield`.

## 1.5 Macros

This section describes some of the macros used on trees. The list should be alphabetical. Eventually all macros should be documented here. There are some postscript drawings that can be used to better understand from the more complex data structures, contact Mike Stump ([mrs@cygnus.com](mailto:mrs@cygnus.com)) for information about them.

### `BINFO_BASETYPES`

A vector of additional binfos for the types inherited by this basetype. The binfos are fully unshared (except for virtual bases, in which case the binfo structure is shared).

If this basetype describes type D as inherited in C, and if the basetypes of D are E and F, then this vector contains binfos for inheritance of E and F by C.

Has values of:

`TREE_VECs`

### `BINFO_INHERITANCE_CHAIN`

Temporarily used to represent specific inheritances. It usually points to the binfo associated with the lesser derived type, but it can be reversed by `reverse_path`. For example:

```
Z ZbY least derived
|
Y YbX
|
X Xb most derived
```

```
TYPE_BINFO (X) == Xb
BINFO_INHERITANCE_CHAIN (Xb) == YbX
BINFO_INHERITANCE_CHAIN (Yb) == ZbY
BINFO_INHERITANCE_CHAIN (Zb) == 0
```

Not sure if the above is really true, `get_base_distance` has its point towards the most derived type, opposite from above.

Set by `build_vbase_path`, `recursive_bounded_basetype_p`, `get_base_distance`, `lookup_field`, `lookup_fnfields`, and `reverse_path`.

What things can this be used on:

`TREE_VECs` that are binfos

### `BINFO_OFFSET`

The offset where this basetype appears in its containing type. `BINFO_OFFSET` slot holds the offset (in bytes) from the base of the complete object to the base of the part of the object that is allocated on behalf of this 'type'. This is always 0 except when there is multiple inheritance.

Used on `TREE_VEC_ELTs` of the binfos `BINFO_BASETYPES (...)` for example.

**BINFO\_VIRTUALS**

A unique list of functions for the virtual function table. See also `TYPE_BINFO_VIRTUALS`.

What things can this be used on:

`TREE_VECs` that are binfos

**BINFO\_VTABLE**

Used to find the `VAR_DECL` that is the virtual function table associated with this binfo. See also `TYPE_BINFO_VTABLE`. To get the virtual function table pointer, see `CLASSTYPE_VFIELD`.

What things can this be used on:

`TREE_VECs` that are binfos

Has values of:

`VAR_DECLS` that are virtual function tables

**BLOCK\_SUPERCONTEXT**

In the outermost scope of each function, it points to the `FUNCTION_DECL` node. It aids in better DWARF support of inline functions.

**CLASSTYPE\_TAGS**

`CLASSTYPE_TAGS` is a linked (via `TREE_CHAIN`) list of member classes of a class. `TREE_PURPOSE` is the name, `TREE_VALUE` is the type (pushclass scans these and calls `pushtag` on them.)

`finish_struct` scans these to produce `TYPE_DECLS` to add to the `TYPE_FIELDS` of the type.

It is expected that name found in the `TREE_PURPOSE` slot is unique, `resolve_scope_to_name` is one such place that depends upon this uniqueness.

**CLASSTYPE\_METHOD\_VEC**

The following is true after `finish_struct` has been called (on the class?) but not before. Before `finish_struct` is called, things are different to some extent. Contains a `TREE_VEC` of methods of the class. The `TREE_VEC_LENGTH` is the number of differently named methods plus one for the 0th entry. The 0th entry is always allocated, and reserved for ctors and dtors. If there are none, `TREE_VEC_ELT(N,0) == NULL_TREE`. Each entry of the `TREE_VEC` is a `FUNCTION_DECL`. For each `FUNCTION_DECL`, there is a `DECL_CHAIN` slot. If the `FUNCTION_DECL` is the last one with a given name, the `DECL_CHAIN` slot is `NULL_TREE`. Otherwise it is the next method that has the same name (but a different signature). It would seem that it is not true that because the `DECL_CHAIN` slot is used in this way, we cannot call `pushdecl` to put the method in the global scope (cause that would overwrite the `TREE_CHAIN` slot), because they use different `_CHAINS`. `finish_struct_methods` setups up one version of the `TREE_CHAIN` slots on the `FUNCTION_DECLS`.

friends are kept in `TREE_LISTs`, so that there's no need to use their `TREE_CHAIN` slot for anything.

Has values of:

TREE\_VECs

#### CLASSTYPE\_VFIELD

Seems to be in the process of being renamed TYPE\_VFIELD. Use on types to get the main virtual function table pointer. To get the virtual function table use BINFO\_VTABLE (TYPE\_BINFO ()).

Has values of:

FIELD\_DECLS that are virtual function table pointers

What things can this be used on:

RECORD\_TYPES

#### DECL\_CLASS\_CONTEXT

Identifies the context that the \_DECL was found in. For virtual function tables, it points to the type associated with the virtual function table. See also DECL\_CONTEXT, DECL\_FIELD\_CONTEXT and DECL\_FCONTEXT.

The difference between this and DECL\_CONTEXT, is that for virtuals functions like:

```
struct A
{
    virtual int f ();
};

struct B : A
{
    int f ();
};

DECL_CONTEXT (A::f) == A
DECL_CLASS_CONTEXT (A::f) == A

DECL_CONTEXT (B::f) == A
DECL_CLASS_CONTEXT (B::f) == B
```

Has values of:

RECORD\_TYPES, or UNION\_TYPES

What things can this be used on:

TYPE\_DECLS, \_DECLs

#### DECL\_CONTEXT

Identifies the context that the \_DECL was found in. Can be used on virtual function tables to find the type associated with the virtual function table, but since they are FIELD\_DECLS, DECL\_FIELD\_CONTEXT is a better access method. Internally the same as DECL\_FIELD\_CONTEXT, so don't use both. See also DECL\_FIELD\_CONTEXT, DECL\_FCONTEXT and DECL\_CLASS\_CONTEXT.

Has values of:

RECORD\_TYPES

What things can this be used on:

VAR\_DECLs that are virtual function tables  
\_DECLs

#### DECL\_FIELD\_CONTEXT

Identifies the context that the FIELD\_DECL was found in. Internally the same as DECL\_CONTEXT, so don't use both. See also DECL\_CONTEXT, DECL\_FCONTEXT and DECL\_CLASS\_CONTEXT.

Has values of:

RECORD\_TYPEs

What things can this be used on:

FIELD\_DECLs that are virtual function pointers  
FIELD\_DECLs

#### DECL\_NESTED\_TYPENAME

Holds the fully qualified type name. Example, Base::Derived.

Has values of:

IDENTIFIER\_NODEs

What things can this be used on:

TYPE\_DECLs

#### DECL\_NAME

Has values of:

0 for things that don't have names  
IDENTIFIER\_NODEs for TYPE\_DECLs

#### DECL\_IGNORED\_P

A bit that can be set to inform the debug information output routines in the backend that a certain \_DECL node should be totally ignored.

Used in cases where it is known that the debugging information will be output in another file, or where a sub-type is known not to be needed because the enclosing type is not needed.

A compiler constructed virtual destructor in derived classes that do not define an explicit destructor that was defined explicit in a base class has this bit set as well. Also used on \_\_FUNCTION\_\_ and \_\_PRETTY\_FUNCTION\_\_ to mark they are "compiler generated." c-decl and c-lex.c both want DECL\_IGNORED\_P set for "internally generated vars," and "user-invisible variable."

Functions built by the C++ front-end such as default destructors, virtual destructors and default constructors want to be marked that they are compiler generated, but unsure why.

Currently, it is used in an absolute way in the C++ front-end, as an optimization, to tell the debug information output routines to not generate debugging information that will be output by another separately compiled file.

**DECL\_VIRTUAL\_P**

A flag used on FIELD\_DECLs and VAR\_DECLs. (Documentation in tree.h is wrong.) Used in VAR\_DECLs to indicate that the variable is a vtable. It is also used in FIELD\_DECLs for vtable pointers.

What things can this be used on:

FIELD\_DECLs and VAR\_DECLs

**DECL\_VPARENT**

Used to point to the parent type of the vtable if there is one, else it is just the type associated with the vtable. Because of the sharing of virtual function tables that goes on, this slot is not very useful, and is in fact, not used in the compiler at all. It can be removed.

What things can this be used on:

VAR\_DECLs that are virtual function tables

Has values of:

RECORD\_TYPEs maybe UNION\_TYPEs

**DECL\_FCONTEXT**

Used to find the first baseclass in which this FIELD\_DECL is defined. See also DECL\_CONTEXT, DECL\_FIELD\_CONTEXT and DECL\_CLASS\_CONTEXT.

How it is used:

Used when writing out debugging information about vfield and vbase decls.

What things can this be used on:

FIELD\_DECLs that are virtual function pointers FIELD\_DECLs

**DECL\_REFERENCE\_SLOT**

Used to hold the initialize for the reference.

What things can this be used on:

PARAM\_DECLs and VAR\_DECLs that have a reference type

**DECL\_VINDEX**

Used for FUNCTION\_DECLs in two different ways. Before the structure containing the FUNCTION\_DECL is laid out, DECL\_VINDEX may point to a FUNCTION\_DECL in a base class which is the FUNCTION\_DECL which this FUNCTION\_DECL will replace as a virtual function. When the class is laid out, this pointer is changed to an INTEGER\_CST node which is suitable to find an index into the virtual function table. See get\_vtable\_entry as to how one can find the right index into the virtual function table. The first index 0, of a virtual function table it not used in the normal way, so the first real index is 1.

DECL\_VINDEX may be a TREE\_LIST, that would seem to be a list of overridden FUNCTION\_DECLs. add\_virtual\_function has code to deal with this when it uses the variable base\_fnDECL\_list, but it would seem that somehow, it is possible for the TREE\_LIST to persist until method\_call, and it should not.

What things can this be used on:

FUNCTION\_DECLs



**DECL\_SOURCE\_FILE**

Identifies what source file a particular declaration was found in.

Has values of:

"<built-in>" on TYPE\_DECLs to mean the typedef is built in

**DECL\_SOURCE\_LINE**

Identifies what source line number in the source file the declaration was found at.

Has values of:

0 for an undefined label

0 for TYPE\_DECLs that are internally generated

0 for FUNCTION\_DECLs for functions generated by the compiler  
(not yet, but should be)

0 for “magic” arguments to functions, that the user has no  
control over

**TREE\_USED**

Has values of:

0 for unused labels

**TREE\_ADDRESSABLE**

A flag that is set for any type that has a constructor.

**TREE\_COMPLEXITY**

They seem a kludge way to track recursion, popping, and pushing. They only appear in cp-decl.c and cp-decl2.c, so they are a good candidate for proper fixing, and removal.

**TREE\_PRIVATE**

Set for FIELD\_DECLs by finish\_struct. But not uniformly set.

The following routines do something with PRIVATE visibility:  
build\_method\_call, alter\_visibility, finish\_struct\_methods, finish\_struct,  
convert\_to\_aggr, CWriteLanguageDecl, CWriteLanguageType, CWriteUseObject,  
compute\_visibility, lookup\_field, dfs\_pushdecl, GNU\_xref\_member,  
dbxout\_type\_fields, dbxout\_type\_method\_1

**TREE\_PROTECTED**

The following routines do something with PROTECTED visibility:  
build\_method\_call, alter\_visibility, finish\_struct, convert\_to\_aggr, CWriteLanguageDecl,  
CWriteLanguageType, CWriteUseObject, compute\_visibility,  
lookup\_field, GNU\_xref\_member, dbxout\_type\_fields, dbxout\_type\_method\_1

**TYPE\_BINFO**

Used to get the binfo for the type.

Has values of:

TREE\_VECs that are binfos

What things can this be used on:

RECORD\_TYPES

TYPE\_BINFO\_BASETYPES

See also BINFO\_BASETYPES.

TYPE\_BINFO\_VIRTUALS

A unique list of functions for the virtual function table. See also BINFO\_VIRTUALS.

What things can this be used on:

RECORD\_TYPES

TYPE\_BINFO\_VTABLE

Points to the virtual function table associated with the given type. See also BINFO\_VTABLE.

What things can this be used on:

RECORD\_TYPES

Has values of:

VAR\_DECLS that are virtual function tables

TYPE\_NAME

Names the type.

Has values of:

0 for things that don't have names.

should be IDENTIFIER\_NODE for RECORD\_TYPES UNION\_TYPES and ENUM\_TYPES.

TYPE\_DECL for RECORD\_TYPES, UNION\_TYPES and ENUM\_TYPES, but shouldn't be.

TYPE\_DECL for typedefs, unsure why.

What things can one use this on:

TYPE\_DECLS

RECORD\_TYPES

UNION\_TYPES

ENUM\_TYPES

History:

It currently points to the TYPE\_DECL for RECORD\_TYPES, UNION\_TYPES and ENUM\_TYPES, but it should be history soon.

TYPE\_METHODS

Synonym for CLASSTYPE\_METHOD\_VEC. Chained together with TREE\_CHAIN. dbxout.c uses this to get at the methods of a class.

TYPE\_DECL

Used to represent typedefs, and used to represent bindings layers.

Components:

DECL\_NAME is the name of the typedef. For example, foo would be found in the DECL\_NAME slot when `typedef int foo;` is seen.

DECL\_SOURCE\_LINE identifies what source line number in the source file the declaration was found at. A value of 0 indicates that this TYPE\_DECL is just an internal binding layer marker, and does not correspond to a user supplied typedef.

DECL\_SOURCE\_FILE

#### TYPE\_FIELDS

A linked list (via TREE\_CHAIN) of member types of a class. The list can contain TYPE\_DECLS, but there can also be other things in the list apparently. See also CLASSTYPE\_TAGS.

#### TYPE\_VIRTUAL\_P

A flag used on a FIELD\_DECL or a VAR\_DECL, indicates it is a virtual function table or a pointer to one. When used on a FUNCTION\_DECL, indicates that it is a virtual function. When used on an IDENTIFIER\_NODE, indicates that a function with this same name exists and has been declared virtual.

When used on types, it indicates that the type has virtual functions, or is derived from one that does.

Not sure if the above about virtual function tables is still true. See also info on DECL\_VIRTUAL\_P.

What things can this be used on:

FIELD\_DECLS, VAR\_DECLS, FUNCTION\_DECLS, IDENTIFIER\_NODES

#### VF\_BASETYPE\_VALUE

Get the associated type from the binfo that caused the given vfield to exist. This is the least derived class (the most parent class) that needed a virtual function table. It is probably the case that all uses of this field are misguided, but they need to be examined on a case-by-case basis. See history for more information on why the previous statement was made.

What things can this be used on:

TREE\_LISTs that are vfields

History:

This field was used to determine if a virtual function table's slot should be filled in with a certain virtual function, by checking to see if the type returned by VF\_BASETYPE\_VALUE was a parent of the context in which the old virtual function existed. This incorrectly assumes that a given type `_could_` not appear as a parent twice in a given inheritance lattice. For single inheritance, this would in fact work, because a type could not possibly appear more than once in an inheritance lattice, but with multiple inheritance, a type can appear more than once.

#### VF\_BINFO\_VALUE

Identifies the binfo that caused this vfield to exist. Can use TREE\_VIA\_VIRTUAL on result to find out if it is a virtual base class. Related to the binfo found by

```
get_binfo (VF_BASETYPE_VALUE (vfield), t, 0)
```

where 't' is the type that has the given vfield.

```
get_binfo (VF_BASETYPE_VALUE (vfield), t, 0)
```

will return the binfo for the the given vfield.

May or may not be set at `modify_vtable_entries` time. Set at `finish_base_struct` time.

What things can this be used on:

TREE\_LISTs that are vfields

#### VF\_DERIVED\_VALUE

Identifies the type of the most derived class of the vfield, excluding the the class this vfield is for.

What things can this be used on:

TREE\_LISTs that are vfields

#### VF\_NORMAL\_VALUE

Identifies the type of the most derived class of the vfield, including the class this vfield is for.

What things can this be used on:

TREE\_LISTs that are vfields

#### WRITABLE\_VTABLES

This is a option that can be defined when building the compiler, that will cause the compiler to output vtables into the data segment so that the vtables maybe written. This is undefined by default, because normally the vtables should be unwritable. People that implement object I/O facilities may, or people that want to change the dynamic type of objects may want to have the vtables writable. Another way of achieving this would be to make a copy of the vtable into writable memory, but the drawback there is that that method only changes the type for one object.

## 1.6 Typical Behavior

Whenever seemingly normal code fails with errors like `syntax error at `\'`, it's highly likely that `grokdeclarator` is returning a `NULL_TREE` for whatever reason.

## 1.7 Coding Conventions

It should never be that case that trees are modified in-place by the back-end, *unless* it is guaranteed that the semantics are the same no matter how shared the tree structure is. `fold-const.c` still has some cases where this is not true, but rms hypothesizes that this will never be a problem.

## 1.8 Error Reporting

The C++ frontend uses a call-back mechanism to allow functions to print out reasonable strings for types and functions without putting extra logic in the functions where errors are found. The interface is through the `lang_error` function (or `lang_warning`, etc.). The syntax is exactly like that of `error`, except that a few more conversions are supported:

- `%D` indicates a `*_DECL` node
- `%T` indicates a `*_TYPE` node

- %E indicates a \*\_EXPR node (currently only used for default parameters in FUNCTION\_DECLS)

For a more verbose message (`class foo` as opposed to just `foo`, including the return type for functions), use %`#c`. To have the line number on the error message indicate the line of the DECL, use `lang_error_at` and its ilk.

## 1.9 Concept Index

### D

delete, two argument ..... 1

### P

parse errors ..... 12  
pointers to members ..... 1  
pushdecl\_class\_level ..... 1

### V

visibility checking ..... 1  
volatile ..... 1