Introducción al lenguaje C

Programación 3

Instituto de Computación, Facultad de Ingeniería,
Universidad de la República, Uruguay

4 de agosto de 2015

Lenguaje

- Lenguaje: C*
  - Es el lenguaje C, pero sumándole algunas (pocas) cosas de C++
  - Es "ficticio"

Comparación con Modula-2

hola.mod:

```
MODULE hola;
FROM InOut IMPORT;


BEGIN
    WriteLine("¡Hola, mundo!");

END hola.
```

hola.cpp:

```
#include <stdio.h>

int main()
{
    printf("¡Hola, mundo!\n");
    return 0;
}
```

Comparación con Modula-2

hola.mod:

```
MODULE hola;
FROM InOut IMPORT;


BEGIN
    WriteLine("¡Hola, mundo!");

END hola.
```

hola.cpp:

```
#include <stdio.h>

int main()
{
    printf("¡Hola, mundo!\n");
    return 0;
}
```

`main` es una función especial, a partir de la cual comienza la ejecución del programa

Compilación

- Módulos de definición con extensión **.h**

Compilación

- Módulos de definición con extensión **.h**
- Módulos de implementación con extensión **.cpp**

Compilación

- Módulos de definición con extensión **.h**
- Módulos de implementación con extensión **.cpp**
- Se compila y linkedita con **g++** (compilador de C++)

Compilación

- Módulos de definición con extensión **.h**
- Módulos de implementación con extensión **.cpp**
- Se compila y linkedita con **g++** (compilador de C++)
  - Compilo *mod1.cpp*, *mod2.cpp* y *mod3.cpp*:

    ```
    g++ -c mod1.cpp
    g++ -c mod2.cpp
    g++ -c mod3.cpp
    ```

    generando los archivos *mod1.o*, *mod2.o* y *mod3.o*

Compilación

- Módulos de definición con extensión **.h**
- Módulos de implementación con extensión **.cpp**
- Se compila y linkedita con **g++** (compilador de C++)
  - Compilo *mod1.cpp*, *mod2.cpp* y *mod3.cpp*:
    ```
    g++ -c mod1.cpp
    g++ -c mod2.cpp
    g++ -c mod3.cpp
    ```
    generando los archivos *mod1.o*, *mod2.o* y *mod3.o*
  - Los linkedito:
    ```
    g++ mod1.o mod2.o mod3.o -o programa
    ```
    generando el ejecutable *programa*

Compilación

- Módulos de definición con extensión **.h**
- Módulos de implementación con extensión **.cpp**
- Se compila y linkedita con **g++** (compilador de C++)
  - Compilo *mod1.cpp*, *mod2.cpp* y *mod3.cpp*:
    ```
    g++ -c mod1.cpp
    g++ -c mod2.cpp
    g++ -c mod3.cpp
    ```
    generando los archivos *mod1.o*, *mod2.o* y *mod3.o*
  - Los linkedito:
    ```
    g++ mod1.o mod2.o mod3.o -o programa
    ```
    generando el ejecutable *programa*
  - Otra opción:
    ```
    g++ mod1.cpp mod2.cpp mod3.cpp -o programa
    ```

Tipos de datos elementales

- Entero: `int`

Tipos de datos elementales

- Entero: `int`
- Caracter: `char`

Tipos de datos elementales

- Entero: `int`
- Caracter: `char`
- Real: `float`

Tipos de datos elementales

- Entero: `int`
- Caracter: `char`
- Real: `float`
- Booleano: `bool` (de C++)

Tipos de datos elementales

- Entero: `int`
- Caracter: `char`
- Real: `float`
- Booleano: `bool` (de C++)

Tipos de datos elementales

- Entero: `int`
- Caracter: `char`
- Real: `float`
- Booleano: `bool` (de C++)

Ejemplos:

```
int i;
char c;
float f;
bool b;

i = 1;
b = false;
```

Comentarios

```
/* comentario
de
varias
lineas */
int i = 1; /* asigno 1 a i */
char c; // comentario de una linea (C++)
float f;
// otro comentario
```

Expresiones I

- Operador de asignación: =

```
int a;
int b = 2;

a = 7;
a = b;
```

Expresiones I

- Operador de asignación: =

```
int a;
int b = 2;

a = 7;
a = b;
```

  - La asignación retorna un valor, por lo que es válido: $a = b = 9$

Expresiones I

- Operador de asignación: =

```
int a;
int b = 2;

a = 7;
a = b;
```

  - La asignación retorna un valor, por lo que es válido: $a = b = 9$
  - **Error común:** confundir con comparación booleana de Modula-2

Expresiones II

- Operadores de comparación: ==, !=, <, <=, > y >=

Expresiones II

- Operadores de comparación: ==, !=, <, <=, > y >=
- Operadores lógicos: &&, || y !

Expresiones II

- Operadores de comparación: ==, !=, <, <=, > y >=
- Operadores lógicos: &&, || y !
- Operadores aritméticos: +, -, *, / y %

Expresiones II

- Operadores de comparación: ==, !=, <, <=, > y >=
- Operadores lógicos: &&, || y !
- Operadores aritméticos: +, -, *, / y %

Expresiones II

- Operadores de comparación: ==, !=, <, <=, > y >=
- Operadores lógicos: &&, || y !
- Operadores aritméticos: +, -, *, / y %

Precedencia:

```
a+1 < b && c == 9*d || e < 7
```

equivale a:

```
(((a+1) < b) && (c == (9*d))) || (e < 7)
```

Expresiones III

- Incremento y decremento: ++ y --

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

```
int a = 1;
int b, c;
b = ++a;
c = a++;
```

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

```
int a = 1;
int b, c;
b = ++a;
c = a++;
```

Valores finales:

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

```
int a = 1;
int b, c;
b = ++a;
c = a++;
```

Valores finales:

- $a \rightarrow 3$

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

```
int a = 1;
int b, c;
b = ++a;
c = a++;
```

Valores finales:

- $a \rightarrow 3$
- $b \rightarrow 2$

Expresiones III

- Incremento y decremento: ++ y --
  - *++a* incrementa el valor de *a* y retorna su valor **luego** del incremento
  - *a++* incrementa el valor de *a* y retorna su valor **antes** del incremento
  - Análogo para decrementar

```
int a = 1;
int b, c;
b = ++a;
c = a++;
```

Valores finales:

- $a \rightarrow 3$
- $b \rightarrow 2$
- $c \rightarrow 2$

Estructuras de control I

- Selección

Estructuras de control I

- Selección
  - Sentencia `if`:

```
if (6 <= valor && valor <= 12) {
    printf ("Aprobado");
    cantidad_aprobados++;
} else if (valor >= 3)
    printf ("Examen");
else if (valor >= 0)
    printf ("Reprobado");
else
    printf ("Valor incorrecto");
```

Estructuras de control II

- Selección
  - Sentencia `switch`:

```
switch (valor) {
    case 6: case 7: case 8: case 9: case 10: case 11:↩
         case 12:
        printf ("Aprobado");
        cantidad_aprobados++;
        break;
    case 3: case 4: case 5:
        printf ("Examen");
        break;
    case 0: case 1: case 2:
        printf ("Reprobado");
        break;
    default:
        printf ("Valor incorrecto");
}
```

Estructuras de control III

- Iteración

Estructuras de control III

- Iteración
  - Sentencia `while`:

```
while (condicion)
    cuerpo
```

```c
int i = 0;
while (i < 10) {
    printf ("*");
    i++;
}
```

Estructuras de control IV

- Iteración
  - Sentencia `for`:

```
for (inicio; condicion; paso)
    cuerpo
```

```
for (int i = 0; i < 10; i++)
    printf ("*");
```

Tipos de datos estructurados I

- Enumerados: `enum`

```
enum mes {enero, febrero, marzo, abril, mayo, junio, ↩
    julio, agosto, setiembre, octubre, noviembre, ↩
    diciembre};
mes este_mes = agosto;
```

Tipos de datos estructurados II

- Estructuras: `struct`

```
struct fecha {
    int f_dia;
    mes f_mes;
    int f_anio;
};
```

Tipos de datos estructurados II

- Estructuras: `struct`

```
struct fecha {
    int f_dia;
    mes f_mes;
    int f_anio;
};
```

  - Se usa . para acceder a los miembros:

```
fecha hoy;

hoy.f_dia = 4;
hoy.f_mes = agosto;
hoy.f_anio = 2015;

int dia_hoy = hoy.f_dia;
```

Tipos de datos estructurados III

- Punteros

```
int * p; // p es un puntero a un número entero

int i;
p = &i; // p apunta a la dirección de i
*p = 10; // i toma el valor 10

int* p2;
p2 = p; // p2 apunta a la dirección de i

p = NULL; // así se deja en NULL un puntero

p = new int; // así se pide memoria
delete p; // se libera la memoria apuntada por p
```

Tipos de datos estructurados III

- Punteros

```
int * p; // p es un puntero a un número entero

int i;
p = &i; // p apunta a la dirección de i
*p = 10; // i toma el valor 10

int* p2;
p2 = p; // p2 apunta a la dirección de i

p = NULL; // así se deja en NULL un puntero

p = new int; // así se pide memoria
delete p; // se libera la memoria apuntada por p
```

  - Notar que estaría mal liberar la memoria asignada a *p2* ya que fue pedida de forma estática

Tipos de datos estructurados III

- Punteros

```
int * p; // p es un puntero a un número entero

int i;
p = &i; // p apunta a la dirección de i
*p = 10; // i toma el valor 10

int* p2;
p2 = p; // p2 apunta a la dirección de i

p = NULL; // así se deja en NULL un puntero

p = new int; // así se pide memoria
delete p; // se libera la memoria apuntada por p
```

  - Notar que estaría mal liberar la memoria asignada a *p2* ya que fue pedida de forma estática
  - `new` y `delete` son de C++

Tipos de datos estructurados IV

- Punteros

```
int *p3; // el asterisco puede ir junto a la variable (a ↩
    la izquierda)

int* p4, p5;
```

Tipos de datos estructurados IV

- Punteros

```
int *p3; // el asterisco puede ir junto a la variable (a ↩
    la izquierda)

int* p4, p5;
```

  - ¿p5 es un puntero? **No**, la declaración es como la siguiente:

```
int* p4;
int p5;
```

Tipos de datos estructurados IV

- Punteros

```
int *p3; // el asterisco puede ir junto a la variable (a ↩
    la izquierda)

int* p4, p5;
```

  - ¿p5 es un puntero? **No**, la declaración es como la siguiente:

```
int* p4;
int p5;
```

  - Es como si el asterisco se pegara a la variable y no al tipo

Tipos de datos estructurados IV

- Punteros

```
int *p3; // el asterisco puede ir junto a la variable (a ↩
    la izquierda)

int* p4, p5;
```

  - ¿p5 es un puntero? **No**, la declaración es como la siguiente:

```
int* p4;
int p5;
```

  - Es como si el asterisco se pegara a la variable y no al tipo
  - Si queremos dos punteros:

```
int* p4, * p5;
```

Tipos de datos estructurados V

- Punteros y estructuras

```
(*puntero_fecha).f_dia = 4;
puntero_fecha->f_dia = 4; // más fácil
```

Tipos de datos estructurados VI

- Arreglos

Tipos de datos estructurados VI

- Arreglos
  - Varios objetos del mismo tipo puestos consecutivamente en memoria

Tipos de datos estructurados VI

- Arreglos
  - Varios objetos del mismo tipo puestos consecutivamente en memoria
  - El primer elemento está en bajo el índice 0

Tipos de datos estructurados VI

- Arreglos
  - Varios objetos del mismo tipo puestos consecutivamente en memoria
  - El primer elemento está en bajo el índice 0
  - Estáticos:

```
int arr[2]; // valores posibles arr[0] y arr[1]

int vector[5] = {1, 2, 3, 4, 5};
int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};

int suma = 0;
for (int i = 0; i < 5; i++)
    suma += vector[i];
```

Tipos de datos estructurados VI

- Arreglos
  - Varios objetos del mismo tipo puestos consecutivamente en memoria
  - El primer elemento está en bajo el índice 0
  - Estáticos:

```
int arr[2]; // valores posibles arr[0] y arr[1]

int vector[5] = {1, 2, 3, 4, 5};
int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};

int suma = 0;
for (int i = 0; i < 5; i++)
    suma += vector[i];
```

  - Dinámicos:

```
int* vector = new int[10];
delete [] vector;
```

Tipos de datos estructurados VII

- Arreglos
  - Tener en cuenta que ni C ni C++ chequean que el índice esté dentro del rango permitido. Te deja así acceder a otra dirección de memoria, y **a veces** puede dar segmentation fault.

Conversión de tipos I

- La mayoría de las conversiones son implícitas

```
float vf = 1.6;
int vi = 1 + vf; // vi = 2 (float se trunca)
vi = 1 + vf + vf; // vi = 4 (cast al "más grande")
vi = vi + true; // vi = 5 (true es 1)
vi = vi + false; // vi = 5 (false es 1)
vi = 'a' + 1; // vi = 98 (valor ASCII)
char vc = 'a' + 1; // vc = 'b'
vf = 1.5 + vi // vf = 99.500000
bool vb = 237; // vb = true (0 es false, otro true)
vf = 3 / 2; // vf = 1.000000
```

- Se puede hacer cast explícito

```
vf = (float)3 / 2; // vf = 1.500000
```

Conversión de tipos II

- ¿Cuál es el valor de `res`?

```
int res;
int i = 5 - 4.3;
bool b = 100.1;

if (i = 0)
    res = b + 100.9;
else
    res = b + i;
```

- ¿El resultado es 1?
- **Error común:** haber puesto = en lugar de ==. Aunque el resultado es 1, si cambiamos por == obtenemos 101.

Listas I

- Tipo de datos para una lista de enteros:

```
struct nodo {
    int dato;
    nodo * sig;
};
typedef nodo* lista; // lista es un sinónimo de nodo*
```

- Agregar un elemento:

```
lista mi_lista = new nodo;
mi_lista->dato = 1;
mi_lista->sig = NULL;
```

Listas II

- Agregar otro elemento:

```
mi_lista->sig = new nodo;
mi_lista->sig->dato = 1;
mi_lista->sig->sig = NULL;
```

- Imprimir el contenido:

```
for (lista l = mi_lista; l != NULL; l = l->sig)
    printf ("-> %d ", l->dato);
printf ("\n");
```

- Sacar el primer elmento:

```
lista aux = mi_lista->sig;
delete mi_lista;
mi_lista = aux;
```

Funciones I

- Funciones

```
lista agregar(int dato, lista l) {
    lista res = new nodo;
    res->dato = dato;
    res->sig = l;
    return res;
}
```

  - Se puede invocar así:

```
mi_lista = agregar(3, mi_lista);
```

Funciones II

- Procedimientos

```
void imprimir(lista l) {
    for (; l != NULL; l = l->sig)
        printf("-> %d ", l->dato);
    printf("\n");
}
```

Funciones III

- Las funciones no se puede anidar
- En C todos los parámetros se pasan por valor
  - En C++ (y C*) existe el pasaje por referencia (&)

```
void eliminar (lista & l) {
    lista tmp = l->sig;
    delete l;
    l = tmp;
}
```

    - No confundir con el operador & de punteros
  - En C el pasaje por referencia se simula utilizando punteros

```
void inc(int *i) {
    (*i) += 1;
}
```

    - Se puede invocar así: `inc(&valor);`

Módulos I

- Para poder invocar una función el compilador necesita haber visto al menos su declaración.

```
int pordos(int valor) {
    return suma(valor, valor);
}

int suma(int valor1, int valor2) {
    return valor1 + valor2;
}
```

Incorrecto

Módulos II

- Para poder invocar una función el compilador necesita haber visto al menos su declaración.

```
int suma(int, int);

int pordos(int valor) {
    return suma(valor, valor);
}

int suma(int valor1, int valor2) {
    return valor1 + valor2;
}
```

Correcto

Módulos III

- No es obligatorio que la implementacion se encuentre en el mismo archivo

Módulos III

- No es obligatorio que la implementacion se encuentre en el mismo archivo

Módulos III

- No es obligatorio que la implementacion se encuentre en el mismo archivo

mod1.cpp:

```
int suma(int, int);

int pordos(int valor) {
    return suma(valor, valor);
}

int main() {
    int a = pordos(4);
    return 0;
}
```

Módulos III

- No es obligatorio que la implementacion se encuentre en el mismo archivo

mod1.cpp:

```
int suma(int, int);

int pordos(int valor) {
    return suma(valor, valor);
}

int main() {
    int a = pordos(4);
    return 0;
}
```

mod2.cpp:

```
int suma(int v1, int v2) {
    return v1 + v2;
}
```

Módulos III

- No es obligatorio que la implementacion se encuentre en el mismo archivo

mod1.cpp:

```
int suma(int, int);

int pordos(int valor) {
    return suma(valor, valor);
}

int main() {
    int a = pordos(4);
    return 0;
}
```

mod2.cpp:

```
int suma(int v1, int v2) {
    return v1 + v2;
}
```

- Al linkeditar se encuentra la implementación para cada declaración

`g++ mod1.cpp mod2.cpp -o programa`

Módulos IV

- Para hacer un TAD pongo las declaraciones en un archivo .h

par.h:

```
#ifndef PAR_H
#define PAR_H

struct par_struct;
typedef par_struct* par;

par crear(int, int);
int primero(par);
int segundo(par);

#endif
```

par.cpp:

```
#include "par.h"

struct par_struct {
    int pri, seg;
};

par crear(int i1, int i2) {
    par ret = new par_struct;
    ret->pri = i1;
    ret->seg = i2;
    return ret;
}

int primero(par p) {
    return p->pri;
}
```

Módulos V

- Para usar el TAD importo el .h
  prog.cpp

```
#include "par.h"
#include <stdio.h>

int main() {
    par mi_par = crear(4, 5);
    int pri = primero(mi_par);
    int seg = segundo(mi_par);

    printf("(%d, %d)\n", pri, seg);
    return 0;
}
```

Entrada/Salida I

- `printf`: impresión en salida estándar

```
printf("hola mundo\n");
printf("-> %d, ", l->dato);
printf("(%d, %d)\n", pri, seg);
```

  - Función "rara" que recibe una cantidad variable de parámetros
  - El primer parámetro es la **cadena de texto de formato**
  - El resto depende de los **especificadores de formato** que se encuentren en el primero
  - Especificadores:

    %d int
    %c char
    %f float
    %s char*

  - Algunas secuencias de escape: \', \", \\, \n, \t

Entrada/Salida II

- `scanf`: lectura en la entrada estándar
  - La cadena de texto de formato es igual que en `printf`
  - Pero los parámetros tienen que ser punteros (para poder modificarlos)

```
int val, cant;
char str [10];
cant = scanf("%d-%s", &val, str);
```

`89-bla` → cant = 2, val = 89, str = "bla"
`89bla` → cant = 1, val = 89, str = ??
`bla` → cant = EOF, val = ??, str = ??

C*: C con algunas cosas de C++

En resumen, C* es C pero con las siguientes cosas de C++:

- `new` y `delete`

C*: C con algunas cosas de C++

En resumen, C* es C pero con las siguientes cosas de C++:

- `new` y `delete`
- Comentarios en línea

C*: C con algunas cosas de C++

En resumen, C* es C pero con las siguientes cosas de C++:

- `new` y `delete`
- Comentarios en línea
- Declaración de tipos como en C++ para registros y enumerados

C*: C con algunas cosas de C++

En resumen, C* es C pero con las siguientes cosas de C++:

- `new` y `delete`
- Comentarios en línea
- Declaración de tipos como en C++ para registros y enumerados
- Pasaje por referencia

C*: C con algunas cosas de C++

En resumen, C* es C pero con las siguientes cosas de C++:

- `new` y `delete`
- Comentarios en línea
- Declaración de tipos como en C++ para registros y enumerados
- Pasaje por referencia
- `bool`

Más información

- Buscar en Internet (recuerden las cosas que sí usamos de C++)

Más información

- Buscar en Internet (recuerden las cosas que sí usamos de C++)
- Probar en la computadora qué pasa con cierto comportamiento que tengan duda

Más información

- Buscar en Internet (recuerden las cosas que sí usamos de C++)
- Probar en la computadora qué pasa con cierto comportamiento que tengan duda
- Libro *Cómo programar en C++*, Deitel y Deitel

Más información

- Buscar en Internet (recuerden las cosas que sí usamos de C++)
- Probar en la computadora qué pasa con cierto comportamiento que tengan duda
- Libro *Cómo programar en C++*, Deitel y Deitel
- Consulten en EVA