INF-134
Estructuras de Datos

Conceptos Fundamentales y Lenguaje de Programación C++

Diego Arroyuelo - diego.arroyuelo@usm.cl
Roberto Díaz - roberto.diazu@usm.cl

Universidad Técnica Federico Santa María
Campus Santiago San Joaquín

¿Qué son las estructuras de datos?

La mayoría de los programas necesitan manipular datos.

En la actualidad, muchas aplicaciones deben manipular grandes volúmenes de datos:

Hacerlo de manera eficiente se vuelve una tarea no trivial

Por ejemplo:

- Registrar las transacciones de usuario con su tarjeta de débito, para poder consultar las transacciones realizadas por esa persona durante un mes.
- Almacenar información geográfica de una región para poder obtener aquellos elementos que se ubican dentro de un rectángulo de búsqueda.
- Almacenar un mapa de una ciudad para poder encontrar el camino más corto entre dos puntos dados del mapa.

¿Qué son las estructuras de datos?

En informática, una estructura de datos es una manera particular de organizar los datos en un computador de manera que puedan ser usados y procesados eficientemente a través de un programa.

Paul E. Black (ed.), definición de estructura de datos en Dictionary of Algorithms and Data Structures. U.S. National Institute of Standards and Technology. 15 de Diciembre, 2004.

Definición de estructura de datos en Encyclopædia Britannica (2009).

¿Qué son las estructuras de datos?

Las diferentes estructuras de datos que existen son adecuadas para diferentes aplicaciones.

Algunas estructuras de datos son altamente especializadas para tareas específicas.

Las estructuras de datos proveen una manera de manejar grandes cantidades de datos eficientemente:

- Grandes bases de datos.
- Indexación de la web.
- Etc.

Usualmente, las estructuras de datos son la clave para diseñar algoritmos eficientes.

Modelo de Computación RAM

Durante el curso asumiremos el modelo de computación RAM (Random Access Machine)

Definición: una RAM está compuesta por:

- **Unidad de Procesamiento** (CPU).
- **Memoria Principal** compuesta por una cantidad infinita de celdas, M[0], M[1], M[2], …, las cuales pueden ser accedidas de manera aleatoria. Cada celda (o palabra de memoria) puede almacenar un entero (por ejemplo, de 32 bits). Sólo una cantidad finita de celdas de memoria está en uso en cada momento.
- Un número limitado de **registros del procesador** R1, R2, …, RK.
- Un **conjunto de instrucciones** que pueden ser ejecutadas por la CPU. Cada instrucción toma 1 ciclo del procesador para ser ejecutada.

Programa en tiempo de ejecución

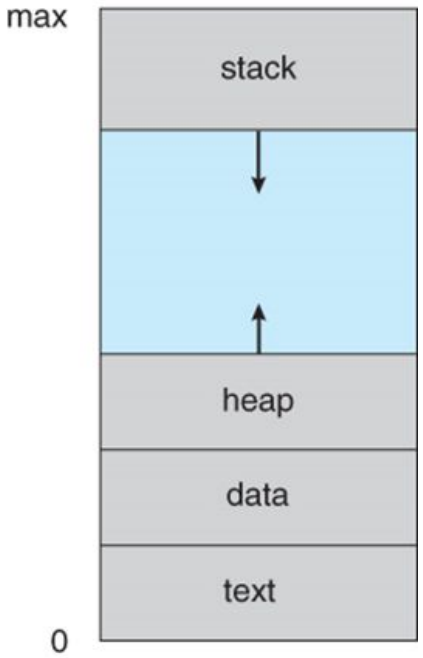
Asumimos además el modelo de John von Neumann, en donde para su ejecución los programas son almacenados en memoria principal, junto con los datos.

En tiempo de ejecución, un programa es usualmente implementado por el sistema operativo de la siguiente manera:

- **Segmento de código**: contiene el código del programa que es usado por el procesador para la ejecución.
- **Segmento de datos**: contiene las variables locales de las funciones que son invocadas a lo largo de la ejecución.
- **Segmento de memoria dinámica**: contiene la memoria dinámica que es pedida por un programa en tiempo de ejecución.

Programa en tiempo de ejecución

Gráficamente, un programa en ejecución puede verse de la siguiente manera en memoria principal:



Segmento de datos: variables locales, crece dinámicamente

Segmento de memoria dinámica: crece dinámicamente

Variables globales y estáticas

Segmento de código: contiene el programa compilado y es cargado en memoria cuando se ejecuta el programa

Lenguaje C++

Para apoyar el aprendizaje de EDDs en el curso usaremos principalmente el **lenguaje C++**.

- Definido por Bjarne Stroustrup en 1985.
- Se recomienda el tutorial que se encuentra en aula.
- Este curso no pretende enseñar de manera completa el lenguaje C++

A diferencia de Python, **C++ es un lenguaje compilado**:

- Un compilador de C++ traduce un programa escrito en lenguaje C++ a lenguaje máquina (es decir, código que es capaz de ser interpretado por un computador).

C++ es un lenguaje con tipificación explícita:

- Al escribir un programa se deben indicar los tipos de cada uno de los identificadores introducidos por el programador

Lenguaje C++

En el curso usaremos principalmente el **compilador g++**.

La forma de usarlo en Linux para compilar un programa en un archivo test.cpp es:

```
g++ test.cpp -o test-objeto
```

El compilador generará un código objeto (ejecutable) llamado test-objeto.

Se recomienda buscar documentación del compilador, por ejemplo en Linux mediante el comando:

```
man g++
```

Lenguaje C++

Ejemplo de programa básico en C++:

```
#include <iostream>

/* este programa imprime Hola Mundo por pantalla */

int main(){
    std::cout << "Hola Mundo" << std::endl;
    return 0;
}
```

Lenguaje C++

Ejemplo de programa básico en C++:

```
#include <iostream>
using namespace std;
/* este programa imprime Hola Mundo por pantalla */

int main(){
     cout << "Hola Mundo" << endl;
     return 0;
}
```

Lenguaje C++ - Tipos de Datos

La sentencia de declaración de variables es una de las más importantes.

Para declarar una variable x de tipo entero se debe escribir:

```
int x;
```

El tipo condiciona los valores que puede contener la variable.

El compilador usa esta información para reservar espacio de almacenamiento en memoria para la variable.

En C++ es obligatorio declarar todas las variables que van a ser usadas.

Lenguaje C++ - Tipos de Datos

C++ provee los siguientes tipos básicos:

- Tipo entero: `int x;`
- Tipo booleano: `bool x;`
- Tipo flotante (real): `float x;`

Nota: usar el operador `sizeof` para saber cuántos bytes usa el compilador para implementar un tipo dado, por ejemplo:

```
sizeof(int)
```

Se puede transformar una variable de un tipo a otro por medio de un *cast*. Para transformar la variable entera `x` a flotante:

```
(float) x;
```

Lenguaje C++ - Tipos de Datos

Como se vió anteriormente, el objeto **`cout`** permite mostrar texto por pantalla, pero también puede usarse para mostrar valores de variables.

Se puede pasar a **`cout`** una cadena de texto o variables usando el operador **`<<`**. El orden en que se pasan los elementos es el orden en que se muestran.

Para mostrar un texto junto a dos variables de tipo entero **`x`** e **`y`**, se debe utilizar como:

```
cout << "Los valores son: x=" << x << " y=" << y;
```

Si x = 100 e y = 200, entonces se mostrará por pantalla:

```
Los valores son: x=100, y=200
```

Lenguaje C++ - Operadores

C++ provee un conjunto importante de operadores, lo cual lo distingue de otros lenguajes. Entre los operadores más básicos se encuentran:

Aritméticos	`+ - * / %`
Comparación	`== != < <= > >=`
Lógicos	`&& || !`
Asignación	`= += -= *= /=`
Incremento/decremento prefijo y postfijo	`++ --`

Lenguaje C++ - Operadores

Es importante entender la tabla de Precedencia y Asociatividad de las operaciones

Precedencia	Asociatividad
`() [] -> . ++(sufijo) --(sufijo)`	izqda a drcha
`++(prefijo) --(prefijo) ! sizeof(tipo) +(unario) -(unario) *(indir.) &(dirección) new delete`	drcha a izqda
`* / %`	izqda a drcha
`+ -`	izqda a drcha
`< <= > >=`	izqda a drcha
`== !=`	izqda a drcha
`&&`	izqda a drcha
`||`	izqda a drcha
`?:`	drcha a izqda
`= += -= += /=`	drcha a izqda
`, (operador coma)`	izqda a drcha

Lenguaje C++ - Estructuras de Control

C++ ofrece las siguientes estructuras de control: `if`, `if-else`, `while`, `do-while` y `for`.

Un `if-else` permite elegir entre dos alternativas en base a un valor dado.

En C++ un valor igual a cero se considerará falso, mientras que cualquier valor distinto de cero se considerará verdadero.

La sintaxis de `if-else` es:

```
if(condición){
    /* instrucciones si condición es verdadera */
} else {
    /* instrucciones si condición es falsa */
}
```

Lenguaje C++ - Estructuras de Control

C++ ofrece las siguientes estructuras de control: `if`, `if-else`, `while`, `do-while` y `for`.

Un `while` permite repetir la ejecución de una sección de código mientras se cumpla una cierta condición de repetición.

La sintaxis de `while` es:

```
while(condición){
    /* instrucciones a ejecutar mientras condición sea verdadera */
}
/* esto se ejecutara apenas condición sea falsa */
```

Lenguaje C++ - Estructuras de Control

C++ ofrece las siguientes estructuras de control: **`if`**, **`if-else`**, **`while`**, **`do-while`** y **`for`**.

Un **`do-while`** es similar a un while, pero verifica la condición de repetición luego de la primera iteración. Es decir, se ejecuta el código al menos una vez.

La sintaxis de **`do-while`** es:

```
do {
    /* Instrucciones a ejecutar mientras condicion sea verdadera.
    Se ejecutaran al menos una vez */
} while(condición);
/* esto se ejecutara apenas condición sea falsa */
```

Lenguaje C++ - Estructuras de Control

C++ ofrece las siguientes estructuras de control: `if`, `if-else`, `while`, `do-while` y `for`.

Un `for` es una estructura de repetición que incorpora tres expresiones: asignación inicial, condición de repetición y un incremento.

Al toparse con un `for` se ejecuta la expresión de **asignación inicial** y se procede a iterar sobre el código correspondiente mientras se cumpla la **condición de repetición**. Al final de cada iteración se ejecuta la expresión de **incremento**.

La sintaxis del `for` es:

```
for ( asignacion-inicial ; condicion ; incremento ){
    /* operación a llevar a cabo en el for */
}
```

Lenguaje C++ - Estructuras de Control

C++ ofrece las siguientes estructuras de control: `if`, `if-else`, `while`, `do-while` y `for`.

Por ejemplo, el siguiente programa imprime los números desde el 1 al 100

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

int main() {
    for(int i=1; i<=100; ++i){
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

Lenguaje C++ - Arreglos

Los arreglos son quizás la estructura de datos más básica que estudiaremos.

- Son provistos como tipo de datos por la mayoría de los lenguajes de programación

Permiten almacenar un conjunto de datos en memoria principal.

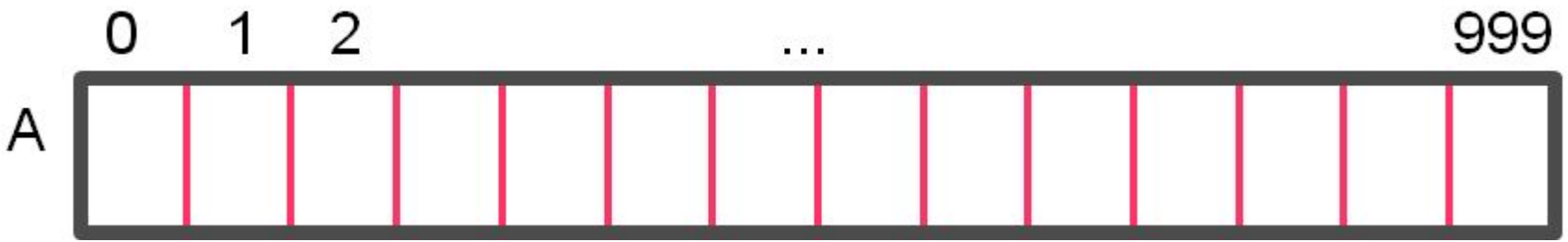
- Todos los datos tienen el mismo tipo, conocido como el **tipo base** del arreglo.
- Los elementos del arreglo son almacenados de manera contigua dentro de la memoria
- Los elementos del arreglo se acceden usando un índice: primer elemento, segundo elemento, etc.

Lenguaje C++ - Arreglos

En el lenguaje un arreglo que almacena 1000 números enteros se puede declarar de la siguiente manera:

```
int a[1000];
```

Este arreglo puede verse gráficamente de la siguiente manera:



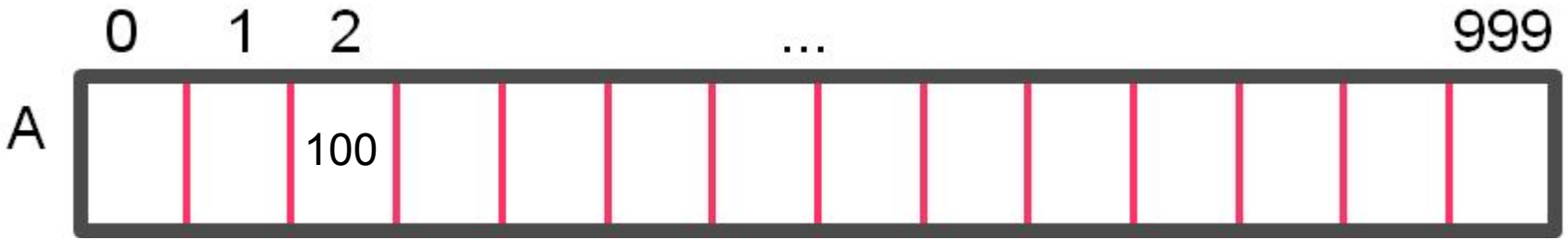
En C++, los arreglos siempre se enumeran desde 0.

Lenguaje C++ - Arreglos

Una vez declarado, se puede usar del operador de selección "`[ ]`" para acceder a los elementos del arreglo:

```
a[2] = 100;
```

Lo cual produce un cambio en el arreglo:



Importante: un arreglo ocupa siempre un área contigua de memoria.

Lenguaje C++ - Arreglos

Esquema de Almacenamiento: como se mencionó anteriormente un arreglo en C++ se almacena en celdas contiguas de memoria.

Importante: el nombre de un arreglo siempre se evalúa a su dirección base

Ejemplo:

```
int a[10];
cout << "La direccion base del arreglo A es " << a << "\n";
for (int i=0; i<10; ++i)
    cout << "Direccion de a[" << i << "] es " << &a[i] << "\n";
```

El operador `&` permite obtener la dirección de memoria que almacena una variable.

Lenguaje C++ - Structs

Los **structs (o registros)** son conjuntos de una o más variables que pueden ser de distintos tipos, agrupadas bajo un sólo nombre.

Ejemplo de declaración:

```
struct empleado {
    int codigo;
    float sueldo;
};
```

Permite que un grupo de variables relacionadas se les trate como una unidad.

Cada una de las variables que componen un struct es llamada **campo**.

Lenguaje C++ - Structs

Una vez declarado un struct, se pueden declarar variables de la siguiente manera:

```
empleado emp;
```

Los campos de un struct se pueden acceder usando el selector de campo ”.”

Ejemplo:

```
emp.codigo = 1000;
emp.sueldo = 2540.87;
cout << "Dirección base de emp:" << &emp << "\n";
cout << "Dirección del campo emp.codigo:" << &emp.codigo << "\n";
cout << "Dirección del campo emp.sueldo:" << &emp.sueldo << "\n";
```

Lenguaje C++ - Arreglos de Structs

En C++ los arreglos y structs se pueden combinar de manera de tener arreglos de structs, los structs con campos de tipo arreglo, etc.

```
empleado emp[10]; /*arreglo de 10 empleados*/
```

Ejemplo:

```
emp[0].codigo = 1000;
emp[0].sueldo = 2540.87;
emp[1].codigo = 1001;
emp[1].sueldo = 8740.25;
```

Lenguaje C++ - Punteros

Un puntero es una variable que puede almacenar la dirección en donde se almacena otra variable.

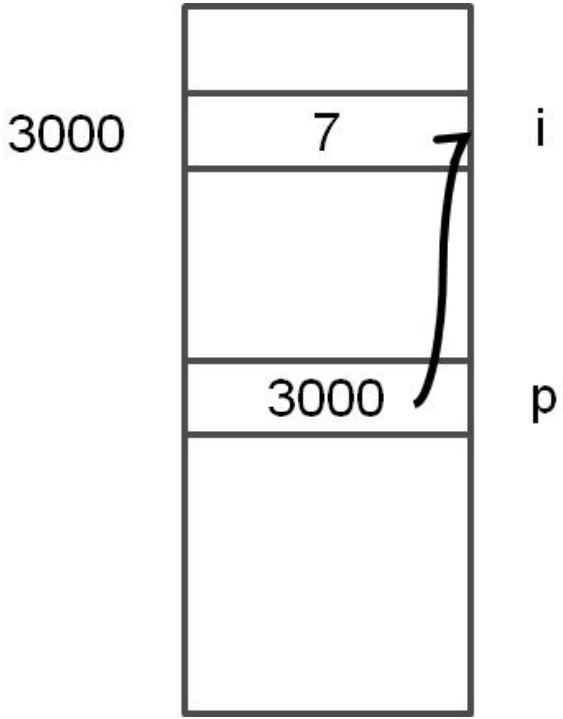
Se dice que el puntero *apunta* a dicha variable.

Ejemplo:

```
int main() {
  int *p; /* puntero para apuntar a una variable de tipo entero */
  int i = 7;
  p = &i;
  (*p)++; /* es equivalente a *p = *p + 1;  */
  cout << "valor de i=" << i << " que es igual a *p=" << *p << "\n";
  return 0;
}
```

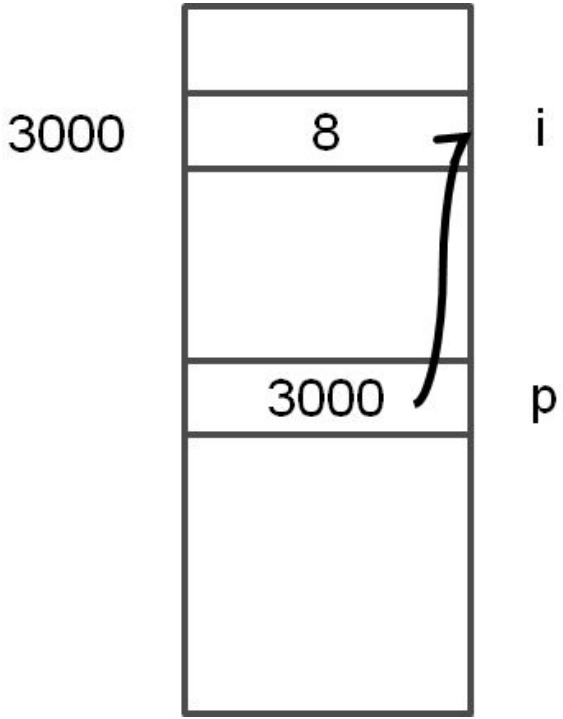
Lenguaje C++ - Punteros

Gráficamente, antes del incremento:



Lenguaje C++ - Punteros

Gráficamente, después del incremento:



Lenguaje C++ - Punteros

Ejemplo usando structs:

```
int main() {
     empleado *p; /*puntero a una variable de tipo empleado*/
     empleado emp;
     emp.codigo = 1000;
     emp.sueldo = 2540.87;
     p = &emp;
     (*p).codigo++;
     p->codigo++;  /*las dos últimas líneas son equivalentes*/
     cout << "Valor del codigo es igual a " << emp.codigo << "\n";
     return 0;
}
```

Lenguaje C++ - Punteros y Arreglos

Los punteros y arreglos están muy relacionados en C++: recordar que el nombre de un arreglo es en realidad un puntero a su dirección base.

```
int main() {
    int *p; /* puntero a una variable de tipo entero */
    int A[10], i;
    p = A; /* p apunta a la dirección base de A */
    for (i=0; i < 10; i++) {
        cout << *p << " ";
        p++; /* apunta al siguiente elemento del arreglo */
    }
    cout << endl;
    return 0;
}
```

Lenguaje C++ - Funciones

Una función es un conjunto de definiciones, expresiones y sentencias que realizan una tarea específica dentro de un programa.

Para la definición de una función en C++, el formato general a seguir es:

```
tipo-de-retorno nombre-de-funcion ( lista-de-parametros ){
    declaracion-variables-locales;
    codigo-de-la-funcion;
}
```

La lista de parámetros puede ser vacía.

Las funciones terminan y regresan a su invocante al llegar al último "`}`", o bien se puede forzar el regreso con la sentencia `return`.

Lenguaje C++ - Funciones

- Los parámetros formales de una función son los definidos en la lista de parámetros de dicha función.
- A través de los parámetros, una función recibe datos de entrada desde la función invocante.
- La definición de los mismos es similar a la de una variable normal.
- Los parámetros formales actúan como una variable local de la función.
- Los parámetros usados en la invocación a una función son llamados parámetros reales.
- Los parámetros reales se asocian con los parámetros formales correspondientes.
- **El paso de parámetros en C++ es por valor o copia si no se especifica nada, aunque permite paso por referencia.**

Lenguaje C++ - Funciones

- Dado que el paso de parámetros en C++ es por valor, no es posible que los cambios realizados en un parámetro de una función `f` se vean reflejados en la función que invoca a `f`.
- En muchos casos es necesario que dichos cambios se vean reflejados en la función que invoca.
- En C++ existen dos soluciones para que la función pueda modificar el parámetro real:
  - Pasarlo como un puntero
  - Pasarlo como una referencia

Lenguaje C++ - Funciones

Considerar la siguiente función:

```
void swap(int* a, int* b){
     int tmp = *a;
     *a = *b;
     *b = tmp;
}

int main(){
     int a = 10;
     int b = 5;
     swap(&a, &b);
     cout << "Post-swap: a = " << a << ", b = " << b << endl;
     return 0;
}
```

Lenguaje C++ - Funciones

Considerar la siguiente función:

```
void swap(int& a, int& b){
     int tmp = a;
     a = b;
     b = tmp;
}

int main(){
     int a = 10;
     int b = 5;
     swap(a, b);
     cout << "Post-swap: a = " << a << ", b = " << b << endl;
     return 0;
}
```

Lenguaje C++ - Funciones

- Muchas funciones necesitan recibir arreglos como parámetros, cuyos datos deben ser procesados de alguna manera.
- En C++, se debe definir el parámetro formal como un puntero al tipo de los elementos del arreglo.
- Como parámetro real se usa el nombre del arreglo (recordar que el nombre del arreglo se evalúa a la dirección base del mismo).
- En definitiva, estaremos pasando como parámetro un puntero a la dirección base del arreglo.
- Además, se debe incluir un parámetro de tipo `int` que indique el tamaño del arreglo
- C++ no tiene otra forma de conocer el tamaño de un arreglo.

Lenguaje C++ - Funciones

Considerar la siguiente función:

```
float calcPromedio(int A[], int n){
    int sum = 0, i;
    for(i=0;i<n;i++)
        sum += A[i];
    return (float)sum/n;
}

int main(){
    int A[5] = {8, 3, 12, 15, 9};
    int B[10] = {2, 8, 2, 5, 9, 4, 7, 3, 8, 4};
    cout << "Promedio arreglo A:" << calcPromedio(A, 5)  << "\n";
    cout << "Promedio arreglo B:" << calcPromedio(B, 10) << "\n";
    return 0;
}
```

Lenguaje C++ - Funciones

Recordar que en C++ un arreglo es básicamente un puntero. Además cuando una función termina, las variables locales son liberadas de la memoria.

Considerar la siguiente función:

```
int* arreglodenumeros(int t){
    int arr[t];
    for(int i=0;i<t;i++){
        arr[i] = i;
    }
    return arr;
}
```

Lenguaje C++ - Funciones

Recordar que en C++ un arreglo es básicamente un puntero. Además cuando una función termina, las variables locales son liberadas de la memoria.

Considerar la siguiente función:

```
int* arreglodenumeros(int t){
    int arr[t];
    for(int i=0;i<t;i++){
        arr[i] = i;
    }
    return arr;
}
```

Una vez que se retorna desde esta función, la memoria asociada al arreglo `arr` se considera como liberada. Su uso puede llevar a comportamiento indeterminado y debe evitarse.

Lenguaje C++ - Memoria Dinámica

Los problemas antes descritos pueden evitarse con un manejo manual de la memoria.

C++ permite manejar la memoria del heap por medio del uso de los operadores `new[]`, `delete[]`.

- `new[]` asigna una región contigua de memoria y retorna un puntero apuntando a dicha dirección de memoria. Recibe el tipo de datos.
- `delete[]` permite liberar una región de memoria asignada con `new[]`. Recibe el puntero a la dirección de memoria a liberar.

Cada llamado a `new` debe tener su correspondiente llamado a `delete`. Si no, la memoria quedará asignada.

Lenguaje C++ - Memoria Dinámica

La función `arreglodenumeros()` antes vista ahora se puede escribir como:

```
int* arreglodenumeros(int t){
    int* arr = new int[t];
    for(int i=0;i<t;i++)
        arr[i] = i;
    return arr;
}
```

Lenguaje C++ - Memoria Dinámica

La función `arreglodenumeros()` antes vista ahora se puede escribir como:

```
int* arreglodenumeros(int t){
    int* arr = new int[t];
    for(int i=0;i<t;i++)
        arr[i] = i;
    return arr;
}
int main(){
    int* a;
    a = arreglodenumeros(10);
    /* ... usar arreglo a */

    return 0;
}
```

Lenguaje C++ - Memoria Dinámica

La función `arreglodenumeros()` antes vista ahora se puede escribir como:

```
int* arreglodenumeros(int t){
    int* arr = new int[t];
    for(int i=0;i<t;i++)
        arr[i] = i;
    return arr;
}
int main(){
    int* a;
    a = arreglodenumeros(10);
    /* ... usar arreglo a */
    delete[] a;
    return 0;
}
```

Lenguaje C++ - Memoria Dinámica

Considere la función `numerospares()` que filtra un arreglo `inarr` de tamaño `m` dejando solo los números pares:

```
int* numerospares(int inarr[], int m){
    int count=0, i=0, j=0;
    for(i=0;i<m;++i)
        if( inarr[i]%2 == 0 )
            count++;
    int* outarr = new int[count];
    for(i=0;i<m;++i)
        if( inarr[i]%2 == 0 )
            outarr[j++] = inarr[i];
    return outarr;
}
```

Lenguaje C++ - Memoria Dinámica

Considere la función `numerospares()` que filtra un arreglo `inarr` de tamaño `m` dejando solo los números pares:

```
int* numerospares(int inarr[], int m){
    int count=0, i=0, j=0;
    for(i=0;i<m;++i)
        if( inarr[i]%2 == 0 )
            count++;
    int* outarr = new int[count];
    for(i=0;i<m;++i)
        if( inarr[i]%2 == 0 )
            outarr[j++] = inarr[i];
    return outarr;
}
```

No hay como saber el tamaño de `outarr`, luego del retorno de la función.

Lenguaje C++ - Memoria Dinámica

Considere la función `numerospares()` que filtra un arreglo `inarr` de tamaño `m` dejando solo los números pares:

```
int* numerospares(int inarr[], int m, int* count){
    int i=0, j=0;
    (*count)=0;
    for(i=0;i<m;++i)
        if( inarr[i]%2 == 0 )
            (*count)++;
    int* outarr = new int[*count];
    for(i=0;i<m;++i)
        if( inarr[i]%2 == 0 )
            outarr[j++] = inarr[i];
    return outarr;
}
```

Lenguaje C++ - Variables globales

Una variable global es una variable que se encuentra definida fuera de una función. Puede ser accedida desde cualquier punto del programa.

```
#include <iostream>

int a = 0;
void una_funcion(){
    a += 3;
}

int main(){
    a = 5;
    una_funcion();
    cout << "El valor de a es " << a << endl;
    return 0;
}
```

Lenguaje C++ - Constantes

En C++ se pueden definir variables constantes con el modificador **`const`**. El compilador lanzará un error si se intenta modificar dicha variable.

```
float const pi = 3.141592;
```

El modificador **`const`** limita de forma distinta la modificación de una variable puntero dependiendo de donde se ubique:

- **`int * const i;`** implica que el puntero es constante y no puede ser modificado, pero el dato al que apunta sí.
- **`int const * i;`** implica que el puntero puede cambiar, pero este no se puede usar para modificar el dato al que apunta.

Lenguaje C++ - Macros

Las macros son expresiones que el compilador va a reemplazar por un valor dado.

```
#define N 10
```

Si luego en el código se tiene la expresión **N** en algún punto:

```
int a[N];
```

N se reemplazará por **10**, o sea, el compilador transformará la expresión anterior en:

```
int a[10];
```

Así las macros son otra forma útil para definir constantes con la ventaja de que no ocupan espacio en memoria. Se deben usar con cuidado porque reemplazan cualquier expresión que calce. Por ejemplo ahora no se podría declarar una variable como:

```
int N = 0;
```

Lenguaje C++ - Strings

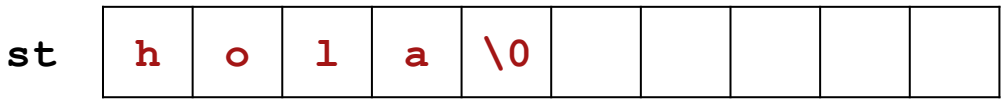
Un string corresponde a una cadena de caracteres. En C++ existen dos representaciones:

- Arreglos de caracteres (y punteros)
- Objetos de tipo string

Los strings en arreglos de caracteres pueden ser de tamaño variable estando delimitados dentro del arreglo por el carácter nulo `'\0'`.

```
char st[10] = "hola";
```

La expresión anterior se puede ver gráficamente como:



Lenguaje C++ - Strings

Para manejar strings representados como arreglos de caracteres se deben ocupar las funciones que se encuentran en la librería `<cstring>`, algunas de las cuales son:

- `size_t strlen(char* s)`: retorna el largo del string apuntado por `s`.
- `int strcmp(char* lhs, char* rhs)`: que compara dos strings, retornando 0 si es que son iguales.
- `char* strcpy(char* d, char* s)`: copia el string `s` en `d`, reemplazando su contenido.
- `char* strcat(char* d, char* s)`: concatena el string `s` en `d`.
- `char* strchr(char* s, char c)`: retorna un puntero a la primera aparición del carácter `c` en el string `s`.

Lenguaje C++ - Strings

Los objetos de tipo string son una forma más conveniente de manejar cadenas de caracteres. Se debe agregar la libreria `<string>`. Se puede crear un string asignado a una cadena de la siguiente forma:

```
string s1 = "hola";
```

Se pueden realizar múltiples operaciones sobre el objeto `s1`:

- `s1.length()`: retorna el largo del string `s1`
- `s1.empty()`: verifica si `s1` está vacío
- `s1[i]`: accede al i-ésimo carácter de `s1`
- `s1 + s2`: concatena los strings `s1` y `s2`
- `s1 == s2`: compara si ambos strings son iguales, retornando verdadero de serlo
- `s1.find('o')`: posición de la primera ocurrencia en `s1` del carácter `'o'`
- `s1.find(s2)`: posición de la primera ocurrencia en `s1` del substring `s2`

Lenguaje C++ - Archivos

Los archivos en C++ son vistos como una secuencia de bytes, que usualmente se almacena en memoria secundaria.

Permiten almacenar grandes volúmenes de información, dado que su capacidad máxima está dada por el espacio disponible en disco.

Para leer y escribir archivos se debe incluir la libreria **<fstream>** y usar un stream de archivo de entrada o salida, que se declaran respectivamente como:

```
ifstream fin;  // archivo de entrada (lectura)
ofstream fout; // archivo de salida (escritura)
fstream  file; // archivo de entrada-salida
```

Lenguaje C++ - Archivos

Un stream de archivo debe asociarse con un archivo usando la operación **open()**.

```
#include <fstream>

int main(){
    std::fstream file; /*stream de archivo*/
    /* asociar stream con archivo de salida.txt */
    file.open("salida.txt", opciones);
    /* ... */
    return 0;
}
```

En donde **opciones** permitirá definir opciones en la apertura del archivo.

Lenguaje C++ - Archivos

El listado de **opciones** al abrir un archivo es el siguiente:

- **ios::in**: abre el archivo para lectura de datos. Se puede obviar si se usa **ifstream**.
- **ios::out**: abre el archivo para escritura de datos, destruye los contenidos si ya existe. Se puede obviar si se usa **ofstream**.
- **ios::app**: abre el archivo para escritura de datos al final del archivo
- **ios::binary**: abre el archivo en modo binario

Se pueden mezclar una o más opciones con el operador *or* a nivel de bits (`|`)

Ejemplo:

```
file.open("archivo.txt", std::ios::out | std::ios::binary);
```

Lenguaje C++ - Archivos

La operación `close()` cierra el vínculo entre el stream y un archivo en disco. Al cerrarse un archivo de salida, cualquier operación pendiente se realiza. Permite reutilizar el stream para otro archivo.

```
int main(){
    fstream file;
    file.open("entrada.txt", ios::in);
    if( !file.is_open() ){
        cout << "Error al abrir el archivo\n";
        exit(1);
    }
    /* ... */
    file.close();
    return 0;
}
```

Lenguaje C++ - Archivos ASCII

Los archivos ASCII consisten de texto legible por un humano a través de un editor de texto. Se pueden usar los operadores de stream >> y << para leer y escribir respectivamente en un archivo.

```
int main(){
    ofstream fp; /*stream de archivo*/
    fp.open("salida.txt");
    if (!fp.is_open()){ /*...*/ }

    fp << "Hola mundo\n";
    fp << 2021;

    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos ASCII

Los archivos ASCII consisten de texto legible por un humano a través de un editor de texto. Se pueden usar los operadores de stream >> y << para leer y escribir respectivamente en un archivo.

```
int main(){
    ifstream fp; /*stream de archivo*/
    int n;
    fp.open("entrada.txt");
    if (!fp.is_open()){ /*...*/ }

    fp >> n; // lee un entero de un archivo
    /*...*/
    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos ASCII

Los archivos ASCII consisten de texto legible por un humano a través de un editor de texto. Se pueden usar los operadores de stream >> y << para leer y escribir respectivamente en un archivo.

```
int main(){
    ifstream fp; /*stream de archivo*/
    string s;
    fp.open("entrada.txt");
    if (!fp.is_open()){ /*...*/ }

    fp >> s; // lee un string hasta un espacio en blanco
    /*...*/
    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos ASCII

Los archivos ASCII consisten de texto legible por un humano a través de un editor de texto.
Para leer una línea de texto desde el archivo se puede utilizar la función **`getline()`**

```
int main(){
    ifstream fp; /*stream de archivo*/
    string s;
    fp.open("entrada.txt");
    if (!fp.is_open()){ /*...*/ }

    getline(fp, s); // lee una línea desde el archivo
    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos ASCII

Los archivos ASCII consisten de texto legible por un humano a través de un editor de texto.
Para leer un solo carácter desde el archivo se puede utilizar la operación **`get()`**

```
int main(){
    ifstream fp; /*stream de archivo*/
    char c;
    fp.open("entrada.txt");
    if (!fp.is_open()){ /*...*/ }

    fp.get(c); // lee una línea desde el archivo
    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos Binarios

Los archivos binarios permiten almacenar datos usando el formato de bits que tienen en memoria principal. Usualmente, no serán legibles para un humano.

La operación `write()` permite escribir datos en un archivo binario.

```
int main(){
    ofstream fp; /*stream de archivo*/
    int i = 5;
    fp.open("archivo.dat", ios::binary );

    fp.write((char*)&i, sizeof(int));

    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos Binarios

Los archivos binarios permiten almacenar datos usando el formato de bits que tienen en memoria principal. Usualmente, no serán legibles para un humano.

La operación `read()` permite leer datos desde un archivo binario.

```
int main(){
    ifstream fp; /*stream de archivo*/
    int i;
    fp.open("archivo.dat", ios::binary );

    fp.read((char*)&i, sizeof(int));

    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos Binarios

Con la operación `write()` se pueden escribir valores desde un arreglo a un archivo

```
int main(){
    ofstream fp; /*stream de archivo*/
    int a[10];
    for(int i=0;i<10;++i) a[i] = i;

    fp.open("archivo.dat", ios::binary);

    fp.write((char*)a, 10*sizeof(int));

    fp.close();
    return 0;
}
```

Lenguaje C++ - Archivos Binarios

De igual forma, con la operación `read()` se pueden leer valores a un arreglo de un archivo

```
int main(){
    ifstream fp; /*stream de archivo*/
    int a[10];
    fp.open("archivo.dat", ios::binary);

    fp.read((char*)a, 10*sizeof(int));
    for(int i=0;i<10;++i )
        cout << "El valor de a[" << i << "]=" << a[i] << endl;

    fp.close();
    return 0;
}
```

Lenguaje C++ - Argumentos de Programa

Hasta este punto se ha visto la función main() como una función sin parámetros. Esta función puede recibir los argumentos del programa como parámetros en su inicio. Para ello se debe definir como:

```
int main(int argc, char **argv){}
```

Estos parámetros corresponden a:

- `int argc`: un entero que indica el número de argumentos pasados al programa incluyendo el nombre del programa
- `char **argv`: un arreglo de punteros a strings, donde el primer string contiene el nombre del programa mientras que el resto contendrá cada uno de los argumentos pasados al programa

Lenguaje C++ - Argumentos de Programa

El siguiente código muestra por pantalla los contenidos de **`argv:`**

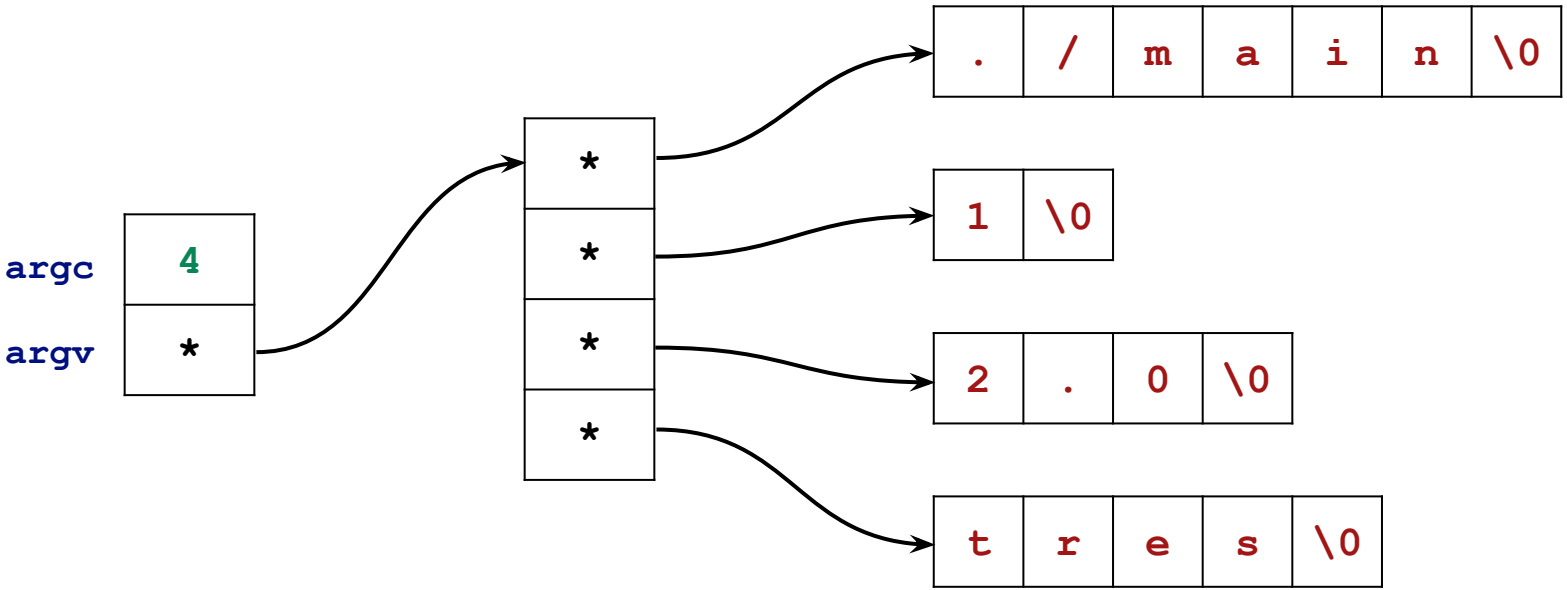
```
#include <iostream>
int main(int argc, char** argv){
    for(int i=0;i<argc;i++){
        std::cout << i << ": " <<  argv[i] << std::endl;
    }
    return 0;
}
```

Si este programa se ejecutará como **`./main 1 2.0 tres`** se mostraría por pantalla:

```
0: ./main
1: 1
2: 2.0
3: tres
```

Lenguaje C++ - Argumentos de Programa

Considerando la ejecución del programa como "`./main 1 2.0 tres`" la disposición de **`argc`** y **`argv`** se puede ver gráficamente como:



Lenguaje C++ - Extra

- Lenguaje C
- Definición básica de tipos: **`typedef`**
- Operadores a nivel de bits (**`<<`**, **`>>`**, **`&`**, **`|`**, **`^`**, **`~`**)
- Conversión de string a número
  - **`stoi()`**, **`stol()`**, **`stof()`** y **`stod()`**
  - **`<sstream>`** y **`stringstream`**
- Generación de números aleatorios
  - C: **`rand()`** y **`srand()`**
  - C++: **`<random>`**
- Herramientas
  - GNU Debugger (gdb)
  - Valgrind