

---

# **Felix Tutorial**

***Release 2016.07.12-rc1***

**Aug 01, 2020**



---

## Contents

---

<b>1</b>	<b>Hello</b>	<b>3</b>
1.1	Basics: Hello . . . . .	3
<b>2</b>	<b>Felix 101: the Basics</b>	<b>5</b>
2.1	Comments . . . . .	5
2.2	Identifiers . . . . .	6
2.3	Variables . . . . .	6
2.4	Logic Type Bool . . . . .	8
2.5	Integers . . . . .	9
2.6	Slices . . . . .	11
2.7	Floating Point Numbers . . . . .	12
2.8	Strings . . . . .	14
2.9	Characters . . . . .	16
2.10	Simple Control Flow . . . . .	17
2.11	Loops . . . . .	18
2.12	Arrays . . . . .	19
2.13	Tuples . . . . .	20
2.14	Procedures . . . . .	20
2.15	Functions . . . . .	22
2.16	Function Application . . . . .	24
2.17	Pythagoras . . . . .	24
<b>3</b>	<b>Felix 102: More Detail</b>	<b>27</b>
3.1	Record . . . . .	27
3.2	Classes . . . . .	28
3.3	Generic Functions . . . . .	28
3.4	Lists . . . . .	29
3.5	Option Type . . . . .	30
3.6	Varray . . . . .	31
3.7	Objects, Values and Pointers . . . . .	32
3.8	The new operator . . . . .	33
3.9	Pointer projections . . . . .	33
<b>4</b>	<b>Felix 103: Polymorphism</b>	<b>35</b>
4.1	Polymorphic Functions . . . . .	35
4.2	Higher Order Functions . . . . .	35

<b>5</b>	<b>Felix GUI</b>	<b>37</b>
5.1	Getting Started With the Felix GUI . . . . .	37
5.2	GUI Basics . . . . .	38
5.3	Putting Stuff in the Window . . . . .	39
5.4	Using An Event Handler . . . . .	42
5.5	Using a Window Manager . . . . .	46

An introduction to the Felix programming language.

Contents:



---

Hello

---

This is the Felix tutorial

## 1.1 Basics: Hello

### 1.1.1 Running a Program

Let's start with a simple script:

```
println$ "Hello World!";
```

To run it you just say:

```
flx hello.flx
```

It's pretty simple. Felix runs programs like Python does, you run the source code directly. Behind the scenes, Felix translates the program into C++, compiles the program, and runs it. All the generated files are cached in the `.felix/cache` subdirectory of your `$HOME` directory on Unix like systems, and `$USERPROFILE` on Windows.

This means the script can be run in a read-only directory.





---

## Felix 101: the Basics

---

The first module gives a brief overview of some basics.

### 2.1 Comments

Felix has two kinds of comments.

#### 2.1.1 C++ comments

C++ style comments begin with `//` and end at the end of the line.

```
println$ "Hi"; // say Hi!
```

#### 2.1.2 Nested C comments

C style comments begin with `/*` and end with a matching `*/`.

```
/* This is an introductory program,  
   which says Hello!  
*/  
println$ "Hello"; /* Say it! */
```

Unlike C comments, in Felix C style comments nest. Take care of and leadin or leadout marks hidden in string literals!

```
// a comment in C++ style  
/* a C style comment  
   /* nested comment */  
   still commented  
*/
```

Nested comments are often used to temporarily remove code:

```
/*  
/* This is an introductory program,  
   which says Hello!  
*/  
println$ "Hello"; /* Say it! */  
/*  
println$ "Bye";
```

## 2.2 Identifiers


Felix has a rich lexicology for identifiers.

### 2.2.1 C like identifiers

C like identifiers start with a letter, and are optionally followed by a sequence of letters, underscores, single quotes, hyphens, or ASCII digits.

The first character may also be an underscore but that is reserved for the system.

A letter may be any Unicode code point accepted by ISO C++ Standard as a letter, it must be encoded as UTF-8.

```
Hello  
X123  
a '  
julie-marx  
x 
```

Note again please, only ASCII digits are permitted.

### 2.2.2 Tex Identifiers

A leading slash followed by a nonempty sequence of ASCII letters is recognised as an identifier. With suitable output machinery, the corresponding LaTeX or AmSTeX symbol should display if there is one.

```
\alpha
```

displays as

```
\(\alpha\)
```

No space is required after a TeX identifier. Therefore

```
\alpha2
```

encodes the symbol alpha followed by the integer 2.

## 2.3 Variables

Felix provides three simple forms to define and initialise variables. The *var* binder is used to define a variable, it binds a name to a storage location.

### 2.3.1 Variables with type and initialiser

A variable can be defined with a type annotation and an initialiser.

```
var c : bool = true;
var k : int = 1;
var s : string = "Hello";
var b : double = 4.2;
```

The specified type must agree with the type of the initialiser.

### 2.3.2 Variables without type annotation

A variable can also be defined without a type annotation provided it has an initialiser.

```
var c = true;
var k = 1;
var s = "Hello";
var b = 4.2;
```

In these cases the type of the variable is the type of the initialiser.

### 2.3.3 Variables without initialiser

Variables can be defined without an initialiser.

```
var c : bool;
var k : int;
var s : string;
var b : double;
```

In this case the variable will be initialised by the underlying C++ default initialiser. It is an error to specify a variable this way if the underlying C++ type does not have a default initialiser.

If the underlying C++ default initialiser is trivial, so that the store is not modified, then the Felix variable is uninitialised.

### 2.3.4 Simple Assignment

An assignment can be used to assign the first value stored in the location of a variable, to modify the value which an explicit initialiser previously provided, or to modify the value which the underlying C++ default initialiser provided.

```
var c : bool;
var k = 1;
c = false;
k = 2;
```

Assignments are executed when control flows through the assignment.

### 2.3.5 Variable Hoisting

Var binders are equivalent to declaration of an uninitialised variable and an assignment. The location of the declaration within the current scope is not relevant. The position of an initialising assignment is. For example:

```
a = 1;  
var b = a;  
var a : int;
```

is equivalent to

```
var a = 1;  
var b = a;
```

## 2.4 Logic Type Bool

Felix provides a type for simple logic which traditionally is called *bool* after mathematician George Bool.

### 2.4.1 Type

In Felix, *bool* is a special case of a more general mechanism we will meet later. It is an *alias* for the type *2*, which is the type that handles two alternatives:

```
typedef bool = 2;
```

The *typedef* binder binds a name to an existing type, that is, it creates an alias.

### 2.4.2 Constants

There are two predefined constants of type *bool*, *true* and *false*.

### 2.4.3 Operations

The prefix operator *not* provides negation, infix *and* conjunction, and infix *or* disjunction, with weak precedences, of decreasing strength.

```
not a and b or c
```

is parsed as

```
((not a) and b) or c
```

These operators are all weaker than the comparisons they often take as arguments, so that

```
a < b and b < c
```

is parsed as

```
(a < b) and (b < c)
```

## 2.4.4 Summary: Logical operations

Operator	Type	Syntax	Semantics
or	bool * bool -> bool	Infix	Disjunction
and	bool * bool -> bool	Infix	Conjunction
not	bool -> bool	Prefix	Negation

## 2.5 Integers

In Felix we have a type named *int* which has values which are small integers close to zero.

### 2.5.1 Type

This type is defined by lifting it from C++ in the library by:

```
type int = "int";
```

The exact range of integers represented is therefore determined by the underlying C++ implementation.

### 2.5.2 Literals

Non-negative values of type *int* can be written as a sequence of the decimal digits like this:

```
0
1
23
42
```

You can also put an underscore or single quote between any two digits:

```
123_456
123'456
```

### 2.5.3 Negation

There are no negative integer literals. However you can find the negative of an integer using a prefix negation operator, the dash character -, or the function *neg*:

```
-1
neg 1
```

### 2.5.4 Infix Operators

Integers support simple formulas:

```
(12 + 4 - 7) * 3 / 6 % 2
```

Here, + is addition, infix - is subtraction, \* is multiplication, / is division, and % is the remainder after a division.

### 2.5.5 Sign of quotient and remainder

Division and remainder require a little explanation when negative numbers are involved. The quotient of a division in C, and thus Felix, always rounds towards zero, so:

```
-3/2 == -1
```

The the quotient is non-negative if the two operands are both negative or both non-negative, otherwise it is non-positive. The remainder, then, must satisfy the formula:

```
dividend == quotient * divisor + remainder
```

so that we have

```
remainder == dividend - quotient * divisor
```

Therefore the remainder is non-negative if, and only if, the dividend is non-negative, otherwise it is non-positive.

### 2.5.6 Comparisons

We provide the usual comparisons from C: `==` is equality, `!=` is inequality, `<` is less than, `>` is greater than, `<=` is less than or equal to, and `>=` is greater than or equal to.

The result of a comparison is value of *bool* type.

### 2.5.7 Constant Folding

If you write a formula involving only literals of type *int*, the Felix compiler will perform the calculation according to mathematical rules, using a very much bigger integer representation. At the end, the result will be converted back to the smaller *int* representation.

If the result of the calculations exceeds the size of the compiler internal representation, or, the final result is too large for an *int*, the result is indeterminate.

### 2.5.8 Division by Zero

If a division or remainder operation has a divisor of zero, the compiler may abort the compilation, or it may defer the problem until run time. If the problem is deferred and the code is executed, an exception will be thrown and the program aborted. However the code may not be executed.

### 2.5.9 Out of bounds values

If the result of a calculation performed at run time is out of bounds, the result is indeterminate.

## 2.5.10 Summary: Integer Comparisons

Operator	Type	Syntax	Semantics
<code>==</code>	<code>int * int -&gt; bool</code>	Infix	Equality
<code>!=</code>	<code>int * int -&gt; bool</code>	Infix	Not Equal
<code>&lt;=</code>	<code>int * int -&gt; bool</code>	Infix	Less or Equal
<code>&lt;</code>	<code>int * int -&gt; bool</code>	Infix	Less
<code>&gt;=</code>	<code>int * int -&gt; bool</code>	Infix	Greater or Equal
<code>&gt;</code>	<code>int * int -&gt; bool</code>	Infix	Greater

## 2.5.11 Summary: Integer Operations

Operator	Type	Syntax	Semantics
<code>+</code>	<code>int * int -&gt; int</code>	Infix	Addition
<code>-</code>	<code>int * int -&gt; int</code>	Infix	Subtraction
<code>*</code>	<code>int * int -&gt; int</code>	Infix	Multiplication
<code>/</code>	<code>int * int -&gt; int</code>	Infix	Division
<code>%</code>	<code>int * int -&gt; int</code>	Infix	Remainder
<code>-</code>	<code>int -&gt; int</code>	Prefix	Negation
<code>neg</code>	<code>int -&gt; int</code>	Prefix	Negation
<code>abs</code>	<code>int -&gt; int</code>	Prefix	Absolute Value

## 2.5.12 More Integers

Felix has many more integer types. See the reference manual:

<https://felix.readthedocs.io/en/latest/integers.html>

for details.

## 2.6 Slices

A slice is a range of integers.

### 2.6.1 Type

The type is defined in the library as

```
slice[int]
```

### 2.6.2 Inclusive Slice

From first to last, inclusive:

```
first..last
```

### 2.6.3 Exclusive Slice

From first to last, excluding last:

```
first..<last
```

### 2.6.4 Counted Slice

From first for a certain number of values:

```
first..+count
```

### 2.6.5 Infinite Slices

Felix provides three apparently infinite slices, which are actually bounded by the limits of the integer type:

```
first..       // first..maxval[int]  
..last      // minval[int]..last  
..<last      // minval[int]..<last
```

There is also a slice over the whole range of a type:

```
Slice_all[int]  
..int]
```

### 2.6.6 Empty Slice

There is an empty slice too:

```
Slice_none[int]
```

### 2.6.7 More Slices

More detail on slices in the reference manual:

<https://felix.readthedocs.io/en/latest/slices.html>

## 2.7 Floating Point Numbers

Floating point literals are local approximations to reals. Local means close to zero. Floats are dense near zero and lose precision far from it.

### 2.7.1 Type

We lift the main floating point type, *double* from C.

```
type double = "double";
```

It is a double precision floating point representation usually conformat to IEEE specifications.



## 2.7.2 Literals

Floating literals have two parts, a decimal number known as the mantissa, and a power of 10 known as the exponent.

```
12.34
12.34E-4
```

The mantissa must contain a decimal point with a digit on either side. The exponent is optional, and consists of the letter *E* or *e* followed by a small decimal integer literal, or a + sign or minus sign, and a small decimal integer literal.

If the exponent is present, the mantissa is multiplied by 10 raised to the power of the signed integer part exponent.

## 2.7.3 Operations

Floating numbers support negation with prefix -, addition with infix +, subtraction with infix -, multiplication with infix \* and division with infix / as well as many other operations given by functions in the library.

It is also possible to perform comparisons, equality ==, inequality !=, less than <, less than or equal to <=, greater than > and greater than or equal to >=. However these comparisons reflect floating point arithmetic which only approximates real arithmetic. Do not be surprised if the formula

```
1.0 / 3.0 * 3.0 == 1.0
```

is false. To remedy this properly requires a deep knowledge of numerical analysis. Felix helps by providing the function *abs* which can be used like this:

```
abs ( 1.0 / 3.0 * 3.0 - 1.0 ) < 1.0e-3
```

to check the result is with about 3 decimal places of 1.0.

## 2.7.4 Summary: Double Comparisons

Operator	Type	Syntax	Semantics
==	double * double -> bool	Infix	Equality
!=	double * double -> bool	Infix	Not Equal
<=	double * double -> bool	Infix	Less or Equal
<	double * double -> bool	Infix	Less
>=	double * double -> bool	Infix	Greater or Equal
>	double * double -> bool	Infix	Greater

## 2.7.5 Summary: Double Operations

Operator	Type	Syntax	Semantics
+	double * double -> double	Infix	Addition
-	double * double -> double	Infix	Subtraction
*	double * double -> double	Infix	Multiplication
/	double * double -> double	Infix	Division
-	double -> double	Prefix	Negation
neg	double -> double	Prefix	Negation
abs	double -> double	Prefix	Absolute Value

## 2.7.6 More Floats

Felix has more floating types. See the reference manual:

<https://felix.readthedocs.io/en/latest/floats.html>

for details.

## 2.8 Strings

A string is basically a sequence of characters with a definite length. The type is the traditional C++ string type, and supports Unicode only by UTF-8 encoding.

Strings are 8 bit clean, meaning all 256 characters, including nul, may occur in them.

However string literals may not directly contain a nul. String literals are actually C nul terminated char strings which are lifted to C++ automatically.

### 2.8.1 Type

The string type in Felix is the based on C++:

```
type string = "::std::basic_string<char>"
  requires Cxx_headers::string
;
```

### 2.8.2 Literals

There are two simple forms of string literals:

```
var x = "I am a string";
var y = 'I am a string too';
```

using double quote delimiters and single quote delimiters. These literals cannot exceed a single line. However you can concatenate them by juxtaposition:

```
var verse =
  "I am the first line,\n"
  'and I am the second.'
;
```

Notice the special encoding `\n` which inserts an end of line character into the string rather than a `\` followed by an `n`. This is called an escape.

You can prevent escapes being translated with raw strings like this:

```
r"This \n is retained as two chars"
```

Only double quoted strings can be raw.

Felix also has triple quoted strings, which span line boundaries, and include end of lines in the literal:

which contains two end of line characters in the string, whilst this one:

```
'''
Here is another
long string
'''
```

has three end of lines (one after the first triple quote).

### 2.8.3 Length

Use the *len* function:

```
var x = "hello";
var y = x.len.int;
```

### 2.8.4 Concatenation

Strings can be concatenated with the infix `+` operator or just written next to each other, juxtaposition has a higher precedence than infix `+`.

```
var x = "middle";
var y = "Start" x + "end";
```

In this case, the first concatenation of `x` is done first, then the second one which appends “end”. The result is independent of the ordering because concatenation is associative, the run time performance, however, is not, because concatenation requires copying.

### 2.8.5 Substring Extraction

A substring of a string can be extracted using a slice with the notation shown:

```
var x = "Hello World";
var y = x.[3..7]; // 'lo Wo'
```

### 2.8.6 Indexing

Select the *n*’th character:

```
var x = "Hello World";
var y = x.[1]; // e
```

### 2.8.7 Comparisons

Strings are totally ordered using standard lexicographical ordering and support the usual comparison operators.

## 2.8.8 Summary: String Comparisons

Operator	Type	Syntax	Semantics
==	string * string -> bool	Infix	Equality
!=	string * string -> bool	Infix	Not Equal
<=	string * string -> bool	Infix	Less or Equal
<	string * string -> bool	Infix	Less
>=	string * string -> bool	Infix	Greater or Equal
>	string * string -> bool	Infix	Greater

## 2.8.9 Summary: Double Operations

Operator	Type	Syntax	Semantics
len	string -> size	Prefix	Length
+	string * string -> string	Infix	Concatenation
.[_]	string * slice -> string	Postfix	Substring
.[_]	string * int -> char	Postfix	Indexing

## 2.9 Characters

The atoms of a string are ASCII characters.

### 2.9.1 Type

Lifted from C++:

```
type char = "char";
```

### 2.9.2 Extraction from String

The first character of a string can be extracted:

```
var ch : char = char "Hello";
```

This sets `ch` to the letter `H`. If the string is empty, the `char` becomes `NUL`.

### 2.9.3 Ordinal Value

The code point of the character, from 0 to 255, as an *int*:

```
var ch = char "Hello"; // H
var j = ch.ord; // 72
```

## 2.9.4 Construction from Ordinal Value

Finds the n'th character in the ASCII character set.

```
var ch = char 72; // H
```

## 2.10 Simple Control Flow

### 2.10.1 Sequential Flow

Normally, control flows from one statement to the next.

```
first;
second;
third;
```

### 2.10.2 No operation

Felix provides several statements which do nothing. A lone semi-colon ; is a statement which does nothing.

```
first;
; // does nothing
second;
```

### 2.10.3 Labels and Gotos

The default control flow can be modified by labelling a position in the code, and using a goto or conditional goto targetting the label.

```
var x = 1;
next:>
println$ x;
if x > 10 goto finished;
x = x + 1;
goto next;
finished:>
println$ "Done";
```

An identifier followed by :> is used to label a position in the program.

The unconditional *goto* transfers control to the statement following the label.

The conditional goto transfers control to the statement following the label if the condition is met, that is, if it is true.

### 2.10.4 Chained Conditionals

A chained conditional is syntactic sugar for a sequence of conditional gotos. It looks like this:

```
if c1 do
  stmt1a;
  stmt1b;
elif c2 do
  stmt2a;
  stmt2b;
else
  stmt3a;
  stmt3b;
done
```

At least one statement in each group is required, the no-operation `;` can be used if there is nothing to do. A semicolon is not required after the *done*.

## 2.11 Loops

Loops statements are compound statements that make control go around in circles for a while, then exit at the end of the loop.

### 2.11.1 While loop

The simplest loop, repeatedly executes its body whilst its condition is true. If the condition is initially false, the body is not executed. On exit, the statement following the loop is executed.

```
var x = 10;
while x > 0 do
  println$ x;
  x = x - 1;
done
println$ "Done";
```

A semicolon is not required after the *done*. Make sure when writing while loops that the condition eventually becomes false, unless, of course, you intend an infinite loop.

### 2.11.2 For loop

For loops feature a control variable which is usually modified each iteration, until a terminal condition is met. The simplest for loop uses a slice:

```
for i in 0..<3 do
  println$ i;
done
```

Here, we print the variable *i*, which is initially 0, and takes on the values 1,2 as well before the loop terminates. The slice used indicates it is exclusive of the last value. An inclusive slice is illustrated here:

```
for i in 0..3 do
  println$ i;
done
```

and the loop iterations include the value 3. The values of a the slice start and slice end delimiters can be arbitrary expressions of type *int*. Slices can be empty if the end is lower than the start, in this case the loop body is not executed.

The control variable, *i* above, is automatically defined and goes out of scope at the end of the loop. It should not be modified during the iteration.

## 2.12 Arrays

In Felix, arrays are first class values.

### 2.12.1 Type

An array is given a type consisting of the base type and length.

```
int^4
```

is a the type of an array of 4 ints. Note, the *4* there is not an integer but a unitsum type.

### 2.12.2 Value

An array is given by a list of comma separated expressions:

```
var a :int^4 = 1,2,3,4;
```

### 2.12.3 Operations

#### Projection

The most fundamental operation is the application of a projection to extract the *n*'th component of an array. Components are numbered from 0 up.

```
var a :int^4 = 1,2,3,4;
for i in 0..<4 do
  println$ a.i;
done
```

The projection here is indicated by the *int i*. An expression can be used provided it is in bounds.

#### Length

The length of an array may be obtained with the *len* function. The value returned is of type *size* which can be converted to *int* as shown:

```
var x = 1,2,3,4;
var lx = x.len.int;
println$ lx; // 4
```

#### Value Iteration

A for loop may take an array argument. The control variable takes on all the values in the array starting with the first.

```
var x = 1, 2, 3, 4;
var sum = 0;
for v in x do
  sum = sum + v;
done
println$ sum;
```

## 2.13 Tuples

A tuple is a heterogeneous array. It is a sequence of values of any type.

### 2.13.1 Type

The type of a tuple is written as a product of the types of the components:

```
int * string * double
```

### 2.13.2 Values

A tuple value is written as a comma separated sequence:

```
var x : int * string * double = 1, "hello", 4.2;
```

If all the components have the same type, you get an array instead.

### 2.13.3 Projections

A tuple projection is like an array projection except that only literal integer index is allowed. This is so that the type is known. The indices are zero origin, as for arrays.

```
var x : int * string * double = 1, "hello", 4.2;
println$ x.1; // string
```

### Unit Tuple

There is a special tuple with no components. It is given the type *1* or *unit*. The value is written *()*.

## 2.14 Procedures

A sequence of statements can be wrapped into a named entity called a *procedure*. In addition, a procedure may accept an argument. The accepting variable is called a parameter.



### 2.14.1 Type

The type of a procedure with parameter type T is written

```
T -> 0
```

A procedure is a subroutine, it returns control, but it does not return a value. To be useful, a procedure must change the state of the program or its environment. This is called an effect.

Procedures in Felix are first class and can be used as values.

### 2.14.2 Definition

A procedure is defined like this:

```
proc doit (x:int) {
  println$ x;
  x = x + 1;
  println$ x;
}
```

A procedure may explicitly return control when it is finished.

```
proc maybe doit (x:int) {
  if x > 0 do
    println$ x;
    return;
  done
  x = -x
  println$ x;
}
```

If the procedure does not have a return statement at the end, one is implicitly inserted.

A procedure can have a unit argument:

```
proc hello () {
  println$ "Hello";
}
```

### 2.14.3 Invocation

A procedure is called with a call statement. The identifier *call* may be omitted. If the argument is unit, it also may be omitted.

```
proc hello () {
  println$ "Hello";
}
call hello ();
hello ();
hello;
```

## 2.15 Functions

Functions encapsulate calculations.

### 2.15.1 Type

The type of a function with a parameter type *D* returning a value type *C* is written

```
D -> C
```

### 2.15.2 Definition by Expression

A function can be defined by an expression:

```
fun square (x:int) : int => x * x;
```

or without the return type:

```
fun square (x:int) => x * x;
```

in which case it is deduced.

### 2.15.3 Definition by Statements

More complex functions can be defined by statements.

```
fun addUp(xs:int^4) : int = {  
  var sum = 0;  
  for x in xs do  
    sum = sum + x;  
  done  
  return sum;  
}
```

The return type can be elided:

```
fun addUp(xs:int^4) = {  
  var sum = 0;  
  for x in xs do  
    sum = sum + x;  
  done  
  return sum;  
}
```

### 2.15.4 No side effects

The effect of a function must be entirely captured in its returned value; that is, it may not have any side effects. This assumption is currently not checked, so you could write code like this:

```

var mutMe = 0;

fun addUp(xs:int^4) : int = {
  mutMe = 1; // bad!
  var sum = 0;
  for x in xs do
    sum = sum + x;
  done
  return sum;
}

```

However, this kind of usage may be useful from time to time, for example for debugging.

The lack of side effects in a function are used in optimizations, and the optimizations may have an effect on program behavior. For example, the following toy program takes the second projection (`. 1`) on a tuple involving three function calls. Since functions are assumed to have no side effects, the other function calls (`f` and `h`) are erased as their return values are never used.

```

fun f(x:int) = {
  println "hi from f!";
  return 2*x;
}
fun g(x:int) = {
  println "hi from g!";
  return 3*x;
}
fun h(x:int) = {
  println "hi from h!";
  return 4*x;
}

val res = (f 5, g 5, h 5) . 1;
println res;

```

The output of the program is just:

```

hi from g!
15

```

### 2.15.5 Purity

Functions can further be annotated to be pure or impure, but at the moment, the semantics of these are not defined and are not checked:

```

pure fun addUp(xs:int^4) : int = {
  // ...
}

// or

impure fun addUp(xs:int^4) : int = {
  // ...
}

```

## 2.16 Function Application

In an expression, a function  $f$  can be applied to an argument  $x$  like this:

```
f x
```

This is known as forward, or prefix, application, although it can also be considered, somewhat whimsically, that the gap between the  $f$  and the  $x$  is an infix operator known as *operator whitespace*.

Although this is the mathematical syntax for application, many programmers, may prefer to swap the order of the function and argument like this:

```
x.f
```

This is known as reverse application. Operator dot binds tighter than operator whitespace so this expression:

```
g x.f
```

is parsed as

```
g (x.f)
```

Both operator dot and operator whitespace are left associative.

Another application operator is stolen from Haskell:

```
h $ g $ h $ x;
```

Operator `$` binds more weakly than either dot or whitespace, and is right associative so the above parses as:

```
h (g (h x));
```

Finally, there is a shortcut for applying a function to the unit tuple `()`:

```
#f
```

which means the same as:

```
f ()
```

Operator hash `#` is a prefix operator that binds more tightly than the other application operators.

## 2.17 Pythagoras

The ancient Greek mathematician Pythagoras is famous for the invariant of right angle triangles:

$$h^2 = w^2 + a^2$$

where  $h$  is the hypotenuse,  $w$  is the width of the base, and  $a$  is the altitude.

We can calculate the hypotenuse in Felix like this:

```
fun hypot (w:double, a:double) : double =>
  sqrt (w^2 + a^2)
;

println$ hypot (3.0, 4.0);
```

The type *double* is a standard double precision floating point real number.

The *sqrt* function is in the library, and calculates the square root of a double precision number.

The operator  $^$  denotes exponentiation, in this case we are squaring, or multiplying the argument by itself twice, the literal 2 is a value of type *int*, a type of small integers.

Of course, the operator  $+$  is addition.

The *fun* binder is used here to define a function. Then we give the function name we want to use, in this case *hypot*.

Then, in paranthesis we give a comma separated list of parameter specifications. Each specification is the name of the parameter, followed by its type.

It is good practice, but not required, to follow the parameters with  $:$  and the return type of the function.

Then the  $=>$  symbol is used to begin the formula defining the function in terms of the parameters.

The function can be used by applying it to an argument of the correct type, in this case a pair, or tuple, of two numbers of type *double*.

The *println* is then called on the application using the application operator  $\$$ .



## 3.1 Record

A record is like a tuple, except the components are named:

```
var x = (a=1, b="hello", c=42.0);  
println$ x.b;
```

Actually, you can use a blank name, or leave a name out:

```
var x = (a=1, 42.0, n""="What?");
```

Note the use of the special identifier form `n""` in which the text of the identifier is zero length.

### 3.1.1 Duplicate Fields

Fields names in a record can be duplicated:

```
var x = (a=1, a=2, 32, 77);
```

In this case, when the field name is used to access a component it refers to the left most instance of the field. While this may seem like an unusual feature in isolation, it is needed to support polyrecords (row polymorphism).

There is a special case: if all the field names are blank, the the record is a tuple. So in fact tuples are just a special case of records.

### 3.1.2 Function Application

Earlier we saw examples of function application, but function application is implicitly performed on tuples and records:

```
fun f(x:int,y:double)
// accepts either of the following
f (1,2.1)
f (x=1,y=2.1)
```

The order doesn't matter if you use names, except for duplicates.

## 3.2 Classes

In Felix a class is used to provide a space for defining functions and procedures.

```
class X {

  // quadratic ax^2 + bx + c, solution:
  fun qplus (a:double, b:double, c:double) =>
    (-b + sqrt (sqr b + 4.0 * a * c)) / ( 2.0 * a)
  ;

  // square it
  private fun sqr(x:double) => x * x;
}
```

Notice the *qplus* function can call the *sqr* function even though it is defined later. Felix uses random access, or setwise, lookup, not linear lookup.

In order to use a function in a class, we can use explicit qualification:

```
println$ X::qplus(1.0, 2.0, 3.0);
```

Alternatively, we can open the class:

```
open X;
show (qplus 42);
```

However we cannot access the function *sqr*, because it is private to the class.

A class definition must be contained in a single file, it cannot be extended.

## 3.3 Generic Functions

Generic functions and procedures provide a simple definition.

```
// generic function
fun add3 (x,y,z) => x + y + z;

// used with different argument types
println$ add3 (1,2,3); // 6
println$ add3 (1.0,2.0,3.0); // 6.0
println$ add3 ('Hello',' ','World'); // Hello World
```

For each uses of a generic function, Felix makes a copy and adds the argument types. So the three calls above actually call these automatically generated functions:



```
fun add3 (x:int, y:int, z:int) => x + y + z;
fun add3 (x:double, y:double, z:double) => x + y + z;
fun add3 (x:string, y:string, z:string) => x + y + z;
```

Note that the rewritten functions are generated in the same scope as the generic function so any names used in the generic function refer to the names in the generic function's original scope.

## 3.4 Lists

Lists are a fundamental, polymorphic data type. Felix list is a singly linked, purely function data type. All the values in a list have the same type.

### 3.4.1 Creating a list.

A list can be created from an array:

```
var x = list (1,2,3,4);
```

A more compact notation is provided as well:

```
var x = ([1,2,3,4]);
```

### 3.4.2 Empty lists

For an empty list there are two notations:

```
var x = list[int] (); // empty list of int
var y = Empty[int];  // empty list of int
```

### 3.4.3 Displaying a list

A list can be converted to a human readable form with the *str* function, provided the values in the list can be converted as well:

```
println$ "A list is " + ([1,2,3,4]).str;
```

### 3.4.4 Concatenation

Lists can be concatenated with the *+* operator:

```
println$ list (1,2,3) + ([4,5,6]);
```

### 3.4.5 Length

The length of a list is found with the *len* function, the result is type *size*:

```
println$ ([1,2,3,4]).len; // 4
```

### 3.4.6 Prepending an element

A new element can be pushed on the front of a list with the `Cons` function or using the infix `!` operator, or even with `+`:

```
var a = ([2,3,4]);
var b = Cons (1, x);
var c = 1 ! a;
var d = 1 + a;
```

The lists *b*, *c* and *d* all share the same tail, the list *a*. This means the prepend operation is  $O(1)$ . It is safe because lists are immutable.

The use of `+` is not recommended because it is rather too heavily overloaded. In particular note:

```
1 + 2 + ([3,4]) // ([3,3,4])
1 + (2 + ([3,4]) // ([1,2,3,4])
```

because addition is left associative.

### 3.4.7 Pattern matching lists

Lists are typically decoded by a recursive function that does pattern matching:

```
proc show(x:list[int]) =>
  match x with
  | Empty => println$ "end"
  | head ! tail =>
    println$ "elt= " + head.str;
    show tail;
  endmatch
;
```

The text between the `|` and `=>` is called a pattern. To analyse a list, there are two cases: the list is empty, or, the list has a head element and a following tail. The procedure prints “end” if the list is empty, or the head element followed by the tail otherwise.

## 3.5 Option Type

Another useful type that requires pattern matching is the polymorphic option type *opt*. It can be used to capture a value, or specify there is none:

```
fun divide (x:int, y:int) =>
  if y == 0 then None[int]
  else Some (x/y)
endif
;
```

### 3.5.1 Pattern matching optional values

This is done like:

```
printopt (x: opt[int]) {
  match x with
  | Some v => println$ "Result is " + v.str;
  | None => println$ "No result";
  endmatch;
}
```

## 3.6 Varray

A varray is a variable length array with a construction time bounded maximum length. Unlike ordinary arrays, varrays are mutable and passed by reference. Underneath a varray is just a pointer.

### 3.6.1 Empty Varray

An empty varray can be constructed by giving the bound, the bound must be of type size:

```
var x = varray[int] 42.size;
```

The type of the varray must be specified in this form.

### 3.6.2 Construction from container

A varray can be constructed from an ordinary array, another varray, list, or string, with or without a bound specified:

```
var v4 = varray (1,2,3,4);           // varray, length 4, maxlen 4
var v8 = varray (v4, 8.size);        // length 4, maxlen 8
var y8 = varray (v8);                // length 4, maxlen 8
var y12 = varray (v8, 12.size);       // length 4, maxlen 12
var z4 = varray ([1,2,3,4]);          // length 4, maxlen 4
var z4 = varray ([1,2,3,4], 12.size); // length 4, maxlen 12
var s12 = varray "Hello World";       // trailing NUL included!
```

### 3.6.3 Construction from default value

A varray can also be built to given size and filled with a default value:

```
var v100 = varray(100.size, char 0); // buffer
```

### 3.6.4 Length

The length of a varray is given by the *len* function, the bound is given by the *maxlen* function:

```
var x = varray 42.size);
println$ "Length zero=" + x.len.str + ", max=" + x.maxlen.str;
```

### 3.6.5 Extend and Contract

A new element can be pushed at the end of a varray with the `push_back` procedure, provided the resulting array doesn't exceed its `maxlen` bound. Similarly an element can be removed from the end, provided the array isn't empty:

```
var x = varray[int] 42.size;
x.push_back 16; // length now 1
x.pop_back;    // remove last element
```

### 3.6.6 Insert at position

An element can be inserted at a given position, provided the position does not exceed the current length, and is greater than or equal to the `maxlen`:

```
var x = varray[int] 42.size;
insert (x, 0.size, 42);
insert (x, 0.size, 41);
insert (x, 2.size, 42);
```

### 3.6.7 Erase elements

Elements can be erased by giving the position to erase, or, an inclusive range:

```
var x = varray (1,2,3,4,5,6);
erase (x, 2);
erase (x, 2, 20);
```

This procedure cannot fail. Attempts to erase off the ends of the array are simply ignored.

### 3.6.8 Get an element

An element can be fetched with the `get` function, provided the index is in range:

```
var x = varray (1,2,3,4,5,6);
println$ get (x, 3.size); // 4
println$ x.3;             // 4
```

The last form allows any integer type to index a varray.

### 3.6.9 Set an element

An element can be modified with the `set` procedure:

```
var x = varray (1,2,3,4,5,6);
set (x, 3.size, 99); // 4 changed to 99
```

## 3.7 Objects, Values and Pointers

Felix supports both functional and imperative programming styles. The key bridge between these models is the pointer.

An array in Felix is an immutable value, it cannot be modified as a value. However an array can be stored in a variable, which is the name of an object. An object has two significant meanings: it names a storage location, and also refers to the value located in that store.

In Felix, the name of a variable denotes the stored value, whilst the so-called address-of operator applied to the variable name denotes the location:

```
var x = 1,2,3,4; //1: x is an array value, and ..
var px = &x;    //2: it also denotes addressable store
var y = *px;    //3: y is a copy of x
px <- 5,6,7,8;  //4: x now stores a new array
```

This code illustrates how to get the address of a variable on line 2, to fetch the value at the address in line 3, and to modify the value at the address, in line 4.

The prefix symbol `&` is sometimes called the address-of operator, however it is not an operator! Rather, it is just a way to specify that we want the address of the store a variable denotes, rather than the value stored there, which is denoted by the plain variable name.

The address is a Felix data type called a pointer type. If a variable stored a value of type `T`, the pointer is denoted by type `&T`.

In line 3 we use the so-called dereference operator, prefix `*`, to denote the value held in the store to which a pointer points. Dereference is a real operator.

In line 4, we use the infix left arrow operators, which is called *store-at*, it is used to store right hand argument value in the location denoted by the left hand pointer value.

## 3.8 The new operator

Felix also provide the prefix *new* operator which copies a value onto the heap and returns pointer to it.

```
var px = new 42;
var x = *px; // x is 42
px <- 43;    // px now points to 42
```

This is another way to get a pointer to an object, which allows the value stored to be replaced or modified.

## 3.9 Pointer projections

All product types including arrays, tuples, records, and structs provide value projections for fetching parts of the value, the parts are called components:

```
var ax = 1,2,3,4;           // array
var ax1 = x.1;              // apply projection 1 to get value 2

var tx = 1, "hello", 42.0;  // tuple
var tx1 = tx.1;             // apply projection 1 to get value "hello"

var rx = (a=1, b="hello", c=42.0); // record
var rx1 = rx.b;             // apply projection b to get value "hello"

struct X {
  a:int;
```

(continues on next page)

(continued from previous page)

```

    b:string;
    c:double;
}
var sx = X (1, "hello", 42.0);      // struct
var sx1 = sx.b;                    // apply projection b to get value "hello"

```

Arrays and tuples have numbered components, and thus are accessed by numbered projections, records and structs have named components and thus values are accessed by named projections.

Although the indicators here are numbers and names, value projections are first class functions. The functions and their types, respectively, are:

```

proj 1: int^4 -> int
proj 1: int * string * double -> string
b: (a:int, b:string, c:double) -> string
b: X -> string

```

These are value projections. To store a value in a component of a product type, we must first obtain a pointer to the store in which it is located, and then we can apply a *pointer projection* to it, to obtain a pointer to the component's store. Then we can use the store-at procedure to set just that component, leaving the rest of the product alone:

```

&ax . 1 <- 42;           // array
&tx . 1 <- "world";       // tuple
&tx . b <- "world";       // record
&sx . b <- "world";       // struct

```

In each case we use the same projection index, a number or a name, as for a value projection, but the projections are overloaded so they work on pointers too. These pointer projections are first class functions, here are their types, respectively:

```

proj 1: &(int^4) -> &int
proj 1: &(int * string * double) -> &string
b: &(a:int, b:string, c:double) -> &string
b: &X -> &string

```

What is critical to observe is that pointers are values, and the pointer projections are first class, purely functional functions. Unlike C and C++ there is no concept of lvalues or references. The store-at operator is a procedure, and so it is used in imperative code, but the calculations to decide where to store are purely functional.

The programmer should note that C address arithmetic is also purely functional, however, C does not have any well typed way to calculate components of products other than arrays: you do the calculations only by using the *offsetof* macro and casts.

C++ has pointers to members, but the calculus is incomplete, they cannot be added together!

In Felix, projections are functions so adding component offsets in products is, trivially, just function composition!

---

Felix 103: Polymorphism

---

An introduction to parametric polymorphism.

## 4.1 Polymorphic Functions

Felix allows function to be polymorphic. This means you can write a function that works, in part, for any type.

```
fun swap[A,B] (x:A, y:B) : B * A => y, x;
```

This is called parametric polymorphism. The names  $A$  and  $B$  are called type variables. The above function will work for any actual types:

```
println$ swap[int,string] (42, "Hello");
```

Here, the specific types used we given explicitly. This is not required if the types can be deduced from the arguments of the application:

```
println$ swap(42, "Hello");
```

Here,  $A$  must be *int* because parameter  $x$  has type  $A$ , and the argument 42 has type *int*. Similarly,  $B$  must be *string* because “hello” has type *string*.

## 4.2 Higher Order Functions

In Felix, functions are first class which means a function can be used as a value. A function which accepts another function as a parameter, or returns a function as a result, is called a *higher order function*, abbreviated to *HOF*.

Here’s an example:

```
fun twice (x:int):int => x + x;
fun thrice (x:int):int => x + x + x;

fun paired (f:int->int, a:int, b:int) =>
  f a, f b
;

println$ paired (twice,2,3);
println$ paired (thrice,2,3);
```

Here, the function *twice* is passed as an argument to *paired*, binding to the parameter *f*, which is then applies to the arguments *a* and *b* to obtain the final result.

Then, we do it again, this time passing *thrice* instead.



### 5.1 Getting Started With the Felix GUI

Felix comes with a library to create platform independent graphical user interfaces. It uses the Simple Direct Media Layer, version 2, SDL2 system with add-ons to do this.

SDL2 runs on Linux, OSX, Windows, iOS and Android and is designed for implementation of portable games.

#### 5.1.1 Installation

For the Felix GUI to work, *development versions* of the following components must be installed:

```
SDL2
SDL2_image
SDL2_ttf
SDL2_gfx
```

On Linux using apt package manager do this:

```
sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2_image-dev
sudo apt-get install libsdl2_ttf-dev
sudo apt-get install libsdl2_gfx-dev
```

Felix already contains database entries for these packages, but at the time of writing this tutorial, the libraries are expected to be in */usr/local* which is where you would put them if you built them yourself.

However Debian filesystem layout standards used by Linux OS such as Ubuntu that use *apt* package manager put components in */usr/* instead. So unfortunately you will have to modify the database by hand by editing these files

```
build/release/host/config/sdl2.fpc
build/release/host/config/sdl2_image.fpc
build/release/host/config/sdl2_ttf.fpc
```

replacing */usr/local* with just */usr*. To preserve these modifications across upgrades, you should also copy the files:

```
cp build/release/host/config/sdl2.fpc $HOME/.felix/config
cp build/release/host/config/sdl2_image.fpc $HOME/.felix/config
cp build/release/host/config/sdl2_ttf.fpc $HOME/.felix/config
```

I hope this problem can be reduced in future versions, but it is likely to be an issue for some time because most developers will have a libraries installed in both places, depending on whether they're using a package manager to install them, or building from source.

To check your installation works, do this:

```
make tutopt-check
```

and a series of tests will run which use the Felix GUI and SDL2.

## 5.2 GUI Basics

We will now show how to do basic GUI programming. The first thing we want to do is open a window!

```
include "gui/__init__";
open FlxGui;

println$ "Basic Window Test";
FlxGui::init();

var w = create_resizable_window("Felix:gui_01_window_01",100,100,400,400);
w.add$ mk_drawable FlxGuiSurface::clear lightgrey;
w.update();
w.show();

sleep(15.0);
```

The Felix gui is not included by default, so we have to first include the library with

```
include "gui/__init__";
```

Although this makes the library available, we would have to prefix the names of all functions in the library with `FlxGui::` to use them. Since programmers hate typing stuff, we will open the library, so the functions are all in the current scope and can be used without the prefix.

```
open FlxGui;
```

Next, we will print a diagnostic to standard output so we know which program is running, and then initialise library. Initialisation is a requirement imposed by SDL, which has a lot of work to do on some platforms to connect to the GUI devices such as the screen, touch (haptic) inputs, joysticks, mice, keyboards, audio, and other multi-media hardware.

```
println$ "Basic Window Test";
FlxGui::init();
```

Now it is time for the fun! We will create a resizable window:

```
var w = create_resizable_window("Felix:gui_01_window_01",100,100,400,400);
```

The first parameter is the title of the window, which should appear on the titlebar (it doesn't on OSX! Because there is no pause to accept input).

The next four parameters describe the window geometry. The first two are the x and y coordinates. The SDL coordinate system puts the origin in the top left corner, and x increases down the screen.

The unit of measurement is approximately the pixel. I say approximately because on a Mac with a Retina display, each pixel is often four display elements on the screen. To confuse the issue, the Mac can do hardware scaling. You'll just have to experiment!

The second two values are the width and height of the window's client area, this does not include the title bar. However the x and y coordinates are the top left corner of the whole window including the title bar!

What we have created is a data structure representing the window. The next thing we want to do is put some coloured pixels in it.

```
w.add$ mk_drawable FlxGuiSurface::clear lightgrey;
```

This is an example of a fundamental operation, to add to a windows display surface, the commands for drawing something.

The `w.add` method adds a *drawable* to a list of drawables kept for window `w`.

The `mk_drawable` method is a function which constructs a drawable object. Its first argument is the actual drawing command, `FlxGuiSurface::clear` which clears a whole surface. The second argument is an argument to that command, in this case `lightgrey`, which tells the clearing command what colour to write on the surface it is clearing.

We have not actually drawn on the window at this point. What we have done is packaged up the drawing instructions, and *scheduled* them for drawing later.

To actually draw, we do this:

```
w.update();
```

Now we have drawn the objects we scheduled to be drawn on the systems internal representation of the window's surface but still, nothing appears on the screen!

This is because the window has not been shown yet. We've been drawing on it whilst it was invisible. So we now make it visible:

```
w.show();
```

Finally, we want the window to hang around for 15 seconds so you can admire your fine art work.

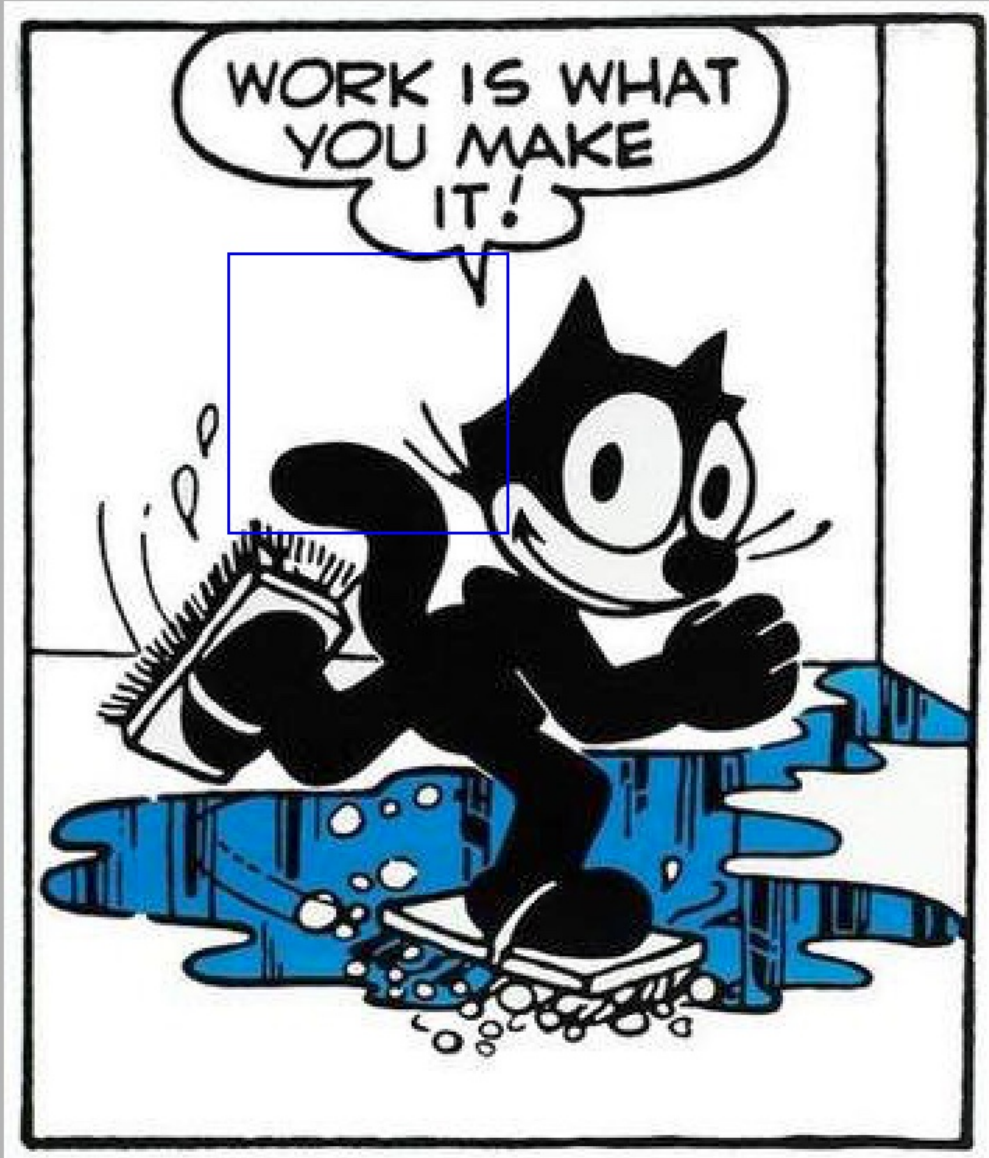
```
sleep(15.0);
```

This causes the program to sleep for 15 seconds. The argument is a double precision floating point number representing a delay in seconds. The decimal point is mandatory and trailing zero is mandatory!

## 5.3 Putting Stuff in the Window

Now we have created a window, we want to put stuff on it!

## Basic Drawing Test



Here's how:

```
include "gui/__init__";  
open FlxGui;  
  
println$ "Basic Drawing Test";  
FlxGui::init();
```

(continues on next page)

(continued from previous page)

```

var w = create_resizable_window("Felix:gui_03_draw_01",100,100,400,600);
w.add$ mk_drawable FlxGui::clear lightgrey;

var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;
w.add$ mk_drawable FlxGui::write (10,10,font,black,"Basic Drawing Test");

fun / (x:string, y:string) => Filename::join (x,y);
var imgfile = #Config::std_config.FLX_SHARE_DIR / "src" / "web" / "images" /
  ↪ "FelixWork.jpg";

var ppic : surface_t = surface (IMG_Load imgfile.cstr);

w.add$ mk_drawable blit (20,20, ppic.get_sdl_surface ());

w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,200,110);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,210,200,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,100,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 200,110,200,210);

w.update();
w.show();

Faio::sleep(15.0);

```

Here, the program is as before, except we now do three basic ways to draw on a window.

### 5.3.1 Text

First, we want to be able to put ordinary text on the window. To do that, we have to first create a font to write the text with:

```

var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;

```

The first line gets the name of a default sans serif font file which is packaged with Felix so you don't have to figure out where all the fonts on your system are.

The second line actually creates the font from the file at a particular size, in this case 12 point. The size is a conventional printers measure. You'll just have to get used to what it means!

The third line helps tell how big the font is. We retrieve from the font the distance in pixels we should allow between lines, for readable text. Anything less and the characters would bump into each other.

Now we have a font, we schedule drawing some text on the window:

```

w.add$ mk_drawable FlxGui::write (10,10,font,black,"Basic Drawing Test");

```

This is our usual machinery for adding a drawable object to the windows list of drawables, to be drawn when we say to *update*. The drawing function is `FlxGui::write`. Notice we used the fully qualified name of the function, to avoid confusion with other functions named *write*.

The argument to write is the x and y coordinates of the initial base point, the font to use, the colour to write in, and the actual text to write.

Text is always written with respect to a base point. The base point is origin of the first character which is approximately the left edge of the character, and the point at which an invisible underline would occur: in other words, under the main body of the character, but on top of any descender that letter like *g* may have.

The exact location is font dependent. Font rendering is an arcane art, not an exact science so you will have to practice to get text to appear where you it has the correct visual significance.

### 5.3.2 Picture

Now we are going to put a picture in the window. The image is a JPEG image, and is supplied for testing purposed in Felix at a known location.

First we define a little helper function:

```
fun / (x:string, y:string) => Filename::join (x,y);
```

What this says is that when we try to divide one string by another string, we actually mean to join the strings together using `Filename::join` which is a standard function which sticks a `/` character between strings on unix platforms, and a slosh on Windows.

The file is here:

```
var imgfile = #Config::std_config.FLX_SHARE_DIR / "src" / "web" / "images" /  
↪ "FelixWork.jpg";
```

The prefix of this code finds the share subdirectory of the Felix installation, which contains the picture we went in the images subdirectory of the web subdirectory of the src subdirectory.

Now to schedule the drawing we do this:

```
var ppic : surface_t = surface (IMG_Load imgfile.cstr);  
w.add$ mk_drawable blit (20,20, ppic.get_sdl_surface ());
```

The first line loads the image file into memoy using a low level primitive from `SDL2_image`. That primitve requires a C char pointer, not a C++ string, which is what Felix uses, so we use `cstr` to convert. Then the *surface* function translates the loaded file into an Felix surface object.

In the second line we add the drawable to the window based on the `blit` function. This copies one surface to another. We copy the image surface to the window surface at position 20,20 in the window, and use the `get_sdl_surface()` method to translate the Felix surface object into a lower level SDL surface.

Its all a bit mysterious, so you just have to so some things by copying the patterns that work.

### 5.3.3 Lines

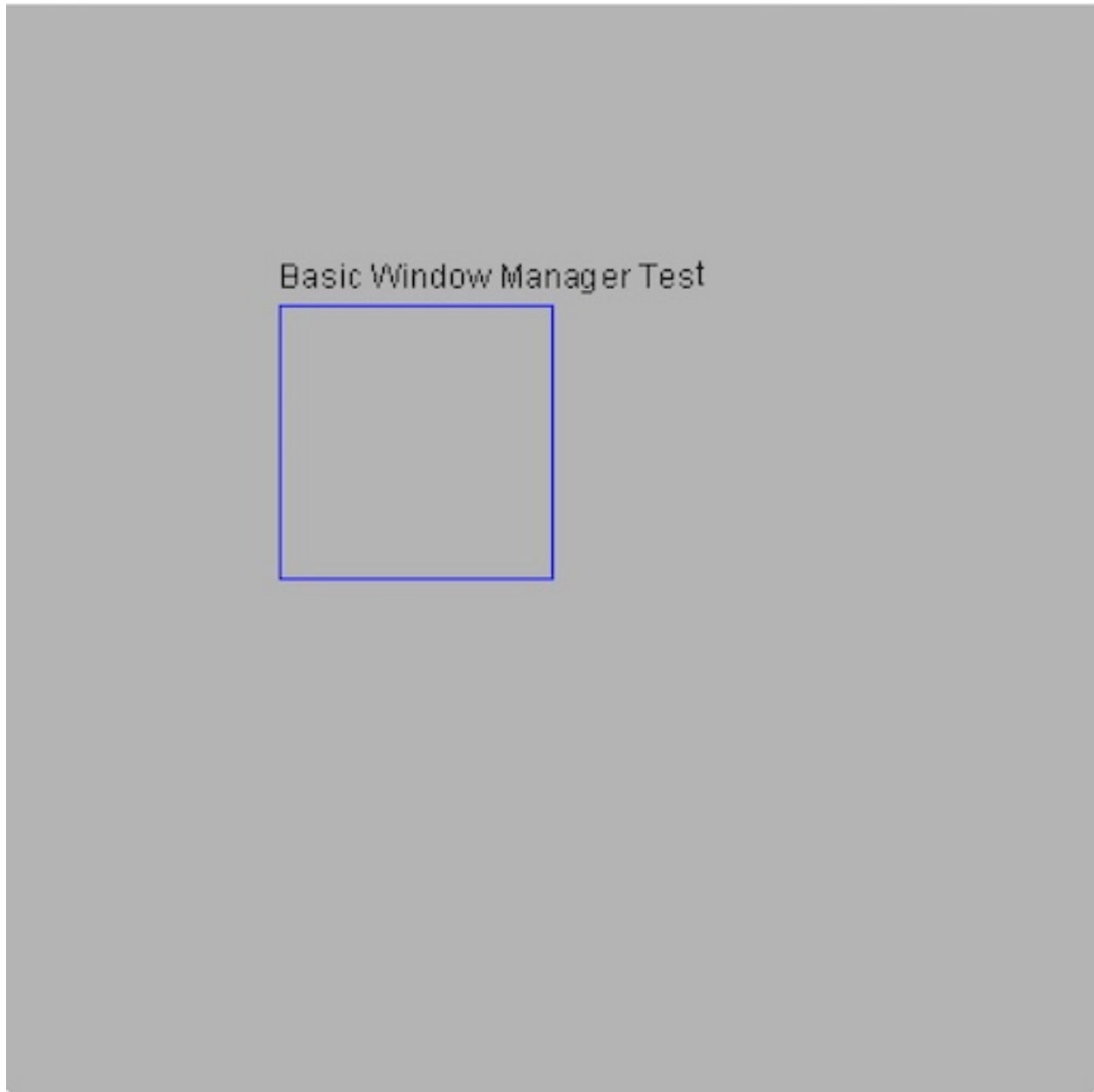
Finally, we draw a blue rectangle on top of the picture. I'm sure you can figure out how that works!

## 5.4 Using An Event Handler

So far we have just done some drawing. But now, we want to respond interactively to user input. To do this, we need to use an *event handler*.

### 5.4.1 The initial window

Lets start by making a window that looks like this:



which we do with this code as usual:

```
include "gui/__init__";
open FlxGui;
FlxGui::init();

var w = create_resizable_window("Felix:gui_04_wm_01",100,100,400,400);
w.add$ mk_drawable FlxGui::clear lightgrey;
```

(continues on next page)

(continued from previous page)

```

var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;

w.add$ mk_drawable FlxGui::write (100,100,font,black,"Basic Event Handler Test");
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,200,110);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,210,200,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 100,110,100,210);
w.add$ mk_drawable draw_line (RGB(0,0,255), 200,110,200,210);

w.update();
w.show();

```

## 5.4.2 The event handler

Now, the next thing is we are going to make a *chip* which can handle events:

```

chip event_displayer (w:window_t)
connector pins
  pin inevent : %<event_t
  pin quit: %>int
{
  while true do
    var e = read pins.inevent;
    var s =
      match e with
      | WINDOWEVENT we =>
        we.type.SDL_EventType.str + ": " +
        we.event.SDL_WindowEventID.str +
        " wid=" + we.windowID.str
      | MOUSEMOTION me =>
        me.type.SDL_EventType.str
      | _ => ""
    ;
    var linepos = 100 + 4 * lineskip;
    if s != "" do
      println$ s;
      var r = rect_t (100,linepos - 2*lineskip,300,4*lineskip);
      w.remove "evt";
      w.add$ mk_drawable "evt" 100u32 fill (r, green);
      w.add$ mk_drawable "evt" 100u32 FlxGui::write (100,linepos,font,black,"EVENT:
↪"+ s);
      w.update();
    done
  done
}

```

We are using a powerful new idiom: fibrated programming. What you see is a special kind of routine called a *coroutine*. Lets see what it does.

First, the interface tells us that it displays events on window *w*. Now our chip has a connector named *pins*, and on that connector, we have two pins named *inevent* and *quit*.

The pin *inevent* is an input pin for data of type *event\_t* whilst the pin *quit* is an output pin for an int. I can tell the direction of the pin from the channel type: %< is for input, and %> is for output. The type of data the pin handles



comes next.

Now let's look at the code. We can see immediately this chip runs in an infinite loop. It starts off by reading an event from the *inevent* pin.

Next, we analyse the event, to see what it is, using a pattern match. There are two kinds of event we're interested in: a *WINDOWEVENT* and a *MOUSEMOTION*.

For now, the weird code for these events just converts some of the event information into a string *s* we can display on the window, let's not worry about exactly what it means (you'll see, when you try it!).

Now the next bit calculates the position inside the box we drew to display the string, then, if the event description *s* is not the null string, we print the string to standard output.

Now we calculate a bounding rectangle for the string. It's not very accurate!

Now comes the fun bit! The next thing we do is *remove* all the drawables from the window tagged with the string "evt". Then we add two drawables, the first one fills our bounding rectangle with green, and the second writes some text. Then we update the window.

Now what is that magical *100u32* you ask? The answer is, this is the *z* coordinate of the drawing operation, which is a 32 bit unsigned integer. When Felix is drawing on a surface, it draws at the smallest *z* coordinate first. Then it draws at the next smallest, and so on. At any particular *z* coordinate, it draws in the order you add the drawable to the list of drawables.

By default, drawing occurs at *z=0u32*. So why are we specifying a *z* coordinate? The answer is: the background of the window was drawn at *z=0*. It was not given a tag, so it has the default tag "". Importantly, we did not remove drawables with that tag, so the background drawable is still in the drawable list.

The thing is, we want to draw *on top* of the background, so we have to ensure we draw at a higher *z* coordinate.

### 5.4.3 The Mainline

Now, as promised, it is time to install our event handler:

```
begin
  var qin,qout = mk_ioschannel_pair[int]();
  device windisp = event_displayer w;
  circuit
    connect windisp.inevent, event_source.src
    wire qout to windisp.quit
  endcircuit

  C_hack::ignore(read qin);
  println$ "QUIT EVENT";
end
```

The *begin* and *end* here are important for reasons that will be explained later, for the moment it suffices to know you need to do this to ensure the channels we create become inaccessible when you click to quit, so that the program actually terminates.

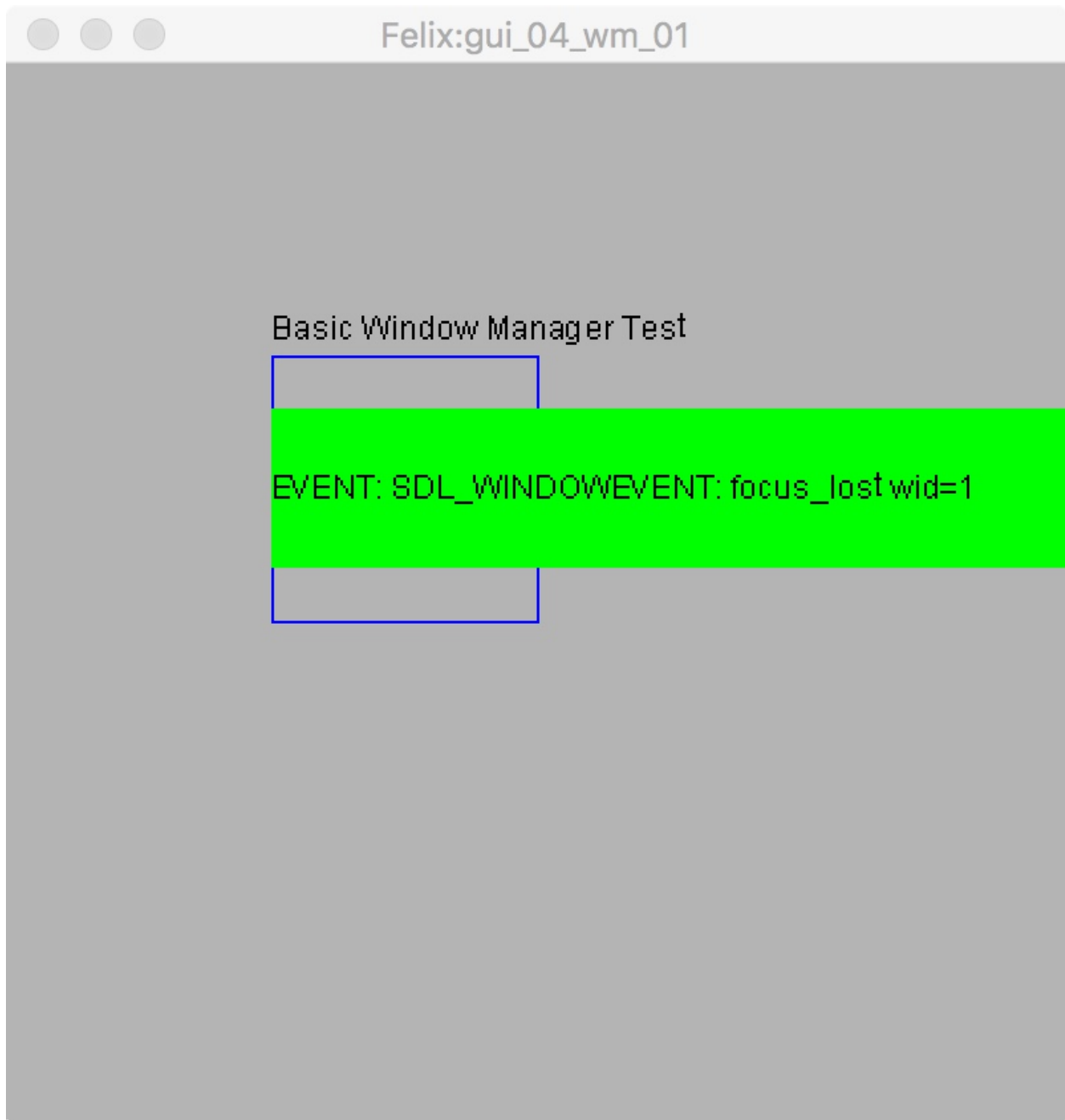
The first thing we do is make a single synchronous channel of type *int* which has two end points: *qin* and *qout*. The first one, *qin* is a read channel, and the second one, *qout* is a write channel.

Next, we make a *device* called *windisp* by applying our event handler function to the window it is to operate on.

Then we build a circuit by connecting the event handler to an event source, and wiring up the output end of the quit channel to the event handler as well. Our circuit begins running immediately.

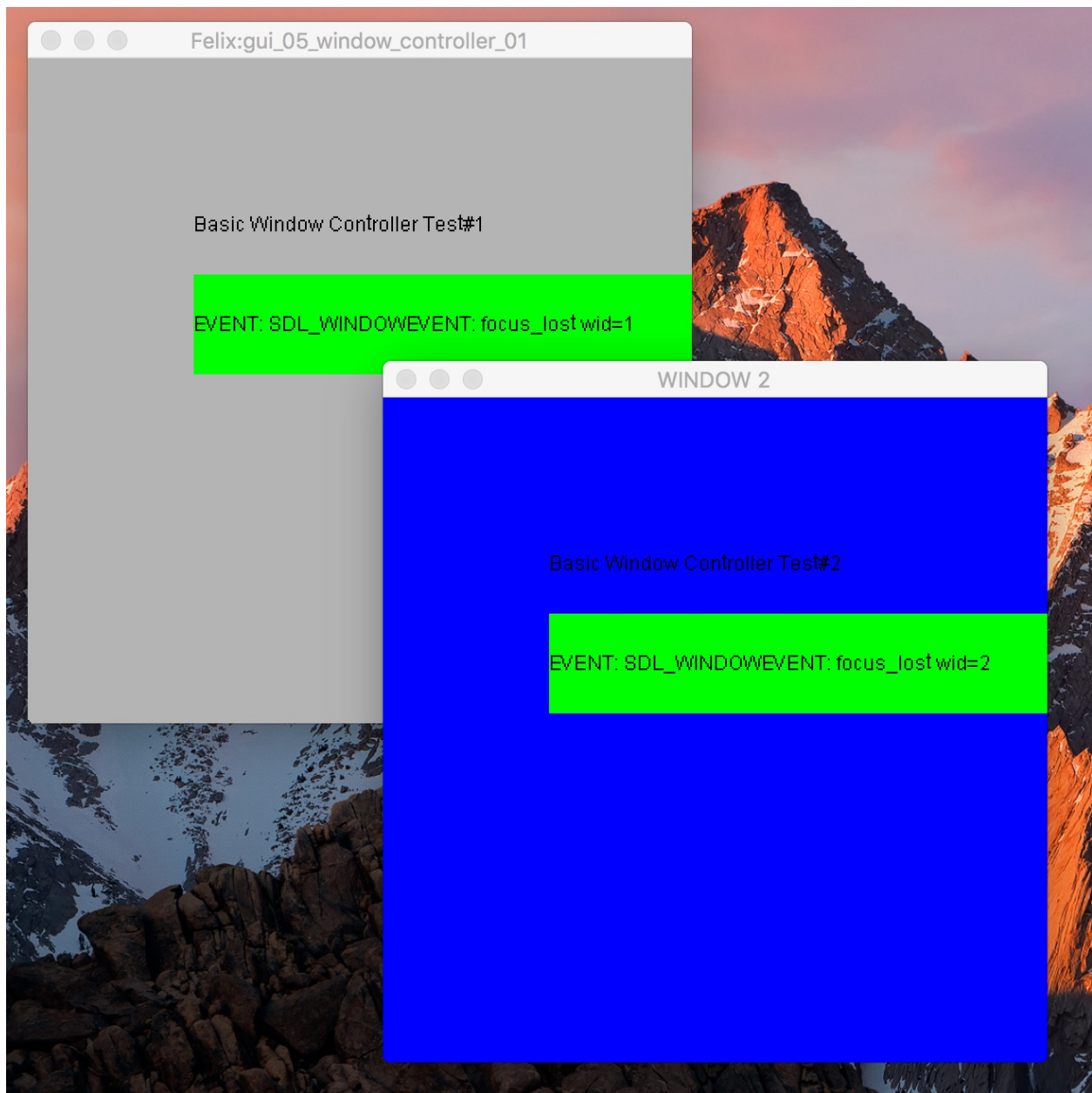
Now we wait until the user clicks to close the window, or presses the quit key. On a Mac, Apple-Q is the quit key. We use *C\_hack::ignore* because we don't care what the quit reason is.

You should see something like this:



## 5.5 Using a Window Manager

A *window manager* is a component that automates distribution of message such as mouse clicks and key presses to one of several event handlers.



### 5.5.1 The initial windows

First our usual setup:

```
include "gui/__init__";
open FlxGui;

println$ "Basic Window Controller Test";
FlxGui::init();

var font_name = dflt_sans_serif_font();
var font : font_t = get_font(font_name, 12);
var lineskip = get_lineskip font;
```

Now, we make two similar windows, at different locations but different titles.

```
var w1 = create_resizable_window("Felix:gui_05_window_controller_01",100,100,400,400);
w1.add$ mk_drawable FlxGui::clear lightgrey;
w1.add$ mk_drawable FlxGui::write (100,100,font,black,"Basic Window Controller Test#1
↪");
w1.show();
w1.update();

var w2 = create_resizable_window("WINDOW 2",400,100,400,400);
w2.add$ mk_drawable FlxGui::clear blue;
w2.add$ mk_drawable FlxGui::write (100,100,font,black,"Basic Window Controller Test#2
↪");
w2.show();
w2.update();
```

## 5.5.2 The Event handler

The same as before!

```
// make an event handler for our window
chip ehandler
  (var w>window_t)
connector pins
  pin input : %<event_t
{
  // get a first event from the window manager
  var e: event_t = read pins.input;
  // while the event isn't a quit event ..
  while e.window.event.SDL_WindowEventID != SDL_WINDOWEVENT_CLOSE do
    // print a diagnostic
    var s =
      match e with
      | WINDOWEVENT we =>
        we.type.SDL_EventType.str + ": " + we.event.SDL_WindowEventID.str + " wid=" + ↪
↪we.windowID.str
      | _ =>
        e.type.SDL_EventType.str
    ;
    var linepos = 100 + 4 * lineskip;
    if s != "" do
      println$ s;
      var r = rect_t (100,linepos - 2*lineskip,300,4*lineskip);
      w.add$ mk_drawable fill (r, green);
      w.add$ mk_drawable FlxGui::write (100,linepos,font,black,"EVENT: "+ s);
      w.update();
    done
    // get another event
    e= read pins.input;
  done

  // we must have got a quit ..
  println$ "++++++CLOSE EVENT";
}
```

### 5.5.3 The Window manager

Noe for the fun bit. First, our mainline creates a window manager object:

```
begin
  //create a window manager
  var wm = window_manager();
```

Now, we create two window controllers. There will be clients of the window manager.

```
// create a window controller for our window
var eh1 = ehandler w1;
var wc1 = window_controller (w1, eh1);
var eh2 = ehandler w2;
var wc2 = window_controller (w2, eh2);
```

Note that in this case the same event handler is bound to two distinct windows, and then a window controller is bound to them, as well as the window (again!)

Next, we simply add the window controller clients to the window manager.

```
// attach controller to window manager
var wno1 = wm.add_window wc1;
println$ "Window number " + wno1.str;

var wno2 = wm.add_window wc2;
println$ "Window number " + wno2.str;
```

When we do this, we get back a window number, assigned by the window manager, so we can refer to the windows in a way the window manager understands (although we're not doing that here).

Finally:

```
wm.run_with_timeout 10.0;
println$ "gui05 quitting";
end
```

we just run the window manager, in this case with a timeout because its a demo.