A Language that Compiles to C

Patrick Schönberger

Hochschule RheinMain, Wiesbaden, Germany

Abstract. In this paper a compiler is presented that translates a language that is similar to C in both syntax and semantics into actual C code. The new language has some additional features that C does not which are supposed to ease development, most notably struct methods and generic functions and structs. The selected features map well to C code and the generated code should be readable and usable from existing code bases.

1 Introduction

The C programming language is one of the oldest languages that is still widely used today. Partly because it has almost unrivaled performance, low-level control and is available on virtually any platform, but also because it is used in many legacy systems which need to be maintained. Either way, many programmers still have to write C today, which can be cumbersome compared to writing code in a more modern programming language, because many features that make development using one of these modern languages more pleasant are not present in C. This not only increases the effort involved in programming in C, but it also introduces potential sources for errors.

Some of the features present in programming languages today, like garbage collection or closures, make programming easier and prevent bugs, but make fundamental assumptions about their environment, most notably that there is a runtime present that enables these features. But some features ease programming a lot while making very basic or no assumptions at all.

The idea behind this project is to create a compiler that translates a language that has a basic feature set similar to that of C into actual C code, while adding some quality of life features that are easily and predictably representable by C. In the process, some of the oddities of C, most of which are owed to the fact it is a really old language, are alleviated, streamlining the language.

What C code is going to be generated should be immediately obvious, and the resulting code should be usable from existing C code bases. This specifically means that no name mangling occurs and that resulting code makes no assumptions about whether it is being called from within the language itself or from external C code.

2 Language

The basic feature set is virtually identical to C, and the syntax resembles C somewhat closely, since the main target audience is programmers that are famil-

iar with C. The global context can contain four types of declarations: variables, functions, structs and namespaces. Namespaces can contain the same declarations as the global context.

Listing 1.1. Example code demonstrating the basic syntax and concepts

```
var global1 : int = 123;

func globalFunc() : void {
  puts("Hello\n");
}

namespace N1 {
  var v1 : int;
  func f1() : void {
    puts("Hello\n");
  }
  struct S1 {
    test : char *;
  }
  // nested namespaces
  namespace N2 {
    var v1 : int;
    struct S1 {
      i1 : int;
      i2 : int;
    }

    func f1() : void {
      // these have the same type
      var s1 : N1::N2::S1;
      var s2 : S1;

      s1.m1(123);
    }
  }
}

func main(argc : int, argv : char**) : int {
  var s1 : N1::N2::S1;
  var s2 : N1::N2::S1;
  var s3 : N1::S1;
  s1.i1 = 123;
  s1.i2 = 456;
  s1.m1(s2.m1(s3.m1(89)));

  return 0;
}
```

Functions Functions are the same as in C semantically. They are declared using the `func` keyword, followed by a name, a list of parameters and a mandatory return type. If the function is defined, the next thing is the function body, delimited by {}. Any body of code can contain variable declarations and statements.

Structs Structs are declared using the `struct` keyword. They have a name and a list of members, which can either be variables or methods. Note that struct declarations are not delimited by a semicolon and that there is no `typedef`s, a declared struct can just be referenced by name.

Statements As mentioned earlier, any body of code contains a sequence of variable declarations and statements. The following statements are available, all functioning exactly like in C:

- if
- switch
- for
- while
- assignment
- return

Additionally, any expression can also be treated as a statement.

Expressions The difference between statements and expressions is that expressions have a return type and statements do not. This means that an expression can be used almost everywhere a value is expected. Since the return value can simply be ignored, any expression can also act as a statement, but a statement can not act as an expression. Like with statements, the available expressions behave exactly as in C:

- function call
- method call
- literal (integer, decimal, string or bool)
- struct access (`./->`)
- unary operator (prefix/postfix)
- binary operator
- ternary operator
- array access
- variable

Operator precedence is not parsed, but since the the resulting code is compiled by a C compiler, C's operator precedence applies. Additionally, operators and all other expressions can be parenthesized to explicitly specify order of evaluation.

2.1 Differences from C

The additions to C consist of:

- Order independant declarations
- Namespaces
- Generics
- Struct Methods

Order Independant Declarations Every declared function and struct can be used independently from where it was defined within the same compilation unit, provided it is visible in the namespace. This does not alleviate the problem of self-referential structs or cyclic dependencies, since this would necessitate implicit use of pointers for struct members, which imposes too big of an assumption.

Namespaces Namespaces simply group declarations into a lexical context, and hierarchically control visibility. This is a compile time feature; since C has no such concept, the visibility of the resulting code can not be restricted. When resolving identifiers, namespaces are iterated from the inside to the outside, so if there's two functions with the same name and the one to call is the outer one, it's namespace has to be specified explicitly.

Listing 1.2. Example code showcasing namespaces

```
namespace N1 {
  func f1(): void {
    printf("N1\n");
  }

  namespace N2 {
    func f1(): void {
      printf("N2\n");
    }
    func f2(): void {
      f1();
      N1::f1();
    }
  }
}

func main() : int {
  N1::f1();
  N1::N2::f1();
  N1::N2::f2();

  return 0;
}
```

Listing 1.3. C code generated from namespaces

```
void N1_N2_f1 () {
  printf("N2\n");
}
void N1_N2_f2 () {
  N1_N2_f1();
  N1_f1();
}
void N1_f1 () {
  printf("N1\n");
}

int main () {
  N1_f1();
  N1_N2_f1();
  N1_N2_f2();
  return 0;
}
```

Generics Every function and struct can be defined generically. This allows to define placeholder type names which can be used in it's definition without having to know what types it will be used with. This is also only available directly from within the language, only already instantiated structs and functions are available from C.

Listing 1.4. Generic declaration and instantiation of a struct and a function

```
struct List<T> {
  array: T *;
  get(index: int): T {
    return this->array[index];
  }
}

func generic1<A>(a1 : A, a2 : A) : A {
  return a1 + a2;
}

func main(argc : int, argv : char**) : int {
  var l1: List<int>;
  l1.get(0);

  var i1: int = generic1<int>(1, 2);

  return 0;
}
```

For each unique instantiation, one version of the struct/function is generated with the specific types appended to it's name and the appropriate function call is inserted.

Listing 1.5. Example code showcasing generics

```
func generic1<A>(a1 : A, a2 : A) : A {
  return a1 + a2;
}

func main() : int {
  var i1: int =
    generic1<int>(1, 2);
  var d1: double =
    generic1<double>(1.23, 4.56);

  return 0;
}
```

Listing 1.6. C code generated from generics

```
int generic1_int (int a1, int a2)
{
  return a1 + a2;
}
double generic1_double (double a1, double a2)
{
  return a1 + a2;
}

int main ()
{
  int i1;
  double d1;

  i1 = generic1_int(1, 2);
  d1 = generic1_double(1.23, 4.56);
  return 0;
}
```

Struct Methods Struct methods provide a very basic form of object-orientation. They are simply functions defined within a struct, that can be called as struct members and get passed a pointer to the enclosing struct implicitly. This makes most calls a little shorter and also makes it apparent that the "object" that the method is called on has special meaning to the call, instead of just being the first parameter. It also makes chaining function calls cleaner.

Listing 1.7. Example code showcasing methods

```
struct S1 {
  i: int;

  m1(): void {
    printf("i: %d\n", i);
  }
  m2(): S1 * {
    return this;
  }
  m3(): void {
    printf("chained function calls\n");
  }
}

func main() : int {
  var s1: S1;
  s1.i = 123;
  s1.m1();
  s1.m2().m3();

  return 0;
}
```

Listing 1.8. C code generated from methods

```
struct S1 {
  int i;
};
void S1_m1 (struct S1 *(this)) {
  printf("i: %d\n", i);
}
struct S1 *(S1_m2) (struct S1 *(this)) {
  return this;
}
void S1_m3 (struct S1 *(this)) {
  printf("chained function calls\n");
}

int main () {
  struct S1 s1;

  s1.i = 123;
  S1_m1(&s1);
  S1_m3(S1_m2(&s1));
  return 0;
}
```

3 Compiler

The compiler is written in C++ and makes us of ANTLR 4[1] to generate C++ classes for the lexer and parser. The resulting code produces a parse tree from a string input, which is transformed into an internal representation (IR) of the program code. This representation can be traversed to check types, variable visibility etc. After analysing the code one more traversal is done to instantiate generic functions and structs and then the appropriate C code is generated.

3.1 Internal Representation

Internally, the code is represented as a hierarchy of structs, that contains comprehensive information about the program. The IR follows a Single Source of Truth approach, where everything is defined exactly once. There is no redundancy, so if a struct refers to anything that it doesn't own, it does so using a pointer. This allows making changes in one place and having them reflected throughout the entire program, which is mostly used for generic instantiation.

Listing 1.9. Struct representing a function

```
struct Function
{
  // function signature
  std::string name;
  Type returnType;
  std::vector<Variable> parameters;
  bool defined;

  // (potential) list of generic parameters and list of
      concrete instantiations
  std::vector<std::string> genericTypeNames;
  std::vector<std::vector<Type>> genericInstantiations;

  // function body
  Body body;
};
```

To get the corresponding IR from a parse tree, there exist a set of functions that each extract a single part of the program. Each of these functions generates one of the structs that make up the IR.

Listing 1.10. Function to get a Variable instance from the corresponding parse tree node

```
Variable getVariable(TocParser::VarContext * ctx)
{
  Variable result;
  result.name = ctx->varName()->NAME()->toString();
```

[1] https://antlr.org

```
  result.type = getType(ctx->type());
  return result;
}
```

Listing 1.10 shows a very simple of this example of this. The function gets a `Variable` instance by extracting the name directly from the parse tree. The type is handled by a separate function, since types need to be obtained in many different places. `getVariable` is used in variable and struct member declarations, as well as function parameters. This is possible because all of these use the same grammar rule, which means all of them generate an instance of `TocParser::VarContext`. This example shows how grammar rules are reused and how every function only does what is specific to the concept it handles, delegating everything else to other specialised functions to avoid redundancy.

3.2 Resolving identifiers in the namespace hierarchy

In order to generate the correct variable names and function calls, every identifier has to be resolved according to the namespace hierarchy. This is done by walking up the hierarchy and checking for e.g. the referenced variable name (see listing 1.11).

Listing 1.11. Find a variable in the specified context

```
// return the Variable and the absolute namespace path
// or nothing if it cannot be found
// parameters are the name, namespace prefix (e.g. Namespace1
    ::varName)
// and the context (e.g. the function body in which the
    variable is referenced)
opt<tup<Variable, std::vector<std::string>>>
findVariable(
  const std::string & name,
  const std::vector<std::string> & namespacePrefix,
  std::shared_ptr<Context> ctx)
{
  // iterate from given context up to the global context
  for (auto it = ctx; it != nullptr; it = it->parent)
  {
    // try finding the supplied namespace
    // if no prefix is present, this just returns the current
        context
    auto n = getContext(it, namespacePrefix);
    if (n.has_value())
    {
      // find the variable by name
      // note that the namespace prefix is no longer needed
      // because n references the given namespace
      auto x = find<Variable>(std::get<0>(*n)->variables, [&](
          Variable _) { return _.name == name; });
```

```
      if (x.has_value())
        return std::make_tuple(x.value(), std::get<1>(*n));
    }
  }
  // return nothing if variable cannot be found
  return nullopt;
}
```

This is done analogously for functions and structs, and both can also be returned as a pointer, making it possible to search for a function or struct and change it globally.

3.3 Generics

Generic functions and structs are instantiated for each unique invocation. This is done in two steps. First, all function calls and references to generic structs are collected and saved in the functions/structs definition. To be able to extract all function calls, there exists a Visit class that iterates over the whole program and calls supplied callbacks for every struct in the IR (see listing 1.12). Function calls are then collected by supplying only a callback for expressions and checking the expression type to be a function call inside the callback.

When generating the C code for a generic function, the list of concrete instantiations is iterated. For each iteration, the instantiated types are stored in a global map. The function that generates the code for types checks this map and replaces the generic types where applicable.

Listing 1.12. Visiting a Struct

```
void visit(const Struct & x)
{
  v.onStruct(x, ctx); // invoke callback with Struct

  VISIT(x.members)     // visit struct members
  VISIT(x.methods)     // visit methods
}
```

3.4 Generating C

Code generation is done by recursively calling functions to output the corresponding C for a single construct, similarly to how the IR is obtained from the parse tree. Every function only outputs what is specific to that element, for example for a function, the name and parameters are printed, but for the body a separate function is called. That same function is also called for if statements and other constructs that are also associated with a block of code. All the code is printed to a generic `std::ostream`, so it can easily be output to stdout, a file or a string.

Listing 1.13. Excerpt from the code that generates a C function

```
// print one copy per instantiation
for (auto instantiation : f.genericInstantiations)
{
  // set global type mapping
  for (int i = 0; i < f.genericTypeNames.size(); i++)
  {
    currentInstantiation[f.genericTypeNames[i]] =
        instantiation[i];
  }

  out << f.returnType << " " <<
    generateModifiers(
      namespacePrefix() + f.name + genericAppendix(
          instantiation),
      f.returnType.modifiers) <<
    " (" << vectorStr(f.parameters, ", ") << ")";

  if (stub)
    out << ";\n";
  else
    out << "\n" << f.body;

  currentInstantiation.clear();
}
```

The code in listing 1.13 shows what printing a generic function looks like. To get the full name used for the C function, the namespace prefixes and the function name itself are combined with an appendix for the generic instantiation (see listings 1.5 and 1.6). Any modifiers the return type has are then appended to the resulting name, since they are applied to the name instead of the type in C[1, Section 3.5.4].

4 Conclusion

Parsing and code generation works well for basic features and for most cases also for the new features. Type and identifier lookup should also work in most cases. In general, the code is very modular, making adding new features or making changes to existing ones easy. The generated C code is very readable, even when using some of the advanced features like struct methods or generics. It generates code the way most C programmers would probably approach these problems when writing C by hand.

4.1 Benefits

Namespaces and struct methods are nice and can make code more terse, but the biggest time saver is probably generics. These usually have to be approached

by either writing the same code for each type, making the parameters a void pointer, which has its own problems, or wrapping all related code in a macro that inserts the type names. Apart from being worse to write and breaking auto completion in IDEs, the macro approach also has the issue that macros work on a lexical level[1, Section 3.8.3]. This can lead to accidentally replacing non-type parts of the code. Compared to this, writing and using generics in the new language is a lot easier, while still making it easy to use the generated generic instantiations from C.

4.2 Limitations

While code generation works well in the general case, it can still be broken in many places. Most of this is owed to the lack of a proper type system, which can break type lookup. This is also the reason why there are no generic methods and generic type instantiations cannot be nested (e.g. `List<List<int>>`). Other than that the compiler is missing some basic functionality due to a lack of time that can easily be added in the future. Most notably this includes support for multiple input and output files and some basic C functionality like enums/unions or const/static/extern type qualifiers.

4.3 Outlook

Apart from fixing what is currently broken, there are some changes and additions imaginable in the future. The most notable of these is refactoring the IR to allow easier traversal, especially of parent nodes. This is currently very limited. Apart from that it would be interesting to add additional compilation targets, which could be other programming languages or possibly a more low-level target like LLVM[2] or assembly. It would also be interesting to see what kind of possibilities open up when the language is also available at runtime to replace the C preprocessor with an actual programming language. This would necessitate writing an interpreter for the language, but could potentially have a lot of benefits.

References

1. American National Standards Committee on Computers and Information Processing: Programming Language C (1988), `http://port70.net/~nsz/c/c89/c89-draft.html`(accessed 14.08.21)

[2] `https://llvm.org/`