Organización y documentación de programas grandes en 'C' (ansi)

Victor Manuel Toro C.
Grupo "Informatica Fundamental e Ingenieria de Software FIDIAS"
Departamento de Sistemas y Computacion - Universidad de los Andes
Apartado Aereo 4976
Bogota - COLOMBIA
(e-mail: vmtoro@andescol.bitnet • Telex: 42343 UNAND CO • Fax: (+57) (1) 2841590)

RESUMEN

El desarrollo de programas grandes requiere establecer de antemano normas claras de particion, organizacion y documentacion. Esto es aún mas patente si el desarrollo se va a hacer en grupo. Así mismo, ceñirse a un conjunto claro de normas es requisito imprescindible para la añorada re-utilizacion de codigo. De otra parte, la posterior, costosa e inevitable labor de Mantenimiento puede ser grandemente beneficiada si a lo largo de toda la vida del programa se ha seguido algun estandar claro y conciso de organizacion y documentacion. Este articulo presenta una propuesta de estandar para organizacion y documentacion de programas grandes en C (Ansi) que ayuda a hacer mas agil la labor de desarrollo y mas certera la labor Mantenimiento.

CLASIFICACION ACM

D.1.4 Software/Programming Technique/Sequential Programming
D.2.3 Software/Software Engineering/Coding
D.2.7 Software/Software Engineering/Distribution and Maintenance

INTRODUCCION

> We must change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct the *computer* what to do, let us concentrate rather on explaining to *humans being* what we want the computer to do.
>
> Donald E. Knuth [KNUTH 84]

A lo largo de su corta historia, la Programacion ha sido vista como **Arte**, posteriormente como **Disciplina** y mas recientemente como **Ciencia**. El reto actual es que en los proximos años la programacion sea percibida como una verdadera **Ingeniería**.

En efecto hace unos 20 años se la consideraba un **Arte** ("The Art of Computer Programming: Fundamental Algorithms", [KNUTH 68]). Constaba de un conjunto de conocimientos aislados logrados por inspiracion o genialidad de sus inventores sin una teoria o lenguaje que los unificara. Solo era suceptible de ser aprendida por imitacion. Desarrollar un programa era un enorme esfuerzo de imaginacion, que eventualmnte convergia a algunas ideas utiles cuyo origen o correcion era muy dificil de justificar.

Años despues se calificaba a la Programacion como **Disciplina** ("The Discipline of Programming", [DIJKSTRA 76]) se habian logrado identificar algunos patrones y esquemas unificadores dentro de los cuales se podia realizar una argumentacion ordenada y metodica. Se disponia de una serie de ejemplos y casos primorosamente resueltos, los cuales se ensenaban a los estudiantes por su gran valor formativo. Sin embargo en la practica rara vez se podian aplicar tan elegantes y rigurosos metodos, pues la incipiente teoria de programacion era excesivamente formal (i.e. llena de formulas) y compleja.

Desde comienzos de los 80 la Programacion se esta empezando a enfocar como **Ciencia** ("The Science of Programming", [GRIES 81]), es decir, sustentada por principios y teorias generales que permiten no solo explicar el porque de sus diferentes aspectos, sino ademas, y mas importante aun, que orientan clara, efectiva y eficientemente el proceso mismo de diseño [MEYER 88][TORO 87, 91] y construccion de software [CARDOSO 90]. Inclusive, ya se vislumbra una etapa de madurez en la que preservando intacto el rigor se logra una razonable libertad e independencia del formalismo.

Para que en los proximos años la Programacion sea realmente una **Ingeniería** es necesario aumentar drasticamente la eficiencia y calidad del proceso de desarrollo de programas. Para lograr esto es requisito afinar y estandarizar metodos para las etapas anteriores a la Programacion (Analisis, Diseño, Especificacion), estandarizar la organizacion y la documentacion de programas, fijar criterios de calidad claros y verificables, y desarrollar herramientas que ayuden a aplicar dichas metodologias, y a ayudar/verificar el cumplimiento de los estandares de calidad, organizacion y documentacion [YUDKIN 88]. La juiciosa aplicacion de dichos metodos, estandares y herramientas deberia facilitar la posterior, importante, costosa e inevitable labor de Mantenimiento [SCHWARTZ 82] [EINBU 88].

Este articulo presenta una propuesta de estandar para organización y documentacion de programas medianos y grandes en Ansi-C [K&R 88]. El estándar planteado da cabida a los avances en especificacion y correccion de software [GRIES 81] [CARDOSO 90], es compatible con cualquier metodo de Diseño (particularmente con el de las Jerarquias de Tipos Abstractos de Datos [TORO 87, 91]), y es susceptible de instrumentalización con herramientas concretas y simples. El articulo empieza entonces discutiendo el verdadero tiempo de vida de un programa. Posteriormente se resalta la necesidad de fijar y usar estandares de organizacion y documentación de programas. A continuacion se plantea la estrategia de uso del estandar propuesto. Finalmente se presenta el estandar de organizacion y documentacion. Dicha presentación se hace en tres etapas: inicialmente se describe el estandar para programas C en un solo archivo fuente, luego se presenta el estandar para programas C en varios archivos fuente, y finalmente se sugieren algunas normas generales (indentación, nomenclatura, ordenamiento).

EL TIEMPO DE VIDA DE UN PROGRAMA

> los programas realmente buenos viven para siempre y nunca terminaran de escribirse, al menos mientras dure el hardware e incluso mas
>
> Charles Simonyi (autor de Multiplan, Word, Excel) [LAMMERS 86]

En las primeras etapas de su vida estudiantil muchos programadores creen que los programas correctos y eficientes nunca necesitaran mantenimiento. Sin embargo al menos tres razones importantes desvirtuan la anterior ilusion. Por un lado, las organizaciones en las cuales existen dichos programas y los usuarios en general, normal e inevitablemente evolucionan, generando nuevos procedimientos, produciendo nuevos datos, requiriendo nueva informacion. Por otro lado, los rapidos avances del hardware, pantallas, sistemas operacionales, manejadores de bases de datos, etc., periodicamente ponen en manos del programador facilidades y posibilidades que seria absurdo no aprovechar tanto en sus nuevos programas como en los ya existentes. Y finalmente, la busqueda de reutilizacion de software requeriere volver varias veces sobre el mismo codigo para ajustarlo o generalizarlo. El Mantenimiento de Software es entonces una importante, necesaria, e inevitable labor.

En realidad y desafortunadamente, en muchos casos el idilico panorama que plantean los libros de Ingenieria de Software [PRESSMAN 87], no se puede llevar a cabo. Por diverso tipo de limitaciones, muchas veces no es posible adelantar concienzudamente las etapas de planeacion, analisis de requerimientos, especificacion formal de los requerimientos, diseño global, especificacion formal del diseño y programacion. No pocas veces solo se realiza superficialmente el analisis y el diseño y se entra en un ciclo programacion - correccion/ajustes - programacion - correccion/ajustes -.

La consecuencia es evidente: muchos departamentos de informatica terminan dedicando el grueso de sus recursos humanos a labores de mantenimiento y solo una fraccion a desarrollo de nuevas aplicaciones. Con el tiempo se llega a que en la mayoria de los sistemas grandes de software el costo y cantidad de trabajo acumulado en labores de Mantenimiento ha sobrepasado muy ampliamente el costo y cantidad de trabajo del Desarrollo original [BOEHM 84].

Obviamente la 'culpa' de esta situacion no radica unicamente en el no uso de estandares de programacion y documentacion. Sin duda la mayor parte del problema radica en que los metodos ofrecidos por la Ingenieria de Software son aun incapaces de enfrentar el caracter frecuentemente dinamico y volatil de los requerimientos y diseños.

De todas maneras, disponer y utilizar consciente y profesionalmente un estandar de organizacion y documentacion de programas contribuye significativamente a aclarar el panorama de programacion-mantenimiento de aplicaciones grandes.

IMPORTANCIA DE UN ESTANDAR DE DOCUMENTACION Y ORGANIZACION

> La profesora: a ver Montecristico: un señor compro una gallina en $1.721,35 y la vendio en $1.514,80 ¿ El señor gano o perdio en el negocio ?.
>
> Montecristico: El señor perdio en los pesos y gano en los centavos.
>
> Montecristo - Humorista Colombiano

La mejor documentacion para un programa es el programa mismo. Toda la demas documentacion (manuales de usuario, manuales de referencia, manuales de mantenimiento, documentos de diseño, etc.) desafortunadamente termina por desactualizarse. Ante cualquier labor de mantenimiento el documento que realmente se consulta es el listado mismo. La

demas documentacion usualmente solo es util para que el responsable de mantenimiento se familiarice con las caracteristicas generales del programa que debe mantener ¡EINBU SS¡.

En consecuencia, es de crucial importancia organizar los programas de una manera adecuada y proveerlos de comentarios e informacion estructural apropiados. Esta organizacion y la documentacion debe hacerse siguiendo esquemas y patrones claros y estandares De lo contrario lo que se crearia seria aun mas variedad y desorden.

Sin embargo organizar y documentar profesionalmente un programa es una labor que tambien requiere tiempo experiencia y conocimiento, casi tanto como escribir buenos algoritmos !. La dificultad principal radica en que la documentacion de un programa se hace fundamentalmente en lenguaje natural (español, ingles, etc.). Y es un hecho comprobado que el lenguaje natural es tanto o mas dificil de usar que C ó Pascal, pues su sintaxis y semantica es obviamente menos rigurosa !.

El programador tendra entonces que dedicar su tiempo tanto a escribir las instrucciones que materialicen los algoritmos y estructuras de datos que tiene en mente, como a organizar y comentar adecuadamente su programa. Sin embargo, en la practica habitual casi siempre ocurre que la organizacion y documentacion de un programa solo se realiza en forma superficial y apresurada, despues de que el programa "funciona". Los programadores suelen creer que dedicar tiempo a documentar, a organizar, a escoger nombres apropiados para las variables, etc., es en buena medida desperdiciar su tiempo y retrasar la terminacion del programa. Inclusive es frecuente ver programadores que sienten remordimiento por el desperdicio de papel o los bytes extra que implica dejar algunas lineas en blanco para separar las diversas partes del programa Muchos programadores solo se sienten "produciendo" cuando estan escribiendo instrucciones, y piensan que las labores de organizacion y documentacion son requisitos molestos y practicamente inutiles.

Nuestra experiencia directa, y particularmente la de muchos programadores realmente expertos ¡LAMMERS Só.., es que es mucho mas eficiente organizar y documentar de acuerdo a algun estandar fijado de antemano, y que esta labor debe hacerse *al momento mismo en que se va escribiendo el programa*. Seria temerario afirmar que el tiempo total de desarrollo se acorta significativamente. Pero si podemos asegurar sin ninguna duda que se obtiene mayor calidad y que se sientan bases firmes para un mantenimiento mas agil y certero El tiempo adicional necesario para la adecuada documentacion y organizacion o el papel o bytes extras que se requiere para mejorar la apariencia de un listado son los "centavos" que se mencionaban al comienzo de esta seccion. Los "pesos" son la calidad del producto y el tiempo de mantenimiento

PRACTICA DEL ESTANDAR

muchas cosas estaban prohibidas y las que no estaban prohibidas eran obligatorias ..."

G Garcia Marquez - "El Otoño del Patriarca"

Son varios los objetivos buscados al partir, organizar y documentar profesionalmente el codigo fuente:

- Descentralizar tanto como se pueda el desarrollo de cada uno de los modulos del programa garantizando al mismo tiempo la coherencia global
- Facilitar la ubicacion de las diferentes secciones que componen un programa fuente grande, resaltando su estructura global
- Plasmar en el texto mismo del programa la informacion y las aserciones necesarias para facilitar la escritura de codigo correcto
- Proveer la informacion necesaria para que futuros programadores de mantenimiento puedan realizar su labor agil y certeramente

Hay que evitar extremos por defecto, no escribiendo comentarios suficientemente detallados, o por exceso escribiendo comentarios innecesarios o redundantes. La documentación debe ser una información precisa y breve, que se trasmite entre programadores altamente profesionales y competentes. La medida ideal podría ser entonces que el programador se planteara a sí mismo la siguiente pregunta: ¿ cuál es la mínima información necesaria para que yo mismo pudiera modificar este programa dentro de cinco años ?. ¡ Esa información mínima es la que hay que incluir en los comentarios !. Y ahora solo resta establecer una forma estándar de realizar las anteriores labores.

Es inutil intentar legitimar un estándar demostrando que cada una de las normas que lo componen es superior a cualquier otra norma posible [EINBU 88]. La estandarización es valiosa por sí misma. En la práctica es mejor acogerse a un solo estándar, posiblemente imperfecto, a debatirse entre varios candidatos a estándar (puntualmente mejores !), segun el responsable de turno.

El estándar que aquí se plantea esta compuesto por una serie de recomendaciones generales, un esquema de organización de programas C en un archivo fuente, y un esquema de partición de programas C grandes en varios archivos fuente. Para facilitar la aplicación del estándar se han escrito una serie de plantillas, en cada una de las cuales se proveen todos los títulos y campos para comentarios, lo mismo que los `#include` necesarios para poder acoplar los diferentes archivos fuente. Existe entonces una plantilla por cada uno de los bloques o modulos en que se debe descomponer un programa C de acuerdo al estándar planteado

Todas estas plantillas deberán ser colocadas en un directorio publico, al cual tengan acceso todos los programadores. El proceso normal de programación de acuerdo al estándar consiste típicamente de

- Empezar a editar un archivo inicialmente vacío.
- Leer desde el editor la plantilla correspondiente al contenido de dicho archivo (definición de constantes, declaración de prototipos de funciones, definición de variables globales, etc.)
- Diligenciar los campos que provee la plantilla hasta el nivel de detalle que el programador considere adecuado
- Eliminar los campos que el programador no considere necesarios
- A lo largo de la escritura del archivo fuente el programador podrá ir incorporando a su texto otras plantillas para documentar las nuevas funciones que vaya creando, hacer aserciones parciales o globales, dejar registro de modificaciones, etc.

LA FUNCION: UNIDAD BASICA DE UN PROGRAMA

Cada función esta precedida por un encabezado donde se presenta su descripción. La plantilla estándar corresponde a la necesidad maximal de documentación. Obviamente la mayoría de las funciones no requiere una documentación tan detallada como la que plantea dicha plantilla. Tal como se dijo anteriormente, hay que evitar extremos de sobre-documentación, o de documentación insuficiente. En consecuencia, el programador debera diligenciar cuidadosamente los campos que considere necesarios para documentar adecuadamente su función (recordar que el objetivo final es el mantenimiento !), y eliminar los campos innecesarios. En la página siguiente se presenta la plantilla "encfunc.i" (encabezado de función) y se describe el significado de cada uno de los campos

```
/*****************************************************************************
* tipo nombre (tipo parm1, tipo parm2, tipo parm3,...)                       *
*****************************************************************************
* PROPOSITO:  ..................................................             *
* DESCRIPCION:  ................................................             *
*             ..................................................             *
* PARAMETROS:   tipo    nombre          comentario                           *
*               tipo    nombre          comentario                           *
*               tipo    nombre          comentario                           *
* RESULTADO:    tipo                    comentario                           *
* PRE:  ........................................................             *
* POST:  .......................................................             *
* OJO!:  .......................................................             *
*****************************************************************************
* AUTOR: i.i. apellido                                  FECHA: dd-mmm-91     *
*****************************************************************************/
tipo
nombre (tipo parm1, tipo parm2, tipo parm3,...)
{/*-------------------------------------------------------------------------*/
<definicion de variables locales>

<instrucciones>
/*-------------------------------------------------------------------------*/}
```

- PROPOSITO: Qué hace la funcion
- DESCRIPCION: Breve informacion sobre como lo hace (solo cuando sea necesario)
- PARAMETROS: Papel desempenado por cada uno de los parametros. Lo ideal es que la información suministrada en PROPOSITO y DESCRIPCION, junto con el sólo nombre del parametro ya provean la informacion suficiente. En consecuencia, el campo PARAMETROS debera ser diligenciado solo en los casos en que realmente se requiera.
- RESULTADO: Tipo y significado del valor retornado por la funcion (solo cuando no sea claro a partir de la demas informacion)
- PRE: Precondicion caracterizando las situaciones iniciales necesarias para el correcto desempeño de la funcion.
- POST: Postcondicion caracterizando la situación final, luego del desempeño correcto de la función [GRIES 81, CARDOSO 90].
- OJO!: Advertencia sobre posibles errores o limitaciones en la funcion.
- AUTOR y FECHA.

Programas 'C' en un solo archivo fuente

La plantilla "unfuente.c" presente una organizacion de un programa C (ansi) en las siguientes secciones:

- #include s de los encabezados estandares.
- Declaración de constantes y macros.
- Declaracion de struct s, enum s, union s y typedef s (sin definir de variables)
- Declaracion de todos los prototipos de las funciones del programa (orden alfabetico).
- Definicion (e inicializacion) de las variables globales (sin declarar de nuevos tipos)
- Definicion de la funcion main
- Definicion de las demas funciones del programa.

```
/*************************************************************************
* PROGRAMA: prg                                          FECHA: ---mes---- 1991 *
*************************************************************************
*                                                                       *
* PROPOSITO                                                             *
* DEL PROGRAMA: .....                                                   *
*               .....                                                   *
* SINTAXIS USO: prg [-opciones1] [-opciones2] file1 file2 ...           *
* OPCIONES-PARAMETROS:                                                  *
*      -x       .....                                                   *
*      -y       .....                                                   *
*      file1    .....                                                   *
*      file2    .....                                                   *
*                                                                       *
* AUTOR: i. i. apellido                                                 *
*************************************************************************/

        /*-------------------------------------------------------+
        |          Encabezados de librerias estandares           |
        +-------------------------------------------------------*/

#include <stdio.h>
#include <string.h>
#include <otros encabezados.h>

        /*-------------------------------------------------------+
        |    Declaracion de Constantes y Macros del programa     |
        +-------------------------------------------------------*/

#define .....   .....
#define .....   ....

        /*-------------------------------------------------------+
        | Declaracion de struct's, enum's, union's y typedef's   |
        +-------------------------------------------------------*/


... .... ......;

        /*-------------------------------------------------------+
        |        Prototipo de las funciones del programa         |
        +-------------------------------------------------------*/

tipo
nombre (... ..., ... ..., ... ...);

tipo
nombre (... ...., ... ...., ... ....);

        /*-------------------------------------------------------+
        |     Definicion de variables globales del programa      |
        +-------------------------------------------------------*/

... ..., ...., ...., ....;
... ...., ...., ...., ....;
... ..., ....,  ...., ....;
```

```
        /*-------------------------------------------------------+
        |                  Programa Principal                   |
        +-------------------------------------------------------*/

/*************************************************************************
* int   main (int argc, char *argv[])                            *
*************************************************************************
* PROPOSITO:  ..................................................  *
* DESCRIPCION:  ................................................  *
*  ............................................................  *
* PARAMETROS:   int     argc            # de parametros          *
*               char    *argv[0]        nombre comando invocado  *
*               char    *argv[1]        primer parametro         *
*               ....    ......          ........                 *
*               char    *argv[argc-1]   ultimo parametro         *
* RESULTADO:    int     0               ejecucion exitosa        *
*                       -1              ejecucion fracasada      *
* PRE:  ........................................................  *
* POST  ........................................................  *
* OJO!  ........................................................  *
*************************************************************************/
int
main (int argc, char *argv[])
{/*-------------------------------------------------------------------*/
<definicion variables locales>

<instrucciones>
return 0;
/*-------------------------------------------------------------------*/}

        /*-------------------------------------------------------+
        |  Definicion de las demas funciones del programa       |
        |                (orden alfabetico)                     |
        +-------------------------------------------------------*/

/*************************************************************************
* tipo nombre (tipo parm1, tipo parm2, tipo parm3,...)           *
*************************************************************************
* PROPOSITO:  ..................................................  *
* DESCRIPCION:  ................................................  *
*  ............................................................  *
* PARAMETROS:   tipo    nombre          comentario               *
*               tipo    nombre          comentario               *
*               tipo    nombre          comentario               *
* RESULTADO:    tipo                    comentario               *
* PRE:  ........................................................  *
* POST  ........................................................  *
* OJO!  ........................................................  *
*************************************************************************/
tipo
nombre (tipo parm1, tipo parm2, tipo parm3,..
{/*-------------------------------------------------------------------*/
<definicion de variables locales>

<instrucciones>
/*-------------------------------------------------------------------*/}

....
```

Programas 'C' en varios archivos fuente

Cuando un programa C se va a desarrollar en varios archivos fuente todos los archivos tendran en su nombre algún prefijo que identifique a que proyecto pertenecen. En la presentacion usaremos el nombre de proyecto "**pry**". Segun el estandar aquí propuesto un programa C en varios archivos fuente debe partirse y administrarse de la siguiente manera.

Archivos de Definiciones:

- Un archivo con el programa principal `main` y las funciones que dependen directamente del `main`. Se construye a partir de la plantilla "**pry_main.c**". El dueño de este archivo será el jefe del grupo, y todos los demas miembros tendrán unicamente permiso de lectura.
- Uno o mas archivos donde se definen grupos de funciones. El criterio de agrupacion puede ser por el tipo de tarea que desempeñan las funciones o por el objeto sobre el cual actuan, pero en cualquier caso, debe ser reflejo de la estructura global del diseño [TORO 91] (i.e., en un archivo irían las funciones de manejo de video, en otro las funciones de manejo de la base de datos, etc.). Cada uno de estos archivos se construye a partir de la plantilla "**pry_funx.c**". El dueño de cada uno de estos archivos será programador que lo esta escribiendo y todos los demas miembros del grupo tendrán unicamente permiso de lectura.
- Un archivo con la definicion de todas las variables globales. Se construye a partir de la plantilla "**pry_vglb.c**". El dueño de este archivo será el jefe del grupo, y todos los demas miembros tendrán unicamente permiso de lectura.

Archivos de Declaraciones:

- Un archivo con todas las declaraciones de constantes y macros globales del proyecto. A este archivo se le hace `#include` en todos los archivos de definiciones. Se construye a partir de la plantilla "**pry_defs.h**". El dueño de este archivo será el jefe del grupo, y todos los demas miembros tendrán unicamente permiso de lectura.
- Un archivo con todas las declaraciones globales de `struct`s, `union`s, `enum`s y `typedef`s (sin definicion de variables). A este archivo se le hace `#include` en todos los archivos de definiciones. Se construye a partir de la plantilla "**pry_typs.h**". El dueño de este archivo será el jefe del grupo, y todos los demas miembros tendrán unicamente permiso de lectura.
- Un archivo de declaracion **extern** de todos los prototipo de funciones globales del proyecto. Así, cada vez que algun programador del grupo pone a punto una nueva funcion global, debe actualizar este archivo de prototipos añadiendole una línea en la cual haga la declaracion `extern` del prototipo de la funcion que acaba de crear (y un breve comentario). A este archivo se le hace `#include` en todos los archivos de definiciones. Se construye a partir de la plantilla "**pry_prot.h**". El dueño de este archivo será el jefe del grupo, y todos los demas miembros tendrán permiso de lectura y escritura.
- Un archivo con la declaración **extern** de todas las variables globales (sin declaracion de nuevos tipos). A este archivo se le hace `#include` en todos los archivos de definiciones. Se construye a partir de la plantilla "**pry_vglb.h**". El dueño de este archivo será el jefe del grupo, y todos los demas miembros tendrán unicamente permiso de lectura.

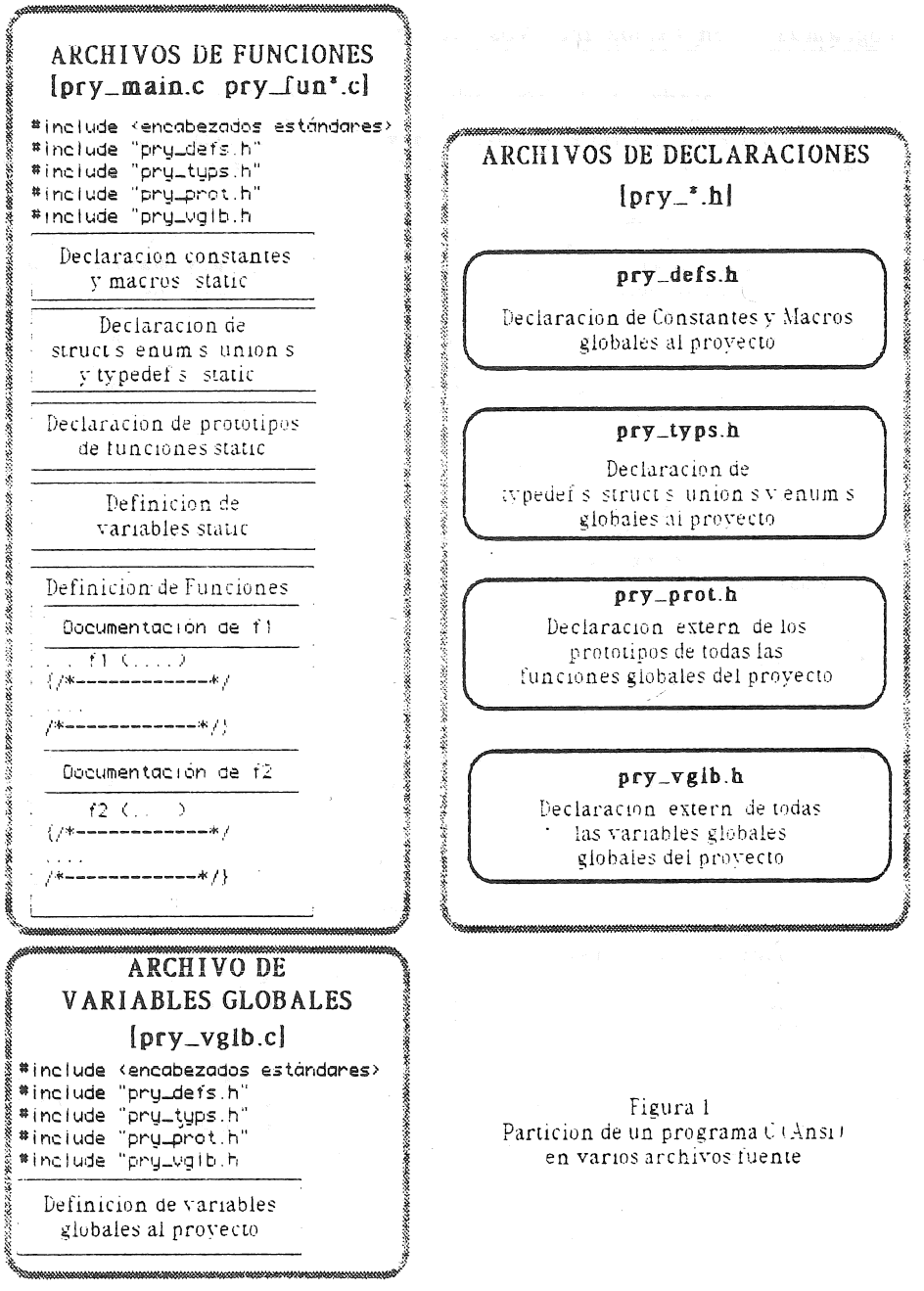

Figura 1
Particion de un programa C (Ansi)
en varios archivos fuente

A continuación se muestran una a una las plantillas de base para construir cada uno de los archivos anteriormente mencionados

Plantilla **pry_defs.h**

```
/*************************************************************************
* PROYECTO: pry                                          FECHA: ---mes---- 1991 *
*************************************************************************
*        DECLARACION DE CONSTANTES Y MACROS GLOBALES AL PROYECTO         *
*************************************************************************/

        /*-------------------------------------------------------+
        |                      CONSTANTES                       |
        +-------------------------------------------------------*/

#define ....  ........
#define ....  ........

        /*-------------------------------------------------------+
        |                       MACROS                          |
        +-------------------------------------------------------*/

#define ....(....., .....)  ........
#define ....(....., .....)  ........
```

Plantilla **pry_typs.h**

```
/*************************************************************************
* PROYECTO: pry                                          FECHA: ---mes---- 1991 *
*************************************************************************
*     DECLARACION DE typedef's, struct's, union's y enum's GLOBALES      *
*(En este archivo NO se deben definir variables ni declarar constantes) *
*************************************************************************/

....... ....  ..........;
....  .....  ..........,
```

Plantilla **pry_prot.h**

```
/*************************************************************************
* PROYECTO: pry                                          FECHA: ---mes---- 1991 *
*************************************************************************
*   DECLARACION DEL PROTOTIPO DE LAS FUNCIONES EXTERNAS DEL PROYECTO     *
*            (En este archivo NO se deben definir funciones)             *
*************************************************************************/

extern .... ......  (.... ....., .... ......, .... ......,  );
        /* proposito ..................................................*/

extern .... ......  (.... ....., .... ......, .... ......,  );
        /* proposito ..................................................*/
```

Plantilla **pry_vglb.h**

```
/*************************************************************************
* PROYECTO: pry                                          FECHA: ---mes---- 1991 *
*************************************************************************
*      DECLARACION 'extern' DE LAS VARIABLES GLOBALES DEL PROYECTO       *
*    (No definir variables ni declarar nuevas categorias de objetos)     *
*************************************************************************/

extern ....  ..........;     /* papel de la variable */
extern ....  ..........;     /* papel de la variable */
```

```
/*****************************************************************************
* PROYECTO: pry          MODULO: pry_main.c      FECHA: ---mes---- 1991 *
*****************************************************************************
* PROPOSITO                                                                 *
* DEL PROYECTO: ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*  ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
* SINTAXIS USO: pry [-opciones1] [-opciones2] file1 file2 ...               *
* OPCIONES-PARAMETROS:                                                      *
*     -x      ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*     -y      ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*     file1   ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*     file2   ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*                                                                           *
*    PROPOSITO GLOBAL DE LAS FUNCIONES IMPLEMENTADAS EN ESTE MODULO         *
* ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
* ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*****************************************************************************
* AUTOR DE ESTE MODULO: i.i. apellido                                       *
*****************************************************************************
* AUTORES DEL    i.i. apellido      i.i. apellido      i.i. apellido       *
*  PROYECTO:     i.i. apellido      i.i. apellido      i.i. apellido       *
*****************************************************************************/

        /*-------------------------------------------------------+
        |          Encabezados de librerias estandares           |
        +-------------------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include <otros_encabezados.h>

        /*-------------------------------------------------------+
        |    Declaracion de Constantes y Macros del proyecto     |
        +-------------------------------------------------------*/
#include "pry_defs.h"

        /*-------------------------------------------------------+
        | Declaracion de struct's, enum's, union's y typedef's   |
        +-------------------------------------------------------*/
#include "pry_typs.h"

        /*-------------------------------------------------------+
        |   Prototipo de las funciones externas del proyecto     |
        +-------------------------------------------------------*/
#include "pry_prot.h"

        /*-------------------------------------------------------+
        |    Declaracion de variables globales del proyecto      |
        +-------------------------------------------------------*/
#include "pry_vglb.h"

        /*-------------------------------------------------------+
        |          Declaracion de Constantes y Macros            |
        |           usados unicamente en este modulo             |
        +-------------------------------------------------------*/
#define .... ....
#define .... ....
```

```
        /*-------------------------------------------------------+
        | typedef's, struct's, union's y enum's de este modulo  |
        +-------------------------------------------------------*/

   .. ........ .....;
   .   ......   ......;

        /*-------------------------------------------------------+
        |  Prototipo de las funciones 'static' de este modulo   |
        +-------------------------------------------------------*/
static ..... ...... (.... ....., .... ......., .... .......,   );
static  ..... ..... (.... ....., ..... ......., ..... .......,  );

        /*-------------------------------------------------------+
        |    Definicion de variables 'static' de este modulo    |
        +-------------------------------------------------------*/
static ..... ......;
static ....   .......;

        /*-------------------------------------------------------+
        |                   Programa Principal                  |
        +-------------------------------------------------------*/
/*************************************************************************
* int   main (int argc, char *argv[])                                    *
*************************************************************************
* PROPOSITO:    ..................................................       *
* DESCRIPCION:  ..................................................       *
*               ..... ..... ..... ..... ..... ..... ..... .....        *
* PARAMETROS:   int     argc            * de parametros                  *
*               char    *argv[0]        nombre comando invocado          *
*               char    *argv[1]        primer parametro                 *
*               char    ......          ........                         *
*               char    *argv[argc-1]   ultimo parametro                 *
* RESULTADO:    int     0               ejecucion exitosa                *
*                       -1              ejecucion fracasada              *
*************************************************************************/
int main (int argc, char *argv[])
{/*-------------------------------------------------------------------*/
<definicion variables locales>

<instrucciones>
/*-------------------------------------------------------------------*/}

        /*-------------------------------------------------------+
        |   Definicion de las demas funciones de este modulo    |
        +-------------------------------------------------------*/
/*************************************************************************
* tipo nombre (tipo parm1, tipo parm2, tipo parm3,...)                   *
*************************************************************************
* PROPOSITO:    ..................................................       *
* DESCRIPCION:  ..................................................       *
*               ..... ..... ..... ..... ..... ..... ..... .....        *
*************************************************************************/
tipo nombre (tipo parm1, tipo parm2, tipo parm3,...)
{/*-------------------------------------------------------------------*/
<definicion de variables locales>

<instrucciones>
/*-------------------------------------------------------------------*/}
```

Plantilla: **pry_funx.c**

```
/*************************************************************************
* PROYECTO: pry          MODULO: pry_funx.c      FECHA: ---mes---- 1991 *
*************************************************************************
*    PROPOSITO GLOBAL DE LAS FUNCIONES IMPLEMENTADAS EN ESTE MODULO     *
* ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
* ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... *
*************************************************************************
* AUTOR DE ESTE MODULO:                                                 *
*************************************************************************
* AUTORES DEL   i.i. apellido      i.i. apellido      i.i. apellido     *
*  PROYECTO:    i.i. apellido      i.i. apellido      i.i. apellido     *
*************************************************************************/

<*includes y declaraciones estáticas idénticos a "pry_main.c">
...
        /*-------------------------------------------------------+
        |      Definicion de las funciones de este modulo       |
        +-------------------------------------------------------*/

/*************************************************************************
* tipo nombre (tipo parm1, tipo parm2, tipo parm3,...)                  *
*************************************************************************
* PROPOSITO:    ..........                                              *
* DESCRIPCION:  ..........                                              *
*               ..........                                              *
* PARAMETROS:   tipo    nombre          comentario                      *
*               tipo    nombre          comentario                      *
* RESULTADO:    tipo                    comentario                      *
* PRE:          ..........                                              *
* POST          ..........                                              *
*************************************************************************/
tipo
nombre (tipo parm1, tipo parm2, tipo parm3,...)
{/*-------------------------------------------------------------------*/
<definicion de variables locales>

<instrucciones>
/*-------------------------------------------------------------------*/}
....
```

Plantila: **pry_vglb.c**

```
/*************************************************************************
* PROYECTO: pry          MODULO: pry_vglb.c      FECHA: ---mes---- 1991 *
*************************************************************************
*          DEFINICION DE LAS VARIABLES GLOBALES DEL PROYECTO            *
* (NO declarar struct,s, union's, enum's ni typedef's en este archivo)  *
*************************************************************************/

<*includes idénticos a "pry_main.c">
...
        /*-------------------------------------------------------+
        |   Definicion de las Variables Globales al proyecto    |
        +-------------------------------------------------------*/

....    ......., ....., ......;
....    ......., ....., ......;
```

Manejar 'a mano' tantos archivos fuente puede ser causa de errores de coherencia o de ineficiencias (encadenar versiones desactualizadas, recompilar innecesariamente todos los archivos...). Es entonces importante apoyarse en las herramientas usualmente disponibles para manejo de programas partidos en varios fuentes, como **make** [KERNIGHAN 84, cap. 8] o archivos **.prj** [BORLAND 88]. A continuacion se muestran las plantillas estandar para el uso de tales herramientas.

Plantilla **makefile**

```
#============================================================================#
# makefile del proyecto "pry"                                  ---mes--- 1991 #
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# INSTRUCCIONES.                                                             #
#   - Cambie "pry" por el nombre abreviado de su proyecto.                   #
#   - Ajuste el nombre de los archivos pry_fun1.c ... pry_funN.c, y com-     #
#     plete los 'path' necesarios.                                           #
#   - Revise y eventualmente redefina las macros CC, CFLAGMOD, CFLAGEXE.     #
#============================================================================#

# +-----------------------------------+
# | Re-Definicion de Macros estandares |
# +-----------------------------------+
# Coloque en la siguiente definicion su compilador C preferido
CC = "/usr/local/bin/gcc"

# Coloque en la siguiente definicion los flags
# deseados para la compilacion de cada modulo
CFLAGMOD = "-ansi"

# Coloque en la siguiente definicion los flags
# deseados para la compilacion del ejecutable principal
CFLAGEXE =

# +-----------------------+
# | Definicion de Macros |
# +-----------------------+
OBJETOS =       pry_vglb.o pry_main.o pry_fun1.o pry_fun2.o . . pry_funN.o
FUENTES =       pry_vglb.c pry_main.c pry_fun1.c pry_fun2.c ... pry_funN.c
ENCABEZADOS =   pry_defs.h pry_typs.h pry_prot.h pry_vglb.h

# +-----------------------------------+
# | Re-Definicion de Reglas implicitas |
# +-----------------------------------+
.c.o:
        $(CC) -c $(CFLAGMOD) $<

# +-------------------------------------------------+
# | Reglas para armar el programa pry y sus modulos |
# +-------------------------------------------------+

pry:        $(OBJETOS) ;   $(CC) $(CFLAGEXE) -o pry $(OBJETOS)
pry_vglb.o: $(ENCABEZADOS) pry_vglb.c
pry_main.o: $(ENCABEZADOS) pry_main.c
pry_fun1.o: $(ENCABEZADOS) pry_fun1.c
....
pry_funN.o: $(ENCABEZADOS) pry_funN.c
```

Plantilla **pry.prj**

```
# Archivo de definicion del proyecto pry para Turbo-C (2.0)
# Instrucciones:
#   - Cambie "pry" por el nombre abreviado de su proyecto.
#   - Ajuste el nombre de los archivos pry_fun1.c ... pry_funN.c, y com-
#     plete los 'path' necesarios.
#   - Hecho lo anterior, elimine esta linea y las anteriores !!!.

pry_vglb ( pry_defs.h pry_prot.h pry_typs.h pry_vglb.h )
pry_main ( pry_defs.h pry_prot.h pry_typs.h pry_vglb.h )
pry_fun1 ( pry_defs.h pry_prot.h pry_typs.h pry_vglb.h )
pry_fun2 ( pry_defs.h pry_prot.h pry_typs.h pry_vglb.h )
....
pry_funN ( pry_defs.h pry_prot.h pry_typs.h pry_vglb.h )
libreria_especial_1.lib
....
```

ALGUNAS RECOMENDACIONES GENERALES

> puedo decir si un programa es malo con un simple vistazo del listado. Obviamente, el solo listado no bastaría para afirmar que sea bueno, pero si desde tres metros el listado tiene mal aspecto, puedo garantizar que el programa no ha sido escrito con cuidado.
>
> Charles Simonyi (autor de Multiplan, Word, Excel). [LAMMERS 86]

Son varias las publicaciones con recomendaciones generales sobre nomenclatura, indentación, sintaxis y estilo, orientadas a aumentar la legibilidad y facilitar el mantenimiento de programas C (GNU 90), [LEDGARD 87]. A continuación mencionaremos unas pocas reglas básicas.

No alterar la sintáxis de C

No cambiar la apariencia ni las estructuras de control de C a través del preprocesador [ARNOLD 88]. En efecto, la versatilidad del preprocesador de C permite extender o cambiar la sintaxis del texto a compilar, de manera que se puedan compilar programas escritos con sintaxis intermedias entre C y Pascal (u otros lenguajes). Son entonces nocivas macros del estilo:

```
#define   begin   {
#define   end     }
#define   if      if (
#define   then    )
#define   and     &&
#define   or      ||
#define   not     !
#define   equal   ==
```

pues generan grandes complicaciones para el mantenimiento y dificultan la portabilidad entre programadores (¡aún más importante que la portabilidad entre computadores!). Además, contribuye a impedir que el programador adopte realmente el estilo de C.

Nomenclatura

Los programas grandes deben crear y bautizar una gran cantidad de objetos (variables, procedimientos, constantes, archivos). Es entonces conveniente establecer criterios de nomenclatura, de manera tal que el nombre mismo del objeto provea alguna información sobre su categoría, características, ... [FOSTER 87]

- Los nombres de las variables importantes (i.e., excluyendo contadores locales, apuntadores temporales, etc.) deberán construirse a partir de un sustantivo eventualmente seguido de un adjetivo, abreviados. Ejemplos válidos son `linea_activa`, `promed_total`, `limite_credito`, `acum_ventas`, `colum_cursor`, `color_actual`, `ventas_dia_anterior`, ...
- Los nombres de las funciones deberán construirse a partir de un verbo en infinitivo, eventualmente seguido de un sustantivo y un adjetivo, abreviados. Ejemplos válidos son `verificar_nombre`, `acumular_venta`, `cargar_linea_nueva`, `guardar_plano`, ...
- Las variables apuntadoras deberán tener nombre de variable, seguido de un sufijo (`_ptr`) que resalte su condición (`prox_linea_ptr`, `sig_instruccion_ptr`, `ant_columna_ptr`, ...)
- Los nombres de categorías de objetos deberán estar construidos a partir de un sustantivo y estar seguidos de un sufijo que los distinga de los objetos en sí (`_str`: struct; `_enm`: enum; `_uni`: union; `_typ`: typedef)

```
struct cliente_str {
    ---- ------,
    ---- --------;
    ---  -----;
};
enum comandos_enm {borre, suba_cursor, baje_cursor, inserte, termine};
```

Declaraciones y Definiciones

- Evitar declarar categorías de objetos y definir objetos de dicha categoría en la misma frase. Preferiblemente declarar todas las categorías de objetos en la sección adecuada de la plantilla, y definir posteriormente las variables importantes de preferencia en definiciones separadas. Así por ejemplo en lugar de escribir

```
struct cliente_str {
    ---- ------;
    --   -------,
    ---  -----;
} cliente_viejo, cliente_nuevo, cliente_actual;
```

es más claro escribir las frases:

```
struct cliente_str {
    ---- ------,
    --   -------;
    ---  -----;
};

struct cliente_str  cliente_actual;
struct cliente_str  cliente_nuevo;
struct cliente_str  cliente_viejo;
```

Indentación

Un aspecto un tanto marginal, pero que suele dar lugar a acaloradas discusiones, es el esquema de indentacion. Tal como lo señalan los autores de C [K&R 88, pag. 10], lo realmente importante es que todos los miembros del equipo de desarrollo usen consistentemente algun estilo. De hecho es facil construir herramientas que reformateen un programa de acuerdo a diversos esquemas de indentacion [ECUYER 87] [CAMERON 88].

El esquema aqui presentado es una ligera variante del estilo presentado por los autores de C [K&R 88]. Nuestro esquema resalta el alcance del control, de manera tal que de un solo vistazo, y sin necesidad de ningun análisis sintáctico, se ubiquen fácilmente los límites de control de una instruccion. El uso consistente de este estilo nos ha dado excelentes resultados, permitiendo inclusive con el solo reformateo de antiguos programas, encontrar graves errores que habian permanecido ocultos largo tiempo [K&R 88, pag. 56].

Las reglas del esquema propuesto son las siguientes: a nivel mas externo las instrucciones empiezan en la columna 1, las estructuras de control se cierran siempre y explícitamente a la misma altura que se abren; los cuerpos de instrucciones de control se indentan 4 columnas; los fines o comienzos de seccion dentro de una misma estructura se indentan 2 columnas. El esquema de indentacion tiene entonces la siguiente apariencia.

```
while (expresion)
    frase simple
;

for (----; ----; ----)
    frase simple
;

do {
    frase simple;
} wile (expresion);

if (expresion)
    frase simple
;

if (expresion)
    frase simple;
  else
    frase simple
;

if (expresion) {
    --------;
    --------;
  }
  else
    frase simple
;
```

```
while (expresion) {
    --------,
    --------;
}

for (----; ----; ----) {
    --------;
    --------;
}

do {
    -------;
    -------;
} while (expresion);

if (expresion) {
    -------;
    -------,
};

if (expresion) {
    --------;
    --------;
  }
  else {
    --------,
    --------;
  }
;

if (expresion)
    frase simple;
  else {
    --------,
    --------;
  }
;
```

Finalmente, se presentan los esquemas de indentación del switch y de la sucesión de else-if's. Es de resaltar que la sucesión de else-if's, que implementa un esquema de control mas general que el switch, se trata como una estructura en sí, y no como anidamiento de if's.

```
switch (expresion entera) {
  case <constante1>:
    ---------;
    break;
  case <constante2>:
    ---------;
    break;
  case <constante3>:
    ---------;
    ---------;
    break;
  default:
    ---------;
    break;
}
```

```
if (expresion1) {
    ---------;
    ---------;
}
else if (expresion2)
    ---------;
else if (expresion3) {
    ---------;
    ---------;
}
else {
    ---------;
    ---------;
}
```

Tal como se observa, ocasionalmente aparecen algunos pocos ; o } redundantes. Tales frases nulas son perfectamente validos en la gramatica de C y evidencian clara y brevemente el alcance del control en cada función, lo cual contribuye grandemente a la legibilidad del programa. Como ejemplo de comparación de presenta el siguiente trozo de codigo tomado de [K&R 88, pag. 134]:

```
int main (void)   /* Indentación de [K&R 88, pág. 134] */
{   int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n", keytab[n].count, keytab[n].word);
    return 0;
}
```

```
int main (void)   /* Indentación con alcance de control explícito */
{/*-------------------------------------------------------------------*/
int   n;
char  word[MAXWORD];

while (getword(word, MAXWORD) != EOF)
    if (isalpha(word[0]))
        if ((n = binsearch(word, keytab, NKEYS)) >= 0)
            keytab[n].count++
        ;
    ;
;
for (n = 0; n < NKEYS; n++)
    if (keytab[n].count > 0)
        printf("%4d %s\n", keytab[n].count, keytab[n].word)
    ;
;
return 0;
/*-------------------------------------------------------------------*/}
```

CONCLUSIONES

> hay que estar siempre dispuesto a leer codigo escrito por otros, y a dar nuestro propio codigo a revisión por otras personas. Hay que desear meterse en ese increible ciclo de realimentacion en el que se consigue que otras personas le digan a uno lo que está mal, y uno lo acepta ... No se puede permitir que pequeñas idiosincracias se interpongan y estorben esta realimentación ...".
>
> no creemos ni toleramos el sistema de la *primma donna* en el que solo porque alguien es muy bueno, se le permite que no comente su codigo, que no se comunique con los demas, que arbitrariamente imponga sus creencias a los demas ..."
>
> el equipo de programación ha de estar formado por personas que confien en si mismas y en los demas miembros del equipo, y que se respeten mutuamente, puesto que se trata de un trabajo realmente intimo. Es como representar juntos en una misma obra musical"
>
> Bill Gates (fundador-director de MicroSoft) ,LAMMERS 86.

La verdadera justificacion de una propuesta de estandar de organizacion y documentacion solo puede surgir de la practica. Durante casi tres años hemos venido usando y afinando el estandar aqui presentado en nuestros proyectos de desarrollo, en numerosas tesis y proyectos de estudiantes de pregrado y postgrado, y en asesorias directas a empresas. Podemos afirmar que una vez superada la resistencia inicial a alterar los habitos que traen los programadores, habitos formados mas por costumbre que por reflexion, los resultados son muy positivos. No solo ha sido evidente que el desarrollo y mantenimiento se han vuelto mas coordinados y certeros, sino tambien, que el uso de herramientas claves para el desarrollo de sistemas complejos (make, sccs, rcs) [KERNIGHAN 84],[GNU 90] ha sido mas claro y ortogonal. Inclusive en varias ocasiones hemos visto como los mas excepticos y reacios terminaron convirtiendose en los mas entusiastas defensores del estandar, convencidos de que la creatividad e individualidad tiene mejor campo de expresión en el analisis y el diseño que en detalles de indentacion, nomenclatura, forma de documentar, etc.

¡Las plantillas aquí presentadas (y algunas otras) puede ser obtenidas en versión magnetica via e-mail, escribiendo a vmtoro@andescol.bitnet!

BIBLIOGRAFIA

[ARNOLD 88] Ken Arnold • "C Advisor: fun with the preprocessor". Unix Review, pags. 73-77, april 1988.

[BOEHM 84] Barry Boehm • "Software Economics". IEEE Transactions on Software Engineering.

[BORLAND 88] Borland International • "Turbo C Version 2.0" © 1988.

[CAMERON 88] Robert D. Cameron • "An Abstract Pretty Printer". IEEE Software, Nov 1988.

[CARDOSO 89] Rodrigo Cardoso • "Verificacion y Desarrollo de Programas". Publicaciones del Departamento de Sistemas - Universidad de los Andes.

[DIJKSTRA 76] Edsger W. Dijkstra • "A Discipline of Programming". Prentice-Hall © 1976.

[ECUYER 87] Pierre L. Ecuyer • "Formal Formatting Rules for Pascal Programs". The Journal of Systems and Software, Vol. 7, pags. 311-322, 1987.

[EINBU 88] John M. Einbu • "An Architectural Approach to Improved Program Maintainability". Software Practice and Experience, Vol. 18, Nº 1, enero 88.

[FOSTER 87] John R. Foster; Malcolm Munro • "A Documentation Method based on Cross-Referencing". IEEE Conference on Software Maintenance (Austin-Texas) 1987, pags. 181-185

[GNU 90] "GNU's is not Unix" project • Archivo /pub/gnu/standards.text (Noviembre2.1990); Obtenido via ftp anonimo al computador prep.ai.mit.edu (internet) en MIT

[GRIES 81] David Gries • "The Science of Programming". Springer-Verlag © 1981

[K&R 88] Brian W. Kernighan; Dennis M. Ritchie • "The C Programming Language (2º edition)". Prentice Hall © 1988

[KERNIGHAN 84] Brian W. Kernighan; Rob Pike • "The UNIX Programming Environment" Prentice Hall © 1984

[KNUTH 68] Donald E. Knuth • "The Art of Computer Programming: Fundamental Algorithms" (Vol. I). Addison-Wesley © 1968

[KNUTH 84] Donald E. Knuth • "Literate Programming". The Computer Journal (Wiley-Heyden). Vol. 27, Nº 2, pags. 97-111, 1984

[LAMMERS 86] Susan Lammers • "Programmers at Work: interviews (Programadores en Accion: entrevistas")". MicroSoft Press © 1986 (Ediciones Anaya Multimedia S.A. © 1988)

[LEDGARD 87] Henry Ledgard; John Tauer • Professional Software (Vol. II) "Programming Practice". Addison-Wesley © 1987

[MEYER 88] Bertrand Meyer • Object-Oriented Design. Prentice Hall © 1988

[PRESSMAN 87] Roger S. Pressman • Software Engineering: a practitioner's approach (2º edition). McGraw-Hill © 1987

[SCHWARTZ 82] Barbara Schwartz • Eight Myths about Software Maintenance Datamation, pags. 125-128, august 1982

[TORO 87] Victor M. Toro C. • Diseno Especificacion y Prototipificacion de Sistemas Interactivos. XIII Conferencia Latinoamericana de Informatica. Bogota - Noviembre 1987; Memo de Investigacion Nº 28. Facultad de Ingenieria. Universidad de los Andes

[TORO 91] Victor M. Toro C. • Sistemas Interactivos = Tipos Abstractos de Datos + Control + Interfaz. Método y Estandar de Diseno de Sistemas Interactivos Memo de Investigacion (proxima publicacion). Facultad de Ingenieria - Universidad de los Andes

[YUDKIN 88] Howard L. Yudkin • Emerging trends present opportunities, challenges for standards developpment. IEEE Computer, pags. 57-58, August 1988