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.
- Access checking is unimplemented for nested types.
- `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.
- `this` will be wrong in virtual members functions defined in a virtual base class, when they are overridden in a derived class, when called via a non-left most object.

  An example would be:

```
extern "C" int printf(const char*, ...);
struct A { virtual void f() { } };
struct B : virtual A { int b; B() : b(0) {} void f() { b++; } };
struct C : B {};
struct D : B {};
struct E : C, D {};
int main()
{
  E e;
  C& c = e; D& d = e;
  c.f(); d.f();
  printf ("C::b = %d, D::b = %d\n", e.C::b, e.D::b);
  return 0;
}
```

  This will print out 2, 0, instead of 1,1.

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_OFFSET`s of zero are shared and will have themselves as their own `MAIN_VARIANT`s. The non-shared right ones, are copies of the left-most one, hence if it is its own `MAIN_VARIENT`, 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 access.
- `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. The virtual function table from a virtual base class is never reused in a derived class. `is_normal` depends upon this.

virtual function table pointer

These are `FIELD_DECL`s 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 understnad from of the more complex data structures, contact Mike Stump (`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 anf 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 is the above is really true, get_base_distance has is 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 us 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 us 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 back-end 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 exlicit destructor that was defined exlicit 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 desctructors 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:

PARM_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 pursist 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, poping, and pushing. They only appear in cp-decl.c and cp-decl2.c, so the 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 access: build_method_call, alter_access, finish_struct_methods, finish_struct, convert_to_aggr, CWriteLanguageDecl, CWriteLanguageType, CWriteUseObject, compute_access,

lookup_field, dfs_pushdecl, GNU_xref_member, dbxout_type_fields, dbxout_type_method_1

`TREE_PROTECTED`

The following routines do something with PROTECTED access: build_method_call, alter_access, finish_struct, convert_to_aggr, CWriteLanguageDecl, CWriteLanguageType, CWriteUseObject, compute_access, 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 suppiled typedef.

DECL_SOURCE_FILE

`TYPE_FIELDS`

A linked list (via `TREE_CHAIN`) of member types of a class. The list can contain `TYPE_DECL`s, 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.

Set at `finish_base_struct` time.

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. If this vfield is from the first direct base class that has a virtual function table, then VF_BINFO_VALUE is NULL_TREE, otherwise it will be the binfo of the direct base where the vfield came from. 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.

Set at `finish_base_struct` time.

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.

Set at `finish_base_struct` time.

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 Templates

g++ uses the simple approach to instantiating templates: it blindly generates the code for each instantiation as needed. For class templates, g++ pushes the template parameters into the namespace for the duration of the instantiation; for function templates, it's a simple search and replace.

This approach does not support any of the template definition-time error checking that is being bandied about by X3J16. It makes no attempt to deal with name binding in a consistent way.

Instantiation of a class template is triggered by the use of a template class anywhere but in a straight declaration like `class A<int>`. This is wrong; in fact, it should not be triggered by typedefs or declarations of pointers. Now that explicit instantiation is supported, this misfeature is not necessary.

Important functions:

`instantiate_class_template`
: This function

1.9 Access Control

The function compute_access returns one of three values:

`access_public`
: means that the field can be accessed by the current lexical scope.

`access_protected`
: means that the field cannot be accessed by the current lexical scope because it is protected.

`access_private`
: means that the field cannot be accessed by the current lexical scope because it is private.

DECL_ACCESS is used for access declarations; alter_access creates a list of types and accesses for a given decl.

Formerly, DECL_{PUBLIC,PROTECTED,PRIVATE} corresponded to the return codes of compute_access and were used as a cache for compute_access. Now they are not used at all.

TREE_PROTECTED and TREE_PRIVATE are used to record the access levels granted by the containing class. BEWARE: TREE_PUBLIC means something completely unrelated to access control!

1.10 Error Reporting

The C++ front-end 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 `cp_error` function (or `cp_warning`, etc.). The syntax is exactly like that of `error`, except that a few more conversions are supported:

- %C indicates a value of 'enum tree_code'.
- %D indicates a *_DECL node.
- %E indicates a *_EXPR node.
- %L indicates a value of 'enum languages'.
- %P indicates the name of a parameter (i.e. "this", "1", "2", ...)
- %T indicates a *_TYPE node.
- %O indicates the name of an operator (MODIFY_EXPR -> "operator =").

There is some overlap between these; for instance, any of the node options can be used for printing an identifier (though only `%D` tries to decipher function names).

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 `cp_error_at` and its ilk; to indicate which argument you want, use `%+D`, or it will default to the first.

1.11 Parser

Some comments on the parser:

The `after_type_declarator` / `notype_declarator` hack is necessary in order to allow redeclarations of `TYPENAME`s, for instance

```
typedef int foo;
class A {
  char *foo;
};
```

In the above, the first `foo` is parsed as a `notype_declarator`, and the second as a `after_type_declarator`.

Ambiguities:

There are currently four reduce/reduce ambiguities in the parser. They are:

1) Between `template_parm` and `named_class_head_sans_basetype`, for the tokens `aggr identifier`. This situation occurs in code looking like

```
template <class T> class A { };
```

It is ambiguous whether `class T` should be parsed as the declaration of a template type parameter named `T` or an unnamed constant parameter of type `class T`. Section 14.6, paragraph 3 of the January '94 working paper states that the first interpretation is the correct one. This ambiguity results in two reduce/reduce conflicts.

2) Between `primary` and `type_id` for code like '`int()`' in places where both can be accepted, such as the argument to `sizeof`. Section 8.1 of the pre-San Diego working paper specifies that these ambiguous constructs will be interpreted as `typename`s. This ambiguity results in six reduce/reduce conflicts between '`absdcl`' and '`functional_cast`'.

3) Between `functional_cast` and `complex_direct_notype_declarator`, for various token strings. This situation occurs in code looking like

```
int (*a);
```

This code is ambiguous; it could be a declaration of the variable '`a`' as a pointer to '`int`', or it could be a functional cast of '`*a`' to '`int`'. Section 6.8 specifies that the former interpretation is correct. This ambiguity results in 7 reduce/reduce conflicts. Another aspect of this ambiguity is code like 'int (x[2]);', which is resolved at the '[' and accounts for 6 reduce/reduce conflicts between '`direct_notype_declarator`' and '`primary`'/'`overqualified_id`'. Finally, there are 4 r/r conflicts between '`expr_or_declarator`' and '`primary`' over code like 'int (a);', which could probably be resolved but would also probably be more trouble than it's worth. In all, this situation accounts for 17 conflicts. Ack!

The second case above is responsible for the failure to parse 'LinppFile ppfile (String (argv[1]), &outs, argc, argv);' (from Rogue Wave Math.h++) as an object declaration, and must be fixed so that it does not resolve until later.

4) Indirectly between `after_type_declarator` and `parm`, for type names. This occurs in (as one example) code like

```
typedef int foo, bar;
class A {
  foo (bar);
};
```

What is `bar` inside the class definition? We currently interpret it as a `parm`, as does Cfront, but IBM xlC interprets it as an `after_type_declarator`. I believe that xlC is correct, in light of 7.1p2, which says "The longest sequence of *decl-specifiers* that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*." However, it seems clear that this rule must be violated in the case of constructors. This ambiguity accounts for 8 conflicts.

Unlike the others, this ambiguity is not recognized by the Working Paper.

1.12 Copying Objects

The generated copy assignment operator in g++ does not currently do the right thing for multiple inheritance involving virtual bases; it just calls the copy assignment operators for its direct bases. What it should probably do is:

1) Split up the copy assignment operator for all classes that have vbases into "copy my vbases" and "copy everything else" parts. Or do the trickiness that the constructors do to ensure that vbases don't get initialized by intermediate bases.

2) Wander through the class lattice, find all vbases for which no intermediate base has a user-defined copy assignment operator, and call their "copy everything else" routines. If not all of my vbases satisfy this criterion, warn, because this may be surprising behavior.

3) Call the "copy everything else" routine for my direct bases.

If we only have one direct base, we can just foist everything off onto them.

This issue is currently under discussion in the core reflector (2/28/94).

1.13 Exception Handling

Note, exception handling in g++ is still under development.

This section describes the mapping of C++ exceptions in the C++ front-end, into the back-end exception handling framework.

The basic mechanism of exception handling in the back-end is unwind-protect a la elisp. This is a general, robust, and language independent representation for exceptions.

The C++ front-end exceptions are mapping into the unwind-protect semantics by the C++ front-end. The mapping is describe below.

Objects with RTTI support should use the RTTI information to do mapping and checking. Objects without RTTI, like int and const char *, have to use another means of matching. Currently we use the normal mangling used in building functions names. Int's are `"i"`, const char * is PCc, etc...

Unfortunately, the standard allows standard type conversions on throw parameters so they can match catch handlers. This means we need a mechanism to handle type conversion at run time, ICK. I read this part again, and it appears that we only have to be able to do a few of the conversions at run time, so we should be ok.

In C++, all cleanups should be protected by exception regions. The region starts just after the reason why the cleanup is created has ended. For example, with an automatic variable, that has a constructor, it would be right after the constructor is run. The region ends just before the finalization is expanded. Since the backend may expand the cleanup multiple times along different paths, once for normal end of the region, once for non-local gotos, once for returns, etc, the backend must take special care to protect the finalization expansion, if the expansion is for any other reason than normal region end, and it is 'inline' (it is inside the exception region). The backend can either choose to move them out of line, or it can created an exception region over the finalization to protect it, and in the handler associated with it, it would not run the finalization as it otherwise would have, but rather just rethrow to the outer handler, careful to skip the normal handler for the original region.

In Ada, they will use the more runtime intensive approach of having fewer regions, but at the cost of additional work at run time, to keep a list of things that need cleanups. When a variable has finished construction, they add the cleanup to the list, when the come to the end of the lifetime of the variable, the run the list down. If the take a hit before the section finishes normally, they examine the list for actions to perform. I hope they add this logic into the back-end, as it would be nice to get that alternative approach in C++.

On an rs6000, xlC stores exception objects on that stack, under the try block. When is unwinds down into a handler, the frame pointer is adjusted back to the normal value for the frame in which the handler resides, and the stack pointer is left unchanged from the time at which the object was throwed. This is so that there is always someplace for the exception object, and nothing can overwrite it, once we start throwing. The only bad part, is that the stack remains large.

Flaws in g++'s exception handling. The stack pointer is restored from stack, we want to match rs6000, and propagate the stack pointer from time of throw, down, to the catch place.

Only exact type matching of throw types works (references work also), catch variables cannot be used. Only works on a Sun sparc running SunOS 4.1.x. Unwinding to outer catch clauses works. All temps and local variables are cleaned up in all unwinded scopes. Completed parts of partially constructed objects are not cleaned up. Don't expect exception handling to work right if you optimize, in fact the compiler will probably core dump. If two EH regions are the exact same size, the backend cannot tell which one is first. It punts by picking the last one, if they tie. This is usually right. We really should stick in a nop, if they are the same size.

If we fall off the end of a series of catch blocks, we return to the flow of control in a normal fasion. But this is wrong, we should rethrow.

When we invoke the copy constructor for an exception object because it is passed by value, and if we take a hit (exception) inside the copy constructor someplace, where do we go? I have tentatively choosen to not catch throws by the outer block at the same unwind level, if one exists, but rather to allow the frame to unwind into the next series of handlers, if any. If this is the wrong way to do it, we will need to protect the rest of the handler in some fashion. Maybe just changing the handler's handler to protect the whole series of handlers is the right way to go.

The EH object is copied like it should be, if it is passed by value, otherwise we get a reference directly to it.

EH objects make it through unwinding, but are subject to being overwritten as they are still past the top of stack. Don't throw automatic objects if this is a problem.

Exceptions in catch handlers now go to outer block.

1.14 Free Store

operator new [] adds a magic cookie to the beginning of arrays for which the number of elements will be needed by operator delete []. These are arrays of objects with destructors and arrays of objects that define operator delete [] with the optional size_t argument. This cookie can be examined from a program as follows:

```
typedef unsigned long size_t;
extern "C" int printf (const char *, ...);

size_t nelts (void *p)
{
  struct cookie {
    size_t nelts __attribute__ ((aligned (sizeof (double))));
  };

  cookie *cp = (cookie *)p;
  --cp;

  return cp->nelts;
}
```

```
struct A {
  ~A() { }
};

main()
{
  A *ap = new A[3];
  printf ("%ld\n", nelts (ap));
}
```

1.15 Concept Index