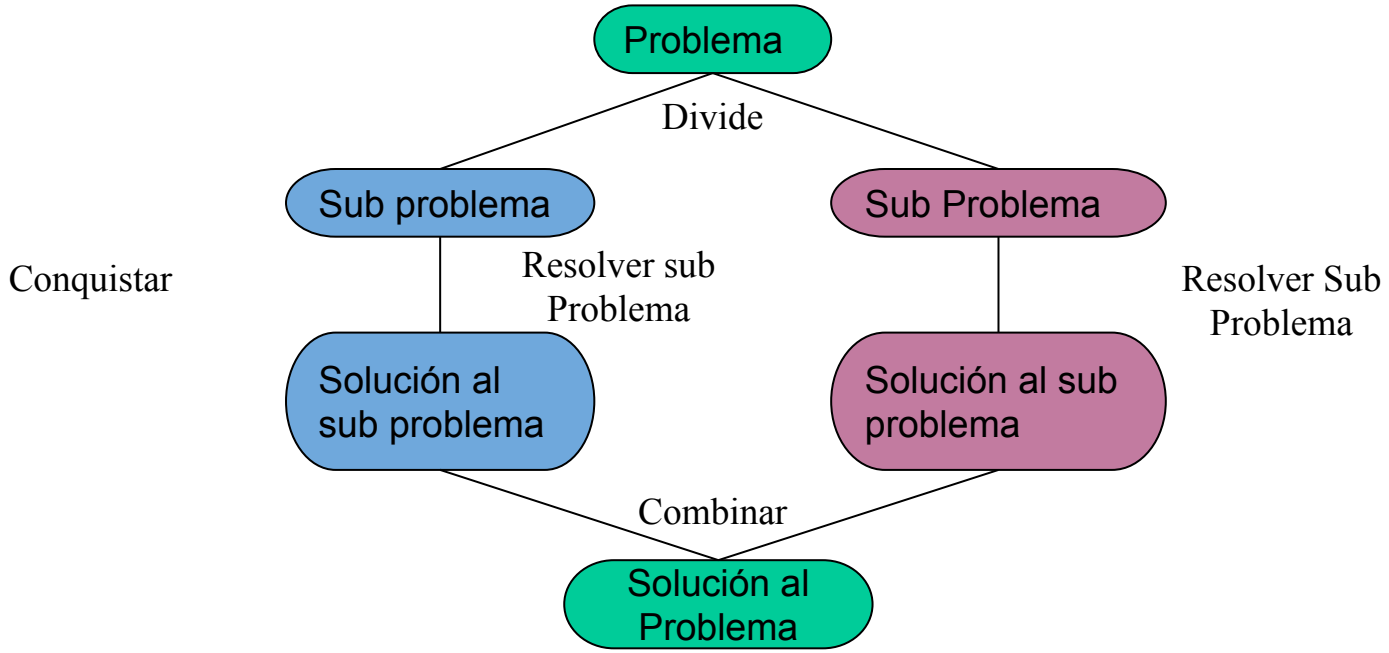
# Cap. 5 - Funciones





# 5.1 Introducción

- Divide y Conquista
  - Construir un programa a partir de piezas o componentes más pequeños
  - Cada pieza es más manejable que el programa original





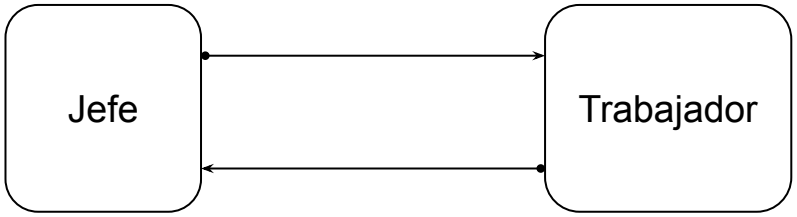

# 5.2 Módulos de Programas en C

- Functiones
  - Modulos en C
  - Programas escritos combinando las funciones definidas por el usuario con las funciones de la biblioteca
    - Librería estándar de C tiene una amplia variedad de funciones
    - Facilita el trabajo del programador, evita reinventar la rueda.



# 5.2 Módulos de Programas en C (II)

- Llamadas a Funciones
  - Invocando funciones
    - Proporcionar el nombre de la función y los argumentos (datos)
    - La función realiza operaciones o manipulaciones
    - La función devuelve los resultados
  - El jefe pide al trabajador que complete la tarea
    - El trabajador obtiene información, hace la tarea, devuelve el resultado
    - Ocultación de información: el jefe no conoce los detalles






# 5.3 Librería de Funciones de Matemática

- Librería de funciones de Matemática
  - realizar cálculos matemáticos comunes
  - `#include <math.h>`

- Formato de las funciones de llamada

  `FunctionName (argument);`
    - Si hay múltiples argumentos, use la lista separada por comas
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Llama a la función `sqrt`, el cual retorna la raíz cuadrada de su argumento
    - Todas las funciones matemáticas retornan tipo de dato **`double`**
  - Argumentos pueden ser constantes, variables, o expresiones



# 5.4 Funciones

- Funciones
  - Modularizar un programa
  - Todas las variables declaradas dentro de las funciones son variables locales
    - Conocido sólo en la función definida
  - Parámetros
    - Comunicar la información entre las funciones
    - Variables locales
- Beneficios
  - Divide y Conquista
    - Desarrollo de programas manejables
  - Reusabilidad de Software
    - Usar las funciones existentes como bloques de construcción para nuevos programas
    - Abstracción - ocultar detalles internos (funciones de la biblioteca)
  - Evita repeticiones de códigos.



# 5.5 Definición de Funciones

- Formato de definición de la función

```
tipo-valor-retornado  nombre-función ( lista-parametros )
   {
        declaraciones y manifestaciones
   }
```

  - Nombre-función: cualquier identificador válido
  - Typo-valor-retornado: tipo de dato del resultado (default **`int`**)
    - **`void`** - la función retorna nada
  - Lista-parametros: lista separada por coma, parametros declarados (default **`int`**)



# 5.5 Definición de Funciones (II)

- Formato de definición de la función (continue)

  *tipo-valor-retornado nombre-función* **(** *lista-parametros* **)**

  **{**

  *declaraciones y manifestaciones*

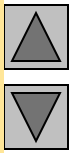
  **}**

  – Declaraciones y enunciados: cuerpo de la función (bloque)
    - Las variables pueden ser declaradas dentro de los bloques (pueden ser anidadas)
    - La función no puede ser definida dentro de otra función
  – Devolver el control
    - Si no se devuelve nada
      - **`return;`**
      - o, hasta llegar a la llave derecha
    - Si algo regresa
      - **`return`** *expresión* **`;`**



```c
/* Fig. 5.4: fig05_04.c
   Hallar el máximo de tres enteros */
#include <stdio.h>

int maximum( int, int, int );   /* pototipado de funciones */

int main()
{
   int a, b, c;

   printf( "Ingrese tres enterios: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximo es: %d\n", maximum( a, b, c ) );

   return 0;
}

/* definición de la función maximum */
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

```
Ingrese tres enteros: 22 85 17
Maximo es: 85
```



Outline

1. Prototipado de Funciones (3 parametros)

2. Valores de Entrada

2.1 Llamada función

Definición de función

Program Output

# 5.6 Prototipos de funciones

- Prototipos de funciones
  - Nombre de la Función
  - Parámetros - lo que la función toma
  - Tipo Retornado - la función de tipo de datos regresa (default **`int`**)
  - Utilizado para validar funciones
  - El prototipo sólo se necesita si la definición de la función viene después del uso en el programa

```
int maximum( int, int, int );
```

    - Toma en 3 **`int`**s
    - Retorna un **`int`**
- Normas de promoción y conversiones
  - La conversión a tipos inferiores puede llevar a errores



# 5.7 Archivos de Cabecera

- Archivos de Cabecera
  - contienen prototipos de funciones para las funciones de la biblioteca
  - **`<stdlib.h>`**, **`<math.h>`**, etc
  - Se cargan con **`#include <filename>`**

    **`#include <math.h>`**

- Archivos de cabecera personalizados
  - Crear un archivo con funciones
  - Guarda como **`filename.h`**
  - Carga en otros archivos con **`#include "filename.h"`**
  - Funciones reusadas



# 5.8 Llamando a Funciones: Llamar por valor y llamar por referencia

- Usado cuando invocamos funciones
- Llamada por valor
  - Copia del argumento pasado a la función
  - Los cambios en la función no afectan al original
  - Se utiliza cuando la función no necesita modificar el argumento
    - Evita los cambios accidentales
- Llamada por referencia
  - Pasa el argumento original
  - Cambios en la función afecta al original
  - Solo usado con funciones verdaderas
- Por ahora, nos centramos solo en llamadas por valor



# 5.9 Generación de Números Aleatorios

- Función **rand**
  - Cargada por **<stdlib.h>**
  - Retorna número "aleatorio" entre **0** y **RAND_MAX** (hasta 32767)

    ```
    i = rand();
    ```

  - Pseudorandom
    - Secuencia preestablecida de números "aleatorios"
    - La misma secuencia para cada llamada de función
- Escalado
  - Para obtener un número aleatorio entre `1` y `n`

    ```
    1 + ( rand() % n )
    ```

    - **rand % n** retorna a número entre **0** y **n-1**
    - Suma **1** para hacer un número aleatorio entre **1** y **n**

      ```
      1 + ( rand() % 6)   // number between 1 and 6
      ```



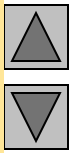
# 5.9 Generación de Números Aleatorios (II)

- Función **`srand`**
  - **`<stdlib.h>`**
  - Toma una semilla entero- salta al lugar en una secuencia "aleatoria"

    ```
    srand( seed );
    ```
  - **`srand( time( NULL ) ); //load <time.h>`**
    - **`time( NULL )`** – el programa de tiempo fue compilado en segundos
    - "aleatoriza" la semilla



```
1   /* Fig. 5.9: fig05_09.c
2      programa que lanza el dado */
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   int main()
7   {
8      int i;
9      unsigned seed;
10
11     printf( "Ingrese Semilla: " );
12     scanf( "%u", &seed );
13     srand( seed );
14
15     for ( i = 1; i <= 10; i++ ) {
16        printf( "%10d", 1 + ( rand() % 6 ) );
17
18        if ( i % 5 == 0 )
19           printf( "\n" );
20     }
21
22     return 0;
23  }
```



## Outline

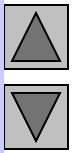
1. Inicializa semilla

2. Valor de entrada para semilla

2.1 Usa `srand` para cambiar la secuencia aleatoria

2.2 Define Bucle

3. Genera y retorna un número aleatorio



Outline

Program Output

```
Ingrese Semilla: 67
         6          1          4          6          2
         1          6          1          6          4
```

```
Ingrese Semilla: 867
         2          4          6          1          6
         1          1          3          6          2
```

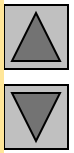
```
Ingrese Semilla: 67
         6          1          4          6          2
         1          6          1          6          4
```

# 5.10 Ejemplo: Un juego de azar

- Simulador de dados
- Reglas
  - Lanzar dos dados
    - 7 o 11 en el primer lanzamiento, jugador gana
    - 2, 3, o 12 en el primer lanzamiento, jugador pierde
    - 4, 5, 6, 8, 9, 10 - el valor se convierte en el “punto” de jugador
  - El jugador debe lanzar su punto antes de lanzar 7 para ganar




```c
/* Fig. 5.10: fig05 10.c
   Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int rollDice( void );

int main()
{
   int gameStatus, sum, myPoint;

   srand( time( NULL ) );
   sum = rollDice();          /* primera tirada de los dados */

   switch ( sum ) {
      case 7: case 11:           /* ganar en la primera tirada
         gameStatus = 1;
         break;
      case 2: case 3: case 12:  /* perder en la primera tirada
         gameStatus = 2;
         break;
      default:                   /* recuerde el punto */
         gameStatus = 0;
         myPoint = sum;
         printf( "Punto es %d\n", myPoint );
         break;
   }

   while ( gameStatus == 0 ) {    /* Sigue tirando */
      sum = rollDice();

```



## Outline

1. `rollDice` prototype

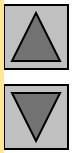
1.1 Initialize variables

1.2 Seed `srand`

2. Define `switch` statement for win/loss/continue

2.1 Loop

```
33          if ( sum == myPoint )       /* ganan al hacer el punto*/
34             gameStatus = 1;
35          else
36             if ( sum == 7 )          /* perder por rodar 7 */
37                gameStatus = 2;
38       }
39
40       if ( gameStatus == 1 )
41          printf( "Jugador gana\n" );
42       else
43          printf( "Jugador pierde\n" );
44
45       return 0;
46    }
47
48    int rollDice( void )
49    {
50       int die1, die2, workSum;
51
52       die1 = 1 + ( rand() % 6 );
53       die2 = 1 + ( rand() % 6 );
54       workSum = die1 + die2;
55       printf( "Jugador lanzo %d + %d = %d\n", die1, die2, workSum
`
56       return workSum;
57    }
```



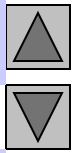
Outline

2.2 Print win/loss

Program Output

```
Jugador lanzó 6 + 5 = 11
Jugador gana
```

```
Jugador lanzó 6 + 6 = 12
Jugador pierde
```



Outline

```
Jugador lanzó 4 + 6 = 10
Punto es 10
Jugador lanzó 2 + 4 = 6
Jugador lanzó 6 + 5 = 11
Jugador lanzó 3 + 3 = 6
Jugador lanzó 6 + 4 = 10
Jugador gana
```

Program Output

```
Jugador lanzó 1 + 3 = 4
Punto e 4
Jugador lanzó 1 + 4 = 5
Jugador lanzó 5 + 4 = 9
Jugador lanzó 4 + 6 = 10
Jugador lanzó 6 + 3 = 9
Jugador lanzó 1 + 2 = 3
Jugador lanzó 5 + 2 = 7
Jugador pierde
```

# 5.11 Clases de almacenamiento

- Especificadores de clase de almacenamiento
  - Duración del almacenamiento - cuánto tiempo existe un objeto en la memoria
  - Alcance - donde el objeto puede ser referenciado en el programa
  - Vinculación - qué archivos se conoce como identificador

- Almacenamiento automático
  - El objeto creado y destruido dentro de su bloque
  - **`auto`:** por defecto para las variables locales
    - `auto double x, y;`
  - **`register`:** trata de poner variables en registros de alta velocidad
    - Sólo se puede utilizar para las variables automáticas
    - `register int counter = 1;`



# 5.11 Clases de almacenamiento (II)

- Almacenamiento estático
  - Existen variables para toda la ejecución del programa
  - Valor por defecto de cero
  - **`static`:** variables locales definidas en funciones.
    - Mantener el valor después de que la función termine
    - Sólo se conocen en su propia función
  - **`extern`:** Por defecto para las variables y funciones globales.
    - Conocido en cualquier función



# 5.12 Reglas de Alcance

- File scope
  - Identificador definido fuera de la función, conocido en todas las funciones
  - Variables globales, definiciones de funciones, prototipos de funciones

- Function scope
  - Solo puede ser referenciado dentro de un cuerpo funcional



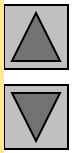
# 5.12 Reglas de Alcance (II)

- Block scope
  - Identificador declarado dentro de un bloque
    - El alcance del bloque comienza en la declaración, termina en la abrazadera derecha
  - Variables, parámetros de función (variables locales de función)
  - Bloques externos "ocultos" de bloques internos si el mismo nombre de la variable

- Prototipo de función del ámbito de aplicación
  - Identificadores en la lista de parámetros
  - Los nombres en el prototipo de la función son opcionales, y pueden ser usados en cualquier lugar



```c
/* Fig. 5.12: fig05_12.c
   Ejemplo de alcance */
#include <stdio.h>

void a( void );   /* prototipado de funciones */
void b( void );   /* prototipado de funciones */
void c( void );   /* prototipado de funciones */

int x = 1;        /* variable global */

int main()
{
   int x = 5;   /* variable local a principal */

   printf("local x en el ámbito exterior de la principal es %d\n",x );

   {            /* inicia nuevo alcance */
      int x = 7;

      printf("local x en el ámbito interno de la principal es
   }            /* termina nuevo alcance */

   printf("local x en el ámbito exterior de la principal es %d\n",

   a();         /* a tienen variable x local automático*/
   b();         /* b tiene una variable x local estatica*/
   c();         /* c use variable global x */
   a();         /* a reinicia variable x local automática*/
   b();         /* local estático x retiene su valor anterior
   c();         /* global x también retiene su valor */
```



## Outline

1. Function prototypes

1.1 Initialize global variable

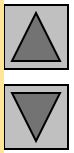
1.2 Initialize local variable

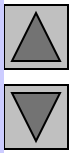
1.3 Initialize local variable in block

2. Call functions

3. Output results

```
31
32      printf( "local x in main is %d\n", x );
33      return 0;
34   }
35
36   void a( void )
37   {
38      int x = 25;  /* inicializado cada vez que se llama a */
39
40      printf( "\nlocal x en a es %d después de entrar a\n", x );
41      ++x;
42      printf( "local x en a es %d antes de existir\n", x );
43   }
44
45   void b( void )
46   {
47       static int x = 50;  /* sólo inicialización estática */
48                           /* la primera vez que b se llama*/
49       printf( "\nlocal estatico x es %d al entrar b\n", x );
50       ++x;
51       printf( "local estatico x es %d al entrar b\n", x );
52   }
53
54   void c( void )
55   {
56      printf( "\nglobal x es %d al entrar c\n", x );
57      x *= 10;
58      printf( "global x es %d al entrar c\n", x );
59   }
```





Outline

Program Output

```
local x en el ámbito exterior de la principal es 5
local x en el ámbito interno de la principal es 7
local x en el ámbito exterior de la principal es 5

local x en a es 25 después de entrar a
local x in a is 26 antes de salir a

local estatico x es 50 al entrar b
local estatico x es 51 al salir b

global x es 1 al entrar c
global x es 10 al entrar c

local x en a es 25 después de entrar a
local x en a es 26 antes de salir a

local estatico x es 51 al entrar b
local estatico x es 52 al salir b

global x es 10 al entrar c
global x es 100 al salir c
local x en main es 5
```

# 5.13 Recursividad

- Funciones Recursivas
  - Funciones que se llaman a sí mismas
  - Puede sólo resolver un caso base
  - Divide el problema en
    - Lo que puede hacer
    - Lo que no puede hacer - se parece al problema original
      - Lanza una nueva copia de sí mismo (paso de recursividad)
- Eventualmente el caso base se resuelve
  - Se conecta, se abre camino y resuelve todo el problema



# 5.13 Recursividad (II)

- Ejemplo: factorial:

  `5! = 5 * 4 * 3 * 2 * 1`

  Note que

  `5! = 5 * 4!`

  `4! = 4 * 3!` ...

  – Puede computar los factores de forma recursiva

  – Resuelve caso base (1! = 0! = 1) entonces conecta

    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

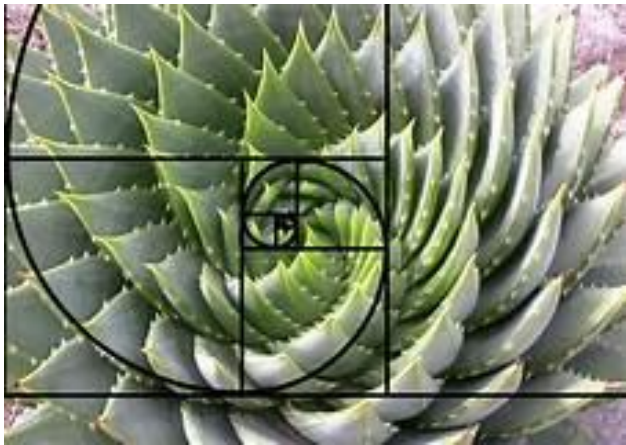


# 5.14 Ejemplo de uso de Recursivo: La serie de Fibonacci

- Serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8...
  - Cada número suma los dos anteriores

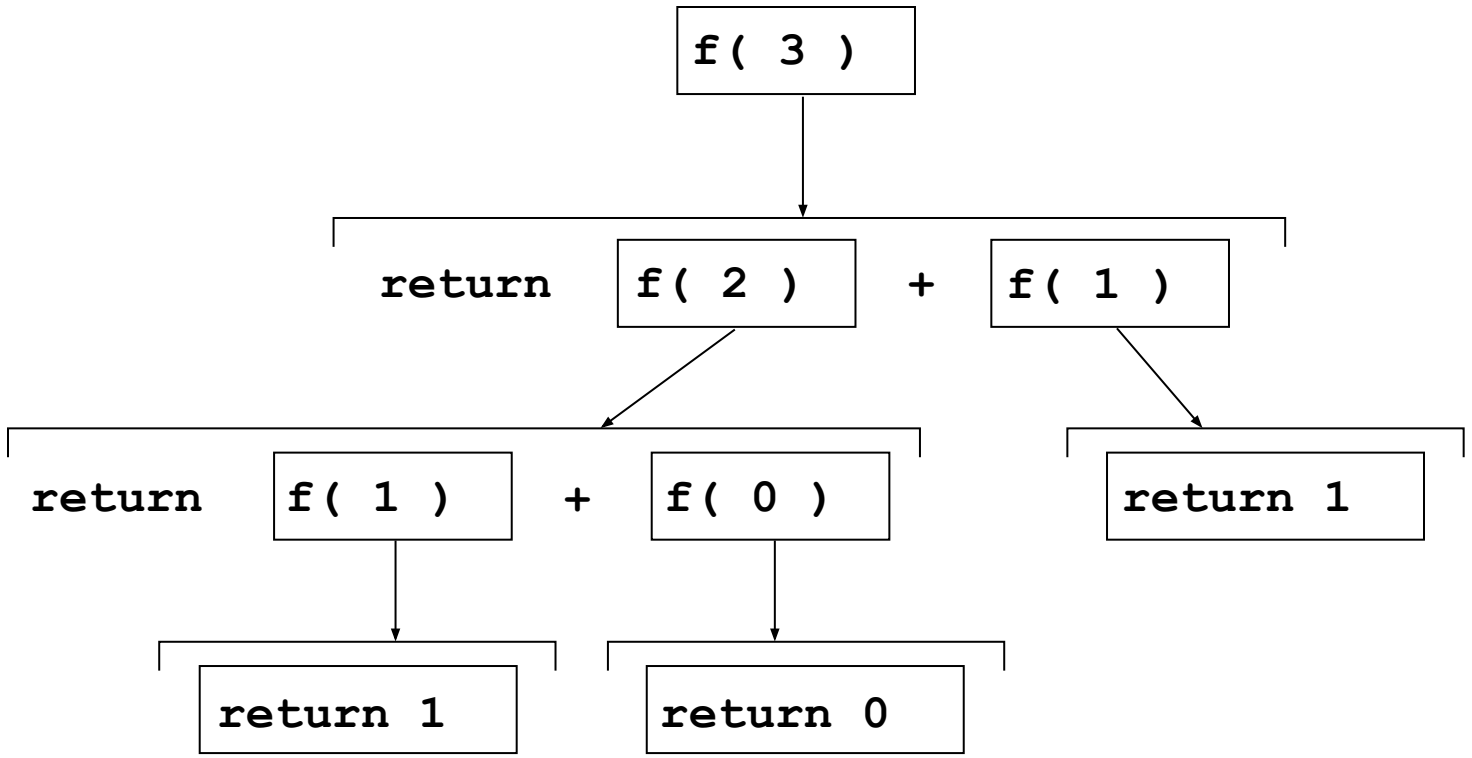
**`fib(n) = fib(n-1) + fib(n-2)`** - formula recursiva

```
long fibonacci(long n)
{
  if (n==0 || n==1)  //base case
       return n;
  else
       return fibonacci(n-1) + fibonacci(n-2);
}
```





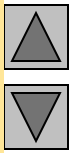
# 5.14 Ejemplo de uso de Recursivo: La serie de Fibonacci (II)






```c
/* Fig. 5.15: fig05_15.c
   Función Recursiva de Fibonacci */
#include <stdio.h>

long fibonacci( long );

int main()
{
   long result, number;

   printf( "Ingrese un entero: " );
   scanf( "%ld", &number );
   result = fibonacci( number );
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0;
}

/* Definición recursiva de la función fibonacci */
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

```
Ingrese un entero: 0
Fibonacci(0) = 0

Ingrese un entero: 1
Fibonacci(1) = 1
```



## Outline

1. Function prototype

1.1 Initialize variables

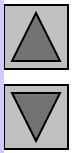
2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

Program Output



Outline

Program Output

```
Ingrese un entero: 2
Fibonacci(2) = 1

Ingrese un entero: 3
Fibonacci(3) = 2

Ingrese un entero: 4
Fibonacci(4) = 3

Ingrese un entero: 5
Fibonacci(5) = 5

Ingrese un entero: 6
Fibonacci(6) = 8

Ingrese un entero: 10
Fibonacci(10) = 55

Ingrese un entero: 20
Fibonacci(20) = 6765

Ingrese un entero: 30
Fibonacci(30) = 832040

Ingrese un entero: 35
Fibonacci(35) = 9227465
```

# 5.15 Recursividad vs. Iteración

- Repetición
  - Iteración: ciclo explicito
  - Recursión:  llamadas a función repetidamente
- Finalización
  - Iteración: falla condición de bucle
  - Recursión: reconoce caso base
- Ambos pueden tener bucles infinitos
- Balance
  - Elegir entre desempeño (iteración) y buena ingeniería de software (recursión)