Teoría N° 5

- Librería "*string.h*"
- Estructuración de datos con "*struct*"
- Edición de texto en Linux
- Compilación en Linux
- Depuración en Linux

Bibliotecas

Las bibliotecas son una colección de código (variables y funciones) que ya se encuentran programadas para poder utilizarse en un programa, facilitando la tarea del programador.

Es un módulo de software preconstruido, cuya utilización no es necesario conocer los detalles internos de su funcionamiento, sino su interfaz, es decir, que respuestas nos puede dar y cómo hay que preguntarle a la librería para obtenerlas.

Una biblioteca de C es una colección de funciones que realizan servicios esenciales y proporcionan implementaciones eficaces de operaciones que son utilizadas con frecuencia como: entrada/salida, herramientas para crear Interfaz Gráfica de Usuario, manipular cadenas de caracteres, conversiones de tipo, etc.

Pueden ser **estándar** (son distribuidas con el compilador) ó **no-estándar** (son codificados por terceros).

Biblioteca Estándar de C

La **Biblioteca Estándar de C** (conocida como ***libc***) es una recopilación de archivos cabecera y bibliotecas (o librerías) con funciones estandarizadas por el comité de la Organización Internacional para la Estandarización (*ISO*). Dicha biblioteca implementa operaciones comunes, por lo tanto, cualquier programa implementado en C se basa en la utilización de la librería estándar para funcionar.

Las principales son:

Biblioteca	Descripción
stdio.h	funciones para manipular datos de entrada y salida: *printf*, *scanf*, etc.
stlib.h	funciones para utilidades de uso general: *system, exit, etc*
string.h	funciones para manipular cadenas de caracteres: *strlen, strcmp, etc.*
math.h	funciones matemáticas: *sin, cos, pow, etc*
ctype.h	Clasificación de caracteres: alfabéticos, numéricos, imprimibles, etc.

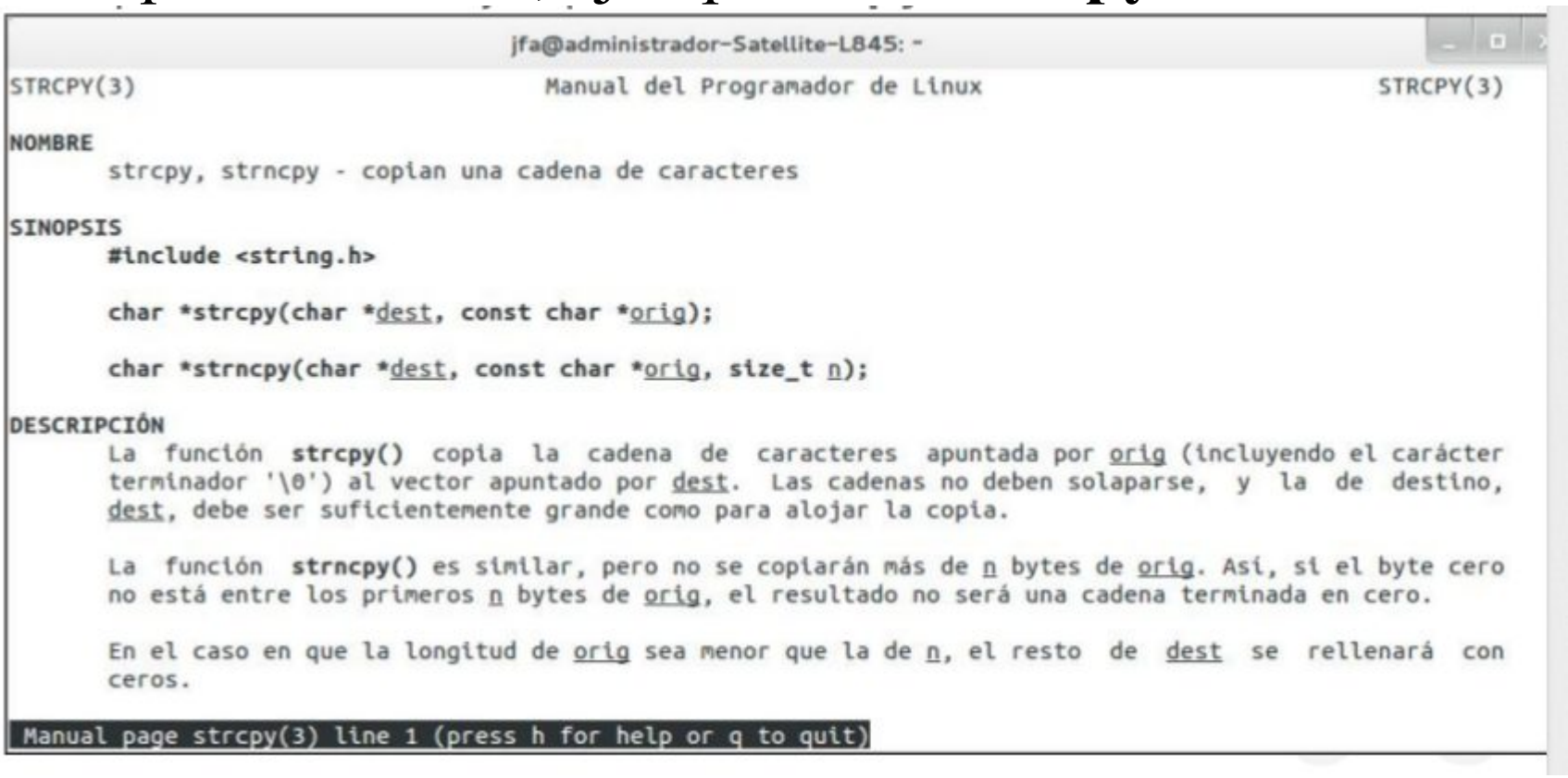
Biblioteca "*string.h*"

Provee funciones para manipular cadenas de caracteres.

Sintaxis	Descripción
char *strcpy(char *dest, const char *orig);	copia la cadena de caracteres apuntada por orig (incluyendo el carácter terminador '\0') al vector apuntado por dest.
char *strcat(char *dest, const char *src);	une la cadena src a la cadena dest sobreescribiendo el caracter '\0' al final de dest,
size_t strlen(const char *s);	Calcula la longitud de la cadena s, sin incluir el caracter terminador '\0'
int strcmp(const char *s1, const char *s2);	Compara las dos cadenas de caracteres s1 y s2. Devuelve un entero menor, igual o mayor que cero si se encuentra que s1 es, respectivamente, menor que, igual a (concordante), o mayor que s2.
char *strchr(const char *s, int c);	Devuelve un puntero a la primera ocurrencia del carácter c en la cadena de caracteres s.
char *strstr(const char *s2, const char *s1);	Encuentra la primera ocurrencia de la subcadena s1 en la cadena s2. Los caracteres de terminacion '\0' no se comparan.

Sintaxis de la función strcpy

Es posible solicitar la ayuda del manual al S.O. Linux de cualquier comando, ejemplo: **man strcpy**



Ejemplo con Funciones de cadenas de caracteres

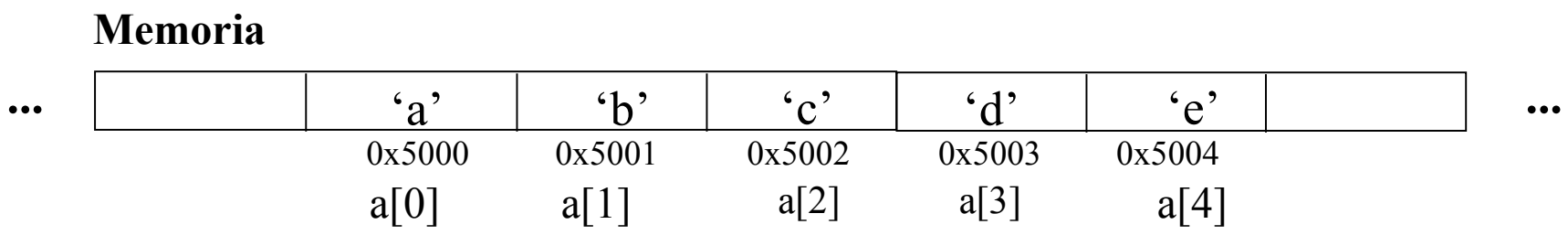
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
  /* Declaracion de variables */
  char palabra1[32], palabra2[32];

  printf("Ingrese la 1 palabra: ");
  scanf("%s", palabra1);
  printf("Ingrese la 2 palabra: ");
  scanf("%s", palabra2);
  printf("%s vs %s \n",palabra1, palabra2);
  printf("Iguales? %s\n");
  if  (strcmp(palabra1,palabra2)==0)
      printf("  SI \n");
  else
      printf("  NO \n");
  printf("Largos: %d y %d\n", strlen(palabra1), strlen(palabra2));
  printf("Concatenacion: %s\n", strcat(palabra1, palabra2));

  return (0);
}
```

Lenguaje “C” (parte 2): Estructuración de Datos

Arreglos:

- En él todos sus elementos **deben ser** del mismo tipo,
- Todos sus elementos son accesibles a través de un mismo nombre (es único dentro del ámbito) y definen una forma de acceder a cada una de la variables del “grupo” con un subíndice,
- Pueden ser locales, globales y parámetros,
- Cuando se declara un arreglo, sus elementos son creados en posiciones consecutivas de memoria,

```
char a[5] = {'a', 'b', 'c', 'd', 'e'};
```



Tipos de datos definidos por el usuario (I)

El usuario puede definir sus propios tipos de datos:

- » Mayor claridad,
- » Aumenta el significado semántico del código,
- » Simplificar declaración de variables.

Typedef

- Define un nuevo nombre para un tipo de dato.
- El nombre original sigue siendo válido.

```
typedef <tipo> <nuevo nombre>;
```

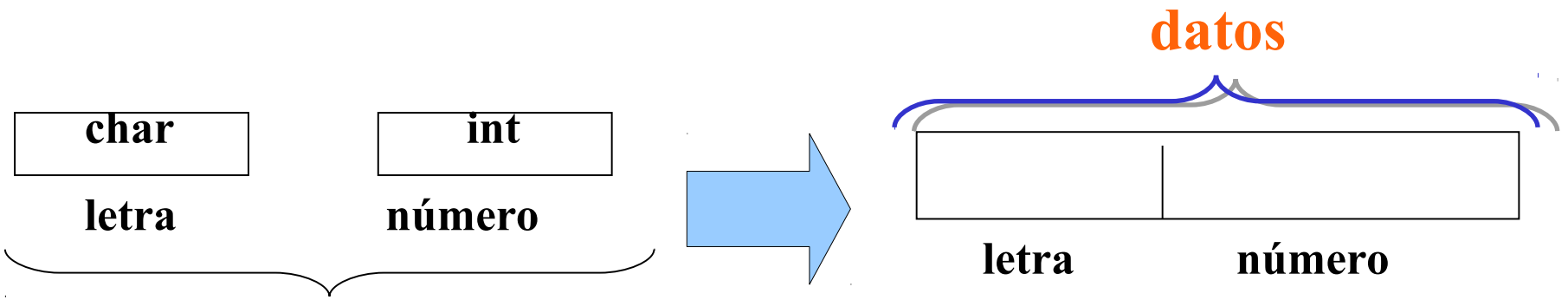
```
typedef int positivo;
```

Tipos de datos definidos por el usuario (II)

```c
#include <stdio.h>
#include <stdlib.h

typedef int positivo;
typedef int negativo;

int main(){

   positivo a,b;
   negativo c,d;

   a=1;
   b=2;

   c=-a;
   d=-b;

   printf("%d %d %d %d \n",a,b,c,d);
   return (0);
}
```

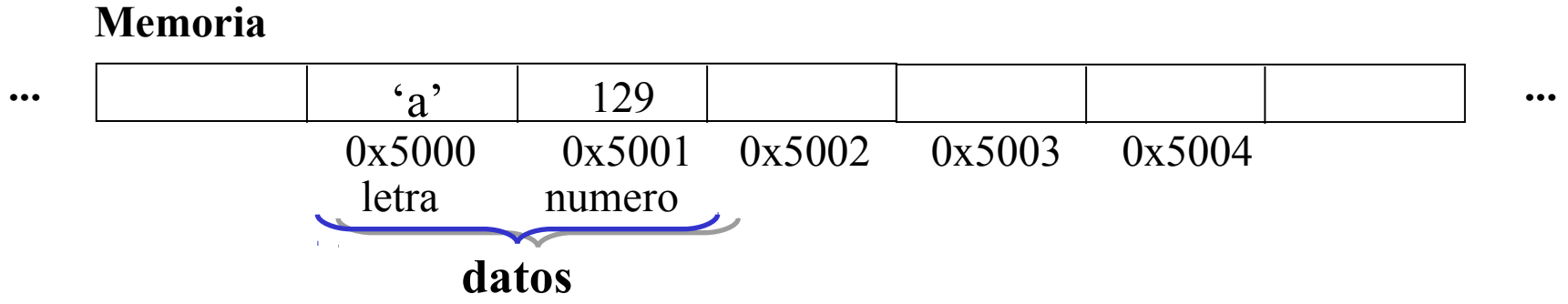
Tipos de datos definidos por el usuario (III)

- Otra forma de definir tipos de datos es componer varios datos simples en uno solo (esto se denomina **estructura**).
- Una estructura es un tipo de dato que contiene un conjunto de valores relacionados entre sí de forma lógica,
- Generalmente, se refiere a un concepto más complejo que un número o una letra,
- Conjunto de N elementos heterogéneos que están agrupados bajo un único nombre.
- Sus elementos **pueden ser** de diferentes tipos.
- Es un **tipo definido** por el programador.
- Todos sus elementos son accesibles a través de una combinación de nombres.
- Cuando se declara, sus elementos son creados en posiciones consecutivas de memoria.



antes: dos objetos independientes

ahora: un objeto con nombre único



Declaración de las Estructuras en C (I)

Se pueden declarar de 3 formas:

1) Se define la estructura y se declara la variable de tipo estructura al mismo tiempo.

```
struct  <nombre de la estructura> {
    <tipo> <nombre del campo> ;
    <tipo> <nombre del campo> ;
    ..........................
} <variable de tipo estructura> ;
```

puede ser de cualquier tipo
(char, int , float, etc)

Ejemplo:

```
struct alumno {
   char sexo;
   int  registro;
} luis, marta;
```

luis y **marta** son variables de tipo **alumno**

Declaración de las Estructuras en C (II)

2) Primero se define la estructura y después se declara la variable de tipo estructura.

```
struct <nombre de la estructura> {
    <tipo>   <nombre del campo>;
    <tipo>   <nombre del campo>;
    ..........................
};
```

Ejemplo:

```
struct alumno {
   char sexo;
   int  registro;
} ;

struct alumno luis,marta;
```

Luego,

struct <nombre de la estructura> <nombre de la variable> ;

Ventaja: pueden definirse otras variables del mismo tipo sin tener que repetir la estructura.

Declaración de las Estructuras en C (III)

3) Se declara un nuevo tipo de datos que se puede usar como cualquier otro tipo de datos predefinido en C.

```
typedef struct [<nombre de la estructura>]{
    <tipo> <nombre del campo>;
    <tipo> <nombre del campo>;
    ..........................
} <nombre del tipo estructura>;
```

Ventaja: independencia de la definición del tipo de datos y la declaración variables de ese tipo.

Ejemplo:

```
typedef struct {
  char sexo;
  int  registro;
} alumno ;

struct alumno luis;
```

Acceso a los campos de la Estructura en C

Formas de acceso a los campos de una estructura:

1. Mediante la variable de tipo estructura:

<variable de tipo estructura> . <nombre del campo>;

2. Mediante un puntero a la variable de tipo estructura:

<puntero a la variable de estructura> → <nombre del campo>;

A una estructura se le pueden asignar valores de múltiples formas:

1. Asignando valores en la declaración:

```
struct alumno luis = {'M',303896};
```

2. Asignado valores con sentencias de asignación a sus campos:

```
luis.registro = 303896;
```

3. Mediante operaciones de lectura a cada uno de sus campos:

```
scanf("%d",& luis.registro);
```

4. Por copia de una estructura idéntica:

```
luis = marta;  //asigna una estructura a otra
```

Ejemplo 1: almacenar información de alumnos

- A la antigua:

```
char nombreAlumno [64];
int edadAlumno;
double promedioAlumno;
```

- Con estructuras:

```
struct alumno {
        char nombre[64];
        int edad;
        double promedio;
};
```

Ejemplo 2: una variable de tipo estructura

```
#include <stdio.h>
int main() {
struct alumno {
  char sexo;
  int  registro;
} luis;
```

// Variable **luis** de "tipo estructura **alumno**",
// con 2 campos: uno char y otro entero.

```
  printf("Ingrese el sexo del alumno: ");
  scanf("%c", & luis.sexo);
  getchar();

  printf("Ingrese el registro del alumno: ");
  scanf("%d", & luis.registro);
  getchar();

  printf("El Sexo ingresado es: %c", luis.sexo);
  printf("EL Registro ingresado es: %d", luis.registro);
  return(0);
}
```

Pregunta: ¿Se pueden declarar varias variables del mismo tipo?

Ejemplo 3: dos variables del mismo tipo de estructura

```
#include <stdio.h>
int main() {
   struct alumno {
   char sexo;
   int  registro;
   };
   struct alumno var1 = {'F', 571843};
   struct alumno var2;


   printf("Ingrese el sexo para var2: ");
   scanf("%c", & var2.sexo);
   getchar();
   printf("Ingrese el registro de var2: ");
   scanf("%d", & var2.registro);
   getchar();


   printf("El Sexo y el Reg. de var1 es: %c - %d", var1.sexo, var1.registro);
   printf("El Sexo y el Reg. de var2 es: %c - %d", var2.sexo, var2.registro);
   return(0);
}
```

Definición del tipo estructura de nombre **alumno**
No existe reserva de memoria.

Variables **var1** y **var2** de tipo estructura **alumno**
Existe reserva de memoria !!!

Ejemplo 4: *Variable* "Arreglo de Estructuras"

```
#include <stdio.h>
int main() {
  struct empleado {
    int   codigo;
    float sueldo;
  };
  struct empleado todos[10];
  int ii;

  for (ii= 0; ii <=3; ii++) {
    printf("Empleado Nro. %d: ", ii);
    printf("Ingrese el código: ");
    scanf("%d", & todos[ii].codigo);
    getchar();
    printf("Ingrese el sueldo: ");
    scanf("%f", & todos[ii].sueldo);
    getchar();
  }
  printf("Datos del empleado Nro.2: ");      // Impresión de los datos
  printf("Código: %d", todos[2].codigo);     // del 2° empleado: Código
  printf("Sueldo: %d", todos[2].sueldo);     // Código y Sueldo
  return(0);
}
```

Definición del "tipo estructura", cuyo nombre es **empleado**

Arreglo **todos** de "tipo estructura" **empleado**

Ejemplo 5: Variable Estructura cuyo campo es un arreglo

```
#include <stdio.h>
int main() {
  struct empleado {
    int   codigo;
    char  nbre[10];
    float sueldo;
  };
  struct empleado un_empl;

  printf("Carga datos del Empleado \n\n");

  printf("Ingrese el código: ");
  scanf("%d", & un_empl.codigo); getchar();    // Ingreso del Código

  printf("Ingrese el sueldo: ");               // Ingreso del Sueldo,
  scanf("%f", & un_empl.sueldo); getchar();    // puede ser en cualquier orden


  printf("Ingrese el nombre: ");               // Ingreso del Nombre del
  scanf("%s", un_empl.nbre); getchar();       // Empleado

  printf("Nombre del empleado: ");             // Muestra del Nombre del
  printf("%s", un_empl.nbre);                  // Empleado
  return(0);
}
```

Tipo Estructura de nombre **empleado**

Variable **un_empl** de tipo estructura **empleado**

Estructuras y Funciones (I)

Para pasar ***miembros*** de una estructura a una función, se utiliza el mismo esquema de las variables comunes.

```
void mostrarNota(int cod){
    .....
};
int validarNota(int *cod)  {
   ....
};
int main (){
   struct empleado e1;
   ...
   validarNota(&e1.codigo);
   mostrarNota(e1.codigo);
   ...
}
```

Estructuras y Funciones (II)

Para pasar ***estructuras completas*** como parámetros se debe especificar el tipo completo de la estructura en la definición del parámetro.

```
void mostrarEmp(struct empleado e){
      printf("codigo: %d, sueldo: %f \n",
                        e.codigo, e.sueldo);
}
void inicializarEmp(struct empleado *e){
      (*e).codigo = 0;       // e->codigo = 0;
      (*e).sueldo = 100,90; // e->sueldo = 100,90;
      ...
}
…
  struct empleado e1;
  ...
  inicializarEmp(&e1);
  ...
  mostrarEmp(e1);
```

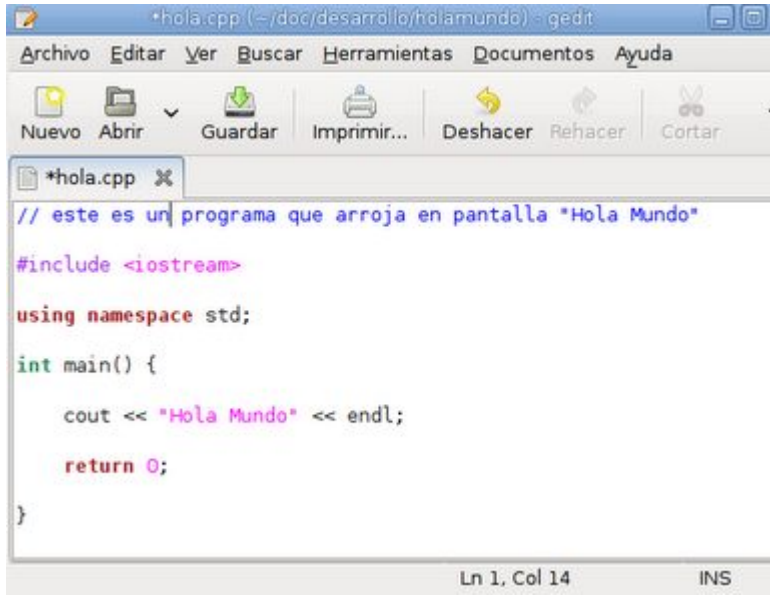
Edición de texto en Linux

Editor de Textos: es un utilitario destinado a la producción y manipulación de texto.

La manipulación de texto implica copiar, mover y cortar texto.

Linux posee numerosos editores de textos:

- **en modo Texto**: vi, joe, jove, emacs, etc..
- **en modo Gráfico**: kedit, kyle, gedit, xemacs, etc.

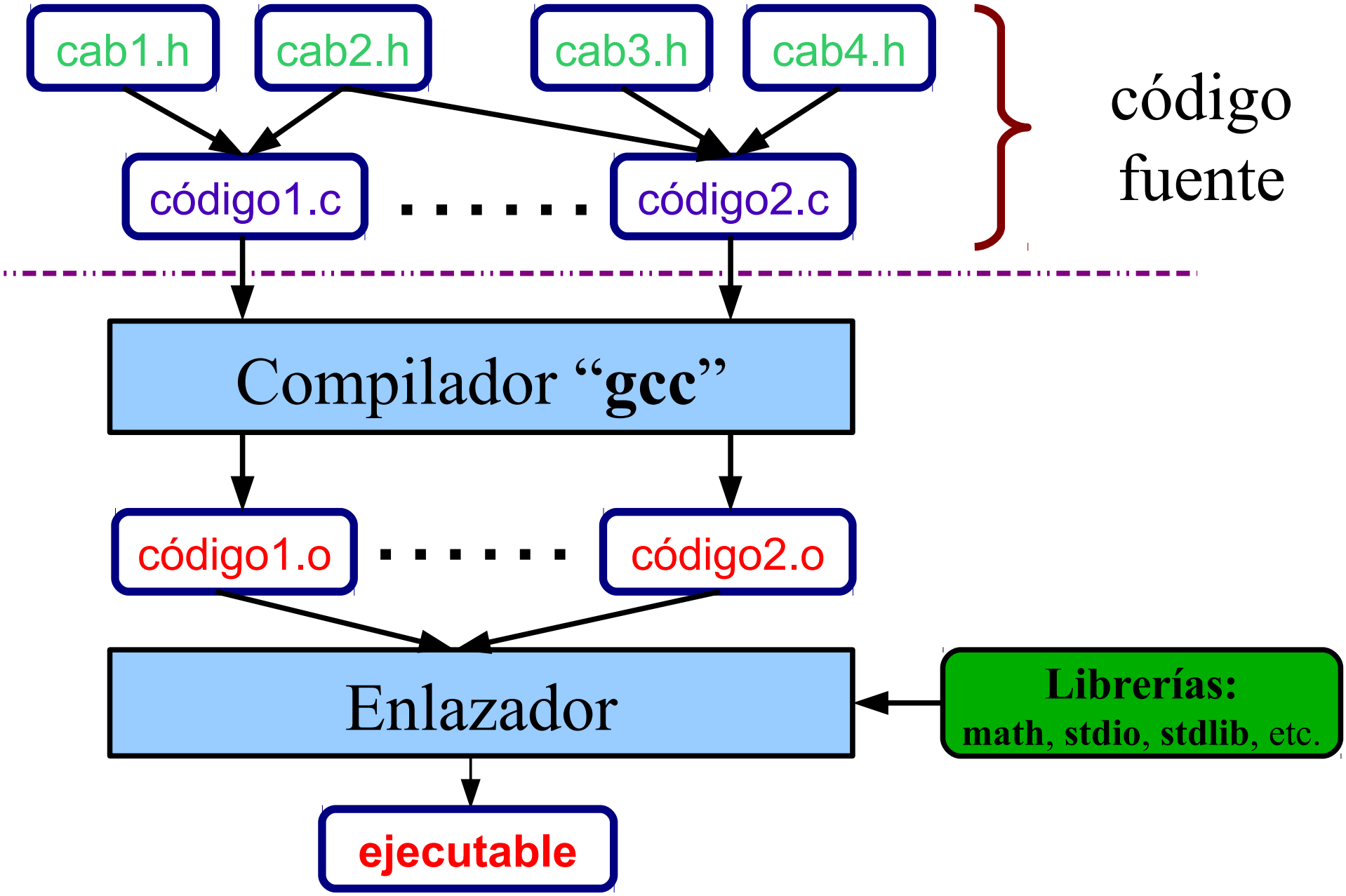


Las características más importante para los editores usados para programar son:

- Resaltado de sintaxis para varios lenguajes,
- Indentado automático de código,
- Auto complementado de código dependiendo del lenguaje,
- El área de edición se puede dividir en multiples vistas,
- Correctores ortográficos,
- Posibilidad de editar ficheros remotos vía FTP o SSH.

Compilación en Linux

- Los programas son texto puro sin formato (o formato *ASCII*),
- La generación de código debe realizarse con **editores de texto puro (ASCII),**
- Todo programa expresado en un lenguaje de programación debe ser traducido al lenguaje de máquina para que éste pueda luego ser ejecutado,
- La traducción la realiza un utilitario denominado: intérprete o **compilador**,
- En linux, el compilador más usado es "**gcc**".


cab1.h
cab2.h
cab3.h
cab4.h
código1.c
código2.c
código fuente
Compilador “gcc”
código1.o
código2.o
Enlazador
Librerías:
math, stdio, stdlib, etc.
ejecutable

La sintaxis general del compilador ***gcc*** en la línea de comandos es:

prompt$ **gcc [opciones] [nbre_archivo]**

código fuente a compilar en .c, .cc, .C, .s

- La forma más fácil de compilar es cuando se tiene todo el código fuente en un solo archivo. Esto evita el trabajo de sincronizar muchos archivos al compilar.

  prompt$ **gcc simple.c**

  Si la compilación resultó exitosa, se obtendrá un archivo ejecutable **"a.out"**.

- Si se desea que el compilador genere un archivo con nombre distinto a "a.out" lo especificamos con la opción **-o nombre_archivo_salida**:

  prompt$ **gcc simple.c -o simple**

- Si deseamos ejecutar el código paso a paso necesitamos que el compilador inserte información en el código ejecutable para que ayude al depurador en su tarea (**-g**):

  prompt$ **gcc -g simple.c -o simple**

- Podemos decirle al compilador que nos de más avisos (warnings) para mejorar la calidad de nuestro código fuente (**-Wall**):

  prompt$ **gcc -Wall -g simple.c -o simple**

- Es posible separar el código en varios archivos (**simple.c** y **principal.c**):

  prompt$ **gcc simple.c principal.c -o princ**

Depuración en Linux

Todo código necesita ser chequeado para verificar que efectivamente realiza lo que se desea. Usualmente es necesario poder "visualizar" lo que está sucediendo con el contenido de las variables durante la ejecución de un programa.

Un depurador (o "debugger") permite observar cómo funciona un programa en desarrollo durante su ejecución. Estas herramientas pueden trabajar en modo texto o gráfico.

Linux provee varios depuradores:

- ***gdb:*** es un depurador se ejecuta en modo Ttexto.
- ***ddd:*** depurador con una interfaz gráfica para el depurador gdb.

La sintaxis general del depurador ***gdb*** en la línea de comandos es:

prompt$ **gdb [opciones] [nbre_archivo]**

Archivo de código ejecutable

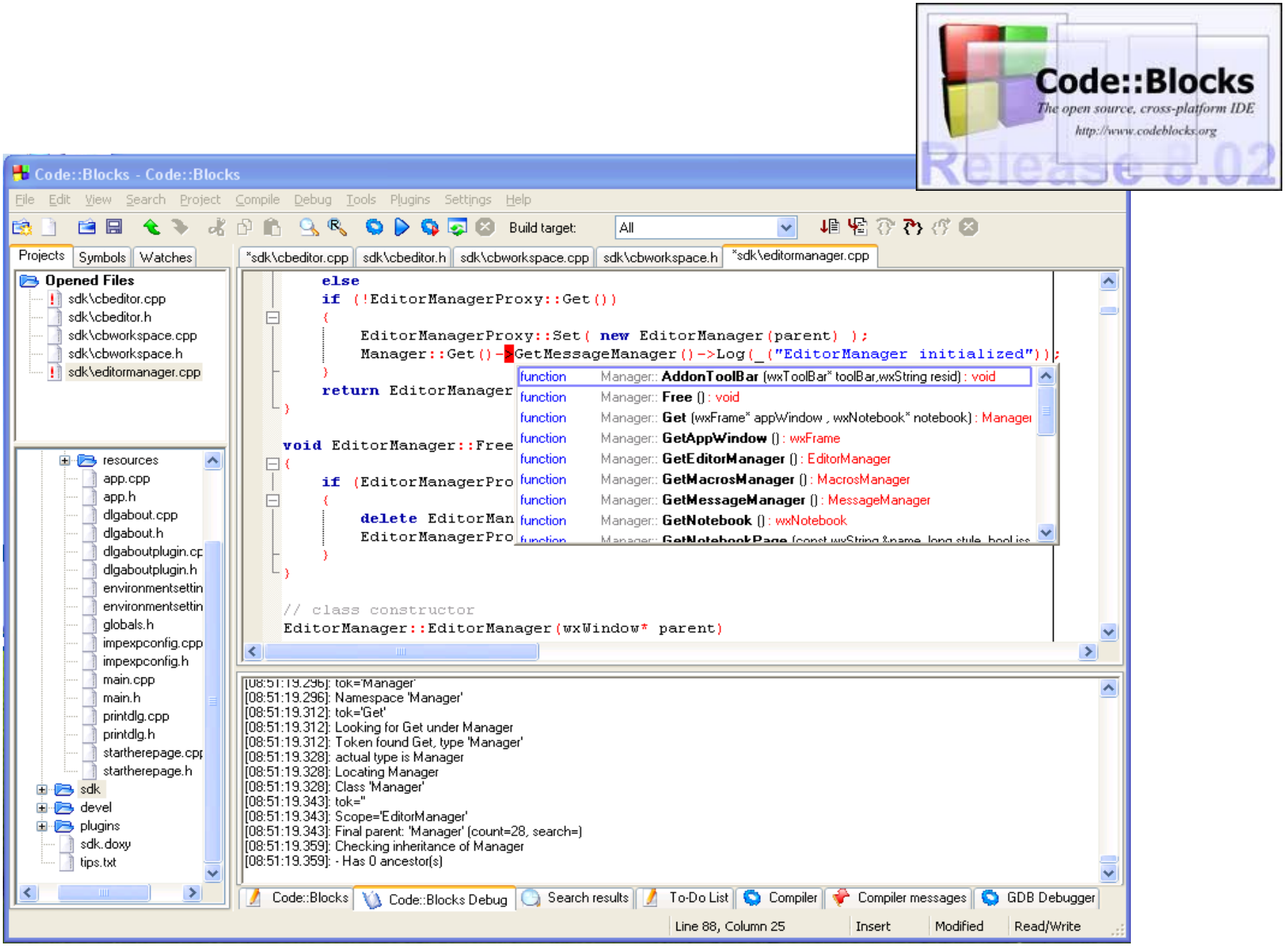
Una vez cargado el binario lo deja dispuesto para comenzar su ejecución, esperando alguna orden de tipo *gdb* (break, list, display, run, next, step, quit, etc.):

(gdb) help // muestra una pequeña explicación de cada una de ellas.

Entorno de Desarrollo Integrado (IDE)

El IDE es un programa informático compuesto por un conjunto de herramientas de programación.
Puede dedicarse en exclusiva a un sólo lenguaje de programación o bien, pueden utilizarse para varios lenguajes.
Consisten generalmente en un **editor de código**, un **compilador**, un **depurador** y un **constructor de interfaz gráfica (GUI)**
Los más conocidos en Linux son:

- KDevelop
- Anjuta
- **Code::Blocks**


Code::Blocks
The open source, cross-platform IDE
http://www.codeblocks.org
Release 8.02
Code::Blocks - Code::Blocks
File Edit View Search Project Compile Debug Tools Plugins Settings Help
Build target: All
Projects Symbols Watches
Opened Files
sdk\cbeditor.cpp
sdk\cbeditor.h
sdk\cbworkspace.cpp
sdk\cbworkspace.h
sdk\editormanager.cpp
*sdk\cbeditor.cpp sdk\cbeditor.h sdk\cbworkspace.cpp sdk\cbworkspace.h *sdk\editormanager.cpp
else
if (!EditorManagerProxy::Get())
{
EditorManagerProxy::Set(new EditorManager(parent));
Manager::Get()->GetMessageManager()->Log(_("EditorManager initialized"));
}
return EditorManager
}
void EditorManager::Free
{
if (EditorManagerPro
{
delete EditorMan
EditorManagerPro
}
}
// class constructor
EditorManager::EditorManager(wxWindow* parent)
function Manager:: AddonToolBar (wxToolBar* toolBar,wxString resid) : void
function Manager:: Free () : void
function Manager:: Get (wxFrame* appWindow , wxNotebook* notebook) : Manager
function Manager:: GetAppWindow () : wxFrame
function Manager:: GetEditorManager () : EditorManager
function Manager:: GetMacrosManager () : MacrosManager
function Manager:: GetMessageManager () : MessageManager
function Manager:: GetNotebook () : wxNotebook
resources
app.cpp
app.h
dlgabout.cpp
dlgabout.h
dlgaboutplugin.cp
dlgaboutplugin.h
environmentsettin
environmentsettin
globals.h
impexpconfig.cpp
impexpconfig.h
main.cpp
main.h
printdlg.cpp
printdlg.h
startherepage.cpp
startherepage.h
sdk
devel
plugins
sdk.doxy
tips.txt
[08:51:19.296]: tok='Manager'
[08:51:19.296]: Namespace 'Manager'
[08:51:19.312]: tok='Get'
[08:51:19.312]: Looking for Get under Manager
[08:51:19.312]: Token found Get, type 'Manager'
[08:51:19.328]: actual type is Manager
[08:51:19.328]: Locating Manager
[08:51:19.328]: Class 'Manager'
[08:51:19.343]: tok=''
[08:51:19.343]: Scope='EditorManager'
[08:51:19.343]: Final parent: 'Manager' (count=28, search=)
[08:51:19.359]: Checking inheritance of Manager
[08:51:19.359]: - Has 0 ancestor(s)
Code::Blocks Code::Blocks Debug Search results To-Do List Compiler Compiler messages GDB Debugger
Line 88, Column 25 Insert Modified Read/Write