Ordenamiento Recursivo y Referencias

Programación de Estructuras de Datos y Algoritmos Fundamentales (TC1031)

M.C. Xavier Sánchez Díaz
sax@tec.mx

Outline

La idea es insertar...

Review D & Q

Como ya vimos, la **búsqueda binaria** (la que trabaja con el arreglo ordenado) hace un *árbol* que es más rápido de buscar que si nos vamos de uno en uno en las celdas de un arreglo para encontrar lo que buscamos.

La idea detrás del *sorting* recursivo es parecida. En lugar de comparar *todos contra todos los demás*, podemos comparar con menos elementos si nos basamos en el principio de la **inserción**

1) Este elemento... ¿va a la derecha o a la izquierda del que ya conozco?
2) Este elemento nuevo... ¿va a la derecha o a la izquierda de lo que ya conozco?

La idea es insertar...

Review D & Q

Como ya vimos, la **búsqueda binaria** (la que trabaja con el arreglo ordenado) hace un *árbol* que es más rápido de buscar que si nos vamos de uno en uno en las celdas de un arreglo para encontrar lo que buscamos.

La idea detrás del *sorting* recursivo es parecida. En lugar de comparar *todos contra todos los demás*, podemos comparar con menos elementos si nos basamos en el principio de la **inserción**

1) Este elemento... ¿va a la derecha o a la izquierda del que ya conozco?
2) Este elemento nuevo... ¿va a la derecha o a la izquierda de lo que ya conozco?

La idea es insertar...

Review D & Q

Como ya vimos, la **búsqueda binaria** (la que trabaja con el arreglo ordenado) hace un *árbol* que es más rápido de buscar que si nos vamos de uno en uno en las celdas de un arreglo para encontrar lo que buscamos.

La idea detrás del *sorting* recursivo es parecida. En lugar de comparar *todos contra todos los demás*, podemos comparar con menos elementos si nos basamos en el principio de la **inserción**

1) Este elemento... ¿va a la derecha o a la izquierda del que ya conozco?
2) Este elemento nuevo... ¿va a la derecha o a la izquierda de lo que ya conozco?

La idea es insertar...

Review D & Q

Como ya vimos, la **búsqueda binaria** (la que trabaja con el arreglo ordenado) hace un *árbol* que es más rápido de buscar que si nos vamos de uno en uno en las celdas de un arreglo para encontrar lo que buscamos.

La idea detrás del *sorting* recursivo es parecida. En lugar de comparar *todos contra todos los demás*, podemos comparar con menos elementos si nos basamos en el principio de la **inserción**

1) Este elemento... ¿va a la derecha o a la izquierda del que ya conozco?
2) Este elemento nuevo... ¿va a la derecha o a la izquierda de lo que ya conozco?

. . . en listas más pequeñas

Review D & Q

Ahora bien, recordemos el enfoque que vimos hace unas semanas: *Divide & Conquer* o en español **Divide y Vencerás**.

Partamos el problema en problemas más pequeños para resolver más fácilmente cada uno de ellos, y que su reconstrucción nos ayude a resolver el problema original.

. . . en listas más pequeñas

Review D & Q

Ahora bien, recordemos el enfoque que vimos hace unas semanas: *Divide & Conquer* o en español **Divide y Vencerás**.

Partamos el problema en problemas más pequeños para resolver más fácilmente cada uno de ellos, y que su reconstrucción nos ayude a resolver el problema original.

Merge Sort

El merge sort divide el contenedor de manera recursiva para tener en cada nivel un arreglo ya ordenado:

a) Divide el vector en dos vectores de la mitad del tamaño
b) Vuelvo a partir cada mitad en otros dos
c) ⋮
d) Cuando tenga sólo dos elementos, los pongo en el orden correcto y regreso el *pedacititito* ordenado
e) Junto todos los *pedacititos* ordenados para formar un *pedacito* ordenado
f) Junto todos los *pedacitos* ordenados para formar un *pedazo* ordenado
g) ⋮
h) Junto las dos mitades ordenadas para formar el vector completo ordenado

Merge Sort

El merge sort divide el contenedor de manera recursiva para tener en cada nivel un arreglo ya ordenado:

a) Divide el vector en dos vectores de la mitad del tamaño

b) Vuelvo a partir cada mitad en otros dos

c) ⋮

d) Cuando tenga sólo dos elementos, los pongo en el orden correcto y regreso el *pedacititito* ordenado

e) Junto todos los *pedacitititos* ordenados para formar un *pedacito* ordenado

f) Junto todos los *pedacitos* ordenados para formar un *pedazo* ordenado

g) ⋮

h) Junto las dos mitades ordenadas para formar el vector completo ordenado

Merge Sort

El merge sort divide el contenedor de manera recursiva para tener en cada nivel un arreglo ya ordenado:

a) Divide el vector en dos vectores de la mitad del tamaño
b) Vuelvo a partir cada mitad en otros dos
c) $\vdots$
d) Cuando tenga sólo dos elementos, los pongo en el orden correcto y regreso el *pedacititito* ordenado
e) Junto todos los *pedacititos* ordenados para formar un *pedacito* ordenado
f) Junto todos los *pedacitos* ordenados para formar un *pedazo* ordenado
g) $\vdots$
h) Junto las dos mitades ordenadas para formar el vector completo ordenado

Merge Sort

El merge sort divide el contenedor de manera recursiva para tener en cada nivel un arreglo ya ordenado:

a) Divide el vector en dos vectores de la mitad del tamaño
b) Vuelvo a partir cada mitad en otros dos
c) $\vdots$
d) Cuando tenga sólo dos elementos, los pongo en el orden correcto y regreso el *pedacititito* ordenado
e) Junto todos los *pedacititos* ordenados para formar un *pedacito* ordenado
f) Junto todos los *pedacitos* ordenados para formar un *pedazo* ordenado
g) $\vdots$
h) Junto las dos mitades ordenadas para formar el vector completo ordenado

Merge Sort

El merge sort divide el contenedor de manera recursiva para tener en cada nivel un arreglo ya ordenado:

a) Divide el vector en dos vectores de la mitad del tamaño
b) Vuelvo a partir cada mitad en otros dos
c) ⋮
d) Cuando tenga sólo dos elementos, los pongo en el orden correcto y regreso el *pedacititito* ordenado
e) Junto todos los *pedacitititos* ordenados para formar un *pedacito* ordenado
f) Junto todos los *pedacitos* ordenados para formar un *pedazo* ordenado
g) ⋮
h) Junto las dos mitades ordenadas para formar el vector completo ordenado

Merge Sort

El merge sort divide el contenedor de manera recursiva para tener en cada nivel un arreglo ya ordenado:

a) Divide el vector en dos vectores de la mitad del tamaño
b) Vuelvo a partir cada mitad en otros dos
c) $\vdots$
d) Cuando tenga sólo dos elementos, los pongo en el orden correcto y regreso el *pedacititito* ordenado
e) Junto todos los *pedacitititos* ordenados para formar un *pedacito* ordenado
f) Junto todos los *pedacitos* ordenados para formar un *pedazo* ordenado
g) $\vdots$
h) Junto las dos mitades ordenadas para formar el vector completo ordenado

Merge Sort

Es decir que si tengo un vector de n elementos:

1) Haré $\log_2 n - 1$ particiones
2) En cada partición compararé a lo mucho con la cantidad de elementos del número de vuelta en el que esté, e.g. en la vuelta 1 compararé con n, en la vuelta 2 con $n/2 \ldots$

Es decir

$$\mathcal{O}(n \log n)$$

Este algoritmo es óptimo . . . *¿Por qué?*

Merge Sort

Es decir que si tengo un vector de n elementos:

1) Haré $\log_2 n - 1$ particiones
2) En cada partición compararé a lo mucho con la cantidad de elementos del número de vuelta en el que esté, e.g. en la vuelta 1 compararé con n, en la vuelta 2 con $n/2 \ldots$

Es decir

$$\mathcal{O}(n \log n)$$

Este algoritmo es óptimo . . . *¿Por qué?*

Merge Sort

Es decir que si tengo un vector de n elementos:

1) Haré $\log_2 n - 1$ particiones
2) En cada partición compararé a lo mucho con la cantidad de elementos del número de vuelta en el que esté, e.g. en la vuelta 1 compararé con n, en la vuelta 2 con $n/2 \ldots$

Es decir

$$\mathcal{O}(n \log n)$$

Este algoritmo es óptimo . . . *¿Por qué?*

Merge Sort

Es decir que si tengo un vector de n elementos:

1) Haré $\log_2 n - 1$ particiones
2) En cada partición compararé a lo mucho con la cantidad de elementos del número de vuelta en el que esté, e.g. en la vuelta 1 compararé con n, en la vuelta 2 con $n/2 \ldots$

Es decir

$$\mathcal{O}(n \log n)$$

Este algoritmo es óptimo . . . *¿Por qué?*

Merge Sort

Es decir que si tengo un vector de n elementos:

1) Haré $\log_2 n - 1$ particiones
2) En cada partición compararé a lo mucho con la cantidad de elementos del número de vuelta en el que esté, e.g. en la vuelta 1 compararé con n, en la vuelta 2 con $n/2 \ldots$

Es decir

$$\mathcal{O}(n \log n)$$

Este algoritmo es óptimo . . . *¿Por qué?*

Merge Sort I

```
5  void joinHelper(vector<int> &A, int startpoint, int midpoint, int
   ↪  endpoint);
6  void mergeSort(vector<int> &A, int startpoint, int endpoint);
7
8  void mergeSort(vector<int> &A, int startpoint, int endpoint){
9      if (startpoint < endpoint){
10         int mid = (startpoint + endpoint) / 2;
11         mergeSort(A, startpoint, mid);
12         mergeSort(A, mid+1, endpoint);
13         joinHelper(A, startpoint, mid, endpoint);
14     }
15 }
```

Merge Sort II

```
17 void joinHelper(vector<int> &A, int startpoint, int midpoint, int endpoint){
18     int i = startpoint;
19     int k = startpoint;
20     int j = midpoint + 1;
21     vector<int> aux(endpoint);
22 
23     while(i <= midpoint && j <= endpoint){
24         if (A[i] <= A[j]){
25             aux[k] = A[i];
26             k++;
27             i++;
28         } else{
29             aux[k] = A[j];
30             k++;
31             j++;
32         }
33     }
34     while (i <= midpoint){
35         aux[k] = A[i];
36         k++;
37         i++;
38     }
39     while (j <= endpoint){
40         aux[k] = A[j];
41         k++;
42         j++;
43     }
44     for(int z=startpoint; z < k; z++){
45         A[z] = aux[z];
46     }
47 }
```

Quicksort

El quicksort es otro método **recursivo** para ordenamiento. En este caso, se escoge un pivote y se ordena por partes:

a) Parto en dos mitades y tomo algún elemento como pivote

b) Ordeno la lista poniendo a la izquierda del pivote todo lo menor y a la derecha del pivote lo mayor

c) Vuelvo a partir en mitades de mitades (que ya están medio arregladas con respecto al pivote), y selecciono de nuevo un pivote. . .

⋮

d)

e) Cuando llego al último nivel, ya están ordenados todos los elementos

Quicksort

El quicksort es otro método **recursivo** para ordenamiento. En este caso, se escoge un pivote y se ordena por partes:

a) Parto en dos mitades y tomo algún elemento como pivote

b) Ordeno la lista poniendo a la izquierda del pivote todo lo menor y a la derecha del pivote lo mayor

c) Vuelvo a partir en mitades de mitades (que ya están medio arregladas con respecto al pivote), y selecciono de nuevo un pivote. . .

⋮

d)

e) Cuando llego al último nivel, ya están ordenados todos los elementos

Quicksort

El quicksort es otro método **recursivo** para ordenamiento. En este caso, se escoge un pivote y se ordena por partes:

a) Parto en dos mitades y tomo algún elemento como pivote

b) Ordeno la lista poniendo a la izquierda del pivote todo lo menor y a la derecha del pivote lo mayor

c) Vuelvo a partir en mitades de mitades (que ya están medio arregladas con respecto al pivote), y selecciono de nuevo un pivote. . .

⋮

d)

e) Cuando llego al último nivel, ya están ordenados todos los elementos

Quicksort

El quicksort es otro método **recursivo** para ordenamiento. En este caso, se escoge un pivote y se ordena por partes:

a) Parto en dos mitades y tomo algún elemento como pivote

b) Ordeno la lista poniendo a la izquierda del pivote todo lo menor y a la derecha del pivote lo mayor

c) Vuelvo a partir en mitades de mitades (que ya están medio arregladas con respecto al pivote), y selecciono de nuevo un pivote...

⋮

d)

e) Cuando llego al último nivel, ya están ordenados todos los elementos

Quicksort

El quicksort es otro método **recursivo** para ordenamiento. En este caso, se escoge un pivote y se ordena por partes:

a) Parto en dos mitades y tomo algún elemento como pivote

b) Ordeno la lista poniendo a la izquierda del pivote todo lo menor y a la derecha del pivote lo mayor

c) Vuelvo a partir en mitades de mitades (que ya están medio arregladas con respecto al pivote), y selecciono de nuevo un pivote...

⋮

d)

e) Cuando llego al último nivel, ya están ordenados todos los elementos

QuickSort

Es decir que si tengo un vector de n elementos:

1) Haré en promedio $\log_2 n - 1$ particiones, y en el peor caso $n - 1$ particiones
2) En cada partición compararé siempre al pivote con **todos los demás** elementos... ¿Qué pasa si con una pésima suerte, mi pivote es el número más pequeño en cada vuelta?

Es decir $\mathcal{O}(n^2)$ en el peor caso y $\mathcal{O}(n \log n)$ en el caso promedio.

Este algoritmo también es óptimo[1] en el caso promedio.

[1]

QuickSort

Es decir que si tengo un vector de n elementos:

1) Haré en promedio $\log_2 n - 1$ particiones, y en el peor caso $n - 1$ particiones
2) En cada partición compararé siempre al pivote con **todos los demás** elementos... ¿Qué pasa si con una pésima suerte, mi pivote es el número más pequeño en cada vuelta?

Es decir $\mathcal{O}(n^2)$ en el peor caso y $\mathcal{O}(n \log n)$ en el caso promedio.

Este algoritmo también es óptimo[1] en el caso promedio.

[1]

QuickSort

Es decir que si tengo un vector de n elementos:

1) Haré en promedio $\log_2 n - 1$ particiones, y en el peor caso $n - 1$ particiones
2) En cada partición compararé siempre al pivote con **todos los demás** elementos... ¿Qué pasa si con una pésima suerte, mi pivote es el número más pequeño en cada vuelta?

Es decir $\mathcal{O}(n^2)$ en el peor caso y $\mathcal{O}(n \log n)$ en el caso promedio.

Este algoritmo también es óptimo[1] en el caso promedio.

[1]

QuickSort

Es decir que si tengo un vector de n elementos:

1) Haré en promedio $\log_2 n - 1$ particiones, y en el peor caso $n - 1$ particiones
2) En cada partición compararé siempre al pivote con **todos los demás** elementos... ¿Qué pasa si con una pésima suerte, mi pivote es el número más pequeño en cada vuelta?

Es decir $\mathcal{O}(n^2)$ en el peor caso y $\mathcal{O}(n \log n)$ en el caso promedio.

Este algoritmo también es óptimo[1] en el caso promedio.

[1]

QuickSort

Es decir que si tengo un vector de n elementos:

1) Haré en promedio $\log_2 n - 1$ particiones, y en el peor caso $n - 1$ particiones
2) En cada partición compararé siempre al pivote con **todos los demás** elementos... ¿Qué pasa si con una pésima suerte, mi pivote es el número más pequeño en cada vuelta?

Es decir $\mathcal{O}(n^2)$ en el peor caso y $\mathcal{O}(n \log n)$ en el caso promedio.

Este algoritmo también es óptimo[1] en el caso promedio.

1

Quicksort I

```
4  void mysplit(std::vector<int> &A, int startpoint, int endpoint, int
   ↪  &pivot);
5  void quickSort(std::vector<int> &A, int startpoint, int endpoint);
6
7  void quickSort(std::vector<int> &A, int startpoint, int endpoint){
8      int pivot;
9
10     if(startpoint < endpoint){
11         mysplit(A, startpoint, endpoint, pivot);
12         quickSort(A, startpoint, pivot-1);
13         quickSort(A, pivot+1, endpoint);
14     }
15 }
```

Quicksort II

```
17 void mysplit(std::vector<int> &A, int startpoint, int endpoint, int
   ↪  &pivot){
18     int pivot_value = A[startpoint];
19     int j = startpoint;
20     int i, aux;
21 
22     for(i=startpoint+1; i <= endpoint; i++){
23         if(A[i] < pivot_value){
24             //go left
25             j++;
26             aux = A[i];
27             A[i] = A[j];
28             A[j] = aux;
29         }
30     }
31 
32     pivot = j;
33     aux = A[startpoint];
34     A[startpoint] = A[pivot];
35     A[pivot] = aux;
36 }
```

Funciones y procedimientos

Definición y diferencias

En programación hablamos comúnmente de *la función* `print` o "es una *función* que ordena una lista de números"...

En realidad, muchas veces nos referimos a un **procedimiento** en lugar de una **función**.

- Una función **asocia** entradas con salidas—recibe argumentos y **devuelve** un resultado.
- Un procedimiento simplemente es una secuencia de instrucciones para resolver un problema.

Funciones y procedimientos

Definición y diferencias

En programación hablamos comúnmente de *la función* `print` o "es una *función* que ordena una lista de números"...

En realidad, muchas veces nos referimos a un **procedimiento** en lugar de una **función**.

- Una función **asocia** entradas con salidas—recibe argumentos y **devuelve** un resultado.
- Un procedimiento simplemente es una secuencia de instrucciones para resolver un problema.

Funciones y procedimientos

Definición y diferencias

En programación hablamos comúnmente de *la función* `print` o "es una *función* que ordena una lista de números"...

En realidad, muchas veces nos referimos a un **procedimiento** en lugar de una **función**.

- Una función **asocia** entradas con salidas—recibe argumentos y **devuelve** un resultado.
- Un procedimiento simplemente es una secuencia de instrucciones para resolver un problema.

Funciones y procedimientos

Definición y diferencias

En programación hablamos comúnmente de *la función* `print` o "es una *función* que ordena una lista de números"...

En realidad, muchas veces nos referimos a un **procedimiento** en lugar de una **función**.

- Una función **asocia** entradas con salidas—recibe argumentos y **devuelve** un resultado.
- Un procedimiento simplemente es una secuencia de instrucciones para resolver un problema.

Funciones y procedimientos

Definición y diferencias

En programación hablamos comúnmente de *la función* `print` o "es una *función* que ordena una lista de números"...

En realidad, muchas veces nos referimos a un **procedimiento** en lugar de una **función**.

- Una función **asocia** entradas con salidas—recibe argumentos y **devuelve** un resultado.
- Un procedimiento simplemente es una secuencia de instrucciones para resolver un problema.

¿Por qué es importante entender la diferencia?

Funciones y procedimientos

La principal diferencia entre ellas es que la función tiene un valor de retorno y el procedimiento no:

- **Imprimir todos los contenidos de un vector**. Los imprimes y ya.
- **Multiplicar dos matrices**. NECESITAS el resultado (para guardarlo en algún lugar, por ejemplo).
- **Pausa el sistema por 20 segundos**. No necesitas resultado.
- **Ordena una lista de números**. Necesitas un resultado. O lo ordenas y ya...

Es poco común tener **procedimientos**. Lo que se acostumbra a hacer en esos casos es devolver 0 si todo salió correcto, o -1 si hubo algún error.

¿Por qué es importante entender la diferencia?

Funciones y procedimientos

La principal diferencia entre ellas es que la función tiene un valor de retorno y el procedimiento no:

- **Imprimir todos los contenidos de un vector**. Los imprimes y ya.
- **Multiplicar dos matrices**. NECESITAS el resultado (para guardarlo en algún lugar, por ejemplo).
- **Pausa el sistema por 20 segundos**. No necesitas resultado.
- **Ordena una lista de números**. Necesitas un resultado. O lo ordenas y ya...

Es poco común tener **procedimientos**. Lo que se acostumbra a hacer en esos casos es devolver 0 si todo salió correcto, o -1 si hubo algún error.

¿Por qué es importante entender la diferencia?

Funciones y procedimientos

La principal diferencia entre ellas es que la función tiene un valor de retorno y el procedimiento no:

- **Imprimir todos los contenidos de un vector**. Los imprimes y ya.
- **Multiplicar dos matrices**. NECESITAS el resultado (para guardarlo en algún lugar, por ejemplo).
- **Pausa el sistema por 20 segundos**. No necesitas resultado.
- **Ordena una lista de números**. Necesitas un resultado. O lo ordenas y ya...

Es poco común tener **procedimientos**. Lo que se acostumbra a hacer en esos casos es devolver 0 si todo salió correcto, o -1 si hubo algún error.

¿Por qué es importante entender la diferencia?

Funciones y procedimientos

La principal diferencia entre ellas es que la función tiene un valor de retorno y el procedimiento no:

- **Imprimir todos los contenidos de un vector**. Los imprimes y ya.
- **Multiplicar dos matrices**. NECESITAS el resultado (para guardarlo en algún lugar, por ejemplo).
- **Pausa el sistema por 20 segundos**. No necesitas resultado.
- **Ordena una lista de números**. Necesitas un resultado. O lo ordenas y ya...

Es poco común tener **procedimientos**. Lo que se acostumbra a hacer en esos casos es devolver 0 si todo salió correcto, o -1 si hubo algún error.

¿Por qué es importante entender la diferencia?

Funciones y procedimientos

La principal diferencia entre ellas es que la función tiene un valor de retorno y el procedimiento no:

- **Imprimir todos los contenidos de un vector**. Los imprimes y ya.
- **Multiplicar dos matrices**. NECESITAS el resultado (para guardarlo en algún lugar, por ejemplo).
- **Pausa el sistema por 20 segundos**. No necesitas resultado.
- **Ordena una lista de números**. Necesitas un resultado. O lo ordenas y ya...

Es poco común tener **procedimientos**. Lo que se acostumbra a hacer en esos casos es devolver 0 si todo salió correcto, o -1 si hubo algún error.

¿Por qué es importante entender la diferencia?

Funciones y procedimientos

La principal diferencia entre ellas es que la función tiene un valor de retorno y el procedimiento no:

- **Imprimir todos los contenidos de un vector**. Los imprimes y ya.
- **Multiplicar dos matrices**. NECESITAS el resultado (para guardarlo en algún lugar, por ejemplo).
- **Pausa el sistema por 20 segundos**. No necesitas resultado.
- **Ordena una lista de números**. Necesitas un resultado. O lo ordenas y ya...

Es poco común tener **procedimientos**. Lo que se acostumbra a hacer en esos casos es devolver 0 si todo salió correcto, o -1 si hubo algún error.

Functions e Inplace Functions

Definiciones y diferencias

Ya vimos que una función me devuelve un resultado. Sin embargo, existen algunas funciones (como la del caso de ordenar una lista) en donde no está muy claro si lo que se desea hacer necesito hacerlo en un lugar *aparte* o ahí en el *mismo contenedor*.

Este tipo de funciones que operan en **mismo lugar** al que hacen *referencia* se les llama funciones *inplace*.

Su principal uso es para ahorrar memoria, cuando se trabaja en ambientes que necesitan ser optimizados por limitaciones de espacio o si estamos trabajando con contenedores gigantescos que no podamos copiar completamente en RAM.

Functions e Inplace Functions

Definiciones y diferencias

Ya vimos que una función me devuelve un resultado. Sin embargo, existen algunas funciones (como la del caso de ordenar una lista) en donde no está muy claro si lo que se desea hacer necesito hacerlo en un lugar *aparte* o ahí en el *mismo contenedor*.

Este tipo de funciones que operan en **mismo lugar** al que hacen *referencia* se les llama funciones *inplace*.

Su principal uso es para ahorrar memoria, cuando se trabaja en ambientes que necesitan ser optimizados por limitaciones de espacio o si estamos trabajando con contenedores gigantescos que no podamos copiar completamente en RAM.

Functions e Inplace Functions

Definiciones y diferencias

Ya vimos que una función me devuelve un resultado. Sin embargo, existen algunas funciones (como la del caso de ordenar una lista) en donde no está muy claro si lo que se desea hacer necesito hacerlo en un lugar *aparte* o ahí en el *mismo contenedor*.

Este tipo de funciones que operan en **mismo lugar** al que hacen *referencia* se les llama funciones *inplace*.

Su principal uso es para ahorrar memoria, cuando se trabaja en ambientes que necesitan ser optimizados por limitaciones de espacio o si estamos trabajando con contenedores gigantescos que no podamos copiar completamente en RAM.

Scopes y Variables

Más definiciones

Específicamente en C++ las variables que declaramos en una función *viven* solamente **dentro** de esa función. Sin embargo, podemos enviarlas de distintas maneras a otras funciones para que las modifiquen:

Pass by value

Cuando enviamos una variable por su valor, se genera una **copia** en la función que la recibe y trabaja con su copia de manera local para hacer todos los cálculos. Los resultados que te envíen tienen que ser guardados en algún lugar.

Pass by reference

Cuando enviamos una variable por referencia, la función que la recibe trabajará **directamente** sobre ella. Las modificaciones que se hagan a dicha variable continuarán incluso si el resultado no se guarda.

Scopes y Variables

Más definiciones

Específicamente en C++ las variables que declaramos en una función *viven* solamente **dentro** de esa función. Sin embargo, podemos enviarlas de distintas maneras a otras funciones para que las modifiquen:

Pass by value

Cuando enviamos una variable por su valor, se genera una **copia** en la función que la recibe y trabaja con su copia de manera local para hacer todos los cálculos. Los resultados que te envíen tienen que ser guardados en algún lugar.

Pass by reference

Cuando enviamos una variable por referencia, la función que la recibe trabajará **directamente** sobre ella. Las modificaciones que se hagan a dicha variable continuarán incluso si el resultado no se guarda.

Scopes y Variables

Más definiciones

Específicamente en C++ las variables que declaramos en una función *viven* solamente **dentro** de esa función. Sin embargo, podemos enviarlas de distintas maneras a otras funciones para que las modifiquen:

Pass by value

Cuando enviamos una variable por su valor, se genera una **copia** en la función que la recibe y trabaja con su copia de manera local para hacer todos los cálculos. Los resultados que te envíen tienen que ser guardados en algún lugar.

Pass by reference

Cuando enviamos una variable por referencia, la función que la recibe trabajará **directamente** sobre ella. Las modificaciones que se hagan a dicha variable continuarán incluso si el resultado no se guarda.

Referencias en C++

Scopes y Variables

Una función simple con argumentos pasados por **valor**:

Pass by value

```
1 int addTwo(int x){
2     return x + 2;
3 }
```

¿Cuánto vale x y cuánto y si en el `main` la invocamos como `x = 5; y = addTwo(x);`?

Referencias en C++

Scopes y Variables

Una función simple con argumentos pasados por **referencia**:

Pass by reference

```
1 void addThreeIfOddOrOneIfEven(int &x){
2     if(x % 2 == 1){
3         x += 3;
4     }
5     else{
6         x++;
7     }
8 }
```

¿Cuánto vale x si en el `main` la invocamos como `x = 5; addThreeIfOddOrOneIfEven(x);`?