Green Hills Software Inc.

Mixing Ada & C/C++

Technical Report

Rev. 0.2 -- 11 June 1999

1. General Goals.

The Green Hills AdaMulti Integrated Development Environment develops applications written in Ada, C++, C and FORTRAN. This report describes how a customer can use AdaMulti to develop a single application written in multiple languages. In particular, this report will discuss mixing Ada with C and C++. A basic understanding of Ada and C/C++ is necessary for comprehending this report.

Some reasons to mix Ada with C/C++ are as follows:

- A customer has a large body of legacy code written in Ada and they want to do future development in C/C++.
- A customer prefers to develop applications in Ada, but is required to use a preexisting C/C++ library.

With some limitations, both cases are possible.

In general, an application is composed of one or more globally visible items. These items have a name that is associated with either the address of a data item or the address of a function entry point. The source code that generates these items is stored in one or more source files on the host development computer.

Each source file is written in either Ada or C/C++ and is processed by its own language compiler. The compilation process converts the source code to object code. The linker then combines these object code modules into a single executable image. Both Ada and C/C++ use this “separate compile and link” approach. As long as all source files are written in the same language, the compile and link process proceeds smoothly. Problems arise when some files are written in Ada and others are written in C or C++.

Let’s assume that a program is composed of two parts. One part is written in Ada and the other in C/C++. We might ask:

1. How does one part refer to the other part’s globally visible data?
2. How does one part dynamically create and subsequently refer to data items in the other part? How are those data items destroyed?
3. How can one part call the other part’s functions? How are arguments passed? How is the return value passed back?
4. Where is the program entry point? Is it “main()” in the C/C++ part or is it a procedure in the Ada part?

This report will investigate these questions in the following sections.

For the remainder of this report, the word “item” refers to an intrinsic data item, a composite data item, or a function. Further, the term “function” refers to both functions and procedures.

2. Globally Visible Data

A statically allocated data item occupies a fixed location in memory that can be determined at link time. A source file can define static data items that are visible to other source files or it can access static data items that are defined in other source files.

In C/C++, statically allocated data items are globally visible if they are defined outside of a function and the **static** qualifier is not present. A C/C++ function can refer to a data item that has been statically allocated in another source file by declaring that data item with the **extern** keyword. For example, the following C++ declarations create a data item visible to Ada and refer to a data item Ada has created.

```
extern struct adastuff FromAda2Cpp;      // Import from Ada to C++
struct cppstuff FromCpp2Ada;             // Export from C++ to Ada
```

A similar code fragment accomplishes the same thing in C:

```
extern struct adastuff FromAda2C;        // Import from Ada to C
struct cstuff FromC2Ada;                 // Export from C to Ada
```

Ada programs make statically allocated data items globally visible to C/C++ by using **pragma export** statements. An Ada procedure or function can refer to a data item that has been statically allocated in a C/C++ source file by using a **pragma import** statement. Examples are:

```
pragma import(CPP, CPP_HIS_STUFF, “FromCpp2Ada”);
pragma export(CPP, CPP_MY_STUFF,  “FromAda2Cpp”);

pragma import(C, C_HIS_STUFF,  “FromC2Ada”);
pragma export(C, C_MY_STUFF,   “FromAda2C”);
```

These Ada statements associate a local Ada name with a global C/C++ name. Note that an Ada name can contain embedded dots. This feature of the Ada language is incompatible with C/C++, since C/C++ uses the dot character to indicate structure, union and class membership. These Ada **pragmas** perform the important functions of associating an internal Ada name with an external symbol and translating the external symbol into a form that is compatible with C/C++.

Ada and C/C++ can share any data type if the machine representations are binary compatible. The Ada package INTERFACES.C is useful for enforcing this correspondence. The following table shows this correspondence between Ada and C++ types.

Ada type	**C/C++ type**
Integer	int
Interfaces.C.int	int
Short_Integer	short
Interfaces.C.short	short
Long_Integer	long
Long_Long_Integer	long
Interfaces.C.long	long
Character	char
Interfaces.C.char	char
Interfaces.C.plain_char	char
Interfaces.C.unsigned_char	unsigned char
Interfaces.C.signed_char	signed char
Byte_Integer	signed char
Short_Short_Integer	signed char

Interfaces.C.unsigned	unsigned int
Interfaces.C.unsigned_short	unsigned short
Interfaces.C.unsigned_long	unsigned long
Float	float
Interfaces.C.C_float	float
Long_Float	double
Interfaces.C.double	double
Interfaces.C.long_double	double
Interfaces.C.char_array	char *
Interfaces.C.wchar_t	wchar_t

Ada **access** types map to corresponding C++ pointer types.

Ada, C and C++ can share structures, unions and arrays as well. Note that all Green Hills Ada, C and C++ compilers allocate members of records and structures in the order that the members are declared. They also impose the same alignment requirements on the members. These two facts taken together ensure that if structures have the same number and type of members, they will be binary compatible.

An example code fragment of Ada accessing a C++ data structure is as follows:

C++ SIDE

```
class cpprec                            // Define the structure
{
    int numitems;
    float weightofitems;
};

cpprec cpp_name;                        // Declare the shared structure
```

ADA SIDE

```
type adarec is                          -- Define the structure
record
    itemcount  : INTEGER;
    itemweight : FLOAT;
end record;

ada_name : adarec;                      -- Declare the shared structure

pragma import(cpp, ada_name, “cpp_name”);-- Tie the two names together
```

Now, any reference by the Ada code to “ada_name.itemcount” will access the same integer memory location seen by C++ as “cpp_name.numitems”.

If the Ada record is tagged, then the C++ definition of the record needs to allow space for the Ada record tag. A dummy variable of type **void*** should be inserted in the class definition before the first data member. The following code fragment shows this:

C/C++ SIDE

```
class cpprec                                // Define the structure
{
    void* tagspace;                         // Allow space for the Ada tag
    int numitems;
    float weightofitems;
};

cpprec cpp_name;                            // Declare the shared structure
```

ADA SIDE

```
type adarec is tagged                       -- Define the structure
record
    itemcount  : INTEGER;
    itemweight : FLOAT;
end record;

ada_name : adarec;                          -- Declare the shared structure

pragma import(cpp, ada_name, “cpp_name”);-- Tie the two names together
```

Note that the Ada definition of records shared with C++ does not have to allow for the vptr.

C/C++ code cannot directly access global data items from an existing Ada package since Ada global symbols have embedded dot characters. In this case, the programmer must use an Ada **pragma export** statement for every data item that needs to be visible from C/C++.

1. Globally Visible Functions

There are two closely related programming concepts: function and procedure. A procedure is simply a function that does not return a value. In Ada, a subprogram is either a procedure or a function. C/C++, on the other hand, only supports the concept of a function. However, a C/C++ function can be declared as returning **void**; in which case, this is effectively the same as an Ada procedure. The **void** declaration simply tells the compiler not to expect a return value.

There are many different kinds of functions in both Ada and C/C++. The following common function types can be used from either side.

1. Globally visible functions – Ada can call C/C++ global functions and C/C++ can call Ada global functions.

2. Member functions – Ada can call C++ member functions. C++ cannot call Ada dispatching operations.

The Ada subprograms that cannot be accessed from C/C++ are the dispatching operations and the INITIALIZE, FINALIZE and ADJUST functions of controlled types. All C/C++ functions are accessible from Ada, although C++ constructors and destructors should not be accessed directly from Ada.

As with global data, the Ada **pragma export** and **pragma import** statements associate the Ada function name with a C/C++ function name.

An example code fragment of an Ada call to a C++ function follows:

ADA SIDE

```
function CPROC(input : in integer) return integer;
pragma import(CPP, CPROC, “c_proc”);
result : integer;
...

result := CPROC(123); -- Put answer in variable “result”
```

C/C++ SIDE

```
int c_proc(int input)
{
    int tmp;          // Accumulate the result here.

    ...               // Do some processing here.
    return tmp;       // Return the result.
}
```

For the case of functions, the Ada compiler uses C++ name encoding to generate the actual global symbol. The programmer should ensure that the types of the arguments match between the declaration of the Ada function and the C++ function. For record and enumeration argument types, the case-sensitive base name of the types must match between the Ada definition and the C++ definition. The actual specification of the argument type does not matter. The table in section 2 shows the association of the argument types between Ada and C/C++. If the argument types do not match between the C++ and Ada declarations of a function, there will be an undefined symbol at link time.

In the above example, the actual external symbol generated for “c_proc” is “c_proc__Fi”. Note that the **pragma import** statement gets the argument types from the definition of the Ada function but gets the name of the function from the third argument of the **pragma**. Also, note that this third argument is a string. This allows the programmer to specify legal C++ function identifiers that can’t be expressed in Ada; e.g., “MyClass::DoStuff”.

For the case of non-static C++ member functions, the Ada program must explicitly pass the “**this** pointer” to the C++ function. The Ada function or procedure declaration referenced in the **pragma import** statement must explicitly specify “**this**” as the name of its first argument.

C/C++ code cannot directly call global functions from an existing Ada package since virtually all Ada global symbols have embedded dot characters. In this case, the programmer must use a **pragma export** statement for every Ada function that needs to be visible to C/C++. For example:

```
procedure MyProc(MyArg : integer);
pragma import (C, MyProc, “c_funct”);
```

Note that the second and third arguments in the **pragma import** statement are not required to be the same. In practice, they often are.

For a description of C++ name encoding, refer to section 7.2c of: Ellis & Stroustrup, *The Annotated C++ Reference Manual*, 1990, Addison-Wesley Publishing Co., ISBN 0-201-51459-1.

2. Program Execution

Program Initialization

The main program can be written in Ada or C/C++. Green Hills supports either choice, so the programmer can pick whatever language is most applicable for the application at hand. In the case of both Ada and C/C++, the runtime system initialization code needs to be executed before the body of the main program starts. This initialization code is handled automatically and transparently when the program is written in a single language. With a mixed language system, the user's initialization code must explicitly call the run-time system initialization code before items in the other language are accessed.

If the main function is written in Ada and the Ada code references C++ items, then the Ada main program should call the function "_main()" before any reference to a C/C++ item.

If the main function is written in C/C++, then "main()" must call a function named "adainit" before the first reference to any Ada items. It must also call "adafinal" after the last reference to all Ada items.

For VxWorks, there can be multiple main programs running at one time. If the VxWorks main programs are written in C++, then the programmer should write a dummy Ada main program (e.g., named "adadummymain"). "adadummymain" should use **with** statements to include any package that the C++ code may need to call. The first C++ main program to run should then call "adadummymain_adainit". Further, one of the C++ main programs should call "adadummymain_adafinal" after the last call to the Ada code.

If all VxWorks main programs have been written in Ada and they reference a C++ library, then one of the Ada main programs should call "_main" before C++ as described above.

Exception Handling

Both Ada and C++ support their own version of exception handling. As a program runs, the run-time system maintains a "call stack". For example, function "main" calls function sub1(). Function sub1() calls function sub2() and function sub2() calls function sub3(). If function sub3() detects an exceptional condition, it may choose to "raise" (or "throw") an exception. This exception transfers control to sub1() for exception handling.

If all four functions are written in the same language, then the native exception handling mechanism will work as expected. Will this work if all the above functions are written in one language except sub2, which is written in the other language? If in the above example, sub2() was written in C++ and it was compiled with the "–Xnewexchandling" switch, then the answer is yes. Exception handling is a (possibly) non-local transfer of control up the call stack. To accomplish this, the compiler builds tables that are used by the run-time exception handling mechanism. The "–Xnewexchandling" switch causes the C++ compiler to build Ada-compatible exception handling tables.

If, in the above example, all functions were written in C++ and sub2() was written in Ada, then the answer is no. There is currently no mechanism in the Green Hills Ada compiler to build C++ exception handling tables.

3. Interface Functions

Many of the more complicated interface chores can be implemented by writing functions that act as an interface between C/C++ and Ada. A good approach is to establish a canonical set of interface functions for each C++ class. This set should include, as a minimum, a function to create an instance of the class, copy an instance of the class, and delete an instance of the class. All other member functions can be accessed directly.

For example, you might define a C++ class named MyClass which has a member function named "Print". The function "MyClass_AdaCreate" creates an instance of an object of type MyClass and returns a pointer to it.

ADA SIDE

```
-- Define and import the three canonical interface functions
--
function MyClass_AdaCreate return MyClass_AdaPtrType;
function MyClass_AdaCopy(arg : in MyClass_AdaPtrType)
                           return MyClass_AdaPtrType;
procedure MyClass_AdaDelete(arg : in  MyClass_AdaPtrType);

pragma import (CPP, MyClass_AdaCreate, "MyClass_AdaCreate");
pragma import (CPP, MyClass_AdaCopy,   "MyClass_AdaCopy");
pragma import (CPP, MyClass_AdaDelete, "MyClass_AdaDelete");

-- Define the normal member function "Print".  Note that the
-- name of the first argument MUST be "this".
--
procedure MyClass_Print(this : in  MyClass_AdaPtrType);
pragma import (CPP, MyClass_Print, "MyClass::Print");

-- Define the variables
--
Instance1 : MyClass_AdaPtrType;               -- Pointer to 1st object
Instance2 : MyClass_AdaPtrType;               -- Pointer to 2nd object

...

-- Executable code.
--
Instance1 := MyClass_AdaCreate;               -- Create the object
Instance2 := MyClass_AdaCopy(Instance1);      -- Make a copy
MyClass_Print(Instance1);                     -- Print Instance1
MyClass_Print(Instance2);                     -- Print Instance2
MyClass_AdaDelete(Instance1);                 -- Delete Instance1
MyClass_AdaDelete(Instance2);                 -- Delete Instance2
```

C++ SIDE

```
// Define the three canonical functions to create, copy,
// and delete objects of type MyClass.
//
MyClass* MyClass_AdaCreate(void)
{
    return new MyClass;            // Create a new one on the heap
}

MyClass* MyClass_AdaCopy(MyClass* src)
{
    return new (*src);             // Create a copy on the heap
}

void MyClass_AdaDelete(MyClass* deletethis)
{
    delete *deletethis;            // Delete this object
}
```

Note the **pragma import** statements in the above code fragments. The first parameter to the **pragma** specifies the external language. The second parameter to the **pragma** is the Ada name of the external function. The third parameter to the **pragma** is a string that specifies the fully qualified C++ name of the function. External symbol names in C++ are generated by a process called name mangling (or encoding). The Ada compiler knows how to generate this mangled symbol by using information from the preceding **procedure** declaration.

4. Things to avoid

Do not call Ada dispatching operations from C/C++.

Do not call C++ constructors or destructors directly from Ada. Create C++ interface functions to invoke **new** and **delete** for the object. See the example in the section on interface functions.

Don't attempt to call C++ virtual member functions via the C++ vptr/vtbl mechanism. Ada can directly call any C++ member function if the Ada program has enough knowledge to determine exactly which function to call. For example, a C++ base class B is defined with a member function "print". Two classes D0 and D1 are derived from B and each has a member function "print" as well. In this example, there are three member functions B::print, D0::print and D1::print. If the Ada program has a pointer to an object of type D0, it can call D0::print. If the Ada program has a pointer to an object of type D1, it can call D1::print. If the Ada program has a pointer to an object of type B, it doesn't know whether to call B::print, D0::print or D1::print.

5. Example

The following example implements a small symbol table class. A symbol table is a mechanism that stores items that contain name/value pairs. For each item, the name is a string of characters and the value is an integer. The name of the class is “symtbl” and it has a member function called “add”. The arguments to symtbl::add are the name string and the value. If the name string passed to symtbl::add is already in the table, the operation is discarded.

This example contains six files that fall into three categories:

File Name	Category	Description
symtbl.h	Class Library	C++ interface specification for class symtbl.
symtbl.cpp	Class Library	C++ implementation for class symtbl.
symtbl_ads.cpp	Ada Interface	C++ interface routines for Ada to access class symtbl.
symtbl.ads	Ada Interface	Ada specification for class symtbl
adamain.ada	Main Program	Ada main program.
cppmain.cpp	Main Program	C++ main program.

An Ada program can be build from the files symtbl.h, symtbl.cpp, symtbl_ads.cpp, symtbl.ads and adamain.ada. The C++ program can be build from files symtbl.h, symtbl.cpp and cppmain.cpp.

The C++ program creates a symbol table and stores some words in it. It then prints out a list of the stored words. Note that the duplicates are not printed. The Ada program does exactly the same thing as the C++ program.

```
////////////////////////////////////////////////////////////////////////////////
//  File: symtbl.h -- Interface for symbol table class
//
class symtbl;                   ////////// Symbol table class
class ste;                      ////////// Symbol table entry

class symtbl                            // Symbol Table
{
public:                                 // Everything is public for this example

    int  howmany;                       // Number of elements in the list
    ste* elements;                      // List head pointer
    ste* current;                       // The "current" table element

    symtbl();                           // Default constructor
    symtbl(const symtbl&);              // Copy constructor
   ~symtbl();                           // Destructor

    ste* add(char* name, int value);    // Add an element to a table
    ste* first(void);                   // Find the first element of a table
    ste* next(void);                    // Find the successor element
    ste* print(void);                   // Print the current element of a table
};

class ste                               // Symbol Table Entry
{
public:                                 // Everything is public for this example

    ste*  link;                         // Elements are linked in a list
    char* namestring;                   // Pointer to the name string
    int   value;                        // Associated value

    ste();                              // Default constructor
    ste(const ste&);                    // Copy constructor
    ste(char* name, int value);         // Normal constructor
   ~ste();                              // Destructor
};
```

```
////////////////////////////////////////////////////////////////////////////////
//  File: symtbl.cpp -- Body for symbol table class
//
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

#include "symtbl.h"

//  class ste member functions.
//
ste::ste()                                    // Default constructor
{
    link = NULL;                              // Init the link
    namestring = new char[10];                // Allocate memory
    if(namestring)                            // Default string
        strcpy(namestring, "$DEFAULT$");
    value = 0;                                // Default value
}

ste::ste(const ste& src)                      // Copy constructor
{
    link = NULL;                              // Init the link
    namestring =                              // Allocate memory
        new char[strlen(src.namestring)+1];
    if(namestring)                            // Copy the string
        strcpy(namestring, src.namestring);
    value = src.value;                        // Default value
}

ste::ste(char* nam, int val)                  // Normal constructor
{
    link = NULL;                              // Init the link
    namestring = new char[strlen(nam)+1];     // Allocate memory
    if(namestring) strcpy(namestring, nam);   // Copy the string
    value = val;                              // Save the value
}

ste::~ste()                                   // Destructor
{
   delete namestring;                         // Delete the string’s memory
}

//  class symtbl member functions
//
symtbl::symtbl()                              // Default constructor
{
    howmany  = 0;                             // No elements in the table
    elements = NULL;                          // Element list is empty
    current  = NULL;                          // No current element
}

symtbl::symtbl(const symtbl& src)             // Copy Constructor
{
    ste* p = src.elements;                      // Get listhead from src

    while(p)                                    // While something to do...
    {
        add(p->namestring, p->value);           // Add this new element
        p = p->link;                            // Point at the next one
    }
}

symtbl::~symtbl()                               // Destroy the table
{
    while(elements)                             // Something is in the list...
    {
        ste* p = elements->link;                // Get a pointer to next one
        delete elements;                        // Delete this element
        elements = p;
    }
}

ste* symtbl::add(char *nam, int val)
{
    ste* p = elements;                          // Get a pointer to the list
    ste* q = NULL;                              // Get a place to save the end

    while(p)                                    // Go look for it
    {
        if(strcmp(p->namestring, nam) == 0)     // Already here
                                return NULL;
        q = p;                                  // Remember this pointer
        p = p->link;                            // Link to the next one
    }

    p = new ste(nam, val);                      // Make a new one

    if(q)                                       // Something already on list
        q->link = p;                            // Put this new one on too
    else                                        // Nothing is on the list
        elements = p;                           // Make this new one first

    howmany += 1;                               // Show one more in the list

    return current = p;                         // Make this new one current
}

ste* symtbl::first(void)
{
    return current = elements;                  // Make list head current
}

ste* symtbl::next(void)
{
    if(current) current = current->link;        // Return next element pointer

    return current;
```

```
}

ste* symtbl::print(void)
{
    if(current)                                 // Only if current exists
    {
        cout << ’<’ << current->value << "> ";
        cout << current->namestring;
        cout << endl;
    }

    return current;
}
```

```
////////////////////////////////////////////////////////////////////////////////
//  File: symtbl_ads.cpp -- Symtbl interface routines
//
#include "symtbl.h"

//      This file contains the create/copy/delete interface routines.  Since
//      they need to be able to access the C++ storage allocation mechanism,
//      they are written in C++.
//
symtbl* symtbl_create(void)
{
    return new symtbl;
}

symtbl* symtbl_copy(symtbl* src)
{
    return new symtbl(*src);
}

void symtbl_destroy(symtbl* src)
{
    delete src;
}
```

```
--------------------------------------------------------------------------------
--  File: symtbl.ads -- Ada interface spec. for C++ symtbl package.
--
WITH Interfaces.C;

PACKAGE symtbl IS

    --  Declare some useful types.
    --
    TYPE ste        IS RECORD NULL; END RECORD;         -- Dummy entry record
    TYPE ste_ptr    IS ACCESS ste;                      -- Entry record pointer

    TYPE symtbl IS                                      -- Table record
    RECORD
        howmany  : INTEGER;                             -- How many in this table
        elements : ste_ptr;                             -- List head
        current  : ste_ptr;                             -- Current element
    END RECORD;

    TYPE symtbl_ptr IS ACCESS symtbl;                   -- Table record pointer

    TYPE char_ptr IS NEW Interfaces.C.char_array;       -- Pointer to C string

    --  Declare the create, copy and destroy interface functions.  Note
    --  that in the following three PRAGMA IMPORTs, the second and third
    --  arguments are the same.  This is just a convention and not a
    --  requirement.
    --
    FUNCTION  symtbl_create RETURN symtbl_ptr;
    FUNCTION  symtbl_copy(src : IN symtbl_ptr) RETURN symtbl_ptr;
    PROCEDURE symtbl_destroy(src : IN symtbl_ptr);

    PRAGMA IMPORT(CPP, symtbl_create,  "symtbl_create");
    PRAGMA IMPORT(CPP, symtbl_copy,    "symtbl_copy");
    PRAGMA IMPORT(CPP, symtbl_destroy, "symtbl_destroy");


    --  Declare the C++ member functions.  These functions are accessed directly;
    --  i.e., there is no intermediate interface routine.
    --
    FUNCTION symtbl_add  (this  : IN symtbl_ptr;
                          name  : IN char_ptr;
                          value : IN Interfaces.C.int)  RETURN ste_ptr;
    FUNCTION symtbl_first(this  : IN symtbl_ptr)        RETURN ste_ptr;
    FUNCTION symtbl_next (this  : IN symtbl_ptr)        RETURN ste_ptr;
    FUNCTION symtbl_print(this  : IN symtbl_ptr)        RETURN ste_ptr;


    PRAGMA IMPORT(CPP, symtbl_add,     "symtbl::add");
    PRAGMA IMPORT(CPP, symtbl_first,   "symtbl::first");
    PRAGMA IMPORT(CPP, symtbl_next,    "symtbl::next");
    PRAGMA IMPORT(CPP, symtbl_print,   "symtbl::print");

END symtbl;
```

```
--------------------------------------------------------------------------------
--  File: adamain.ada -- Ada main test program for C++ symtbl package
--
WITH text_io;
WITH symtbl;  USE symtbl;
with interfaces.c; use interfaces.c;

PROCEDURE symtbl_main IS

    PROCEDURE cpp_Initialization;
    PRAGMA IMPORT(C, cpp_Initialization, "_main");

    pt_my_table : symtbl_ptr;                   -- Ptr to a table
    tmp : ste_ptr;                              -- Ptr to a table entry

BEGIN

    cpp_Initialization;

    text_io.put_line("adamain -- Ada test driver for C++ symtbl package");

    --  Create a table
    --
    pt_my_table := symtbl_create;

    -- Add a bunch of entries
    --
    tmp := symtbl_add(pt_my_table, TO_C("the"),      1);
    tmp := symtbl_add(pt_my_table, TO_C("boy"),      2);
    tmp := symtbl_add(pt_my_table, TO_C("stood"),    3);
    tmp := symtbl_add(pt_my_table, TO_C("on"),       4);
    tmp := symtbl_add(pt_my_table, TO_C("the"),      5);
    tmp := symtbl_add(pt_my_table, TO_C("burning"),  6);
    tmp := symtbl_add(pt_my_table, TO_C("deck,"),    7);
    tmp := symtbl_add(pt_my_table, TO_C("eating"),   8);
    tmp := symtbl_add(pt_my_table, TO_C("peanuts"),  9);
    tmp := symtbl_add(pt_my_table, TO_C("by"),      10);
    tmp := symtbl_add(pt_my_table, TO_C("the"),     11);
    tmp := symtbl_add(pt_my_table, TO_C("peck."),   12);

    --  Print out the table.
    --
    text_io.put("The number of items in this table is:");
    text_io.put(INTEGER'IMAGE(pt_my_table.howmany));
    text_io.new_line;

    tmp := symtbl_first(pt_my_table);

    WHILE(tmp /= NULL) LOOP
        tmp := symtbl_print(pt_my_table);
        tmp := symtbl_next (pt_my_table);
    END LOOP;

    --  Clean up.
    --
    symtbl_destroy(pt_my_table);                -- Destroy my table

END symtbl_main;
```

The following is the console output for the Ada main program. Note that the duplicate values are not printed.

```
$ adamain
adamain -- Ada test driver for C++ symtbl package
The number of items in this table is: 10
<1> the
<2> boy
<3> stood
<4> on
<6> burning
<7> deck,
<8> eating
<9> peanuts
<10> by
<12> peck.
$
```

```
////////////////////////////////////////////////////////////////////////////////
//  File: cppmain.cpp -- Symbol table test driver
//
#include <stdlib.h>
#include <iostream.h>

#include "symtbl.h"

char* input[] =
{
    "the","boy","stood","on","the","burning","deck,",
    "eating","peanuts","by","the","peck.", NULL
};

main()
{
    symtbl mytbl;
    int i = 0;

    for(char** p = input; *p; ) mytbl.add(*p++, i+=1);

    cout << "cppmain -- C++ test driver for C++ symtbl package\n"
         << "The number of items in this table is: "
         << mytbl.howmany << endl;

    for(mytbl.first(); mytbl.print(); mytbl.next()) ;
}
```

The following is the console output for the C++ main program. Note that the duplicate values aren't printed.

```
$ cppmain
cppmain -- C++ test driver for C++ symtbl package
The number of items in this table is: 10
<1> the
<2> boy
<3> stood
<4> on
<6> burning
<7> deck,
<8> eating
<9> peanuts
<10> by
<12> peck.
$
```