Harlequin Dylan

Library Reference: Interoperability

Version 1.0 Beta Preview

harlequin

Copyright and Trademarks

Harlequin Dylan: Library Reference: Interoperability

Version 1.0 Beta Preview

July 1997

Part number: DYL-1.0.BP-RM3

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
U.K.

telephone +44 1223 873 800
fax +44 1223 873 873

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
U.S.A.

telephone +1 617 374 2400
fax +1 617 252 6505

Electronic Access:

http://www.harlequin.co.uk/
http://www.harlequin.com/

Contents

1 The C-FFI Library

1.1 Introduction

The C-FFI (*C foreign function interface*) library provides a means of interfacing a Dylan application with code written in the C language. The C-FFI library is available to applications as the module **`c-ffi`** in the library **`c-ffi`**.

The C-FFI library consists of macros, classes, and functions that you can use to write a complete description of the Dylan interface to a C library. Compiling this description generates a set of Dylan classes and functions through which Dylan code can manipulate the C library's data and call its functions. Interface descriptions can also allow C code to call into Dylan; compiling such a description generates entry points compatible with C's calling conventions.

1.2 Overview

This section is an overview of the C-FFI library, introducing the most commonly used constructs as the basis for examples.

The C-FFI library provides a set of macros that can be used to describe a C interface in a form that the Harlequin Dylan compiler can understand; we call these macros the *C interface definition language*.

The C interface definition language provides macros that correspond to each of C's type, function, variable, and constant defining forms. These macros define Dylan classes that designate and encapsulate instances of C types, Dylan methods through which to manipulate C variables and call out to C functions, and functions with C-compatible entry points through which to call in to Dylan from C.

In addition to the interface definition language, the C-FFI library provides run-time methods and functions for allocating, manipulating and destroying instances of C data structures. For example, using these facilities you can allocate C structs and arrays, and access and set their elements.

1.2.1 C types in Dylan

When you use the interface definition language to describe a C type to the Dylan compiler, the compiler generates a new Dylan class. This class is said to *designate* the C type, which means that it carries with it the essential properties of the C type such as its size and alignment.

You can use this *designator class* in subsequent interface definition forms to specify which elements involve the designated C type. A designator class also carries with it the information on how to interpret the untyped C data as a tagged Dylan object.

The C-FFI library contains predefined designator classes for C's fundamental types like **`int`** and **`double`**. The names of these predefined Dylan classes are formed from the C name of the fundamental type being designated. The designator class name for a particular C type formed using Dylan's standard class-naming convention; it is prefixed with "C-", hyphenated if it contains more than one word, and enclosed in angle brackets. For example, the C-FFI library provides the class **`<C-int>`** to designate the C type **`int`**; it designates **`double`** by the class **`<C-double>`**, and **`unsigned long`** by the class **`<C-unsigned-long>.`**

Note: Since Dylan variable names are compared without sensitivity to case, the capitalization of the "C" in the names above, and in other Dylan names appearing in this document, is not binding and can safely be ignored.

The C-FFI library also provides predefined classes designating pointers to C's fundamental numeric types. To do so, it adds a **`*`** to the fundamental C type designator. For example **`<C-double*>`** designates the C type **`double *`**.

The following is an example of defining and using designator classes. Suppose we have the following C struct:

```
typedef struct {
  unsigned short x_coord;
  unsigned short y_coord;
} Point;
```

We describe C structs to Dylan using the macro **`define C-struct`**:

```
define C-struct <Point>
  slot x-coord :: <C-unsigned-short>;
  slot y-coord :: <C-unsigned-short>;
end C-struct;
```

This form defines a new designator class **`<Point>`** for a structure type corresponding to the C type **`Point`**. We designate the types of the slots of **`<Point>`** using the Dylan classes designating the C types used in the definition of **`Point`**. In this case, both slots are of the C type **`unsigned short`** which is designated by the predefined class **`<C-unsigned-short>`**. The information about the C type **`unsigned short`** carried by this designator class allows the compiler to compute the size, alignment, and layout of the struct. The compiler records the struct's size and alignment and associates them with **`<Point>`**. The designator class **`<Point>`** can then be used in the definition of other types, functions, and variables. For example, we could describe

```
typedef struct {
  Point start;
  Point end;
} LineSegment;
```

like this:

```
define C-struct <LineSegment>
  slot start :: <Point>;
  slot end   :: <Point>;
end C-struct;
```

As well as acting as a static information carrier for use in other FFI definitions, a designator class can also be instantiable, in which case Dylan uses an instance of the designator class to represent an object of the C type it designates when that object is passed from the "C world" to the "Dylan world".

Note: Only classes that designate C pointer types can be instantiated in this way. Instances of C's fundamental numeric value types like **`int`**, **`char`**, and **`double`** are just converted to an equivalent Dylan object with the same value. The **`<Point>`** class is not an instantiable class in Dylan because there is nothing in Dylan that corresponds to a C struct. However, the C-FFI does provide a Dylan representation of a pointer to a C struct.

To illustrate, here is an example interaction involving a C struct containing some pointer-typed slots and some slots whose types are fundamental numeric types:

```
define C-struct <Example>
  slot count        :: <C-int>;
  slot statistic    :: <C-double>;
  slot data         :: <C-char*>;
  slot next         :: <Example*>;
  pointer-type-name :: <Example*>;
end C-struct;
```

This example defines the two designator types **`<Example>`** and **`<Example*>`**; the slots **`count`** and **`statistic`** have fundamental numeric types while **`data`** and **`next`** have pointer types. The getter and setter methods for the slots are defined for instances of **`<Example*>`**.

Suppose there is a function **`current-example`** that returns an initialized **`<Example*>`** struct. The following transactions illustrate what you get when you read the slots of the structure it returns:

```
? define variable example = current-example();
// Defined example

? example.count;
4

? instance?(example.count, <integer>);
#t

? example.statistic;
10.5

? instance?(example.statistic, <float>);
#t
```

The interactions above show that if we access structure slots that were defined as being of one of C's fundamental numeric types, we get a Dylan number of the equivalent value. The same thing happens if an imported C function returns a fundamental numeric type: a Dylan number with the same value appears in Dylan. Similarly, when setting slots in structs expecting numbers or passing objects out to C functions expecting numeric arguments, you should provide a Dylan number, and the C-FFI will convert it automatically to its C equivalent.

```
? example.data;
{<C-char> pointer #xff5e00}

? instance?(example.data, <C-char*>);
#t

? example.next;
{<Example> pointer #xff5f00}

? instance?(example.next, <Example*>);
#t
```

The interactions above show that accessing structure slots with a pointer type results in an instance of the Dylan class that designates that type. Again, the same thing happens if an imported C function returns a pointer type: an instance of the corresponding designator class is created. Similarly, when setting slots in structs expecting pointers or passing objects out to C functions expecting pointer arguments, you should provide an instance of the Dylan designator class for that pointer type, and the C-FFI will convert it automatically to the raw C pointer value.

Later sections describe all the macros available for defining C types and the functions available for manipulating them.

1.2.2 C functions in Dylan

When you use the interface definition language to describe a C function to the Dylan compiler, the compiler generates a new Dylan function. This *wrapper function* accepts Dylan arguments and returns Dylan results. It converts each of its arguments from a Dylan object to a corresponding C value before calling the C function it wraps. The C-FFI converts any results that the C function returns into Dylan objects before returning them to the caller.

In order for Dylan to be able to call into C correctly, C functions must be described to Dylan in the same detail a C header file would provide a calling

C program. Specifically, for every function we must provide the C name and the type of its arguments and results. As with struct definitions, these types are indicated by naming the designator classes corresponding to the C types involved in the C-FFI description of the C function.

The following is an example of defining and using wrapper functions. Suppose we have the following **`extern`** C function declaration:

```
extern double cos (double angle);
```

We describe C functions to Dylan using the C-FFI macro **`define C-function`**:

```
define C-function C-cos
  parameter angle :: <C-double>;
  result    cos   :: <C-double>;
  c-name: "cos"
end C-function;
```

The name appearing immediately after the **`define C-function`** is the name we want to give to the Dylan variable to which our wrapper function will be bound. We call it **`C-cos`**. We also give the actual C name of the function we want to wrap as the value of the keyword **`c-name:`**.

Once we have compiled the definition — and assuming the compiled version of the C library implementing **`cos`** has been linked in with the Dylan application — we can call the wrapper function just like any other Dylan function:

```
? C-cos(0.0);
1.0
```

As we noted above, when values are passed back and forth between Dylan and C, the C-FFI performs automatic conversions. In this case, the type of the parameter and the result are both fundamental numeric types which means that the C-FFI will accept and return Dylan floats, converting to and from raw C floats as necessary.

As well as making C functions available to Dylan code, the C-FFI allows us to make Dylan functions available to call from C code. We do this by defining a *C-callable* wrapper function. A C-callable wrapper is a Dylan function that a C program can call. The C-callable wrapper has a C calling convention. When a C program calls a C-callable wrapper, the C-FFI performs the necessary data conversions and then invokes a Dylan function.

You can pass C-callable wrappers into C code for use as callbacks. You can also give them names visible in C, so that C clients of Dylan code can call into Dylan directly by invoking a named function.

The argument and result conversions performed by C-callable wrappers are just like those done within Dylan wrapper functions. The macro that defines C-callable wrappers is called **define C-callable-wrapper** and we describe it in detail later. For now, consider the following simple example. Suppose we have a C **extern** function declaration **AddDouble**:

```
extern double AddDouble (double x, double y);
```

This function is intended to return the sum of two **double** values. Instead of implementing the function in C, we can implement it in Dylan using Dylan's generic function **+**. All we need to do is define a C-callable wrapper for **+**, as follows:

```
define C-callable-wrapper AddDoubleObject of \+
  parameter x :: <C-double>;
  parameter y :: <C-double>;
  c-name: "AddDouble";
end C-callable-wrapper;
```

We can now call **AddDouble** in C. Our wrapper will be invoked, the C arguments will be converted and passed to Dylan's + generic function, and then the result of the computation will be converted and passed back to C:

```
{
  extern double AddDouble (double x, double y);
  double result;

  result = AddDouble(1.0, 2.0);
}
```

The C-FFI binds the Dylan variable **AddDoubleObject** to a Dylan object representing the function pointer of the C-callable wrapper. This reference allows the C-callable wrapper to be passed to a C function expecting a callback argument.

1.2.3 C variables in Dylan

When you use the interface definition language to describe a C variable to the Dylan compiler, the compiler generates new Dylan getter and setter functions for reading and setting the variable’s value from Dylan. If the variable is constant, it defines a getter function only.

The getter function converts the C value to a Dylan value before returning it according to the variable’s declared type. Similarly, the setter function converts its argument, as Dylan value, into a C value before setting the C variable. These conversions happen according to the same rules that apply to other C-Dylan world transition points, such as argument passing or structure slot access.

In order for Dylan to be able to access a C variable correctly, we must describe the variable to Dylan in the same detail that a C header file would give to a C program that uses it. Specifically, we must provide the C name and the type of the variable. As with struct and function definitions, we indicate C types by naming the appropriate Dylan designator classes.

Here is an example of defining and using C variables. Suppose we have the following **`extern`** C variable declaration:

```
extern double mean;
```

We describe C variables to Dylan using the C-FFI macro **`define C-variable`**:

```
define C-variable C-mean :: <C-double>
  c-name: "mean";
end C-variable;
```

The name immediately after the **`define C-variable`** is the name of the Dylan variable to which the getter for our C variable will be bound. In this case it is **`C-mean`**.

We give the C name of the variable as the value of the keyword **`c-name:`**. Once we have compiled the definition — and assuming the compiled version of the C library defining **`mean`** has been linked in with the Dylan application — we can call the getter function just like any other Dylan function:

```
? C-mean();
1.5
```

By default, the C-FFI also defines a setter function for the variable. The setter name uses Dylan’s convention of appending ”-setter” to the getter name.

```
? C-mean() := 0.0;
0.0
```

```
? C-mean();
0.0
```

As described above, when values are passed back and forth between Dylan and C, the C-FFI performs automatic conversions. In this case, the type of the variable is a fundamental numeric type which means that the C-FFI accepts and returns Dylan floats, converting to and from raw C floats as necessary.

Note: We could achieve the same result by using the **`define C-address`** macro, which defines a constant that is a pointer to the storage allocated for the C variable.

1.3 Terminology

For the rest of this chapter, we adopt the following terminology, hopefully not too inconsistent with common C terminology:

Base type	Basic units of data storage (C's variously sized integers, characters, and floating point numbers) and aggregate records (structs and unions).
Derived type.	A type based on some other type (C's pointer, array, and function types).
Fundamental numeric type.	One of C's integer or floating point types. This does not include pointer types, structure types, or union types.

1.4 Basic options in C-FFI macros

The defining macros of the C-FFI share a consistent core set of options which are worth describing here:

- A *c-name* argument. Every defining form allows you to specify the corresponding C entity through the keyword **`c-name:`**. It is optional in some forms but required in others. You can define types that have no named opposite number in C, and the *c-name* option is always optional in type definitions. On the other hand, you must always name an imported C function or variable so that Dylan knows the correct name from the compiled C library to link with.

In general, any C entity you can declare in C using **`extern`** can only be found by the C-FFI if you pass a *c-name* argument to the corresponding C-FFI definition.

- A *pointer-type-name* argument. All the type-defining forms allow you to name the type for a pointer to the type being defined. This is normally specified throughout the **`pointer-type-name:`** keyword option.

1.5 Designator classes

As Section 1.2 explained, the C-FFI defines some Dylan classes to designate C types and to describe how they are passed to and from Dylan. These *designator classes* carry with them static information about the C type they designate.

The C-FFI library provides an initial set of designator classes corresponding to C's fundamental types, as well as macros for generating designator classes corresponding to C's pointer types and for extending the translation between C data and Dylan objects.

Designator classes that correspond to fundamental numeric types are not instantiable. When you pass a numeric value to Dylan from C, the C-FFI simply generates a Dylan number with the same value. Similarly, a Dylan number passed to C is converted to a C number of the appropriate type and value.

Each of the fundamental designator classes indicate a conversion to or from a unique Dylan class. The conversions that take place are described in detail in the documentation for each designator class.

The main reasons for this design are increased efficiency, simplified implementation, and added convenience when working with numeric values. The designator classes for the numeric types could have been made instantiable and placed beneath the appropriate number protocol classes in Dylan, but these extra representations in such a fundamental area could cause problems for Dylan compilers. In addition, to make these instantiable designator classes convenient to work with, the C-FFI would also have to define methods on the standard arithmetic and comparison operators. It is simpler to represent these fundamental types with existing Dylan objects.

However, the designator classes that correspond to pointer types *are* instantiable. When you pass a pointer from C to Dylan, the C-FFI constructs an instance of the appropriate designator class that contains the raw address. A wrapped pointer like this can be passed out to some C code that is expecting a compatible pointer — the C-FFI extracts the raw address before handing it to C code. The documentation for the abstract class **`<C-pointer>`** describes the compatibility rules for pointer types.

This feature of pointer designator classes allows Dylan code to be typed to a specific kind of pointer. For example, you can define methods that specialize on different kinds of pointer on the same generic function.

1.5.1 Designator type properties

To understand how designator classes work, it is useful to know about their properties. A few of these properties are accessible programmatically, but others are implicit and only really exist in the compiler. Some of the properties may be empty.

A *referenced type* is the designator type to which a pointer refers. A designator's *referenced-type* only has a value for subtypes of **`<C-statically-typed-pointer>`**. Programs can access the referenced type through the function **`referenced-type`**.

A designator class's *pointer-type* only has a value for each of those types that has a pointer designator type that refers to it. Most of the constructs that define a new designator type also define a pointer-type for that designator. Many of the macros that define designators accept a **`pointer-type-name:`** keyword to bind the *pointer-type* of the defined designator to a given variable. The pointer-type is not programmatically available because it may not have been defined. You can assure that there is a pointer-type for a particular designator by using the macro **`define c-pointer-type`**.

A designator class's *import type* and *export type* are instantiable Dylan types that describe the Dylan instantiation of a designator class when it is used in a position that *imports* values from C, or *exports* values to C.

Nearly all of the C-FFI's designators have import and export types that are equivalent. Some, such as **`<C-string>`**, have different import and export types because it is possible to pass a pointer to a Dylan object to C directly without creating a C pointer object, or copying the underlying data, but when importing a string from C it is not practical to copy the contents and create a Dylan string. By default, the import and export types for any subtype of **`<C-pointer>`** are the class itself. You can override this by defining a new subclass with the macro **`define C-mapped-subtype.`**

You can define a designator's *import-function* and *export-function* by using the macro **`define c-mapped-subtype`**. These functions are merely the procedural specifications for translating the C data to Dylan and back. The *import* and *export* functions are inherited when you define a subclass for a designator.

1.5.2 Designator class basics

<C-value> *Sealed abstract class*

The abstract superclass of all designator classes. It is a subclass of **`<object>`**. It has neither an *export-type* nor an *import-type,* so you cannot use it when designating a transition between C and Dylan.

<C-void> *Sealed abstract class*

The abstract superclass of all designator classes. It is a subclass of **`<C-value>`**. It has neither an *export-type* nor an *import-type*, so you cannot use it when designating a transition between C and Dylan.

This class is only useful in that it is the *referenced-type* for **`<C-void*>.`**

size-of *Function*

`size-of` ***designator-class*** **`=>`** ***size***

Takes a designator class and returns the size of the C type that the class designates.

The **`size-of`** function can be applied to any designator class. It corresponds to C's **`sizeof`** operator and returns an integer, *size*, in the same units as **`sizeof`** does on the target platform. It can be useful when allocating a C object whose declared size is not accurate and has to be adjusted manually.

alignment-of *Function*

`alignment-of` ***designator-class*** **`=>`** ***alignment***

Takes a designator class and returns the alignment of the C type that the class designates. The **`alignment-of`** function can be applied to any designator class. It returns the alignment as an integer, in the same units as **`size-of`** does.

1.5.3 Fundamental numeric type designator classes

This section describes the pre-defined designator classes for fundamental C numeric types. On page 10 we saw that none of these designator types are instantiable:. a number on one side of the interface is converted to a number on the other side with the same value.

There are some additional details to note about integer representations. Because Dylan's integer representations do not match C's exactly, for each of the C integer types there are three designator classes that can be used to translate Dylan representations to that C integer. The categories are *plain*, *unsafe*, and *raw* integers.

Plain integer designators — of which the class **`<C-unsigned-short>`** is an example — translate C integer values to instances of **`<integer>`**. If the integer being translated is too big for the destination, the C-FFI signals an error. There are two ways this can happen.

- On export, the C-FFI signals an error if the Dylan value has more significant bits than the C integer.

  This can happen if, for example, the designator is **`<C-unsigned-short>`**, and the Dylan value is negative, or if **`unsigned short`** on that platform is 16 bits wide, but the Dylan integer has more than 16 significant bits. The check will be omitted if the compiler can determine that no Dylan value outside the safe range can reach there. This can be done using a limited integer type.

- On import into Dylan, the C-FFI signals an error if it cannot represent the C value using a Dylan **`<integer>`**.

> This can happen with any C integer type that is more than 30 bits wide. The size of a Dylan **`<integer>`** depends on the particular platform, but it is guaranteed to be at least 30 bits in length.

The C-FFI never signals an error for the *unsafe* designator classes — of which the class **`<C-unsafe-unsigned-short>`** is an example — but if the destination is too small for the value, the most significant bits of the value are chopped off to fit into the destination. Because there is no checking, using the unsafe designator classes brings a very small performance improvement, but nonetheless you should not use them unless you are certain you will not lose any bits.

Raw designator classes — of which the class **`<C-raw-unsigned-int>`** is an example — represent the integer on the Dylan side as a **`<machine-word>`**. An instance of **`<machine-word>`** is guaranteed to have enough bits to represent any C **`long`** value, or any C **`void *`** value. Note that a **`<machine-word>`** value may still have more significant bits than some C integer types, and so the C-FFI may still signal an overflow error if the **`<machine-word>`** value, interpreted as indicated by the designator, has more significant bits than may be held in the indicated C type.

Table 1.1 shows all raw, plain, and unsafe integer designator types exported from the **`c-ffi`** module.

Table 1.1 The integer designator classes and their mappings.

Designator name	C type	Dylan type(s)
`<C-int>`	**`int`**	**`<integer>`**
`<C-raw-int>`	**`int`**	**`<machine-word>`**
`<C-unsafe-int>`	**`int`**	**`<integer>`**
`<C-raw-signed-int>`	**`signed int`**	**`<machine-word>`**
`<C-unsafe-signed int>`	**`signed int`**	**`<integer>`**
`<C-signed-int>`	**`signed int`**	**`<integer>`**
`<C-raw-unsigned-int>`	**`unsigned int`**	**`<machine-word>`**
`<C-unsafe-unsigned-int>`	**`unsigned int`**	**`<integer>`**

Table 1.1 The integer designator classes and their mappings.

Designator name	C type	Dylan type(s)
`<C-unsigned-int>`	**`unsigned int`**	**`<integer>`**
`<C-unsigned-long>`	**`unsigned long`**	**`<integer>`**
`<C-signed-long>`	**`signed long`**	**`<integer>`**
`<C-unsafe-unsigned-long>`	**`unsigned long`**	**`<integer>`**
`<C-unsafe-signed-long>`	**`signed long`**	**`<integer>`**
`<C-raw-unsigned-long>`	**`unsigned long`**	**`<machine-word>`**
`<C-raw-signed-long>`	**`signed long`**	**`<machine-word>`**
`<C-unsigned-short>`	**`unsigned short`**	**`<integer>`**
`<C-signed-short>`	**`signed short`**	**`<integer>`**
`<C-unsafe-unsigned-short>`	**`unsigned short`**	**`<integer>`**
`<C-unsafe-signed-short>`	**`signed short`**	**`<integer>`**
`<C-raw-unsigned-short>`	**`unsigned short`**	**`<machine-word>`**
`<C-raw-signed-short>`	**`signed short`**	**`<machine-word>`**
`<C-unsigned-char>`	**`unsigned char`**	**`<integer>`**
`<C-signed-char>`	**`signed char`**	**`<integer>`**
`<C-unsafe-unsigned-char>`	**`unsigned char`**	**`<integer>`**
`<C-unsafe-signed-char>`	**`signed char`**	**`<integer>`**
`<C-raw-unsigned-char>`	**`unsigned char`**	**`<machine-word>`**
`<C-raw-signed-char>`	**`signed char`**	**`<machine-word>`**
`<C-char>`	**`char`**	**`<integer>`**
`<C-unsafe-char>`	**`char`**	**`<integer>`**
`<C-raw-char>`	**`char`**	**`<machine-word>`**

For each of the fundamental integer designator types, **`<C-`***xxx***`>`**, there is also a type designating pointers to that type called **`<C-`***xxx***`*>`**. In addition, the C-FFI defines methods for **`pointer-value`** and **`pointer-value-setter`**, with appropriate translation behavior for each of the types designating pointers to the fundamental integer designator types.

<C-number> *Sealed abstract class*

The abstract superclass of all classes that designate a fundamental numeric C type. Superclass is **`<C-value>`**.

<C-float> *Sealed abstract class*

<C-double> *Sealed abstract class*

<C-long-double> *Sealed abstract class*

These three classes designate the C floating point types **`float`**, **`double`**, and **`long double`** respectively. The superclass of each is **`<C-number>`**.

The Dylan representation of any C floating point type is the corresponding Dylan floating point value of at least equivalent precision. By default, where C expects one of these types, only Dylan floats are accepted on the Dylan side.

1.5.4 Pointer designator classes and related functions

This section describes the pre-defined classes that designate C pointer types. Subclasses of the abstract classes documented here are instantiable, and C pointers are represented in Dylan by instances of these classes.

Note: Pointer designator classes are defined for all the designator classes in Table 1.1, but are not listed here. To form the name of the pointer designator class for a particular designator class, append a "*" to the part of the name enclosed in angle brackets. Thus for **`<C-int>`** the pointer designator class is **`<C-int*>`**.

<C-pointer> *Primary open abstract class*

The abstract superclass of all classes that designate a C pointer type. It is a subclass of **`<C-value>`**. Instances of concrete subclasses of **`<C-pointer>`**

encapsulate a raw C address. The make methods on subclasses of **`<C-pointer>`** accept the keyword argument **`address:`**, which must be a Dylan **`<integer>`** or **`<machine-word>`** representation of the C address.

pointer-address *Function*

`pointer-address` ***C-pointer*** **`=>`** ***address***

Recovers the address from an instance of **`<C-pointer>`** and returns it as a Dylan **`<machine-word>`**.

as *G.f. method*

`as` ***pointer-designator-class C-pointer*** **`=>`** ***new-C-pointer***

Converts a pointer from one pointer type to another. The new pointer will have the same address as the old pointer.

null-pointer *Function*

`null-pointer` ***pointer-designator-class*** **`=>`** ***null-pointer***

Returns a null pointer whose type is given by *pointer-designator-class*. Note that different calls to **`null-pointer`** may return the same object.

null-pointer? *Function*

`null-pointer?` ***C-pointer*** **`=>`** ***boolean***

Returns true if a pointer is null and false otherwise.

<C-untyped-pointer> *Open abstract class*

The abstract superclass of all classes encapsulating a C pointer type for an unspecific base type. It is a subclass of **`<C-pointer>`**.

<C-void*> *Open concrete class*

The class designating C's **`void*`** pointer type. It is a subclass of **`<C-pointer>`**. No **`pointer-value`** methods are defined on this class.

<C-statically-typed-pointer> *Open abstract class*

The abstract superclass of all classes designating a C pointer type for a non-**`void`** base. It is a subclass of **`<C-pointer>`**.

define C-pointer-type *Definition macro*

`define C-pointer-type` ***pointer-class-name*** **`=>`** ***designator-class-name***

Defines a constant bound to a pointer class designating pointers to *designator-class-name*. Note that the pointer type may already exist. If the pointer type does not already exist then this macro defines that class and also defines **`pointer-value`**, **`pointer-value-setter`**, and **`make`** methods on the defined type. The class defined will be open, abstract and instantiable. Objects returned by **`make(`***pointer-class-name***`)`** will be instances of a sealed concrete subclass of *pointer-class-name*.

referenced-type *Function*

`referenced-type pointer-designator-class => designator-class`

Returns the class designating the contents type of the C pointer type designated by *pointer-designator-class*. The same designator class is returned whenever **`referenced-type`** is called with the same argument.

pointer-value *Open generic function*

`pointer-value` ***C-typed-pointer*** **`#key`** ***index*** **`=>`** ***object***

pointer-value-setter *Open generic function*

`pointer-value-setter` ***new-value C-typed-pointer*** **`#key`** ***index*** **`=>`** ***new-value***

These functions dereference and set C pointer values from Dylan. It is an error if *C-typed-pointer* does not point to a valid address or is a null pointer.

The **`pointer-value`** function dereferences *c-typed-pointer* using its encapsulated raw C address, and returns a Dylan object representing the object at that address. If you supply *index*, the pointer is treated as a pointer to an array, and the function returns the appropriate element indexed by the correct unit size.

The **`pointer-value-setter`** function allows you to set pointer values. The options are the same as those for **`pointer-value`**.

These two functions are part of a protocol for extending the C type conversions. You can define new methods for **`pointer-value`** and **`pointer-value`** for types defined by **`define C-subtype`** that are subtypes of **`<C-pointer>`**.

These functions perform the primitive Dylan-to-C and C-to-Dylan conversions as documented with the designator class of the pointer's contents type (see Table 1.1). The C-FFI signals an error if it cannot convert the object you attempt to store in the pointer to a compatible type.

pointer-value-address *Open generic function*

`pointer-value-address` ***C-typed-pointer*** **`#key`** ***index*** **`=>`** ***object***

Returns a pointer of the same type as *C-typed-pointer* that points to the *index*th object offset into *C-typed-pointer*. The following expression is guaranteed to be true:

```
pointer-value(C-typed-pointer, index: i)
  = pointer-value (pointer-value-address(C-typed-pointer, index: i))
```

element *G.f. method*

`element` ***C-statically-typed-pointer index*** **`=>`** ***object***

Synonymous with a call to **`pointer-value`** that includes the optional index. Thus it does the same thing as:

```
pointer-value(C-statically-typed-pointer, index: index)
```

element-setter *G.f. method*

`element-setter` ***new C-statically-typed-pointer index*** **`=>`** ***object***

Synonymous with a call to **`pointer-value-setter`** that includes the optional index. Thus it does the same thing as:

`pointer-value-setter(`*new*`,` *C-statically-typed-pointer*`, index:` *index*`)`

`-` *G.f method*

`-` ***C-pointer-1 C-pointer-2*** **`=>`** ***machine-word***

`=` *G.f method*

`=` ***C-pointer-1 C-pointer-2*** **`=>`** ***boolean***

`<` *G.f method*

`<` ***C-pointer-1 C-pointer-2*** **`=>`** ***boolean***

These methods allow pointer comparison operations to be performed on instances of **`<C-pointer>`**.

The result of **`-`** is given in terms of element units rather than bytes.

The method for = is equivalent to:

`(pointer-address(`*C-pointer-1*`) = pointer-address(`*C-pointer-2*`))`

Note that operations corresponding to C pointer arithmetic are not defined on **`<C-pointer>`**. If pointer arithmetic operations are required, use **`pointer-value`** with an **`index:`** argument.

The following functions comprise the conceptual foundation on which the pointer accessing protocol is based. In the signatures of these functions, *byte-index* is in terms of address units (typically bytes) and *scaled-index* is scaled by the size of the units involved. In the setters, *new* is the new value to which the value in the pointed-at location will be set. These functions can be used to deference any general instance of **`<C-pointer>`**.

C-unsigned-char-at *Function*

`C-unsigned-char-at` ***C-pointer*** **`#key`** ***byte-index scaled-index*** **`=>`** ***machine-word***

C-unsigned-char-at-setter *Function*

`C-unsigned-char-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
=> ***machine-word***

C-signed-char-at *Function*

`C-signed-char-at` ***C-pointer*** `#key` ***byte-index scaled-index*** **=>** ***machine-word***

C-signed-char-at-setter *Function*

`C-signed-char-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
=> ***machine-word***

C-char-at *Function*

`C-char-at` ***C-pointer*** `#key` ***byte-index scaled-index*** **=>** ***machine-word***

C-char-at-setter *Function*

`C-char-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
=> ***machine-word***

C-unsigned-short-at *Function*

`C-unsigned-short-at` ***C-pointer*** `#key` ***byte-index scaled-index***
=> ***machine-word***

C-unsigned-short-at-setter *Function*

`C-unsigned-short-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
=> ***machine-word***

C-signed-short-at *Function*

`C-signed-short-at` ***C-pointer*** `#key` ***byte-index scaled-index*** **=>** ***machine-word***

C-signed-short-at-setter *Function*

`C-signed-short-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
=> ***machine-word***

C-short-at *Function*

`C-short-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-short-at-setter *Function*

`C-short-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-unsigned-long-at *Function*

`C-unsigned-long-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-unsigned-long-at-setter *Function*

`C-unsigned-long-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-signed-long-at *Function*

`C-signed-long-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-signed-long-at-setter *Function*

`C-signed-long-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-long-at *Function*

`C-long-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-long-at-setter *Function*

`C-long-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-unsigned-int-at *Function*

`C-unsigned-int-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-unsigned-int-at-setter *Function*

`C-unsigned-int-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-signed-int-at *Function*

`C-signed-int-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-signed-int-at-setter *Function*

`C-signed-int-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-int-at *Function*

`C-int-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***machine-word***

C-int-at-setter *Function*

`C-int-at-setter` ***new C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***machine-word***

C-double-at *Function*

`C-double-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***float***

C-double-at-setter *Function*

`C-double-at-setter` ***new-double-float C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***float***

C-float-at *Function*

`C-float-at` ***C-pointer*** `#key` ***byte-index scaled-index*** `=>` ***float***

C-float-at-setter *Function*

`C-float-at-setter` ***new-single-float C-pointer*** `#key` ***byte-index scaled-index***
`=>` ***float***

C-pointer-at *Function*

`C-pointer-at` ***C-pointer*** **`#key`** ***byte-index scaled-index*** **`=>`** ***C-pointer***

C-pointer-at-setter *Function*

`C-pointer-at-setter` ***new C-pointer*** **`#key`** ***byte-index scaled-index***
`=>` ***C-pointer***

1.5.5 Structure types

<C-struct> *Open abstract class*

The abstract superclass of all classes designating a C struct type. It is a subclass of **`<C-value>`**. It is a subclass of **`<C-value>`**. You can describe new struct types using the **`define C-struct`** macro.

Classes designating C structs are not instantiable. Where a slot, array element, function parameter or function result is typed as a struct value, pointers to that struct type are accepted and returned.

1.5.6 Union types

<C-union> *Open abstract class*

The abstract superclass of all classes designating a C union type. It is a subclass of **`<C-value>`**. You can describe new union types with the macro **`define C-union`**. Classes designating C unions are not instantiable. Where a slot, array element, function parameter or function result is typed as a union value, pointers to that union type are accepted and returned.

1.5.7 Notes on C type macros

The C-FFI's C interface description language does not model all of the ways of defining new types in C, but all C types should be expressible in it. As a simplification, we do not support anonymous base types in the C interface description language. If a structure or union field has an in-line type definition in C, that definition must be extracted and given a name in order for it to be used. For example, the following C struct

```
struct something {
  char *name;
  long flags;
  union {
   long int_val;
   char *string_val;
  } val;
}
```

can be described with these definitions:

```
define C-union <anonymous-union-1>
  slot int-val    :: <C-long>;
  slot string-val :: <C-string>;
end C-union;

define C-struct <anonymous-struct-1>
  slot name  :: <C-string>;
  slot flags :: <C-long>;
  slot val   :: <anonymous-union-1>;
end C-struct;
```

The slots of these ex-inline types must be accessed through a chain of accesses, for example `o.val.string-val`.

1.6 Defining types

This section covers the definition macros that create Dylan designators for C types, structs and unions.

1.6.1 Defining specialized versions of designator classes

define C-subtype *Definition macro*

```
define [modifiers*] C-subtype name (superclasses)
  [slot-spec; …] [;]
  [type-options] [;]
end [C-subtype] [name]
```

Defines a specialized designator class for a C type based on an existing designator class for that type. It does this by defining a subclass of the original designator class, and is a simple wrapper around **`define class`** from which it takes its syntax. The *superclasses*, *slot-spec*s, and *modifiers* are passed on to **`define class`** unchanged. In effect, it expands to:

```
define class name (superclasses)
  slot-spec; ...
end class;
```

In terms of C, **`define C-subtype`** can be thought of as implementing a strongly typed version of **`typedef`** because a new designator class is generated that Dylan's type system can distinguish from the designator class on which it was based. As well as inheriting from an existing designator class, other Dylan classes can be mixed in too.

The optional *type-options* must be a property list. The **`c-name:`** keyword is recognized, allowing the original C name of the type designated by the class to be documented. The **`pointer-type-name:`** keyword option can be used to name the designator for pointers to *name*.

Some example C declarations:

```
typedef void *Handle;
typedef Handle WindowHandle;
typedef Handle StreamHandle;

extern WindowHandle CurrentWindow (void);
extern StreamHandle CurrentStream (void);
```

Example FFI definitions:

```
define C-subtype <Handle> (<void*>) end;
define C-subtype <WindowHandle> (<Handle>) end;
define C-subtype <StreamHandle> (<Handle>) end;

define C-function CurrentWindow
  result value :: <WindowHandle>;
  c-name: "CurrentWindow";
end C-function;

define C-function CurrentStream
  result value :: <StreamHandle>;
  c-name: "CurrentStream";
end C-function;
```

Example transactions:

```
? <void*> == <WindowHandle> | <WindowHandle> ==
               <StreamHandle>;
#f

? define variable *cw* = CurrentWindow();
// Defined *cw*

? *cw*
{<WindowHandle> #xff5400}

? define variable *cs* = CurrentStream();
// Defined *cs*

? *cs*
{<StreamHandle> #xff6400}

? instance?(*cs*, <WindowHandle>) |
            instance?(*cw*, <StreamHandle>);
#f
```

The following example uses the ability to specify extra superclasses to place a type beneath an abstract class.

Example C declarations:

```
struct _Matrix {
  int rank;
  int *dimensions;
  int *values;
};

typedef struct _Matrix *Matrix;

extern Matrix MatrixAdd (Matrix m, Matrix n);
```

Example FFI definitions:

```
define C-struct <_Matrix-struct>
  slot rank :: <C-int>;
  slot dimensions :: <C-int*>;
  slot values :: <C-int*>;
  pointer-type-name: <_Matrix-struct*>;
end C-struct;

define C-subtype <Matrix> (<_Matrix-struct*>, <number>) end;
define C-function MatrixAdd
  parameter m :: <Matrix>;
  parameter n :: <Matrix>;
  result value :: <Matrix>;
  c-name: “MatrixAdd”;
end C-function;

define method \+ (m1 :: <Matrix>, m2 :: <Matrix>) =>
   (r :: <Matrix>)
  MatrixAdd(m1, m2)
end method;
```

1.6.2 Defining specialized designator classes

define C-mapped-subtype *Definition macro*

```
define modifiers C-mapped-subtype type-name (superclasses)
  [pointer-type-name pointer-type-name[,
                   pointer-value-setter: boolean-value];]
  [map high-level-type  [, import-function: import-fun]
                        [, export-function: export-fun];]
  [import-map high-level-type,
   import-function: import-function;]
  [export-map high-level-type,
   export-function: export-function;]
end
```

The *modifiers* may be **sealed** or **open**. (The default is **sealed**.) Their effect on the class defined is the same as the similar modifiers on an ordinary class.

The possible combinations are, a map clause, an import-map clause, an export-map clause, or both an import-map and an export-map clause. Any other combinations are illegal.

The **import-map** clause specifies that a type conversion takes place when *type-name* is used as a designator for values are imported from C to Dylan. The conversion is accomplished by calling the *import-function* on the imported value. This call is automatically inserted into function wrappers, structure member getters, pointer-value dereference functions and so on by the C-FFI. The *high-level-type* is used as the Dylan type specifier for the appropriate parameter or result in any wrapper function or **c-struct** accessors which uses the defined class. The **export-map** clause specifies a similar type conversion for exported values. The *high-level-type* must in either case name an instantiable Dylan type.

```
  map <type-c>;
```

is equivalent to:

```
  import-map <type-c>;
  export-map <type-c>;
```

The import and export functions are monadic functions whose single argument is the appropriate low-level value for export functions and the appropriate Dylan type for import functions. Any mapped subtype which specifies an import-map must specify an *import-function*. Any mapped subtype which specifies an export-map must specify an *export-function*.

Map boolean example:

```
bool-header.h:

typedef int bool;

bool bool_function (bool b);
void bool_pointer_function (bool *b);

//eof

Module: my-module

define C-mapped-subtype <bool> (<C-int>)
  pointer-type <bool*>;
  map <boolean>,
    export-function:
      method (v :: <boolean>) => (result :: <integer>)
        as(<integer>, if(v) 1 else 0 end if) end,
    import-function:
      method (v :: <integer>) => (result :: <boolean>)
       ~zero?(v) end;
end;

//end module
```

Mapped string example: an alternate version of C-string which automatically converts instances of `<byte-string>` to instances of `<C-example-string>` on export.

```
string-header.h
```

```
typedef char *string;

string string-filter(string s);
void string-modifier(string *s);

//eof

module: my-module

define C-mapped-subtype <C-example-string> (<C-char*>,
                                            <string>)
  pointer-type <c-string*>;
  export-map type-union(<byte-string>,
                        <C-example-string>),
  export-function: c-string-exporter;
end;

define method c-string-exporter
  (s :: <byte-string>) => (result :: <C-char*>)
    as(<C-example-string>, s)
end;

define method c-string-exporter
  (s :: <C-example-string>) => (result :: <C-example-string>)
    s
end;

//end module
```

It is possible to define an ordinary subtype of a mapped supertype. The mapping characteristic of the subtype is inherited from the supertype. It is also possible to define a mapped subtype of a mapped supertype. When the subtype and supertype both specify an export function, the export functions of the subtype and the supertype are composed with the subtype's export function applied to the result of the supertype's export function. Import functions of a mapped subtype and supertype are similarly composed. Mapping characteristics are inherited from the supertype where the subtype does not define them. (You can think of this as composition with identity when either the supertype or subtype fails to specify an import or export function.) This shadowing is only useful when import and export maps are defined separately. Here is an example of a mapped subtypes which adds an import map to the mapped version of **`<C-example-string>`** defined above.

```
define C-mapped-subtype <other-string>
    (<C-example-string>)
  pointer-type <other-string*>;
  import-map <byte-string>,
    import-function: method (v :: <byte-string>) =>
      (result :: <C-example-string>)
        as(<C-example-string>, v)
      end method;
end;
```

The import signature is **`<byte-string>`**. The export signature is inherited from **`<C-example-string>`** **`type-union(<byte-string>, <C-example-string>)`**. For a example involving composition of mapped types consider the following (hypothetical) definitions of **`<C-raw-int>`**, **`<C-mapped-int>`** and **`<bool>`**. The **`<C-raw-int>`** class is a primitive which returns and accepts instances of **`<machine-word>`**. The **`<C-mapped-int>`** class is a mapped subtype which converts the instances of **`<machine-word>`** to instances of **`<integer>`**. The **`<bool>`** class is a mapped subtype of **`<C-mapped-int>`** which converts to and from **`<boolean>`**.

```
define C-mapped-subtype <C-mapped-int> (<C-raw-int>)
  pointer-type <bool*>;
  map <boolean>,
    export-function:
      method (v :: <integer>) =>
        (result :: <machine-word>)
          as(<machine-word>, v) end,
    import-function:
      method (v :: <machine-word>) =>
        (result :: <integer>)
          as(<integer>, v) end;
end;

define C-mapped-subtype <bool> (<C-mapped-int>)
  pointer-type <bool*>;
  map <boolean>,
    export-function:
      method (v :: <boolean>) => (result :: <integer>)
        if(v) 1 else 0 end if) end,
    import-function:
      method (v :: <integer>) => (result :: <boolean>)
        ~zero?(v) end;
end;
```

1.6.3 Describing structure types

define C-struct *Definition macro*

```
define C-struct name
  [slot-spec; ...] [;]
  [type-options] [;]
end [C-struct] [name]
```

The **define C-struct** macro describes C's aggregate structures. The *name* is defined to be a designator class encapsulating the *value* of a structure, *not* a pointer to the structure. This is significant because many of the protocols associated with structures work only on pointers to structures — pointers to structures being the most common form and the form closest to Dylan's object model. The new designator class is defined to be a subclass of **<C-struct>**.

Once defined, a structure-designating class is most likely to be used as the basis for a pointer type definition in terms of which most further transactions will take place. Structure-designating classes are abstract and cannot have direct instances. Accessor methods defined for the slots of the structure are specialized on the structure designator's pointer-type. However, the class itself may be needed to specify an in-line structure in another structure, union, or array, or a value-passed structure argument or result in a C function.

A *slot-spec* has the following syntax:

```
[slot-adjective] slot getter-name :: c-type #key setter address-getter c-name
                                                 dimensions width
```

The *slot-adjective* can be either **array** or **bitfield**. The **array** slot adjective indicates that the slot is repeated and the *dimensions* option is used to indicate how many repetitions are defined, and how it is accessed. The **bitfield** slot adjective indicates that the slot is really a bitfield. If **bitfield** is given then the *width* option must also be given. The *c-type* given for a **bitfield** slot must be a subclass of **<C-integer>**. The *c-type* for a **bitfield** slot indicates how the value is interpreted in Dylan by the slot accessor. A slot may not be specified as both an **array** and a **bitfield**.

The *getter-name* keyword specifies the name of the Dylan function to which the getter method for the structure slot will be added. The specializer of the getter method's single argument will be a designator indicating a pointer to the struct's *name*.

The *c-type* specifies the field's C type, and must be a designator class. Unlike Dylan slot specifications, the type declaration here is *not* optional.

The optional *setter* keyword specifies the generic function to which the setter method for the structure slot will be added. It defaults to *getter-name*`-setter`. No setter method is defined if the *setter* option is `#f`.

The optional *address-getter* specifies the name of a function that can be used to return a pointer to the data in the member. It must return a `<C-pointer>` object that points to a C type. No *address-getter* is defined by default.

You can use the *dimensions* keyword only if you used the `array` slot adjective. This *dimensions* value can be either a list of integers or a single integer. The accessor for an array slot is defined to take an extra integer parameter for each dimension given.

You can use the *width* keyword option only if you used the `bitfield` adjective.

The optional *c-name* keyword allows you to document the original C name of the slot.

The *type-options* clause is a property list allowing you to specify properties of the type as a whole. It accepts the optional keyword *c-name:*, allowing you to document the original C name of the struct to be documented. The optional keyword `pointer-type-name:` is also accepted. When given, the name is bound to the struct pointer type on which the accessors are defined.

The type option `pack:` *n* indicates that the struct has the packing semantics of Microsoft's `#pragma pack(`*n*`)`.

Example C declaration:

```
struct Point {
  unsigned short x;
  unsigned short y;
};

Point *OnePoint();   /* Returns a pointer to a Point */
Point *PointArray(); /* Returns a Point array */
```

Example FFI definition:

```
define C-struct <Point>
  slot x :: <C-unsigned-short>;
  slot y :: <C-unsigned-short>;
  pointer-type-name: <Point*>;
end C-struct;

define C-function one-point
  result point :: <Point*>;
  c-name: “OnePoint”;
end C-function;

define C-function point-array
  result array :: <Point*>;
  c-name: “PointArray”;
end C-function;
```

Example transactions:

```
? define variable p = one-point();
// Defined p.

? values(p.x, p.y);
100
50

? define variable array = point-array();
// Defined array.

? array[5].object-class; // implicit conversion to
                         // the pointer type
{<Point> pointer #xff5e00}

? begin array[5].x := 10; array[5].y := 20 end;
20

? values(array[5].x, array[5].y)
10
20
```

1.6.4 Describing union types

define C-union

Definition macro

```
define C-union name
  [slot-spec; ...] [;]
  [type-options] [;]
end [C-union] [name]
```

Describes C union types to the C-FFI. The syntax for the macro and its use are similar to **define c-struct** except that bitfield slots are not allowed.

Each of the slots in a union is laid out in memory on top of one another just as in C's **union** construct.

Example C declaration:

```
union Num {
  int    int_value;
  double double_value;
};

Num *OneNum();   /* Returns a pointer to a Num */
Num *NumArray(); /* Returns a Num array */
```

Example FFI definition:

```
define C-union <Num>
  slot int-value    :: <C-int>;
  slot double-value :: <C-double>;
  pointer-type-name: <Num*>;
end C-union;

define C-function one-num
  result num :: <Num*>;
  c-name: “OneNum”;
end C-function;

define C-function num-array
  result array :: <Num*>;
  c-name: “NumArray”;
end C-function;
```

Example transactions:

```
? define variable n = one-num();
// Defined n.

? values(p.int-value, p.double-value);
154541
92832.e23 // or something

? define variable array = num-array();
// Defined array.

? array[5].object-class; // implicit conversion to
                         // the pointer type
{<Num> pointer #xff5e00}

? array[5].int-value := 0;
0

? array[5].double-value;
11232e-12 // or a different something
```

1.7 Functions

This section describes the C FFI macros that allow C functions to be made available to Dylan and Dylan functions available to C.

1.7.1 Function types

This section describes classes that designate C function types and how to construct them.

<C-function-pointer> *Open abstract class*

The superclass of all classes that designate a C function type. It is a subclass of **`<C-pointer>`**. The Dylan variable bound by **`define c-callable`** is of this type.

1.7.2 Describing C functions to Dylan

define C-function *Definition macro*

```
define C-function name
  [parameter-spec; ...]
  [result-spec;]
  [function-option, …; …]
end [C-function] [name]
```

Describes a C function to the C-FFI. In order for a C function to be called correctly by Dylan, the same information about the function must be given as is needed by C callers, typically provided by **extern** declarations for the function in a C header file: the function's name and the types of its parameters and results.

The result of processing a **define C-function** definition is a Dylan function which is bound to *name*. This function takes Dylan objects as arguments, converting them to their C representations according to the types declared for the parameters of the C function before calling the C function with them. If the C function returns results, these results are converted to Dylan representations according to the declared types of those results before being returned to the Dylan caller of the function. By default the function created is a raw method, not a generic function. A generic function method can defined by using the **generic-function-method:** option.

Either the *c-name* function option must be supplied, or the **indirect:** option must be supplied with a value other than **#f**, but not both.

A *parameter-spec* has the following syntax:

```
[adjectives] parameter [name ::] c-type #key c-name
```

If no parameters are specified, the C function is taken to have no arguments.

The *adjectives* can be either **output**, **input**, or both. The calling discipline is specified by the **input** and **output** adjectives.

By itself, **input** indicates that the argument is passed into the function by value. This option is the default and is used primarily to document the code. There is a parameter to the generated Dylan function corresponding to each **`input`** parameter of the C function.

The **`output`** adjective specifies that the argument value to the C function is used to identify a location into which an extra result of the C function will be stored. There is no parameter in the generated Dylan function corresponding to an **`output`** parameter of the C function. The C-FFI generates a location for the extra return value itself and passes it to the C function. When the C function returns, the value in the location is accessed and returned as an extra result from the Dylan function. The C-FFI allocates space for the output parameter's referenced type, passes a pointer to the allocated space, and returns **`pointer-value`** of that pointer. A pointer to a struct or union type may not be used as an output parameter.

Example of **`output`** parameter definition:

```
define C-function mix-it-up
  output parameter out1 :: <some-struct*>;
  output parameter out2 :: <C-int*>;
  result value :: <C-int>;
  c-name: "mix_it_up";
end C-function mix-it-up;
```

Example transaction:

```
? mix-it-up();
1
{<some-struct> pointer #xfefe770}
42
```

If both **`input`** and **`output`** are supplied, they specify that the argument value to the C function is used to identify a location from which a value is accessed and into which an extra result value is placed by the C function. There is a parameter to the generated Dylan function corresponding to each **`input output`** parameter of the C function that is specialized as the union of the export type of the referenced type of the type given for the parameter in **`define c-function`**, and **`#f.`** When the C function returns, the value in the location is accessed and returned as an extra result from the Dylan function. If an **`input output`** parameter is passed as **`#f`** from Dylan then a **`NULL`** pointer is passed to the C function, and the extra value returned by the Dylan function will be **`#f`**.

Example of **input output** parameter definition:

```
  define C-function mix-it-up
    input output parameter inout :: <C-int*>;
    result value :: <C-int>;
    c-name: "mix_it_up";
  end C-function mix-it-up;
```

Example transaction:

```
? mix-it-up(7);
1
14
```

Note that neither **output** nor **input output** affects the declared type of an argument: it must have the same type it has in C and so, because it represents a location, must be a pointer type.

A *result-spec* has the following syntax:

  result [*name* ::] ***c-type***

If no **result** is specified, the Dylan function does not return a value for the C result, and the C function is expected to have a return type of **void**.

Each *function-option* is a keyword–value pair. The **generic-function-method:** option may be either **#t** or **#f**, indicating whether to add a method to the generic function name or to bind a bare constant method directly to name. The default value for **generic-function-method:** is **#f**. The option **C-modifiers:** can be used to specify platform dependent modifiers for the C function being called. For example, on Windows, use **C-modifiers: "__stdcall"** if the C function to be called is defined to be a **__stdcall** function.

The **c-name:** option is used to specify the name of the C function as it is defined in the object or shared library file. The *c-name* must be a constant string.

The **indirect: #t** option defines a function that accepts a C function pointer as its first argument. and calls the function given with the signature described by the parameters and result given. In this case the Dylan function defined accepts one more argument than if *c-name* was given. The type specified for the first parameter of the Dylan function is

<c-function-pointer>. One of *c-name* or **indirect: #t** must be supplied, but not both.

Example C declarations:

```
/* Compute the length of a string */
int strlen(char *string);

/* Set the given locations to values,
   returning an error code */
int fill_locations(int *loc1, int* loc2);

/* Read at most as far as indicated in max_then_read,
   updating it to contain how much was actually read */

void read_stuff(int *max_then_read);
```

Example FFI definitions:

```
define C-function strlen
  parameter string :: <C-char*>;
  result    value  :: <C-int>;
  c-name: "strlen";
end C-function;

define C-function fill-locations
  output parameter loc1 :: <C-int*>;
  output parameter loc2 :: <C-int*>;
  result return-code    :: <C-int>;
  c_name: "fill_locations";
end C-function;

define C-function read-stuff
  input output parameter :: <C-int*>;
  c-name: "read_stuff";
end C-function;
```

Example transactions:

```
? strlen($my-c-string);
44

? fill-locations();
0
101 // extra output value
102 // extra output value

? read-stuff(100);
50 // extra output value
```

In effect, a **define C-function** such as:

```
define C-function foo
  parameter string :: <C-char*>;
  parameter count  :: <C-int>;
  result    value  :: <C-int>;
  c-name: "foo";
end C-function;
```

expands into something like:

```
define constant foo =
  method (string, count)
    let c-string = %as-c-representation(<C-char*>,
                                        string);
    let c-count = %as-c-representation(<C-int>, count);
    let c-result = %call-c-function("foo", c-string,
                                    c-count);
    %as-dylan-representation(<C-int>, c-result);
end;
```

with the declared type.

1.7.3 Describing Dylan functions for use by C

define C-callable-wrapper *Definition macro*

define C-callable-wrapper ***dylan-rep-name*** **of** ***dylan-function***
  [*parameter-spec*; ...] [;]
  [*result-spec*] [;]
  [*function-options*][;]
end [C-callable-wrapper]

Makes a Dylan function callable from C, by describing a C contract for the function. In order to generate a correct C-callable function wrapper, the same information about the function must be given as would be needed by C callers, typically provided by **extern** declarations for the function in a C header file: the types of its parameters and results.

The result of processing a **define C-callable-wrapper** definition is a function with a C entry point with the contract described. This function takes C values as arguments, converting them to Dylan representations according to the types declared for the parameters of the C function before calling the Dylan function with them. If the C function was described as returning results, the results of the call to the Dylan function are converted to C representations according to the declared types of those results before being returned to the C caller of the function.

If *dylan-rep-name* is supplied, it is a name that is bound to a C function-pointer object that points to the C entry point of the function being defined.

The *dylan-function* is a Dylan function that accepts the correct number of parameters, and is called by the C callable wrapper.

The *function-options* are a property list. This list may contain a string value for the *c-name* keyword. If a *c-name* is specified, that name is made visible to C as the name of the generated C-callable wrapper function. Given a compatible **extern** declaration, this allows C code to call Dylan code simply by invoking a named function.

If *dylan-rep-name* is specified, it is bound to an instance of a function-pointer designator class identifying the generated C-callable wrapper function. You can pass this pointer to C code for use as, for example, a callback.

A *parameter-spec* has the following syntax:

[*adjectives*] parameter [*name* ::] *c-type* #key *c-name*

If no parameters are specified, the C function is taken to have no arguments.

An *adjective* can be **input**, **output**, or both. The calling discipline is specified by the **input** and **output** adjectives.

If a parameter is **output**, the corresponding parameter is not passed to the Dylan function, but the Dylan function is expected to return an extra value that is placed in the location pointed to by the parameter. The type designated for the parameter must be a pointer type.

If a parameter is both **input** and **output**, the parameter must be a pointer type, and the value accepted by the Dylan function will be the result of **pointer-value** on that pointer. The Dylan function is expected to return an extra value which is placed into the location specified by the pointer passed to the C function. If the pointer passed to the C function is **NULL**, then the value passed to the Dylan function will be **#f**, and the extra value returned will be ignored.

There is currently no way to define a C-callable function that accepts a variable number of arguments.

A *result-spec* has the following syntax:

result [*name* ::] *c-type*

If no **result** is specified, the C function defined does not return a value. It is defined as what in C terminology is known as a *void* function.

Example C declarations:

```
/* Compute the length of a string */
int strlen(char *string);

/* Set the given locations to values, returning an
   error code */

int fill_locations(int *loc1, int* loc2);

/* Read at most as far as indicated in max_then_read,
   updating it to contain how much was actually read */

void read_stuff(int *max_then_read);
```

Example FFI definitions:

```
define method dylan-strlen (string) => (length) … end;

define C-callable-wrapper of dylan-strlen
  parameter string :: <C-char*>;
  result    value  :: <C-int>;
  c-name: "strlen";
end C-function;

define method dylan-fill-locations ()
 => (return-code :: <integer>,
     val1 :: <integer>,
     val2 :: <integer>)
…
end;
```

```
define C-callable-wrapper of dylan-fill-locations
  output parameter loc1 :: <C-int*>;
  output parameter loc2 :: <C-int*>;
  result return-code    :: <C-int>;
  c-name: "fill_locations";
end C-function;

define method dylan-read-stuff (max :: <integer>) =>
  (read :: <integer) …
end;

define C-callable-wrapper of dylan-read-stuff
  input output parameter max-then-read :: <C-int*>;
  c-name: "read_stuff";
end C-function;
```

Example C calls:

```
{
  int length, *loc1, *loc2, max_then_read;

  length = strlen("ABC");

  fill_locations(loc1, loc2);

  max_then_read = 100
  read_stuff(&max_then_read);
}
```

In effect, a **`define C-callable-wrapper`** such as:

```
define C-callable-wrapper of foo
  parameter string :: <C-char*>;
  parameter count  :: <C-int>;
  result    value  :: <C-int>;
  c-name: "foo";
end C-function;
```

expands into something like:

```
%c-callable-function "foo" (c-string, c-count)
    let dylan-string
      = %as-dylan-representation(<C-char*>, c-string);
    let dylan-count
      = %as-dylan-representation(<C-int>, c-count);
    let dylan-result
      = foo(dylan-string, dylan-count);
    %as-c-representation(<C-int>, dylan-result);
  end;
```

where the `%` functions perform the primitive conversions between Dylan and C representations, checking that their arguments are compatible with the declared type.

Callback example:

```
? define C-function c-sort
    parameter strings     :: <C-string*>;
    parameter compare     :: <C-function-pointer>;
    result sorted-strings :: <C-string*>;
    c-name: "sort";
  end C-function;

// Defined c-sort.

? define C-callable-wrapper callback-for-< of \<
    parameter string1 :: <C-string>;
    parameter string2 :: <C-string>;
    result    int     :: <C-int>;
  end C-callable-wrapper;

// Defined callback-for-<

? callback-for-<
{function pointer #xff6e00}
? c-sort(some-c-strings, callback-for-<);
{<C-string> array}
```

1.7.4 Variables

This section covers describing and accessing C variables.

define C-variable *Definition macro*

```
define C-variable getter-name :: c-type
  #key setter c-name
end [C-variable]
```

Describes C variables to the C-FFI. It defines a getter and setter function for accessing the variable's value. The *c-name* keyword argument is required and gives the C name of the variable to be accessed. The *setter* keyword allows you to specify the name of the setter function, or if a setter function is to be defined at all. If *setter* is `#f`, no setter function will be defined.

For integer, float, or pointer-typed C variables the representation is clear and unambiguous. For C struct or union typed variables the translation is not so simple. A C union or struct has no direct representation in Dylan. You may only have a reference to the C object in Dylan through a **`<c-pointer>`** object. For this reason, **`define c-variable`** is not permitted for variables with C aggregate types. Use **`Define C-address`** for those variables.

```
? define C-variable process-count :: <C-int>,
    c-name: "process_count" end;

? process-count();
57

? process-count() := 0;
0

? process-count();
0

? define C-variable machine-name-1 :: <C-char*>,
    c-name: "MachineName";
end;

? machine-name-1();
#{<C-char*> #xaaabc00}
```

In C and other static languages what is known as a variable is a named allocation of memory. To access a global C variable from Dylan it is occasionally necessary to get a handle to the location where that variable is kept. The **`define C-address`** macro can be used for this purpose.

define C-address *Definition macro*

```
define C-address name :: pointer-designator-type
  #key c-name
end [C-address] [name]
```

Defines a Dylan constant binding, *name*, that is a **`<C-pointer>`** which points to the location of the C global variable *c-name*.

Pointer-designator-type must be the type of the constant to be defined, and a subtype of **`<C-pointer>`**.

1.8 Allocating and deallocating C storage

C objects can be allocated by calling **`make`** on an associated wrapper class or by allocating them on the stack using the macro **`with-stack-structure`**.

The C component of a **`make`**-allocated object is not deallocated by default when the Dylan designator object is reclaimed by the garbage collector, so we provide a manual means of freeing this storage with the function **`destroy`**.

make *subclass(<C-pointer>)* *G.f. method*

`make` ***subclass(<c-pointer>)*** **`#key`** ***allocator element-count***
extra-bytes address **`=>`** ***C-pointer***

Allocates a C object on the heap, using whatever standard C allocation function is in use on the target platform (typically **`malloc`**) to allocate the storage. This method is applicable to subclasses of **`<C-pointer>`** and returns an instance of its argument class.

If the *pointer* option is provided, no new storage is allocated, but instead, a new pointer with the given machine word address is returned.

The *allocator* argument should be a Dylan function that can serve as an allocator. It must accept a single integer argument — the number of bytes to be allocated — and return a Dylan **`<machine-word>`** that represents the address of the memory it allocated.

The amount of storage allocated by default is the result of:

`size-of(`***pointer-wrapper-class*****`.referenced-type)`**

If a positive integer is passed as an *extra-bytes* option, that number of extra bytes is also allocated.

If a positive integer is passed as a *element-count* option, space for *element-count* copies of the referenced type is allocated, taking into account the *extra-bytes* option for each of them. The *element-count* argument can be used for allocating arrays of sizes that are not known statically. The keyword *element-count* is used for this option rather than *size* in order to avoid conflict with the *size* collection keyword. The logical size of a collection represented by a pointer wrapper and the number of array elements that implement it may differ; a null-terminated string is an example of such a case.

This **make** method calls **initialize** on the wrapper object it generates before returning it.

```
? define variable *space-for-one-int* = make(<C-int*>);

? *space-for-one-int*[0];
97386437634   // Could have been anything unless the
              // default

// allocator guarantees to zero new memory.

? *space-for-one-int*[0] := 0;
0

? *space-for-one-int*[0];
0

? define variable *space-for-ten-ints*
    = make(<C-int*>, element-count: 10);

? define C-struct <Z-properties>
    slot type :: <C-int>;
    array slot properties :: <C-int>,
  end C-struct <Z-properties>;

? define variable *props* =
    make(<Z-properties>,
         extra-bytes: 10 * size-of(<C-int>));
```

destroy *Open generic function*

destroy ***C-pointer*** **#key** ***deallocator*** **=> ()**

Frees the allocated heap memory at the address encapsulated in *C-pointer.*

The *deallocator* argument should be a Dylan function that can serve as a deallocation facility. It must accept an address as a **<machine-word>** and free the storage allocated at that address.

You should only use **destroy** on pointers allocated using **make** where no address was given. If *allocator* was passed to **make**, the matching deallocator should be passed to **destroy**.

There is a default method for destroy on **`<C-statically-typed-pointer>`**.

with-stack-structure *Macro*

```
with-stack-structure (name :: wrapper-type
                      #key element-count extra-bytes)
  body
end [with-stack-structure]
```

Allocates an object *name* within the scope of a *body*. The *element-count* and *extra-bytes* options behave as in **`make`**. The memory that was allocated is freed after *body* exits. This macro gives the object *dynamic extent*.

```
? define C-struct <PointStruct>
    slot x-coord :: <C-unsigned-short>;
    slot y-coord :: <C-unsigned-short>;
    pointer-type-name: <PointStruct*>
  end C-struct;

// Defined <PointStruct>, x-coord, x-coord-setter,
   y-coord, and y-coord-setter.

? define constant <Point> = <PointStruct*>;
// Defined <Point>.

? define C-function PlotPoint
    parameter point :: <Point>;
    c-name: "PlotPoint";
  end C-function;
// Defined PlotPoint.

? define method plot (x, y)
    with-stack-structure (point :: <Point>)
      point.x-coord := 20;
      point.y-coord := 30;
      PlotPoint(point);
    end;
  end;
// Defined plot.

? plot(20, 20);

? plot(50, 50);
```

1.9 Utility designator classes

The following designator classes are defined for convenience purposes using **`define c-mapped-subtype`**.

<C-boolean> *Open abstract class*

A mapped subclass of **`<C-int>`** that provides an analogue to Dylan's **`<boolean>`** class. The Dylan type for both import and export is **`<boolean>`**, and the C type is **`int`**. The C integer 0 is mapped to **`#f`** in Dylan, and all other values are mapped to **`#t`**.

<C-string> *Open abstract class*

A mapped subclass of **`<C-char*>`** and **`<string>`**. On export the Dylan types **`<C-string>`**, or **`<byte-string>`** may be passed to C. On import all values are mapped to **`<C-string>`**. A **`<byte-string>`** may be passed to C directly and no copying takes place. The value in C will be a pointer to the data of the byte-string. The implementation of **`<byte-string>`** is such that, unless there are **`NULL`** characters embedded in the string, **`strlen`** in C and size in Dylan will return the same value.

A **`<byte-string>`** may only be safely passed to a C function if its value is never stored and used after the call returns.

<C-character> *Open abstract class*

A mapped subclass of **`<C-raw-unsigned-char>`**. The Dylan type for inport and export is **`<character>`**. The C type is **`unsigned char`**.

with-c-string *Macro*

```
with-c-string (variable = string-valued-expression)
  body
end
```

Use this macro when you need to pass C a pointer to the contents of a **`<byte-string>`**, but for some reason it cannot be passed directly. Inside the *body*, *variable* is bound to a **`<C-string>`** object that refers to the contents of the string returned by *string-valued-expression*.

Note: The **`<c-string>`** object is only live during the period that *body* is executing. If the program holds onto the pointer after that, the data it refers to cannot be guaranteed to be correct, because the garbage collector can no longer keep track of it.

<C-Dylan-object> *Open abstract class*

A mapped subclass of **`<C-void*>`**. Objects of this type correspond to specific Dylan objects. The Dylan type for import and export is **`<C-Dylan-Object>`**. The C type is **`void*`**.

To pass a reference to an arbitrary Dylan object to C, the Dylan object first must be registered using **`register-C-Dylan-object`**. Then a **`<C-Dylan-object>`** *handle* to the object can be created using the function **`export-C-Dylan-object`**. The handle can then be passed directly to any C transition point designated as **`<C-Dylan-object>`**. Any object received by Dylan from a transition point designated as **`<C-Dylan-object>`** may be passed to **`import-C-Dylan-object`** to get the Dylan object for which it was a handle.

register-C-Dylan-object *Function*

`register-C-Dylan-object` ***object***

unregister-C-Dylan-object *Function*

`unregister-C-Dylan-object` ***object***

These functions allows objects to be passed to a C function as instances of **`<C-Dylan-object>`**.

The **`register-C-Dylan-object`** function arranges for the garbage collector to leave the storage used by *object* unclaimed, and assures that the handle passed to C is not accidentally corrupted (from C's point of view) by the memory manager.

When the handle is no longer needed from C, you should call **`unregister-C-Dylan-object`** to allow the object to be normally reclaimed by the memory manager. Calls to **`register-C-Dylan-object`** and **`unregister-C-Dylan-object`** on the same object nest or interleave without interference. That is, if **`register-C-Dylan-object`** is called exactly twice on an object then **`unregister-C-Dylan-object`** must be called exactly twice before the memory manager can reclaim the space for the object as it normally would.

export-C-Dylan-object *Function*

`export-C-Dylan-object` ***object*** **`=>`** ***c-dylan-object***

Fetches the **`<C-Dylan-object>`** handle for a Dylan object.

import-C-Dylan-object *Function*

`import-c-dylan-object` ***c-dylan-object*** **`=>`** ***object***

Fetches the Dylan object for a **`<C-Dylan-object>`** handle.

2

Dylan Win32 API Libraries

2.1 Introduction

This chapter describes a set of Dylan libraries providing a low-level interface to Microsoft Windows. Each Dylan library is a Dylan C FFI built by a simple translation of the Windows API's C and C++ header files into C-FFI declarations. Windows programs can therefore be written in Dylan by using the same functions and types as you would in C, albeit with slightly modified names so that they conform to Dylan naming conventions and requirements.

The Dylan Win32 API has been constructed from several Dylan libraries. Win32 functionality is divided among the libraries to match the contents of Microsoft's DLLs, allowing Dylan applications to avoid references to DLLs they do not need to use, and also limiting the amount of information that must be loaded into the compilation environment.

Since, with the exception of changes necessitated by Dylan naming conventions and requirements, the names of C items have been preserved in the Dylan Win32 API libraries, this chapter does not provide an exhaustive list of items available. Instead, this chapter explains the name-mapping scheme that was used in the conversion and provides a collection of tips for writing Dylan applications with the libraries.

Note: This release supports the Win32 API for Windows NT and Windows 95. No support for Win16 (16-bit applications on Windows 3.*x*) is planned.

2.2 Win32 libraries provided

Each library has a single API module with the same name as the library. For example, the library **win32-common** has an API module also called **win32-common**.

You can access the libraries from the **System\Registry** folder in the distribution. For a default installation, this can be found in
C:\Program Files\Harlequin\Dylan.

win32-common	Data types, constants (including error codes), and structure accessors that are shared by the other modules. Most of these come from the Win32 header files **WINDEF.H**, **WINNT.H**, and **WINERROR.H**. (There is no DLL file supplied as standard with Windows that corresponds with this library, because there are no C functions in the header files to which it forms an interface.)
win32-kernel	Non-GUI system services, as implemented in **KERNEL32.DLL** and declared in **WINBASE.H** (files, pipes, semaphores, atoms, time, and so on), **WINCON.H** (NT console subsystem), and **WINNLS.H** (National Language Support). **Note:** This library does not provide thread support. Thread support is being handled at a higher level by Dylan's own Threads library. See the *Core Features and Mathematics* manual for details.
win32-gdi	Graphics Device Interface, drawing graphics and text, and printing. Corresponds to **WINGDI.H** and **GDI32.DLL**.
win32-user	Other windowing functions. Corresponds to **WINUSER.H** and **USER32.DLL**.
win32-version	Version management. Corresponds to **WINVER.H** and **VERSION.DLL**.
win32-dialog	Common dialog boxes, as implemented in **COMDLG32.DLL** and declared in **COMMDLG.H**, **DLGS.H**, and **CDERR.H**.

`win32-controls`

“Common controls”, including list view, tree view, property sheets, and so on (**`COMMCTRL.H`** and **`COMCTL32.DLL`**).

`win32-registry`

Registry (**`WINREG.H`** and **`ADVAPI32.DLL`**).

`win32-rich-edit`

“Rich edit” controls (**`RICHEDIT.H`** and **`RICHED32.DLL`**).

`win32-dde` Dynamic Data Exchange (**`DDE.H`** and **`DDEML.H`**).

Libraries already being considered for future releases include COM and OLE support. These libraries will be implemented with a single API module of the same name as the library.

2.3 Content and organization of the Win32 API libraries

The Dylan Win32 libraries are modeled closely upon the Win32 C libraries. Most names available in the Dylan libraries are the same as those available in the C libraries, though the Dylan libraries are a subset of what is available to C programmers. Moreover, to conform to Dylan naming conventions and restrictions, many of the C names have been translated.

Inefficiencies in the preliminary version of the Dylan compiler also limited the extent to which the **`win32-user`**, **`win32-gdi`**, and **`win32-kernel`** libraries could be completed. These libraries define only the minimal subset of names required for running example programs and for other short-term requirements.

Note: Look at the **`library.dylan`** file in each library to see what each library provides. (Look in **`comlib.dylan`** for **`win32-common`**.) If there is an additional area of Win32 you would like to see these libraries support, please tell the Dylan library team. E-mail: **`dylan-libraries`**.

The initial implementations of the libraries generally include only features that apply to both Windows NT and Windows 95.

2.3.1 Notes on the translations

The **`Win32-common`** module exports some names from the **`c-ffi`** module that its user may need to use directly, without needing to use (or know about) the **`c-ffi`** module itself. These names are: **`null-pointer`**, **`null-pointer?`**, **`pointer-address`**, **`pointer-value`**, **`pointer-value-setter`**, **`pointer-cast`**, **`<C-string>`**, **`<C-unicode-string>`**, **`destroy`**, and **`with-stack-structure.`**

Names that are documented as being obsolete and / or included in Win32 only for compatibility with Win16, are generally not defined in the Dylan libraries. The names excluded are listed in Section 2.4 on page 56.

The extended API macros, defined in the optional C header file **`WINDOWSX.H`**, are not supported.

2.4 Index of Win32 IDs excluded from the Dylan libraries

The names listed in the index below are excluded from the Dylan Win32 API libraries because they are obsolete.

Functions for old-style metafiles (**`CreateMetaFile`**, **`CloseMetaFile`**, and so on) are described in the Win32 SDK as being obsolete, but they are being supported because they are needed for OLE applications to exchange data with 16-bit applications.

Functions **`wsprintf`** and **`wvsprintf`** are not supported because the Dylan function **`format-to-string`** serves the same purpose. Also, the FFI doesn't currently support C functions with a variable number of arguments.

The extended API macros defined in optional C header file **`windowsx.h`** will not be supported by the Dylan interface.

The 64-bit utility macros **`Int32x32To64`**, **`Int64ShllMod32`**, **`Int64ShraMod32`**, **`Int64ShrlMod32`**, and **`UInt32x32To64`** are not planned to be supported since there is no clear need for them and the functionality can be obtained by using Dylan extended integers. However, an interface to function **`MulDiv`** is provided, since it is an ordinary C function that is commonly used.

2.4.1 Characters

`_hread, _hwrite, _lclose, _lcreat, _llseek, _lopen, _lread, _lwrite`

2.4.2 A

`AccessResource, AllocDSToCSAlias, AllocResource, AllocSelector, AnsiLowerBuff, AnsiNext, AnsiPrev, AnsiToOem, AnsiToOemBuff, AnsiUpper, AnsiUpperBuff`

2.4.3 B

`BN_DBLCLK, BN_DISABLE, BN_DOUBLECLICKED, BN_HILITE, BN_PAINT, BN_PUSHED, BN_UNPUSHED, BS_USERBUTTON`

2.4.4 C

`CPL_INQUIRE, ChangeSelector, CloseComm, CloseSound, CopyLZFile, CountVoiceNotes,`

2.4.5 D

`DOS3Call, DefHookProc, DefineHandleTable, DeviceMode, DlgDirSelect, DlgDirSelectComboBox`

2.4.6 E

`EnumFonts, ERR_..., ExtDeviceMode`

2.4.7 F

`FixBrushOrgEx, FlushComm, FreeModule, FreeProcInstance, FreeSelector`

2.4.8 G

`GCW_HBRBACKGROUND`, `GCW_HCURSOR`, `GCW_HICON`, `GWW_HINSTANCE`, `GWW_HWNDPARENT`, `GWW_ID`, `GWW_USERDATA`, `GetAspectRatioFilter`, `GetAtomHandle`, `GetBitmapBits`, `GetBitmapDimension`, `GetBrushOrg`, `GetCharWidth`, `GetCodeHandle`, `GetCodeInfo`, `GetCommError`, `GetCurrentPDB`, `GetCurrentPosition`, `GetEnvironment`, `GetFreeSpace`, `GetFreeSystemResources`, `GetInstanceData`, `GetKBCodePage`, `GetMetaFile`, `GetMetaFileBits`, `GetPrivateProfileInt` `GetPrivateProfileSection`, `GetPrivateProfileSectionNames`, `GetPrivateProfileString`, `GetPrivateProfileStruct`, `GetProfileInt`, `GetProfileSection`, `GetProfileString`, `GetStringTypeA`, `GetStringTypeW`, `GetTempDrive`, `GetTextExtent`, `GetTextExtentEx`, `GetTextExtentPoint`, `GetThresholdEvent`, `GetThresholdStatus`, `GetViewportExt`, `GetViewportOrg`, `GetWindowExt`, `GetWindowOrg`, `GlobalCompact`, `GlobalDosAlloc`, `GlobalDosFree`, `GlobalFix`, `GlobalLRUNewest`, `GlobalLRUOldest`, `GlobalNotify`, `GlobalPageLock`, `GlobalPageUnlock`, `GlobalUnWire`, `GlobalUnfix`, `GlobalUnwire`, `GlobalWire`

2.4.9 H

`HFILE`, `HFILE_ERROR`

2.4.10 L

`LZDone`, `LZStart`, `LimitEmsPages`, `LocalCompact`, `LocalInit`, `LocalNotify`, `LocalShrink`, `LockSegment`

2.4.11 M

`MAKEPOINT`, `MakeProcInstance`, `MoveTo`

2.4.12 N

`NetBIOSCall`

2.4.13 O

`OemToAnsi`, `OemToAnsiBuff`, `OffsetViewportOrg`, `OffsetWindowOrg`, `OpenComm`, `OpenFile`, `OpenSound`

2.4.14 P

PM_NOYIELD, **ProfClear**, **ProfFinish**, **ProfFlush**, **ProfInsChk**, **ProfSampRate**, **ProfSetup**, **ProfStart**, **ProfStop**

2.4.15 R

READ, **READ_WRITE**, **ReadComm**, **RegCreateKey**, **RegEnumKey**, **RegOpenKey**, **RegQueryValue**, **RegSetValue**

2.4.16 S

SYSTEM_FIXED_FONT, **ScaleViewportExt**, **ScaleWindowExt**, **SetBitmapDimension**, **SetCommEventMask**, **SetEnvironment**, **SetMetaFileBits**, **SetResourceHandler**, **SetScrollPos**, **SetScrollRange**, **SetSoundNoise**, **SetSwapAreaSize**, **SetViewportExt**, **SetViewportOrg**, **SetVoiceAccent**, **SetVoiceEnvelope**, **SetVoiceNote**, **SetVoiceQueueSize**, **SetVoiceSound**, **SetVoiceThreshold**, **SetWindowExt**, **SetWindowOrg**, **SetWindowsHook**, **StartSound**, **StopSound**, **SwitchStackBack**, **SwitchStackTo**, **SyncAllVoices**

2.4.17 U

UngetCommChar, **UnhookWindowsHook**, **UnlockSegment**

2.4.18 V

ValidateCodeSegments, **ValidateFreeSpaces**

2.4.19 W

WM_CTLCOLOR, **WNetAddConnection**, **WNetCancelConnection**, **WRITE**, **WaitSoundState**, **WriteComm**, **WritePrivateProfileSection**, **WritePrivateProfileString**, **WritePrivateProfileStruct**, **WriteProfileSection**, **WriteProfileString**

2.4.20 Y

`Yield`

2.5 API naming and mapping conventions

A Dylan Windows program will generally use the same API names as a C program would, with the following modifications for consistency with Dylan conventions.

2.5.1 Simple naming conventions

Type names are enclosed in angle brackets. For example, **`HANDLE`** becomes **`<HANDLE>`**.

Names of constants are prefixed by **`$`**. For example, **`OPAQUE`** becomes **`$OPAQUE`**.

Underscores are replaced by hyphens. Thus, a constant called **`NO_ERROR`** becomes **`$NO-ERROR`** and a class called **`LIST_ENTRY`** becomes **`<LIST-ENTRY>`**.

Hyphens will *not* be inserted between capitalized words (for example, **`CreateWindow`** does not become **`Create-Window`**) since that is a less obvious mapping that is more likely to cause confusion when switching between Dylan code and Windows documentation.

2.5.2 Mapping the null value

In place of **`NULL`**, there are several constants providing null values for frequently used types, such as **`$NULL-HANDLE`**, **`$NULL-RECT`**, and **`$NULL-STRING`**. Null values for other pointer types may be designated by the expression **`null-pointer(<FOO>)`**. Use the function **`null-pointer?`** to test whether a value is null. Do not use the expression **`if(ptr)...`** as is often done in C, since a null pointer is not the same as **`#f`**. There are also functions **`null-handle`** and **`null-handle?`** for creating and testing handles, since conceptually they are not necessarily pointers.

2.5.3 Mapping C types onto Dylan classes

The multitude of integer data types in C code (**`int`**, **`long`**, **`unsigned`**, **`ULONG`**, **`DWORD`**, **`LRESULT`**, and so on) are all designated as **`<integer>`** (or some appropriate subrange thereof) in Dylan method argument types. However, a **`<machine-word>`** needs to be used to represent values that do not fit in the signed 30-bit representation of an integer.

Names such as **`<DWORD>`** should not be used in application code because they refer to the FFI designation of the C value representation, not to a Dylan data type.

The C types **`BOOL`** and **`BOOLEAN`** are both mapped to **`<boolean>`** in Dylan. Use **`#t`** and **`#f`** instead of **`TRUE`** and **`FALSE`**.

Note: Beware that some functions, such as **`TranslateAccelerator`**, though documented to return **`TRUE`** or **`FALSE`**, actually return **`int`** instead of **`BOOL`**; in such a case, you will have to compare the result to 0.

Similarly, watch out for cases where C code passes **`TRUE`** or **`FALSE`** as an integer argument. To handle one common case, the Dylan implementation of **`MAKELPARAM`** accepts either an **`<integer>`** or **`<boolean>`** as the first argument.

The C types **`CHAR`**, **`WCHAR`**, and **`TCHAR`** are all mapped to **`<character>`** in Dylan. However, **`UCHAR`** is mapped to **`<integer>`** since that is how it is actually used.

Most of the pointer types in the Windows API have several names; for example: **`PRECT`**, **`NPRECT`**, and **`LPRECT`**. In 16-bit code, these distinguished between "near" and "far" pointers, but in 32-bit code there is no difference. Rather than carry the duplicate names over into Dylan, it would be simpler to use only the basic **`P...`** prefix names. However, the **`LP...`** names seem to be used much more often, and hence may be more familiar, and the Microsoft documentation still tends to use the **`LP...`** names in most places. So the Dylan interface defines both the **`<P...>`** and **`<LP...>`** names even though they have identical values. The **`NP...`** names are not defined in Dylan since they are not as commonly used.

Values of type **`char*`** in C are represented as instances of class **`<C-string>`** in Dylan. This is a subclass of **`<string>`**, so all of the normal string operations can be used directly. C function parameters of type **`char*`** will also accept an instance of **`<byte-string>`**; in the current implementation, that involves automatically copying the string at run time, but the need for copying is intended to be removed later.

The **`TEXT`** function can be used to coerce a string literal to a **`<C-string>`** such that the copy is performed only once. This usage is consistent with the Win32 **`TEXT`** macro, although the current purpose is different.

The Dylan declarations for C types will generally follow the *strict* alternative versions of the C declarations. This means, for example, that the various handle types such as **`<hmenu>`** and **`<hwnd>`** are disjoint subclasses of **`<handle>`**, instead of all being equivalent.

2.5.4 Creating methods from Windows alias functions

Consider a Windows function called, for example, **`Foo`**, which is an alias for either **`FooA`** (an 8-bit character version) or **`FooW`** (a 16-bit character version). In Dylan, only the name **`Foo`** will be defined, but it will be a generic function with separate methods for arguments of types **`<C-string>`**, **`<C-unicode-string>`**, **`<byte-string>`** or **`<unicode-string>`**. (Only the 8-bit versions will be supported in the initial implementation, both because the compiler is not ready to handle Unicode and because it will not work on Windows 95.)

2.5.5 Mapping C structure fields onto Dylan slot names

Because slot names are not in a separate name space in Dylan, the names of C structure fields will have the suffix **`-value`** added to form the name of the Dylan accessor function. For example, the C statement:

```
pt->x = x;
```

becomes in Dylan:

```
pt.x-value := x;
```

There is not any attempt to append **`?`** to the names of predicate functions since it is not obvious exactly which functions that should apply to. The Dylan convention of **`*...*`** for global variables is not relevant since there are no global variables involved.

2.5.6 Handling return of multiple values

In cases where the C library function takes a pointer argument as a place to store a pointer, integer, or boolean value, the corresponding Dylan function uses multiple return values to return such output parameters following the original function return value. For example, where C code does:

```
BOOL ok;
DWORD NumberRead;
ok = ReadConsoleInput(handle, buffer, length, & NumberRead);
```

in Dylan it would be:

```
let ( ok :: <boolean>, NumberRead :: <integer> ) =
    ReadConsoleInput(handle, buffer, length);
```

Similarly, this function returns multiple values instead of a structure:

```
let ( x, y ) = GetLargestConsoleWindowSize(handle);
```

2.6 Defining callback functions

The **win32-common** library provides a **define callback** macro to make it easy to define callback functions without the application programmer needing to use the FFI **define c-callable-wrapper** macro directly. It is used like this:

```
define callback WndProc :: <WNDPROC> = my-window-function;
```

This says that **WndProc** is being defined as a C function pointer of type **<WNDPROC>**, which when called from C causes the Dylan function **my-window-function** to be run. The Dylan function will be defined normally using **define method** or **define function**, and it is the responsibility of the programmer to ensure that its argument signature is consistent with what **<WNDPROC>** requires. For example:

```
define method my-window-function(
               hWnd  :: <HWND>,       // window handle
               message :: <integer>,  // type of message
               uParam,                // additional information
               lParam)                // additional information
        => return :: <integer>;
...
```

Note that the **uParam** and **lParam** arguments might receive values of either type **<integer>** or **<machine-word>**, so it may be best not to specialize them. Often these values are not used directly anyway, but are passed to other functions (such as **LOWORD** and **HIWORD**) which have methods for handling either representation.

The other types of function supported by **define callback** are dialog functions (**<DLGPROC>**) and dialog hooks (**<LP...HOOKPROC>**), both of which have the same argument types as a window function, but return a **<boolean>**. (The dialog hook functions are actually declared in **COMMDLG.H** as returning a **UINT**, but the value is always supposed to be **TRUE** or **FALSE**, so the Dylan callback interface has been implemented using **BOOL** instead.)

2.7 Dealing with the C function WinMain

In C, the programmer has to supply a **WinMain** function as the main program for a GUI application, but in Dylan there is no main program as such. The beginning of execution is indicated simply by a top-level function call expression; this needs to be at the bottom of the last file listed in the LID file (See *Library Reference Volume II: I/O and Networks* for details about the LID file format). Functions are provided to obtain the values which would have been the arguments to **WinMain**:

```
application-instance-handle() => <HINSTANCE>
application-command-line() => <string>
                             // arguments without program name
application-show-window() => <integer>   // one of $SW-...
```

There is no accessor provided for the **WinMain** previous instance parameter because on Win32, that parameter is always null, even for Win32s as well as NT and Win95.

In the Dylan environment, there is no concept of a command line, so **application-command-line** always returns a null pointer, and **application-show-window** always returns the default value **$SW-SHOWNORMAL**.

The program can be terminated by calling the **exit-application** function in the Operating-system library so that an exit code can be specified. But do not do that if the program is to be used in the Harlequin Dylan environment, since you will close down the remote application. Instead, use the standard Dylan **abort** function to return to the remote listener. There is no way to return an exit code.

The start of an application program might look something like this:

```
define method my-main ()
  let hInstance :: <HINSTANCE> = application-instance-handle();
  let wc :: <PWNDCLASS> = make(<PWNDCLASS>);
  wc.style-value := 0;
  wc.lpfnWndProc-value := MainWndProc;
...
  RegisterClass(wc);
  let hWnd = CreateWindow( ... );
  ShowWindow(hWnd, application-show-window());
  UpdateWindow(hWnd);
  let msg :: <PMSG> = make(<PMSG>);
  while ( GetMessage(msg, $NULL-HWND, 0, 0) )
    TranslateMessage(msg);
    DispatchMessage(msg);
  end;
  ExitProcess(msg.wParam-value);
end method my-main;

my-main();  // this is what initiates execution.
```

For a complete example program, see

```
Projects\Examples\windows-ffi-example\example.dylan
```

in the directory that you installed Harlequin Dylan (**C:\Program Files\Harlequin\Dylan\** by default).

2.8 Combining bit mask constants

Where C code would use the **|** operator to combine bit mask constants, Dylan code usually uses the **logior** function. However, a few such constants have values of type **<machine-word>** when they will not fit in a small integer, and **logior** only works on instances of **<integer>**. Because of this, the **win32-common** library exports a **%logior** function which is used like **logior** except that it accepts values of either type **<integer>** or **<machine-word>** and returns a **<machine-word>** result. It can be used in most places that accept a bit mask (C types **DWORD**, **ULONG**, **LPARAM**, and so on), but must be used if any of the arguments are a **<machine-word>**. The contexts where this is likely to occur are:

- Window style parameter of **CreateWindow ($WS-...)**
- Flags value for **CreateFile** or **CreateNamedPipe ($FILE-FLAG-...)**
- **$LOCALE-NOUSEROVERRIDE** for **LCTYPE** parameters for **GetLocaleInfoA**, **GetLocaleInfo**, and possibly others, or **dwFlags** parameter of

GetTimeFormat, **GetNumberFormat**, **GetCurrencyFormat**, or **GetDateFormat**.

- Mask and effects values in **CHARFORMAT** structure for “rich edit” controls **($CFM-...** and **$CFE-...)**
- Mask value in **PARAFORMAT** structure for “rich edit” controls **($PFM-...)**

2.9 Other minor details

The types **<FARPROC>** and **<PROC>** are defined as equivalent to **<C-function-pointer>**, so any C function wrapper object can be passed to a routine taking a **<FARPROC>** without needing to do any type conversion like that needed in C.

Type casts between handles and integers (**<integer>** or **<machine-word>**) can be done by using **as**. For example:

```
window-class.hbrBackground-value :=
                                    as(<HBRUSH>, $COLOR-WINDOW + 1);
```

For type casts from one pointer type to another, use the function **pointer-cast** instead of **as**. Think of **as** as converting the data structure pointed to, while **pointer-cast** operates on just the pointer itself.

The Dylan function **pointer-value** can be used to convert between a Dylan integer and a **LARGE-INTEGER** or **ULARGE-INTEGER**. For example:

```
let li :: make( <PLARGE-INTEGER> ); pointer-value(li) := 0;
```

allocates a **LARGE-INTEGER** and sets its value to 0, without needing to be concerned with the individual fields of the internal representation. Alternatively, you can use an initialization keyword:

```
let li :: make( <PLARGE-INTEGER>, value: 0 );
```

The C macros **MAKEPOINT**, **MAKEPOINTS**, and **LONG2POINT** do not easily translate to Dylan. Instead, use the Dylan function **lparam-to-xy** to split a parameter into two signed numbers. For example:

```
let ( x, y ) = LPARAM-TO-XY(lParam);
```

Where the Windows documentation specifies using a value of **0xFFFFFFFF**, this should be denoted in Dylan by using the constant **$FFFFFFFF** because the literal **`#xFFFFFFFF`** is outside the range for a small integer. (In the SCC or VM-Tether, it would be silently truncated, while the DFMC would report a type error.)

In Dylan, **`<RECTL>`** is an alias of **`<RECT>`** instead of being a distinct type. (In Win32, they are structurally equivalent but were separate types for the sake of source code compatibility with Win16; there is no need to maintain that artificial distinction in Dylan.)