UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Telecomunicación

DISEÑO DESCENDENTE: SUBPROGRAMAS

Tema 3

Programación I

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

DISEÑO DESCENDENTE: ABSTRACCIÓN

- La **ABSTRACCIÓN** es la herramienta mental que nos permite **analizar**, **comprender** y **construir** sistemas complejos.
- Identificamos y denominamos conceptos abstractos con significado. Aplicamos **Refinamientos Sucesivos**.

DISEÑO DESCENDENTE: ABSTRACCIÓN

- La **ABSTRACCIÓN** es la herramienta mental que nos permite **analizar**, **comprender** y **construir** sistemas complejos.
- Identificamos y denominamos conceptos abstractos con significado. Aplicamos **Refinamientos Sucesivos**.

COCHE

CARROCERIA

BASTIDOR

ASIENTOS

MOTOR

PANEL DE CONTROL

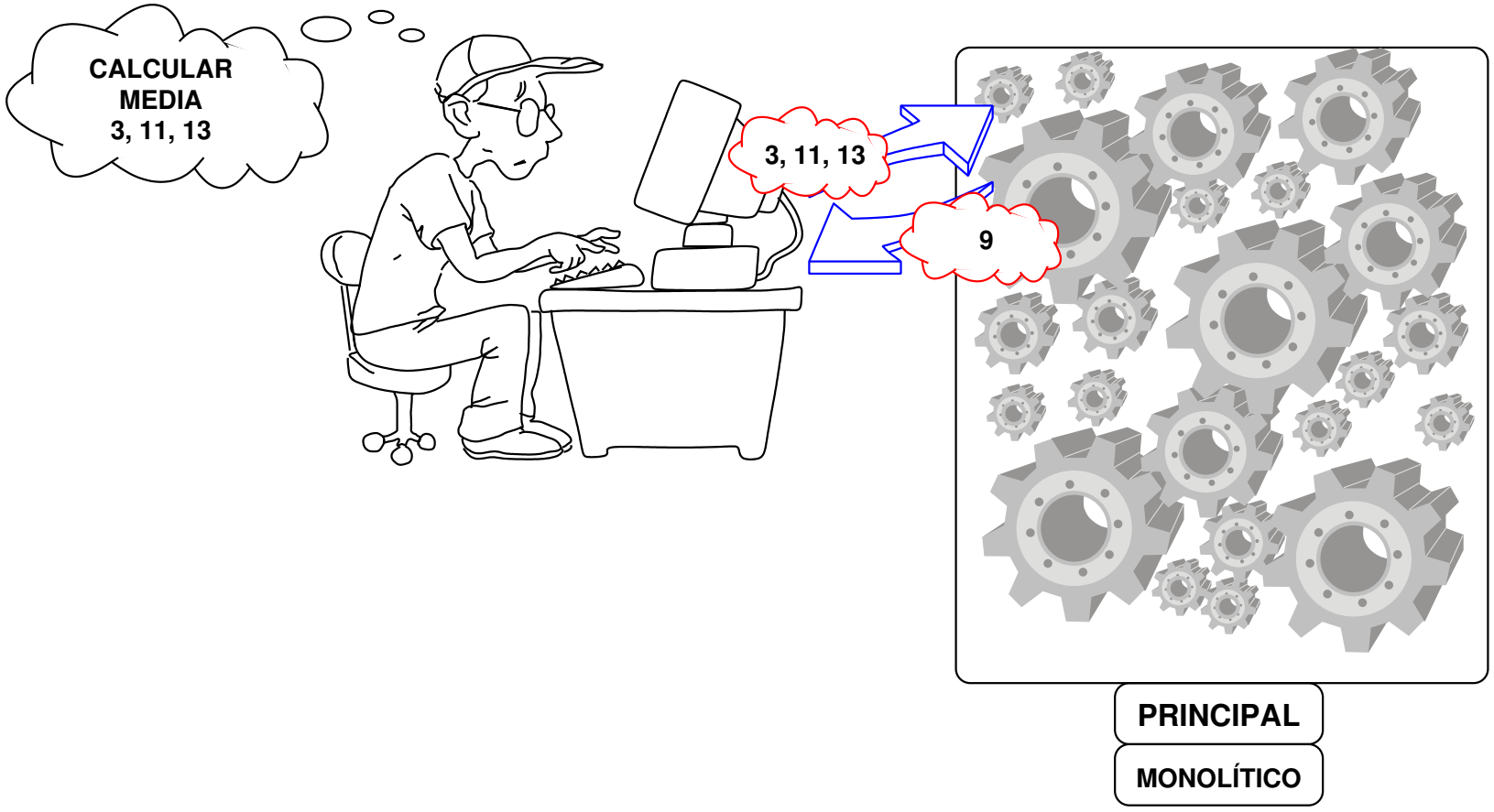
- DISEÑO DESCENDENTE: ABSTRACCIÓN Y REFINAMIENTOS
  - Divide un PROBLEMA en SUBPROBLEMAS
  - Asocia a cada subproblema un SUBPROGRAMA
  - Utilizamos ABSTRACCIÓN:
    - **NO** importa como se resuelve un subproblema
    - Nos centramos en la FUNCIONALIDAD del subprograma
    - Que subproblema resuelve, que datos necesita, que datos produce
    - Bajo que condiciones se debe ejecutar
  - Aplicamos REFINAMIENTOS SUCESIVOS
  - Objetivo: división en SUBPROGRAMAS INDEPENDIENTES
    - Funcionalidad clara y bien definida
    - Aislados: minimizar las dependencias con otros subprogramas
    - Transferencia de información simple y bien definida

- DISEÑO DESCENDENTE: INTERFAZ (interface)
  - Genéricamente define la interacción entre dos entidades.
  - La forma en que se comunican y cooperan dos subprogramas.
  - Define la transferencia de información entre subprogramas.
  - Debemos considerar:
    - Que información necesita para resolver el problema
    - Que información produce como resultado de la solución del problema
    - Bajo que condiciones se realiza el intercambio de información
  - Cuanto más simple sea el interfaz:
    - El subprograma estará más aislado del entorno
    - El subprograma será más fácil de utilizar
    - Menos errores se producirán en su utilización

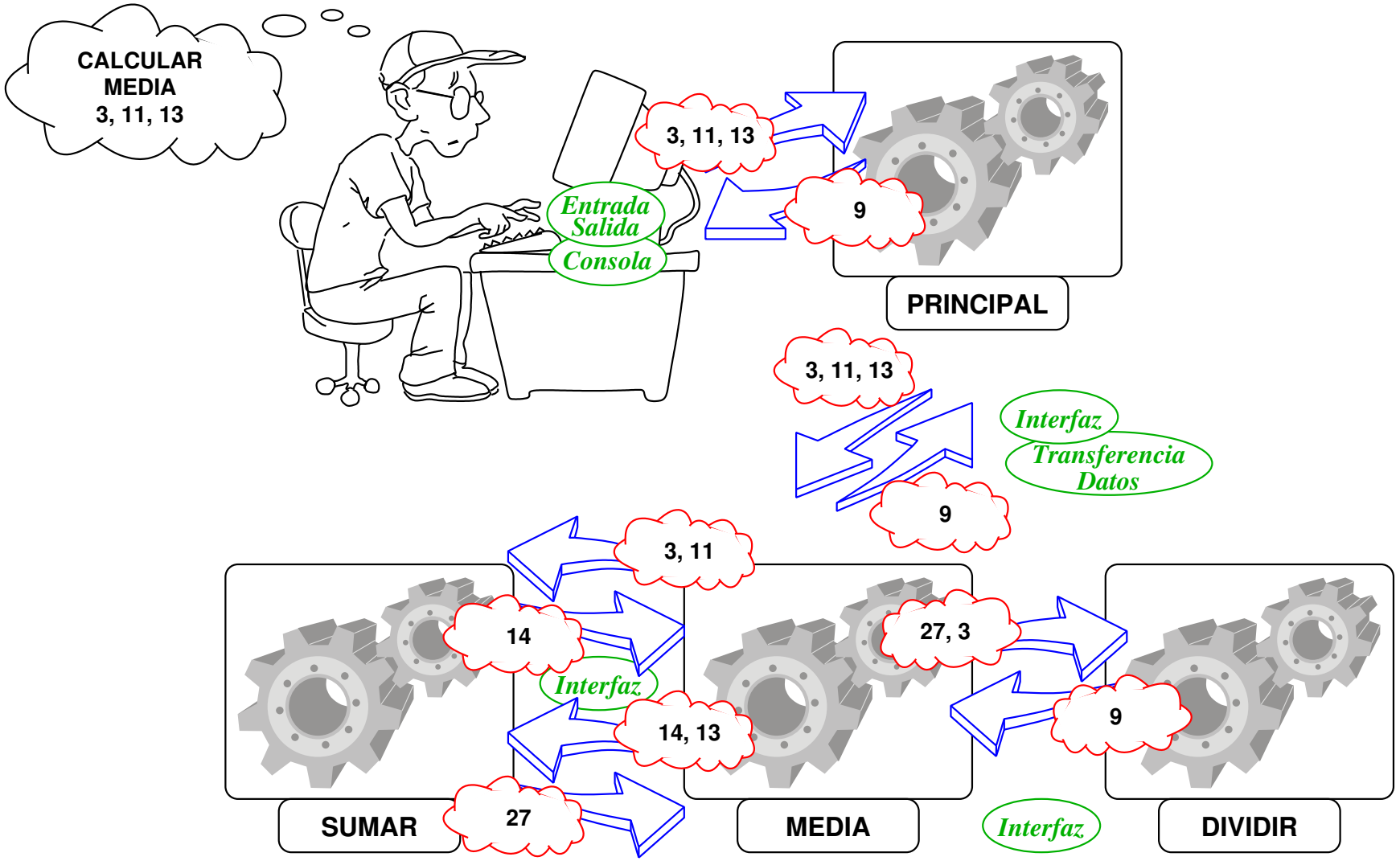
DISEÑO DESCENDENTE: VENTAJAS

- Los subprogramas encapsulan y aislan las diferentes tareas que componen un programa.
  - Simplificación en el diseño y solución del programa.
  - Si el método para solucionar una tarea debe cambiar, el aislamiento evita que dicho cambio influya en las otras tareas.
  - Permite que el programador esté concentrado en la solución individual del subproblema concreto.
- Simplifica la comprensión y legibilidad del programa.
- Facilita la detección y corrección de errores (depuración).
- Posibilidad de reutilización del subprograma en otro contexto.

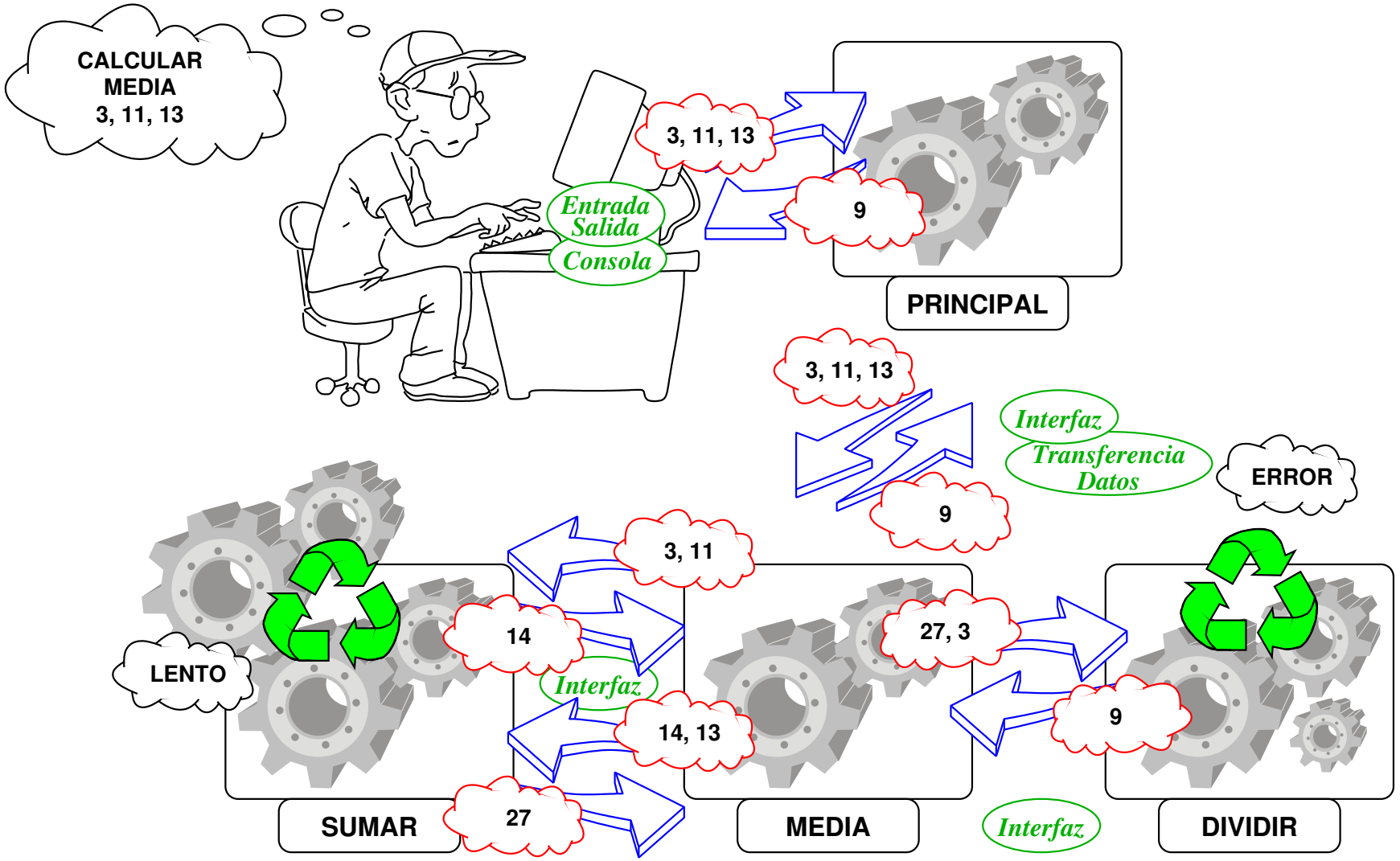
DISEÑO DESCENDENTE: DIVISIÓN MODULAR



DISEÑO DESCENDENTE: DIVISIÓN MODULAR



DISEÑO DESCENDENTE: DIVISIÓN MODULAR



Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

SUBPROGRAMAS

- **Definición** de subprogramas:
  - La cabecera de un subprograma especifica el tipo del valor devuelto (o `void`), el nombre del subprograma y los parámetros formales.
  - El cuerpo del subprograma especifica la secuencia de **acciones** que resuelven un subproblema. Define sus propias variables locales de trabajo.
- Donde sea necesaria la resolución de dicho subproblema, se realizará una **invocación** (llamada) al subprograma, especificando el nombre del subprograma y los parámetros actuales.
- La transferencia de información en la invocación se realiza a través de los parámetros:
  - **Parámetros Formales**: aparecen en la definición del subprograma.
  - **Parámetros Actuales**: aparecen en la invocación al subprograma.
- La definición de un subprograma debe aparecer **antes** de su invocación.
- Los subprogramas proporcionan abstracción de procesamiento y de operaciones:
  - **Funciones**: calculan **un único** valor a partir de la información de entrada.
  - **Procedimientos**: procesamiento **general** de información.

```
// Definición de Función
// Parámetros formales
int menor (int x, int y)
{
    int z = x; // Vble Local
    if (y < z) {
        z = y;
    }
    return z; // Única,Última
}
```

```
// Def. de Procedimiento
// Parámetros formales
void ordenar (int& x, int& y)
{
    if (x > y) {
        int z = x; // V. Local
        x = y;
        y = z;
    }
}
```

```
// - Principal ----
int main ()
{
    int a, b; // Vbles Locales
    cin >> a >> b;
    int c = menor(a, b); // Invocación
    ordenar(a, b); // Par. actuales
    bool ok = (c == a);
}
```

DEFINICIÓN DE SUBPROGRAMAS

- Definición de Funciones: calculan **un único** valor a partir de la información de entrada.
  - La cabecera de una función especifica el tipo del valor devuelto, el nombre del subprograma y los parámetros formales.
  - El cuerpo de una función sólo debe tener una única sentencia `return`, y será la última sentencia del cuerpo de la función.
  - El valor resultado de la función es el resultado de evaluar la expresión de la sentencia `return`.
  - **No está permitida** cualquier otra utilización de la sentencia `return`.
- Definición de Procedimientos: procesamiento **general** de información.
  - La cabecera de un procedimiento especifica que no devuelve ningún valor (`void`), el nombre del subprograma y los parámetros formales.
  - El cuerpo de un procedimiento no debe tener ninguna sentencia `return`.
  - Procesa información que se transfiere a través de los parámetros.

```
// Definición de Función
// Parámetros formales
int menor (int x, int y)
{
    int z = x; // Vble Local
    if (y < z) {
        z = y;
    }
    return z; // Única,Última
}
```

```
// Def. de Procedimiento
// Parámetros formales
void ordenar (int& x, int& y)
{
    if (x > y) {
        int z = x; // V. Local
        x = y;
        y = z;
    }
}
```

```
// - Principal ----
int main ()
{
    int a, b; // Vbles Locales
    cin >> a >> b;
    int c = menor(a, b); // Invocación
    ordenar(a, b); // Par. actuales
    bool ok = (c == a);
}
```

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

- PARÁMETROS (ARGUMENTOS)
  - Todo el intercambio/transferencia de información con un subprograma se realiza a través de los parámetros (argumentos)
  - El flujo de información posee una determinada direccionalidad: (⇓) Entrada (⇑) Salida (⇕) Entrada/Salida
  - Los parámetros poseen:
    - TIPO que define el intercambio de información
    - NOMBRE con el que referenciarlos
  - **Parám. Formales:** aparecen en la definición del subprograma
  - **Parám. Actuales:** aparecen en la invocación al subprograma

PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA

- Es la forma práctica en que se implementa la transferencia de información del interfaz.
- *Parámetros de Entrada de Tipos Simples:* en el **paso por valor** (`int x`) se almacena en el parámetro formal una **copia del valor** del parámetro actual, manteniendose ambos como objetos distintos.
- *Parámetros de Salida y Entrada/Salida:* en el **paso por referencia** (`int& x`) el parámetro formal se **vincula a la variable** situada como parámetro actual, referenciando ambos al mismo objeto.
- *Parámetros de Entrada de Tipos Compuestos:* en el **paso por referencia constante** (`const Persona& x`) el parámetro formal constante se **vincula al valor** del parámetro actual, referenciandolo de forma constante y permitiendo acceder a dicho valor.

	Tipos	
	Simples	Compuestos
(⇓) Entrada	P.Valor (`int x`)	P.Ref.Cte (`const Persona& x`)
(⇑) Salida, (⇕) E/S	P.Ref (`int& x`)	P.Ref (`Persona& x`)

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

UTILIZACIÓN DE SUBPROGRAMAS: INVOCACIÓN

- La invocación (llamada) se realiza mediante el nombre seguido por los parámetros actuales.
  - La invocación a una **función** no puede constituir por sí sola una sentencia, sino que debe aparecer dentro de alguna estructura que **utilice** el valor resultado de la función.
  - La invocación a un **procedimiento** constituye por sí sola una sentencia que puede ser utilizada como tal en el cuerpo de subprogramas y del programa principal.
  - Un subprograma puede invocar a otros subprogramas (definidos previamente)

```
// Definición de Función
// Parámetros formales
int menor (int x, int y)
{
    int z = x; // Vble Local
    if (y < z) {
        z = y;
    }
    return z; // Única,Última
}
```

```
// Def. de Procedimiento
// Parámetros formales
void ordenar (int& x, int& y)
{
    if (x > y) {
        int z = x; // V. Local
        x = y;
        y = z;
    }
}
```

```
// - Principal ----
int main ()
{
    int a, b; // Vbles Locales
    cin >> a >> b;
    int c = menor(a, b); // Invocación
    ordenar(a, b); // Par. actuales
    bool ok = (c == a);
}
```

UTILIZACIÓN DE SUBPROGRAMAS. PARÁMETROS

- El número de parámetros actuales debe coincidir con el número de parámetros formales.
- Correspondencia posicional entre parámetros actuales y formales.
- El tipo del parámetro actual debe coincidir con el tipo del correspondiente parámetro formal.
- Un parámetro formal de **salida** o **entrada/salida** *(paso por referencia)* requiere que el parámetro actual sea una **variable**.
- Un parámetro formal de **entrada** *(paso por valor o referencia constante)* requiere que el parámetro actual sea una **variable**, **constante** o **expresión**.

	Tipos Simples		Tipos Compuestos	
Parámetro Formal	(⇓) Ent P.Valor (`int x`)	(⇑) Sal (⇕) E/S P.Referencia (`int& x`)	(⇓) Ent P.Ref.Constante (`const Persona& x`)	(⇑) Sal (⇕) E/S P.Referencia (`Persona& x`)
Parámetro Actual	Constante Variable Expresión	Variable	Constante Variable Expresión	Variable

UTILIZACIÓN DE SUBPROGRAMAS

- Cuando se produce una invocación a un subprograma:

  1. Se establecen las vias de comunicación entre los algoritmos llamante y llamado.
     - **Copia de valores** en el *paso por valor*.
     - **Vinculación de variables** en el *paso por referencia*.
  2. El flujo de control pasa a ejecutar la primera instrucción del cuerpo del subprograma llamado, ejecutándose éste.
  3. Las variable locales se crearán a medida que se *ejecuten* sus definiciones.
  4. Cuando finaliza la ejecución del subprograma, los parámetros y variables locales previamente creadas se destruyen y el flujo de control continúa por la siguiente instrucción a la invocación realizada.

```
int menor (int x, int y)
{
   int z = x;
   if (y < z) {
      z = y;
   }
   return z;
}
```

```
void ordenar (int& x, int& y)
{
   if (x > y) {
      int z = x;
      x = y;
      y = z;
   }
}
```

```
// - Principal ----
int main ()
{
   int a, b;
   cin >> a >> b;
   int c = menor(a, b);
   ordenar(a, b);
   bool ok = (c == a);
}
```

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const double PI = 3.1416;
const int PI_GRAD = 180;
const int MIN_GRAD = 60;
const int SEG_MIN = 60;
const int SEG_GRAD = SEG_MIN * MIN_GRAD;
// - Subalgoritmos --
void leer_grados (int& grad, int& min, int& seg)
{
   cout << "Grados, minutos y segundos ";
   cin >> grad >> min >> seg;
}
//---------------------------
void escribir_radianes (double rad)
{
   cout << "Radianes: " << rad << endl;
}
//---------------------------
double calc_rad (double grad_tot)
{
   return (grad_tot * PI) / double(PI_GRAD);
}
```

```
// - Subalgoritmos --
double calc_gr_tot (int grad, int min, int seg)
{
   return double(grad)
          + (double(min) / double(MIN_GRAD))
          + (double(seg) / double(SEG_GRAD));
}
//---------------------------
double trans_gr_rad (int grad, int min, int seg)
{
   double gr_tot = calc_gr_tot(grad, min, seg);
   return calc_rad(gr_tot);
}
// - Principal ----
int main ()
{
   int grad, min, seg;
   leer_grados(grad, min, seg);
   double rad = trans_gr_rad(grad, min, seg);
   escribir_radianes(rad);
}
```

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

SOBRECARGA DE SUBPROGRAMAS Y OPERADORES

- Se denomina sobrecarga cuando distintos subprogramas se denominan con el mismo identificador u operador
- En C++ es posible sobrecargar subprogramas y operadores siempre y cuando tengan parámetros diferentes
- Con propósitos de eficiencia, cuando un subprograma se reduce a una simple expresión, es posible definirlo `inline`

```
inline double media (int x, int y, int z)
{
   return double(x + y + z) / 3.0;
}
inline double media (int x, int y)
{
   return double(x + y) / 2.0;
}
```

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

PRECONDICIONES Y POSTCONDICIONES

- **Precondición** es un enunciado que debe ser cierto **antes** de la invocación a un subprograma. Especifica las condiciones bajo las cuales se ejecutará dicho subprograma.

- **Postcondición** es un enunciado que debe ser cierto **tras** la ejecución de un subprograma. Especifica el comportamiento de dicho subprograma.

- Codificar las pre/post-condiciones mediante **asertos** proporciona una valiosa documentación, y tiene varias ventajas:
  - Hace al programador explícitamente consciente de los prerequisitos y del objetivo del subprograma.
  - Durante la **depuración**, las precondiciones comprueban que la invocación al subprograma se realiza bajo condiciones válidas.
  - Durante la **depuración**, las postcondiciones comprueban que el comportamiento del subprograma es adecuado.
  - *A veces no es posible codificarlas adecuadamente.*
  - Ayudan en la detección de errores de programación.

PRECONDICIONES Y POSTCONDICIONES. EJEMPLO

```
#include <iostream>
#include <string>
#include <cassert>
using namespace std;
//---------------------------
// Desactivar asertos: g++ -DNDEBUG -ansi -Wall -Wextra -Werror -o programa programa.cpp
//---------------------------
bool es_par (int num)
{
   return num % 2 == 0;
}
//---------------------------
void dividir_par (int dividendo, int divisor, int& cociente, int& resto)
{
   assert(es_par(dividendo) && (divisor > 0)); // PRECOND
   cociente = dividendo / divisor;
   resto = dividendo % divisor;
   assert(dividendo == (divisor * cociente + resto)); // POSTCOND
}
```

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

- CRITERIOS DE MODULARIZACIÓN

  - No existen métodos objetivos para determinar cómo descomponer un problema en subprogramas, es una labor subjetiva.

  - No obstante, se siguen algunos criterios que pueden guiarnos para descomponer un problema y modularizar adecuadamente.

    - **Acoplamiento**

    - **Cohesión**

- ACOPLAMIENTO

  - Un objetivo en el diseño descendente es crear subprogramas aislados e independientes.

  - Debe haber alguna conexión entre los subprogramas para formar un sistema coherente.

  - Dicha conexión se conoce como acoplamiento.

  - Maximizar la independencia será **minimizar el acoplamiento**.

- COHESIÓN
  - Hace referencia al grado de relación entre las diferentes partes internas a un subprograma.
  - Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro de un subprograma es tal que posteriores modificaciones podrían resultar complicadas.
  - Se busca **maximizar la cohesión** dentro de cada subprograma.

- CRITERIOS DE MODULARIZACIÓN

  - El diseñador de software debe buscar un **bajo acoplamiento** entre subprogramas y una **alta cohesión** dentro de cada uno.

  - Si **no** es posible **analizar** y comprender un subprograma de forma aislada e **independiente** del resto, entonces podemos deducir que la división modular **no es la más adecuada**

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

REGLAS DE ÁMBITO

1. Un identificador es **visible** y accesible en la zona comprendida entre el lugar en que se declara y el final del bloque o cuerpo del subprograma donde ha sido declarado. A esta zona se le denomina **ámbito de visibilidad** del identificador.

2. En el caso de una variable, el ámbito de visibilidad coincide con su **Tiempo de Vida** durante la ejecución del programa. La variable se **crea** cuando la ejecución entra en el ámbito de visibilidad, y se **destruye** cuando la ejecución sale del mismo.

3. Si dentro del ámbito de un determinado identificador, éste mismo aparece declarado en un nivel de anidamiento (bloque) más interno, entonces dentro del ámbito de esta segunda declaración, la declaración más externa quedará **ocultada** (y por lo tanto no accesible).

4. Si la variable de control del bucle **for** se declara en la zona de inicialización del mismo, entonces su ámbito de visibilidad se extiende hasta el final del cuerpo del bucle.

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const int  MAX  = 30;
// - Subalgoritmos --
void  Sub1  (int  m  , int&  i )
{
   int  x ,  z ;
   x = m  +  MAX;
   for (int  i  = 0; i < MAX; ++i) {
      int  z ;
      z = x + m;
   }
}
void  Sub2  (int  z )
{
   int  x ;
   Sub1(3, x);
   // ...
}
// - Principal ----
int main ()
{
   int  i ,  j ;
   // ...
}
```

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const int MAX = 30;
// - Subalgoritmos --
void Sub1 (int m , int& i )
{
   int x, z;
   x = m  +  MAX;
   for (int i = 0; i < MAX; ++i) {
      int z;
      z = x + m;
   }
}
void Sub2 (int z )
{
   int x;
   Sub1(3, x);
   // ...
}
// - Principal ----
int main ()
{
   int i, j;
   // ...
}
```

DECLARACIÓN DE SUBPROGRAMAS: PROTOTIPOS

- **Prototipo**: es posible declarar un determinado subprograma sin necesidad de definirlo explícitamente.
  - Define la cabecera del subprograma, sin necesidad de definir su cuerpo (*punto y coma*).
  - Permite invocar a un determinado subprograma desde cualquier punto dentro del ámbito de visibilidad de dicho prototipo, sin necesidad de que el subprograma haya sido definido previamente.
  - El subprograma deberá ser definido en algún otro lugar del programa.

```
// - Prototipos --
void Sub2 (int z); // Prototipo de Sub2
// - Subalgoritmos --
void Sub1 (int m , int& i)
{
   Sub2(m + 3); // Invocación a Sub2
}
// Definición de Sub2
void Sub2 (int z)
{
   // ...
}
```

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

DECLARACIONES LOCALES Y GLOBALES

- **Variables locales** son aquellas cuyo ámbito de visibilidad se restringe **exclusivamente** al cuerpo del algoritmo en el que han sido declaradas.

- **Variables globales** son aquellas cuyo ámbito de visibilidad se extiende al cuerpo de varios algoritmos.

- Se denonima **efecto lateral** al intercambio de información entre dos algoritmos realizado a través de variables globales (es decir, sin utilizar el interfaz).

- En nuestra metodología de programación, la utilización de **variables globales** y **efectos laterales** esta **prohibida**.

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const int MAX = 30;
// VARIABLES GLOBALES (PROHIBIDAS EN LA ASIGNATURA)
int vble_global;
// - Subalgoritmos --
void Sub1 ()
{
   int vble_local;
   vble_local = vble_global * MAX; // EFECTO LATERAL (PROHIBIDO EN LA ASIGNATURA)
   vble_global = vble_local + MAX; // EFECTO LATERAL (PROHIBIDO EN LA ASIGNATURA)
}
// - Principal ----
int main ()
{
   int i,j;
   vble_global = 5;
   Sub1();
   cout << vble_global << endl;
}
```

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

Depuración. Corrección de Errores

- Detección de errores
  - Asertos. Enunciados que deben ser ciertos.
  - La prueba (ejecución) correcta **no** garantiza la *"ausencia de errores"*.
  - La prueba (ejecución) errónea **sí** garantiza que los errores existen.
    - Pruebas de casos límite y normal.
    - Pruebas de cada camino de ejecución.
  - Verificación formal.
- Búsqueda de errores
  - Los asertos indican el número de línea donde se detecta el error.
  - Los mensajes de depuración ayudan a encontrar donde se produce el error.
  - Es necesario un análisis del código y su comprensión, así como una explicación lógica del error según el código y el comportamiento del programa.
- Corrección de errores
  - Corrección sólo cuando se comprende el código y la naturaleza del error.
  - Los cambios aleatorios no corrigen el error, sólo introducen nuevos errores.

Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]

- ÁREAS DE MEMORIA

  - Áreas de Memoria durante la ejecución de un programa:

    - **Memoria global** (*estática*): almacena constantes y datos globales, con un tiempo de vida que coincide con el tiempo de ejecución del programa.

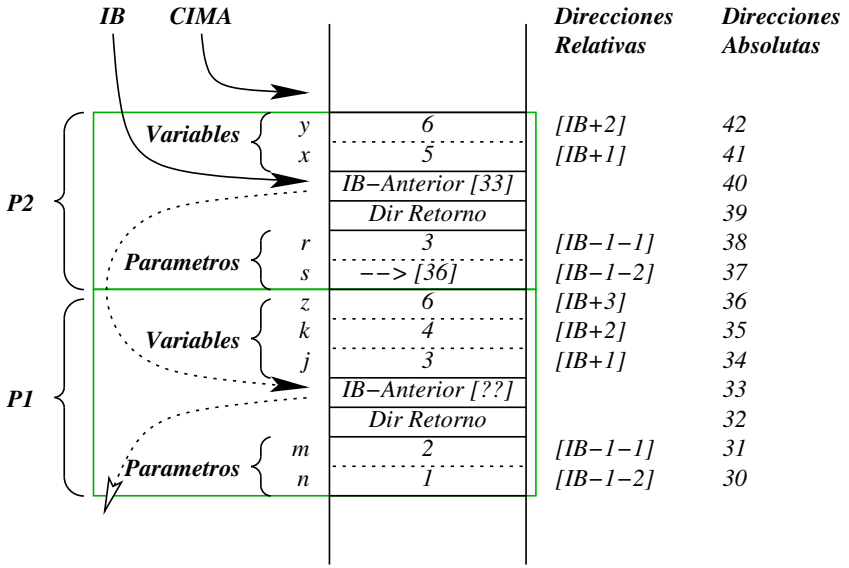
    - **Memoria automática** (*stack–pila de ejecución*): almacena los parámetros y variables locales que se crean y destruyen durante la invocación a subprogramas. Gestionada automáticamente por el compilador y el flujo de ejecución del programa.

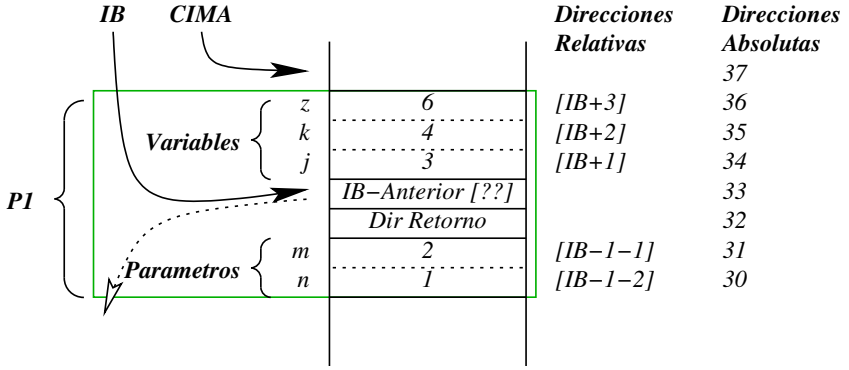
    - **Memoria dinámica** (*heap–montículo*): almacena datos cuyo tiempo de vida está gestionado dinámicamente por el programador, y cuyo acceso se realiza a través de punteros.

MEMORIA AUTOMÁTICA: La Pila de Ejecución

EJEMPLO:

```
void P2 (int r, int& s)
{
   int x, y;
   x = 5;
   y = 6;
}
void P1 (int m, int n)
{
   int j, k, z;
   j = 3;
   k = 4;
   z = 6;
   P2(j, z);
}
```





Tema 3: DISEÑO DESCENDENTE: SUBPROGRAMAS

1. Diseño Descendente. Abstracción. Interfaz
2. Subprogramas. Procedimientos y Funciones
3. Paso de Parámetros
4. Invocación a Subprogramas
5. Sobrecarga
6. Precondiciones y Postcondiciones
7. Criterios de Modularización
8. Ámbitos de Visibilidad
9. Declaraciones locales y globales. Efectos laterales
10. Depuración. Corrección de Errores
11. Áreas de Memoria
12. Bibliografía: [DALE89a], [JOYA03]