Walker:

Working with a Multi-button Keypad in GtkAda

Rick Duley

Perth, Western Australia
2009

Table of Contents

Walker: Working with a Multi-button Keypad in GtkAda

Rick Duley
Perth, Western Australia

1 Preamble

The author would like to acknowledge the gallant and patient assistance of the people from AdaCore, the Gnat Academic Program, the GtkAda Mailing List, CompLangAda newsgroup, and others without which
he would not have made it this far. This document is humbly presented in the hope that its availability w ill lessen those people's future torment. Thanks folks!

2 Introduction

Many, maybe most, Graphical User Interfaces (GUIs) use buttons. They appear on their own, they appear in rows, sometimes they appear in keyboards. They have text labels, they have icons. They are one of the usual means of GUI control of an application.

This exercise is actually much more than an exercise with Gtk.Button. In this exercise you will create in verbose Ada code a keyboard of four buttons, set them with icons instead of labels and make them move the GUI around in the screen.

Although the GUI in this application is considerably simpler than the GUI in *"Gadget: Building a Compound Widget with GtkAda"*, the interesting material in this exercise is in making the application <u>do</u> something — and in <u>how</u> we make it do something. This exercise deals with the use of constants, arrays of widgets, arrays of Unbounded_String, access to procedure types, arrays of access to procedure , and a list of other things.

For resolution of matters not dealt with here go to http://libre.adacore.com /libre/tools/gtkada/ and register with the GtkAda Mailing list. (Go to the bottom of the page and the paragraph beginning, *"Note that the …"*.

2.1 Glade

GtkAda from AdaCore comes with an automated developer called Glade. This lets a programmer create a Graphical Unser Interface interactively, then it creates the code to suit. I unashamedly used this package to learn many of the rudiments of GtkAda programming.

However, the true power and range of the GtkAda package is only realised by manually coding it, and learning some of the skills required to do that is the object of this exercise. One of those skills is to write code so that maintenance people can understand it. Reading machine-written code in general, and Glade's is no exception, is like trying to divine something by gazing at the entrails of a hen.

You can read about Glade and the way it works in the GtkAda User Guide under the heading *"Support for Glade, the Gtk GUI builder"*.

2.2 Why Verbose Code?

One of the problems with newcomers learning GtkAda by working with Glade is that Glade code does not specify the origin of the widgets and things it uses. Glade religiously implements the *'use'* clause. Consequently, beginners do not learn to work with the GtkAda Reference Manual. Because of Glade they get code that works, but they do not get to know how it works.

Verbose style programming changes this because the relevant package is named at each procedure or function call. Furthermore, many of the procedures and functions implemented are inherited from ancestor widgets. Programmers must learn to trace the ancestry of the current widget to find the appropriate operation.

Verbose style programming means a lot of typing, but, in the end, it is quicker for a beginner (and miles better for the maintainer). Programmers tend to think more carefully about each step. Therefore, they avoid unending confusion. Time spent typing is more than compensated for with time saved debugging.

2.3 Naming Parameters

Meticulously naming formal parameters and assigning values to them forces programmer to think carefully about those values as well. GAP's GPS 2009 (see paragraph 3) is a great help here because it offers the parameters of a function or procedure call as you write. That certainly saves having to look up the Reference Manual for a parameter list every five minutes.

Another error, which is frequent when assigning parameter values by association, is that of getting the parameters in the wrong order. This glitch can be one of the hardest to find. This virtually never happens in verbose programming, and if it does parameter order doesn't matter as long as the Actuals match the Formals.

2.4 Initializing Scalar Variables

If you are not in the habit of initializing variables, try a test program like Code Sample 1. Some compilers are sensitive to uninitialized variables and this can cause problem when attempting to produce a 'Release' version of the executable.

```
with Ada.Text_Io;
use Ada.Text_Io;
with Ada.Integer_Text_Io;
use Ada.Integer_Text_Io;
procedure Test_Unitialized_Variables is
   Int : Integer;
   Char : Character;
begin
   Put(':');
   Put(Int);
   Put(':');
   New_Line;
   Put(':' & Char & ':');
end;
```

Code Sample 1: Test Uninitialized Variables

Note well: it pays to initialize variables. In this exercise initialization is carried to an extreme — access types are automatically initialized to '**null**' — just to make the point.

2.5 Connotative Identifiers

Make sure the names of the widgets are distinctive and self-explanatory at the time you draw your Mud Map (see page 3), and use them when you program. Call the widgets anything you wish, but make sure the name describes what the widget or variable does. Do not accept the identifiers I used as any ultimate.

This idea of the use of connotative identifiers is one of the foundation principles of Ada Programming — which is what you will be doing in this exercise. The Style Manual still offers these suggestions:

- *choose names that are as self-documenting as possible;*
- *use names given by the application;*
- *do not use obscure jargon;*
- *avoid using the same name to declare different kinds of identifiers;*
- *do not use an abbreviation of a long word when a shorter synonym exists;*
- *use a consistent abbreviation strategy;*
- *do not use ambiguous abbreviations;*
- *to justify its use, an abbreviation must save many characters over the full word;*
- *use abbreviations that are well-accepted in the application domain; and*
- *maintain a list of accepted abbreviations, and use only abbreviations on that list.* (Software Productivity Consortium, 1995)

2.6 On Being Methodical

All programming is methodical in nature — graphics programming especially so. Consider the exploded construction diagram on page 8. ***Don't Panic!*** For the moment, note only that it starts with the base widget (the window itself) and progressively, working to the right, adds further widgets. In turn, each of these widgets is created, its characteristics are defined, and it is added to the widget to its left. (There are other things that go on but they are not important just yet.) Keep this idea of being methodical in mind. It is the basis of the whole process.

As St Paul put it,

"Let all things be done decently and in order."— 1 Corinthians 14:40.

2.6.1 The Plan

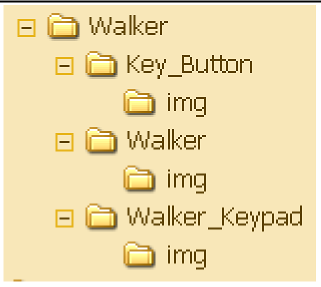
1. Create a button that will know where it is in a keypad;
2. Create a keypad of these buttons with each button aware of its position;
3. Use the keypad in an application widget.

2.6.2 Development Folder Structure

This methodical approach is reflected in the structure of the development folders during application development under GAP's GPS 2009. There is one folder where the key_button is developed and tested, another where the keypad is developed and tested, and a third for developing and testing the application. Code in these folders is interlinked by the GPS program files (.GPR). Walker.gpr is headed by "with "./../Walker_Keypad /Walker_Keypad.gpr"; and Walker_Keypad.gpr is headed by with "./../key_button/Key_Button.gpr";. Each project file calls its predecessor. This permits completely individual testing of components.

Why do they all have image subfolders? Because the images — at least some of them — are needed at each development stage.

When using other systems, work in the one folder.


```
⊟ Walker
    ⊟ Key_Button
        img
    ⊟ Walker
        img
    ⊟ Walker_Keypad
        img
```
Figure 1: Development Folder Structure

3 Getting GtkAda

For a great start contact gap-contact@adacore.com and get registered with the GNAT Academic Program run by AdaCore. Then you can download the GAP Package which includes a quality Integrated Development Environment, complete with compiler and support packages including Win32 and GtkAda. The best advice is to download and install the lot using the default installation settings.

3.1 Setting Up GAP's GtkAda

Some modifications have to be made to GAP's GtkAda installation for PC. Full details are available at AdaSafeHouse.webs.com/GtkAda.html.

3.2 GtkAda Documentation

Full HTML documentation is included in the GtkAda package. If you have used all the default settings it can be found at c:\GtkAda\share\doc\gtkada\. I refer to these documents all the time so I have shortcuts to them on my desktop. Information on the functions of widgets and the meanings of formal parameters presented here is largely taken from these documents.

4 Preparations

Assuming that you have worked through *"Gadget: Building a Compound Widget with GtkAda"* you will know that building such a widget is like building something with LEGO blocks. You simply take the pieces and plug them together. However, the programmer must know exactly what is to be built before coding starts. Anything other than forethought and planning in fine detail is a recipe for frustration and futility. **Be warned!**

4.1 The Mud Map

Programmers should draw what Australians call a *Mud Map* before doing anything else. A Mud Map, strictly speaking, is a sketch scraped in the dirt with a finger or a stick. It's just something to get a rough idea. Draw that sort of a sketch of the layout to be achieved including all the visible and invisible widgets to be included in the compound widget. Leave nothing out, and name all the parts in a logical, distinctive and connotative manner. Remember, an exploded view of Walker is shown on page 8 and that may make Figure 2 more clear.

5 Create the Key

If you have worked through *"Gadget: Building a compound widget with GtkAda"* you will have few problems with the code in Gtk.Key_Button_Pkg starting on page 11. A Key_Button is merely an extension of Gtk_Button.

5.1 General Issues with Gtk_Key_Button

There are, however, some issues in the implementation of Gtk_Key_Button to which we must pay some attention.

5.1.1 Extending Gtk_Button

Keys in a keypad must carry information as to where they are in the pad. (It is up to the programmer to know what each key is supposed to do.)

There are two obvious ways of doing this: (1) keep a record of the row and column at which the button appears; (2) name the button for its position.

Both are employed for Key_Button. In fact, only the Row and Column numbers are used in this project, but a Software Engineer would utilize both sets so that the same Key_Button can be used in other projects in which the Name might be used.

5.1.2 Privacy

Ada Packages provide *'Module Encapsulation'* (see Barnes, 2006, p.42) by which the inner working of a package, as implemented in the package body, are not visible to the user. Declarations in the specification of a package are visible to the user, and can, therefore, be modified by the user. This is not necessarily a good thing from a Software Engineering point of view. In Walker, such access is definitely not desirable.

One thing about keeping the Row and Column numbers — they must stay unchanged when set. Imagine what would happen if keys in the pad behaved as if they were other keys! This means the developer does not want the user to change the values. At the same time, the differences between Key_Button_Record and ordinary Gtk.Button.Gtk_Button_Record must be known to the compiler. Ada's solution is to declare the actual extension of the record to be '**private**' (see line 9 of the Gtk.Key_Button_Pkg specification on page 11).

*The part of the package specification before the reserved word **private** is the visible part and gives the*

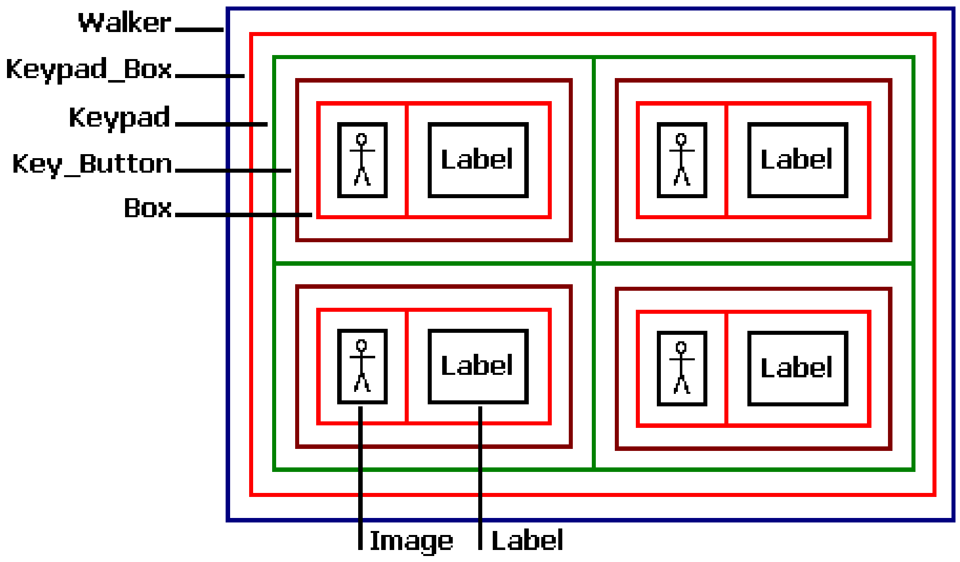


Figure 2: Walker Mud Map

> *information available externally to the package.. …* [The details of the extension are declared after the word **'private'**.] … *This means that outside the package nothing is known of the details of the type.* (Barnes, 2006, p. 221)

In other words, outside Gtk.Key_Button_Pkg or a child of Gtk.Key_Button_Pkg the record variables Row and Column cannot be addressed[1]. Package authors can provide for this by writing procedures and functions which will access the record entities. For the use of developers, the functions Get_Button_Row and Get_Button_Column are implemented at lines 87 and 94 of the body of Gtk.Key_Button (starting on page 13) and used in lines 32 and 33 of Walker_Keypad_Pkg .Local_Callbacks.adb on page 31. Don't worry too much about this yet.

While they may not be directly addressed from outside the package, the inner details of the widget can be read by developers using the package which gives them an understanding of those details. In Gtk_Key_Button, Label, Image, Box, Row and Column for Gtk_Key_Button_Record are defined in lines 51-55 in the specification of Gtk.Key_Button.

5.1.3 Using Access as a Parameter

Declaring the parameters to Get_Button_Row and Get_Button_Column as: **access** Gtk_Key_Button_Record'Class may seem a longwinded way of declaring them as Gtk_Key_Button, but it is necessary. See the discussion on the Google Group 'CompLangAda' starting on page 36. CompLangAda (http://groups.google.com/group/comp.lang.ada/topics) brings a variety of very wise heads together, and opinions vary. It pays to follow the thread for a while — and it pays to read right through this subset of responses to my request for an explanation — if you want to get to the bottom of a matter.

[1] Inside the package private entities can only be accessed for assignment or equality.

5.2 Implementing Key_Button

Here we work through the body of Gtk.Key_Button looking at the implementation issues.

5.2.1 Converting Integer to Character

Lines 36 and 36 in Gtk.Key_Button.adb starting on page 13 show how to convert an Integer into the corresponding character. This technique is an outcome of the definition of the standard type Character in the Ada Language Reference Manual (LRM) Section A.1.

The alphanumeric characters start at the character '0' (nought) with the position 48. So we use that offset of the position of the individual numeral to get the character position:

1. Let Row = 3
2. 3 + 48 = 51
3. The Character at position 51 in LRM A.1(35/2) is '3'

At line 46 in the body of Gtk.Key_Button that actual character is concatenated into the Name of the button.

5.2.2 Naming the Key_Button

For Gtk_Key_Button the naming convention is Row & '_' & Column. Therefore, if the button is at Row=1 and Column=1 its name is to be "1_1". This algorithm is implemented at line 45 in the body of Gtk.Key_Button.

5.2.3 Creating the Button Label

When the button is initialized at line 41 in Gtk.Key_Button.adb, the parameter *'Label'* is left at its default value of a null string. At inintalization, then, the button has no label. We want to create one and that is done at line 70; that is to say, it is done if *'Initialize"* receives a non-null UTF8_String as the actual for *"Label"*.

5.2.4 Including Optional Icons and Labels

It may be that the user of the package Gtk.Key_Button does not want an icon on the key_button (e.g. a key on a computer keyboard); it may be that the user wants only an icon (e.g. a *'STOP'* sign on a stop button). The Software Engineer will allow for both. This is

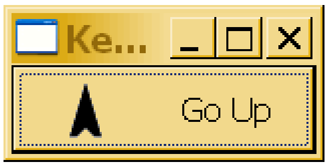


Figure 3: Key_Button_Test

achieved by the conditional statements at lines 58 and 68.

If the user has not included an actual Gtk.Image in the parameters for the call to Gtk_New (i.e. Image = null), no image is added to *'Box'*. In the same way, if no label text is specified in the parameters, no label is added to *'Box'*. If neither is specified you take your chances with the shape of the button.

5.3 Testing Gtk_Key_Button

So, you have built a successful Gtk.Key_Button.Gtk_Key_Button. How do you know?

It is always good practice to test a new package of anything in Ada when you have completed it — and, for larger packages, at stages through its development.

Note well: it is always easier to find bugs when the amount of code you have to search in is small. Frequent, incremental, testing is the salvation of many hair follicles.

5.3.1 Creating an Icon Image

There are several different versions of Gtk.Image.Gtk_New (see the GtkAda Reference Manual). The one used in line 19 of Gtk.Key_Button.adb will accept BMP, JPG, GIF, and PNG image files. In Gtk_Key_Button the images were GIF to take advantage of the transparency facility which shows the image without a background.

5.3.2 Sneak a Peek

Write a test program like the one starting on page 16. If your code is correct you should see something like Figure 3.

6 Create the Keypad

Walker's keypad has just four buttons. The trick here is to get the buttons into a Gtk_Table (Keypad) in a way in which we can address them. Our solution is to use three arrays: one for the images (the arrow symbols), one for the boxes holding the images and the third for the Table (Keypad) which presents the buttons neatly set out in the screen display. More of that as we go.

6.1 Keypad Issues

Some more urgent general points to consider:

6.1.1 Boxes for Widgets

It is a safe practice to place each widget separately in its own HBox or VBox. There are two related issues to deal with here:

1. The Gtk_Table (Keypad) is, to all intents and purposes, another kind of box — both Gtk.Table and Gtk.Box are direct descendants of Gtk_Container. It is simply a special box in which widgets are displayed in rows (down) and columns (across). Therefore, we do not need a box for the button.
2. Keypad forms the base of a distinct widget — Walker_Keypad. Why then use Keypad_Box to hold the keypad? It is certainly not necessary because Keypad's outer layer is, itself, a box. Simply, it is good practice because the programmer may not be aware of the nature of the outer layer of a widget. Do you know the nature of the outer layer of any of the GtkAda standard widgets?

6.1.2 Using Constants

It is not bad practice to use Global Constants from a package specification. In contrast, variables declared in a specification should be passed as parameters and not used Globally (Dale et al., 2000, p. 314).

Constants are replaced in the program by the value represented, so there is no need to look up that value each time as must be done for a Global Variable; hence there is an execution speed advantage. However, the greatest advantage in using them this way, rather than using Literal values, is that there is only one place where the programmer needs to change them should the need arise.

For example, if Walker needed an extra row of keys only KEYPAD_ROWS_COUNT (see line 13 in Walker_Keypad_Pkg.ads starting on page 17) would have to be changed[2]. That is a whole lot simpler than going through the entire program changing Literals, and also stops the generation of very strange and frustratingly obscure errors. Some errors like that are just funny, others cause programs to crash for no apparent reason. Avoid the hassles, **make use of Constants**.

6.1.3 Named Numbers

You will note that the two constants declared at lines 13 and 16 in Walker_Keypad_Pkg.ads do not have specified types. They have become what is known as *"Named Numbers"*.

It is usually good practice to omit they type when declaring numeric constants … But note that the type cannot be omitted in numeric variable declarations even when an initial value is provided." (Barnes, 2006, p. 75)

Integer numbers declared in a number declaration … such as

```
    Ten : constant := 10;
```

are also of type universal_integer. However, there are no universal_integer variables and no universal_integer operations. … since these expressions are all static and thus evaluated at compile time the distinction [from constants] *is somewhat theoretical … However, it is an important rule that a static expression may not exceed the base range of the expected type. (Barnes, 2006, p. 375)*

6.1.3.1 Advantages of Named Numbers

In lines 58 and 59 of Walker_Keypad_Pkg.adb the constants KEYPAD_ROWS_COUNT and KEYPAD_COLUMNS_COUNT are sent as parameters required to be of the type *'Glib.Gint'*. No type conversion is necessary as in lines 88 to 91 where Positive variable values have to be coerced into Gint. By not specifying the type KEYPAD_ROWS_COUNT can be used as any whole number type providing it fits in the range of that type.

[2] The compiler would also force the programmer to add more cells to some arrays and make other related changes.

6.1.4 Keys_Type

(See line 19 in Walker_Keypad_Pkg.ads)

There is no mechanism to address widget in a Table directly. Instead, the keypad has an array of key_buttons (Pad)which can be addressed directly. These are linked to Keypad for display. The link is by means of an access type, so the changes made to Pad will show, where applicable, in Keypad. In fact, the changes made from outside Walker_Keypad_Pkg are non-visible, the connection of callbacks. More of that later.

6.1.5 Privacy

In line 24 of Walker_Keypad_Pkg.ads the extension to Gtk_Table_Record is declared to be **private**. (See discussion in paragraph 5.1.2 on page 4.) Buttons in *'Pad'* need to be accessible for assignment when the user attaches callbacks to the buttons so a function Get_Walker_Keypad_Button is provided in line 44 of Walker_Keypad_Pkg.ads (for its use see line 68 of Walker_Pkg.adb).

6.1.6 Handling Callbacks for the Key_Buttons

Key_Button_Handler_Pkg is declared at line 37 in Walker_Keypad_Pkg.ads. The purpose of this package (which implements a variation of Gtk.Handlers.User_Callback) is to deal with a click on one of the buttons. It is declared locally so it has visibility of the Keypad.

6.2 Keypad Implementation

Initialization of the Walker_Keypad is simplified by use of a loop structure.

6.2.1 Storing Data for Individual Buttons

Perhaps the advantage over individual creation and assignment of the buttons in the pad is not great with a keypad of four buttons. Spare a thought, however, for the creator of a keypad of 100 buttons! This loop operation, however, requires a new data structure to hold data relevant to the individual buttons. This is Button_Arrow_Data_Type at line 11 in Wallker_Keypad_Pkg.adb

Constants of this type are referred to in lines 68 and 77 where they give the

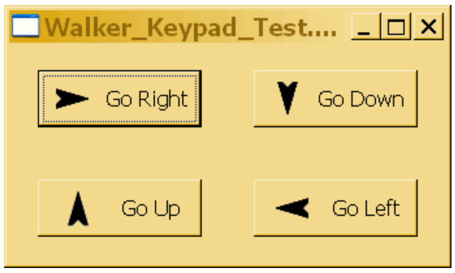


Figure 4: Walker_Keypad_Test

address of the arrow image and the label text for the button respectively. Two things to note:

1. Being declared in the body of the package, this type and these constants are <u>extremely</u> **private**;
2. This is another example of storing information in <u>one</u> place so that if it needs modification it only needs to be modified in one place.

6.2.2 <u>Sneak a Peek</u>

Write a test program like the one starting on page 23 and you should have something that looks like Figure 4. It still doesn't <u>do</u> anything, that is the next item in the plan.

7 Walker

Until now, this project has, with a few exceptions, been a repetition of the *'Gadget'* project, just playing with GtkAda widgets. This section is Ada rather than GtkAda and we deal with things like access types that access subroutines, and arrays of those types. We also dabble in the concept of inheritance as we set callbacks for the buttons on our keypad.

The callback to the window destructor (connected at line 35 of Walker_Pkg.adb) is normal and is handled at Walker_Pkg level. The relevant handler package is in Walker_Pkg_Callback_Pkg (see page 28) and the handler procedure (On_X_Button_Destroy) in Walker_Pkg.Callbacks starting on page 28. However, the callbacks from the buttons present a visibility problem.

7.1 Button Callbacks from the Keypad

Handling callbacks from the buttons in the keypad requires a callback package

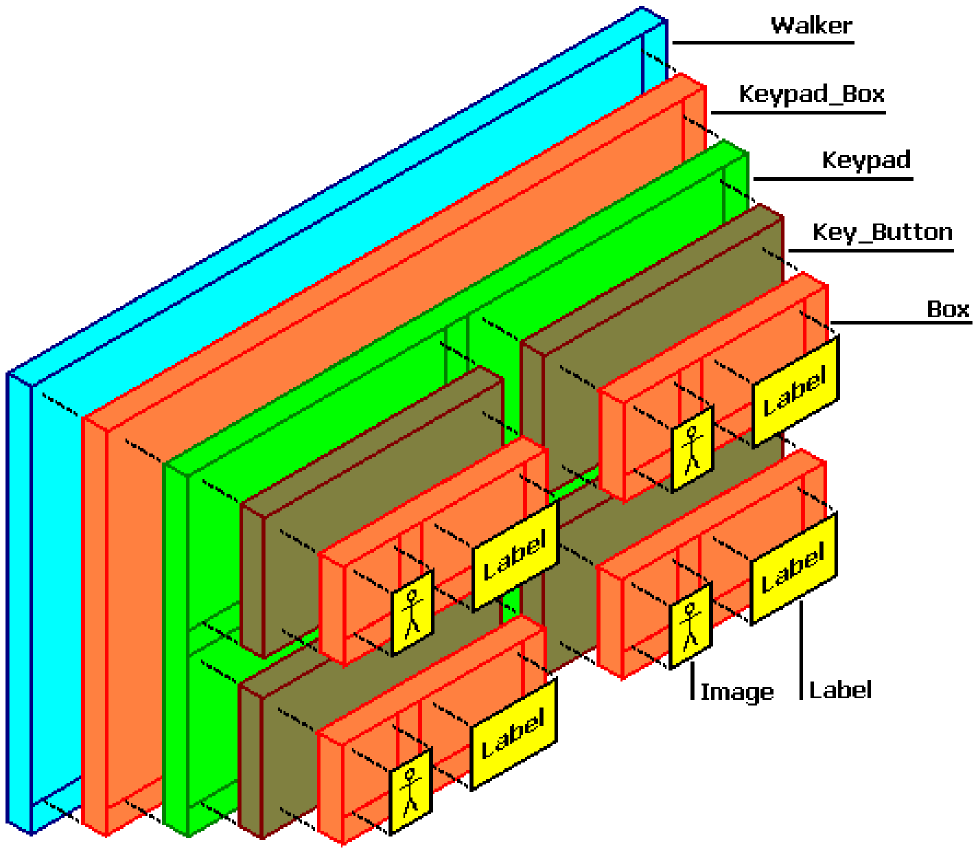


Figure 5: Walker - Exploded View

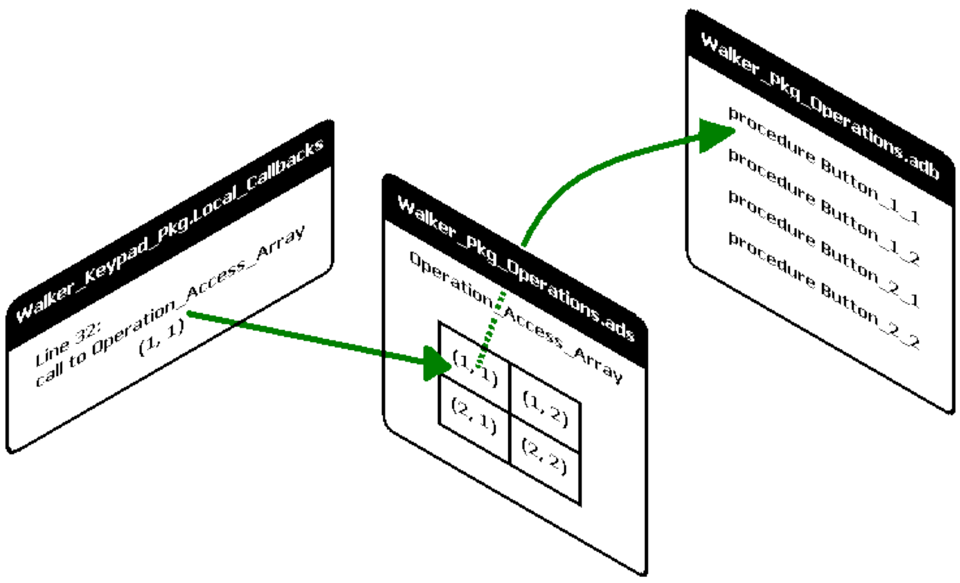


Figure 6: From Array Parameter to Procedure

which is a child of the keypad package. This is Walker_Keypad_Pkg.Local_Callbacks (see page 29) and the connection of the callback procedure at line 69 of Walker_Pkg.adb should be paid careful attention.

7.1.1.1 The Appropriate Callback Package

To handle messages from the keys in the keypad, this connection uses Key_Button_Handler_Pkg which is declared at line 37 of Walker_Keypag.Pkg.ads on page 18. This, remember, is a special version of Gtk.Handlers.User_Callback and it requires access to the keypad.

7.1.1.2 The Appropriate Callback Procedures

The callback procedures (see On_Walker_Keypad_Button_Click in Walker_Keypad_Pkg.Local_Callbacks on page 29) also need some special information:

1. The callback needs to know the coordinates at which the window is displayed on the screen. These are collected at lines 16 and 21 of Walker_Keypad_Pkg. Local_Callbacks;
2. It also needs the *'handle'* of the window (so that it can destroy the window and all it contains). This is collected at line 25.

7.1.1.3 Calling the Appropriate Procedures

Walker_Pkg_Operations.ads (see page 31) holds the key to how signals from the keys in the keypad are converted into actions on the displayed widget. The secret is the *'access to subprogram'* type described at LRM 3.10 (6/2). At line 13 in Walker_Pkg_Operations we declare the array type Operation_Access_Array_Type (which is a dimensional match for keypad). At line 35 we have the constant of that type Operation_Access_Array which serves as index into a list of available operations. In turn, this index is searched by the procedure call in line 32 of Walker_Keypad_Pkg.Local_Callbacks on page 31 Here the coordinates of the array are collected by calls to the functions Get_Button_Row and Get_Button_Column in Gtk.Key_Button and are relative to the button's position on the keypad. It sounds complicated, but, when you understand what is going on, it is a simple solution (especially for keypads with large numbers of keys). Figure 6 may clarify the issue.

7.2 Making Walker Walk

By default, window positions are defined by the top-left corner of the window. The coordinate system used for a screen in windows starts at (0, 0) in the top-left of the screen. The range used in Walker is:

- X = 100 → left;

- X = 400 → right;
- Y = 100 → top;
- Y = 400 → bottom.

So, wherever Walker might be on the screen, the *'Go Up'* key is to cause the window's Y-coordinate to be 100. The X-coordinate will remain unchanged. (See line 39 in Walker_Pkg_Operations on page 34.)

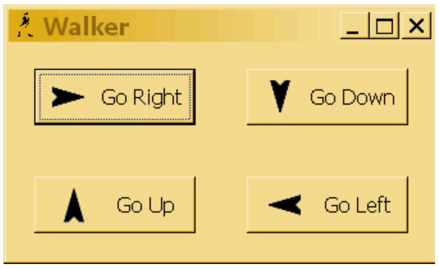


Figure 7: Walker - Complete

8 Finished

The mainline program is simple (see page 35. Walker should look something like Figure 7 and the window should move about the screen when the buttons are clicked.

9 Source Code

9.1 Key_Button

9.1.1 Key_Button Specification

```
with Gtk.Button;
with Gtk.Box;
with Gtk.Label;
with Gtk.Image;

package Gtk.Key_Button is

   type Gtk_Key_Button_Record is
     new Gtk.Button.Gtk_Button_Record with private;
   -- See private section.
   -- Gtk_Key_Button_Record stores the position of the Key_Button
   -- in a keypad by holding Row and Column number information
   -- as Positive.  Use functions Get_Button_Row and Get_Button_Column
   -- to retrieve that information.

   type Gtk_Key_Button is access all Gtk_Key_Button_Record'Class;
   -- Access to all Gtk.Key_Button_Record widgets.

   procedure Gtk_New  (New_Key_Button :    out Gtk_Key_Button;
                       Label          : in     UTF8_String := "";
                       Image          : in     Gtk.Image.Gtk_Image := null
                       Row            : in     Positive := 1;
                       Column         : in     Positive := 1);
   -- Creates new instance of Gtk_Key_Button_Record and sends
   -- it to Initialise.  Delivers initialized instance.

   procedure Initialize
     (New_Key_Button   : in out Gtk_Key_Button;
      Label            : in     UTF8_String := "";
      Image            : in     Gtk.Image.Gtk_Image := null;
      Row              : in     Positive := 1;
      Column           : in     Positive := 1);
   -- Initializes Gtk_Key_Button_Record setting internal parameters
   -- 'Row' and 'Column'.  If Image /= null, sets Icon from Image.
   --  Sets Label if sent one.  Sets Name of button according to 'Row'
   --  and 'Column'.  Convention is that, if 'Row' = 1, and 'Column' = 1
   -- then Name will be "1_1".  Stores button's position in keypad.

   function Get_Button_Row (Button : access Gtk_Key_Button_Record'Class)
                           return Positive;
   -- Fetches the value of Button.Row.

   function Get_Button_Column (Button : access Gtk_Key_Button_Record'Class)
                              return Positive;
   -- Fetches the value of Button.Column.

private
   type Gtk_Key_Button_Record is
     new Gtk.Button.Gtk_Button_Record with
      record
         Label  : Gtk.Label.Gtk_Label := null;
         Image  : Gtk.Image.Gtk_Image := null;
         Box    : Gtk.Box.Gtk_HBox := null;
         Row    : Positive := 1;
         Column : Positive := 1;
      end record;
end Gtk.Key_Button;
```

9.1.2 Key_Button Body

```
package body Gtk.Key_Button is

   -- Creates new inststance of Gtk_Key_Button_Record and sends
   -- it to Initialise.  Delivers initialized instance.
   procedure Gtk_New (New_Key_Button :    out Gtk_Key_Button;
                      Label          : in     UTF8_String := "";
                      Image          : in     Gtk.Image.Gtk_Image := null;
                      Row            : in     Positive := 1;
                      Column         : in     Positive := 1) is
   begin  -- Gtk_New
      New_Key_Button := new Gtk_Key_Button_Record;
      Initialize (New_Key_Button => New_Key_Button,
                  Label          => Label,
                  Image          => Image,
                  Row            => Row,
                  Column         => Column);
   end Gtk_New;

   -- Initializes Gtk_Key_Button_Record setting internal parameters
   -- 'Row' and 'Column'.  If Image /= null, sets Icon from Image.
   --  Sets Label if sent one.  Sets Name of button according to 'Row'
   --  and 'Column'.  Convention is that, if 'Row' = 1, and 'Column' = 1
   -- then Name will be "1_1".  Stores button's position in keypad.
   procedure Initialize
     (New_Key_Button   : in out Gtk_Key_Button;
      Label            : in     UTF8_String := "";
      Image            : in     Gtk.Image.Gtk_Image := null;
      Row              : in     Positive := 1;
      Column           : in     Positive := 1)is

      -- 'use' clause makes use of operator '/=' legal.
      use Gtk.Image;

      -- 'Row_Char' and 'Col_Char' are used in setting button Name.
      Row_Char : Character := Character'Val (Row + 48);
      Col_Char : Character := Character'Val (Column + 48);

   begin  -- Initialize

      -- Get a new button.
      Gtk.Button.Initialize (Button => New_Key_Button,
                             Label  => "");

      -- Give the button a name.
      Set_Name (Widget => New_Key_Button,
                Name   => Row_Char & '_' & Col_Char);

      -- Save the button's position in the keypad.
      New_Key_Button.Row := Row;
      New_Key_Button.Column := Column;

      -- Get a new box in button.
      Gtk.Box.Gtk_New_HBox (Box         => New_Key_Button.Box,
                            Homogeneous => false,
                            Spacing     => 0);

      -- Set Icon if Image /= null.
      if Image /= null then
         -- Put image in box.
         Gtk.Box.Pack_Start (In_Box  => New_Key_Button.Box,
                             Child   => Image,
                             Expand  => TRUE,
                             Fill    => TRUE,
                             Padding => 4);
      end if;  -- Image /= null

      -- Set Label if sent one.
      if Label /= "" then
         -- Create the Label for button.
         Gtk.Label.Gtk_New (Label => New_Key_Button.Label,
                            Str   => Label);

         -- Put label in box.
         Gtk.Box.Pack_Start (In_Box  => New_Key_Button.Box,
                             Child   => New_Key_Button.Label,
                             Expand  => TRUE,
                             Fill    => TRUE,
                             Padding => 4);
      end if;  -- Label /= ""

      -- Put the box in the button.
      Add (Container => New_Key_Button,
           Widget    => New_Key_Button.Box);
   end Initialize;

   -- Fetches the value of Button.Row.
   function Get_Button_Row (Button : access Gtk_Key_Button_Record'Class)
                            return Positive is
   begin  -- Get_Button_Row
      return Button.Row;
   end Get_Button_Row;

   -- Fetches the value of Button.Column.
   function Get_Button_Column (Button : access Gtk_Key_Button_Record'Class)
                               return Positive is
   begin  -- Get_Button_Column
      return Button.Column;
   end Get_Button_Column;

end Gtk.Key_Button;
```

9.1.3 Key_Button_Test

```
with Gtk.Main;
with Gtk.Window;
with Gtk.Box;
with Gtk.Enums;
with Gtk.Image;
with Gtk.Key_Button;
procedure Key_Button_Test is
   Key_Window  : Gtk.Window.Gtk_Window;
   Window_HBox : Gtk.Box.Gtk_HBox;
   Image       : Gtk.Image.Gtk_Image;
   Key_Button  : Gtk.Key_Button.Gtk_Key_Button;
begin  -- Key_Button_Test
   Gtk.Main.Set_Locale;
   Gtk.Main.Init;

   Gtk.Window.Gtk_New (Key_Window, Gtk.Enums.Window_Toplevel);
   Gtk.Window.Set_Title (Window => Key_Window,
                         Title  => "Key_Button Test");
   Gtk.Image.Gtk_New (Image    => Image,
                      Filename => "./img/Up.gif");
   Gtk.Key_Button.Gtk_New (New_Key_Button => Key_Button,
                           Label          => "Go Up",
                           Image          => Image,
                           Row            => 2,
                           Column         => 4);
   Gtk.Box.Gtk_New_HBox (Window_HBox, TRUE, 2);
   Gtk.Box.Pack_Start (Window_HBox, Key_Button);
   Gtk.Window.Add (Key_Window, Window_HBox);
   Gtk.Window.Show_All (Widget => Key_Window);

   Gtk.Main.Main;
end Key_Button_Test;
```

9.2 Keypad

9.2.1 Walker_Keypad Specification

```ada
with GLib;
with Gtk.Table;
with Gtk.Box;
with Gtk.Label;
with Gtk.Handlers;
with Gtk.Key_Button;

pragma Elaborate_All (Gtk.Handlers);

package Walker_Keypad_Pkg is

   KEYPAD_ROWS_COUNT : constant := 2;
   -- The number of rows on a keypad.

   KEYPAD_COLUMNS_COUNT : constant := 2;
   -- The number of columns on a keypad.

   type Keys_Type is array
     (1 .. KEYPAD_ROWS_COUNT, 1 .. KEYPAD_COLUMNS_COUNT)
   of Gtk.Key_Button.Gtk_Key_Button;
   --Intermediate, addressable storage of keys for the keypad.

   type Walker_Keypad_Record is new Gtk.Table.Gtk_Table_Record with private;
   -- The keypad which is a set of keys displayed in a Table.

   type Walker_Keypad is access all Walker_Keypad_Record'Class;
   -- Access to the keypad and its children.

```

```
  procedure Walker_New
     (Walker :    out Walker_Keypad);
   -- Creates and initializes the keypad.

   procedure Initialize
     (Walker : access Walker_Keypad_Record'Class);
   -- Initializes the keypad.

   package Key_Button_Handler_Pkg is
     new Gtk.Handlers.User_Callback (Gtk.Key_Button.Gtk_Key_Button_Record,
                                     Walker_Keypad);
   -- This package can be accessed externally and used to connect
   -- callback procedures to specific key_buttons in a keypad.

   function Get_Walker_Keypad_Button
     (Keypad       : Walker_Keypad;
      Row, Column  : Positive) return Gtk.Key_Button.Gtk_Key_Button;
   -- Returns an individual button from Pad.
private
   type Walker_Keypad_Record is new Gtk.Table.Gtk_Table_Record with
      record
         Pad          : Keys_Type := ((others => null), (others => null));
      end record;
end Walker_Keypad_Pkg;
```

9.2.2 Walker_Keypad Body

```ada
with Gtk.Box;
with Gtk.Image;
with Gtk.Enums;
with GLib;

with Ada.Strings.Unbounded;

package body Walker_Keypad_Pkg is

   -- Type to keep track of the names of the image files.
   type Button_Arrow_Data_Type is
     array (1 .. KEYPAD_ROWS_COUNT, 1 .. KEYPAD_COLUMNS_COUNT)
   of Ada.Strings.Unbounded.Unbounded_String;

   -- Store the names and addresses of the image files.
   Button_Arrows : constant Button_Arrow_Data_Type :=
     ((Ada.Strings.Unbounded.To_Unbounded_String (Source => ".\img\Right.gif"),
       Ada.Strings.Unbounded.To_Unbounded_String (Source => ".\img\Down.gif")),
      (Ada.Strings.Unbounded.To_Unbounded_String (Source => ".\img\Up.gif"),
       Ada.Strings.Unbounded.To_Unbounded_String (Source => ".\img\Left.gif")));

   -- Store the text for the button labels.
   Button_Label_Text : constant Button_Arrow_Data_Type :=
     ((Ada.Strings.Unbounded.To_Unbounded_String (Source => "Go Right"),
       Ada.Strings.Unbounded.To_Unbounded_String (Source => "Go Down")),
      (Ada.Strings.Unbounded.To_Unbounded_String (Source => "Go Up"),
       Ada.Strings.Unbounded.To_Unbounded_String (Source => "Go Left")));

```

```
-- Creates and initializes the keypad.
procedure Walker_New
  (Walker :    out Walker_Keypad) is
begin  -- Walker_New
   Walker := new Walker_Keypad_Record;
   Initialize (Walker  => Walker);
end Walker_New;

-- Initializes the keypad.
procedure Initialize (Walker : access Walker_Keypad_Record'Class) is

   -- 'use' clause lets the operator 'or' be used for
   -- Gtk.Enums in call to Attach.
   use Gtk.Enums;


   -- Image to place in button.
   Image : Gtk.Image.Gtk_Image;

begin  -- Initialize

   -- Initialize a Table to hold the Buttons.
   -- No need to coerce the constants into GLib.Gint
   -- because they are just Named Numbers.
   Gtk.Table.Initialize (Widget      => Walker,
                         Rows        => KEYPAD_ROWS_COUNT,
                         Columns     => KEYPAD_COLUMNS_COUNT,
                         Homogeneous => TRUE);

```

```
      Rows_Loop :
      for Row in 1 .. KEYPAD_ROWS_COUNT loop
         Columns_Loop :
         for Column in 1 .. KEYPAD_COLUMNS_COUNT loop

            -- Create the new arrow image.
            -- Button_Arrows gives the appropriate address for the image.
            Gtk.Image.Gtk_New
              (Image    => Image,
               Filename => Ada.Strings.Unbounded.To_String
                 (Button_Arrows (Row, Column)));

            -- Create a new button in the Pad (the array).
            -- It will already have a name, its coordinates in the
            -- keypad, a label and an icon by initialisation.
            -- Button_Label_Text gives the appropriate text for the label.
            Gtk.Key_Button.Gtk_New
              (New_Key_Button => Walker.Pad (Row, Column),
               Label          => Ada.Strings.Unbounded
               .To_String (Button_Label_Text (Row, Column)),
               Image          => Image,
               Row            => Row,
               Column         => Column);

            -- Attach the button  to the Table.
            -- Procedure inherited from Gtk.Table.
            Attach (Table         => Walker,
                    Child         => Walker.Pad (Row, Column),
                    Left_Attach   => GLib.GUint (Column - 1),
                    Right_Attach  => GLib.GUint (Column),
                    Top_Attach    => GLib.GUint (Row - 1),
                    Bottom_Attach => GLib.GUint (Row),
                    Xoptions      => Expand or Fill,
                    Yoptions      => Expand or Fill,
                    Xpadding      => 20,
                    Ypadding      => 20);

         end loop Columns_Loop;
      end loop Rows_Loop;
   end Initialize;

   -- Returns an individual button from Pad.
   function Get_Walker_Keypad_Button
     (Keypad      : Walker_Keypad;
      Row, Column : Positive) return Gtk.Key_Button.Gtk_Key_Button is
   begin  -- Get_Walker_Keypad_Button
      return Keypad.Pad (Row, Column);
   end Get_Walker_Keypad_Button;
end Walker_Keypad_Pkg;
```

9.2.3 Walker_Keypad_Test

```ada
with Gtk.Window;
with Gtk.Enums;
with Gtk.Box;
with Gtk.Main;
with Walker_Keypad_Pkg;
procedure Walker_Keypad_Test is
   Test_Window : Gtk.Window.Gtk_Window;
   Keypad_Box   : Gtk.Box.Gtk_HBox;
   Keypad      : Walker_Keypad_Pkg.Walker_Keypad;
begin  -- Walker_Keypad_Test
   Gtk.Main.Set_Locale;
   Gtk.Main.Init;

   Gtk.Window.Gtk_New (Window   => Test_Window,
                       The_Type => Gtk.Enums.Window_Toplevel);
   Gtk.Box.Gtk_New_HBox (Box         => Keypad_Box,
                         Homogeneous => TRUE,
                         Spacing     => 0);
   Walker_Keypad_Pkg.Walker_New (Walker => Keypad);
   Gtk.Box.Pack_Start (In_Box  => Keypad_Box,
                       Child   => Keypad,
                       Expand  => TRUE,
                       Fill    => TRUE,
                       Padding => 2);
   Gtk.Window.Add (Container => Test_Window,
                   Widget    => Keypad_Box);

   Gtk.Window.Show_All (Widget => Test_Window);

   Gtk.Main.Main;
end Walker_Keypad_Test;
```

9.3 Walker

9.3.1 Walker_Pkg Specification

```
with Gtk.Window;
with Gtk.Box;
with Walker_Keypad_Pkg;
package Walker_Pkg is

   type Walker_Record is
     new Gtk.Window.Gtk_Window_Record with
      record
         Keypad_Box : Gtk.Box.Gtk_Hbox := null;
         Keypad     : Walker_Keypad_Pkg.Walker_Keypad := null;
      end record;
   -- Define a window which will have a Walker Keypad in it.

   type Walker_Access is access all Walker_Record'Class;
   -- Define access to the window with the keypad.

procedure Gtk_New (Widget :    out Walker_Access);
   -- Create and initialize a new Walker Window.

   procedure Initialize (Widget : access Walker_Record'Class);
   -- Initialize a Walker Window.
end Walker_Pkg;
```

9.3.2 Walker_Pkg Body

```
with Gtk.Enums;
with Gtk.Widget;

with Walker_Pkg.Callbacks;
with Walker_Pkg_Callback_Pkg;
with Walker_Keypad_Pkg.Local_Callbacks;

package body Walker_Pkg is

   procedure Gtk_New (Widget :    out Walker_Access) is
   begin  -- Gtk_New
      Widget := new Walker_Record;
      Initialize (Widget => Widget);
   end Gtk_New;

   procedure Initialize (Widget : access Walker_Record'Class) is
      Icon : Boolean;
   begin  -- Initialize

      -- Create the window.
      Gtk.Window.Initialize (Window   => Widget,
                             The_Type => Gtk.Enums.Window_Toplevel);

      -- Give the window a Title.
      -- Procedure inherited from Gtk.Window;
      Set_Title (Window => Widget,
                 Title  => "Walker");

      -- Give the window an Icon.
      -- Function inherited from Gtk.Window;
      Icon := Set_Icon_From_File (Window   => Widget,
                                  Filename => "./img/Walker.gif");

      -- Connect the window destructor.
      Walker_Pkg_Callback_Pkg.Window_Callback_Pkg.Connect
        (Widget => Widget,
         Name   => "destroy",
         Marsh   => Walker_Pkg_Callback_Pkg.Window_Callback_Pkg
         .To_Marshaller
           (Walker_Pkg.Callbacks.On_X_Button_Destroy'Access));

      -- Put the window at the start position.
      Gtk.Widget.Set_UPosition
        (Widget    => Get_Toplevel (Widget => Widget),
         X         => 100,
         Y         => 100);

      -- Create keypad box
      Gtk.Box.Gtk_New_Hbox (Box          => Widget.Keypad_Box,
                            Homogeneous => TRUE,
                            Spacing     => 0);

      -- Put box in window.
      Add (Container => Widget,
           Widget    => Widget.Keypad_Box);

      -- Create the keypad.
      Walker_Keypad_Pkg.Walker_New (Walker => Widget.Keypad);

      -- Connect the callbacks to the keys in the keypad.
      Row_Loop :
      for Row in
        1 .. Walker_Keypad_Pkg.KEYPAD_ROWS_COUNT
      loop
         Column_Loop :
         for Column in
           1 .. Walker_Keypad_Pkg.KEYPAD_COLUMNS_COUNT
         loop
            Walker_Keypad_Pkg.Key_Button_Handler_Pkg.Connect
              (Widget    => Walker_Keypad_Pkg.Get_Walker_Keypad_Button
                 (Keypad => Widget.Keypad,
                  Row    => Row,
                  Column => Column),
               Name      => "clicked",
               Marsh     =>
                 Walker_Keypad_Pkg.Key_Button_Handler_Pkg
               .To_Marshaller
                  (Walker_Keypad_Pkg.Local_Callbacks
                   .On_Walker_Keypad_Button_Click'Access),
                User_Data => Widget.Keypad);
         end loop Column_Loop;
      end loop Row_Loop;

      -- Install keypad in box in window.
      Gtk.Box.Pack_Start (In_Box   => Widget.Keypad_Box,
                          Child    => Widget.Keypad,
                          Expand   => TRUE,
                          Fill     => TRUE,
                          Padding  => 0);
   end Initialize;
end Walker_Pkg;
```

9.3.3 Walker_Pkg_Callback_Pkg Specification

```
with Gtk.Handlers;
with Gtk.Window;

pragma Elaborate_All (Gtk.Handlers);

package Walker_Pkg_Callback_Pkg is
      package Window_Callback_Pkg is new
     Gtk.Handlers.Callback (Gtk.Window.Gtk_Window_Record);
   -- This instantiation creates a new package to handle
   -- Gtk_Window_Record.
end Walker_Pkg_Callback_Pkg;
```

9.3.4 Walker_Pkg.Callbacks Specification

```
with Gtk.Window;
package Walker_Pkg.Callbacks is
   procedure On_X_Button_Destroy
     (Object : access Gtk.Window.Gtk_Window_Record'Class);
   -- On_X_Button_Destroy is a callback to destroy the main window and end
   -- the program run when the [X] button at top right of the window is clicked.
end Walker_Pkg.Callbacks;
```

9.3.5 Walker_Pkg.Callbacks Body

```
with Gtk.Main;
package body Walker_Pkg.Callbacks is
   -- On_X_Button_Destroy is a callback to destroy the main window and end
   -- the program run when the [X] button at top right of the window is clicked.
   procedure On_X_Button_Destroy
     (Object : access Gtk.Window.Gtk_Window_Record'Class) is
   begin  -- On_X_Button_Destroy
      Gtk.Main.Main_Quit;
   end On_X_Button_Destroy;
end Walker_Pkg.Callbacks;
```

9.3.6 Walker_Keypad_Pkg.Local_Callbacks Specification

```
with Gtk.Window;
with Gtk.Key_Button;
package Walker_Keypad_Pkg.Local_Callbacks is
   procedure On_Walker_Keypad_Button_Click
     (Button : access Gtk.Key_Button.Gtk_Key_Button_Record'Class;
      Keypad : Walker_Keypad);
   -- On_Phone_Keypad_Button_Click reads the Row and Column of the
   -- Keypad_Button and takes appropriate action on that basis.
end Walker_Keypad_Pkg.Local_Callbacks;
```

9.3.7 Walker_Keypad_Pkg.Local_Callbacks Body

```
with Gtk.Main;
with Gtk.Widget;
with GLib;
with Gtk.Key_Button;
with Walker_Pkg_Operations;
with Walker_Pkg;
package body Walker_Keypad_Pkg.Local_Callbacks is
   -- On_Phone_Keypad_Button_Click reads the Row and Column of the
   -- Keypad_Button and takes appropriate action on that basis.
   procedure On_Walker_Keypad_Button_Click
     (Button : access Gtk.Key_Button.Gtk_Key_Button_Record'Class;
      Keypad : Walker_Keypad) is

      -- Find the X coordinate of the top-left-hand corner of the
      -- toplevel widget (the window).
      X_Pos : GLib.Gint := Gtk.Widget.Get_Allocation_X
        (Widget => Get_Toplevel (Widget => Keypad));

      -- Find the Y coordinate of the top-left-hand corner of the
      -- toplevel widget (the window).
      Y_Pos : GLib.Gint := Gtk.Widget.Get_Allocation_Y
        (Widget => Get_Toplevel (Widget => Keypad));

      -- Identify the toplevel widget (the window).
      Window : Gtk.Widget.Gtk_Widget :=
        Gtk.Key_Button.Get_Toplevel (Widget => Button);

   begin  -- On_Walker_Keypad_Button_Click

      -- Call the appropriate operation via the array
      -- of Operation_Access_Type.
      Walker_Pkg_Operations.Operation_Access_Array
        (Gtk.Key_Button.Get_Button_Row (Button),    -- Row coordinate in array
         Gtk.Key_Button.Get_Button_Column (Button) -- Column coordinate in array
        )
        (Window, X_Pos, Y_Pos);

   end On_Walker_Keypad_Button_Click;
end Walker_Keypad_Pkg.Local_Callbacks;
```

9.3.8 Walker_Keypad_Operations Specification

```
with GLib;
with Gtk.Widget;
with Walker_Pkg;
with Walker_Keypad_Pkg;
package Walker_Pkg_Operations is

   type Operation_Access_Type is access
     procedure (Window : Gtk.Widget.Gtk_Widget;
                X, Y   : in     GLib.Gint);
   -- Define an access to the procedures required in response to
   -- the clicking of a button on the keypad.

   type Operation_Access_Array_Type is
     array (1 .. Walker_Keypad_Pkg.KEYPAD_ROWS_COUNT,
            1 .. Walker_Keypad_Pkg.KEYPAD_COLUMNS_COUNT)
   of Operation_Access_Type;
   -- Define an array of procedure access to match the keypad.


   procedure Button_1_1 (Window : Gtk.Widget.Gtk_Widget;
                         X, Y   : in     GLib.Gint);
   -- Define operation in response to click on 'Go Right' button.

   procedure Button_1_2 (Window : Gtk.Widget.Gtk_Widget;
                         X, Y   : in     GLib.Gint);
   -- Define operation in response to click on 'Go Down' button.

   procedure Button_2_1 (Window : Gtk.Widget.Gtk_Widget;
                         X, Y   : in     GLib.Gint);
   -- Define operation in response to click on 'Go Up' button.

   procedure Button_2_2 (Window : Gtk.Widget.Gtk_Widget;
                         X, Y   : in     GLib.Gint);
   -- Define operation in response to click on 'Go Left' button.

   Operation_Access_Array : Operation_Access_Array_Type :=
     ((Button_1_1'Access, Button_1_2'Access),
      (Button_2_1'Access, Button_2_2'Access));
   -- Assign appropriate procedure access to each cell of the array.
end Walker_Pkg_Operations;
```

9.3.9 Walker_Keypad_Operations Body

```
with Gtk.Widget;
package body Walker_Pkg_Operations is

   -- Operation in response to 'Go Right' button.
   procedure Button_1_1 (Window : Gtk.Widget.Gtk_Widget;
                         X, Y   : in     GLib.Gint) is
      -- Set for right horizontal position.
      X_Pos : GLib.Gint := 400;

      -- set for existing vertical position.
      Y_Pos : GLib.Gint := Y;
   begin  -- Button_1_1
      -- Move the window.
      Gtk.Widget.Set_UPosition
        (Widget   => Window,
         X        => GLib.Gint (X_Pos),
         Y        => GLib.Gint (Y_Pos));
   end Button_1_1;


   -- Operation in response to 'Go Down' button.
   procedure Button_1_2 (Window : Gtk.Widget.Gtk_Widget;
                         X, Y   : in     GLib.Gint) is
      -- Set for existing horizontal position.
      X_Pos : GLib.Gint := X;

      -- Set for lower vertical position.
      Y_Pos : GLib.Gint := 400;
```

```
begin   -- Button_1_2
   -- Move the window.
   Gtk.Widget.Set_UPosition
     (Widget    => Window,
      X         => GLib.Gint (X_Pos),
      Y         => GLib.Gint (Y_Pos));
end Button_1_2;


-- Operation in response to 'Go Up' button.
procedure Button_2_1 (Window : Gtk.Widget.Gtk_Widget;
                      X, Y   : in     GLib.Gint) is
   -- Set for existing horizontal position.
   X_Pos : GLib.Gint := X;

   -- Set for upper vertical position.
   Y_Pos : GLib.Gint := 100;
begin   -- Button_2_1
   -- Move the window.
   Gtk.Widget.Set_UPosition
     (Widget    => Window,
      X         => GLib.Gint (X_Pos),
      Y         => GLib.Gint (Y_Pos));
end Button_2_1;


-- Operation in response to 'Go Left' button.
procedure Button_2_2 (Window : Gtk.Widget.Gtk_Widget;
                      X, Y   : in     GLib.Gint) is
   -- Set for left horizontal position.
   X_Pos : GLib.Gint := 100;

```

```
      -- Set for existing vertical position.
      Y_Pos : GLib.Gint := Y;
   begin  -- Button_2_2
      -- Move the window.
      Gtk.Widget.Set_UPosition
        (Widget   => Window,
         X        => GLib.Gint (X_Pos),
         Y        => GLib.Gint (Y_Pos));
   end Button_2_2;

end Walker_Pkg_Operations;
```

9.3.10 Walker — Mainline

```
with Gtk.Main;
with Walker_Pkg;
procedure Walker is
   Walker : Walker_Pkg.Walker_Access;
begin  -- Walker
   Gtk.Main.Set_Locale;
   Gtk.Main.Init;

   Walker_Pkg.Gtk_New  (Widget => Walker);
   Walker_Pkg.Show_All  (Widget => Walker);

   Gtk.Main.Main;
end Walker;
```

10 Access Types as Parameters

The question to CompLangAda was:

```
Given:

  type My_Type is  ...;
and
  type My_Access_Type is access all My_Type'Class;

what is the practical difference between:

  function My_Function (Thing : access My_Type'Class)
     return Positive;
and
  function My_Function (Thing : My Access_Type)
     return Positive;
```

Adam Beneschan replied:

The accessibility level rules say that a value of type My_Access_Type cannot point to an object that might go away before My_Access_Type does. (Unless you use Unchecked_Access to create the access value.) This is to prevent dangling references. The rules apply to parameters also, so if you call the second My_Function, you can't give it the 'Access of an object that is deeper than My_Access_Type.

If My_Access_Type is declared at library level (i.e. in a library package, not inside a procedure or function), then you can't say

```
procedure Some_Procedure is
   X : aliased My_Type;   --or some type derived from My_Type
begin
   My_Function (X'access);   -- i.e. the second My_Function
```

But this restriction doesn't apply to the first My_Function. Since that My_Function has an access parameter, you can pass the 'Access of any object of the right type, no matter how deeply nested inside procedures the object is.

Dmitry A. Kazakov added:

> Why then does GtkAda consistently use the first form,

```
   function My_Function (Thing : access Gtk_XXX_Record'Class)
      return Positive;
```

I think it should better be

```
  function My_Function (Thing : Gtk_XXX) return Positive;
```

since GtkAda has Gtk_XXX declared as access to every Gtk_XXX_Record'Class.

… this is likely a design bug. Gtk_Object_Record should have been limited controlled with Initialize inherited from Ada.Finalization.Limited_Controlled.

Dmitry Kazakov added:

```
   function My_Function (Thing : access Gtk_XXX_Record'Class)
      return Positive;
```

I think it should better be

```
  function My_Function (Thing : Gtk_XXX) return Positive;
```

since GtkAda has Gtk_XXX declared as access to every Gtk_XXX_Record'Class.

```
> It actually uses the form (this for Gtk.Button.Gtk_Button):
> procedure Initialize
>    (Button : access Gtk_Button_Record'Class;
>     Label  : UTF8_String);
```

Well, this is likely a design bug. Gtk_Object_Record should have been limited controlled with Initialize inherited from Ada.Finalization.Limited_Controlled.

Adam added:

*Well, for a *function*, using an access parameter can help get around the rule that you can't have an IN OUT parameter to a function (a rule that I think is going away in the next revision of the language). By making it an anonymous access type, rather than a named access type, that lets you pass local variables that the function can modify.*

For a procedure, there's less reason to do so. I don't know anything about GTK, so I don't know why this wouldn't have worked, if all Initialize is doing is to set up some fields in Button:

```
procedure Initialize
                 (Button : in out Gtk_Button_Record'Class;
                  Label  : UTF8_String);
```

If, on the other hand, Initialize needs Button as an access so that it can store it in a data structure, then it probably would have been best to make it a named access type, to ensure that dangling references aren't stored in that data structure.

*If Initialize is an imported routine from a C library---well, in that case, I guess you do whatever works. But I'd still think that using "in out" would have the same effect---the address of the record is passed. (In fact, for an anonymous access parameter, the code normally CANNOT pass *just* an address; there has to be additional information in case the procedure needs to do an accessibility level check. But for a procedure imported from C, the parameter passing mechanisms may be different.)*

Randy is right, and I thought about saying something along those lines in my first response but decided (maybe unwisely) not to complicate things. You don't need to pass an anonymous access parameter if all you're going to do is modify the accessed object---IN OUT will do. And if you're going to store the access parameter in a global structure, you'll get a runtime error if you try to store a dangling reference, so it's best to use a global access type so that accessibility level errors are caught at compile time. There may be some esoteric situations, other than dispatching, where passing an anonymous access parameter may be useful: perhaps the procedure can determine at runtime whether to store an access value or make a copy of the object depending on the accessibility level, or perhaps the procedure wants the ability to accept NULL to indicate that there is no object to modify, or perhaps a procedure takes two access parameters and sets up the two accessed objects to point to each other. But those all seem very unusual, and may be due to an inferior design anyway.

Stephen Leake, perhaps, explained it best:

Most GTK subprograms do need access values; the same is true of any system that deals with derived types at multiple levels. The only library-level object that can hold any type in the hierarchy is a class-wide pointer. Lists of objects must hold pointers, etc.

So one reason to use 'access' instead of 'in out' is simply to avoid the user having to type '.all' everywhere.

A reason to use 'access Record_Type' instead of 'in Access_Type' is to avoid explicit type conversions. Given:

```
package Widget is
    type Gtk_Widget_Record is ...;
    type Gtk_Widget is access all Gtk_Widget_Record'class;
    procedure Show (Widget : in Gtk_Widget);
end Widget;
package Window is
    type Gtk_Window_Record is new Gtk_Widget_Record with ...;
    type Gtk_Window is access all Gtk_Window_Record'class;
end Window;
    Window : Gtk_Window := ...;
```

then this does not work:

```
   Widget.Show (Window);
```

but this does:

```
   Widget.Show (Gtk_Widget (Window));
```

This is just annoying!

However, if Show is declared:

```
procedure Show (Widget : access constant Gtk_Widget_Record'class);
```

Then Show matches any access type in the class hierarchy;

```
    Widget.Show (Window);
```

works.

11 Reference List

Barnes, J. (2006). *Programming in Ada 2005*. Harlow, UK: Pearson Education Ltd.

Dale, N., Weems, C., & McCormick, J. (2000). *Programming and Problem Solving with Ada95 (second edition)*. Sudbury, MA (USA): Jones and Bartlett.

Software Productivity Consortium (1995). *Ada 95 Quality and Style: Guidelines for Professional Programmers*. Department of Defense Ada Joint Program Office.