# A tiny GCC front end – Part 10

Sep 4, 2016 • Roger Ferrer Ibáñez • compilers, GCC

Today we will add a relatively simple feature that will be very useful for a future extension: type declarations.

## Variable declarations

Our current version of tiny has the concept of variable declaration, where a name is introduced in the program to represent a variable value type.

⟨declaration⟩ → **var** ⟨identifier⟩ **:** ⟨type⟩ **;**

For example, in the code below:

```
var x : int;
x := 3;
```

After this variable declaration, the name `x` can be used as a variable: inside an expression or in the left hand side of an assignment.

But what if we were able to declare things that are not only variables, like types? Associate to a name a type so we can use the type where a type is expected? This is what we are doing today: we are introducing type declarations.

## Syntax

First we will generalize a bit the ⟨declaration⟩ rules to encompass ⟨variable-declaration⟩ and ⟨type-declaration⟩.

⟨declaration⟩ → ⟨variable-declaration⟩ | ⟨type-declaration⟩

Now we can define the syntax of `type-declarations`.

⟨variable-declaration⟩ → **var** ⟨identifier⟩ **:** ⟨type⟩ **;**
⟨type-declaration⟩ → **type** ⟨identifier⟩ **:** ⟨type⟩ **;**

Since we want to be able to use the type declaration where types are specified, we need to extend our syntax for ⟨type⟩ (note the addition of ⟨identifier⟩ after `bool`).

⟨type⟩ → **`int`**
| **`float`**
| **`bool`**
| ⟨identifier⟩
| ⟨type⟩ **`[`** ⟨expression⟩ **`]`**
| ⟨type⟩ **`(`** ⟨expression⟩ **`:`** ⟨expression⟩ **`)`**

# Semantics

If a variable declaration introduce a *variable name*, a type declaration introduces a *type name*. The same rules we use for variable names apply for type declaration names. We need a few restrictions though. The ⟨identifier⟩ of a type declaration cannot appear in the ⟨type⟩ of its own ⟨type-declaration⟩ (e.g. `type T : T[10];` is not valid). A type name can only be used where a type is expected, this means that it cannot be used inside an expression or the left hand side of an assignment. Finally, a name can either be a variable name or a type name but not both.

The interpretation of using a type name inside a ⟨type⟩ is simple: it denotes the ⟨type⟩ of the corresponding ⟨type-declaration⟩ of that type name.

# Implementation

With all that knowledge, we can start implementing type names.

## Lexer

We are introducing a new token `type`. This is easy, we just add it to our list of token keywords.

```
diff --git a/gcc/tiny/tiny-token.h b/gcc/tiny/tiny-token.h
@@ -56,9 +56,11 @@ namespace Tiny
   TINY_TOKEN_KEYWORD (NOT, "not")                                    \
   TINY_TOKEN_KEYWORD (OR, "or")                                      \
   TINY_TOKEN_KEYWORD (READ, "read")                                  \
   TINY_TOKEN_KEYWORD (THEN, "then")                                  \
   TINY_TOKEN_KEYWORD (TO, "to")                                      \
   TINY_TOKEN_KEYWORD (TRUE_LITERAL, "true")                          \
+  TINY_TOKEN_KEYWORD (TYPE, "type")                                  \
   TINY_TOKEN_KEYWORD (VAR, "var")                                    \
   TINY_TOKEN_KEYWORD (WHILE, "while")                                \
   TINY_TOKEN_KEYWORD (WRITE, "write")                                \
```

Our existing lexer machinery will do the rest.

# Parser

This part is as usual a bit more involved. First we need to recognize a new declaration.

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
@@ -136,6 +137,7 @@ public:
   Tree parse_statement ();

   Tree parse_variable_declaration ();
+  Tree parse_type_declaration ();
```

When parsing a statement, if we see a token `type` it means that a type-declaration starts.

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
@@ -388,6 +390,9 @@ Parser::parse_statement ()
     case Tiny::VAR:
       return parse_variable_declaration ();
       break;
+    case Tiny::TYPE:
+      return parse_type_declaration ();
+      break;
```

The implementation is pretty straightforward...

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
@@ -474,6 +479,64 @@ Parser::parse_variable_declaration ()
   return stmt;
 }

+Tree
+Parser::parse_type_declaration ()
+{
+  // type_declaration -> "type" identifier ":" type ";"
+  if (!skip_token (Tiny::TYPE))
+    {
+      skip_after_semicolon ();
+      return Tree::error ();
+    }
+
+  const_TokenPtr identifier = expect_token (Tiny::IDENTIFIER);
+  if (identifier == NULL)
+    {
+      skip_after_semicolon ();
+      return Tree::error ();
+    }
+
+  if (!skip_token (Tiny::COLON))
+    {
+      skip_after_semicolon ();
+      return Tree::error ();
+    }
+
+  Tree type_tree = parse_type ();
```

```
+
+  if (type_tree.is_error ())
+    {
+      skip_after_semicolon();
+      return Tree::error ();
+    }
+
+  skip_token (Tiny::SEMICOLON);
```

... except for a detail: we need to create a type name. This means that the scope of names will contain two different kinds of names: variable names and type names. So before we can continue we will need to be able to distinguish the different kinds of names.

This is not very complicated, though, it is just a matter of extending or `Symbol` class with a `SymbolKind` field.

```
diff --git a/gcc/tiny/tiny-symbol.h b/gcc/tiny/tiny-symbol.h
@@ -13,14 +13,27 @@
 namespace Tiny
 {

+enum /* class */ SymbolKind
+{
+  INVALID,
+  VARIABLE,
+  TYPENAME
+};
+
 struct Symbol
 {
 public:
-  Symbol (const std::string &name_) : name (name_), decl (error_mark_node)
+  Symbol (SymbolKind kind, const std::string &name_) : kind(kind), name (name_), dec
    {
      gcc_assert (name.size () > 0);
    }

+  SymbolKind
+  get_kind () const
+  {
+    return kind;
+  }
+
@@ -41,6 +55,7 @@ public:
    }

 private:
+  SymbolKind kind;
   std::string name;
   Tree decl;
 };
```

Now it is mandatory to specify the kind of `Symbol` when we create it, so `parse_variable_declaration` and `query_variable` in `tiny-parser.cc` will have to be updated.

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
@@ -452,10 +457,10 @@ Parser::parse_variable_declaration ()
   if (scope.get_current_mapping ().get (identifier->get_str ()))
     {
       error_at (identifier->get_locus (),
-                "variable '%s' already declared in this scope",
+                "name '%s' already declared in this scope",
                 identifier->get_str ().c_str ());
     }
-  SymbolPtr sym (new Symbol (identifier->get_str ()));
+  SymbolPtr sym (new Symbol (Tiny::VARIABLE, identifier->get_str ()));
   scope.get_current_mapping ().insert (sym);
@@ -635,6 +728,11 @@ Parser::query_variable (const std::string &name, location_t loc)
       error_at (loc, "variable '%s' not declared in the current scope",
                 name.c_str ());
     }
+  else if (sym->get_kind () != Tiny::VARIABLE)
+    {
+      error_at (loc, "name '%s' is not a variable", name.c_str ());
+      sym = SymbolPtr();
+    }
   return sym;
 }
```

Now we can complete the implementation of `parse_type_declaration` that we left halfways above.

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
+  if (scope.get_current_mapping ().get (identifier->get_str ()))
+    {
+      error_at (identifier->get_locus (),
+                "name '%s' already declared in this scope",
+                identifier->get_str ().c_str ());
+    }
+  SymbolPtr sym (new Symbol (Tiny::TYPENAME, identifier->get_str ()));
+  scope.get_current_mapping ().insert (sym);
+
+  Tree decl = build_decl (identifier->get_locus (), TYPE_DECL,
+                          get_identifier (sym->get_name ().c_str ()),
+                          type_tree.get_tree ());
+  DECL_CONTEXT (decl.get_tree()) = main_fndecl;
+
+  gcc_assert (!stack_var_decl_chain.empty ());
+  stack_var_decl_chain.back ().append (decl);
+
+  sym->set_tree_decl (decl);
+
+  Tree stmt
+    = build_tree (DECL_EXPR, identifier->get_locus (), void_type_node, decl);
+
```

```
+  return stmt;
+}
```

The implementation is pretty identical to parse_variable_declaration (we could of course refactor the code to avoid some duplication here) but instead of a variable name we create a type name. In GCC a declaration of a type is represented using a node with tree code `TYPE_DECL`. That node can then be used in the `TREE_TYPE` of any expression or declaration (including another `TYPE_DECL`).

Once a type has been declared we want to use its type name. The only place where we can currently use a type name in tiny is in 〈type〉 so we will need to update `parse_type`. This will require a `query_type` function that we will see later.

```
diff --git a/gcc/tiny/tiny-parser.cc b/gcc/tiny/tiny-parser.cc
@@ -556,6 +620,16 @@ Parser::parse_type ()
       lexer.skip_token ();
       type = boolean_type_node;
       break;
+    case Tiny::IDENTIFIER:
+      {
+       SymbolPtr s = query_type (t->get_str (), t->get_locus ());
+        lexer.skip_token ();
+       if (s == NULL)
+         type = Tree::error ();
+        else
+          type = TREE_TYPE (s->get_tree_decl ().get_tree ());
+      }
+      break;
```

We will also allow the remaining part of `parse_type` work to work with an erroneous type in case `query_type` fails.

```
@@ -617,16 +690,36 @@ Parser::parse_type ()
 //       break;
 //     }
-      Tree range_type
-       = build_range_type (integer_type_node, it->first.get_tree (),
-                           it->second.get_tree ());
-      type = build_array_type (type.get_tree (), range_type.get_tree ());
+      if (!type.is_error ())
+       {
+         Tree range_type
+           = build_range_type (integer_type_node, it->first.get_tree (),
+                               it->second.get_tree ());
+         type = build_array_type (type.get_tree (), range_type.get_tree ());
+       }
     }

   return type;
 }
```

This uses a new function called `query_type` similar to `query_variable` that does the same query in the lookup but checks the name is a type name.

```
 SymbolPtr
+Parser::query_type (const std::string &name, location_t loc)
+{
+  SymbolPtr sym = scope.lookup (name);
+  if (sym == NULL)
+    {
+      error_at (loc, "type '%s' not declared in the current scope",
+                name.c_str ());
+    }
+  else if (sym->get_kind () != Tiny::TYPENAME)
+    {
+      error_at (loc, "name '%s' is not a type", name.c_str ());
+      sym = SymbolPtr();
+    }
+  return sym;
+}
+
+SymbolPtr
 Parser::query_variable (const std::string &name, location_t loc)
 {
   SymbolPtr sym = scope.lookup (name);
```

# Smoke test

We can try our new extension.

```
type my_int : int;

var x : my_int;
var y : my_int[2];

x := 42;
write x;
y[1] := x + 1;
write y[1];


type my_int_array : my_int[2];

var z : my_int_array;

z[1] := y[1] + 1;
write z[1];
```

```
$ gcctiny -o test test.tiny
$ ./test
42
43
44
```

Yay!

Admittedly this new extension does not look very interesting now but it will be when we add record types to the language.

That's all for today.

« A tiny GCC front end – Part 9 A tiny GCC front end – Part 11 »