Introducción al Lenguaje C (segunda parte)

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

August 6, 2010

Cast en C

- La mayoría de las conversiones son implícitas

Cast en C

- La mayoría de las conversiones son implícitas

Cast en C

- La mayoría de las conversiones son implícitas

```
float vf = 1.6;
int vi = 1 + vf;
```

Cast en C

- La mayoría de las conversiones son implícitas

```
float vf = 1.6;
int vi = 1 + vf;
```

$\rightarrow$ `vi = 2` (`float` se trunca)

Cast en C

- La mayoría de las conversiones son implícitas

```
float vf = 1.6;
int vi = 1 + vf;
vi = 1 + vf + vf;
```

$\rightarrow$ `vi = 2` (`float` se trunca)

Cast en C

- 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 "mas grande")
```

Cast en C

- 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 "mas grande")
vi = vi + true;
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;
```

Cast en C

- 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 "mas grande")
vi = vi + true;           → vi = 5 (true es 1)
vi = vi + false;          → vi = 5 (false es 0)
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
vi = 'a' + 1;
```

Cast en C

- 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 "mas grande")
vi = vi + true;           → vi = 5 (true es 1)
vi = vi + false;          → vi = 5 (false es 0)
vi = 'a' + 1;             → vi = 98 (valor ASCII)
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
vi = 'a' + 1;           → vi = 98 (valor ASCII)
char vc = 'a' + 1;
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
vi = 'a' + 1;           → vi = 98 (valor ASCII)
char vc = 'a' + 1;      → vc = 'b'
```

Cast en C

- 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 "mas grande")
vi = vi + true;           → vi = 5 (true es 1)
vi = vi + false;          → vi = 5 (false es 0)
vi = 'a' + 1;             → vi = 98 (valor ASCII)
char vc = 'a' + 1;        → vc = 'b'
vf = 1.5 + vi;
```

Cast en C

- 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 "mas grande")
vi = vi + true;           → vi = 5 (true es 1)
vi = vi + false;          → vi = 5 (false es 0)
vi = 'a' + 1;             → vi = 98 (valor ASCII)
char vc = 'a' + 1;        → vc = 'b'
vf = 1.5 + vi;            → vf = 99.500000
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
vi = 'a' + 1;           → vi = 98 (valor ASCII)
char vc = 'a' + 1;      → vc = 'b'
vf = 1.5 + vi;          → vf = 99.500000
bool vb = 237;
```

Cast en C

- 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 "mas grande")
vi = vi + true;           → vi = 5 (true es 1)
vi = vi + false;          → vi = 5 (false es 0)
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)
```

Cast en C

- 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 "mas grande")
vi = vi + true;           → vi = 5 (true es 1)
vi = vi + false;          → vi = 5 (false es 0)
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;
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
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

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
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

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
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;
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
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
```

Cast en C

- 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 "mas grande")
vi = vi + true;         → vi = 5 (true es 1)
vi = vi + false;        → vi = 5 (false es 0)
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
```

Cast en C (2)

- 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;
```

Cast en C (2)

- 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

Cast en C (2)

- 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!!

Cast en C (2)

- 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!!
- De haber puesto == en lugar de = el resulado es 101

Usando lo visto en la clase anterior...

- Tipo de datos para una lista de enteros:

```
struct nodo {
  int dato;
  nodo* sig;
};
```

Usando lo visto en la clase anterior...

- Tipo de datos para una lista de enteros:

```
struct nodo {
  int dato;
  nodo* sig;
};
```

Usando lo visto en la clase anterior...

- Tipo de datos para una lista de enteros:

```
struct nodo {
  int dato;
  nodo* sig;
};

typedef nodo* lista;
```

Usando lo visto en la clase anterior...

- Tipo de datos para una lista de enteros:

```
struct nodo {
  int dato;
  nodo* sig;
};
```

`typedef nodo* lista;` $\rightarrow$ `lista` es un sinónimo de `nodo*`

Usando lo visto en la clase anterior...

- 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;
```

Usando lo visto en la clase anterior...

- 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;
```

- Agregar otro elemento:

```
mi_lista->sig = new nodo;
mi_lista->sig->dato = 2;
mi_lista->sig->sig = NULL;
```

Usando lo visto en la clase anterior... (2)

- Imprimir el contenido:

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

Usando lo visto en la clase anterior... (2)

- Imprimir el contenido:

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

- Quitar el primer elemento:

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

Funciones

- Funciones

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

Funciones

- Funciones

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

  - Se invoca: `mi_lista = agregar(3,mi_lista);`

Funciones

- Funciones

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

  - Se invoca: `mi_lista = agregar(3,mi_lista);`

- Procedimientos == Funciones que retornan `void`

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

Funciones (2)

- Las funciones no se pueden anidar

Funciones (2)

- Las funciones no se pueden anidar
- En C todos los parámetros se pasan por valor

Funciones (2)

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

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

Funciones (2)

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

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

    - No confundir con el operador & de punteros

Funciones (2)

- Las funciones no se pueden anidar
- En C todos los parámetros se pasan por valor
  - En C++ (y C*) existe 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;
}
```

Funciones (2)

- Las funciones no se pueden anidar
- En C todos los parámetros se pasan por valor
  - En C++ (y C*) existe 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 invoca: `inc(&valor);`

Módulos

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

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

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

Incorrecto

Módulos

- Para poder invocar una función el compilador necesita haber visto anteriormente 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 (2)

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

Módulos (2)

- No es obligatorio que la implementación 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 valor1
        , int valor2) {
 return valor1 + valor2;
}
```

Módulos (2)

- No es obligatorio que la implementación 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 valor1
        , int valor2) {
 return valor1 + valor2;
}
```

- El linkeditor encuentra la implementación para cada declaración

```
g++ mod1.cpp mod2.cpp -o prog.exe
```

Módulos (3)

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

Módulos (3)

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

par.h:

```
#ifndef PAR_H
#define PAR_H

struct par;

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

#endif
```

par.cpp:

```
#include "par.h"

struct par {
  int pri, seg;
};
par* crear(int i1, int i2){
  par* ret = new par;
  ret->pri = i1;
  ret->seg = i2;
  return ret;
}
int primero(par *p){
  return p->pri;
}
```

Módulos (4)

- Para usar el TAD importo el `.h`

Módulos (4)

- 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

- `printf`: impresión en salida estándar

Entrada/Salida

- `printf`: impresión en salida estándar

```
printf("hola mundo\n");
```

Entrada/Salida

- `printf`: impresión en salida estándar

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

Entrada/Salida

- `printf`: impresión en salida estándar

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

Entrada/Salida

- `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

Entrada/Salida

- `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 formato**

Entrada/Salida

- `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 formato**
  - el resto depende de los **especificadores de formato** que se encuentren en el primero

Entrada/Salida

- `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 formato**
  - el resto depende de los **especificadores de formato** que se encuentren en el primero
  - especificadores:
    %d $\rightarrow$ int
    %c $\rightarrow$ char
    %f $\rightarrow$ float
    %s $\rightarrow$ char*

Entrada/Salida

- `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 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` y `\t`

Entrada/Salida (2)

- `scanf`: lectura de entrada estándar

Entrada/Salida (2)

- `scanf`: lectura de entrada estándar
  - la cadena de formato es igual que para `printf`

Entrada/Salida (2)

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

Entrada/Salida (2)

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

Entrada/Salida (2)

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

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

Entrada/Salida (2)

- `scanf`: lectura de entrada estándar
  - la cadena de formato es igual que para `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` $\rightarrow$ `cant = 2,` `val = 89,` `str = "bla"`

Entrada/Salida (2)

- `scanf`: lectura de entrada estándar
  - la cadena de formato es igual que para `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 = ??
```

Entrada/Salida (2)

- `scanf`: lectura de entrada estándar
  - la cadena de formato es igual que para `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 = ??
```

Entrada/Salida (3)

- Archivos
  - `fopen` (`"r"` - lectura, `"w"` - escritura, `"a"` - añadir)
  - `fclose`
  - `fprintf`
  - `fscanf`

Entrada/Salida (3)

- Archivos
  - `fopen` (`"r"` - lectura, `"w"` - escritura, `"a"` - añadir)
  - `fclose`
  - `fprintf`
  - `fscanf`

```
FILE* fp = fopen("hola.txt", "w");
if (fp != NULL) {
  fprintf(fp, "hola %s", "mundo");
  fclose(fp)
}
```

Entrada/Salida (4)

- Entrada/Salida de C++
  - `cin`
  - `cout`
  - `cerr`

Entrada/Salida (4)

- Entrada/Salida de C++
  - cin
  - cout
  - cerr

```
#include <iostream>
using namespace std;

int main(){
    int a;
    cin >> a;
    cout << "valor: " << a;
    return 0;
}
```