Lazarus IDE Tools/es

From Free Pascal wiki

Contents

Descripción general

El IDE de Lazarus utiliza una librería llamada ***codetools*** para analizar y editar el código fuente pascal. Estas herramientas ofrece funciones para completar código, extraer, mover, insertar y embellecer el código pascal. Utilizarlas permite ahorrar mucho tiempo y evitar duplicar esfuerzos. Son configurables y cada una de las funciones es accesible mediante un atajo (ver Editor Options).

Al trabajar únicamente con código fuente pascal (de FPC, Delphi y Kylix) no necesita unidades compiladas ni instalar el compilador Borland. Se puede editar código Delphi y FPC a la vez. Incluso se puede trabajar con varias versiones de Delphi y FPC al tiempo. Todo lo anterior hace la conversión de código Delphi mucho más fácil.

Tabla resumen de los atajos

Buscar Declaración	Ctrl +Click o Alt + ↑ (Salta a la declaración de la variable)
Salto a Procedimiento	Ctrl + ⇧ Shift + ↑ (Salta entre la definición y el cuerpo de procedimientos y funciones)
Plantillas de Código	Ctrl + J
Edición sincronizada	Ctrl + J (cuándo hay texto seleccionado)
Completar Código	Ctrl + ⇧ Shift + C (Completar Clase)
Completar Identificador	Ctrl + space
Completar Palabra	Ctrl + W
Sugerencia de Parámetros	Ctrl + ⇧ Shift + space
Búsqueda incremental	Ctrl + E

Salto a Procedimiento

Para ir desde el cuerpo de un procedimiento (o función) al lugar donde está su declaración (procedure Nombre;), o viceversa se utiliza la combinación de teclas Ctrl + ⇧ Shift + ↑.

Por ejemplo:

```
interface
procedure HacerAlgo; // declaración del procedimiento
 //...
implementation
 //...
procedure HacerAlgo; // Cuerpo del procedimiento
begin
 //...
end;
```

Si el cursor está en cualquier parte del cuerpo del procedimiento y se pulsan las teclas Ctrl+Shift+Up, el cursor salta a la declaración. Volviendo a pulsar Ctrl + ⇧ Shift + ↑ se regresa al cuerpo, y el cursor se sitúa al principio de la primera línea trás el *begin*.

Esto funciona con procedimientos y funciones, sean o no miembros de una clase.

Nota: 'Salto a Procedimiento' busca el procedimiento (o función) con el mismo nombre y lista de parámetros. Si no existe coincidencia exacta, salta al mejor candidato y situa el cursor en la primera diferencia que encuentra. (Delphi solo busca la coincidencia exacta, en D7 al menos).

Por ejemplo una función con diferente lista de parámetros:

```
interface

function HacerAlgo(p: char); // declaración de la función
```

```
implementation

function HacerAlgo(p: string); // cuerpo de la función
begin
end;
```

El salto desde la definición al cuerpo posicionará el cursor delante de la palabra clave *string*. Esto puede sernos útil para renombrar métodos y/o cambiar sus parámetros.

Por ejemplo:
Remombramos 'HacerAlgo' a 'Hazlo':

```
interface

procedure Hazlo; // declaración del procedimiento

implementation

procedure HacerAlgo; // Cuerpo del procedimiento
begin
end;
```

Ahora saltamos desde el redefinido *Hazlo* al cuerpo. El IDE buscará un cuerpo que se corresponda, al no encontrarlo buscará un candidato plausible. Trás cambiar el nombre existe un único procedimiento sin su declaración (HacerAlgo), por lo que saltará a él, situando el cursor justo delante de "HacerAlgo". Ahora solo resta escribir el nuevo nombre. También funciona si hemos cambiado algo en la lista de parámetros.

Archivos de inclusión

El contenido de los archivos de inclusión es insertado en el código fuente con las directivas de compilación {$I NombreArchivo} o {$INCLUDE NombreArchivo}. Lazarus y FPC utiliza una gran cantidad de ellos para reducir la redundancia y eliminar estructuras ilegibles con {$IFDEF} que dan soporte a múltiples plataformas.

El IDE de Lazarus da soporte completo a los archivos de inclusión, al contrario que Delphi. Se puede saltar desde la declaración de un método en un .pas a su cuerpo en un archivo de inclusión. Todas las funciones de *codetools* consideran los archivos de inclusión un ámbito especial, tal cómo hace *Completar Código*.

Por ejemplo: Cuándo *Completar Código* añade el cuerpo de un nuevo método tras el cuerpo de otro, mantiene ambos en el mismo archivo. Así podemos poner la implementación completa de la clase en el archivo de inclusión, tal cómo se hace en la LCL para todos sus componentes.

Pero aquí hay una trampa para novatos: Si se abre un archivo de inclusión por primera vez y se usa *Procedure Jump* o *Buscar Declaración* se obtiene un eror. El IDE no sabe la unidad a la que pertenece el archivo de inclusión; hay que abrir la unidad previamente pra que la cosa funcione.

Tan pronto como el IDE analiza la unidad, se evalúan las directivas de inclusión y el IDE toma nota de las relaciones entre los archivos.Al cerrar o guardar el proyecto esta información se guarda en el archivo $(LazarusDir)/includelinks.xml. La próxima vez que abramos el archivo de inclusión y realicemos un *Procedure Jump* o *Buscar Declaración* el IDE hará uso de esta información y las funciones trabajarán correctamente.

Este mecanismo tiene límites, por supuesto. Algunos archivos están incluidos dos o más veces, por ejemplo, $(LazarusDir)/lcl/include/winapih.inc.

Los saltos a los cuerpos desde las definiciones de procedimientos o métodos desde este fichero dependerán del contexto actual. Si se está trabajando con lcl/lclintf.pp el IDE saltará a winapi.inc. Si se está trabajando con lcl/interfacebase.pp, el salto se realizará a lcl/include/interfacebase.inc (u otro archivo de inclusión). Si se está trabajando con los dos, habrá una situación ambigua. ;)

Plantillas de código

Las *Plantillas de código* permiten convertir un identificador en un texto o en un fragmento de código.

El atajo, por defecto, para las *Plantillas de código* es Ctrl+J. Se escribe un identificador, se pulsan las teclas Ctrl+J y el identificador es sustituido por el texto definido para el identificador. Las *Plantillas de código* se definen en *Entorno -> Plantillas de Código...*.

Ejemplo: Escribimos el identificador 'classf', con el cursor justo a la derecha de la 'f' pulsamos las teclas Ctrl+J y 'classf' será reemplazado por

```
TClass = class(TBase)
private

public
  constructor Create;
  destructor Destroy; override;
end;
```

además el cursor de posicionará en 'TClass'.

Podemos desplegar la lista de plantillas disponibles, situando el cursor en un espacio (no en un identificador) y pulsando las teclas Ctrl+J. Aparecerá la lista de plantillas. HAciendo uso de las teclas de dirección o escribiendo algo seleccionaremos una de ellas. Con *intro* usaremos la plantilla elegida y con *escape* cerraremos la lista sin hacer nada.

La plantilla que más tiempo nos ahorrará es 'b'+Ctrl+J para begin..end.

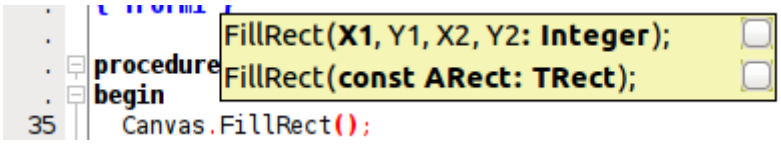
Sugerencia de Parámetros

Sugerencia de Parámetros muestra una caja con la lista de posibles declaraciones con sus parámetros para el procedimiento o función actual.

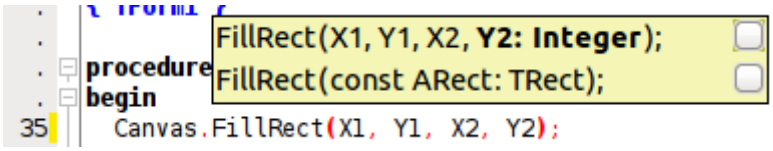
Por ejemplo:

```
Canvas.FillRect(|);
```

Situamos el cursor dentro de los paréntesis y pulsamos Ctrl+⇧ Shift+space. La lista de sugerencias aparecerá con los posibles parámetros, según las distintas declaraciones de *FillRect*.



Desde la versión 0.9.31 hay un botón a la derecha de cada declaración para insertar el parámetro. Esto copiará los nombres de los parámetros de la declaración elegida en la posición del cursor.



Consejo: Utiliza Completar Declaración de Variable para declarar las variables.

Nota: El nombre del atajo es "Show code context" (Mostrar Contexto del Código).

Búsqueda incremental

Búsqueda incremental cambia la barra de estado del editor. Escribe algunos caracteres y el editor buscará y resaltará de inmediato todas las ocurrencias en el texto. El atajo es Ctrl + e.

- Por ejemplo pulsa e buscará y resaltará todas las apariciones de 'e'.
- Pulsa t buscará y resaltará todas las apariciones de 'et' y así sucesivamente.
- Puede saltar a la siguiente con F3 (o Ctrl + e mientras se busca) y a la previa con ⇧ Shift + F3.
- ← Backspace elimina el último carácte
- ↵ Enter detiene la búsqueda sin añadir una nueva línea en el editor.
- Puedes volver a la última búsqueda pulsando Ctrl + e una segunda vez, inmediatamente después de que empezó con la búsqueda con Ctrl + e es decir, mientras que el término de búsqueda está vacío.
- Pegar: Ctrl + V adjuntará el texto del portapapeles al texto de búsqueda actuales (desde lazarus 0.9.27 r19824).

Sugerencia: Búsqueda rápida de un identificador con búsqueda incremental

- Mueve el cursor del texto sobre el identificador (no selecciones nada)
- Pulsa Ctrl + C. En el editor de código fuente se selecciona el identificador y se copia en el portapapeles
- Pulsa Ctrl + E para iniciar la búsqueda incremental
- Pulsa Ctrl + V para buscar el identificador (desde 0.9.27)
- Usa F3 y ⇧ Shift + F3 para ir rápidamente a siguiente / anterior.
- Usa cualquier tecla (por ejemplo, el cursor hacia la izquierda o la derecha) para finalizar la búsqueda.

Edición sincronizada

Edición sincronizada permite editar todas las apariciones de una palabra, al mismo tiempo (sincronizado). Simplemente se edita la palabra en un solo lugar, y mientras escribes, todas las otras apariciones de la palabra también se actualizan.

Edición sincronizada funciona en todas las palabras en un área seleccionada:

- Selecciona un bloque de texto
- pulsa Ctrl + J o haz clic en el icono en el margen. (Esto funciono solamente si hay alguna palabra que se repite más de una vez en la selección.
- usa la tecla Tab ⇆ para seleccionar la palabra que deseas editar (en caso de que varias palabras diferentes se repitan más de una vez)
- Edita la palabra
- pulsa Esc para terminar

Ver un ejemplo animado aquí

- Nota: Ctrl + J también se utiliza para editar plantillas. Cambia su significado si hay seleccionado algo de texto.

Completar Código

Completar código se encuentra en el menú Editar del IDE y su atajo por defecto es Ctrl + ⇧ Shift + C. También Aparece en el menú contextual Refactoring -> Completar código.

Para usuarios de Delphi: En Delphi "code completion" muestra la lista de identificadores en la posición actual del código (Ctrl + Space). En Lazarus esta función se llama "Completar identificador".

Completar código combina varias potentes funciones. Ejemplos:

- Completar Clase (*Class Completion*): completa propiedades, añade cuerpos de métodos, añade variables y métodos privados.
- Completar procedimiento (*Forward Procedure Completion*): añade cuerpos de procedimientos.
- Completar asignación de eventos (*Event Assignment Completion*): completa la asignación de eventos y añade la definición del método y su cuerpo.
- Completar declaración de variables (*Variable Declaration Completion*): añade la declaración de variables locales.
- Completar llamada a procedimiento (*Procedure Call Completion*): añade un nuevo procedimiento.
- Completar procedimiento inverso (*Reversed procedure completion*): añade la definición de cuerpos para procedimientos y funciones.
- Completar clase inversa (*Reversed class completion*): añade declaraciones para cuerpos de métodos.

La función que se activa depende de la posición del cursor en el editor de código.

Completar Clase

La función más potente de completar código es *Completar clase*. Escribe la definición de la clase, añade los métodos y las propiedades y Completar clase añadirá los cuerpos de los métodos, los métodos de acceso a las propiedades y variables y las variables privadas.

Por ejemplo: crea una clase (utiliza Platillas de código para ahorrarte trabajo de escritura):

```
TEjemplo = class(TObject)
public
  constructor Create;
  destructor Destroy; override;
end;
```

Pon el cursor en la clase y pulsa Ctrl+Shift+C. Con esto se crean los cuerpos que faltan de los métodos y el cursor se situará dentro del primer cuerpo de método creado, y ya puedes empezar a escribir el código de la clase:

```
{ TEjemplo }

constructor TEjemplo.Create;
begin
  |
end;

destructor TEjemplo.Destroy;
begin
  inherited Destroy;
end;
```

Nota: El carácter '|' representa el cursor.

Sugerencia: puedes saltar entre la definición y el cuerpo de un método con Ctrl + ⇧ Shift + ↑.

Cómo se puede ver, el IDE añade la llamada 'inherited Destroy' siempre que el método esté definido con *override*.

Ahora añade un método HacerAlgo:

```
TEjemplo = class(TObject)
public
  constructor Create;
  procedure HacerAlgo(i: integer);
  destructor Destroy; override;
end;
```

Ahora se pulsa Ctrl + ⇧ Shift + C y el IDE añadirá

```
procedure TEjemplo.HacerAlgo(i: integer);
begin
  |
end;
```

El método se inserta entre Create y Destroy, igual que en la definición de la clase. De esta forma los cuerpos mantienen el mismo orden lógico que se ha definido. Se puede definir la política de inserción en Entorno -> Opciones de CodeTools -> Creación de código.

Completar Propiedades

Añade la propiedad UnEntero:

```
TEjemplo = class(TObject)
public
  constructor Create;
  procedure HacerAlgo(i: integer);
  destructor Destroy; override;
  property UnEntero: Integer;
end;
```

Pulsa Ctrl+Shift+C y obtendrás esto:

```
procedure TEjemplo.SetUnEntero(const AValue: integer);
begin
  |if FUnEntero=AValue then exit;
  FUnEntero:=AValue;
end;
```

Completar código ha añadido el procedimiento de acceso de escritura a la propiedad (*Write*) y añadido al mismo el código más común. Ve a la definción de la clase con Ctrl + ⇧ Shift + ↑ para ver los cambio realizados en la clase:

```
TEjemplo = class(TObject)
private
  FUnEntero: integer;
  procedure SetUnEntero(const AValue: integer);
public
  constructor Create;
  procedure HacerAlgo(i: integer);
  destructor Destroy; override;
  property UnEntero: integer read FUnEntero write SetUnEntero;
end;
```

La propiedad se ha extendido con los modificadores de acceso Read y Write. La clase tiene una nueva sección *private* con una variable 'FUnEntero' y el procedimiento 'SetUnEntero'. Es habitual en el estilo Delphi añadir 'F' delante del nombre de las variables privadas y 'Set' a los procedimientos. Si no quieres que esto ocurra, cambialo en Entorno -> Opciones de CodeTools -> Creación de Código.

Crear una propiedad de sólo lectura:

```
property NombrePropiedad: TipoPropiedad read;
```

Se expandirá a

```
property NombrePropiedad: TipoPropiedad read FNombrePropiedad;
```

Crear una propiedad de sólo escritura:

```
property NombrePropiedad: TipoPropiedad write;
```

Se expandirá a

```
property NombrePropiedad: TipoPropiedad write SetNombrePropiedad;
```

Crear una propiedad de sólo lectura con una función de lectura:

```
property NombrePropiedad: TipoPropiedad read GetNombrePropiedad;
```

La función GetNombrePropiedad será añadida:

```
function GetNombrePropiedad: TipoPropiedad;
```

Crear una propiedad con un modificador 'stored':

```
property NombrePropiedad: TipoPropiedad stored;
```

Se expandirá a

```
property NombrePropiedad: TipoPropiedad read FNombrePropiedad write SetNombrePropiedad stored NombrePropiedadIs
```

Cómo *stored* se utiliza para el *streaming* los modificadores *read* y *write* se añaden automáticamente.

Sugerencia: Completar identificador también reconoce propiedades incompletas, sugiriendo los nombres por defecto. Por ejemplo:

```
property NombrePropiedad: TipoPropiedad read |;
```

Sitúa el cursor un espacio después del 'read' y pulsa Ctrl+Space para invocar Completar identificador. En la lista desplegable se aparecerán la variable 'FNombrePropiedad' y el procedimineto 'GetNombrePropiedad'.

Completar Procedimiento declarado

Completar Procedimiento declarado *Forward Procedure Completion* es parte del *Completar código* y añade los cuerpos que faltan a los procedimientos declarados. Si se llama cuándo el cursor se encuentra en la declaración de un prodedimiento.

Por ejeamplo: Añade un nuevo procedimiento en la sección *interface*:

```
procedure HacerAlgo;
```

Sitúa el cursor sobre ella y pulsa Ctrl + ⇧ Shift + C para completar el código. Se creará lo siguiente en la sección de implementación:

```
procedure HacerAlgo;
begin
  |
end;
```

Nota: puedes saltar alternativamente entre el cuerpo del procedimiento y su definición pulsando Ctrl + ⇧ Shift + ↑.

El cuerpo del procedimiento se añade antes de los métodos de clase. Si hay más procedimientos declarados en *interface* el IDE procura mantener el orden de los mismos al crear los cuerpos. Por ejemplo:

```
procedure Proc1;
procedure Proc2; // nuevo procedimiento
procedure Proc3;
```

Si los cuerpos de Proc1 y Proc3 existen, el cuerpo de Proc2 se insertará entre ellos. Este comportamiento se puede configurar en Entorno -> Opciones de Codetools -> Creación de código.

Varios procedimientos:

```
procedure Proc1_Viejo; // el cuerpo existe
procedure Proc2_Nuevo; // el cuerpo No existe
procedure Proc3_Nuevo; //  "
procedure Proc4_Nuevo; //  "
procedure Proc5_Existente; // el cuerpo existe
```

Completar código añadirá los cuerpos de los tres procedimientos(Proc2_Nuevo, Proc3_Nuevo, Proc4_Nuevo).

¿Porqué se llama *Forward Procedure Completion*?

Por que no sólo funciona con procedimientos declarados en la *interface* , sino que también lo hace para procedimientos con el modificador *forward*. Las *codetools* tratan los procedimientos en la *interface* cómo si tuvieran un modificador *forward* implícito.

Completar asignación de eventos

Completar asignación de eventos (*Event Assignment Completion*) completa una única asignación de evento (Evento:=| sentencia). Se invoca cuándo el cursor está situado justo tras la asignación del evento.

Por ejemplo: En un método, el evento FormCreate mismamente, añade la línea 'OnPaint:=':

```
procedure TForm1.Form1Create(Sender: TObject);
begin
  OnPaint:=|
end;
```

El caracter '|' representa el cursor y no hay que escribirlo. Pulsa Ctrl+Shift+C para completar el código. La sentencia se completará con

```
OnPaint:=@Form1Paint;
```

Un nuevo método Form1Paint se añadirá a la clase TForm1. Y el cursor se situará en el cuerpo de las misma, tras crear su esqueleto:

```
procedure TForm1.Form1Paint(Sender: TObject);
begin
  |
end;
```

El comportamiento es el mismo que cuándo en el inspector de objetos añadimos código para un evento de un objeto.

Nota:Hay que situar el cursor justo tras el operador de asignación ':='. si se sitúa en el identificador (v.gr. OnPaint) se llamará a *Completar variable local* lo que provoacará un error ya que este ya está definido.

Sugerencia: nosotros podemos definir el nombre del método, por ejemplo:

```
OnPaint:=@ElMetodoParaPintar;
```

Para ello hay que situar el cursor sobre el nombre del método antes de llamar a completar código con Ctrl + ⇧ Shift + C.

Completar Declaración de Variable

Completar Declaración de Variable (*Variable Declaration Completion*) es parte de *Completar Código* y añade la definición de una variable local para una sentencia identificador:=valor;. Se llama cuándo el cursor está sobre el identificador de la asignación o sobre un parámetro.

Por ejemplo:

```
procedure TForm1.Form1Create(Sender: TObject);
begin
 i:=3;
end;
```

Sitúa el cursor en la 'i' o justo detrás. Púlsa Ctrl + ⇧ Shift + C para completar el código y se obtiene:

```
procedure TForm1.Form1Create(Sender: TObject);
var
 i: Integer;
begin
 i:=3;
end;
```

El proceso comprueba primero si el identificador está definido y al no encontrarlo añade la declaración *var i: integer;*. El tipo de la variable es tomada del valor del término derecho de la expresión de asignación. Los números, cómo el 3, se crea por defecto por Integer.

Otro ejemplo:

```
type
 TDonde = (Final, Mitad, Principio);

 procedure TForm1.Form1Create(Sender: TObject);
 var
   a: array[TDonde] of char;
 begin
   for Donde:=Low(a) to High(a) do writeln(a[Donde]);
 end;
```

Sitúa el cursor en 'Donde' y pulsa Ctrl + ⇧ Shift + C, obtendrás:

```
procedure TForm1.Form1Create(Sender: TObject);
 var
   a: array[TDonde] of char;
   Donde: TDonde;
 begin
   for Donde:=Low(a) to High(a) do writeln(a[Donde]);
 end;
```

Desde la versión 0.9.11 Lazarus también completa parámetros. Por ejemplo

```
```

```
procedure TForm1.FormPaint(Sender: TObject);
 begin
   with Canvas do begin
     Line(x1,y1,x2,y2);
   end;
 end;
```

Sitúa el cursor en 'x1' y pulsa Ctrl + ⇧ Shift + C, obtendrás:

```
procedure TForm1.FormPaint(Sender: TObject);
 var
   x1: integer;
 begin
   with Canvas do begin
     Line(x1,y1,x2,y2);
   end;
 end;
```

Completar Llamada a procedimiento

Nota de traductor: lo que sigue no funciona, si lo intentamos no hace nada.

Completar código creará un nuevo procedimiento desde la orden de invocación.

Supongamos que acabas de escribir la orden *HacerAlgo(Ancho);*

```
procedure UnProcedimiento;
var
 Ancho: integer;
begin
 Ancho:=3;
 HacerAlgo(Ancho);
end;
```

Coloca el cursor sobre el identificador "HacerAlgo" y pulsa Ctrl + ⇧ Shift + C para obtener:

```
procedure HacerAlgo(aWidth: LongInt);
begin

end;

procedure UnProcedimiento;
 var
  Width: integer;
begin
 Width:=3;
 HacerAlgo(Width);
end;
```

Completar Clase inversa

"Completar Clase inversa" es otra función de **Completar ya Código** que añade la declaración de un método para el cuerpo de funcion o procedimiento en que esté situado el cursor cuando se invoca; si la declaración existie dará un mensaje de error: *Ya fue definido el identificador ....*

Esta función está disponible desde la versión 0.9.21 de Lazarus.

Por ejemplo:

```
 procedure TForm1.HacerAlgo(Emisor: TObject);
 begin
 end;
```

El método HacerAlgo no está declarado en TForm1. Pulsar Ctrl + ⇧ Shift + C y el IDE añadirá "procedure HacerAlgo(Emisor: TObject);" en la definición de la clase TForm1.

Para usuarios de Delphi: *Completar clase* funciona en Lazarus siempre en un único sentido: de *interface* a *implementation* o viceversa. Delphi invoca siempre los dos sentidos. La forma de Delphi tiene la desventaja de que por un error tipográfico podemos crear inadvertidamente un procedimineto nuevo.

Comentarios y *Completar código*

Completar código procura mantener los comentarios donde corresponde.

Por ejemplo:

```
FLista: TList; // lista de TComponent
FEntero: integer;
```

Cuándo se inserta una nueva variable entre FLista y FEntero, el comentario permanece en la línea de FLista. Esto también es verdad para

```
FLista: TList; { lista de TComponent
  Este es un comentario de varias líneas, comienza
  en la línea FLista, a la que la utilidad considera
  que pertenece y no romperá esta ligazón.
  El código se inserta después del comentario.}
FEntero: integer;
```

Si el comentario comienza en la línea siguiente, será tratado como perteneciente al código que le sigue. Por ejemplo:

```
FLista: TList; // lista de TComponent
  { Este comentario está ligado a la sentencia precedente.
    El nuevo código será insertado delante del comentario
    a continuación del comentario de la línea de FLista. }
FEntero: integer;
```

Refactorización (*Refactoring*)

Invertir Asignaciones

Invertir Asignaciones toma las instrucciones de asignación señaladas e invierte las mismas para ese código. Con esta utilidad es fácil transformar código de guardar en código de cargar, o a la inversa.

Ejemplo:

```
procedure HacerAlgo;
begin
 AValueUnoStudio := BValorUno;
 AValorDos := BValorDos;
 AValorTres  := BValorTres;
end;
```

Selecciona las líneas con las asignaciones (entre begin y end) e invoca *Invert Assignments*. Todas las ordenes de asignación se invertirán y el código se formateará automáticamente; el resultado será este:

```
procedure HacerAlgo;
begin
 BValorUno  := AValueUnoStudio;
 BValorDos  := AValorDos;
```

```
  BValorTres := AValorTres;
end;
```

Encerrar Selección

Selecciona un blouqe de texto e invócala, con el menú contextual (Refactoring -> Encerrar Selección), por ejemplo. En el [diálogo que aparece (http://wiki.lazarus.freepascal.org/images/2/24/Lazarus_IDE_Encerrar_Seleccion.png)] puedes seleccionar en que estructura se encerrará el código seleccionado, **try..finally** o cualquier otro tipo de bloque de los posibles.

Renombrar Identificador

Sitúa el cursor sobre un identificador y llámalo. Aparece un diálogo (http://wiki.lazarus.freepascal.org/images/b/bd/Lazarus_IDE_Buscar_Referencias_Identificador.png) en el que podemos configurar el ámbito de la búsqueda y el nuevo nombre.

- Se renombra´na todas las ocurrencias que utilicen esta declaración. No lo hará con otras declaraciones con el mismo nombre.
- Primero comprueba los posibles conflictos de nombre.
- Límites: Únicamente funciona con código pascal, no cambia el nombre a archivos y no modifica archivos *lfm/lrs* ni archivos *lazdoc*.

Buscar referencias a identificador

Sitúa el cursor sobre un identificador y llámalo (menú Buscar -> Buscar referencias a identificador...), aparece un diálogo (http://wiki.lazarus.freepascal.org/images/b/bd/Lazarus_IDE_Buscar_Referencias_Identificador.png), donde se puede configurar el ámbito de búsqueda. El IDE busca todas las ocurrencias y sólo aquellas que utilizan esta declaración. Esto quiere decir que no muestra otras declaraciones distintas con el mismo nombre.

Mostrar métodos abstractos

Esta utilidad lista y crea el esquema de métodos virtuales y abstractos que necesitan ser implementados.

Sitúa el cursor en la declaración de la clase y llámalo. Si hay métodos abstractos que implemetar un diálogo aparece con su lista. Selecciona el método para implementar y el IDE creará el esqueleto del mismo.

Extraer Procedimientos

Ver Extraer Procedimientos

Buscar Declaración

Coloca el cursor en un identificador y utiliza *Buscar declaración* del menú contextual, se buscará la definición del mismo y se abrirá el archivo que la contiene, situándose el cursor sobre ella. Si el cursor ya se encuentra ya en una declaración se saltará a la declaración anterior del mismo identificador. Esto permite encontrar sobrescrituras y redefiniciones

Cada búsqueda crea un punto de salto (*Jump Point*). Esto permite saltar entre la definición hallada y el punto donde estábamos antes, menú principal: Buscar -> Saltar atrás o Ctrl + H.

Existen algunas diferencias con Delphi: Las utilidades de completar código siguen las convenciones de pascal en vez de utilizar la salida del compilador. El compilador devuelve el tipo final. Las utilidades de completar código miran las fuentes y siguen todos los pasos intermedios. Por ejemplo:

La propiedad *Visible* está definida en primer lugar en *TControl* (controls.pp), está redefinida en *TCustomForm* y por último en *TForm*.

Llamando a buscar declaración para *Visible* se saltará primero a la definición de *Visible* en *TForm*. Si se llama de nuevo se saltará a *Visible* en *TCustomForm* y si lo hacemos otra vez saltará a *Visible* en *TControl*.

Esto también ocurre para tipos cómo *TColor*. Para el compilador es un simple *longint*. Pero en los fuentes está definido cómo

```
TGraphicsColor = -$7FFFFFFF-1..$7FFFFFFF;
TColor = TGraphicsColor;
```

Y ocurre lo mismo con las clases definidas con antelación (*forward defined classes*): Por ejemplo en *TControl*, está esta variable privada

```
FHostDockSite: TWinControl;
```

Buscar declaración sobre *TWinControl* saltará a la definición previa (*forward defined classe*)

```
TWinControl = class;
```

Y haciéndolo de nuevo saltará a la implementación real

```
TWinControl = class(TControl)
```

De esta forma se puede seguir hasta el final la declaración de cada identificador y ver cada sobrecarga o redefinición del mismo.

Sugerencia: Puedes regresar al punto anterior al salto con Ctrl + H.

Completar Identificador

Completar Identificador se llama con Ctrl + Space. Se mostrarán todos los identifiacdores en el ámbito. Por ejemplo:

```
procedure TForm1.FormCreate(Sender: TObject);
 begin
   |
 end;
```

Sitúa el cursor entre *begin* y *end* y pulsa Ctrl + Space. La utilidad e código del IDE analizará todo el código del ámbito y presentará una lista con todos los identificadores encontrados. El IDE guarda los resultados, así que una llamada posterior será mucho más rápida.

Nota para usuarios Delphi: En Delphi esto se denomina *Completar Código*.

Muchos identificadores cómo 'Write', 'ReadLn', 'Low', 'SetLength', 'Self', 'Result', 'Copy' están creados dentro del compilador y no están definidos en código alguno. Completar identificador hace lo propio. Si notas la falta de alguno informa de ello en el seguimiento de errores (*bug tracker*)

Completar identificador no completa palabras del lenguaje. No puede ser utilizado para completar proc a procedura. Para hacer esto utiliza Ctrl + W Completar palabra y no Ctrl + J Plantillas de código.

Completar identificador muestra incluso los identificadores que no son compatibles.

Prefijos

Cuándo activamos *Completar identificador* en una palabra, las letras a la derecha del cursor son consideradas un prefijo y la lista de sugerencias sólo mostrará los identificadores que comienzan por ellas. Por ejemplo:

```
procedure TForm1.FormCreate(Sender: TObject);
 begin
   Ca|ption
 end;
```

Sólo se mostrarán los identificadores que comienzan por 'Ca'.

Teclas

- Letra o numero: añade el carácter al código del editor y al prefijo. La lista se actualiza.
- Borrar hacia atrás (← Backspace): elimina el carácter código del editor y al prefijo. La lista se actualiza.
- Intro / ↵ Enter : reemplaza la palabra completa bajo el cursor con el identificador seleccionado y cierra la lista.
- ⇧ Shift + ↵ Enter : cómo ↵ Enter , pero reemplaza únicamente el prefijo (la parte izquierda) de la palabra bajo el cursor.
- ↑ / ↓ : mueve la selección de identificador en la lista.
- Escape : cierra la lista sin realizar cambios.
- Tab ⇆ : completa el prefijo a la siguiente selección. Por ejrmplo: El prefijo actual es 'But' y la lista sólo muestra 'Button1' y 'Button1Click'. Púlsando Tab ⇆ el prefijo se completa a 'Button1'.
- Otras teclas: realizan su cometido normal y cierran la lista si es necesario.

Métodos

Cuándo el cursor se encuentra en una definición de clase y se utiliza *Completar identificador* aparecerán en la lista los métodos de la clase padre con sus parámetros, e incluirá el *override*. Por ejemplo:

```
TMainForm = class(TForm)
protected
  mous|
end;
```

Completando con **MouseDown** obtendremos esto:

```
TMainForm = class(TForm)
protected
 procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X,
        Y: Integer); override;
end;
```

Propiedades

```
property MiEntero: integer read |;
```

Completar identificador mostrará en la lista ***FMiEntero*** y ***GetMiEntero***.

```
property MiEntero: integer write |;
```

Completar identificador mostrará en la lista ***FMiEntero*** y ***SetMiEntero***.

Sección *Uses* / nombres de unidad

En la sección *Uses* la función *Completar identificador* muestra los nombres de todos los archivos de todas las unidades en la ruta de búsqueda. Se mostraán en minúsculas (v.gr. **avl_tree**), ya que la mayoría tienen de unidades tienen el nombre en minúsculas. Al completar pondrá el nombre verdadero de la unidad (v.gr. **AVL_Tree**).

Sentencias

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ModalRe|;
end;
```

se convierte en:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ModalResult:=|;
end;
```

Completar palabra

Completar palabra se invoca con Ctlr + W. Muestra una lista con todas las palabras y todos los archivos abiertos en ese momento en el editor. Por lo demás se comporta igual que *Completar identificador*.

Ir a directiva de inclusión

Ir a directiva de inclusión en el menú Buscar del IDE salta a la sentencia {$I filename} en que el fichero actual de inclusión es invocado.

Publicar Proyecto

En el menú *Proyecto -> Publicar proyecto...*. Crea una copia del proyecto completo. Si quieres enviar tú código y sus opciones de compilación a alguien esto te ayudará.

Un directorio normal de proyecto contiene mucha información, pero no es necesario hacerla pública toda:

El archivo .lpi contiene información de la sesión (cómo la posición del cursor y los marcadores de unidades cerradas) y el directorio contine archivos *.ppu*, *.o* y el ejecutable.

Para crear un archivo .lpi únicamente con la información básica y con los fuentes, junto con los directorios necesarios utiliza *Publicar Proyecto*.

Nota: Desde la versión 0.9.13 existe una nueva *Opciones de proyecto Sesión (http://wiki.lazarus.freepascal.org/images/8/8f/Lazarus_IDE_Opciones_Proyecto_Sesion.png)* que permite guardar la información de sesión en un archivo distinto del .lpi normal. Este nuevo archivo tiene extensión *.lps* y contiene únicamente la información de la sesión, para dejar el *.lpi* más legible.

En el diálogo (http://wiki.lazarus.freepascal.org/images/c/c4/Lazarus_IDE_Publicar_Proyecto.png) puedes configurar el filtro de archivos a incluir y excluir y con *Command after* puedes procesar la salida para comprimir esta en un archivo.

Sugerencias desde comentarios

En varios lugares el IDE muestra sugerencias de un identificador. Por ejemplo cuando mueves el ratón sobre un identificador en el editor de código fuente y esperas unos segundos.

Cuando el IDE muestra algunas pistas para un identificador busca la declaración y la de todos sus ancestros y busca comentarios y archivos fpdoc. Hay muchos estilos de codificación y muchos estilos de comentario. A fin de apoyar muchos de los estilos de comentario común el IDE utiliza la heurística siguiente

Comentarios mostrados en la sugerencia

Comentarios precediendo a una declaración, sin línea vacía y que no empiezan con el signo <:

```
var
 {comentario}
 Identifier: integer;
```

Comentarios con el signo < pertenecen al identificador precedente.

Comentarios detrás de un identificador en la misma línea:

```
var
 identifier, // comentario
 other,
```

Comentarios tras la definición en la misma línea:

```
var
 identifier:
   char; // comentario
```

Un ejemplo para el signo <:

```
const
 a = 1;
 //< comentario para a
 b = 2;
 // comentario para c
 c = 3;
```

Los tres tipos de comentarios son compatibles:

```
{Comentario}(*Comentario*)//Comentario
 c = 1;
```

Los comentarios que empiezan con **$** y **%** son ignorados.

Comentarios que no se muestra en la sugerencia

Comentarios precedentes separados por una línea en blanco se tratan como no específicas del identificador siguiente. Por ejemplo el comentario de la cabecera de clase siguiente no se muestra en la sugerencia:

```
type
 { TMyClass }

 TMyClass = class
```

Los comentarios de cabecera de la clase se crean en la finalización de clases. Puede desactivar esta opción en *Entorno / Opciones / Codetools / Terminado de Clases / Comentario de cabecera para la clase*. Si desea mostrar el comentario de cabecera en la sugerencia, basta con eliminar la línea vacía.

El siguiente comentario será mostrado para GL_TRUE, pero no para GL_FALSE:

```
 // Boolean
GL_TRUE = 1;
GL_FALSE = 0;
```

Contribuciones y cambios

Esta página ha sido convertida desde la versión de epikwiki (http://lazarus-ccr.sourceforge.net/index.php?wiki=LazarusIdeTools).

- Creación de la página y plantilla original - 4/6/2004 VlxAdmin
- Nuevo contenido inicial - 4/10/2004 MattiasG
- Pequeños retoque sobre formato - 4/11/2004 VlxAdmin
- Adición de la Tabla resumen de los atajos de las *IdeTools* - 12 July 2004 User:Kirkpatc
- Versión en castellano (español) iskraelectrica (jldc) / junio-julio de 2008.