Laboratorio de Programación - Labo01

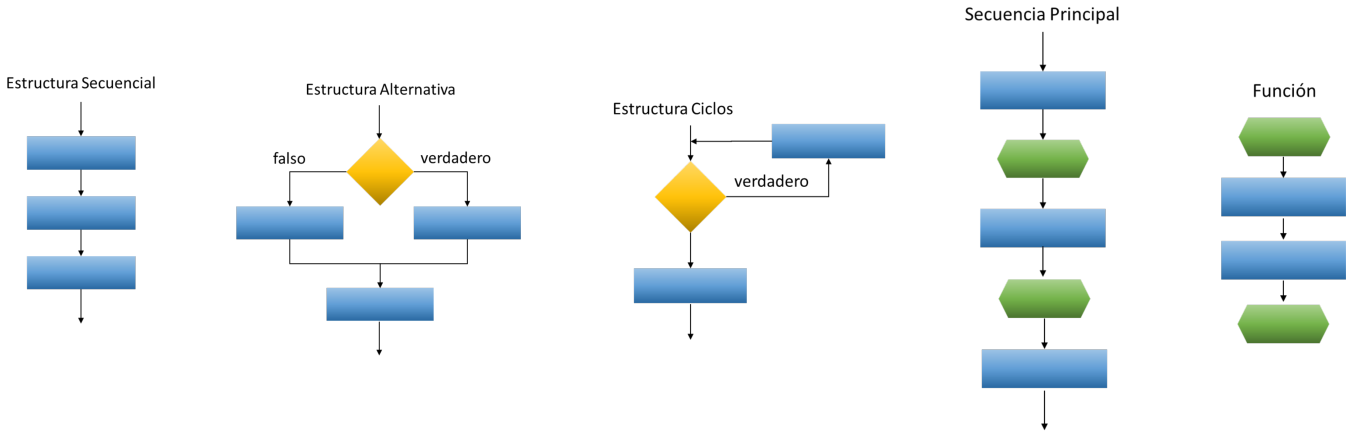
Parte 3: Estructuras de Control

Algoritmos y Estructuras de Datos I

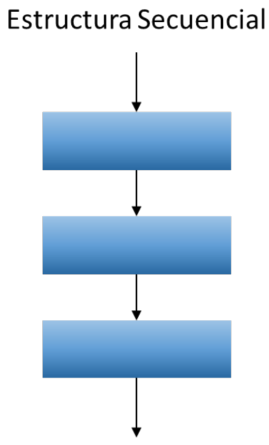
Departamento de Computación, FCEyN, Universidad de Buenos Aires.

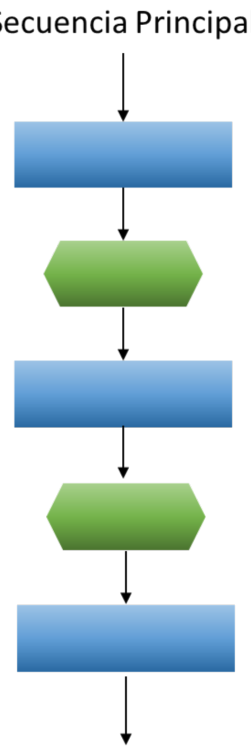
Estructuras de control

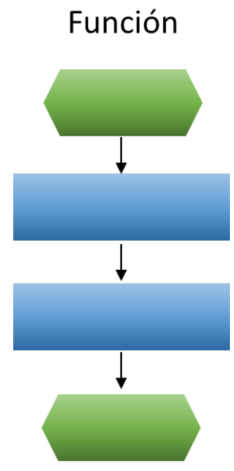
- En los lenguajes imperativos existen construcciones sintácticas que permiten modificar la secuencialidad de un programa.
- Estas construcciones estructuran el programa de forma limpia, sin necesidad, por ejemplo, de repetir la misma línea infinidad de veces para hacer una misma operación, además de permitirle tomar decisiones a partir de evaluaciones lógicas de valores de variables o estados del sistema.
- Dichas construcciones se denominan estructuras de control:
  1. Llamados a Funciones
  2. Alternativas
  3. Ciclos



Funciones







Declarando funciones en C++

- La sintáxis para declarar una función:

$$tipo \quad nombreFunction(lista \quad argumentos);$$

1. Tipo de retorno (puede retornar vacío, `void`)
2. Nombre (obligatorio)
3. Argumentos (o parámetros) (puede ser vacía)

- Los argumentos se especifican separados por comas, y cada uno debe tener un tipo de datos asociado.
- Cuando se llama a la función, el código "llamador" debe respetar el orden y tipo de los argumentos.

Funciones con valores de retorno

- Una función retorna un valor mediante la sentencia **return**.
- Por ejemplo, la siguiente función toma un parámetro entero y devuelve el *siguiente* valor:

```
1 int siguiente(int a) {
2   return a+1;
3 }
```

- Otra versión (quizás menos intuitiva):

```
1 int siguiente(int a) {
2   int b =0;
3   b = a+1;
4   return b;
5 }
```

Llamados a Funciones

- La función **siguiente()** en un programa:
-

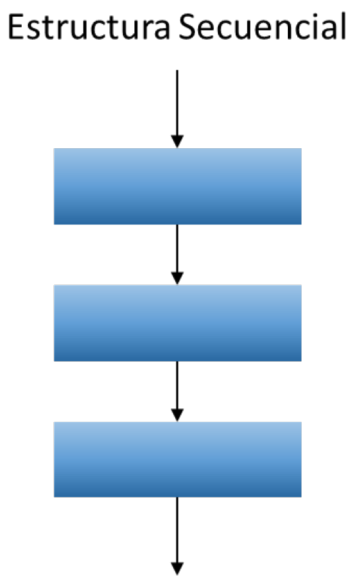
```
1  #include <iostream>
2  using namespace std; // con esto no estoy obligado a escribir std::
3
4  int siguiente(int a) {
5    return a+1;
6  }
7
8  int main() {
9    // declaracion de variables
10   int a, b;
11   // ingreso de datos
12   cout << "Ingrese un numero entero." << endl;
13   cin >> a;
14   b = 2 * siguiente(a); // llamado a siguiente
15   // salida del resultado
16   cout << "El doble del siguiente de " << a << " es " << b <<
     endl;
17   return 0;
18 }
```

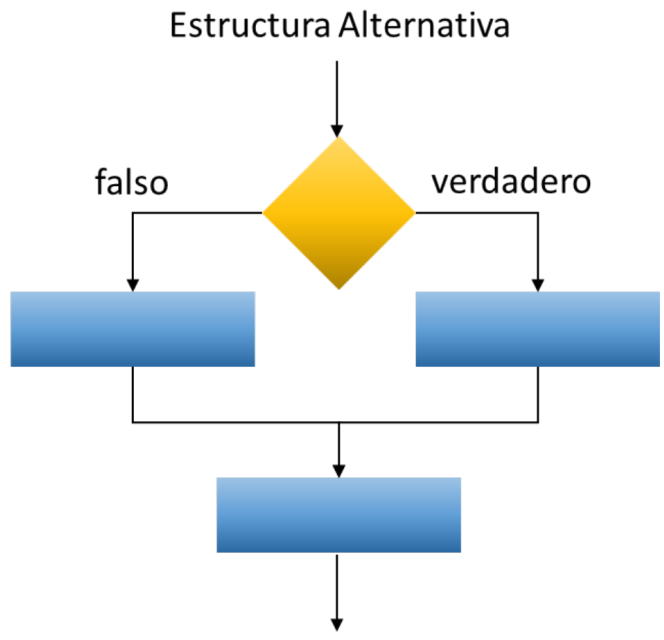
Funciones con n-argumentos

- En caso de que haya más de un parámetro, se separan por comas:

```
1  #include <iostream>
2  using namespace std;
3
4  int suma(int a, int b) {
5    return a+b;
6  }
7
8  int main() {
9    int a = suma(2,3); // llamado con 2 argumentos
10   cout << a;
11   return 0;
12 }
```

Alternativas





Instrucción alternativa

- Tiene la siguiente forma, donde B es una expresión lógica (que evalúa a **boolean**) y S_1 y S_2 son bloques de instrucciones:

```
1  if (B) {
2    S1
3  } else {
4    S2
5  }
```

- Se evalúa la guarda B. Si la evaluación da **true**, se ejecuta S_1. Si no, se ejecuta S_2.
- La rama positiva S_1 es obligatoria. La rama negativa S_2 es optativa.
- Si S_1 o S_2 constan de más de una instrucción, es obligatorio que estén rodeados por llaves.

Instrucción alternativa

- ¿Cómo podemos escribir un programa que calcule el valor absoluto de x?
-

```
1  int valorAbsoluto(int n) {
2    int res = 0;
3    if( n > 0 ) {
4      res = n;
5    } else {
6      res = -n;
7    }
8    return res;
9  }
```

Instrucción alternativa

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

int valorAbsoluto(int n) {
  int res = 0;
  if( n > 0 ) {
    res = n;
  } else {
    res = -n;
  }
  return res;
}

int main() {
  int a = 0; // declaracion de variables
  cout << "Ingrese valor entero: " << endl;
  cin >> a;
  int abs_a = valorAbsoluto(a);
  // impresion del resultado
  cout << "Valor absoluto de " << a << " es: " << abs_a << endl;
  return 0;
}
```

Instrucción alternativa

- Podemos también hacer directamente "**return** res" dentro de las ramas de la alternativa.

```
1  int valorAbsoluto(int n) {
2    if( n > 0 ) {
3      return n;
4    } else {
5      return -n;
6    }
7  }
```

- Cuidado: **return** termina inmediatamente la ejecución de la función.

  $\Rightarrow$ Puede dejar las variables en un estado inconsistente!

Instrucción alternativa

- No hay ningún problema en anidar instrucciones alternativas.
- Retomando el ejemplo del inversoMayor de la 2da. parte:
-

```
1  bool inversoMayor(int n, int m) {
2    bool res;
3    if( n != 0 ) {
4      if ( 1/n > m ) {
5        res = true;
6      } else {
7        res = false;
8      }
9    } else {
10     std::cout << "No puedo calcular inverso" << std::endl;
11     res = false;
12   }
13   return res;
14 }
```

- Esta vez, si $n = 0$, la función devuelve false pero avisamos al operador del problema.

Instrucción alternativa

- La Instrucción alternativa tambien permite obviar el `else` si no se desea ejecutar ninguna instrucción:
- Por ejemplo:

```
1  ...
2  if( x>0 ) {
3    result = true;
4  } else {
5    // no hacer nada
6  }
7  ...
```

- Es equivalente a:

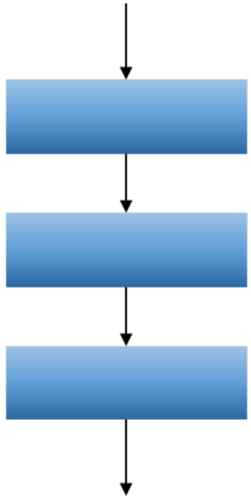
```
1  ...
2  if( x>0 ) {
3    result = true;
4  }
5  ...
```

Programación imperativa

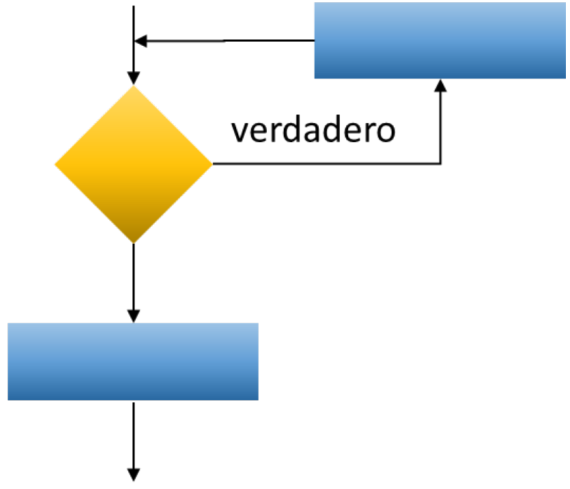
Iteración

(repetición)

Estructura Secuencial



Estructura Ciclos



Ciclos “while”

- Sintaxis:

```
while (B) {
      cuerpo del ciclo
}
```

- Se repite el cuerpo del ciclo mientras la guarda `B` se cumpla, cero o más veces. Cada repetición se llama una iteración.
- La ejecución del ciclo termina si no se cumple la guarda al comienzo de su ejecución o bien luego de ejecutar una iteración.
- Si/cuando el ciclo termina, el estado resultante es el estado posterior a la última instrucción del cuerpo del ciclo.
- Si el ciclo no termina, la ejecución nunca termina (se cuelga).

Ejemplo

```
1  int suma(int n) {
2    int i = 1;
3    int sum = 0;
4    while( i <= n ) {
5      sum = sum + i;
6      i = i + 1;
7    }
8    return sum;
9  }
```

Demo #7: instrucción while

```
1  #include <iostream>
2  using namespace std;
3
4  int suma(int n) {
5      int i = 1;
6      int sum = 0;
7      while( i <= n ) {
8          sum = sum + i;
9          i = i + 1;
10     }
11     return sum;
12 }
13
14 int main() {
15     int valorParaSumar = 0;
16     cout << "Ingrese valor para sumar: ";
17     cin >> valorParaSumar;
18     int resultadoDeSuma = suma(valorParaSumar);
19     cout << resultadoDeSuma;
20     return 0;
21 }
```

Ejemplo

- Estados al finalizar cada iteración del ciclo, para $n = 6$:

Iteración	i	suma
0	1	0
1	2	1
2	3	3
3	4	6
4	5	10
5	6	15

- Al final de las iteraciones (cuando se sale del ciclo porque no se cumple la guarda), la variable **sum** contiene el valor buscado.

Ejemplo

- La variable **i** se denomina la variable de control del ciclo.
  1. Cuenta cuántas iteraciones se han realizado (en general, una variable de control marca el avance del ciclo).
  2. En función de esta variable se determina si el ciclo debe detenerse (en la guarda).
  3. Todo ciclo tiene una o más variables de control, que se deben modificar a lo largo de las iteraciones.
- La variable **sum** se denomina el acumulador (o variable de acumulación) del ciclo.
  1. En esta variable se va calculando el resultado del ciclo. A lo largo de las iteraciones, se tienen resultados parciales en esta variable.
  2. No todo ciclo tiene un acumulador. En algunos casos, se puede obtener el resultado del ciclo a partir de la variable de control.

Ciclos “for”

- La siguiente estructura es habitual en los ciclos:
  1. Inicializar la variable de control.
  2. Chequear en la guarda una condición sencilla sobre las variables del ciclo.
  3. Ejecutar alguna acción (cuerpo del ciclo).
  4. Modificar en forma sencilla la variable de control.

- Para estos casos, tenemos la siguiente versión compacta de los ciclos, llamados ciclos “for”.

- ```
  int sum = 0;
  for(int i=1; i<=n; i=i+1) {
    sum = sum + i;
  }
  ```
  ```

Otro ejemplo usando “for”

Podemos escribir un programa que indique si un numero $n \geq 2$ es primo usando ciclos y la instrucción `for`:

```
1  bool esPrimo(int n) {
2      int divisores = 0;
3      for(int i=2; i<n; i=i+1) {
4          if( n % i == 0 ) {
5              divisores = divisores + 1;
6          } else {
7              // no hacer nada
8          }
9      }
10     if (divisores == 0) {
11         return true;
12     } else {
13         return false;
14     }
15 }
```

Demo #8: Ejemplo de uso de for

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

bool esPrimo(int n) {
    int divisores = 0;
    for(int i=2; i<n; i=i+1) {
        if( n % i == 0 ) {
            divisores = divisores + 1;
        } else {
            // do nothing
        }
    }
    if (divisores == 0) {
        return true;
    } else {
        return false;
    }
}
```

```cpp
int main() {
    cout << "Ingrese un Numero:" << endl;
    int a = 0;
    cin >> a;
    bool soyPrimo = esPrimo(a);
    cout << "El Numero " << a;
    if (soyPrimo) {
        cout << " SI" << endl;
    } else {
        cout << " NO" << endl;
    }
    cout << " es Primo." << endl;
    return 0;
}
```

Recursión en C/C++

- Podemos hacer llamados recursivos en C/C++!
- Sin embargo, el modelo de cómputo es imperativo, y entonces la ejecución es distinta en este contexto.
-

```
1 int suma(int n) {
2   if( n == 0 ) {
3     return 0;
4   } else {
5     return n + suma(n-1);
6   }
7 }
```

- Se calcula primero `suma(n-1)`, y hasta que no se tiene ese valor no se puede continuar la ejecución (orden aplicativo).
- No existe en los lenguajes imperativos el orden normal de los lenguajes funcionales!

¿Por qué un nuevo paradigma?

- Si podemos hacer recursión pero no tenemos orden normal, ¿por qué existen los lenguajes imperativos?
  1. La performance de los programas implementados en lenguajes imperativos suele ser muy superior a la de los programas implementados en lenguajes funcionales (la traducción al hardware es más directa).
  2. En muchos casos, el paradigma imperativo permite expresar algoritmos de manera más natural.
- Aunque los lenguajes imperativos permiten implementar funciones recursivas, el mecanismo fundamental de cómputo no es la recursión.

Recap: C++

Hasta ahora hemos visto:

- función `main` (punto de entrada)
- Librerías (Bibliotecas): `#include <...>`
- `cout`: salida por pantalla
- Tipos de datos: int, bool, char, float
- Declaración, inicialización y asignación de variables
- Ejecución secuencial de sentencias (ordenes/instrucciones)
- Estructuras de Control: Funciones, Alternativas, Ciclos

Bibliografía

- B. Stroustrup. The C++ Programming Language.
  - Part I: Introductory Material - A Tour of C++: The Basics
- P.J. Deitel, H.M. Deitel. Como programar en C++ (6ta. ed.)
  - Capitulos 1, 2, 4, 5 y 6.