Getting acquainted with the C-- language

Pablo Nogueira Iglesias

18th September 2000

Contents

1 Introduction

`C--` is a portable assembly language designed to be a good intermediate target language for high-level language compilers. Its design aims at reducing the 'semantic distance' that separates a high-level language from a typical target language, yet providing enough low-level control so that high-level services such as garbage collection, exceptions, concurrency, etc., can be easily and efficiently implemented. It is designed to run fast on a number of today's major computer architectures and to be as much architecture-independent as loss of efficiency allows. Speed and ease of use has sometimes taken precedence over orthogonality and minimality. `C--` should be rich enough to be a viable back-end for most mainstream and research compilers.

Despite the name, `C--` is not a strict subset of C. The name "`C--`" arose instead from the popularity of C as a compiler target language. Knowledge of C is not a strict prerequisite.

This document aims to be an outward and gentle introduction to the `C--` language. There are two other documents[1] related to this one:

1. The language's design rationale *"C--: A Portable Assembly Language."* It explains the language focusing on decisions about its design.

2. The language's Reference Manual. This short introductory tutorial should be read along with the Reference Manual present for consultation. References to sections of the Manual are denoted as shown in this example:

   (§3.7) stands for Section 3.7 of the Reference Manual.

This paper has been written following the structure of a `C--` program from a high-level point of view. First of all, the available types are listed, and then, the structure of a `C--` program is described from its element components. Each of these components is explained in more detail in appropriate subsections. One can get an idea of what a `C--` program is made of just by looking at the table of contents.

Before starting, two remarks: comments in `C--` start with `/*` and end with `*/`, and cannot be nested. `C--`, like C, does not incorporate input/output facilities as part of the language definition.

2 Type system (§3.4)

`C--` provides a minimal and unsafe type system, in the line of conventional assemblers. Particularly, the available types are:

- `bits8`, `bits16`, `bits32`, `bits64`.
- `float32`, `float64`, `float80`.

Types provide essential information about a value: its size (in bits), and the kind of hardware storage needed to hold it, which also suggests the kind of operations that can be performed upon values of that type. For example, floating point values are usually held in different registers and manipulated by different set of instructions than other values. The `bits` family will be used for integers, characters, and pointers.

Types are used in

1. **Procedure definitions** (Section 5), to declare the type of the formal arguments.
2. **Declaration statements** (Section 5.1), to declare the type of procedure local variables.
3. **Memory write statements and memory read expressions** (Section 5.8), to indicate the type of the value written or read.
4. **Data layout directives** (Section 4), to declare the type of the allocated datum.

[1] They are available at `http://www.cminusminus.org`

3 The elements of a C-- program

A C-- program may be viewed outwardly as a collection of *procedures* and memory *labels*. Procedures provide the code definition and labels provide the access to memory locations that have been conveniently allocated and initialised. Procedure and memory label names have global scope.

More specifically, a C-- program is a sequence of data layout directives, and/or procedure definitions, and/or `import` declarations, and/or `export` declarations, interleaved *in any order*. Let's consider them in turn.

4 Data layout directives (§4)

C-- provides detailed control over static memory layout in the same way ordinary assemblers do. Memory is viewed as an array of bytes from which different sized types can be read and written. Allocation and initialisation of typed data is done via the `data` directive. Each *datum* declaration consists of a type, the number of elements of that type to allocate, and a constant list for initialising the allocated elements to the values given in the list. Allocated memory should be viewed as an array of bytes which can be addressed byte-wise. Data are allocated one after another by default, i.e. after the last byte of a datum goes the first byte of the next datum.

Memory *labels* may be declared inside `data` directives. They are the means to refer to the allocated data. This referencing can be done via label names or expressions on them—it is possible to address any byte in the allocated memory by starting from a label and adding the right offset, for labels point to the next byte after their declaration site. Memory labels are *constant pointers*, not updatable storage locations. Memory label names may be used in expressions before they are declared (forward references), but cannot be assigned to a value. An example will help to understand:

```
data {
 label1: bits32[2]{10};   /* Two   32-bit integers both initialised to 10     */
   bits32[3]{1,0,-2};     /* Three 32-bit integers initialised to 1, 0, -2    */
   bits16[1024];          /* 1024 uninitialised 16-bit words                  */
   bits64{label4};        /* A 64-bit address initialised to a fwd.reference  */
   float64{2.24};         /* A 64-bit float initialised to value 2.24         */
 label2:
 label3: bits8;           /* An uninitialised byte */
 label4:
}
```

This `data` directive allocates 6 data and declares 4 labels: `label1` points to the first byte of the first 32-bit integer initialised to 10, `label2` and `label3` both point to the sixth datum (the uninitialised byte), and `label4` points to the next byte after the uninitialised byte, thus pointing outside the allocated memory.

The six data are allocated in sequence, thus, the next byte after the second `bits32` initialised to 10 is the first byte of the `bits32` initialised to 1. Likewise, the next byte after the `float64` is the uninitialised byte. Neither implicit alignment nor padding is performed. Nonetheless, for performance reasons and also to comply with some architecture requirements, the alignment of data can be explicitly specified using the `align` directive, which inserts padding as needed, ensuring that the next datum is placed on the appropriate boundary (§4.3).

The type of a label is architecture-dependent: it will be bits$n$, where n is the number of bits needed to hold a *data pointer* in the particular architecture. We will call this type the 'architecture's *native data-pointer type*'. Indeed, any expression that yields a value of such type may be used to refer to the allocated memory. Note that *a label contains no type information about the datum it points to*—it is just the address of a particular byte. In the example above, the four labels may be of type `bits32` or `bits64` depending on whether the architecture's natural data-pointer type is `bits32` or `bits64` respectively. In the former case, the C-- compiler should produce a compilation error at the fifth allocated datum initialised to a (forward reference) label of type `bits64`, for it should be of `bits32` type. It will be easy for the `C--` compiler, however, to inform the high-level language front-end about the particular types for a given architecture.

Memory allocated with `data` is readable and writable. A memory read *expression* fetches a value of the requested type from the allocated memory using a label or an expression on labels for determining the reading address. A memory write *statement* writes a value of a particular type to the address specified by a label or an expression on labels. *Since labels provide no type information, it is the reading and writing operations that specify the type (size, storage) of the data to read or to write.* (Section 5.8).

The declaration of labels is optional; it is perfectly possible to allocate an unaccessible memory block that would be unusable, though a compiler should warn about such possibility.

5 Procedure definitions (§5)

Most machine architectures provide usually a 'jump' and a 'call' instruction. These instructions, however, deal simply with flow of control. The programmer has to deal with parameter passing through registers, through the stack, or through any other mechanism. `C--` supports parameterised procedures like most high-level languages. The reasons for providing procedures are explained in Section 6 of the design rationale, but let it suffice to say here that `C--` procedures reduce the 'semantic distance' between `C--` and a high-level languages.

In particular, `C--` procedures are very similar to C functions, e.g.: similar definition syntax, pass-by-value parameter passing, forbidden nesting of procedures inside procedures, etc. But there are some notable differences: procedure jumps (hence, efficient tail-calling), multiple returned values, no return type in signature, procedure call as a statement (not an expression), fixed number of arguments, no nesting of local scopes, etc. Procedure bodies may contain local variable declarations and other statements. Statements are only possible inside procedure bodies.

The term 'procedure' is used instead of the term 'function' despite that values may be returned from a procedure. The term function should be used more appropriately for *pure* mathematical functions, i.e., functions which return the same value(s) for the same argument(s), being free of side effects. (Of course, some `C--` procedures may be pure functions.)

Figure 1 shows three example procedures `sum_prod`*i* that compute for an integer argument n the sum $1 + 2 + ... + n$ and the product $1 \times 2 \times ... \times n$ by ordinary recursion, by looping, and by efficient tail recursion. Note that the example assumes that an integer is 32-bits long in the particular architecture.

Other `C--` procedures could invoke any of these three. For instance, to get the value of the sum and the product up to 23 using procedure `sum_prod3`, the call would be something like

```
sum, prod = sum_prod3(23);
```

where `sum` and `prod` would be local variables declared inside the invoking procedure body.

5.1 Local variables (or virtual registers) (§5.2.2)

`C--` allows any arbitrary number of typed local variables to be declared anywhere inside a procedure body. In Figure 1, `s` and `p` are local variables to procedure `sum_prod1` of type `bits32`. Declarations can be placed anywhere another statement can. A declaration statement is very much like C's: the type is specified followed by the list of variable names of the same type.

`C--` maps these local variables to machine registers—typed storage locations that don't have an address—, if enough registers are available at one time to keep all local variables in them. Otherwise, local variables are mapped to memory locations, like, for instance, the stack.

In our example, variables `s` and `p` in procedures `sum_prod1` and `sum_prod2` will be allocated almost certainly in machine registers and not in stack slots since they are the only two local variables of the procedures.

```
sum_prod1(bits32 n)
{
  /* Computes by ordinary Recursion */
  bits32 s, p;

  if n==1 {
    return (1, 1);
  } else {
    s, p = sum_prod1(n-1);
    return (s+n, p*n);
  }
}

sum_prod2(bits32 n)
{
  /* Computes by looping. */
  bits32 s, p;

  s = 1;
  p = 1;
  loop:
    if n==1 {
      return (s, p);
    } else {
      s = s + n;
      p = p * n;
      n = n - 1;
      goto loop;
    }
}
```

```
sum_prod3(bits32 n)
{
  /* Computes by tail recursion */
  jump sum_prod3_help(n, 1, 1);
}

sum_prod3_help(bits32 n, bits32 s, bits32 p)
{
  if n==1 {
    return (s, p);
  } else {
    jump sum_prod3_help(n-1, s+n, p*n);
  }
}
```

Figure 1: Four C-- procedures.

5.2 Procedure call statement (§5.2.8)

Procedures sum_prod1 and sum_prod3 illustrate the two procedure invocation mechanisms supported by C--, namely, ordinary procedure *calls* and procedure *jumps*. Jumps are discussed in the next section.

Procedure sum_prod1 recursively calls itself with a call statement:

```
s, p = sum_prod1(n-1);
```

The semantics is that of a pass-by-value function invocation: sum_prod1 is invoked passing the value of n-1 as argument and returns two values to local variables s and p.

Procedure calls are explicit in C--, *they are complete statements.* This means that the previous statement *is not* an assignment statement in which its right hand side is an expression that happens to be a procedure invocation. The whole statement is a call statement (§5.2.8), and should not be confused with the assignment statement (§5.2.3) despite their syntactic similarity. Procedure calls are not expressions and cannot be embedded in expressions. Calls such as

```
s, p = sum_prod1(n-1) + 4;
```

are not legal. The expression notation makes no sense since procedures may return multiple values (and C-- supports no tuple type). By extension, the rule applies also to procedures that return single values. Instead than

```
r = f( g(x) );
```

we must write:

```
tmp = g(x);
r = f(tmp);
```

This makes explicit the evaluation order, the type and name of the intermediate value `tmp` (its type is specified in its declaration as a local variable), and the location of each call site. For these reasons, a procedure's return type cannot and will not be specified in the procedure definition.

It should be noted, however, that procedure *names* are expressions, and that *any expression that evaluates to a procedure entry-point address may be used in a call statement.* It is correct to write, for example, `x = (f+5)(y);`, where `f` is any label, if the value of expression `(f+5)` is a procedure address. Hence, the call statement takes an *expression* that is evaluated to find the address of the target procedure. A more illustrative example follows:

```
data {
   p: bits32{proc1};  /* forward reference */
      bits32{p};
}

proc1(bits16 c) {
  return (c);
}

proc2() {
  bits32 x, y;
  x = bits32[p](45);
  y = bits32[ bits32[p+4] ](45);
  return (x, y);
}
```

Label `p` points to a datum that is initialised to a procedure name (forward reference). This procedure name is an address and will have type `bits32`, assuming that the *native architecture's* ***code****-pointer type* is `bits32`. Assuming also that an integer is 16-bits long, and that the native data-pointer type is `bits32` (for `p`), procedure `proc2` returns twice the same value, namely 45, for each call statement invokes the same procedure (`proc1`). The kind of expression used in the calls is a *read* expression (Section 5.8) that yields `proc1`'s entry point address as result.

The number and types of the actual arguments in a call statement should match the number and types of the formal arguments of the called procedure. Also, the number and types of the values returned should match the number and types of the names in the name list (on the left of the `=`). No checks are performed at compile time and at run time. The `C--` compiler needs to know nothing about the called procedure.[2]

A procedure call may transfer control to any `C--` procedure, even one in another *compilation unit* (Section 6).

If a procedure returns no values, its return statement should have an empty list. Furthermore, the name list and the `=` must not be written in a call statement to that procedure. Procedures that don't take arguments should have an empty argument list. In the following example, procedures `g` and `f` return no value. Procedure `f` takes no arguments and invokes `g` without returning values to a name list.

```
g(bits8 c)
{
   ...
```

[2]Note that it is impossible to check the correct correspondence of number and types of formal and actual arguments at compile time, for a call statement takes an expression which is evaluated at execution time to find out which procedure is actually invoked. Such checks can be done at execution time, but it is up to the `C--` compiler to generate code that performs them.

```
   return();
}
f()
{
   g('A');
   return ();
}
```

5.3 The `jump` statement (§5.2.9)

The `jump` statement transfers control to a procedure without automatically returning back. It may be thought of as a "control jump carrying parameters". Local variable names die when jumping.[3] Jumps are the means to perform efficient tail-calls. In Figure 1, procedure `sum_prod3` jumps to procedure `sum_prod3_help`, which depending on the value of `n` either tail-calls (jumps) to itself or returns to `sum_prod3`'s **caller**. Thus, `sum_prod3` implements a 'recursive' call with no stack growth. Procedure `sum_prod3` just provides the initial arguments to `sum_prod3_help`, the rest of the computation being done by accumulating parameters.

Like a call statement, a `jump` statement takes an expression of the native code-pointer type that is evaluated to yield the entry-point address of the target procedure. The number and types of the actual arguments in a `jump` should match the number and types of the formal arguments of the target procedure. Again, no checks are performed at compile time and at run time.

The target procedure may be located in another compilation unit (Section 6).

5.4 Jump nesting and `return`

Procedure `sum_prod3_help` returns the computed result carried by the accumulating parameters when the value of `n` is 1. Recall that `sum_prod3_help` returns to `sum_prod3`'s **caller** and not to `sum_prod3`.

A `jump` statement transfers control to the target procedure and kills the current local variables, but *it keeps the return address*. For instance, if procedure calls are implemented using the stack to hold the return address and the local variables, the latter are removed from the stack frame when jumping, but not the former. Thus, the return address is available to the target procedure. Indeed, we may have any number of nested jumps, as in Figure 2—all of them keep the original return address.

In Figure 2, procedure `proc1` calls procedure `f`. Procedure `f` jumps to procedure `g`, procedure `g` jumps to procedure `h`, and so forth. The last procedure in the series (`w`) ends with a return statement. Since `f` keeps the return address and so do all the other target procedures in the series, control can be transfered back to `proc1` from `w`.

If, instead, procedure `proc2` invokes procedure `f`, when reaching the return statement in procedure `w`, control is transfered back to the procedure that called `proc2` if there is actually one, otherwise the result is unspecified. The `C--` program writer should ensure the correspondence between the number of calls and the number of returns, and also that they follow the desired semantics.

It should be noted, therefore, that a procedure call is the same independently of whether the target procedure either returns or jumps.

5.5 Every control path must end

Every control path in a `C--` procedure should end either in a `return` statement or in a `jump` statement, otherwise the effect in the control flow is unspecified.

5.6 Other statements

Procedure `sum_prod2` illustrates some of the remaining statements provided by `C--`.

[3] In fact, a `jump` either deallocates or reuses the current stack frame.

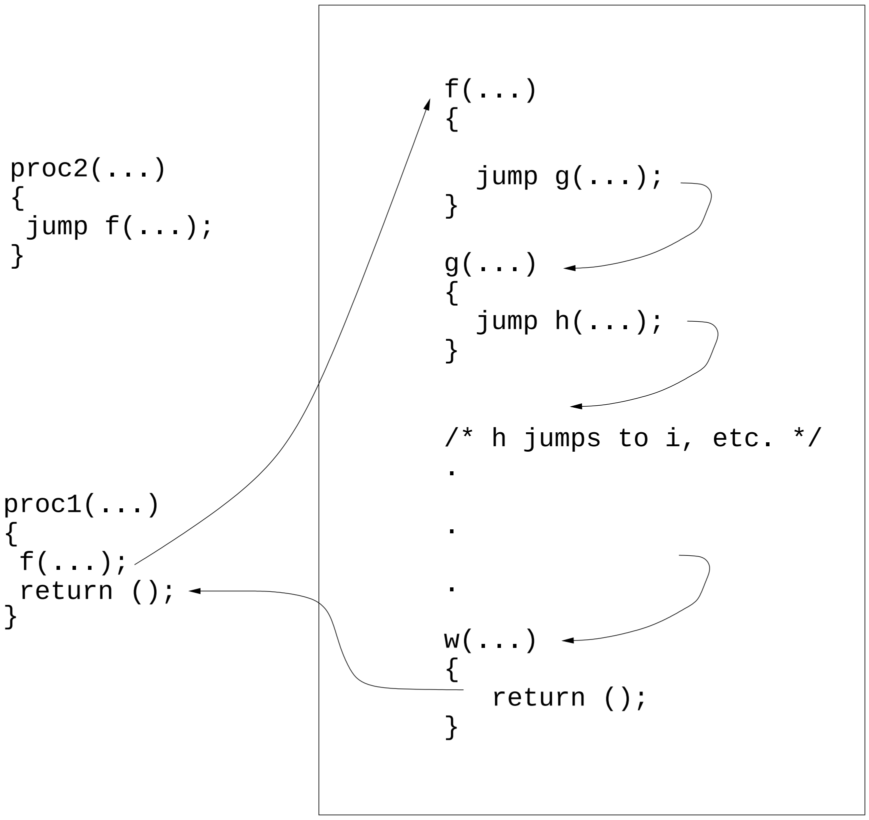

Figure 2: Nested jumps and return.

5.6.1 Conditionals

`C--` provides conditional statements `if` (§5.2.5) and `switch` (§5.2.6), but it neither provides a boolean type nor has `int` as implicit type for boolean results, like C does.[4]

The `if` statement consist of an assembly-like condition test and the branch to execute when the test holds. It may also have an optional `else` branch. Condition tests can only have the form of a binary relational *operation* on any two `C--` expressions of the same type. The available relational operations are `==`, `!=`, `<`, `>`, `<=`, and `>=`. They are called *operations* instead of *operators* because they can only appear in `if` condition tests, not inside expressions as do the `C--` operators.

The `switch` statement performs multi-way branching depending on the value of a `bits`*n* expression. It may optionally include a range within which the expression's value is guaranteed to be by the programmer. There is no bounds checking at run-time, for the range is a hint to the compiler so that such check can be omitted for performance. It is possible to specify multiple alternatives in front of every branch and also a `default` branch that is taken when none of the other branches are taken.

5.6.2 Local *control* labels and `gotos` (§5.2.7)

Alterations to flow of control inside procedure bodies can be achieved using local *control* label declarations and `goto` statements. Local control labels have nothing to do with the *memory* labels or pointers discussed so far. Unlike a memory label, a control label is only in scope of the procedure body where it is declared, hence the adjective "local". Once declared, a control label can only be used in a `goto` statement.

[4]The boolean type has been recently added to the language.

In turn, a `goto` statement can only have a control label as target. The semantics is simple: a `goto` statement forces the control flow to resume at the point where the control label is declared.

In Figure 1, the control label `loop` is declared inside procedure `sum_prod2` and it is used to implement a loop.

5.6.3 Assignment (§5.2.3)

As expected, the assignment statement has the form *name* `=` *expr*. It stores the value of expression *expr* in the local variable *name*, provided they have the *same type*. For example, the assignment statement `x = x + 1;` increments the value of variable `x`.

5.6.4 Memory write

The memory write statement is used to store values in the allocated memory. It has the form

type`[`$expr_1$`]` `=` $expr_2$

where $expr_1$ is an expression that yields an address value of the native data-pointer type and $expr_2$ is an expression of type *type*. Expression $expr_1$ will typically contain at least a memory label. Notice that memory writes are clearly distinguished from register storages. For example, the memory write statement

```
float64[label] = 3.14163;
```

stores the value of π in the 64-bit float datum pointed to by `label`.

5.7 Expressions

The power of a language resides in the richness of its expressions. `C--` expressions can be recursively defined. Expression can be names, constants, operators on expressions, primitives on expressions, casting operators on expressions, and memory read expressions that have subexpressions to yield the reading address. Let's consider them in more detail:

- **Names**, that is, local variable names, procedure names, and memory label names are all expressions. (§3.7)
- **Constants** like integer and floating point numbers, single-quoted character constants, double-quoted strings, and symbolic constant names are expressions. `C--` follows C's syntax for denoting constants. For example, `'A'` is a character constant and `2.43` is a `float` constant whose size depends on the particular architecture. (§2.6)
- **Built-in operators**, that is, arithmetical (`+`, `-`, `*`, ...) and bitwise (`&`, `|`, `>>`, ...) operators. *These operators are strongly typed*, i.e., operators can only take as arguments expressions of the appropriate type. Like conventional assemblers, there are different operators for the different storage classes. For example, the addition operator for `bits` expressions is `+`, and the addition operator for `float` expressions is `+f`. (§6.3)
- **Built-in primitives** that complete the set of operations that can be performed upon expressions. Primitives are very similar to operators, but their evaluation order, however, is not implicit. Their syntax is very similar to that of a procedure call, but no call is actually being made—in fact, primitive names are not expressions. For example, primitive `abs(x-1)` yields the absolute value of its `bits`*n* argument `x-1`. Primitives are also *strongly typed.* (§6.4)
- **Casting operators** that perform type casts, not conversions, between types. For example, given the local variable `x` of type `bits16` and the local variable `y` of type `bits8`, the expression `y = bits8(x)` simply stores in `y` the lower-order byte of `x` without converting `x` to a `bits8`. (§6.6)
- **Memory read expressions** which have the form *type*`[`*expr*`]`, where *expr* is any expression that yields a memory address value of the native data-pointer type, and *type* tells the size and the storage class of the data to read from that address. (§6.2)

5.8 Memory write vs. Memory read

Suppose we have the following toy C-- program:

```
data { foo: bits32{17};}

f() {
   bits32[foo] = bits32[foo] + 1;
   return ();
}
```

The memory write statement stores the value 18 in the datum pointed to by `foo`, updating the value 17. There are two important points worth mentioning:

1. The addressing mode *type*`[`*expr*`]` is interpreted as a memory write or a memory read depending on whether it appears on the left or right hand side. Since memory reads are expressions, they may appear anywhere a expression can, e.g., on the right hand side of assignment statements and memory write statements, in condition test expressions of `if` statements, etc.

2. The size and storage class of the value to be read or written must be given explicitly. Recall that labels and expressions on them have native data-pointer type. It is impossible to know the type of the stored value because that information is not provided by the label name.

5.9 Alignment in memory writes and reads

When discussing data layout directives, it was said that explicit alignment of data was achieved with the `align`*n* directive. Memory reads and writes must be done also with the proper alignment.

Memory reads and writes are assumed aligned to the size of the type to be loaded or stored. For example, given the read expression `bits32[foo]`, C-- assumes that `foo` is 32-bit aligned. If it is not, the value fetched is unspecified.

Sometimes it is necessary to perform misaligned or over-aligned memory accesses. C-- supports misaligned or over-aligned memory reads and writes. A memory read statement and a memory read expression both may be optionally qualified with an alignment flag. (§5.2.4) (§6.2)

6 The `import` and `export` declarations (§5.3)

A C-- *compilation unit* is a file containing a complete C-- program that can be successfully compiled and that is suitable for linking. In general, any procedure or memory label name declared in a C-- compilation unit may be invoked or referred to, respectively, from other C-- compilation units or from foreign language programs. All names used outside a C-- compilation unit must be explicitly exported. Likewise, names used but not declared by a compilation unit must be explicitly imported. Importing and exporting of names is achieved with `import` and `export` declarations respectively:

- `import`: imports names belonging to other C-- compilation units or to foreign language programs. The syntax is:

  `import` $name_1$,...,$name_n$;

  where $name_i$ is an external imported name that will have native data-pointer or code-pointer type depending on whether it stands for a label or for a procedure name respectively. Notice that it is not necessary to indicate the name of the compilation unit—`import` is much like C's `extern`.

- `export`: exports procedure and label names which may be invoked or referred to, respectively, from other C-- compilation units or from foreign language programs. The syntax is:

  `export` $name_1$,...,$name_n$;

```
/* prog1.c-- */                    /* prog2.c-- */

import f, la;                      export f, la, lb;
export g;                          data { la: bits32{0};
                                          lb: bits8{'\n'};
g(bits8 x)                                lc: bits8;
{                                  }
  x = f(la+4);                     import g;
  return (x);                      f(bits8 x) { return (x); }
}
```

Figure 3: Importing and exporting example.

where $name_i$ is a procedure or label name.

In Figure 3, the C-- compilation unit `prog1.c--` imports from the C-- compilation unit `prog2.c--` a procedure name `f`, and a label name `la`, both used in a call statement. The importing is successful because `f` and `la` have both been exported in `prog2.c--`. Note that `prog1.c--` would not be able to import label `lc` from `prog2.c--`, that `g` is imported by `prog2.c--` but not used, and that `lb` is exported by `prog2.c--` but not imported by `prog1.c--`.

Section 9 provides more examples of importing and exporting regarding inter-operation with foreign languages.

7 The structure of a C-- program

Procedure definitions, `data` directives, and `import`/`export` declarations may be interleaved in any order in a C-- program: a `data` directive may appear before or after a procedure definition, as well as may an `import` or an `export` declaration. Furthermore, any of the three may not appear at all; we may have a C-- program with just `data` directives or a program with just procedure definitions. Note also that it is possible to write a stupid C-- program with no exports that would be unusable (see next section). Procedure and label names, either declared or imported, have global scope within the C-- program independently of their declaration site.

8 Start-up invocation

C-- does not have a `main()` procedure that is first called at run-time. Instead, as mentioned in Section 3, a C-- compilation unit is viewed as a collection of procedures and labels. The procedures defined inside a C-- compilation unit have no hierarchy, any of the them may be called from outside the compilation unit as long as they are exported and the calling convention is known. The way to invoke C-- code, then, is to call C-- procedures from a foreign language program. Labels allow the foreign program to access the allocated data. All this will become clearer in Section 9.

9 Foreign language interface (§3.8) (§5.3)

C-- is designed to inter-operate with foreign language programs. This means that

1. C-- exported procedures and labels may be used by foreign language programs. In the former case, since foreign programs follow particular calling conventions, C-- procedures should be able to use them.

2. C-- types and the foreign language types should match in the particular architecture.

The next two sections elaborate.

9.1 Calling convention declaration

Figure 4 shows an example of a C program inter-operating with a C-- program. The C program uses the C-- label foo and invokes two C-- procedures, f and g. The C-- program, invokes C's printf().[5] The names f, g, and foo are declared as extern inside the C program—making them visible to the C functions—, and exported by the C-- program using export. The name printf is imported by the C-- program. Notice that no information is provided about what the name stands for (is it a procedure or a variable?). The C-- compiler needs to know nothing about the imported names. Their use inside the C-- program should, therefore, be (semantically) correct. The C-- compiler will not attempt any checks.

```
#include <stdio.h>

extern int  g(), *foo;
extern void f();

int  main()
{
  int x;

  f( );
  x = g(3);
  printf("%d", *foo);

  return 0;

}


/* Other C functions and/or
 * data declarations.
 */
```

A "C" Example Program.

```
data {
  foo:bits32{17};
  str:bits8[3]{'%','d',0};
}

export f, g, foo;

foreign C f( )
{
   foreign C printf(str, 3);
   foreign C return ();
}

import printf;

foreign C g(bits32 y)
{
   bits32 x;
   x = bits32[foo] + y;
   foreign C return (x);
}

/* Other functions and/or */
/* data directives and/or */
/* import/export decl.    */
```

A "C--" Example Program.

Figure 4: C-- code invocation from a C program.

Procedures f and g are both called from a C program so the calling convention to be used upon their invocation is declared in their definition using the keyword foreign followed by the calling convention to be used, in this case C's. The calling convention for returning is also specified in the return statement. The C-- program must use C's calling convention to call printf(), and so the convention is also specified in the calling statement. Summarising:

- The calling convention must be declared in the *definition* of exported procedures in order to call them from a foreign language program.
- The calling convention must be declared in return statements that transfer control back to foreign procedures/functions. The reason for this is that there is another way of exiting a procedure (namely, the jump statement), and procedure calls are independent of whether the procedure's control path ends with a return or with a jump. (The jump statement needs

[5]The C program includes the header stdio.h in which printf() is defined; thus, printf() is defined inside the C program after preprocessing.

not declare the calling convention, for it is C--'s convention.) In Figure 2, procedure f could have been called from a C program. The new arrangement is shown in Figure 5. It can be noted that *a return statement may appear in a place where it is not obvious to tell the calling convention*, so therefore, it should be indicated.

- The calling convention must be declared in C-- *call statements* to imported foreign language procedures/functions.

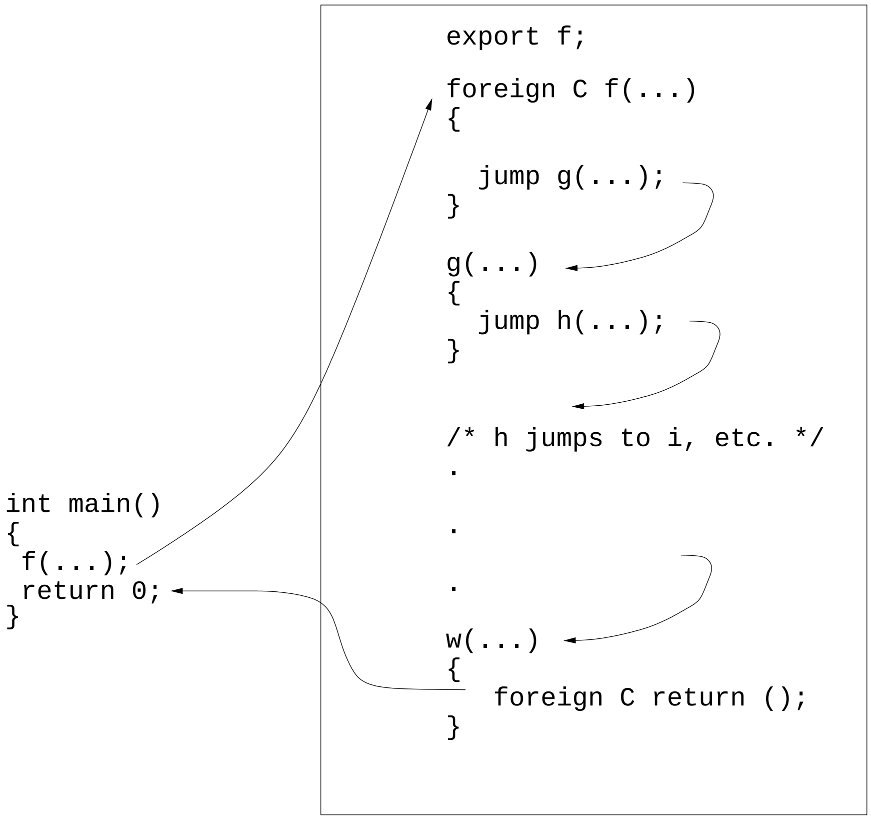

Figure 5: Return statement needs calling convention declared.

9.2 Types should match

Foreign language programs offer a type system somewhat architecture-independent. This is not the case with C--. Therefore, C-- types should match the foreign language types *in the particular architecture* when inter-operating. More precisely:

- When calling C-- *procedures* from foreign language programs, the types of the actual arguments should match the types of the formal arguments. For example, looking at procedure g in Figure 4, C's int must be equivalent to a bits32, so that the types of y (formal argument) and x (returned value) correspond to C's int.

- When using C-- *labels* in foreign language programs, the type of these labels should match a particular foreign language pointer type. For example, label foo in Figure 4 has native data-pointer type bits$n$ for a given n, and it should be equivalent to C's *int.

- When importing names belonging to foreign language programs, there are two cases to consider:

1. **Foreign *global* variable names.** These names are considered as *labels* that point to the location in memory where the value of the variable is stored—therefore, imported variable names have native data-pointer type. Inside the `C--` program, the type of the values read from or stored to those foreign variables should match the variable's foreign type. For example, if a `C--` program imports two C (global) variable names, say `i` and `p`, each declared as `int` and `*int` respectively, to read the values stored in them, the read expressions should be `bits32[i]` and `bits64[p]` if an `int` is 32-bit long and 64 bits are needed to hold a pointer to an integer in the particular architecture. The following `C--` read expression would fetch the value pointed to by `p`

   ```
   bits32[bits64[p]]
   ```

   Note that it does not make sense to import foreign local variables.

2. **Foreign function or procedure names.** These names are considered as *procedure names* that stand for the foreign function/procedure entry point—therefore, imported procedure names have native code-pointer type. Such names may be used in expressions and call statements, *but not* `injumps`.[6]

For any architecture there will be a correspondence between the foreign language types and the `C--` types. The reader should refer to the manual for the particular implementation. Note that inter-operation makes a `C--` program more architecture-dependent due to the requirement of type matching between the `C--` program and the foreign program *in a particular architecture.*

10 Acknowledgements

Some examples and ideas used in this document belong to Professor Simon L. Peyton Jones. Mistakes are only my own.

[6] At the moment, `jump` uses only `C--`'s own calling convention.