Dr. Oscar BRUNO – Esp. Ing. Jose Maria Sola

Sintaxis y semántica de los lenguajes

Bibliotecas en ANSI C

Objetivos

- Presentar el **concepto de biblioteca**.
- Presentar los pasos necesarios para la **creación de una biblioteca** con ANSI C.
- Presentar los pasos para la **construcción de biblioteca con BCC32**.
- Presentar los pasos para **compilar** (y linkeditar) **con BCC32 programas fuente que utilizan bibliotecas a parte de la Standard**.

Los objetivos se cumplen mediante la presentación de un caso de estudio.

Introducción

En este artículo se introduce el concepto de biblioteca y dos casos de estudio que presentan aplicaciones prácticas del concepto. El diseño de la solución para cada caso de estudio se modela con UML.

El proceso para la creación de una biblioteca es genérico y puede ser reproducido en diferentes entornos de desarrollo ANSI C, teniendo en cuentas las características de cada uno. En este artículo se ejemplifica mediante el compilador de línea de comando "***C++ Compiler 5.5 with Command Line Tools***", también conocido como ***BCC32***, disponible en

http://www.borland.com/ y ftp://ftpd.borland.com/download/bcppbuilder/freecommandLinetools.exe

Conceptos

Bibliotecas

Una biblioteca es una colección de herramientas para el programador. En una biblioteca se encuentran subprogramas (funciones para ANSI C), tipos de datos, constantes, enumeraciones y otros identificadores. Las bibliotecas permiten la modularización, el desacoplamiento y la centralización de funcionalidad común.

Una biblioteca tiene una parte **privada**, la **implementación**, y otra **pública**, la **interfaz**. La biblioteca **encapsula** la implementación, y la interfaz debe estar diseñada para que **oculte información** sobre el diseño de la implementación. ANSI C provee una biblioteca Standard, y varios archivos header que hacen de interfaz. Cada archivo header representa una *agrupación funcional* (e.g. string.h para el manejo de cadenas y stdio.h para la entrada y salida).

Las bibliotecas se utilizan también para el desarrollo de **Tipos de Datos Abstractos** (TADs).
Una biblioteca puede contener la implementación de un TAD y venir acompañada por su interfaz; o bien una misma biblioteca puede contener varios TADs y estar acompañada por diferentes interfaces, una por cada TAD. Las bibliotecas son utilizadas por programa u otras bibliotecas.a

Archivos Header

Los archivos header (encabezados) son archivos con código fuente ANSI C que deben ser compartidas por varios programas. En su gran mayoría, son declaraciones y macros; no contienen *definiciones de funciones*, sino *declaraciones de funciones* (i.e. *prototipos*). Los archivos header, por convención, tienen como extensión **".h"**.
No se debe confundir el concepto de archivo header con biblioteca. Por ejemplo, la biblioteca Standard posee varios archivos header, en los cuales header hay declaraciones para utilizar las funciones que están precompiladas (i.e. código objeto) en la biblioteca Standard.

Funciones Públicas

Una función pública de una biblioteca es una función que puede ser invocada por diferentes programas u otras bibliotecas.
En ANSI C, por defecto, las funciones son públicas; pueden ser invocadas desde otros programas o bibliotecas (i.e. unidades de traducción).
Leer la sección **4.6 Static Variables** del capítulo **4. Functions and Program Structure** de **[K&R1988]**.

Funciones Privadas (static)

Una función privada de una biblioteca es una función que el programa o biblioteca llamante no necesita conocer para poder hacer uso de la biblioteca.
En general son funciones helper ("auxiliares") que ayudan al diseño de la implementación de la biblioteca. Por ejemplo, la función pública `Alumno *ImprimirAlumno(int legajo)` puede, internamente, invocar (i.e. hacer uso) de la función privada `BuscarAlumno` para abstraer la tarea de la búsqueda del alumno a imprimir.
En ANSI C, las funciones privadas se implementan precediendo el prototipo y la definición de la función con la palabra clave `static`. Las funciones static sólo pueden ser invocadas desde la unidad de traducción (i.e. archivo fuente) dónde están definidas, no pueden ser invocadas por otras programas o bibliotecas. El prototipo de una funcione static se encuentran al comienzo del archivo fuente donde está definida, junto a otras declaraciones externas; sus prototipos no se escriben en los archivos header.
Leer la sección **4.6 Static Variables** del capítulo **4. Functions and Program Structure** de **[K&R1988]**.

Directivas al Preprocesador

Permiten dirigir las acciones del preprocesador. Otorgan funcionalidades de compilación condicional y de substitución.

La directiva **`#include`** incorpora el contenido completo del archivo indicado. Existen dos sintaxis, la primera es particular para los *headers Standard* (e.g. **`#include <time.h>`**), y la segunda es para otros *archivos fuente no Standard* (e.g. **`#include "stack.h"`**).

La directiva **`#define`** introduce un identificador, opcionalmente asociándolo con una porción de código fuente.

La expresión **`defined`** `<identificador>` se reemplaza por *cero* o por *uno* en función de si el identificador ya fue definido o no.

Las directivas de preprocesador **`#ifndef`** `<expresión constante>` y **`#endif`** le indican al preprocesador que procese el bloque que encierran si y solo si la expresión constante es *cero* (falsa), permiten la compilación condicional.

Las siguientes líneas son equivalentes

```
#if ! defined <identificador>
#ifndef <identificador>
```

La siguiente estructura sintáctica evita que el contenido de un archivo *header* sea procesado más de una vez.

```
#ifndef <identificador>
#define <identificador>
Líneas de código fuente del archivo header.
#endif
```

Leer de **[K&R1988]** las siguientes secciones del capítulo **A.12 Preprocessing** · **A.12.3 Macro Definition and Expansion**

· **A.12.4 File Inclusion**

· **A.12.5 Conditional Compilation**

Caso de Estudio – Saludos

Este caso presenta dependencias más complejas entre bibliotecas y un programa. La dependencias están dadas por la relación de invocación y de inclusión. Una de las bibliotecas utiliza una función privada.

El programa hace uso de dos bibliotecas y también de la Biblioteca Standard. Una biblioteca llama a la otra, y la última invoca a una función de la Standard.

La única y simple acción del programa es la emitir por stdin un saludo una determinada cantidad de veces.

Construcción

Archivos Fuentes

Biblioteca Saludar

Saludar.h

```
#ifndef SaludarHeaderIncluded
#define SaludarHeaderIncluded
/* Saludar.h
*/
void Saludar( const char* unNombre );
#endif
```

Saludar.c

```
/* Saludar.c
*/
#include <stdio.h> /* printf */
#include "Saludar.h" /* Saludar */
/* Prototipo de función privada */
static void SaludarEnCastellano( void );
static void SaludarEnCastellano( void ){
printf("Hola");
return;
}
void Saludar( const char* unNombre ){
SaludarEnCastellano();
printf(", %s\n", unNombre);
return;
}
```

Biblioteca SaludarMuchasVeces

SaludarMuchasVeces.h

```
#ifndef SaludarMuchasVecesHeaderIncluded
#define SaludarMuchasVecesHeaderIncluded
/* SaludarMuchasVeces.h
*/
void SaludarMuchasVeces( const char* unNombre, int cuantasVeces );
#endif
```

SaludarMuchasVeces.c

```
/* SaludarMuchasVeces.c
*/
#include "SaludarMuchasVeces.h" /* SaludarMuchasVeces */
#include "Saludar.h" /* Saludar */
void SaludarMuchasVeces( const char* unNombre, int cuantasVeces ){
```

```
int i;
for( i = 0; i < cuantasVeces; i++)
Saludar( unNombre );
return;
}
```

Aplicación

Aplicación.c

```
/* Aplicacion.c */
#include <stdio.h> /* puts */
#include <stdlib.h> /* EXIT_SUCCESS */
#include "Saludar.h" /* Saludar */
#include "SaludarMuchasVeces.h" /* SaludarMuchasVeces */
int main ( void ){
/* de biblioteca Saludar */
Saludar("Mundo");
puts("");
/* de biblioteca Saludar */
/* pero no se puede invocar porque es static */
/* SaludarEnCastellano("Mundo"); */
/* puts(""); */
/* de biblioteca SaludarMuchasVeces */
SaludarMuchasVeces("Mundo", 4);
puts("");
return EXIT_SUCCESS;
}
```

Conclusión

Las bibliotecas demuestran ser una excelente forma de implementar la **modularización** y para la construcción de **TADs**.

Se debe prestar especial cuidado al diseño de las interfaces que exponen las bibliotecas para cumplir con determinada funcionalidad, se debe aplicar el **ocultamiento de información**. Mediante la **abstracción** se debe evitar que el cliente de la biblioteca necesite conocer los detalles de implementación para hacer uso de la biblioteca. ANSI C permite encapsulamiento de componentes de las bibliotecas mediante la palabra clave **static**.

Bibliografía

K&R1988] "The C Programming Language, 2nd Edition", B. W. Kernighan & D. M. Ritchie, 1988, Prentice-Hall, ISBN 0-13-110362-8

Documentación Borland C++ Compiler 5.5 with Command Line Tools

OMG-Unified Modeling Language, v1.5, http://www.omg.org .—

Normativas y Guías para construcción de TADs

TAD

Tipos de datos definidos por

- Conjunto de valores
- Conjunto de operaciones

Pueden ser

- Fundamentales, básicos, primitivos, predefinidos.
- Derivados
- Definidos por el usuario o TAD.

ABSTRACCION

Capacidad para encapsular y aislar la información, del diseño y ejecución.

Es un proceso mental, mas sencillo que lo que modela para ser útil. El mapa es una abstracción del camino, la palabra león no ruge. Es una herramienta fundamental para tratar la complejidad.

ABSTRACCION EN EL SOFTWARE

- Instrucciones binarias
- Nemotécnicos de los lenguajes ensambladores
- Agrupamiento de instrucciones primitivas formando macroinstrucciones
- Encapsular y aislar en procedimientos, funciones, módulos, TAD y objetos.

TAD

- Lo crea el usuario y los puede manipular como los creados por el sistema.
- Para su creación se utilizan los módulos, para construirlos se debe poder
    - Dar una definición precisa del tipo
    - Hacer disponible de un conjunto de operaciones utilizables para manipular instancias del tipo.
    - Proteger los datos asociados de modo que solo se pueda trabajar con las operaciones definidas.
    - Hacer instancias múltiples del tipo.

OBJETOS

- TAD al que se añaden innovaciones para la reutilización
    - Conceptos
        - Abstracción
        - Encapsulamiento
        - Herencia
        - Polimorfismo
        - Persistencia
    - Entidades básicas
        - Objeto
        - Instancia
        - Clases

- Mensajes
- Métodos
- Propiedades
- Herencia – Jerarquía

MODULARIZACION

- Parte publica
  - Primitivas de acceso
  - Descripción de las propiedades de los datos
- Parte privada
  - Atributos
  - Representación

ABSTRACCION EN LENGUAJES DE PROGRAMACION

- Abstracción de datos
  - Representación: elección de la estructura del dato
  - Operaciones: Elección del algoritmo
- Abstracción de control
  - Abstracción procedimental
  - Parámetros

VENTAJAS DE LOS TADS

- Permite mejor conceptualización y modelización
- Mejora el rendimiento
- Separa implementación de especificación
- Permite agregados y mejoras sin afectar la parte publica

ESPECIFICACION DE UN TAD

- Informal
  - Detallar los valores del tipo y las operaciones que se pueden vincular con esos valores
  - TAD NombreTipo(valores y descripción)
- Formal
  - Axiomas que describen el compòrtamieno
  - TAD NombreTipo (valores que pueden tomar los datpos)
  - Sintaxis
    - Operación (Tipo Argumento) → Tipo resultado
    - ConjuntoVacio → Conjunto
    - Agregar(Conjunto, Elemento) → Conjunto
    - Pertenece(Conjunto, Elemento) → Booleano
    - ----------
  - Semantica
    - Operación(Valores particulares) → Expresión resultado
    - Union (ConjuntoVacio, ConjuntoVacio) → ConjuntoVacio
    - Union(ConjuntoVacio, Agregar(Conjunto, Elemento) → Agregar (conjunto, Elemento)

Especificación

Guía para la estructuración de la especificación de los valores

Descripción, n-upla, descripción de cada miembro de la n-upla, tipo de dato al que pertenece cada miembro de la n-upla, restricciones.

Guía para la estructuración de la especificación de las operaciones

· f : A × B × C → D × E (*Título de la operación*)

· Clasificación de la operación.

· Breve descripción.

· f : M × K × K → S × M (*Refinación de los conjuntos*)

· f (m_1, k_1, k_2) = (s, m_2) (*Identificación de los datos y resultados*)

· Precondiciones y Poscondiciones.

· Dominio e Imagen.

· Semántica descripta en lenguaje matemático con la ayuda de axiomas o de lenguaje natural.

· Ejemplo(s).

Normativas para las implementaciones

Archivos Encabezado

El contenido de los archivos encabezado debe estar "rodeado" por las siguientes directivas al preprocesador:

```
#ifndef _INCLUIR_TAD_
#define _INCLUIR_TAD_
/* Contenido del archivo encabezado. */
#endif
```

Esto permite la compilación condicional, evitando incluir más de una vez el contenido del archivo encabezado. Reemplazar *TAD* por un nombre del TAD que se está incluyendo.

Convención de Nomenclatura para los Identificadores

Estilos de "Capitalización"

PascalCase

Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen enminúsculas.

Ejemplos:

EstaEsUnaFraseEnPascalCase

Pila

AgregarElemento

camelCase

Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen en minúsculas, excepto la primera que está en minúsculas.

Ejemplos:

estaEsUnaFraseEnCamelCase

unaPila

laLongitud

UPPERCASE

Todas las palabras juntas, sin espacios. Todas las palabras en mayúsculas. En general se separan las palabras con undescores.

Ejemplos:

ESTAESUNAFRASEENUPPERCASE

LIMITE_SUPERIOR

MAXIMO

Underscores (Guiones Bajos)

· No usar con camelCase ni con PascalCase,

· Sí usar con UPPER_CASE.

· No usar delante de identificadores.

Identificadores para diferentes elementos

Nombres de tipo (typedef)

En camelCase, sustantivo. SistemaPlanetario, Planeta y NumeroAstronomico.

Funciones

En PascalCase, deben comenzar con un verbo en infinitivo. Ejemplos: OrdenarArreglo(),

SepararUsuarioDeDominio(), Planeta_SetNombre().

Funciones públicas de cada TAD

Los identificadores de las funciones públicas que implementan las operaciones se prefijan con el nombre

del TAD seguido de un *underscore* ("_"). SistemaPlanetario_,.

Variables Locales y Parámetros

En camelCase, sustantivo, en plural para arreglos.

Variables Globales.

En PascalCase, sustantivo, en plural para arreglos.

Enumeraciones.

Su typedef y sus elementos en PascalCase y en singular.

Constantes Simbólicas (#define).

En UPPER_CASE_CON_UNDERSCORES.

Cadenas

Implementadas sin tamaño máximo (malloc y free).

Funciones privadas

Definidas como static. No forman parte de la especificación.

Operaciones de Destrucción

No forman parte de la especificación. Permiten liberar los recursos tomados durante la creación.

Operaciones de Creación

Las operaciones de creación serán implementadas como funciones que retornan un puntero a un objeto del tipo del TAD. Deberán usar malloc para obtener memoria para ese objeto. Si no hay memoria disponible, retornarán NULL. Por ejemplo:

```
NumeroAstronomico *NumeroAstronomico_CrearDesdeCadena(
const char *unaCedena );
```

Valores de los TADs como parámetros

Un valor de un TAD será pasado como parámetro mediante un puntero a ese valor. Si el parámetro es del tipo entrada/salida no tendrá el calificador const, si es de entrada sí.

Por ejemplo:

```
SistemaPlanetario* SistemaPlanetario_AgregarPlaneta(
SistemaPlanetario* unSistemaPlanetario, /*inout*/
const Planeta* unPlaneta /*in*/
);
int NumeroAstronomico_EsOverflow(
const NumeroAstronomico* unNumeroAstronomico /*in*/
);
```

Operaciones de Modificación (Mutación, Setter)

Estas operaciones seguirán el modelo impuesto por funciones como strcat de ANSI C.

La función strcat recibe dos parámetros: Primero, la cadena que será modificada concatenándole al final otra cadena, y segundo, la cadena que será concatenada al final de la primera. La función retorna el puntero a la primera cadena.

```
char *strcat(char *s1, const char *s2);
```

Este modelo permite evitar construcciones del tipo:

```
char s1[4+1]="ab", *s2="cd";
strcat(s1, s2);
puts(s1);
```

al ser reemplazadas por:

```
char s1[4+1]="ab", *s2="cd";
puts( strcat(s1, s2) );
```

Análogamente

```
Planeta* Planeta_SetNombre(Planeta* unPlaneta, const char* unNombre);
```

permite reemplazar:

```
Planeta* unPlaneta=Planeta_Crear(…);
Planeta_SetNombre(unPlaneta, "Krypton");
puts( Planeta_GetNombre(unPlaneta) );
```

por:

```
Planeta* unPlaneta=Planeta_Crear(…);
puts( Planeta_GetNombre( Planeta_SetNombre(unPlaneta, "Krypton") ) );
```

Nombres de archivos

· Cada TAD se implementará en una biblioteca. El código fuente de la implementación estará en un archivo *TAD*.c, la declaración de la parte pública en el archivo encabezado *TAD*.h, y se generará la biblioteca *TAD*.lib.

· El código fuente del programa de aplicación que prueba el TAD será *TADAplicacion*.c, y se generará *TADAplicacion*.exe.

· Sólo se entregarán los códigos fuentes: *TAD*.c, *TAD*.h y *TADAplicacion*.c.

· No se aceptarán *TAD*.lib y *TADAplicacion*.exe.

Nombres de los archivos de los TADs de este TP:

· SistemaPlanetario.c, SistemaPlanetario.h, SistemaPlanetario.lib, SistemaPlanetarioAplicacion.c y SistemaPlanetarioAplicacion.exe

· Planeta.c, Planeta.h, Planeta.lib, PlanetaAplicacion.c y PlanetaAplicacion.exe.

· NumeroAstronomico.c, NumeroAstronomico.h, NumeroAstronomico.lib, NumeroAstronomicoAplicacion.c y NumeroAstronomicoAplicacion.exe.

Guía de secuencia de actividades para la generación de los TADs

A continuación se presenta una guía de una posible secuencia de actividades para la construcción de TADs.

En base a una *correcta especificación*, se diseña un *correcto programa de prueba*, luego se implementa el TAD. Si la implementación realizada pasa el programa de prueba, la *implementación es correcta*.

Se debe considerar que *el proceso es en general iterativo*, en el sentido de *la especificación siempre es la entrada para la implementación* y que debe estar *completamente definida*, pero que *hay veces que la implementación retroalimenta a la especificación para mejorarla* y volver a comenzar el proceso.

1. Comprensión del contexto y del problema que el TAD ayuda a solucionar.
2. Diseño de la Especificación.
3. Diseño de los casos de prueba a nivel especificación.
4. Implementación.
   - 4.1. Diseño de prototipos.
   - 4.2. Codificación de prototipos.
   - 4.3. Diseño programa de prueba (aplicación)
   - 4.4. Diseño y codificación de implementación de valores.
   - 4.5. Codificación de funciones públicas y privadas (static).
   - 4.6. Construcción de biblioteca.
   - 4.7. Ejecución de programa de prueba.
   - 4.8. ¿Hubo algún error? Entonces volver a 4.4

Presentación

Forma

· El trabajo debe presentarse en **hojas A4 abrochadas en la esquina superior izquierda**.

- En el **encabezado de cada hoja** debe figurar el **título del trabajo**, el **título de entrega**, el **código de curso** , **número de equipo** y los **apellidos de los integrantes del equipo**.
- Las hojas deben estar enumeradas en el pie de las mismas con el formato "**Hoja n de m**" .
- El **código fuente de cada componente del TP** debe comenzar con un **comentario encabezado**, con todos los datos del equipo de trabajo: **curso**; **legajo**, **apellido y nombre de cada integrante del equipo** y **fecha de última modificación.**
- La **fuente** (estilo de caracteres) a utilizar en la **impresión** de los **códigos fuente** y de las **capturas de las salidas** debe ser una fuente de ***ancho fijo*** (e.g. **`Courier New`**, `Lucida Console`).

El TP tiene **tres secciones imporatnes, una por cada TAD**, a su vez, **cada sección** tiene las siguientes **tres grandes sub-secciones**:

Nombre del TAD.

1. Especificación.

Especificación completa, extensa y sin ambigüedades de los valores y de las operaciones del TAD.

2. Implementación

Biblioteca que implementa el TAD.

2.1. **Listado** de código fuente del **archivo encabezado**, **parte pública**, *TAD*.h.

2.2. **Listado** de código fuente de la **definición de la Biblioteca**, **parte privada**, *TAD*.c. Las funciones privadas deberán ser precedidas por su documentación, un comentario con la documentación de la función indicando, entre otras cosas, propósito de la función, semántica de los parámetros in, out e inout.

1.2.3. **Salidas**. Captura impresa de la salida del **proceso de traducción** (BCC32 y TLIB).

3. Aplicación de Prueba

3.1. **Código Fuente**. Listado del código fuente de la aplicación de prueba, *TAD*Aplicacion.c.

3.2. **Salidas**

3.2.1. Captura impresa de la salida del **proceso de traducción** (BCC32).

3.2.2. Captura impresa de las salidas de la **aplicación de prueba**.

3.2.3. Impresión de **archivos de prueba de entrada** y de **archivos de salida generados durante la prueba**.

· El TP será acompañado por:

A. Copia Digitalizada

CD ó disquette (preferentemente CD) con copia de **solamente los 3 archivos de código fuente de cada TAD** (i.e.: *SistemaPlanetario.c, SistemaPlanetario.h, SistemaPlanetarioAplicacion.c Planeta.c, Planeta.h, PlanetaAplicacion.c, NumeroAstronomico.c, NumeroAstronomico.h y NumeroAstronomicoAplicacion.c*) **No se debe entregar ningún otro archivo**.

B. Formulario de Seguimiento de Equipo.

Tiempo

- Luego de la aprobación del TP se **evaluará individualmente a cada integrante del equipo**.

Normativas de Codificación

Introducción

Este artículo presenta las normas de codificación para ANSI C que se aplican en SSL. El objetivo es facilitar el entendimiento y modificación de programas. Su gran mayoría son aplicables a distintos lenguajes de programación y no se restringen solo a ANSI C.

Indentación (Sangría)

De **[K&R1988] "Chapter 1 – A Tutorial Introduction – 1.2 Variables and Arithmetic Expressions"**

El cuerpo de un while puede ser unas o más sentencias incluidas en llaves, como en el convertidor de temperatura, o una sola declaración sin las llaves, como en:

```
while (i < j)
i = 2 * i;
```

En cualquier caso, siempre indentaremos las sentencias controladas por el while con un tabulado (que hemos mostrado como cuatro espacios) así podemos ver de un vistazo qué sentencias están dentro del ciclo. La indentación acentúa la estructura lógica del programa

Aunque a los compiladores de C no les interesa la apariencia de un programa, las indentaciones y espaciados correctos son críticos para hacer los programas fáciles de leer. Recomendamos escribir solo una sentencia por línea, y usar espacios en blanco alrededor de operadores para clarificar la agrupación. La posición de las lleves es menos importante, aunque hay personas que mantienen creencias apasionadas sobre la ubicación de las mismas. Hemos elegido uno de varios estilos populares. Escoja un estilo que le quede bien, y luego úselo de manera consistente. >>

```
/* Incorrecto */
unsigned long int Factorial(unsigned int n)
{
if(n<=1)
return 1; return n * Factorial(n-1);}
/* Correcto */
unsigned long int Factorial(unsigned int n){
→ if( n <= 1 )
→ → return 1;
→ return n * Factorial( n – 1 );
}
```

Estilos de Indentación

A continuación se muestran diferentes estilos, la elección es puramente subjetiva, pero su aplicación al largo de un mismo desarrollo debe ser consistente.

Estilo K&R – También conocido como "The One True Brace Style"

El libro **[K&R1988]** usa siempre este estilo, salvo para las definiciones de las funciones, que usa el estilo **BSD/Allman**. Las Secciones principales de **Java** también usan este estilo. Este es el estilo que recomienda y que usa la Cátedra para todas las construcciones.

```
while( SeaVerdad() ) {
        HacerUnaCosa();
        HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo BSD/Allman

Microsoft Visual Studio 2005 impone este estilo por defecto. Nuevas secciones de Java usan este estilo. Es un estilo recomendable.

```
while( SeaVerdad() )
{
        HacerUnaCosa();
        HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Whitesmiths

```
while( SeaVerdad() )
   {
    HacerUnaCosa();
    HacerOtraCosa();
   }
HacerUnaUltimaCosaMas();
```

Estilo GNU

```
while( SeaVerdad() )
{
        HacerUnaCosa();
        HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Pico

```
while( SeaVerdad()
{  HacerUnaCosa();
HacerOtraCosa(); }
HacerUnaUltimaCosaMas();
```

Estilo Banner

```
while( SeaVerdad() ) {
HacerUnaCosa();
HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Formato del Código

Espacios

Utilizar los espacios horizontales y verticales de manera tal que el código quede compacto pero que a la vez permita una fácil lectura.

Tabulado de la Indentación

Utilizar el tabulado para indentar, no utilizar múltiples espacios.

Longitud de las líneas

Formatear el código de manera que las líneas no se corten en medios con solo 70 columnas.

Convención de Nomenclatura para los Identificadores

Separamos el estilo de los identificadores en dos partes: el estilo de "capitalización" y el estilo para identificar los elementos de las distintas categorías (e.g. variables, funciones, tipos de datos).

Estilos de "Capitalización"

En el contexto de los lenguajes de programación *case-sensitive* (i.e. diferencian mayúsculas de minúsculas), como ANSI C, las posibles combinaciones de mayúsculas, minúsculas y *underscores* (guiones bajos) establecen estilos para construir los identificadores.

PascalCase

Todas las palabras juntas, sin espacios. Todas las palabras, inclusive la primera, comienzan con mayúsculas y siguen en minúsculas. Ejemplos:

```
EstaEsUnaFraseEnPascalCase
Pila
AgregarElemento
```

camelCase

Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen en minúsculas, excepto la primera que está en minúsculas. Ejemplos:

estaEsUnaFraseEnCamelCase

unaPila

laLongitud

UPPERCASE

Todas las palabras en mayúsculas, pueden estar todas juntas o todas separadas con underscores. Ejemplos:

ESTAESUNAFRASEENUPPERCASE

ESTA_ES_OTRA

MAXIMO

Underscores (Guiones Bajos "_")

· Sí usar con UPPER_CASE.

· No usar delante de identificadores, ya que la biblioteca estándar tiene reservado esa forma de identificadores.

[K&R1988] "Chapter 1 – Chapter 2 - Types, Operators and Expressions – 2.1 Variables names"

· No usar con camelCase ni con PascalCase. Hay una excepción a esta última regla relacionada a los prefijos de los identificadores para las operaciones de los TAD.

Identificadores para diferentes categorías de elementos

La regla general es: usar *nombres significativos* para *identificadores significativos* y *nombres no significativos* para *identificadores no significativos*.

Los nombres significativos son en general largos y los no significativos cortos.

La misma regla se puede aplicar para identificadores de acceso global e identificadores de acceso local.

Así, una función que permite calcular un balance –tiene acceso global y es un concepto significativo– debe tener un identificador como GetBalance; mientras que una variable que se utiliza para iterar un arreglo –tiene acceso local y no es un concepto significativo– debe tener un identificador tan simple como i.

Nombres de Tipo (typedef)

En PascalCase, sustantivo en singular.

SistemaPlanetario

Planeta

NumeroAstronomico.

Funciones y Macros-con-parámetros

En PascalCase, deben comenzar con un verbo en infinitivo y en general son seguidas por un sustantivo o frase.

OrdenarArreglo()

SepararUsuarioDeDominio()

Cuando la función o macro-con-parámetros implementan una operación de un TAD el identificador debe comenzar con un prefijo formado por el nombre del TAD y un *underscore*:

Planeta_SetNombre()

NumeroAstronomico_EsOverflow()

Variables Locales y Parámetros

En camelCase, sustantivo, en plural para arreglos.

Variables Externas (Globales)

En PascalCase, sustantivo, en plural para arreglos.

Enumeraciones

El nombre del typedef de la enumeración y sus elementos en PascalCase y en singular.

Constantes Simbólicas (#define)

En UPPER_CASE_CON_UNDERSCORES.

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.4 Symbolic Constants"**

<< Las constantes simbólicas se escriben por convención en mayúsculas para que puedan ser rápidamente distinguidas de los nombres de variables escritos en minúscula. >>

No Usar Notación Húngara

La notación Húngara es una notación, que entre otras cosas, promueve la práctica de prefijar los identificadores para incluir *metadata* (datos sobre los datos) al identificador, como por ejemplo el tipo de dato de una variable.

Charles Simonyi desarrolló esta notación en Microsoft –llamada así por lo extraño de los identificadores resultantes y por la nacionalidad del autor– pero luego, la propia Microsoft desestimó su uso.

Una copia del paper original por *Charles Simonyi* se encuentra en

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp

<< No use notación Húngara. Los nombres buenos describen semántica, no tipo de dato. >>

Este tipo de notación genera dependencia con el lenguaje y complica el mantenimiento de los programas. La funcionalidad de la Notación Húngara hoy la proveen los IDEs (Integrated Development Enviroments) modernos, que con solo posicionar el cursor sobre el identificador

informa los atributos anunciados en su declaración. Por otro lado, las funciones (procedimientos, métodos) deben ser diseñadas de una forma tal que requieran pocas líneas de código, por lo que la declaración de las variables y parámetros se encuentran en un contexto acotado que facilita la ubicación de las declaraciones.

[Design Guidelines for Class - Naming Guidelines - Parameter Naming Guidelines]

http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconFieldUsageGuidelines.asp

"Best Practices"

Buenas Prácticas de Programación

No Usar Números Mágicos

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.4 Symbolic Constants"**

<< Es mala práctica enterrar en un programa "números mágicos" como 300 y 20; contienen poca información para alguien que pueda tener que leer el programa más adelante, y son difíciles de cambiar de una manera sistemática [durante el mantenimiento].

Una forma a tratar el tema de los números mágicos es darles nombres significativos. Una línea *#define* define a un nombre simbólico o constante simbólica como una cadena particular de caracteres:

#define <*nombre*> <*texto de reemplazo*>

Después de eso, cualquier ocurrencia del nombre (pero no entre comillas ni como parte de otro nombre) será substituida por el texto de reemplazo correspondiente. El nombre tiene la misma forma que un nombre de variable: una secuencia de letras y de dígitos que comienza con una letra. El texto de reemplazo puede ser cualquier secuencia de caracteres; no se limita a los números.>>

Evitar Variables Innecesarias

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.3 The For Statement"**

<<… en cualquier contexto donde se permite usar el valor de algún tipo, se pude usar una expresión más complicada de ese tipo. >>

Esto evita el uso de variables innecesarias

```
/* Suponiendo una variable int a */
/* Correcto */
printf("Cociente: %d\nResto: %d\n", a / 2, a % 2);
/* Suponiendo una variable int a */
/* Incorrecto */
int c = a / 2;
int r = a % 2;
printf("Cociente: %d\nResto: %d\n", c, r);
```

Escribir las Constantes Punto Flotante con Puntos Decimales

De **[K&R1988] "Chapter 1 – A Tutorial Introduction – 1.2 Variables and Arithmetic Expressions"**

<< … escribir constantes de punto flotante con puntos decimales explícitos inclusive cuando son valores enteros enfatiza su naturaleza de punto flotante a los lectores humanos. >>

No Usar Sentencias `goto`

De **[K&R1988] "Chapter 3 – Control Flow – 3.8 Goto and labels"**

<< Formalmente, la sentencia goto nunca es necesaria, y en la práctica es casi siempre fácil escribir código sin ella…>>

<<…Sin embargo, hay algunas situaciones donde los gotos pueden encontrar su lugar. El más común es abandonar el proceso dentro de alguna de la estructura profundamente anidada, por ejemplo salir de dos o ciclos inmediatamente.

La sentencia break no puede ser usada directamente ya que solo sale del ciclo más interno…>>

<< …El código que involucre un goto puede siempre ser escrito sin él, aunque quizás al precio de algunas pruebas repetidas o una variable extra…>>

<< … Con algunas excepciones como las citadas aquí, un código que utilice sentencias goto es en general más difícil de comprender y de mantener que un código sin gotos. Aunque no somos dogmáticos sobre el tema, sí parece que las sentencias goto se deben utilizar en raras oportunidades, o quizás nunca…>>

De **[K&R1988] "Chapter 5 – Pointers and Arrays"**

<<…Los punteros, junto con la sentencia goto, han sido duramente castigados y son conocidos como una manera maravillosa de crear programas imposibles de entender. Pero esto es verdad solo cuando son utilizados negligentemente…

>>

Usar Variables Externas (globales) Solo Cuando se Justifica

Las variables externas son llamadas así porque son definidas fuera de cualquier función. Son también conocidas como variables globales por que son globalmente accesibles.

De **[K&R1988] "Chapter 5 – Pointers and Arrays"**

<< Basarse demasiado en variables externas es peligroso puesto que conduce a programas con conexiones de datos que no son del todo obvias –las variables se pueden cambiar en maneras inesperadas e incluso inadvertidas– y el programa se torna difícil de modificar. La segunda versión del programa "línea más larga" es inferior a la primera, en parte por estas razones, y en parte porque destruye la generalidad de dos útiles funciones al escribir dentro de ellas los nombres de las variables que manipulan. >>

Estructurar los archivos del proyecto correctamente

Conozca la forma correcta de diseñar la distribución de archivos que forman parte de un proyecto. Utilice los archivos encabezados para incluir declaraciones públicas de cualquier tipo salvo definiciones de funciones. Las definiciones de las funciones se escriben en un archivo fuente aparte,

mientras que en el encabezado se ubican los prototipos las funciones públicas. No utilice directivas del tipo:

#include "mal.c"

Diseñe bibliotecas de forma genérica para que pueden volver a utilizarse eviatnado así el "copy-paste" de código.

No Permitir Inclusiones Múltiples de Headers (Archivos Encabezado)

El contenido de los archivos header debe estar "rodeado" por las siguientes directivas al preprocesador:

#ifndef HEADER_*NOMBRE*_YA_INCLUIDO

#define HEADER_*NOMBRE*_YA_INCLUIDO

/* Contenido del archivo encabezado. */

#endif

Esto permite la compilación condicional, evitando incluir más de una vez el contenido del archivo encabezado. Reemplazar *NOMBRE* por el nombre del header.

Diseñar las Funciones Correctamente

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.7 Functions"**

<< Una función proporciona una manera conveniente de encapsular un cómputo, la cual puede entonces ser utilizada sin preocuparse de su implementación. Con funciones correctamente diseñadas, es posible ignorar *cómo* se hace un trabajo; saber *qué* es lo que se hace es suficiente. C hace el tema de funciones fácil, conveniente y eficiente; usted verá a menudo una función corta definida y llamada una única vez, solo porque clarifica una cierta sección del código.>>

Las funciones deben abstraer un proceso, acción, actividad o cálculo. Lo importante para quien invoca a la función es "que" hace pero no "como" lo hace. Para diseñar correctamente una función considerar los siguientes parámetros de diseño:

Ocultamiento de Información

Diseñe las funciones de tal manera que no expongan detalles de implementación. No debe ser conocido por quien invoca la función "como" es que la función realiza la tarea.

Alta Cohesión

Significa que una función debe realizar solo una tarea.

Supóngase que se esta escribiendo un programa que resuelve integrales, la función MostrarResultado deberá solo imprimir un resultado, no calcularlo. En forma similar, la función encargada de calcular este resultado, Integrar, no deberá imprimirlo. Esto no descarta que exista una

función que utilice a ambas funciones para resolver la integral y mostrar su resultado. Tampoco es correcto que dos funciones hagan una tarea a medias. Una manera de estimar la cohesión es por medio de su longitud: ***una función no debe tener más de 20 sentencias o declaraciones***.

Bajo Acomplamiento

Implica que una función tiene poca dependencia con el resto del sistema, para poder utilizar una función debe ser suficiente con conocer su prototipo (su parte pública).

Un ejemplo de alto acoplamiento es una función que retorna un valor modificando una variable global; si el identificador de la variable global cambia, se debe cambiar la función para que siga comportándose correctamente.

Bibliografía

Bibliografía Consultada

- **The C Programming Language**, 2nd Edition [K&R1988]
- **Hugarian Notation**

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp

- **Identation Styles**

http://en.wikipedia.org/wiki/Indent_style

Bibliografía Recomendada

- **The practice of programming**, por Kernighan & Pike, Addison-Wesley
- **Java Coding Style Guide**, por Reddy, Sun Microsystems, Inc.
- **C# Language Specification – Naming guidelines**, ECMA/Microsoft.—

Sistema Planetario – TAD NumeroAstronomico 20051023

Introducción

El NumeroAstronomico permite a programadores del área astronómica manejar cifras muy grandes.

Las siguientes dos secciones son guías y restricciones para el diseño del TAD pero no son ni la especificación y ni la implementación.

Valores

Un valor NumeroAstronomico es un par ordenado formado por una secuencia de hasta 100 dígitos y un indicador de error.

En la implementación, un valor NumeroAstronomico es un par ordenado formado por una secuencia de hasta 100 dígitos y un segundo dato que representará o bien la longitud del número, o bien un código de error.

```
typedef struct {
const char* entero;
int longitudError;
} NumeroAstronomico;
```

El campo entero es un puntero al comienzo de un arreglo que fue definido dinámicamente. Este arreglo contiene cada uno de los dígitos del NumeroAstronomico. Evaluar diferentes posibilidades: dígito más significativo al principio o al final? Guardar el valor numérico del dígito o el código asociado al símbolo? (i.e. el valor cero ó el 48, que es el código ASCII del carácter '0'). Dejar espacio para el *carry* y marcar el *overflow*? Cómo se puede almacenar los diferentes errores y longitud del número en un mismo campo?

Operaciones

Operaciones de Creación

Aplicar malloc para reservar espacio para la estructura y para el arreglo entero dentro de la estructura.

1. **CrearDesdeCadena**: Cadena → NumeroAstronomico

Especificar con y sin precondiciones (pero implementar únicamente sin precondiciones).

2. **CrearDesdeCifraSeguidaDeCeros**: Cifra × CantidadDeCeros → NumeroAstronomico

Especificar con y sin precondiciones (pero implementar únicamente sin precondiciones). Ejemplo:

CrearDesdeCifraSeguidaDeCeros(25, 7) = (250000000, *Ninguno*)

3. **CrearAleatorio**: PróximoNúmeroDeLaSecuenciaAleatoria → NumeroAstronomico

Esta operación tiene precondiciones? En la especificación o en la implementación? Aplicar rand.

En la implementación, no habrá parámetro de entrada, ya que el próximo número de la secuencia está implícito luego de invocar a srand.

Operaciones de Manejo de Errores

4. **EsSecuenciaNula**: NumeroAstronomico → Boolean
5. **EsSecuenciaInvalida**: NumeroAstronomico → Boolean
6. **EsOverflow**: NumeroAstronomico → Boolean
7. **EsPunteroNulo**. Esta operación es propia de la implementación.
8. **GetTipoDeError**: NumeroAstronomico → TipoDeError

TipoDeError = {Ninguno, CadenaNula, CadenaInvalida, Overflow(, PunteroNulo)}

TipoDeError se implementa como un enum. PunteroNulo es propia de la implementación.

9. **EsError**: NumeroAstronomico → Boolean

Esta operación equivale a: EsCadenaNula ∨ EsCadenaInvalida ∨ EsOverflow

Operaciones de Salida

10. **Mostrar**: NumeroAstronomico × GruposEnPrimeraLinea × Flujo → Flujo

Ejemplo:

Sea na = (800700600500400300200100, *Ninguno*) ∈ NumeroAstronomico entonces:

Mostrar(na, 3, $stdout_1$) = $stdout_2$

escribe en stdout las siguientes líneas:

\t\t**800.700.600.**\n

\t\t **500.400.**\n

\t\t **300.200.**\n

\t\t **100**\n

¿Por qué $stdout_1$ es diferente a $stdout_2$? Notar que la primer línea tiene 3 grupos y las siguientes uno menos (como máximo). Analizar la necesidad de precondiciones.

Operaciones Aritméticas

11. **Sumar**: NumeroAstronomico × NumeroAstronomico → NumeroAstronomico

12. **SonIguales**: NumeroAstronomico × NumeroAstronomico → Boolean
13. **EsMenor**: NumeroAstronomico × NumeroAstronomico → Boolean

De Persistencia

Permiten guardar y recuperar valores NumeroAstronomico en memoria externa, en formatos de texto y
binario.

Binario

Definir una representación medianamente eficiente del NumeroAstronomico en disco y en forma binaria.

14. **Read**: Flujo → NumeroAstronomico × Flujo
15. **Write**: NumeroAstronomico × Flujo → Flujo

Texto

El formato de un NumeroAstronómico en un archivo de texto es el siguiente:

secuenciadedigitos#

El dígito más significativo es el primero.

16. **Scan**: Flujo → NumeroAstronomico × Flujo
17. **Print**: NumeroAstronomico × Flujo → Flujo