THINK IN GEEK | In geek we trust

Arm Assembler Raspberry Pi GCC tiny Posts by Bernat Ràfales Archives

A tiny GCC front end – Part 9

Jan 31, 2016 • Roger Ferrer Ibáñez • compilers, GCC

Today we will do something relatively easy: let's add a way to declare boolean variables and express boolean literals.

Syntax

Since tiny already has a boolean type (for instance when doing a comparison like `a > b`) it is just a matter of making it explicit in the language. First let's extend the syntax of types. Some programming languages call this type *logical*, but we will call it **`bool`**.

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

Booleans only have two values: true and false. Technically speaking we already can express these two values in many different ways. For instance a way to express a true value is 1 = 1 and a false value is 1 != 1. So technically, nothing else is mandatory at this point. That said, this would be a poor language design choice, as it would make our programs look a bit weird. So we will add two boolean literals **`true`** and **`false`** that express a true boolean value and a false boolean value respectively.

We will have to extend our primary syntax.

⟨primary⟩ → **`(`** expression **`)`**
| ⟨identifier⟩
| ⟨integer-literal⟩
| ⟨bool-literal⟩
| ⟨float-literal⟩
| ⟨string-literal⟩
| ⟨array-element⟩
⟨bool-literal⟩ → **`true`** | **`false`**

Semantics

`bool` designates the boolean type of tiny.

A 〈bool-literal〉 of the form **`true`** is an expression with boolean type and true boolean value. Similarly, a 〈bool-literal〉 of the form **`false`** is an expression with boolean type and false boolean value.

> Note that in contrast to some programming languages (like C/C++), boolean and integer are different types in tiny and there are no implicit conversions between them.

Implementation

Given that much of the required infrastructure is already there, adding boolean types and literals is quite straightforward.

Lexer

We only have to define three new tokens bool, true and false. Since they are keywords, nothing else is required in the lexer.

```
diff --git a/gcc/tiny/tiny-token.h b/gcc/tiny/tiny-token.h
index 2d81386..fe4974e 100644
@@ -44,9 +44,11 @@ namespace Tiny
   TINY_TOKEN (RIGHT_SQUARE, "]")                                               \
                                                                                \
   TINY_TOKEN_KEYWORD (AND, "and")                                              \
+  TINY_TOKEN_KEYWORD (BOOL, "bool")                                            \
   TINY_TOKEN_KEYWORD (DO, "do")                                                \
   TINY_TOKEN_KEYWORD (ELSE, "else")                                            \
   TINY_TOKEN_KEYWORD (END, "end")                                              \
+  TINY_TOKEN_KEYWORD (FALSE_LITERAL, "false")                                  \
   TINY_TOKEN_KEYWORD (FLOAT, "float")                                          \
   TINY_TOKEN_KEYWORD (FOR, "for")                                              \
   TINY_TOKEN_KEYWORD (IF, "if")                                                \
@@ -56,6 +58,7 @@ namespace Tiny
   TINY_TOKEN_KEYWORD (READ, "read")                                            \
   TINY_TOKEN_KEYWORD (THEN, "then")                                            \
   TINY_TOKEN_KEYWORD (TO, "to")                                                \
+  TINY_TOKEN_KEYWORD (TRUE_LITERAL, "true")                                    \
   TINY_TOKEN_KEYWORD (VAR, "var")                                              \
   TINY_TOKEN_KEYWORD (WHILE, "while")                                          \
   TINY_TOKEN_KEYWORD (WRITE, "write")                                          \
```

Parser

Member function `Parser::parse_type` will have to recognize the bool token. The GENERIC tree type used will be `boolean_type_node` (we already use this one in relational operators).

```
@@ -551,6 +552,10 @@ Parser::parse_type ()
       lexer.skip_token ();
       type = float_type_node;
       break;
+    case Tiny::BOOL:
+      lexer.skip_token ();
+      type = boolean_type_node;
+      break;
     default:
       unexpected_token (t);
       return Tree::error ();
```

Finally, member function Parser::null_denotation has to handle the two new literals.

```
@@ -1333,6 +1338,18 @@ Parser::null_denotation (const_TokenPtr tok)
                           tok->get_locus ());
       }
       break;
+    case Tiny::TRUE_LITERAL :
+      {
+       return Tree (build_int_cst_type (boolean_type_node, 1),
+                           tok->get_locus ());
+      }
+      break;
+    case Tiny::FALSE_LITERAL :
+      {
+       return Tree (build_int_cst_type (boolean_type_node, 0),
+                           tok->get_locus ());
+      }
+      break;
     case Tiny::LEFT_PAREN:
```

Note that GCC function `build_int_cst_type` constructs a GENERIC tree with code `INTEGER_CST`. This does not mean that he node must have integer type. In our case a true boolean value will be represented using the integer 1 (and 0 for the false value), but note that the tree itself has `boolean_type_node`.

Nothing else is required. Compared to arrays this was easy-peasy.

Smoke test

Now we can use boolean variables and use them as operators of logical operators.

```
var a : bool;
var b : bool;

a := true;
b := false;
```

```
if a
then
  write "OK 1";
end

if not b
then
  write "OK 2";
end

if a or b
then
  write "OK 3";
end

if b or a
then
  write "OK 4";
end

if not (a and b)
then
  write "OK 5";
end

if not (b and a)
then
  write "OK 6";
end
```

```
$ gcctiny -o bool bool.tiny
$ ./bool
OK 1
OK 2
OK 3
OK 4
OK 5
OK 6
```

Yay!

Now we can rewrite our `bubble.tiny` program from part 8 in a nicer way.

```
--- bubble.orig.tiny    2016-01-31 10:28:22.486504492 +0100
+++ bubble.new.tiny     2016-01-31 10:28:58.314177652 +0100
@@ -15,11 +15,11 @@
 # Very inefficient bubble sort used
 # only as an example

-var swaps : int;
-swaps := 1;
-while swaps > 0
+var done : bool;
+done := false;
```

```
+while not done
 do
-  swaps := 0;
+  done := true;
   for i := 1 to n - 1
   do
      if a[i - 1] > a[i]
@@ -28,7 +28,7 @@
        t := a[i-1];
        a[i-1] := a[i];
        a[i] := t;
-       swaps := swaps + 1;
+       done := false;
      end
   end
 end
```

That's all for today

« A tiny GCC front end – Part 8

A tiny GCC front end – Part 10 »