Stree design notes

Matt Austern, Robert Bowdidge, Geoff Keating

The stree project is based on three fundamental premises. First: for an important class of development tasks (roughly: GUI programs written in a relatively simple subset of C++, compiled at –O0 –g), compilation time is dominated by the C++ front end. Second: the performance of the C++ front end is dominated by memory allocation and management. This includes memory allocation, initializing newly allocating objects, and bookkeeping for garbage collection. Reducing front end memory usage should thus improve front end performance. Third: many programs consist of small source files that include truly enormous header files. Such header files include <iostream> (25,000 lines), Apple's <Carbon/Carbon.h> (91,000 lines), and the X11 headers. Any given translation unit only uses a tiny fraction of the declarations in one of these headers.

The goal of this project is to reduce the time and memory required for handling unused declarations.

Basic design principles

The main idea of the stree project is to avoid generating decl trees when possible. Instead the parser will generate a compact flat representation for declarations, called an stree, and expand the stree to a decl tree when necessary. Strees are not a substitute for trees. The middle-end and back end will still understand trees, not strees.

Some immediate implications of this basic idea:

- Trees and strees will always coexist. This means that it is acceptable for the parser to generate strees only in simple and common cases, and to fall back to decl trees in more complicated cases. We can add incrementally add cases where we are able to generate strees.
- Usually we generate strees only for declarations that are not also definitions. So, for example, we would generate an stree for "void do_nothing();"(which the middle end and back end don't necessarily have to know about), but we would generate a full tree for "void do_nothing() { }" (which the middle end has to expand to RTL).
- For a declaration that is not a definition, there is a simple way to characterize whether or not the definition is "needed": some other declaration refers to it by name. For example: if a function "xyzzy" is declared but nobody ever defines it, takes its address, or calls a function with that name, then that declaration isn't needed, and generating a decl tree for it is wasted time and space.
- This definition of "needed" immediately leads to an implementation technique. References to a decl by name always go through a cxx_binding object. (See name_lookup.h). So we just need to make cxx_binding a little bit more complicated: when we ask a cxx_binding for the value of the binding, we check to

see whether we have a tree or an stree. If the latter, we expand it into a tree and cache the expanded version.

- Given this definition of whether a declaration is "needed", we have to be careful about putting decls on global lists. This work will be useless if we end up expanding all decls anyway.
- Error checking must all be done in the initial parse phase, while generating the stree, and not as part of stree-to-tree expansion. Rationale: we always need error checking, but not all strees will be expanded. (There is also an implementation reason why we don't want to defer emission of diagnostics. Early diagnostics reduce the need for global state to be remembered for each stree: source code position, current_binding_level, current_class_ptr, etc.)

An example: enumerators

Consider the front end data structure for a simple enumeration declaration, "enum foo { a, b };". We have two enumerators. For each one we need to know its name, its type, the underlying integer type used to represent it, and its value. At present we represent enumerators with CONST_DECL nodes, so each enumerator takes 128 bytes for the tree_decl node, plus additional memory for cp-tree.h's version of lang_decl.

Each enumerator has an entry in the hash table, an identifier. Each identifier has a pointer to a binding of type cxx_binding (this is the bindings field in lang_identifier, defined in name_lookup.h). The binding for "foo" itself points to a tree_type, and the bindings for "a" and "b" point to CONST_DECL nodes. Each CONST_DECL node has pointers to the name and to the ENUMERAL_TYPE node, and additionally has a pointer to a node representing the enumerator's value. In simple examples (like this one) each enumerator's value is an INTEGER_CST, giving us another 36 bytes each. (An INTEGER_CST node contains tree_common as a substructure, with all the generality that implies.)

We don't need 200 bytes to represent the fact that the enumerator "a" has the value 0. First: as an stree it's unnecessary to store a pointer to the name of this enumerator. The stree will only be accessed via a cxx_binding, so any code that accesses the stree already knows the name. Second: it isn't necessary to use anything so large as an INTEGER_CST to represent the value "0". Most of the information stored in an INTEGER_CST (chain nodes, type pointers, etc.) is unnecessary, since we already know we're getting to the value through an enumerator. We only need to store two pieces of information: the enumeration that this enumerator is associated with, and its initial value. This allows us to represent the enumerator in six bytes: a one-byte code for the type of the stree (specifically: the TREE_CODE of the tree that this stree corresponds to), four bytes (a pointer or the equivalent) for the enumeration, and one byte for the value. (Note that this implies a variable-width encoding for the integer values; some enumerations will require seven or more bytes.)

Our current implementation is limited to enumerations defined at namespace scope. First, enumerations defined at class scope require additional context information. Second, enumerators declared at class scope might have values that depend on template parameters, meaning that we can't necessarily represent the values as simple integers. Neither is a serious problem. Because a cxx_binding's value can be either a tree or stree, we can use strees for the common, simple cases, and default to trees otherwise. Because strees are a variable-sized representation, we can add additional values needed for building trees for the complex case as needed without bloating the simpler cases.

Some implementation details

The stree data structure itself is defined in stree.[ch]. Strees are tightly-packed, serialized representations of simple declarations.

Strees are stored on the gc heap, but not directly: instead, they are stored in multi-page blocks of virtual memory ("chunks"), where a single chunk may contain multiple strees. Each stree is represented by an index; a separate table maps each index to the appropriate chunk and position within that chunk. We thus don't traffic in pointers to strees, but rather in integer indices referencing a location in memory. Storing strees in this manner avoids creating new objects and additional work for the garbage collector, and simplifies precompiled headers by ensuring that the chunks don't need to be placed at a specific address or when reloaded -- only the table pointers need to be swizzled.

Clients access stree data via an *iterator*: given an stree with index s, the function get_s_tree_iter (declared in stree.h) creates an iterator pointing to the beginning of s. Other functions declared in stree.h access the iterator to extract each serialized value in turn. This scheme allows us to store data in the most compressed representation possible, and in a way such that clients are insulated from the details of the representation. For enumerators, for example, instead of using a full INTEGER_CST for each value, we can use one or two bytes in the (typical) case where the values are small.

Strees are created with build_s_tree, a varargs function defined stree.c. Its first argument is the stree code, and its remaining arguments are the contents of that stree and tags to identify their types. There is no function for creating an stree by treating it as a “stream” to which values are written one at a time; eventually there probably will need to be one. It won't be hard to add it.

Stree.h and stree.c are language-independent, since, at bottom, strees are just a way of packing bytes and integers into chunks. Creation and expansion of strees are language dependent. The present implementation is focused on C++.

We change cxx_binding::value from type “tree” to type “s_tree_i_or_tree” (a tagged union), and we change IDENTIFIER_VALUE so that it returns the tree value, expanding the stree if necessary. A few changes are required in functions that manipulate cxx_binding directly, but those changes are largely mechanical and are localized to cp/name_lookup.[ch].

Strees are expanded by the s_tree_to_tree function, defined in cp/decl.c. There are three points to notice about it. First, as described above, it uses the stree iterator interface. Second, the first byte of the stree is the stree code; s_tree_to_tree uses that code to determine what kind of tree to create. Third, at present s_tree_to_tree doesn't handle any cases other than enumerators.

The major changes required to use strees for enumerators are in build_enumerator. First, we need to separate parsing and error checking from tree generation, deferring the latter until later. Second, for simple cases we use build_s_tree to create the stree and push_s_decl to enter the stree into the current lexical scope. In principle push_s_decl would need to know all of the same logic that pushdecl does; in practice we only use push_s_decl for the simplest cases, deferring to pushdecl (i.e. using trees instead of strees) for the more complicated cases.

This design has the virtue that most of the C++ front end doesn't have to know about strees: code that goes through binding to get trees looks exactly as before. It has the defect that, as presently written, it requires code duplication. The code required to generate an enumerator node is in both build_enumerator and s_tree_to_tree. Additionally, s_tree_to_tree is manageable only because at the moment it only handles a single case. If this project succeeds, and we're handling many kinds of strees, it would become a monstrosity. The right solution will probably be to replace s_tree_to_tree with a wrapper function that examines the stree code and dispatches to a function for the appropriate code, and, for each code, to write an implementation function that's shared between the tree and stree versions. Similarly, we can probably achieve better code sharing between pushdecl and push_s_decl.

Debugging information

At present the compiler will not generate debugging information for unexpanded strees. This is potentially a serious issue. In principle, there are two ways of dealing with this issue: either figure out a way to generate debugging information without expanding strees, or else decide that it's acceptable to omit debugging information for "unused" declarations. (Note that by "unused", we mean declarations that are irrelevant to the compilation of the code rather than "never executed". As soon as a declaration's name is seen elsewhere in the code, we create a decl tree node for the name.)

We don't believe this will be a serious problem. Consider the effect of missing debug information for unused declarations:

- Unused variables: Only variable declarations are affected; variable definitions require code to be generated, and so will be expanded. If the definition occurs in another translation unit, we can get debug information from there. If the variable declaration is completely unused, we can't do anything with it from the debugger, for it has no memory associated with it.
- Unused functions: ditto. Again, the existence of definition or a function call counts as a use, so we will get debug info even if the function is defined in another translation unit. There's nothing a user can do in the debugger with a function that's never defined anywhere in the program.
- Unused classes and structures: at best, an unused class or structure is uninteresting because no instances of the object could have been created. If the instances exist in another translation unit, the debugging information in that translation unit will

be used by the debugger. The only case where the missing information may be desired is if a programmer wants to cast a pointer to an unused type and view the memory.

- Unused enums: like structures and types, unused enums are uninteresting because no uses could exist if the enum were unused. Like types and structs, programmers may want to see the value of an unused enum. Without additional work, strees would not provide enough debugging information for a programmer to see all enum values. However, that additional work won't be difficult if we decide that unused enums are important.

More importantly, some gcc versions already remove unneeded declarations from debug information. GCC 3.4 does not generated DWARF debug info for function declarations, and does not generate debug info for used types unless -fno-eliminate-unused-debug-types is specified. Apple's gcc has stripped "unused" symbols out of STABS debugging format for the last two and a half years. The debugger team expected many bugs from users trying to examine unused declarations, but have been surprised at how few bugs they've received. One of the few complaints was from a user who had a "debug" version of a struct that was used only for pretty-printing the real structure, and was stripped out because it was never actually referenced.

To do

- Expand to other kinds of declarations: enumerations inside classes, enumeration types, function declarations, and class declarations. We suspect that our biggest performance win with Apple's source code will occur if we can lazily create class declarations, because measurements suggest that creation of implicit constructors for unused POD classes appears to take significant time during our compiles.
- Refactor the stree creation and expansion code in cp/decl.c to avoid code duplication. Or, to put it differently: develop a general framework for separating the early phase (parsing and error checking) from the later stages of tree generation and transformation.
- Separate out the checks required for user declarations from those for compiler-generated declarations.
- Get rid of the tree_or_stree struct. Instead just us an ordinary union, and steal one of the unused bits in cxx_binding to identify which member we want.
- Eliminate remaining global scans of decls, to make sure that we don't expand strees unnecessarily.