Interfacing Prolog with External Programs

This tutorial will show how the SWI-Prolog interpreter can be invoked from a C program.

1. Compiling, linking and running a native C program with external libraries

Building a C program requires several phases that are automatically performed by the building tools (in our case, Visual Studio). In order to create the interface between C and Prolog programs, we need to understand first the separation between the **compilation** and the **linking** phase from the build process.

In the **compilation** phase, the program's source code is transformed into the binary format that is understood by the processor (0s and 1s). If a call to an external function is made (for example we call `printf`), the compiler doesn't care where this function is implemented, just that this function exists. In order to tell the compiler that `printf` exists, we include the `stdio.h` header in our program. `stdio.h` doesn't contain the actual implementation, just the function definition. The binary code generated by the compiler will call a function named `printf`, although it doesn't know yet where the function is.

In the **linking** phase of the build process, the program components will be put together in order to generate the final executable. To continue with the `printf` example, we must tell the linker to also include `msvcrt.lib` at this stage (this stands for MicroSoft Visual C RunTime). This external library may contain the actual code for the desired functions or just a stub that calls them from an external DLL (Dynamical Link Library). In either case, the linker will associate the `printf` symbol with the actual implementation and will be able to build the final executable.

The `printf` function that we discussed before is not implemented in `msvcrt.lib`, but in a dynamical link library, called `msvcrt.dll`. This DLL will be loaded at run-time (when the program is actually executed). For the operating system to know where to find it, the folder that contains it must be in the PATH environment variable.

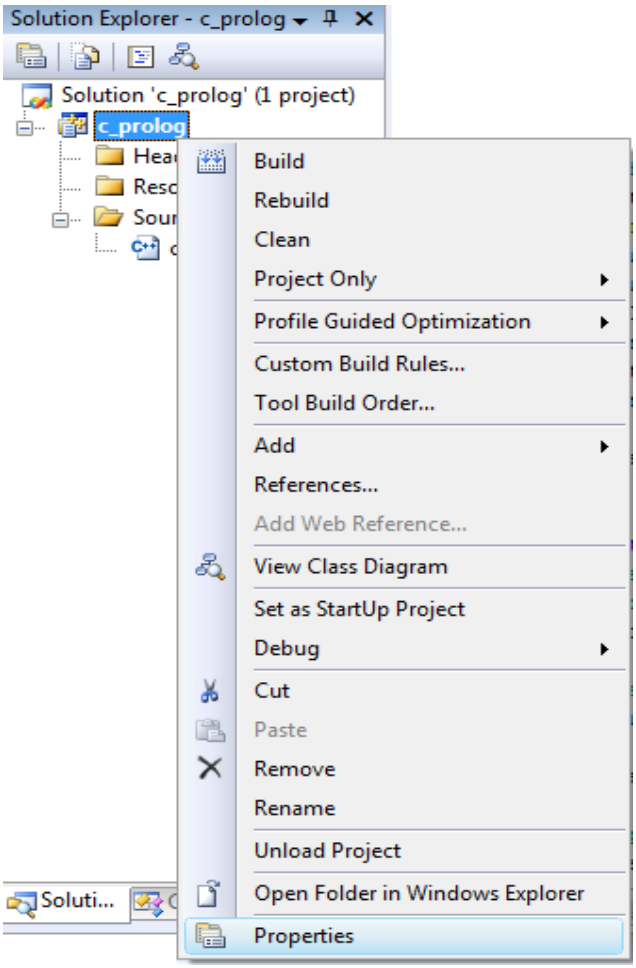
2. Setting the Visual Studio Environment to build and run programs that interface with Prolog

SWI-Prolog offers headers to include, static libraries to link and dynamic link libraries to link at run-time, in order to interface with C programs:

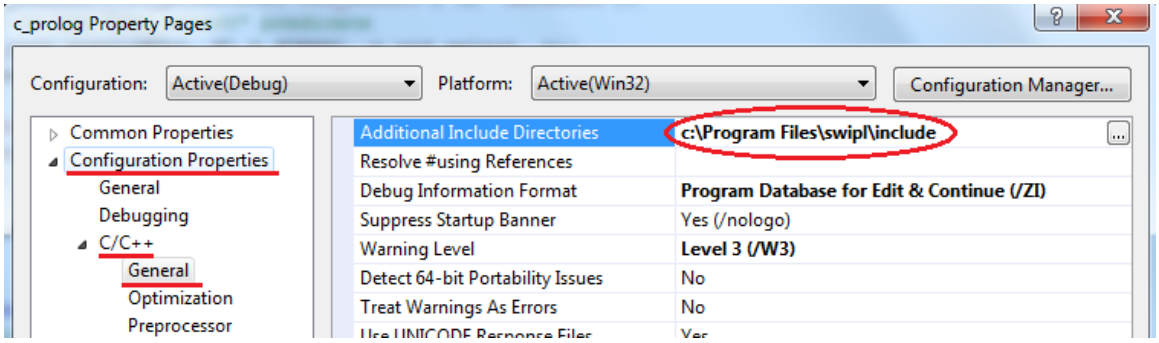
- the header files are located in *c:\Program Files\swipl\include*. We will use `SWI-Prolog.h` (or `SWI-cpp.h` for the C++ library)
- the static libraries are located in *c:\Program Files\swipl\lib*. We will use `libswipl.dll.a`, where the stubs of the actual implementations are present.

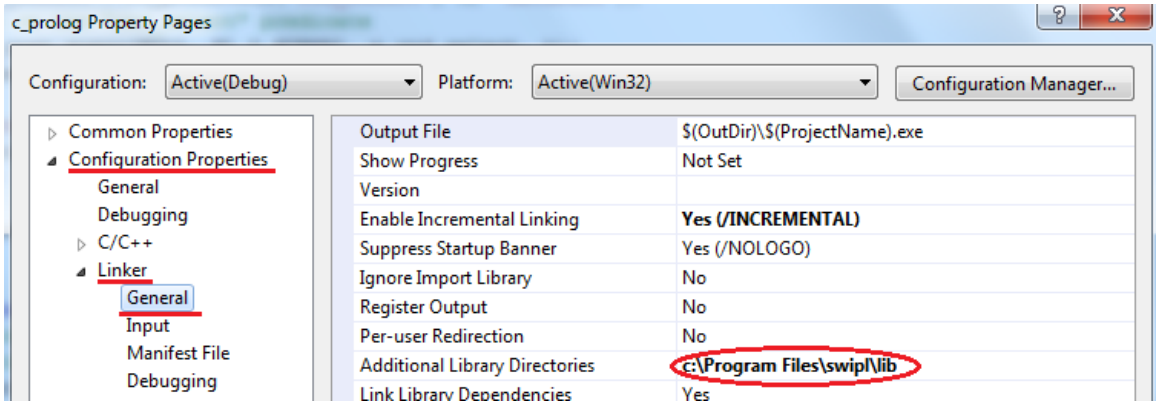
- the dynamic link libraries are located in *c:\Program Files\swipl\bin*. Some of the .dll files from there will be loaded at run-time by our program.

We will edit a project's properties by right-clicking it in the Solution Explorer, then selecting the Properties option.

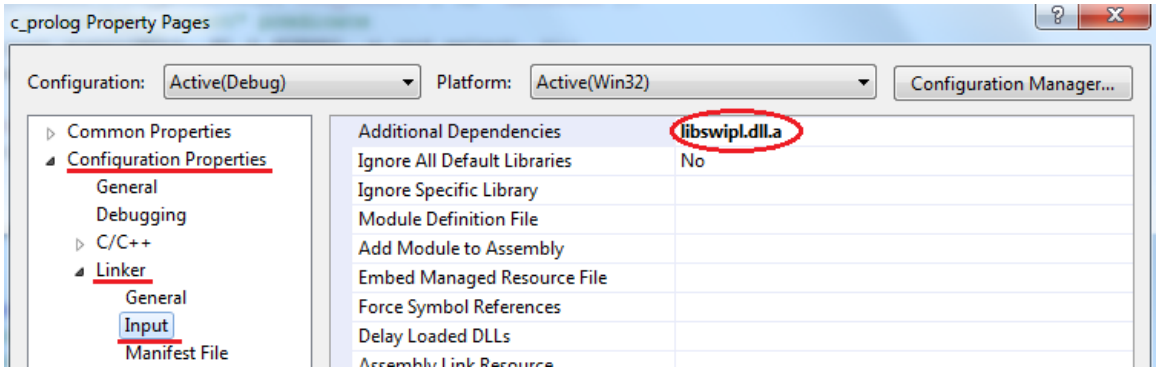


Since our project will include `SWI-Prolog.h`, we need to specify its location: in the Properties window, select *Configuration Properties* → *C/C++* → *General*, then type '*c:\Program Files\swipl\include*' in the *Additional Include Directories* field.

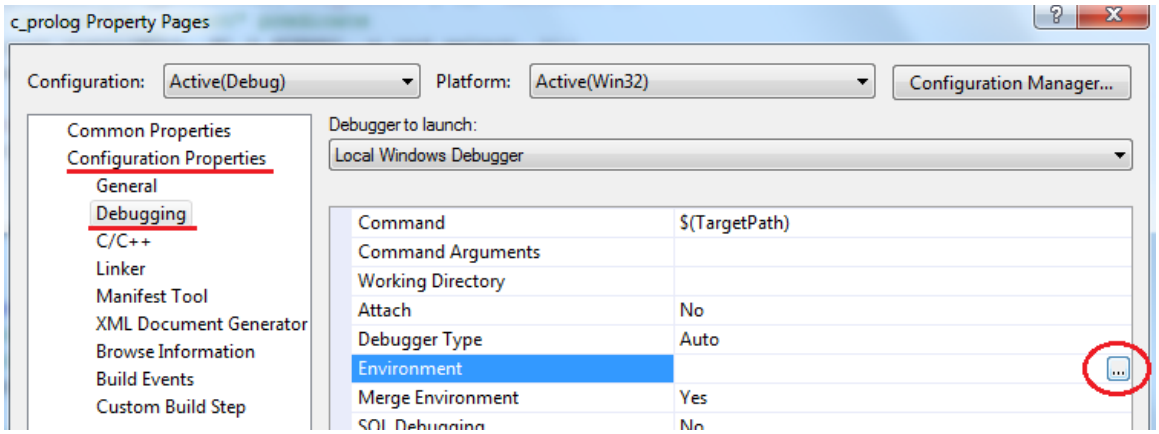




To link with the `libswipl.dll.a` static library, we must include its location (‘*c:\Program Files\swipl\lib*’) in the *Additional Library Directories* field, at *Configuration Properties* → *Linker* → *General* and its name in the Additional Dependencies field, at *Configuration Properties* → *Linker* → *Input.*



Finally, to run or debug the program in Visual Studio, some environment variables must be set. In *Configuration Properties* → *Debugging*, click on the *Environment* field, then on the … button in the right to open the multi-line edit window.



In the new window, type the following lines:

```
PATH=c:\Program Files\swipl\bin;%PATH%
SWI_HOME_DIR=c:\Program Files\swipl
```

The first line adds SWI-Prolog's dll's directory to the system's path. The operating system will now be able to load our executable because all the libraries it depends on will be found. The second line will set the environment variable `SWI_HOME_DIR`, to the prolog installation path, so various resources can be loaded at run-time.
Observation: If you have a 32-bit Prolog installation on a 64-bit Windows, it will be found in *Program Files (x86)* instead of *Program Files* dir.

3. Calling a Prolog built-in predicate from C

In order to use the provided Prolog interface, a C program must include the `SWI-Prolog.h` library:

```
#include <SWI-Prolog.h>
```

At the beginning of the main() function, we must initialize the Prolog engine:

```
int main(int argc, char **argv){
      PL_initialise(argc, argv);
      ...
```

The initialization function will fail if the environment variable `SWI_HOME_DIR` is not set correctly.

In order to call a Prolog predicate, we must get a reference to it, using the function:

```
predicate_t PL_predicate(const char *name, int arity, const char* module)
```

For instance, to get a reference to the predicate `plus` with arity 3 from the Prolog database, we will call:

```
predicate_t p_plus = PL_predicate("plus", 3, "database");
```

Before actually making the call, we also need to define the predicate terms. A term reference can be obtained using the function

```
term_t PL_new_term_ref()
```

for a single term, or

```
term_t PL_new_term_refs(int n)
```

for several terms.
Since the plus predicate has 3 terms, we will ask for a sequence of 3 terms:

```
term_t t = PL_new_term_refs(3);
```

In order to address the first term in the sequence we will write `t`, for the second term `t+1` and for the third term `t+2`.
We will try to use the `plus` predicate to solve the equation *2+X=5*. In the Prolog terminal, we would write *`plus(2, X, 5)`*. To do the same thing in C, we must specify the type and value for each term. The first and the third terms are integers, taking the values 2 and 5, so we will write:

```
PL_put_integer(t, 2);
PL_put_integer(t+2, 5);
```

The second term is a variable that will contain the query result:

```
PL_put_variable(t+1);
```

To call a predicate, we will open a query using the function:

```
qid_t PL_open_query(module_t ctx, int flags, predicate_t p, term_t +t0)
```

For our example, we will call:

```
qid_t query = PL_open_query(NULL, PL_Q_NORMAL, p_plus, t);
```

A query can have zero, one, or several solutions. They can be iterated using the function:

```
int PL_next_solution(qid_t qid)
```

This function returns TRUE if a solution has been found and FALSE if there are no more solutions. In case the query succeeded, the variables should be unified with the correct results. In order to extract the result of our plus query, we will call:

```
int result = PL_next_solution(query);
if(result){
      int x;
      PL_get_integer(t+1, &x);
      printf("Found solution %d.\n", x);
}
```

In case more solutions are expected, they can be retrieved by calling `PL_next_solution` in a do-while loop.

Finally, a query should be closed using the following function:

```
void PL_close_query(qid_t qid)
```

To wrap-up the above example, the following code can be used to solve the equation 2+X=5:

```
predicate_t p_plus = PL_predicate("plus", 3, "database");
term_t t = PL_new_term_refs(3);
PL_put_integer(t, 2);
PL_put_variable(t+1);
PL_put_integer(t+2, 5);
qid_t query = PL_open_query(NULL, PL_Q_NORMAL, p_plus, t);
int result = PL_next_solution(query);
if(result){
      int x;
      PL_get_integer(t+1, &x);
      printf("Found solution %d.\n", x);
}
PL_close_query(query);
```

4. Consulting an external source

To consult an external source file, we can call the built-in predicate `consult` from the C program. The following example can be used to consult the source *mylib.pl*:

```
predicate_t p_consult = PL_predicate("consult", 1, "database");
term_t t = PL_new_term_ref();
PL_put_string_chars(t, "mylib.pl");
PL_call_predicate(NULL, 0, p_consult, t);
```

The function `PL_call_predicate` is a shorthand for opening a query, calling for the first solution, then cutting the query.

The predicates defined in the external source can be used the same way as any built-in predicate from Prolog.

5. Working with lists

A Prolog list is different than atomic types, which have C correspondents. In order to work with a list in C, one needs to decompose it into head and tail.

5.1. Creating a list

In order to create a Prolog list in C, we need to start with the empty list then add each element, from the last to the first, using the function:

`PL_cons_list(lst, h, t);`

The function constructs the list `lst` in the first argument, from the head `h` in the second argument and the list `t` in the second argument as tail.

Assuming the term `lst` has already been created, the following code constructs the list from the elements of the vector `v`:

```
int i;
term_t h = PL_new_term_ref();
PL_put_nil(lst); //initialize with the empty list
for(i=n-1; i>=0; --i){
      PL_put_integer(h, values[i]);
      PL_cons_list(lst, h, lst); //add h in front of the list
}
```

In the 3rd line, the list is initialized with the empty list ([], or nil). Each element of the vector, starting with the last one is put into the term `h` (line 5), which is then added at the beginning of the list (line 6).

A list constructed this way can be fed to a Prolog predicate.

5.2. Traversing a list

In order to traverse a list, we must deconstruct it into head and tail, using the function:

`PL_get_list(lst, h, t);`

The function does the opposite of `PL_cons_list()`. If the resulting tail is not empty, it can be further deconstructed in order to retrieve the remaining elements.

The following code will print all the elements in a list of integers:

```
term_t tail = PL_copy_term_ref(lst);
term_t head = PL_new_term_ref();
int x;
while(PL_get_list(tail, head, tail)){
      PL_get_integer(head, &x);
      printf("%d", x);
}
```

The first line copies the list reference into a new term called `tail`. In each iteration, the tail is further deconstructed into the head, which is printed and a new tail. In order to print the term `head`, we need to extract the integer `x` from it.

6. Problems

6.1. Study the source code of **c_prolog**. In the first part, the equation 2+X=5 is solved, using the built-in `plus` predicate. The second part prints all the decompositions of a list,

using the `append` predicate. Finally, a predicate that extracts K random elements from a list is called. The predicate `rnd_select` is called from an external **file mylib.pl** and it uses the `myrand` predicate which is written in C.

6.2. *Sudoku:* A sudoku solver implemented in Prolog is given. It will take as input a list `Rows` that contains 9 lists, each representing a 9-elements row from a sudoku grid. Each element of the inner lists can be an integer from 1 to 9 or a free variable that will unify with the correct digit in the solution. Complete the C implementation of a program that reads a sudoku problem from a file, calls the Prolog engine and prints the solution on the screen.

6.3. *Wolf-Goat-Cabbage:* Consult your implementation for the Wolf-Goat-Cabbage problem from a C program in order to pretty-print the solution.