gICE
Gnome Intelligent C Editor

Carlos Rodríguez Caminero

20 de junio de 2002

Índice general

I Introducción general **5**

1. C y el código abierto **6**

2. GNU y Linux **8**

3. Entorno Linux y GNOME **9**

II *gICE* : Gnome Intelligent C Editor **11**

4. Objetivos **12**

5. Características **14**

5.1. Escribir y borrar 15
5.2. Desplazamiento horizontal y vertical 15
5.3. Deshacer y Rehacer 15
5.4. Seleccionar, Cortar, Copiar, Pegar 15
5.5. Cargar y Grabar 16
5.6. Buscar texto 16
5.7. Coloreado del texto 16
5.8. Auto-Identación 16
5.9. Auto-Formateado del texto 17
5.10. Señalar el carácter asociado 18
5.11. Sistema de marcadores 18
5.12. Búsqueda de errores 19
5.13. Ir a definición 19
5.14. Sugerir palabras 20

6. Más información **21**

III Manual de usuario **22**

7. ¿Que es *gICE* ? **23**

8. Requerimientos **25**

9. Instalación **27**

10. El entorno **28**

10.1. Menús 28
10.1.1. Fichero 29
10.1.2. Edición 30
10.2. Barra de herramientas 32
10.3. Preferencias 32
10.3.1. Generales 32
10.3.2. Colores 35
10.3.3. Paths 36
10.4. Barra de estado 37
10.5. Barra de progreso 37
10.6. El editor 38
10.6.1. barras de desplazamiento 38
10.6.2. zona de estado 38
10.6.3. zona de edición 38
10.7. Menú emergente 38
10.7.1. Ir a definición 39
10.7.2. Ir a marcador 39
10.7.3. Poner marcador 39
10.7.4. Quitar marcador 40
10.7.5. Deshacer 40
10.7.6. Rehacer 40
10.7.7. Sugerencias 40
10.8. Atajo de teclado 40

IV Descripción detallada **42**

11. Diagrama de bloques: *gICE* , *GtkCEdit* y *CEdit* **43**

11.1. Introducción 43
11.2. Ventajas de la modularidad 44

12. Estructura de datos 46
12.1. Líneas 46
12.2. Cursores 48
12.3. Deshacer / Rehacer 49
12.4. Lista de cambios 50
12.5. Definiciones 51

13. Funciones avanzadas 54
13.1. Auto-Identación 54
13.2. Coloreado del texto 58
13.2.1. cedit_flag_selected 58
13.2.2. cedit_flag_comment 59
13.2.3. cedit_flag_line_comment 59
13.2.4. cedit_flag_text 60
13.2.5. cedit_flag_apostrophe_text 60
13.2.6. cedit_flag_macro 60
13.2.7. cedit_flag_reserved_word 61
13.2.8. cedit_flag_number, cedit_flag_character y cedit_flag_mark 62
13.2.9. cedit_flag_error y cedit_flag_warning 62
13.2.10. cedit_flag_block 62
13.3. Auto-Formateado del texto 62
13.4. Señalar el carácter asociado 64
13.5. Sistema de marcadores 64
13.6. Búsqueda de errores 65
13.6.1. Buscar definiciones 66
13.6.2. Buscar errores 72
13.7. Ir a definición 72
13.8. Sugerir palabras 72

V Conclusiones 73

14. Hitos conseguidos 74

15. ¿Y ahora que? 75

16. Conocimientos adquiridos 76

17. Bibliografía y programas usados 77
17.1. Bibliografía 77
17.2. Programas usados 78

18.Estadísticas **79**

Parte I

Introducción general

Capítulo 1

C y el código abierto

El ordenador solo es capaz de entender un lenguaje, llamado lenguaje máquina, aunque es posible escribir programas directamente en lenguaje máquina, es extraordinariamente difícil hacer programas “grandes”, por lo que se inventan lenguajes que intentan parecerse más al nivel de lenguaje humano.

Según más se acerca al lenguaje humano, se dice que es de más alto nivel. *Ensamblador* fue el primer lenguaje de programación, aunque permite hacer programas en un lenguaje al menos legible por personas, este aún es de muy bajo nivel, por esto se crean lenguajes que una instrucción (normalmente con un nombre ingles fácil de entender) ejecuta diversas instrucciones en ensamblador (a este proceso de transformar instrucciones de un lenguaje de alto nivel a otro de más bajo nivel se le llama compilación).

C fue creado en los años 70 para poder crear un sistema operativo (UNIX), este es un lenguaje a medio nivel entre el lenguaje máquina y el lenguaje natural, hoy en día muy usado en programación a todos los niveles y para muy variadas tareas (sistemas operativos, programas basado en ventanas, videojuegos, editores de texto...).

Para crear un programa, el programador escribe un texto en lenguaje C (código fuente) que es compilado (transformado y unido) al lenguaje máquina que el ordenador entiende y solo él ya que es casi imposible entenderlo por personas.

Durante la década de los 60-70 la comunidad de programadores ofrecía sus programas para que todo el mundo pudiese aprender de la nueva rama de conocimiento que se estaba creando, pero llegó hacia los años 70-80 la fiebre de comercializar ideas y con ello, lo que era tan normal comprar programas con el código para poder modificarlo y adaptarlo a los proble-

mas cotidianos, se cerró esta fuente para evitar que la competencia pudiese aprender los métodos usados en un determinado programa y para que si el usuario deseaba una modificación no pudiese hacerla y tuviese que pactar con la empresa productora del software.

Esto a llegado a extremos que has de pagar por un programa y por todas las actualizaciones que van saliendo, así como por sus modificaciones y por los manuales y no pagas por el programa sino por la licencia para poderlo instalar en 1 ordenador. Un ejemplo claro de todo esto: Windows no trae manuales, a pesar del precio que tiene, y comprando Windows solo puedes instalarlo en 1 ordenador, si quieres instalarlo en 10 ordenadores, entonces te dan 1 CD y 10 licencias (10 claves) al precio de 10 Windows, claro.

Peor aún, si una modificación no le interesa hacerla a la empresa, no se hará o si un programa deja de tener éxito comercial se dejará de producir y de dar servicio técnico a todo aquel que lo compró (habían países del tercer mundo que no tenían Windows en su idioma, pero no podían tenerlo porque eran demasiados pocos para que Microsoft hiciese la personalización, pero tampoco dejaban el código para que ellos mismos se lo hicieses. La solución fue Linux, ya que ellos mismos se lo hicieron).

El término *open-source* se atribuye a programas que con la adquisición del ejecutable se acompaña con el código fuente y con documentación que explica su uso. De esta forma el usuario, si tiene los conocimientos necesarios, puede modificar el programa y adaptarlo, así como buscar funcionalidades no deseadas (programas que espían tu ordenador o puertas traseras) o arreglar fallos de programación o de diseño de quien lo fabricó.

También cabe destacar que con la aparición de GNU y Linux se ha asociado los programa de código abierto (*open source*) a programas gratuitos y a programas no comerciales. Nada más lejos de la realidad ya que nada impide que se pague por conseguir el programa, aún cuando por otros métodos resulte gratuito (descargarlo por Internet, por ejemplo) y también se suele pagar por la formación en el uso del programa o por adaptarlo a tus necesidades si no puedes hacerlo tu.

Y por último destaco que un programa abierto permite en la mayoría de casos copiarlo tantas veces como se quiera ya que se paga por la adquisición del producto, no por las licencias de uso (por ejemplo, es legal copiar Windows pero solo se puede instalar en un ordenador: no has comprado un programa, solo has comprado una licencia)

Capítulo 2

GNU y Linux

GNU (GNU's Not Unix) comenzó su historia hacia 1985, con la creación de diversos programas libres, con código abierto (compiladores, editores...) y llegaría a su apogeo hacia 1991 con la creación de Linux, el sistema operativo clonado a Unix, pero libre para todo usuario que lo desee.

Desde entonces GNU/Linux ha ido en expansión como sistema operativo, actualmente ya ha dejado de ser un sistema operativo usado por gurús informáticos para llegar al publico general, aunque requiere mayores conocimientos para ser usado que Windows, este ofrece una calidad y una estabilidad solo soñadas por este.

GNU/Linux solo es el programa que inicia el ordenador, GNU ha creado el resto de programas (miles y miles), dada la complejidad de que un usuario recopile todos los programas para que puedan ser usados (instalarlos y configurarlos), se han creado diversas empresas que te dan diversos CD'S (o DVD's) que se instalan como otros sistemas operativos y dejan un GNU/Linux instalado con cientos de programas listos para usar.

Entre todas las distribuciones destacaré RedHat: (www.redhat.com), SusE: (www.suse.com), Mandrake (www.mandrake.com) y muchas más. Alguna española: HispaFuentes (www.hispafuentes.com)

Para adquirir GNU/Linux se puede conseguir en distribuciones con varios CD's (7 o 8 y algún DVD) y manuales, gratuitamente por Internet (todas las distribuciones regalan los dos o tres primeros CD's), con este sistema no tienes servicio técnico ni tienes derecho a quejarte o en revistas especializadas que los bajan por ti y los distribuyen.

Capítulo 3

Entorno Linux y GNOME

Linux como ya he explicado es el sistema operativo, nació en 1991 por Linus Torvalds como un proyecto final de carrera, pero ha ido expandiéndose por una comunidad desinteresada que lo ha hecho crecer hasta hoy en día.

Pero los sistemas operativos actuales son en modo gráfico, y Linux, en sus comienzos no lo era, usaba un terminal en modo texto.

En 1986 nació de la mano de Xerox el sistema X, era un sistema gráfico, el cual fue copiado para hacer los sistemas Macintosh, del cual se copio Microsoft para hacer Windows. Posteriormente X se ofreció como código libre, lo cual generó que se continuara el proyecto desde diferentes empresas, la que yo destaco es Xfree que hizo continuar el proyecto con su variante libre.

Pero el sistema gráfico es independiente del sistema de ventanas que se use, a la práctica significa que hace falta un tercer programa (a parte de Linux y Xfree) para poder disfrutar el modo gráfico, este tercer programa es el que dibujara el entorno, las ventanas y todos los sistemas de interacción con el usuario para poder lanzar las aplicaciones.

Las posibilidades actuales son muchas, las más destacadas y evolucionadas son KDE con un entorno que busca la facilidad de uso y una calidad gráfica lo mas agradable posible. Y GNOME que busca un sistema de componentes lo mas potente posible, siendo este de calidad gráfica inferior.

GNOME significa GNU Network Object Model Environment, nació como una alternativa libre a otros sistemas, ya que KDE usa un sistema de componentes llamado Qt que no eran libres (no lo eran cuando comenzó GNOME, en agosto de 1997). GNOME usa un sistema de componentes llamado GTK. Un componente es un elemento en la pantalla (en realidad es una ventana, que puede recibir eventos y emitirlos), en la práctica es un

menú, un icono, una etiqueta...

GTK significa GIMP Toolkit y GIMP es Graphical Image Manipulation, GIMP es un programa de dibujo muy potente, para realizarlo crearon un conjunto de botones, menús, iconos... (a partir de ahora usare el nombre que se le da: widget) tan bueno que fue usado como estándar para el desarrollo de cualquier otro programa, esto es GTK.

Actualmente GTK va por la versión 2.0 que salio en mayo (he usado la versión de desarrollo para poder comenzar el proyecto) y su programación es muy estructurada y sencilla, pudiéndose realizar en diferentes lenguajes de programación (GTK está realizado en C, pero nada impide realizar programas en GTK usando Python, C++, Ada, Perl, Pascal...). GNOME también saca a la par su versión 2.0, está previsto su salida el 21 de Junio.

Parte II

gICE : Gnome Intelligent C Editor

Capítulo 4

Objetivos

Un editor de textos es un programa de uso cotidiano que permite escribir caracteres y almacenarlos (Solo me refiero a un editor de textos para escribir textos planos, no que pueda insertar dibujos, botones, o fórmulas matemáticas), pero existen pocos que busquen ayudar realmente al usuario, por ejemplo, ¿el programa Microsoft Word supone alguna ventaja a una máquina de escribir?, una persona que quiera escribir un texto pensará que si que es una ventaja, ya que, entre otras cosas, corrige faltas ortográficas y permite modificar el texto, pero las máquinas de escribir eléctricas también permiten esto y más. Entonces, si un ordenador tiene miles de veces mas potencia de cálculo respecto a una máquina de escribir, entonces los editores deberían de hacer trabajos miles de veces más complicados que una máquina de escribir.

En los editores de textos para realizar programas la falta es más evidente, hasta hace 6 o 7 años los editores de texto solo podían escribir, borrar, copiar y pegar. Luego se introdujo el coloreado del texto, esto es que las letras adquieren un color u otro, según el uso que tiene dentro del programa.

Algunos editores incluían también un inicio de identación automática: cuando el usuario apretaba el retorno, el cursor no iba al inicio de la línea, sino se ponía debajo de la primera letra escrita de la línea anterior. Algunos más avanzados colocaban el cursor donde realmente debería de ir, haciendo una auto identación real, pero el fallo esta en que solo lo hacen al crear una nueva línea, no cambian la identación de las líneas que ya han sido creadas anteriormente.

Y peor aún, los editores para programar no corrigen las faltas ortográficas de la misma forma que lo hace un editor de uso común a pesar de que un lenguaje natural (Catalán, Castellano, Ingles...) tiene una léxico más

abundante que un programa (C solo contiene unas 40 palabras). El único editor que yo conozco que hace algo parecido a la corrección son los de la gama Visual de Microsoft Corp. (Visual Basic, Visual C...) pero no corrigen todas las faltas, solo las que tienen que ver con estructuras.

Por último, ya intenté durante diversas ocasiones hacer un editor de textos, pero no es tan fácil como parece. Los principales problemas son de memoria y de velocidad.

De memoria, porque hay que recordar el texto escrito, hay que poder insertar un nuevo elemento o borrarlo, hay que acordarse del orden de escritura para poder deshacer cambios o rehacerlos, hay que mantener una estructura que sea fácil de acceder y que permita moverse con libertad usando desplazamiento (scroll) horizontal y vertical.

De velocidad, porque no es viable que cada vez que se inserte una letra se tenga que mover todo el texto en memoria ni que se se dibuje toda la pantalla, hay que buscar el mínimo dibujado, sino escribir puede consumir todos los recursos del procesador. Además cualquier análisis usará más recursos, pudiendo pasar que al escribir muchas teclas de golpe, se vea un retardo en la escritura.

Por lo que el objetivo de mi editor tenían que ser:

- Crear un editor que ayudase realmente al programador, que fuese algo más que una máquina de escribir y que hiciese el todo el trabajo que pudiese ser automatizado.

- Crear un editor desde cero, ya que solo creándolo desde cero podría aprender el funcionamiento de un editor, así como optimizar el análisis del texto, ya que usando programas externos, todas las funciones avanzadas serían mucho más lentas.

Capítulo 5

Características

- Escribir, borrar
- Desplazamiento horizontal y vertical
- Deshacer y Rehacer
- Seleccionar, Cortar, Copiar, Pegar
- Cargar, Grabar
- Buscar texto
- Coloreado del texto
- Auto-Identación
- Auto-Formateado del texto
- Señalar el carácter asociado
- Sistema de marcadores
- Búsqueda de errores
- Ir a definición
- Sugerir palabras

Todas estas características las explicaré a continuación:

5.1. Escribir y borrar

Funciones básicas de cualquier editor, el límite de texto escrito es la memoria del ordenador, el editor va usando más memoria según se va necesitando y va liberando según le va sobrando, esto ya lo explicaré con mas detenimiento.

5.2. Desplazamiento horizontal y vertical

Normalmente los textos no caben todo en la pantalla, todos los editores tienen desplazamiento vertical (Scroll), esto es que si el texto rebasa el límite inferior o superior de la pantalla, no es dibujado, y al mover el cursor por encima o por debajo de estos límites, entonces el texto es movido hacia abajo o hacia arriba.

El desplazamiento horizontal es idéntico al vertical, pero este es con los límites izquierdo y derecho (este desplazamiento existen programas que no lo tienen).

5.3. Deshacer y Rehacer

Esta funcionalidad está integrada en todos los editores. Simplemente es que si te equivocas, puedes deshacer el fallo. A continuación puedes deshacer el deshacer (rehacer) si lo deseas.

El número de deshacer y rehacer que puede contener el editor es tanta como memoria tenga el ordenador, pero como sería desperdiciarla de esta forma, solo usa hasta un máximo de 16 Kbytes por cada opción (puede recordar las últimas 16000 letras escritas o borradas), este máximo puede ser modificado.

5.4. Seleccionar, Cortar, Copiar, Pegar

Otra funcionalidad normal en cualquier programa y que sirve para poder mover y quitar bloques de textos. La dificultad ha sido que si se copia en el editor, se pueda pegar en otro totalmente diferente o que si se copia en otro programa se pueda pegar en el editor.

El portapapeles (una sección de memoria para el intercambio de información entre varios programas) está implementada por X-Windows y facilitada el uso con GTK, por lo que no ha sido muy difícil implementarlo.

5.5. Cargar y Grabar

Indispensable para cualquier editor. Cargar es en realidad usar un texto nuevo e ir escribiendo lo que hay en el fichero, de esta forma un fichero cargado automáticamente estará identado, formateado y coloreado.

Grabar es recorrerse todo el fichero y guardarlo tal cual está, transformando la identación en espacios y sin tener en cuenta ni el color ni los errores.

5.6. Buscar texto

Buscar texto es una función muy útil en un editor de código fuente, pero también es una función muy fácil de implementar por lo que muchos programas la implementan con funciones extras (Búsqueda hacia atrás, sin distinguir mayúsculas y minúsculas, etc).

Como esta parte es controlada por *gICE* , y el entorno no era una de mis prioridades, se ha implementado una búsqueda sencilla pero funcional.

5.7. Coloreado del texto

Esta funcionalidad es especialmente útil y común a muchos editores de textos, se basa en que cada palabra aparezca de un color característico para saber que función tiene dentro del programa, *gICE* sabe diferenciar entre: letras, números, signos, comentarios (línea y bloque), textos, textos entre apostrofes, macros, palabras registradas de C, errores, advertencias, y texto seleccionado.

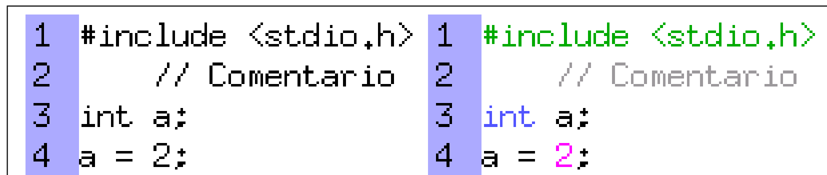


Figura 5.1: Diferencia a no usar coloreado a usarlo

5.8. Auto-Identación

Todos los programas se escriben en bloques funcionales, los bloques se reconocen en C porque comienzan con “{“ y acaban con “}”, o si la instruc-

ción es `for, while, do, if, else` y a continuación no llevan el signo "{", entonces la siguiente instrucción es considerada un bloque. Los bloques normalmente están anidados (existen bloques dentro de los bloques) , entonces para situarse en que bloque se está, cada bloque se comienza a escribir un poco más a la derecha que el anterior

Un ejemplo de identación:

```
if (variable>10)                    // si la variable es mayor a 10
{
    variable=0;                     // la pongo a 0
    while (variable<10)             // mientras la variable sea 10
    {
        iva=calcular_iva(variable); // calculo algo y lo guardo
        variable=variable+iva;      // a variable le sumo iva
    }
    mostrar(iva);                   // otra función
}

|   |   |
|   |   profundidad 2
|   profundidad 1
profundidad 0
```

5.9. Auto-Formateado del texto

Programar en C implica crear un texto con un lenguaje a medio camino entre el ingles y el lenguaje máquina, por lo que, aunque tiene muchas palabras que pueden ser leídas normalmente, contiene muchos signos. Al ordenador le da igual si el texto ha sido escrito usando espacios entre los signos, para dejarlos mas claros, o no.

El formateado de texto borra los espacios al principio y al final de una línea, además incluye espacios entre los signos, para aclararlos. En el siguiente ejemplo se verá la utilidad:

```
def_db->file=(char*)g_malloc(strlen(file)*sizeof(char));
```

por:

```
def_db -> file = (char *) g_malloc (strlen (file) * sizeof (char));
```

5.10. Señalar el carácter asociado

El lenguaje C es un lenguaje muy estructurado, formando fácilmente bloques. Muy a menudo se crean bloques dentro de bloques y eso provoca que no se sepa cuando se está terminando un bloque, o que bloque se esta cerrando. La auto-identación ayuda a ver en que profundidad nos encontramos, pero a veces no es suficiente.

gICE señala el carácter asociado a un abrir o cerrar bloque, esto es que te señala cual es el carácter que te cierra el bloque que estas abriendo o viceversa. Con el siguiente ejemplo espero dejarlo claro:

en esta función:

```
if (var1[funcion1(var2, funcion2(var3))])
```

tenemos los siguientes bloques:

```
if (                      <- si el cursor estuviese sobre el (
    var1[
         funcion1(
                  var2, funcion2(
                                 var3
                                )
                 )
        ]
    )                     <- este aparecería resaltado
```

Por lo que al colocar el cursor sobre el paréntesis del if (primera línea) será señalado el cerrar paréntesis asociado, el de la última línea.

Esta función es especialmente útil cuando hay demasiados bloques de diferentes tipos, de paréntesis (), corchetes [] o llaves , y si el asociado es incorrecto lo indica como error.

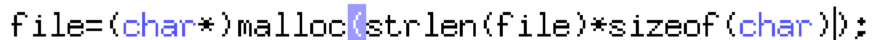


Figura 5.2: Ejemplo real de carácter asociado

5.11. Sistema de marcadores

Los marcadores son líneas señaladas con un número (entre el 0 y el 9) y con el color amarillo (color por defecto). Dejando un marcador (tecla control

+ número o con el menú emergente) se puede volver a ella más adelante cuando se necesite (con las teclas Alt + número o desde el menú emergente).

Esta función es frecuentemente usada ya que es muy común el tener que desplazarse a lo largo de un fichero para ver su contenido (sobre todo en ficheros largos), facilitando el volver al punto inicial.

Para quitar el marcador hay que pulsar nuevamente control + número, el número ha de ser el del marcador.

```
30    {
31 1      { "/_Fichero",                     NULL,
32        { "/Fichero/_Nuevo",               "<control>N",
```

Figura 5.3: Ejemplo real de un marcador

5.12. Búsqueda de errores

Es la función más complicada de todas, la idea es que el lenguaje C solo contiene unas 40 funciones predefinidas, cualquier otra instrucción ha de ser definida usando una sintaxis bien definida.*gICE* busca estas definiciones y las almacena en una base de datos interna, luego repasa el fichero, por cada palabra que aparezca, si no está en la base de datos, no es una palabra válida.

```
449      {
450          filled [i] = 0;
451          virtual_page [i] =- 1;
452          filename [i] [0] ='\0';
453      }
454      palabra inexistente
455      ArrayPath = NULL;
```

Figura 5.4: Ejemplo real de un error

5.13. Ir a definición

En muchas ocasiones tienes una función y quieres ver su código, pero este puede que se encuentre en otro fichero y es realmente saber en cual y

donde. Como anteriormente *gICE* se ha leído todas las definiciones, el sabe en que línea y fichero ha sido definida facilitándote la búsqueda.

El ir a definición también funciona, además de con funciones, con variables, definiciones y estructuras.

5.14. Sugerir palabras

Como consecuencia directa de tener una base de datos con todas las palabras posibles que se pueden usar en un programa se puede sugerir al usuario que palabra quiere escribir, ayudándolo a descubrir todas las instrucciones posibles o recordándole como era la instrucción deseada.

Capítulo 6

Más información

gICE tiene su propia página WEB, donde poder encontrar más información actualiza sobre el editor, así como ayuda.

La página es http://gice.sourceforge.net

Este documento se puede encontrar en formato .tex .html .dvi y .pdf en el apartado manual.

Parte III

Manual de usuario

Capítulo 7

¿Que es *gICE* ?

En líneas generales, *gICE* es un editor de textos, pero pensado para la edición de programas escritos en el lenguaje C. Dado que el lenguaje C es tan restrictivo en su sintaxis y que se basa únicamente en este lenguaje de programación tan específico, el editor es capaz de analizar el contenido del programa y adelantarse al programador, sugiriendo y revisando el programa escrito en tiempo de ejecución.

Además, intenta ser una ayuda al programador facilitando su trabajo gracias a la ayuda que ofrece y las facilidades que dispone su entorno de programación.

Aquí tengo que destacar que *gICE* es solo un entorno, el cual llama a todos los demás programas que son necesarios para la programación, entre estos programas destaco *CEdit* que es un componente del programa principal y que es el editor de textos (el lugar donde se puede escribir) el cual es el centro de este programa y la razón de ser de este proyecto.

gICE fue diseñado dada la dificultad de uso de los editores actuales para los principiantes, y la falta de imaginación de los programadores en la construcción de editores por no programar editores de texto más potentes. En los antiguos editores el programador ha de llevar todo el trabajo de la programación y la escritura del código y los editores son una mera máquina de escribir. *CEdit* busca la inteligencia en la escritura, ya que muchos de los procesos de escribir un programa son casi automáticos y predeterminados (identación, nombre de funciones, coloreado...) *CEdit* busca hacerlos el: que el ordenador haga parte del trabajo de la programación.

Esta idea en realidad no es nueva, ya que Microsoft ya la ha aplicado a su editores de textos (Visual Basic y Visual C entre otros), pero en Unix no conozco ningún editor que llegue tan lejos, todos los editores buscan un

entorno agradable y potente, pero el donde se escribe, es una máquina de escribir.

gICE ha sido diseñado en un entorno gráfico dada la facilidad de uso de esta, pero no está restringido a este, ya que *CEdit* , el núcleo del editor, es independiente del modo gráfico usado e independiente de la arquitectura y sistema operativo. Puede funcionar en cualquier máquina que sepa compilar C.

Entre las funciones que dispone *gICE* destaco las siguientes:

- Edición sencilla en un entorno gráfico potente, portable y flexible como es *Gtk* .
- Capacidad para ficheros enormes (el límite lo pone la memoria del ordenador).
- Velocidad, el editor ha sido diseñado para conseguir rapidez y fluidez de escritura. Si alguna opción del editor retarda su ordenador, puede ser anulada o configurada para que no consuma tantos recursos.
- Coloreado del texto según se escribe (opcional).
- Auto-Identación del texto.
- Auto-Formateado del texto.
- Análisis del texto buscando errores y avisos.
- Ir a definición: No busques más.
- Sugerir palabras.

Capítulo 8

Requerimientos

gICE necesita *Gtk* 2.0 para funcionar, esta librería es un componente del escritorio *Gnome* , con lo cual, o se dispone de este escritorio, o de la librería *Gtk* . Esta librería es nativa de Linux, pero también tiene soporte para otros sistemas operativos (Linux, Windows, Macintosh y Be-os entre otros), pero estos no han sido comprobados todavía. Gracias ha esta portabilidad de *Gtk* , *gICE* podrá funcionar en distintas plataformas sin tener que transformarla mucho.

Actualmente solo ha sido probado en el sistema operativo Linux bajo x86, pero gracias a la versatilidad de Linux para funcionar en diferentes arquitecturas, no debería tener ningún problema en funcionar en otras (Solaris, PPC, Mac...), harán falta mas pruebas para confirmarlo.

Estos son los requerimientos de hardware

- 200MHz (Recomendados 400MHz)
- 4Mb RAM (Contra más ficheros abiertos, más memoria necesita)
- 2Mb de disco duro (Programas y ficheros)
- Entorno gráfico (640x480 mínimo)

Necesitaras tener instalados los siguientes programas:

- Linux 2.2 o superior
- Xfree (probado con 4.0, seguramente con 3.0 también funcione)
- compilador GNU: gcc, djgpp...
- makefile GNU: make (en Linux, windows...)

- GTK 2.0 Conjunto de widgets. (www.gtk.org)
- GDK 2.0 Librería gráfica, base de GTK (www.gtk.org)
- GLIB 2.0 Librería de uso general (www.gtk.org)
- pkg-config herramienta para encontrar los componentes instalados

Equipos testeados:

- P2-400MHz, 450Mb RAM, Linux 2.4.3 (Mandrake 8.1), Gtk 1.3.15
- Athlon-1700MHz, 400Mb RAM, Linux 2.4.3 (Mandrake 8.2), Gtk 2.0
- Athlon-1700MHz, 400Mb RAM, Linux 2.4.18 (RedHat 7.3), Gtk 2.0

Capítulo 9

Instalación

El proceso de instalación no debería dar ningún problema si se tiene todas las librerías correctamente instaladas, para ello el primer paso es comprobar que todo esté correcto y configurar la instalación a su ordenador:

```
./configure
```

Si no ha ocurrido ningún error es que se ha encontrado con éxito todas las librerías, el siguiente paso es compilar la aplicación:

```
make
```

Esto generará el fichero `gice` que será la aplicación, puede copiarlo donde desee que resida, generalmente en `/usr/bin/` o `/usr/local/bin/`.

Ahora ya puede ejecutar *gICE* .

Capítulo 10

El entorno

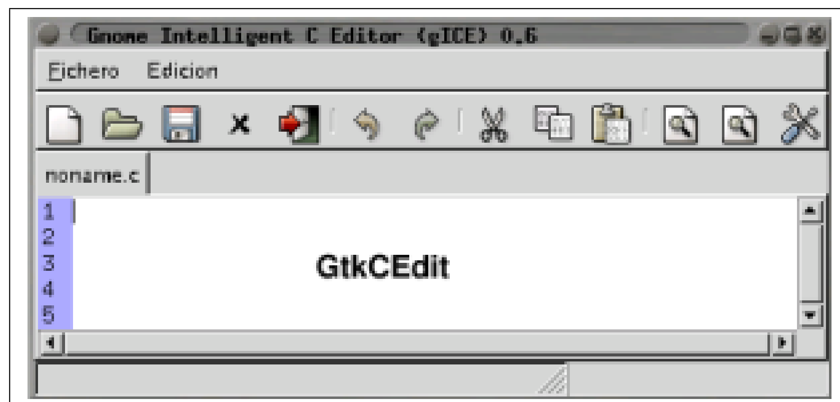


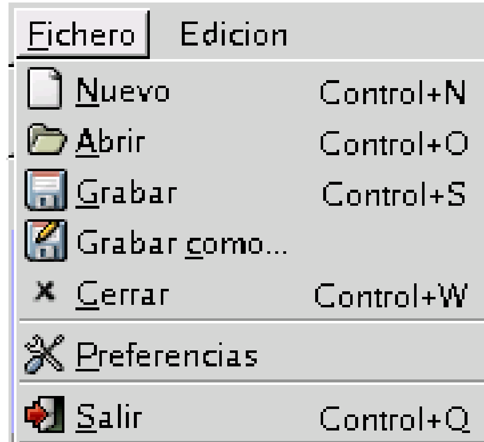
Figura 10.1: Captura de *gICE*

10.1. Menús

El menú es la parte más típica en una aplicación gráfica, desde aquí se puede acceder a las funciones más comunes de *gICE* . Se buscó un carácter sencillo ya que *gICE* no es la parte más desarrollada del proyecto

10.1.1. Fichero

Aquí están todas las opciones relacionadas con ficheros y con la configuración de *gICE* .



Nuevo

Crea una solapa nueva con un fichero en blanco, el nombre de este fichero es `noname.c` y al ser grabado *gICE* pedirá un nuevo nombre para hacerlo.

Abrir

Abre un fichero existente y lo carga en memoria para poder ser editado. Aparecerá una nueva solapa con el nombre del fichero abierto.

El fichero será analizado mientras se carga transformando toda identación que tuviese al estilo de *gICE* , será coloreado y cuando acabe de ser cargado comenzará el análisis léxico.

La barra de progreso abajo a la derecha indicará el estado de la carga del fichero.

Grabar

Guarda el estado actual del fichero en texto plano, los tabuladores son transformados en espacios en blanco como compatibilidad con otros editores de textos.

Grabar como

Opción idéntica a Grabar, pero pide un nuevo nombre para el fichero.

Cerrar

Cierra la solapa actual sin grabar el fichero, cualquier contenido de este es liberado y no puede ser recuperado.

Preferencias

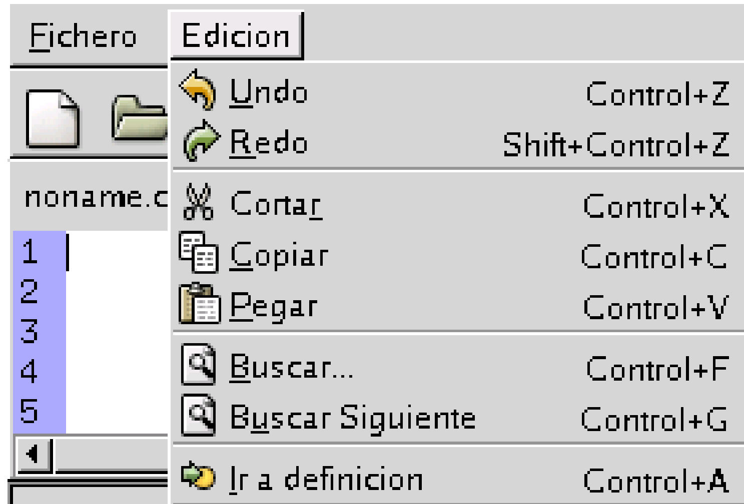
Abre el diálogo de preferencias, mira el punto sobre preferencias más adelante.

Salir

Cierra todas las solapas y libera toda la memoria obtenida por *gICE* . Salir puede que no sea inmediato ya que tiene que esperar a cerrar el sistema de análisis léxico.

10.1.2. Edición

Aquí están las opciones comunes de edición.



Deshacer

Deshace los cambios realizados palabra a palabra. El límite de estos cambios es de 16 Kbytes (16.000 letras). Si después de deshacer cambias de idea, se puede *Rehacer*.

Rehacer

Rehace algún cambio que había sido deshecho, al modificar el texto de nuevo, *Rehacer* es vaciado

Cortar

Después de seleccionar un texto (manteniendo la tecla Shift pulsada y moviendo los cursores o pulsando el botón izquierdo del ratón y arrastrando) se puede borrar esta selección y mantenerla en memoria para poderla copiar en otro momento en esta aplicación o en otra.

Copiar

Copia el contenido de una selección en memoria para poderla pegar en otro momento en esta aplicación o en otra.

Pegar

Pega el contenido copiado o cortado de esta aplicación o de otra en el texto actual. Para poder copiar y pegar de otra aplicación esta debe de dar el texto en formato plano.

Buscar

Abre el diálogo de buscar, en el cual se indica un texto a buscar a partir de la posición del cursor. El texto debe ser introducido teniendo en cuenta que se distinguirá entra mayúsculas y minúsculas (Case-sensitive), al igual que lo hace C.

Buscar siguiente

Repite la última búsqueda a partir de la posición actual del cursor.

Ir a definición

Mueve el cursor a la definición de la palabra que actualmente tiene apuntada el cursor, si está en otro fichero, este es abierto.

De todas las definiciones posibles, *gICE* escogerá la más adecuada. Si se desea ver todas las posibilidades, entonces use el menú emergente.

10.2. Barra de herramientas

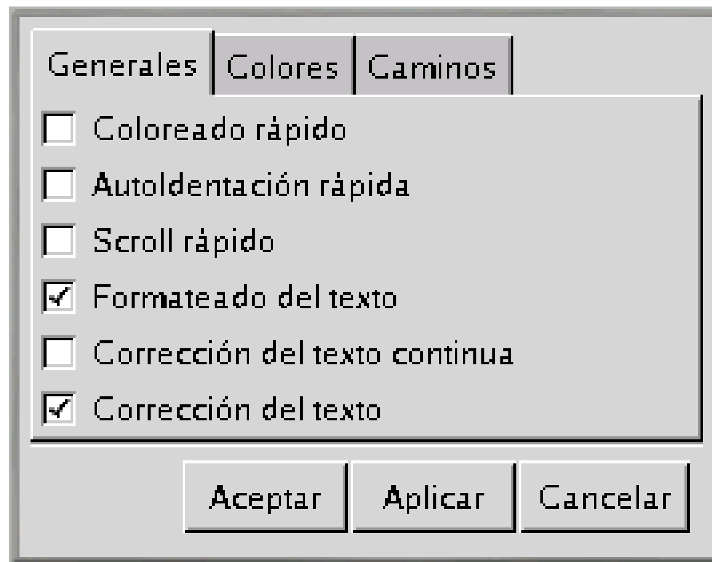
En la barra de herramientas se puede acceder a las opciones más comunes que se usan en *gICE* . Todas ellas se pueden acceder desde el menú, por lo que para más información mire en el apartado anterior o situe el cursor unos segundos encima de cada opción.

10.3. Preferencias

Aunque *gICE* se un entorno sencillo, este puede configurar todos los parámetros que necesita *CEdit* para funcionar, estos parámetros son *generales, colores y paths*. Para poder acceder a ellos se puede acceder por el menú *Fichero – Preferencias* o por la barra de herramientas (el último icono).

10.3.1. Generales

Aquí se pueden configurar las preferencias relacionadas con las opciones avanzadas de *CEdit* . Aquí se puede habilitar y deshabilitar estas características para poder ganar velocidad.



Coloreado rápido

Activándolo el coloreado se realiza cuando se cambie de línea y no mientras se escribe. Esta opción es recomendable para ganar velocidad mientras se escribe.

Es aconsejable (pero no obligatorio) habilitar también la Auto-Identación rápida, ya que esta requiere de la del coloreado para poder identar correctamente, puede provocar identaciones falsas, pero solo hasta que cambie de línea, entonces la identación será correcta.

Desactivándolo (valor por defecto), el coloreado será continuo mientras se escribe.

Auto-Identación rápida

Activándolo la Auto-Identación se realiza solo cuando de cambie de línea y no mientras se escribe. Esta opción es recomendable para ganar velocidad mientras se escribe.

Desactivándolo (valor por defecto), la Auto-Identación será continua mientras se escribe.

Scroll rápido

Activando el scroll rápido cuando el cursor se salga de la pantalla y esta tenga que moverse, la pantalla bajará o subirá más de lo necesario en

previsión de que el cursor siga bajando o subiendo.

Desactivándolo el scroll será suave, pero consumirá más recursos ya que la pantalla tendrá que moverse más.

Formateado del texto

Con esta opción se puede deshabilitar el formateado del texto, no es recomendado quitarlo ya que esta acción es la que menos recursos utiliza.

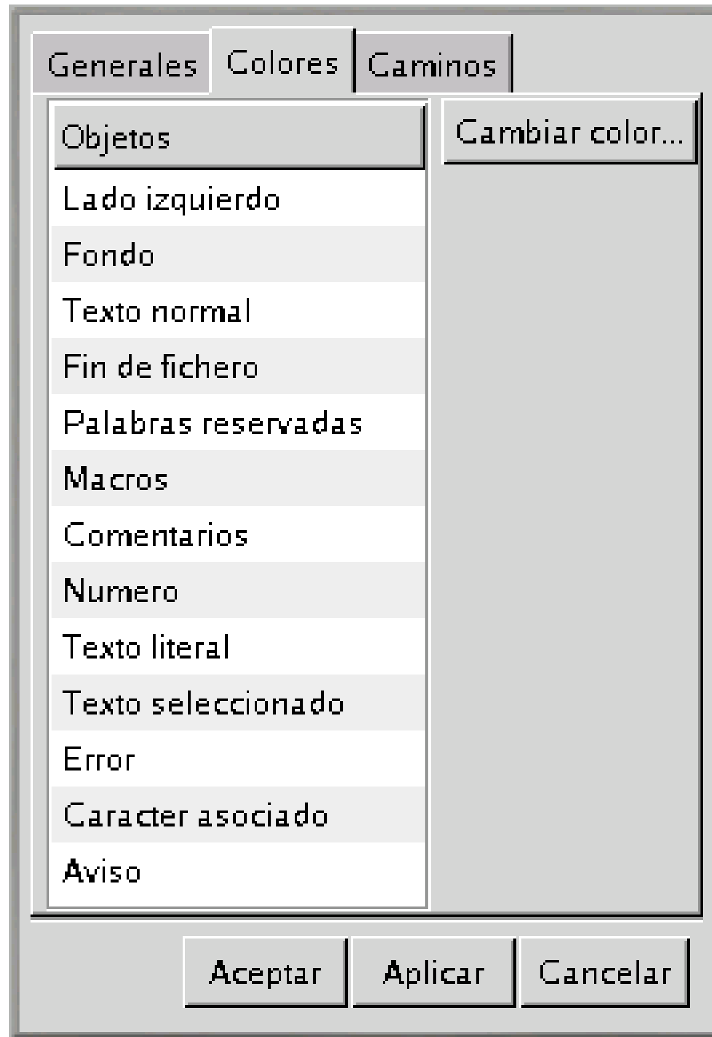
Corrección del texto continua

Activa/Desactiva la corrección del texto mientras se escribe. NO RECOMENDADO USARLO ya que el consumo de recursos puede ser excesivo y la palabra que se está escribiendo en esos momentos puede salir como error.

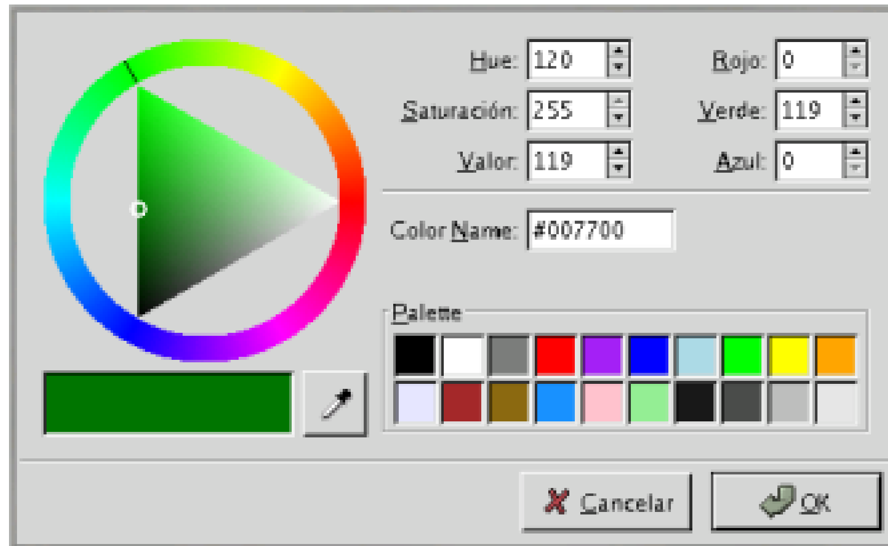
Corrección del texto

Activa/Desactiva el analizador léxico, esta acción requiere un momento.

10.3.2. Colores



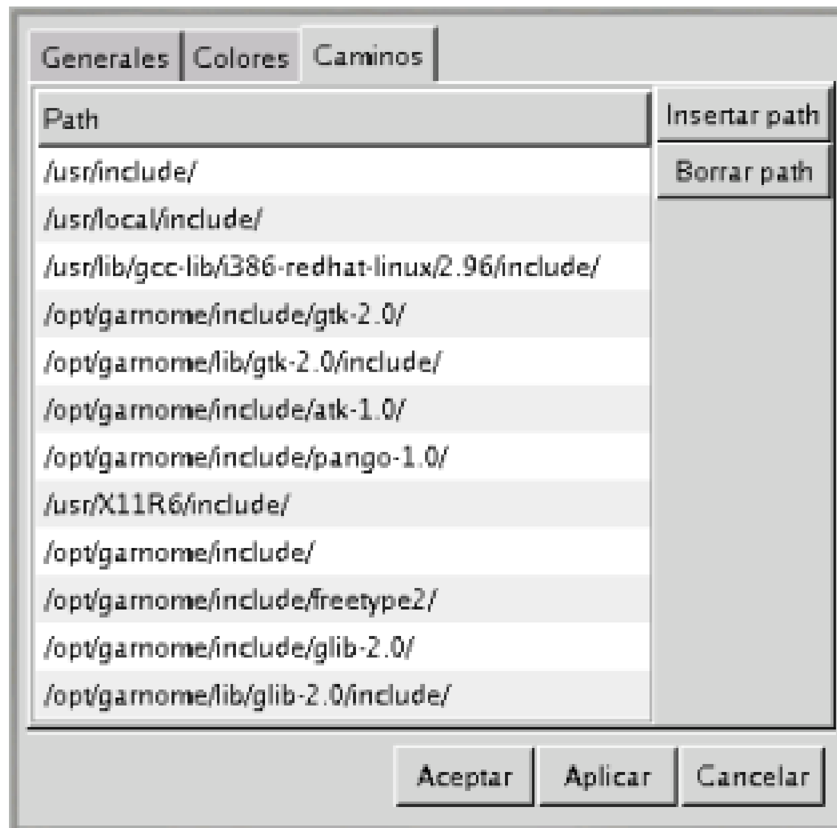
Aquí se puede cambiar el color de todos los elementos de *CEdit* eligiéndolo en la lista y usando el botón *cambiar color...*.



Este es el cuadro para cambiar el color, selecciones un color de la gama de colores a la izquierda y luego una tonalidad en el triangulo del centro (sistema HSV: Hue, Saturación y Valor), o elija uno de los colores ya predefinidos abajo a la derecha o, si lo prefiere, escriba el color con su componente Rojo, Verde y Azul o con el sistema HSV (Hue, Saturación y Valor) arriba a la derecha.

10.3.3. Paths

Los paths son los directorios donde el analizador léxico ha de ir a buscar los ficheros de cabecera del sistema, estos caminos dependen del sistema y dependen del proyecto. Cada programador ha de indicar donde se encuentran.



Con el botón *Insertar path* se inserta un nuevo camino a la lista, para cambiarlo haz doble click sobre la lista.

Para borrar un path de la lista, pulsa *Borrar path*.

10.4. Barra de estado

En esta barra se explica la última acción realizada, de esta forma puede estar seguro de que su fichero ha sido grabado o cual fue su última acción.

10.5. Barra de progreso

La barra de progreso sirve para poder saber cuanto le falta a una acción que se esté realizando. Actualmente solo se usa al cargar un fichero, en el futuro se le dará más uso.

10.6. El editor

El editor es el núcleo de la aplicación y la zona donde se desarrolla la acción de la aplicación.

Consta de 3 partes, la barras de desplazamiento, la zona de estado y la zona de edición.

10.6.1. barras de desplazamiento

Sirven para poder visualizar todo el contenido del fichero editado, la barra vertical usa un scroll suave pixel a pixel, la horizontal, mueve letra a letra.

10.6.2. zona de estado

Muestra información de la línea a la cual pertenece, indicando el número de línea y si posee una marca. En el futuro indicará también puntos de parada (breakpoints) y posición de la ejecución actual.

10.6.3. zona de edición

Zona donde se puede escribir el programa. Es el entorno gráfico del núcleo (*CEdit*) y todas las funciones avanzadas pueden ser visualizadas aquí.

Pulsando el botón izquierdo del ratón se consigue mover el cursor a la posición indicada, arrastrando el ratón manteniendo el botón izquierdo se consigue seleccionar una zona del texto.

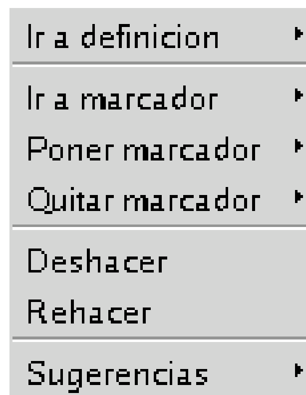
Pulsando el botón derecho se obtiene el menú emergente para poder realizar más acciones avanzadas.

10.7. Menú emergente

Este menú se obtiene al pulsar con el botón derecho del ratón sobre el área de edición, en el se pueden conseguir funciones específicas para la línea donde se pulse y para la palabra actual, sus funciones ahora se detallan:

10.7.1. Ir a definición

Aquí se muestran todas las posibles definiciones de la palabra actual, se suele usar esta opción cuando *gICE* no ha elegido la más adecuada o cuando se quiere ver todas las posibles definiciones.



Una palabra puede tener diferentes definiciones cuando cada una es de un tipo diferente (una variable, una función...) o cuando realmente hay una redefinición (cuando se declara una función y cuando se define una función).

Si esta opción aparece deshabilitada es que la palabra donde está el cursor no tiene ninguna definición o no se puede encontrar una definición válida.

Si se desea ir a la mejor definición simplemente usa el menú *edición – Ir a definición* o usa su atajo del teclado `Ctrl+a`.

Esta opción solo funcionará cuando el analizador léxico haya acabado su función.

10.7.2. Ir a marcador

Los marcadores es una señal que se dejan en una línea para poder volver a ella. Hay 10 marcadores disponibles y una sola línea solo puede tener uno cada vez.

Aquí aparecerán todos los marcadores que tiene el fichero actual numerados del 0 al 9, pulsando sobre uno de ellos el cursor se moverá al marcador deseado.

Esta opción aparecerá deshabilitada si no hay marcadores en el fichero actual.

10.7.3. Poner marcador

Deja uno de los 10 marcadores en la línea actual, solo aparecerán los marcadores que no han sido colocados.

Esta opción estará deshabilitada si no hay marcadores que poner.

10.7.4. Quitar marcador

Elimina un marcador del fichero actual dejándolo libre para poderlo dejar en cualquier otro momento.

Esta opción estará deshabilitada si no hay marcadores que quitar.

10.7.5. Deshacer

Deshace el último cambio, su función es idéntica a la que se puede acceder por el menú (*edición – deshacer*)

Esta opción estará deshabilitada si no hay cambios que deshacer.

10.7.6. Rehacer

Rehace un cambio deshecho, su función es idéntica a la que se puede acceder por el menú (*edición – rehacer*)

Esta opción estará deshabilitada si no hay cambios que Rehacer.

10.7.7. Sugerencias

Palabras válidas que pueden ser escritas en el contexto actual, solo sugiere las palabras que comiencen por la palabra que actualmente se está escribiendo.

Esta opción solo funcionará cuando el analizador léxico haya acabado su función.

10.8. Atajo de teclado

La mayoría de acciones se pueden acceder directamente desde el teclado, aquí está una lista de todas ellas:

Tecla	Descripción
Ctrl + n	Nuevo fichero
Ctrl + o	Abrir un fichero
Ctrl + s	Salva el fichero actual
Ctrl + w	Cierra el fichero actual
Ctrl + q	Sale de la aplicación
Ctrl + z	Deshace los cambios
Ctrl + Shift + z	Rehace los cambios
Ctrl + x	Corta el texto copiándolo en memoria
Ctrl + c	Copia el texto en memoria
Ctrl + v	Pega un texto de memoria
Ctrl + f	Busca un texto
Ctrl + g	Sigue buscando el último texto
Ctrl + a	Va a la definición de la palabra actual
Ctrl + Número	Deja la marca "Número"
Ctrl + Número	Quita la marca "Número" (si está en la línea actual)
Alt + Número	Va a la marca "Número"

Figura 10.2: Atajos de teclado

Parte IV

Descripción detallada

Capítulo 11

Diagrama de bloques: *gICE* , *GtkCEdit* y *CEdit*

11.1. Introducción

gICE se ha diseñado en 3 partes, para que sea más fácil el reutilizar el código para crear otros editores, estas partes son el entorno (*gICE*), el widget (*GtkCEdit*) y el corazón (*CEdit*).

gICE solo es un conjunto de widgets (un widget es cualquier elemento de una ventana: un menú, un botón, una etiqueta, etc. Véase la figura 11.1 para ver los widgets usados.), esta parte es la menos desarrollada ya que el entorno solo va a ser el envoltorio de la aplicación y no era el propósito del proyecto.

GtkCEdit es un widget en concreto, es el widget editor. Su función es la de hacer de intermediario entre la aplicación (*gICE*) y el corazón del editor (*CEdit*), además es el encargado de dibujar el contenido del editor, y ser el encargado de controlar las entradas del usuario (teclado y ratón) así como informar de estas entradas a *CEdit* . Este módulo se encarga además del Sistema de marcadores, del scroll y del menú emergente.

El intercambio de información se puede ver en la figura 11.2, aquí se puede observar que *gICE* no interactúa con *CEdit* , Ya que están a niveles diferentes, hace falta una capa intermedia para poderlos comunicar: *GtkCEdit*

CEdit es el editor realmente, aquí es donde se guarda la información, donde se añade y eliminan letras, se guardan los deshacer-rehacer, el sistema de cursores y donde se hacen casi todas las funciones avanzadas: Auto-Identación, Auto-Formateado del texto, Coloreado del texto, Búsqueda del

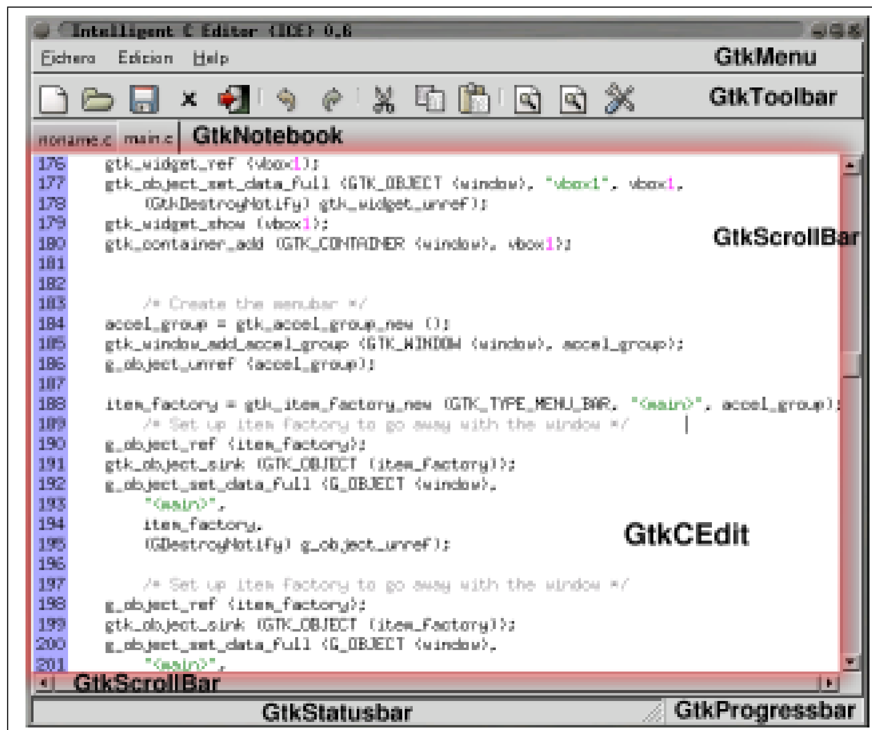


Figura 11.1: Widgets usados y GtkCEdit

signo asociado, Búsqueda de errores, Sugerencias e Ir a definición.

11.2. Ventajas de la modularidad

Gracias a esta modularidad, se puede crear otros módulos para que substituyan a los que yo he hecho.

Haciendo otro módulo *gICE* , conseguimos otro editor diferente con todas las capacidades (seguirá siendo una aplicación GTK al no cambiar *GtkCEdit*) o si además de cambiar *gICE* cambiamos *GtkCEdit* obtenemos otro editor escrito en otro sistema/lenguaje/entorno gracias a que *CEdit* es portable a cualquier sistema, solo necesita que soporte C (cualquier sistema lo acepta) y que tenga GLIB (Linux, Unix, Windows, Macintosh, Be-os y alguno más). También es independiente del sistema de entrada y de salida, por lo que se puede hacerse en modo gráfico (con cualquier librería: QT, GTK, MFC...) o modo texto, También es posible que no use el teclado y el ratón, quizás otras formas de entrada, no importa.

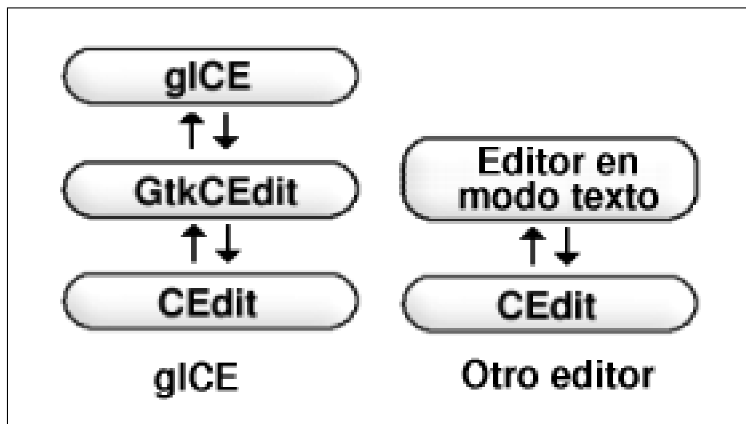


Figura 11.2: Diagrama de bloques comparado con otro posible editor

Capítulo 12

Estructura de datos

Dado que explicar el código sería demasiado costoso (cerca de 15.000 líneas) prefiero explicar el sistema de almacenamiento de los datos y como el programa los gestiona.

Todas las estructuras, sino se dice lo contrario, pertenecen a *CEdit* y están definidas en *cedit.h*.

12.1. Líneas

La forma más fácil de pensar un editor sería reservando tanta memoria como haya de texto e ir ampliando cuando se haga más grande y más pequeña cuando el texto se haga más pequeño.

Esta memoria seria un solo bloque donde iría todo el texto dentro, las ventajas son claras, el texto en pantalla y en la memoria se estructuran de la misma forma. El problema es que si el fichero tiene 100.000 líneas, estamos en la primera e insertamos un retorno, hay que mover las 99.999 líneas siguientes lo cual lo hace lentísimo.

Una mejora a este sistema es que cada línea sea independiente a las demás, esto se consigue con una lista doblemente encadenada, en la cual cada componente (cada línea) tenga un enlace a la línea siguiente y a la anterior.

Ahora si hay que insertar una nueva línea, solo hay que crear la línea nueva y mover los enlaces para incluirla en el texto, con lo cual insertar texto y nuevas líneas es inmediato sea cual sea la longitud del texto.

Este sistema es mucho más eficiente en cuanto espació y velocidad de inserción, pero tiene un fallo bastante grave, cada línea no debe poseer el número de línea que es, porque sino, al insertar una nueva línea se tendría

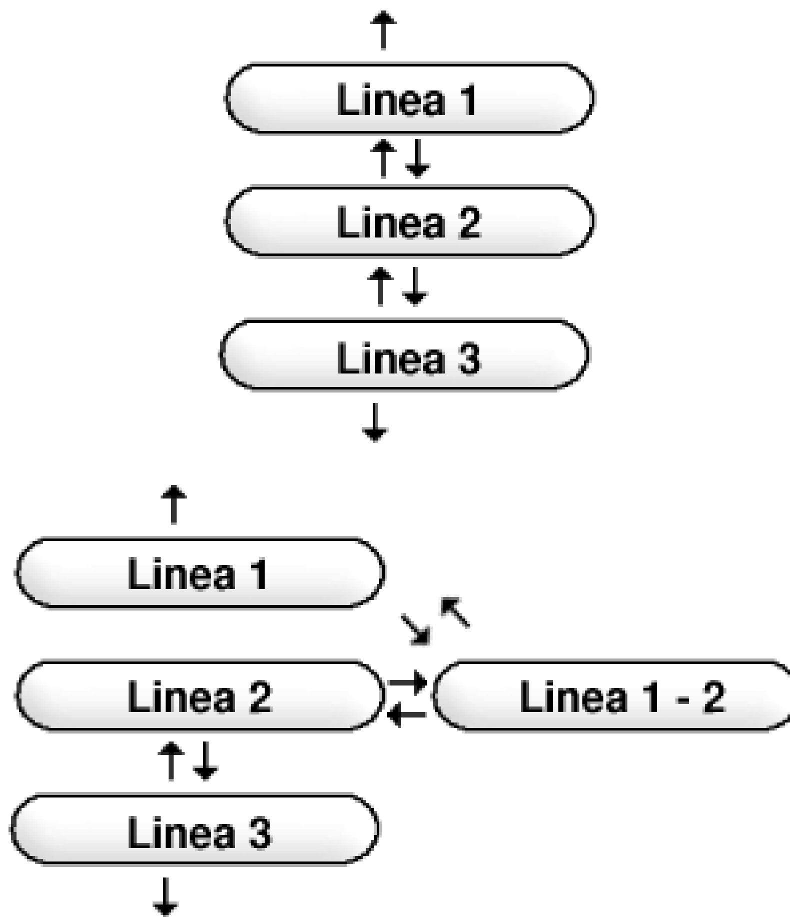


Figura 12.1: Esquema de líneas y como se inserta una línea

que modificar este número de línea por cada línea después de la actual, con lo cual se pierde toda ventaja.

Esto implica que para ir a la línea 124 hay que empezar por la primera e ir a la siguiente y a la siguiente... hasta llegar a la línea que queramos, por esta razón se crea los cursores.

Además del texto se guarda los flags para cada letra, un flag es una propiedad que puede estar activada o desactivada, cada letra tiene asociados 13 flags (Figura 12.2).

Todas ellas se guardan en el entero `flag` y es necesario para el coloreado y para el análisis (para no analizar los comentarios, por ejemplo)

Posición	Nombre	Descripción
00000 00000001	**cedit_flag_selected**	La letra está seleccionada
00000 00000010	**cedit_flag_comment**	Comentario normal: */* Comment */*
00000 00000100	**cedit_flag_line_comment**	Comentario de línea: *// Comentario*
00000 00001000	**cedit_flag_text**	Texto literal: *"texto"*
00000 00010000	**cedit_flag_apostrophe_text**	Texto entre apostrofes: *'c'*
00000 00100000	**cedit_flag_macro**	Si es una macro:*#define pi 3,14*
00000 01000000	**cedit_flag_reserved_word**	Si pertenece a una palabra de C.
00000 10000000	**cedit_flag_number**	Si es un número
00001 00000000	**cedit_flag_character**	Si es un carácter
00010 00000000	**cedit_flag_mark**	Si es un signo
00100 00000000	**cedit_flag_error**	Si la letra es un error
01000 00000000	**cedit_flag_warning**	Si es un advertencia
10000 00000000	**cedit_flag_block**	Si es el signo asociado

Figura 12.2: Listado de FLAGS

Cada línea además guarda 3 valores para poder hacer la Auto-Identación: `tabs, add_tabs, absolute_tabs`, Explicaré la Auto-Identación más adelante.

La estructura de la línea la puedes ver en la figura 12.3.

Tipo	Nombre	Descripción
char*	**text**	texto
int*	**flags**	flags del texto
line*	**next**	apuntador hacia la siguiente línea
line*	**prev**	apuntador hacia la anterior
int	**tabs**	tabuladores que suma o quita esta línea
int	**add_tabs**	tabs que se añaden solo a esta línea
int	**absolute_tabs**	tabs absolutos, este prevalece si es mayor 0
long	**num_char**	número de caracteres
int	**num_assigned_step**	número de bloques de caracteres reservados

Figura 12.3: Estructura `line`

12.2. Cursores

Para poder navegar entre todas las líneas se crea la idea de un cursor, esto es lo que se conoce como punto de interés y agiliza el acceso y la búsqueda de elementos.

Esto implica que la única forma de acceder al texto (por un agente externo, como es *GtkCEdit* o *gICE*) solo lo pueden hacer con la ayuda de los cursores, no se debería de acceder de una forma directa al texto almacenado.

Los cursores además tienen la función de agilizar la Auto-Identación, ya

que entre sus variables se encuentra `tabs` que indica el número de tabuladores que tiene la línea actual, esto se explicará con más detalles más adelante.

Y por último cada cursor tiene su propia pila de Deshacer / Rehacer (Undo/Redo), esta será la siguiente estructura que explique (figura 12.4).

Tipo	Nombre	Descripción
long	**x**	posición X del cursor
long	**y**	posición Y del cursor
line*	**l_line**	línea actual
int	**tabs**	número de tabuladores que tiene esta línea
stack_undo*	**undo**	pila de deshacer y rehacer

Figura 12.4: Estructura `array_cursor`

Hay dos funciones interesantes a comentar:

```
cedit_cpush_cursor (CEdit *cedit, int cursor);
cedit_cpop_cursor (CEdit *cedit, int cursor);
```

La primera almacena en una pila el cursor actual, para que pueda ser movido y luego con la segunda función pueda ser recuperado como estaba antes.

12.3. Deshacer / Rehacer

El deshacer y rehacer quizás pueda parecer difícil de ver como almacenar el texto para que pueda ser deshecho, pero una vez que se piensa en el problema encontré una solución sencilla.

gICE guarda letra por letra todos los cambios que se han hecho (a fin de cuentas, solo se puede ir escribiendo de una en una letra), entonces, cada vez que se pulsa una tecla, se guarda la recíproca. Si se ESCRIBE la letra “a”, yo guardo BORRAR la letra “a” (en su posición, claro) y si se BORRA la letra “b”, yo guardo ESCRIBIR la letra “b”.

El cambio de la pila de deshacer a la pila de rehacer (cuando se pulsa el botón de deshacer) es también fácil, cuando se deshace algo, se pasa el cambio de una pila a la otra cambiando el tipo de cambio (de borrar a escribir y viceversa)

¿Y que pasa si pego un texto y luego lo deshago?, pues el pegar un texto en realidad es escribirlo LETRA A LETRA, por lo que el deshacer es ir borrándolo LETRA A LETRA. Por ahora no se ha implementado un deshacer y rehacer con bloques de textos porque no era esencial, pero

Tipo	Nombre	Descripción
element_undo*	**first_undo**	primer undo de la pila
element_undo*	**last_undo**	último undo de la pila
element_undo*	**first_redo**	primer redo de la pila
element_undo*	**last_redo**	último redo de la pila
long	**undo_num_char**	nº de caráct. almacenados en el undo
long	**redo_num_char**	nº de caráct. almacenados en el redo
long	**limit_undo_redo**	Capacidad del undo/redo en bytes

Figura 12.5: Estructura `stack_undo`

Tipo	Nombre	Descripción
int	**mode**	Si es borrar o insertar
char*	**text**	Texto borrado
int	**join**	Si pertenece a un bloque (no usado)
int	**advance**	Si tiene que avanzar el cursor o no
long	**x, y**	Posición del cursor
line*	**l_line**	Línea del cursor
element_undo*	**next**	Siguiente elemento
element_undo*	**prev**	Anterior elemento

Figura 12.6: Estructura `element_undo`

observando la estructura se puede observar que `text` permite guardar más de una letra (aunque no ha sido usada)

La estructura del deshacer / rehacer que tiene cada cursor la encontraras en la figura 12.5.

Cada apuntador apunta a un elemento undo o redo (son indiferentes), la estructura de los elementos esta en la figura 12.6.

12.4. Lista de cambios

Antes he comentado que escribir en la pantalla es muy costoso, tanto que no es viable dibujar toda la pantalla cada vez que hay un cambio, y por otra parte hace falta informar a *GtkCEdit* que partes de la pantalla ha cambiado cuando se ha pulsado una tecla o se ha corregido el fichero.

Para esto se ha implementado una lista de cambios. Cada vez que un elemento de la pantalla cambia se añade un nuevo cambio a esta lista, los cambios pueden ser de una letra (modificar una letra), hasta el final de la línea (cuando se inserta una letra) o hasta el final del archivo (cuando se inserta o borra una línea). También se indica la posición X e Y del cambio.

Tengo que indicar que añadir un cambio es un poco más inteligente que añadirlo a la pila, ya que si, por ejemplo, hay 300 cambiar una letra, es más rápido hacer un cambiar hasta el final del archivo. También tiene en cuenta el caso de que hayan dos cambios de la misma letra (un cambiar línea y un cambiar hasta el final del archivo, los dos cambiaran las mismas letras en la línea) por lo que se elimina el cambio más pequeño en favor del cambio más grande.

Ahora que tenemos todos los cambios en una lista, *GtkCEdit* solo la tiene que recorrerla para ir dibujando SOLO LO QUE HA CAMBIADO, con lo que se consigue la máxima velocidad de dibujado.

La estructura de cambios está en la figura 12.7.

Tipo	Nombre	Descripción
long	**x**	posición X del cambio
long	**y**	posición Y del cambio
line*	**l_line**	línea del cambio
int	**type**	tipo de cambio (letra, línea o fichero)
list_change*	**next**	siguiente elemento de la lista
list_change*	**prev**	anterior elemento de la lista

Figura 12.7: Estructura `list_change`

12.5. Definiciones

El analizador léxico se basa en ir almacenando todas las definiciones para luego poder corregirlas.

Más importante que como se analiza un texto es como se almacena para su posterior recuperación. El método usado es doble: un árbol binario balanceado para poder almacenar cada elemento y poderlos buscar de una forma rápida y un sistema de particiones donde cada elemento indica a que tipo pertenece.

En el árbol se guarda pares clave valor (como en un diccionario), la clave es el nombre de la definición y el valor es la estructura `definition_db` representada en la figura 12.8.

Por otro lado, la estructura tiene enlaces a ella misma:

- `definition_type` que representa al tipo del cual procede la definición, se puede ver un ejemplo sencillo de una definición de un contador de tipo entero en la figura 12.9.

Tipo	Nombre	Descripción
char*	**name**	nombre de la definición
int	**name_long**	número de letras del nombre
int	**type**	tipo de elemento (figura 13.9).
definition_db*	**definition_type**	tipo del cual procede (padre)
int	**pointer**	1 si es un puntero, 0 sino
char*	**typedef_name**	nombre del tipo al que representa
int	**n_args**	número de argumentos
definition_db**	**definition_type_args**	tipo del cual procede el argumento
long	**n_line**	número de la línea de la definición
char*	**file**	fichero de la definición
long	**n_line_min_boundary**	nº de línea en que comienza el ámbito
long	**n_line_max_boundary**	nº de línea en que finaliza el ámbito

Figura 12.8: Estructura `definition_db`

- `definition_type_args` que es una lista a todas las definiciones que este defina, esto es, una definición a todos los argumentos en las funciones y a todas las variables en las estructuras. Hay un ejemplo del primer caso en la figura 12.10.

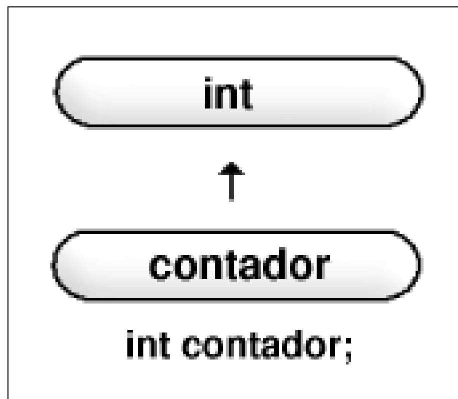


Figura 12.9: Esquema de `definition_type`: padre de las definiciones

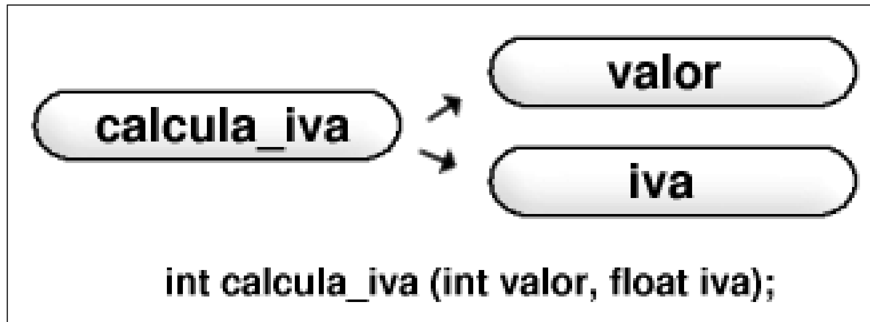


Figura 12.10: Esquema de `definition_type_args`: enlace a los argumentos

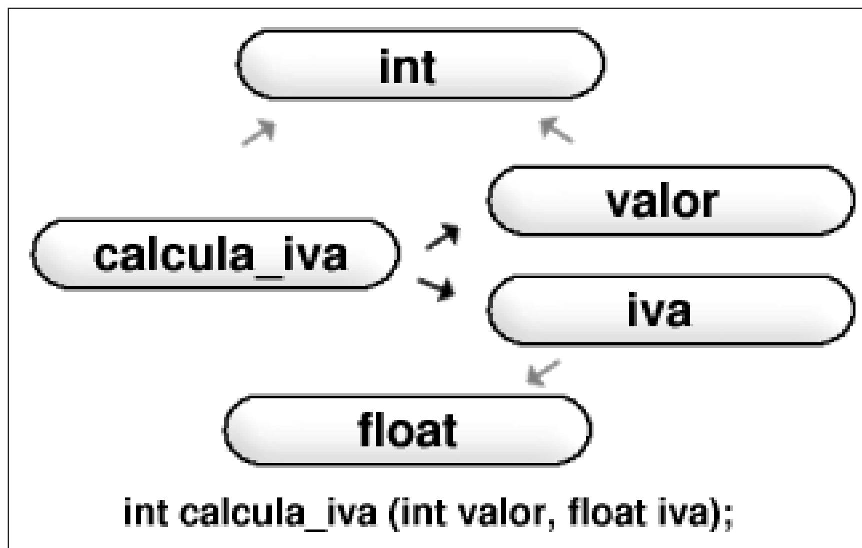


Figura 12.11: Esquema de los dos punteros, en gris `definition_type` y en negro `definition_type_args`

Capítulo 13

Funciones avanzadas

Aquí intentaré explicar, un poco por encima, como funciona todas las funciones avanzadas que implementa *CEdit* , algunas son sencillas y no entraré en ello, otras como el analizador ortográfico no entraré con todo lujo de detalles por la dificultad que supone.

13.1. Auto-Identación

La identación es simplemente colocar espacios al principio de cada linea, indicando a que profundidad se encuentra el bloque. Un ejemplo de identación sería:

```
if (var1>0)             // inicio de la condición IF
{                       // identación normal
    var2=funcion(var1); // hacemos algo
    while(var1);        // espero a que un milagro ponga var1 a 0
                        // excepción, aquí no va tab por el ;

    if (var2>var1)      // si var2 > var1
        return;         // volvemos (el if empieza un bloque,
                        //  pero solo tiene una instrucción)

#define PI 3,14         // una macro definiendo el número pi
    // comentario { traicionero
    else {              // sino
        var1=var2;      // var1 = var2
        return; }       // y volvemos
} funccion2();
```

Este pequeño ejemplo muestra las reglas que usa el programa para saber cuantos tabuladores (4 espacios) coloca, las reglas son:

- Si hay "{", la siguiente línea tiene un tabulador mas.
- Si hay "}':
  - Si es la primera letra, esta línea tiene un tabulador menos.
  - Si no es la primera letra, la siguiente línea tiene un tabulador menos.

Con estas sencillas reglas ya puedo calcular, respecto a la línea anterior, cuantos tabuladores tendría una línea.

Entonces lo que hago es calcular cada línea cuantos tabuladores tiene mas respecto a la anterior (en la estructura `line` la variable se llamaba `tabs`), NO INDICA LOS TABULADORES ABSOLUTOS, SINO RELATIVOS, esto me evita que si un tabulador cambia de 0 a 1, entonces `tabs` sera +1, no tengo que sumar 1 a todos los tabuladores hacia abajo en el fichero.

Entonces cada línea tiene el número de tabuladores relativos:

```
Tabs
 0   if (var1>0)              // Empieza en 0 tabuladores
 0   {
 1       var2=funcion(var1);  // 1 tabulador más
 0       while(var1);

 0       if (var2>var1)
 0       return;              // Excepción, la explico ahora

 0       #define PI 3,14
 1       // comentario { traicionero
 0           else {
 1               var1=var2;
 1               return; }    // -1 de esta
 -2      } funccion2();       // y -1 de esta = -2
```

El `comentario traicionero` me ha destrozado el esquema, para que esto funcione, los comentarios, textos y macros no han de tenerse en cuenta, esto significa que antes de identar se han de haber buscado los flags para poder saber de que tipo es cada letra y no contarlos

```
Tabs
 0  if (var1>0)              // Empieza en 0 tabuladores
 0  {
 1      var2=funcion(var1);  // 1 tabulador más
 0      while(var1);

 0      if (var2>var1)
 0      return;              // Excepción, la explico ahora

 0      #define PI 3,14
 0      // comentario { traicionero
 0      else {
 1          var1=var2;
 1          return; }        // -1 de esta
 -2 } funccion2();           // y -1 de esta = -2
```

Con el método explicado saldría el listado anterior, no es perfecto, ya que el `return;` de la línea 6, no ha quedado en su sitio. Esto se debe a que el retorno lleva un tabulador más porque la instrucción anterior (`if`) es un comienzo de un bloque, pero no lleva el carácter "{" porque solo se ejecuta una instrucción y no varias, esto lo trato como una excepción:

- después de `if, else, while o for` va un tabulador mas si no tiene un "{", o no tiene un ";", pero solo esa línea.

Para poder añadir solo un tabulador más a esta línea y no afectar a la siguiente tendría que hacer lo que acabo de explicar, un tabulador más en esta, pero uno menos en la siguiente.

El método utilizado es más sencillo, en esta excepción se usa una segunda variable (en la estructura `line` la variable se llamaba `add_tabs`). Esta variable indica cuantos tabuladores se han de añadir además de los que hay, y solo a esta línea. Dado que se tiene que definir esta variable solo para este caso, la he usado para que los comentarios tengan una tabulador mas:

```
Tabs Add_tabs
 0      0     if (var1>0)              // Empieza en 0 tabuladores
 0      0     {
 1      0         var2=funcion(var1);  // 1 tabulador más
 0      0         while(var1);

 0      0         if (var2>var1)
 0      1             return;          // Excepción

 0      0         #define PI 3,14
 0      1             // comentario { traicionero
 0      0         else {
 1      0             var1=var2;
 1      0             return; }        // -1 de esta
 -2     0     } funccion2();           //  y -1 de esta = -2
```

Con el método explicado ya tenemos casi todas las posibilidades, solo me falta el caso en que las macros han de ponerse sin ningún tabulador, para ello necesito una tercera variable (en la estructura `line` la variable se llamaba `absolute_tabs`), esta variable si es mayor o igual a 0 me indica cuantos tabuladores tiene esa línea, ignorando el valor que tenga `tabs` y `add_tabs`. El valor que tiene esta variable normalmente es -1, que significa que no ha de usarse.

```
Tabs Add_tabs Absolute_tabs
 0      0         -1        if (var1>0)
 0      0         -1        {
 1      0         -1            var2=funcion(var1);
 0      0         -1            while(var1);

 0      0         -1            if (var2>var1)
 0      1         -1                return;

 0      0          0        #define PI 3,14
 0      1         -1                // comentario { traicionero
 0      0         -1            else {
 1      0         -1                var1=var2;
 1      0         -1                return; }
 -2     0         -1        } funcion2();
```

Ahora el problema es, si yo cojo la línea del `else` ¿Cuantos tabuladores tiene?, la solución seria comenzar desde el principio e ir acumulando los contadores, cosa que seria absurda y lenta. La forma que uso es muy aceptable.

Los cursores se mueven por todo el fichero de línea en línea (lo han de hacer así por culpa de la distribución de líneas), por lo que cuando se mueven podrían ir sumando o restando la variable `tabs` de cada línea que vayan pasando, así ellos contiene el número de tabuladores de una línea en concreto.

13.2. Coloreado del texto

El coloreado de texto se hace en función de los flags que tiene cada letra. Como ya he explicado antes, los flags son como interruptores, pueden estar encendidos o apagados, los flags existentes los puedes ver en la figura 12.2.

Según el flag que esté activo, entonces la letra se coloreará de un color u otro por lo que el coloreado de texto se basa en rellenar la estructura de flags y luego marcar la letra (si ha cambiado alguno de sus flags) como cambiada para que se dibuje.

La función que rellena la estructura de los flags es `int cedit_set_flags (CEdit *cedit, line *l_line, long y);` , dada una línea, esta rellena la estructura de flags.

```
1 #include <stdio.h>    1 #include <stdio.h>
2     // Comentario     2     // Comentario
3 int a;                3 int a;
4 a = 2;                4 a = 2;
```

Figura 13.1: Diferencia a no usar coloreado a usarlo

13.2.1. cedit_flag_selected

Cedit_set_flags si está activado, indica que la letra está seleccionada para poder ser copiada o cortada, esta selección se va cuando el cursor se mueve o se corta el texto.

Este flag no es modificado por la función `cedit_set_flags`, ya que seleccionar todo el texto implicaría activar todos los flags del texto cada vez que se mueva la selección, en vez de esto, solo se rellena cuando se solicita un flag (`cedit_cget_line_flags`). El aumento de velocidad cuando se selecciona textos grandes es notable, y la perdida de rendimiento, ya que hay que mirar si la letra está seleccionada siempre, es ínfimo ya que si no hay selección la comprobación es inmediata.

Para seleccionar *CEdit* usa 6 variables, 3 para el punto inicial de la selección y 3 más para el punto final:

Tipo	Nombre	Descripción
long	**sel_ini_x**	Primer punto X de la selección
long	**sel_ini_y**	Primer punto Y de la selección
line*	**sel_ini_line**	Apuntador a la estructura línea
long	**sel_end_x**	Último punto X de la selección
long	**sel_end_y**	Último punto Y de la selección
line*	**sel_end_line**	Apuntador a la estructura línea

Figura 13.2: Variables de la selección

Si una letra está entre estos dos puntos, entonces `cedit_set_flags` está activado, de lo contrario no lo está.

13.2.2. cedit_flag_comment

`cedit_flag_comment` se activa cuando la letra pertenece a un comentario de múltiples líneas, estos comentarios comienzan con "/*" y finalizan con "*/", algunos ejemplos:

```
/* ------------
    Comentario
   ------------ */

/* Este es el comentario más usual */ Esto no está comentado
```

Para detectar este flag hay que mirar si encuentro un "/*", si es así,activa este flag de las siguientes letras hasta encontrar un "*/", estos inclusives.

13.2.3. cedit_flag_line_comment

Este es otro flag para otro tipo de comentario, comentarios de una sola línea. El anterior comentario funciona hasta encontrar "*/" sea cual sea la línea donde se encuentre, en cambio, este tipo de comentario empieza con "//" y es válido hasta el final de la línea, un ejemplo:

```
// Comentario { traicionero
```

Este se detecta de la misma forma que el anterior, si es encuentra un "//" en el texto, activa un este flag hasta el final de la línea.

Un comentario de este tipo dentro de un comentario normal es ignorado, ya que sino podría comentar texto que no lo está:

```
/* // Esto esta comentado */ Esto no
```

Todo lo que esté comentado (de los dos tipos) o sea texto literal es ignorado por todos los analizadores de *CEdit* .

13.2.4. cedit_flag_text

Este flag se activa cuando la letra pertenece a un texto literal (un array de caracteres), un ejemplo sería:

```
printf(‘‘hola’’);
printf(‘‘hola
        adios’’);      // los textos son multilínea
```

Este también se detecta igual, busco el carácter “ y a partir de hay el flag permanece activo mientras se continúa el análisis.

13.2.5. cedit_flag_apostrophe_text

Este es casi idéntico al anterior, se activa cuando la letra pertenece a un texto literal hecho entre apostrofes, en C sirven para representar un único carácter, aunque a veces un único carácter está representado por varias letras. Dos ejemplos típicos:

```
if (caracter==’a’)      // si carácter es la ‘‘a’’
if (caracter==’\n’)     // si carácter es un return
if (caracter==’\
              n’)      // ERROR, no es válido la multilínea
```

Se detecta buscando el signo ’ y finaliza o al encontrar otro o al finalizar la línea actual.

13.2.6. cedit_flag_macro

Las macros son parte del código que lo tiene que interpretar el preprocesador, su función es la de incluir código en el fichero, borrar texto o intercambiar un texto por otro, algunos ejemplos se pueden observar en la figura 13.3

Como se puede ver las macros comienzan con el signo “#” y siempre se colocan a la izquierda del texto, da igual la identación.

Si la primera letra de la línea es un “#” toda la línea se activa este flag.

```
#include <stdio.h>    // aquí se insertará el fichero stdio.h
#define PI 3,1416     // substituye PI por 3,1416
#ifdef PI             // si ha sido definido
   circulo=PI*radio; // esta línea se queda, sino ha sido
#endif                // definido, la línea es eliminada
```

Figura 13.3: Ejemplo de macros

13.2.7. cedit_flag_reserved_word

C define 31 palabras (no incluyo las palabras de C++ porque el proyecto se fija solo en C, a pesar de que reconocerá palabras de C++), la lista completa está en la figura 13.4.

Estas palabras son resaltadas con otro color para poderlas distinguir, su uso es muy frecuente, por eso reciben un trato especial.

Por cada letra se comprueba si pertenece a alguna de estos tipos (se comprueba que la letra inicie una palabra reservada) si lo inicia, entonces el flag `cedit_flag_reserved_word` se activa para toda la palabra.

Una palabra es válida si inmediatamente antes o después no es precedida por ninguna letra o número:

```
int contador; // int si es una palabra reservada
intranet(contador); // aquí int no lo es claramente.
```

char	double	float	int	long	short	signed	unsigned
void							
auto	extern	register	static	volatile			
break	case	continue	default	do	else	for	goto
if	return	switch	while				
enum	struct	typedef	union				
sizeof							

Figura 13.4: Palabras reservadas de C (no incluidas las de C++)

13.2.8. cedit_flag_number, cedit_flag_character y cedit_flag_mark

Estos tres flags se activan cuando la letra es un número, un carácter y un signo respectivamente.

13.2.9. cedit_flag_error y cedit_flag_warning

Este flag no es modificado en la función `cedit_set_flags`, sino que lo hace el sistema de búsqueda de errores, del cual hablaré más adelante. Estos flags representan una letra errónea o una letra que está mal colocada y es posible que falle respectivamente.

13.2.10. cedit_flag_block

Este flag está relacionado con el sistema de señalar el carácter asociado a un bloque, esta utilidad ya ha sido brevemente explicada, se trata de que cuando el cursor se coloque encima de un carácter {, [, (,),] o } se señala el otro carácter que cierra o abre el bloque, en el siguiente ejemplo se ve claramente:

```
file=(char*)malloc(strlen(file)*sizeof(char));
```

Figura 13.5: Ejemplo real de carácter asociado

Este flag no lo activa o desactiva `cedit_cget_line_flags`, sino que lo hace la función que busca el carácter asociado: `int cedit_csearch_block_flag (CEdit *cedit, int cursor)`

13.3. Auto-Formateado del texto

Con formateado del texto me refiero a colocar espacios de tal forma que la línea quede más clara y más fácil de leer. Para ello se coloca un espacio entre carácter y signo y entre signo y carácter (con carácter me refiero a número o letra), con esto ya tenemos casi el 90 % de los casos.

Este ejemplo es real y como se puede ver queda un poco junto todo:

```
def_db->file=(char*)g_malloc(strlen(file)*sizeof(char));
```

El formateado explicado lo pondría como:

```
def_db -> file = ( char *) g_malloc ( strlen ( file ) * sizeof ( char ));
```

NOTA: el signo "_" no es considerado como signo, sino que C lo considera como un carácter.

Como se puede ver queda demasiado abierto (pero ya casi está), la solución es poner alguna excepción:

- Después de un "(" o "[" o "{" no va espacio
- Antes de un ")" o "]" o "}" no va espacio
- Antes de "," no va espacio
- Antes de ";" no va espacio
- Antes de "." no va espacio
- Antes de ":" no va espacio
- Después de "*" o "&" no va espacio (para que quede *var o &var)
- Después de "!" no va espacio
- etc

Con estas excepciones se consigue casi el resultado deseado:

```
def_db -> file = (char *) g_malloc (strlen (file) *sizeof (char));
```

El casi es por culpa de `*sizeof`, esto es inevitable ya que en este caso el "*" se refiere a una multiplicación y debería de llevar espacio, pero este signo tiene un uso igual de frecuente y es el de apuntador a memoria, entonces su uso es `*variable` sin espacio. En este caso el usuario puede poner un espacio el manualmente y *CEdit* no se lo quitará.

El Auto-Formateado del texto solo quita los espacios antes de una línea y los que hayan después, pero no quita ningún espacio incluido por el usuario entre una línea, ya que considero que la función de todo esto es ayudar y no entorpecer, puede ocurrir que al formatear una línea se hiciese mal, (o no al gusto del usuario) y si quitase los espacios que hubiese introducido el usuario estaría cancelando el gusto de este.

El formateado se realiza SOLO cuando se cambia de línea y no mientras se escribe, esto es así para no entorpecer, ya que si mientras escribes te coloca espacios movería el cursor de una forma inesperada para el usuario estorbándolo, prefiero que se haga solo al salir de la línea (esto podría cambiarse en el futuro)

13.4. Señalar el carácter asociado

Para poder señalar el carácter asociado al inicio del bloque o al de final de bloque se realiza sencillamente recorriendo el texto desde donde está el cursor hasta encontrarlo, cuando se encuentre el signo "{", "(" o "[" aumento la profundidad si avanzo en el texto o la disminuyo si retrocedo en el texto, y si encuentro los caracteres contrarios ("}", ")" o "]") hago lo contrario. Cuando la profundidad sea negativa (si comienzo con profundidad 0) entonces tengo el carácter contrario.

Si corresponde con su tipo: "(" con ")", "{" con "}" y "[" con "]" entonces activo el flag `cedit_flag_block`, sino lo es, activo el flag `cedit_flag_error`.

```
file=(char*)malloc(strlen(file)*sizeof(char));
```

Figura 13.6: Ejemplo real de carácter asociado

13.5. Sistema de marcadores

El sistema de marcadores lo implementa *GtkCEdit* y se trata de dejar una marca en la línea actual para poder volver a ella cuando sea necesario.

Hay disponibles 10 marcas (del 0 al 9) y se pueden dejar en cualquier punto del texto.

Su implementación es sencilla, al dibujar una línea, si está en un marcador, es dibujada con un color de fondo distinto, además de poner un dibujo en la barra lateral con el número de la marca dejada.

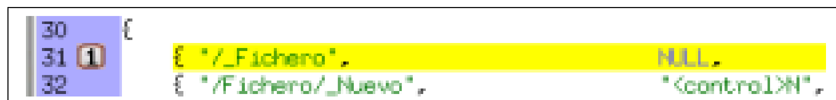


Figura 13.7: Ejemplo real de un marcador

Las variables usadas están en la figura 13.8.

Tipo	Nombre	Descripción
glong	**line_mark[10]**	línea donde se dejó un marcador
GdkPixmap*	**mark_pixmap**	dibujo de marcador

Figura 13.8: Variables usadas para los marcadores

13.6. Búsqueda de errores

La búsqueda de errores ocupa tanto como todo lo explicado hasta ahora por lo que la dificultad de crearlo ha sido enorme, y el otro problema que contiene es que *CEdit* toma que todo el fichero está mal a no ser que se encuentre lo contrario, por lo que hasta que no funcionó perfectamente no corregía nada.

Otra aclaración, la búsqueda de errores es solo léxico, solo dirá si una palabra es correcta o incorrecta, no mirará si esta usada en un contexto erróneo (no por ahora, pero tengo toda la información para poderlo realizar). Esto se debe a una decisión estratégica, al hacerlo así consigo más velocidad, porque el análisis es más sencillo y porque no buscaba hacer un corrector tan perfecto como el que tiene el compilador, sino algo que fuese una ayuda. Otra explicación de porque solo corrige las palabras es que mientras se escribe un programa está plagado de errores gramaticales (porque aún no se ha escrito todo el programa), pero no léxicos (todas las palabras que se escriben han de estar bien escritas). Además la dificultad para crear un analizador perfecto sería desbordante.

¿Porque no se ha usado un analizador externo, como FLEX o BISON?, simplemente porque haciéndolo yo conseguía la base de datos de definiciones, con lo cual podía implementar el sistema de Sugerencias y el Ir a definición. Además de que quería hacerlo yo (si hubiese querido usar programas externos, hubiese cogido un editor externo, en vez de crear *CEdit*).

La base para poder realizar esto es que C solo contiene 31 palabras (figura 13.4) pero un programa llega a utilizar miles. La explicación es que es C ofrece la oportunidad de hacerte tus propias funciones agrupando funciones ya existentes.

Para poder crear una nueva función o usar una nueva variable hay que declararlas, esto es escribir un texto muy específico que yo intento buscar y almacenar.

La búsqueda de errores entonces se basa en dos pasadas, en la primera *CEdit* se recorre todo el fichero buscando definiciones y almacenándolas en una base de datos, en la segunda pasada mira palabra por palabra, si está en la base de datos entonces es una palabra válida, sino lo está no lo es.

La primera dificultad es que recorrer todo el fichero es un proceso lento (un fichero, con la instrucción `#include ‘‘fichero.h’’` puede incluir otro fichero, y este a otros más) por lo que no se podía realizar el análisis cuando el usuario pulsaba una tecla, ni tampoco cuando cambiaba de línea ya que sino lo dejaría esperando y el retardo sería notable.

La solución es usar hilos. La explicación para alguien que no entienda

de informática es algo compleja, sería desdoblar el programa en dos tareas (escribir y analizar el texto) , de esta forma cada uno se dedica a una cosa.

Una ventaja más que me ofrecían los hilos (implementados con GLIB) es que puedo crear el segundo hilo sin prioridad, de esta forma el analizador solo se ejecuta cuando el ordenador no esta haciendo nada más importante, de esta forma el analizador no estorba ni enlentece las aplicaciones que se estén ejecutando.

La dificultad de los hilos es el compartir recursos, con lo cual también he tenido que usar semáforos ya que las líneas son compartidas entre el editor y el analizador, así evito que el editor borre una línea mientras que el analizador la intenta leer (esto es sobretodo el mayor problema).

Ahora explicaré los dos sistemas, el buscador de definiciones y el buscador de errores.

13.6.1. Buscar definiciones

He separado cada palabra que contiene un fichero de C en los tipos que se pueden ver en la tabla 13.9.

Nombre	Descripción
cedit_dw_nothing	palabras que no dan información: static, NULL...
cedit_dw_macro	palabras definidas en macros
cedit_dw_variable	variables comunes: contador, puntero...
cedit_dw_function	funciones comunes: calcular_iva();...
cedit_dw_typedef	tipos de datos nuevos: line, element_undo...
cedit_dw_data_type	tipos de datos estándares: int, char...
cedit_dw_struct	Estructuras, uniones, clases...

Figura 13.9: Tipo de palabras

Nada más comenzar ya hay definidas las palabras reservadas de C, entre ellas las más importantes son: char (carácter), void (nulo), int, long, float, short y double (numéricos) que son los tipos básicos para definir todas las demás (todas ellas son `cedit_data_type`)

Ahora voy palabra por palabra buscando indicios para crear las nuevas palabras e ir introduciendolas en la base de datos, según lo que me encuentre actuaré de una forma u otra. Ahora explicaré las posibilidades

Variables y funciones

Las variables son como contenedores donde se guarda un valor, todas tienen un nombre y un tipo (por ejemplo, empleados = 200 y su tipo es

numérico) y las funciones son agrupaciones de instrucciones para hacer el código más sencillo de leer (por ejemplo, `calcular_iva(precio)` calcula el iva de un precio), todas tienen definido un tipo (el valor que devuelve) y unos parámetros (el precio, es este ejemplo), su esquema es el siguiente:

```
data_type variable1, variable2...;
data_type nueva_función(data_type param1, data_type param2...);
```

Aquí se ve declaración de una variable y función, respectivamente, un ejemplo muy común sería:

```
  int contador,precio; // defino contador y precio de tipo entero
  int calcular_iva();  // función que calcula el iva,
                       //  devuelve un entero
```

Con lo cual esta parte busca los `cedit_data_type` y coge la siguiente palabra para definirla como variable o como función si la siguiente letra es un "(".

Esta es la idea, pero como siempre ocurre, esto solo funciona con el 75 % de los casos (y si no se hace bien al 100 %, esto no funciona), por ejemplo, este sistema falla cuando en el código se hace un cambio de tipo:

```
int contador;           // defino contador de tipo entero
char letra;             // defino letra de tipo carácter

letra=(char)contador;   // cambio el tipo de contador a carácter
                        //  para poderlo igualar
```

Este ejemplo haría que contador se declarase dos veces, como entero y como carácter, la solución es que si después del tipo viene un ")", entonces no se está declarando nada.

También falla con declaraciones complicadas:

```
unsigned long int contador; // declaro contador como
                            //  entero largo sin signo.
```

En este caso se declara `int` y `contador` de tipo `long`, este caso en realidad no es ningún problema, la redeclaración no afectará al encontrar errores ya que se busca que la palabra esté, si está dos veces, no importa. Pero para evitarlo la función que introduce una declaración en la base de datos, mira que no este ya declarada con el mismo tipo de palabra (figura 13.9), así se evita sobrecargar la base de datos.

Sobre el ámbito de las funciones, aunque en C solo hay dos tipos de ámbitos:

- Global: que se puede usar en cualquier lugar del texto
- Local: solo se puede usar en el mismo bloque

Yo lo generalizo de la forma explicaré para evitar problemas (en estructuras y funciones). Todas las definiciones tienen un ámbito, es decir, solo pueden ser usadas en un determinado lugar del código. Una función o variable solo es válido si está en en la misma profundidad o más profundo siempre en el mismo bloque:

```
int funcion() // profundidad 0, funcion es válido en profundidad >=0
{
   char var2; // profundidad 1, es válido en profundidad >=1

   var2=0;    // válido, ya que su profundidad es 1
}
var2=0;       // ERROR, su profundidad es 0 y el bloque es diferente

int funcion2()
{
   var2=0;    // ERROR, bloque diferente.
   funcion(); // válido, estoy en profundidad>=0 y el bloque
              // es el mismo (todo el programa)
}
```

Sobre las funciones tengo que decir que los parámetros de las funciones se consideran como variables normales (porque lo son) y que la función tiene un puntero a todas las variables que define (en `definition_type_args`) así podré más adelante comprobar el número de parámetros que se introduce en una función y el tipo de estos.

El ámbito de una función es calculado como el de una variable y la de sus parámetros depende de si es la declaración de una función o es la definición de la función:

```
    // Declaración de una función
int calcular_iva(int precio);    <- ámbito de la variable precio
                                    solo esta línea
```

Los límites de la variable precio será esa misma línea y nada más, si fuese la definición de la función, el límite inferior sería la primera línea de la función y el límite superior sería el final de la función:

```
    // Función
int calcular_iva(int precio)     <- límite inferior de la variable
{
   precio=precio*0.16;
   return precio;
}                                <- límite superior de la variable
```

Por lo que la variable precio será definida dos veces, una en la declaración de la función y otra en la definición de la función.

Estructuras

Cuando encuentre la palabra `struct` o `class` analizaré en forma de estructura. Las estructuras son uniones de funciones y variables en forma jerárquica, para poder acceder a un miembro de una estructura, primero hay que acceder al padre.

Esta será mi estructura de ejemplo:

```
struct texto             // estructura texto
{
    char *letras;        // texto en sí
    int  n_letras;       // número de letras guardada
    struct
    {
        int cursiva;     // si el texto esta en cursiva
        int negrita;     // si el texto esta en negrita
        int subrayado;   // si el texto esta subrayado
    } flags;             // instancia de la estructura
}
```

Es una estructura que guarda un texto, el número de letras y dentro tiene otra estructura llamada flags que guarda si el texto es cursiva, negrita y subrayado.

Primero se lee la palabra `struct` lo cual inicia el análisis en forma de estructura. La siguiente palabra es el nombre de la estructura, sino estuviese se la da un nombre genérico (`NoNameqmwnebrvtcyxuzi`), este nombre es incluido en la base de datos con el tipo `cedit_dw_struct` (véase la figura 13.9). Este tendrá un apuntador a todas las variables y funciones que pueda contener en el parámetro `definition_type_args`.

Ahora se analiza normalmente la estructura, buscando variables y funciones como si no estuviésemos en una estructura.

Si me encuentro con otra estructura anidada, pues la analizo primero (llamo a la misma función que está analizando la estructura: recursividad) y

así sucesivamente. Gracias al parámetro `definition_type` que guarda el tipo padre del que procede, guardo la jerarquía de estructuras.

Sobre la búsqueda de errores de las estructuras hablaré más adelante ya que no se podría usar ninguna variable definida fuera de la estructura (su ámbito está dentro de un bloque).

typedef

Con la palabra typedef se pueden crear nuevos tipos aparte de los que hay (`int, char, void, short, long, float y double`), por ejemplo, para definir el tipo booleano (si/no), sería hacer:

```
typedef booleano char;
```

A efectos de análisis, `booleano` será `char`, y nunca mejor dicho, porque `booleano` se guardará de tipo `cedit_dw_typedef` y al buscar la en la base de datos y encontrar que es de este tipo, será devuelto el padre (el que apunto `definition_type`): `char`, por lo que pasan a ser LA MISMA DEFINICIÓN.

enum

Las enumeraciones son equivalentes a definir todos sus miembros como globales y de tipo entero. Un ejemplo de enumeración:

```
enum
{
   OROS,
   SOTA,
   ESPADAS,
   BASTOS
};
```

Que es equivalente en *CEdit* a:

```
int OROS;
int SOTA;
int ESPADAS;
int BASTOS;
```

En C++ las enumeraciones son más complicadas, pero *CEdit* solo busca seguir el estándar de C.

definiciones

Las definiciones son un tipo especial de macros, su uso es la de substituir un valor por otro:

```
#define PI 3,1416
```

Con esta definición si se encuentra el preprocesador las letras PI será substituida por su valor: 3,1416.

Para el análisis solo me interesa que PI es una palabra válida (no analiza macros ni los resultados que generan), el analizador almacena `PI` como tipo `cedit_dw_define` y de ámbito global (todo el fichero), pero hay una excepción que me he encontrado:

```
#define entero int
```

En este caso me interesa mucho saber el valor de entero, ya que sino luego usarán:

```
entero precio;  // que es en realidad: int precio;
```

Si no substituyo `entero` por su valor: `int`, esta definición no la leerá. Para ello pongo una excepción: si después de el valor definido viene un tipo `cedit_dw_data_type` entonces almaceno el valor (en este caso `entero`) de tipo `cedit_dw_typedef` con ámbito global y de padre `definition_type` el tipo que representa `int`.

Esta excepción quizás es algo complicada de entender, pero lo que hace es simplemente definir `entero` como si fuese `int` y a efectos de análisis, `entero` será igual a `int` con todas sus propiedades (en realidad cuando encuentre `entero` y lo busque en la base de datos, devolverá `int`)

include

Los include son un tipo de macro que sirve para incluir un fichero dentro de otro. Lo mas común cuando se programa en C es hacer dos ficheros, uno con extensión .c que tiene el código del programa, y otro .h (header) con todas las definiciones que pueden usar cualquier otro programa que use el primer fichero.

CEdit cuando encuentra un `#include` lee el fichero que se incluye e intenta encontrarlo en las direcciones que previamente se les ha indicado (con la función `cedit_insert_include_path`), si lo encuentra comienza a analizarlo recursivamente (si encuentra otro `#include` también lo analiza)

13.6.2. Buscar errores

Ahora que ya tengo una base de datos (la tarea más complicada de todas) con todas las palabras posibles, buscar errores es bastante fácil.

CEdit se recorre el fichero palabra por palabra, si la palabra está en la base de datos (y su ámbito llega hasta la posición actual), es una palabra correcta, sino se busca que este definida sin tener en cuenta el ámbito, si la encuentro solo la señalo como warning.

Si la palabra es de tipo `cedit_dw_struct` la siguiente palabra puede que esté entre sus argumentos (`definition_type_args`) o que sea una nueva palabra.

Las macros son ignoradas, así como los textos literales y los comentarios.

13.7. Ir a definición

Gracias a que tenemos la base de datos con todas las definiciones posibles, al ir recogiéndolas también se ha guardado la línea donde están definidas y en que fichero (`n_line` y `file`), entonces el ir a definición muestra todas las entradas en la base de datos con esa palabra (puede haber varias, una como variable, otra como función, o redefiniciones...) y con el número de línea y fichero, esta información la da *CEdit* a *GtkCEdit* (este es el que muestra el menú emergente) y este a su vez se lo da a *gICE* , porque el es el encargado de abrir una nueva solapa para el nuevo fichero o mandar mover el cursor a la posición de la definición.

También la función que busca todas las definiciones busca cual de todas las posibles es la mejor de todas (según su tipo, el número de argumentos que tenga y en que fichero se encuentra), está única definición puede ser accedida con las teclas Ctrl+a.

13.8. Sugerir palabras

Con la misma razón se puede sugerir palabras, es solo mirar que palabra está escribiendo actualmente y recorrer toda la base de datos buscando todas las definiciones que comiencen con la palabra buscada. Solo se sugieren las palabras que la posición actual del cursor esté dentro del ámbito de la definición.

En el futuro se intentará que solo salgan las palabras que por su tipo o posición dentro del contexto actual sea válida (tengo toda la información, solo necesito hacerlo)

Parte V

Conclusiones

Capítulo 14

Hitos conseguidos

En realidad un editor de textos es la herramienta más común y más útil que brinda la informática, pero eso no significa que hacerlo sea una tarea fácil. Por esta razón los editores actuales buscan estabilidad y velocidad, olvidándose de opciones más complicadas y específicas.

- Estabilidad y robustez en el texto escrito: ante todo no se puede perder el trabajo realizado.
- Opciones básicas que cualquier otro editor posee y que *gICE* tenía que tener.
- Opciones avanzadas que no requieren grandes cantidades de proceso ni memoria, que pueden ser deshabilitadas si se desea y que buscan ayudar a la difícil tarea de programar.
- Búsqueda de errores, ahora no hay que probar a compilar y darse cuenta que se ha escrito mal una palabra (que es el error más común de todos).
- La base para poder continuar con el proyecto y poder explotar las nuevas posibilidades que me abre el analizador léxico.
- Y todo ello en un módulo (*CEdit*) que puede ser portado a otro sistema y entorno para conseguir otro editor de aspecto diferente pero con la misma funcionalidades avanzadas.

Capítulo 15

¿Y ahora que?

Aunque aquí no se ha terminado el trabajo dado que el tiempo ha sido escaso para un proyecto de esta magnitud, queda mucho por hacer en el futuro:

- *gICE* es solo el inicio del entorno, queda la integración con el compilador y con el debugger.
- La opción de crear proyectos y publicación de estos (CVS, autoconf, Makefiles, instaladores...)
- Ahora que tengo la base de datos con las definiciones, se puede crear un sistema de ayudas (referencia rápida) que se pueda saber los parámetros de las funciones y el tipo mientras se edita.
- Integración del Ir a definición y Sugerir palabra en el editor.
- Programación del entorno con GNOME y no con GTK, esto dará homogeneidad a la aplicación con las demás aplicaciones GNOME.
- Guardar la configuración con GCONF (es el registro de GNOME)

Capítulo 16

Conocimientos adquiridos

Por último quisiera explicar todo lo que he realizado por primera vez (y he tenido que aprender para la ocasión):

- Primer programa realizado en GTK, y dada la inminente llegada de GTK 2.0, usando las librerías de desarrollo.
- Primera aplicación gráfica programada por eventos.
- Primer widget creado (*GtkCEdit*).
- Primer uso de hilos y semáforos (parece difícil en la teoría, pero lo es más en la práctica)
- Primer programa que rompe la barrera de las 10.000 líneas (supera en 5.000 líneas mi record anterior)
- Primer documento realizado en LaTeX(este)
- Primer proyecto con código abierto, alojado en sourceforge (donde se alojan proyectos de categoría profesional).
- Primera página WEB usando PHP (http://gice.sourceforge.net)
- Primer programa realizado en Linux para múltiples plataformas.
- Primer proyecto final de carrera (y espero no tener que repetirlo)

Capítulo 17

Bibliografía y programas usados

Todo este proyecto, así como la documentación han sido desarrolladas con software libre, con lo cual, no ha sido violada ninguna ley de propiedad intelectual ni usado programas ilegales o de dudosa calidad, por lo cual *CEdit* , *GtkCEdit* y *gICE* también será de libre distribución.

17.1. Bibliografía

1. Wall et al, Kurt *Programación en Linux 2ª edición*. `Prentice Hall`

2. Hallow, Eric *Desarrollo de aplicaciones Linux con GTK+ y GDK*. `Prentice Hall, New Riders`

3. Hernández Orallo, Enrique y José *Programación en C++*. `Editorial Paraninfo`

4. Waite, Mitchell y Prata Stephen *Programación en C*. `Anaya Multimedia`

5. La página web de GTK/GDK/GLIB: `http://www.gtk.org`

6. La página web de GNOME: `http://www.gnome.org`

7. Referéncia rápida de LaTeX: `http://www.giss.nasa.gov/latex/ltx-2.html`

8. Y muchas más anónimas...

17.2. Programas usados

- Sistema Operativo: Linux 2.4.18-4mdk (Mandrake 8.2 y Red Hat 7.3)
- Editor para crear el programa: emacs 21.1.1, vi, gedit y *gICE*
- Editor de textos para esta documentación: LaTeX
- Compilador: gcc 2.96
- Debugger: gdb 5.1.1, kdbg 1.2.4, ddd 3.3.1
- Libreria gráfica: Gtk 2.0, Gdk 2.0
- Libreria de ayuda: Glib 2.0
- Libreria para ayuda al desarrollo: Electric Fence, Autoconf
- Programa para diseñar la página web: GIMP 1.2, emacs 21.1.1

Capítulo 18

Estadísticas

El proyecto no ha sido fácil, aqui están algunas estadísticas sobre el trabajo realizado:

- **horas trabajadas**: 4 horas/día.
- **líneas de este documentos**: 2500 aprox.
- **páginas web consultadas**: miles

Fichero	Bloque	Nº de líneas	Nº de carácteres	porcentaje
callback.c	*gICE*	581	1961	3,89 %
functions.c	*gICE*	759	3299	5,09 %
main.c	*gICE*	477	2079	3,20 %
preferencias.c	*gICE*	193	1014	1,29 %
language.h	*gICE*	19	37	0,12 %
main.h	*gICE*	254	648	1,70 %
gtkcedit.c	*GtkCEdit*	3291	14574	22,05 %
gtkcedit.h	*GtkCEdit*	233	940	1,56 %
cedit.c	*CEdit*	9579	39020	64,19 %
cedit.h	*CEdit*	447	2724	2,99 %
TOTAL	—	14923	66316	100 %

Figura 18.1: Estadísticas de longitud