Tema 1. Variables, tipos y gestión de la memoria

a. Variables y su representación en memoria
b. El operador dirección (&)
c. El operador de indirección (*)
d. Arrays. Aritmética de punteros

Variables en C++

```
int entero = 5;
```

- Existen tipos de datos fundamentales llamados **tipos primitivos** en C++ los cuales pueden usarse para asignar variables usando las palabras reservadas que las identifican.
- Caracteres conforman palabras reservadas e identificadores que en conjunto permiten asignar valores a variables declaradas.

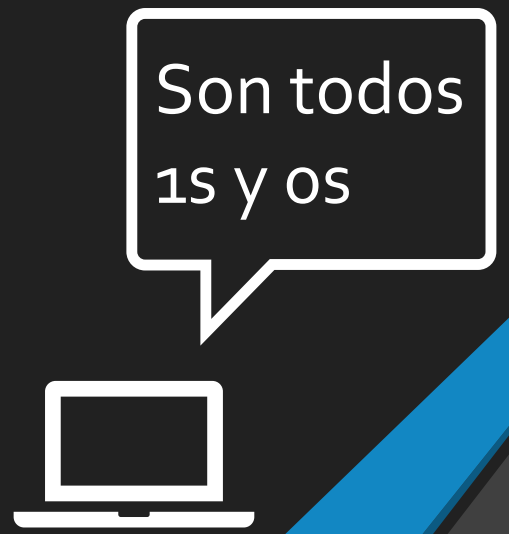
Variables en C++ según una persona

```
1   int entero = 5;
2   float flotante = 5.0f;
3   char caracter = '5';
4   bool estado = true;
```

- Las variables son representaciones abstractas de información con la que deseamos operar.

Variables en C++ según un ordenador

```
1   int entero = 5;
2   float flotante = 5.0f;
3   char caracter = '5';
4   bool estado = true;
```



Para un computador, una variable es un conjunto ordenado de datos binarios.

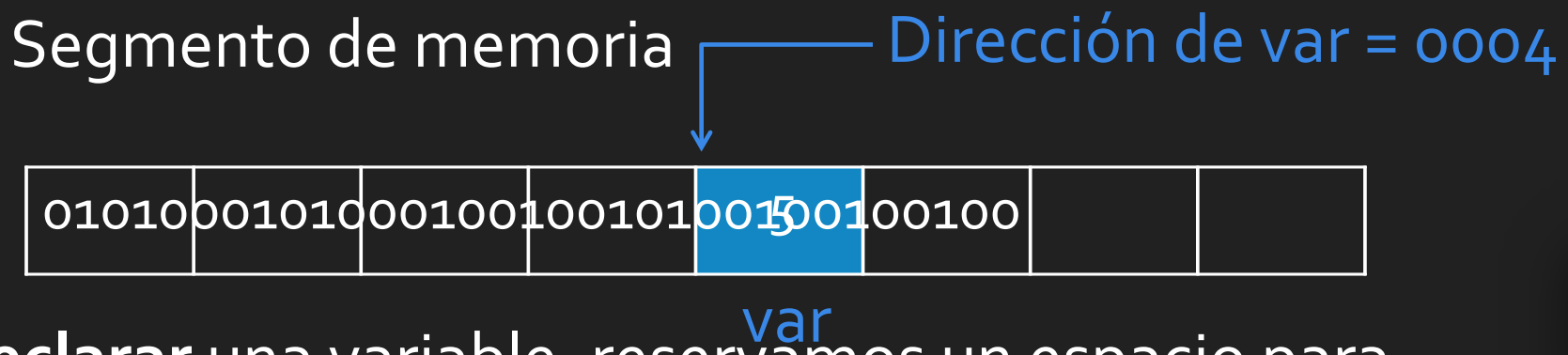
Variables en C++ según un ordenador

- Para un computador, una variable es un conjunto ordenado de datos binarios.
- Conjunto ordenado:

  … 0101000101000100100101001001001oo …

- La pregunta es, ¿dónde empieza y donde acaba una variable?
- Un computador utilizará una memoria para saber cómo están organizados estos conjuntos.
- Esta memoria tiene direcciones que indican donde empieza una variable y según el tipo de variable es posible saber donde acaba (debido al tamaño)

Dirección de una variable y el segmento de memoria



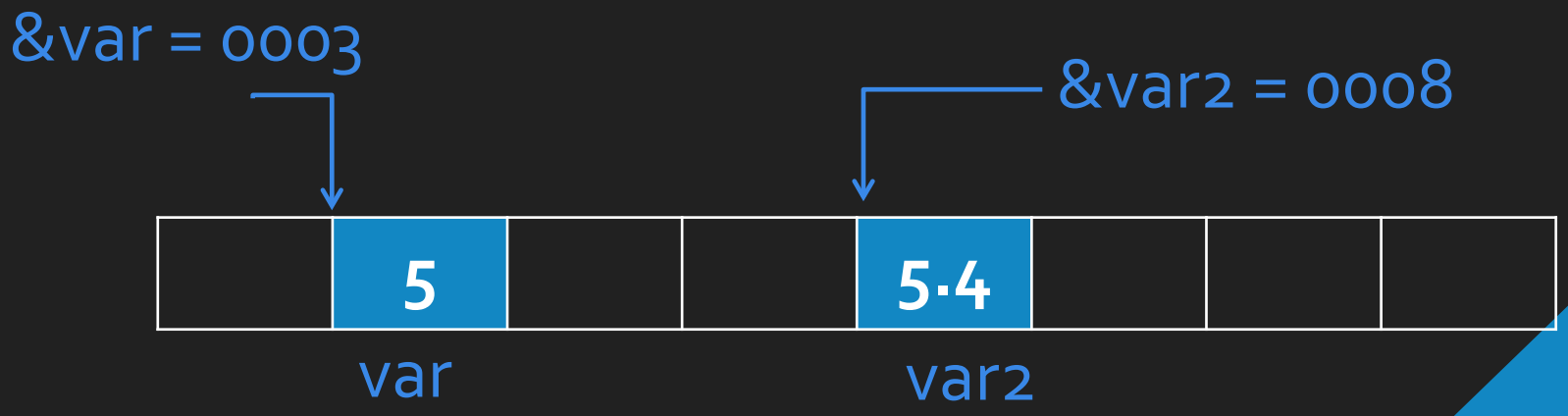
- Al **declarar** una variable, reservamos un espacio para almacenar el valor de dicha variable
- Entonces el nombre var se asigna como un sobrenombre o "apodo" de la dirección y ese espacio de memoria queda reservado.
- Variables declaradas sin inicializadores toman los valores anteriormente guardados en esa dirección

```
int var = 5;
```

```
char c;
```

Dirección de una variable

- Sabemos que cuando las variables se declaran, un espacio de memoria es reservado y cada vez que se desea leer o escribir en este espacio, se utiliza su dirección. Pero ¿Cuál es esta dirección?
- C++ introduce un operador el cual permite obtener dicha dirección: &




```
int var = 5;
float var2 = 5.4;
```

- Las direcciones se deciden en tiempo de ejecución y son aleatorias.

Dirección de una variable

- Veamos el siguiente ejemplo:

```
cout << "La variable var : " << var << " tiene una direccion igual a    : " << &var << endl;
```

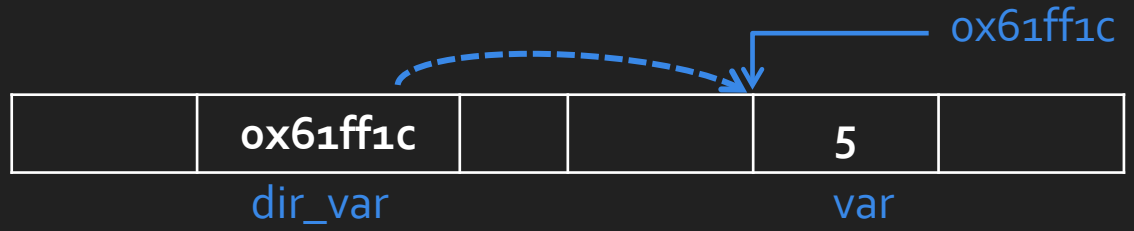
Imprime en pantalla:

"La variable var: 5 tiene direccion **0x2d5a6d53**"

- Como vemos, la dirección es un número **hexadecimal** que facilita observar su representación binaria (la verdadera dirección)
- La operación &i se lee como "dirección de memoria de i"
- ¡No hay ninguna asignación, ni tampoco una declaración!

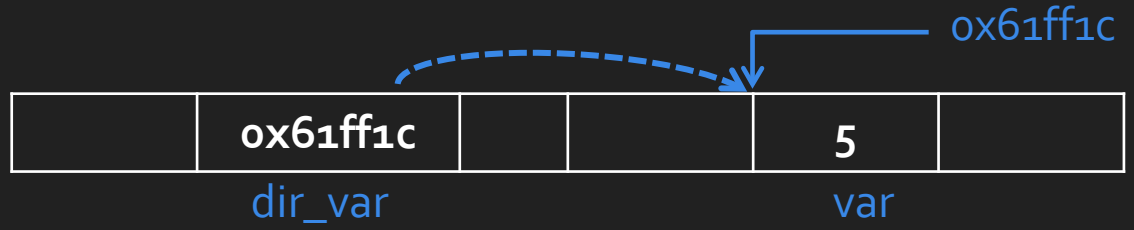
¿Se podrá guardar en una variable esta dirección?

Variable que apunta a la dirección de otra variable



- Las direcciones de memoria de las variables pueden almacenarse en unos tipos especiales de variables.
- Esto implica que otro espacio de memoria almacena la dirección (en bits) de una variable cualquiera.
- Este espacio de memoria es una variable a la que llamaremos de tipo **puntero**.

Variable que apunta a la dirección de otra variable



- Para declarar un puntero se utiliza el * justo delante del nombre de la variable durante la declaración.

```
int var;
int *dir_var;
var = 5;
dir_var = &var;
```

```
int var = 5;
int *dir_var;
dir_var = &var;
```

```
int var = 5;
int *dir_var = &var;
```

Punteros

- El puntero siempre *apunta* al inicio de la variable.
- El puntero solamente guarda o almacena la dirección de memoria, no el contenido de una variable.

```
cout << "dir_var (" << dir_var << ") es un puntero que guarda la direccion de var" << endl;
cout << "Es decir que dir_var es una nueva variable tipo puntero" << endl;
```

- Imprime en pantalla:
  "… (**0x2d5a6d53) es un puntero …**"
- Un puntero int * p_var puede almacenar solamente direcciones de variables tipo int.

Puntero.

- Apunta a un espacio físico en donde se guarda el valor de una variable de interés.
- Un puntero puede apuntar a cualquier tipo, siempre que su declaración sea compatible.
- Siempre apunta al comienzo de la variable.
- La declaración de una variable puntero siempre es
  tipo* nombre_puntero;
- El contenido de un puntero siempre es una dirección
  nombre_puntero = &nombre_var;
- La parte derecha se lee como dirección
  de la variable **nombre_var**.

Evaluando direcciones en punteros


```
char c1 = '!', c2, c3 = 'a';
char * p_c1, * p_c2; // punteros
```

- Durante una asignación de punteros, podemos:
  - Copiar direcciones de variables:
    ```
    p_c1 = &c1; // la dirección de c1 se almacena en p_c1
    ```
  - Copiar direcciones entre punteros:
    ```
    p_c2 = p_c1; // el valor de p_c1 se almacena en p_c2
    ```
- Debido a que los punteros p_c1 y p_c2 tienen los mismos valores almacenados, se puede decir que apuntan a la misma dirección.

Evaluando direcciones en punteros


```
char c1 = '!', c2, c3 = 'a';
char * p_c1, * p_c2; // punteros
```

- Por otro lado, y más importante, podemos utilizar el puntero para obtener el valor al que apunta una dirección mediante el operador de indirección *
  - En la expresión (parte derecha):
    `c2 = *p_c2;`
    Se lee como: el valor de la variable en la dirección almacenada en p_c2 se asigna a la variable c2
  - En la asignacion (parte izquierda):
    `*p_c1 = c3;`

El operador & permite localizar en memoria, mientras que el operador * permite obtener el valor

Operaciones con Punteros

Como utilizar → para asignar ↓	**int var1 (variable simple)**	**int* puntero1 (puntero)**
int var2 (variable simple)	Copia: var2 = var1	Indirección: var2 = *puntero1
int* puntero2 (puntero)	Dirección: puntero2 = &var1 Y, además: *puntero2 = var3	Copia: puntero2 = puntero1

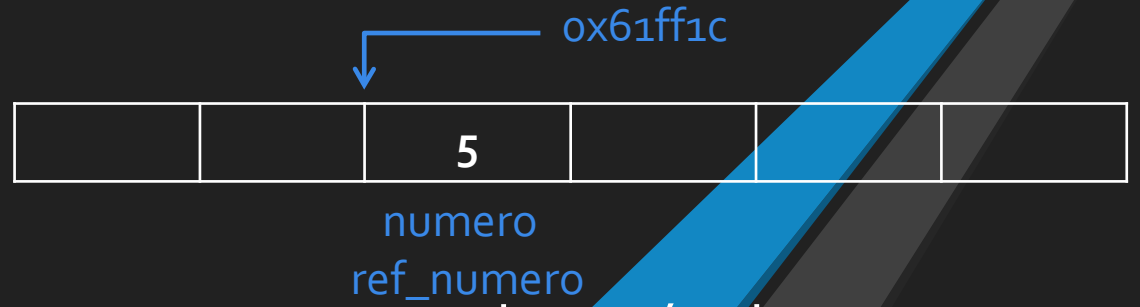
Problema de repaso

- Utilizando las variables flotantes **pi**, **dos_pi**, punteros a flotantes **punt1** y **punt2**:
  - Inicialice todas las variables
  - Modifique el valor de **pi** igualándolo a 3.1415.
  - Modifique el valor de **dos_pi** al doble de **pi** utilizando **punt1** y **punt2**.
  - Muestre en pantalla los valores almacenados en ambas direcciones
  - Restricciones: No está permitido asignar valores directamente a **dos_pi**, ni tampoco utilizar cout con **pi** y **dos_pi**.

Referencia a una variable

- Cuando creamos una variable local estamos creando aliases (sobrenombres) para una dirección de memoria.

```
int numero = 5; // variable
```



- En C++, es posible crear varios sobrenombres para una misma dirección de memoria.
- En este caso se debe utilizar & a la hora de declarar una variable.

```
int &ref_numero = numero; // referencia
```

Referencia a una variable

- Una referencia se utiliza exclusivamente para dar mas aliases al mismo segmento de memoria.
- Una referencia requiere por lo tanto un inicializador o una asignación inicial que indique al compilador de cuál variable se quiere crear un sobrenombre.

```
int &ref_numero = numero; // referencia
```

- La utilidad principal de las referencias se observará al aprender más sobre las 3 regiones de memoria que existen en los programas: Static-Stack-Heap

Representación

- Para una mejor comprensión de la programación, se ha decidido escribir los nombres de variables y funciones según un orden lógico.
- Desde esta clase en adelante, para referirnos a cualquier variable debemos utilizar el siguiente formato, respetando las minúsculas:
- <nombre_de_variable> : <tipo> = <valor inicial si corresponde>
  - ejemplo: int = 5
  - dato: string
  - pi: float = 3.1415
  - encendido: bool
  - var: short unsigned int = 3
  - nombre_completo: string = "Juan Pérez"
  - arreglo_de_dobles: double[5]
  - variables_siempre_en_minuscula: char = 'v'

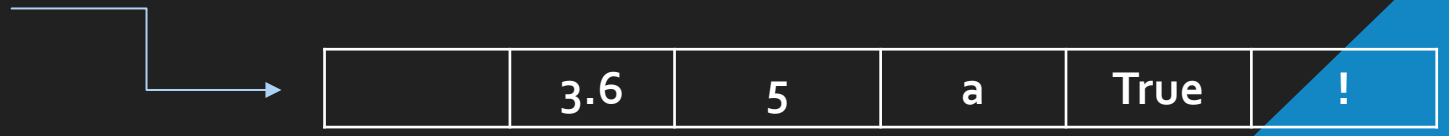
Representación

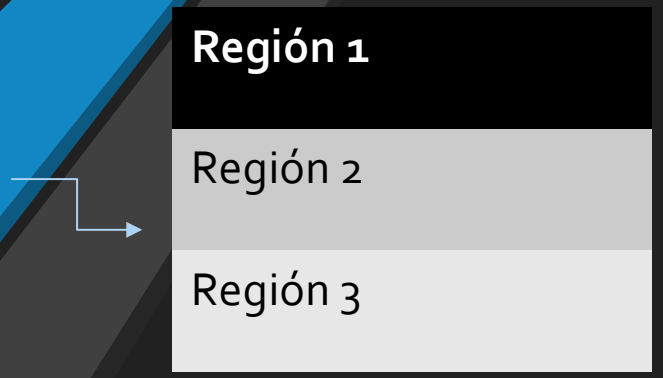
- Asimismo, las funciones se deben escribir de una forma similar:
- <nombreDeFuncion>(variables ....) : <tipo>
  - main( ): int
  - holaMundo( ): string
  - calculaAreaCuadrado(lado: float) : float
  - funcionTipoVoid( ): void
  - funcionQueRecibe2Variables(var1: int, var2: float): void
  - primeraLetraMinuscula(aqui_todo_en_minusculas: bool, ...): string
  - lasSiguientesPrimerasEnMayusculas(estas_son_variables: int[3]): double*

Memorias

- Todas las variables, todos los prototipos de funciones y las instrucciones se almacenan de forma ordenada dentro de la memoria.
- En programación, se han definido 3 regiones importantes que almacenan y se encargar de indicar al ejecutor cuál debe ser la siguiente instrucción por evaluar y donde se almacenarán los resultados.

..0101000101000100100101001000O..





Memorias Static-Stack-Heap

- **Memoria Estática (en inglés, Static)**: Región donde se almacenan los datos que están presentes desde el comienzo del programa hasta el final. Son variables que serán utilizadas en la mayoría de las ocasiones y prototipos de funciones.
- **Pila (en inglés, Stack)**: Área donde las variables aparecen y desaparecen en momentos puntuales de la ejecución de un programa. Se utiliza principalmente para almacenar variables que tienen ámbito reducido, sólo están disponibles mientras se está ejecutando una función en la que las variables hayan sido definidas.
- **Montón (en inglés, Heap):** Contiene memoria disponible para que se reserve, utilice y libere en cualquier momento durante la ejecución de un programa. Es una memoria "dinámica" para estructuras de datos que no se saben si se necesitan, e incluso tampoco se sabe el tamaño deseado hasta que el programa está ejecutando.

Memorias Static-Stack-Heap

- La memoria Static se inicializa al comienzo, es de tamaño fijo en donde se localizan datos que se utilizarán durante toda la ejecución.
- La memoria Stack es del tipo LIFO (Last-in First-out). Se apilan las variables e instrucciones según el programa.
- La memoria Heap no guarda en algún orden específico, se utiliza para variables que se deseen guardar y puedan persistir o eliminarse según la necesidad.

Memorias Static-Stack-Heap

- Las variables que no son parte de ninguna función específica serán siempre partes de la **static**.
- También serán las variables definidas como **static.**
- Los prototipos de funciones también forman parte de la **static**.

Memoria Static

- Al declarar cualquier función o variable fuera de la función main o al declarar explícitamente usando static:

```
1   int variable_static = 42;
2   int funcionCualquiera(){
3       return 42;
4       }
5   int main(){
6       static int variable_static_interna = 42;
7       ...
```

Memoria Static

- Si bien no es posible obtener directamente la dirección de una función, es fácil entender que, si una función puede ser invocada desde cualquier parte de programa, el prototipo de ésta debe estar alojada en la memoria estática.
- Al escribir, debemos indicar el espacio de memoria y sus funciones y variables.

Static

- variable_static: int = 5
- funcionStatic(): int
- main(): int
- variable_static_interna: static int

Utilizando la representación snake_case para variables y camelCase para funciones

Memoria Stack

- El **stack** se llena en el orden en que se han declarado variables.
- Cada llamada a una función crea un **Frame** que no es otra cosa que una dirección guardada que define el espacio en memoria al cual volver cuando finalice la función.
- Si una función se llama más de una vez, se asigna una numeración según el orden de llamada.
- Los argumentos de una función no se guardan en el stack, los parámetros de una función si y son guardados justo después de guardar el Frame de la función.

Memoria Stack (Apilamiento)

- Sigue el orden según se van declarando las variables, de arriba abajo.
- El apilado comienza en el main.
- Cualquier llamada a una función primero define la variable que guarda el resultado y a continuación se crean los Frames o Ámbitos.
- Dentro de cada Frame se vuelven a apilar las variables creadas.
- No importa si se repiten los nombres de variables ya que los ámbitos serán distintos.

Memoria Stack (Apilamiento)


```
double funcionSumaIntFloat(int a, float b){
return a + b; }

int main() {
    int var1 = 1;
    float var2 = 2.0;
    double resultado = fSumaIntFloat(var1, var2);
return 0;
}
```

Memoria Stack (Apilamiento)


```
double funcionSumaIntFloat(int a, float b){
return a + b; }

int main() {
    int var1 = 1;
    float var2 = 2.0;
    double resultado = fSumaIntFloat(var1, var2);
return 0;
}
```

Stack

- var: int = 1
- var2: float = 2.0
- resultado: double
- Frame de `fSumaIntFloat`
  - a: int = var1
  - b: int = var2

Memoria Stack (Apilamiento)

- Múltiples llamadas a funciones crean múltiples frames.

```
double resultado = fSumaIntFloat(var1, var2);
double resultado2 = fSumaIntFloat(var1, var2);
```

Stack

- var: int = 1
- var2: float = 2.0
- resultado: double
- Frame 1 de `fSumaIntFloat`
  - a: int = var1
  - b: int = var2
- Frame 2 de `fSumaIntFloat`
  - a: int = var1
  - b: int = var2

Memorias

```cpp
#include <iostream>
int number = 5;

int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result; }

int main() {
    int result;
    result = factorial(number);
    cout << number << “! = " << result << endl;
    return 0; }
```

Static

- number: int = 5
- factorial(): int
- main(): int

Stack

- result: int
- Frame de factorial
  - n: int = number
  - result: int = 1
  - i: int = 2

Heap

Memorias

```cpp
#include <iostream>

int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result; }

int main() {
    int number = 5;
    int result = factorial(number);
    cout << number << “! = " << result << endl;
    return 0; }
```

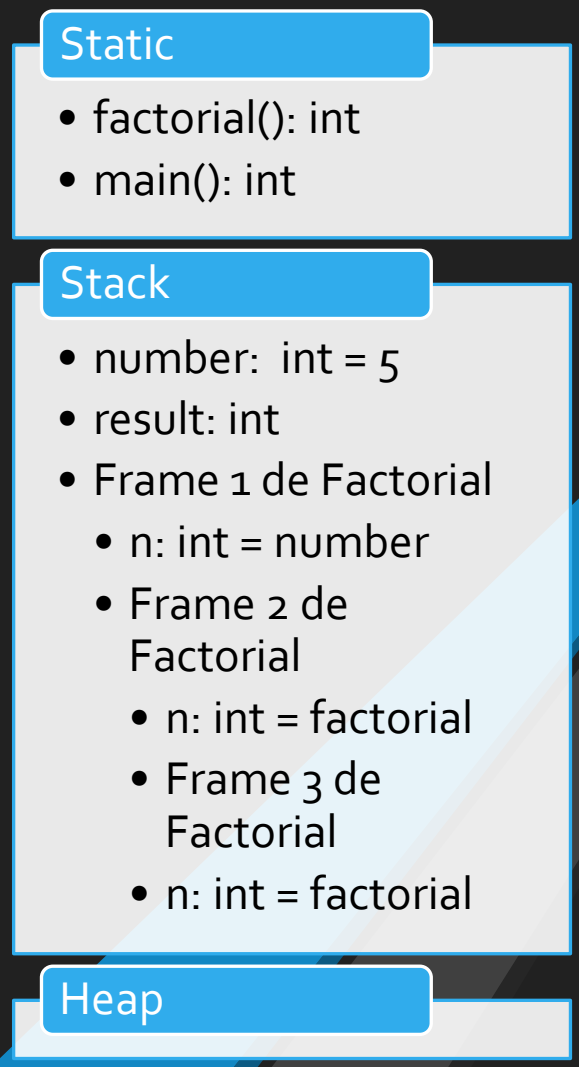
```cpp
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Diferentes implementaciones necesitan diferentes variables.

Cada implementación distinta puede utilizar regiones distintas de memoria.

Memorias

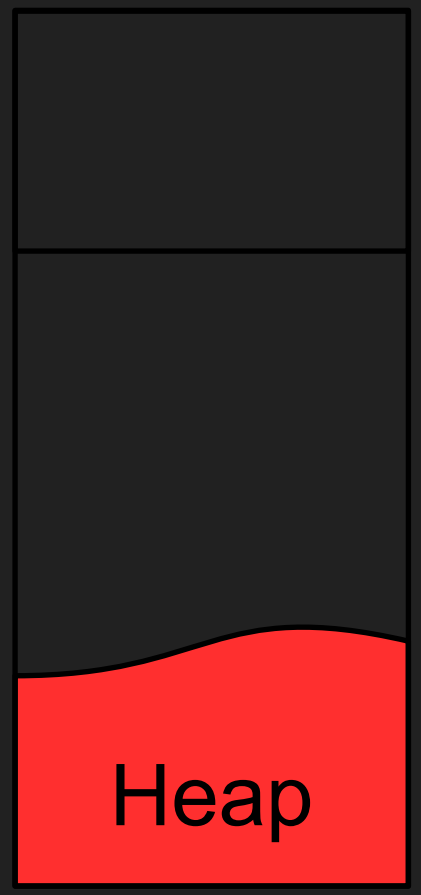
```cpp
#include <iostream>

int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number = 5;

    int result = factorial(number);

    cout << number << "! = " << result << endl;

    return 0; }
```



Memoria Heap

- La memoria **Heap** se utiliza de forma manual en cualquier parte del programa, mediante el uso de la función malloc(nro_de_bytes) o mediante el uso de la palabra reservada new.
- Las variables se almacenan como direcciones de memoria, pero en lugar de la dirección explícita, se guarda el nombre de la variable original.

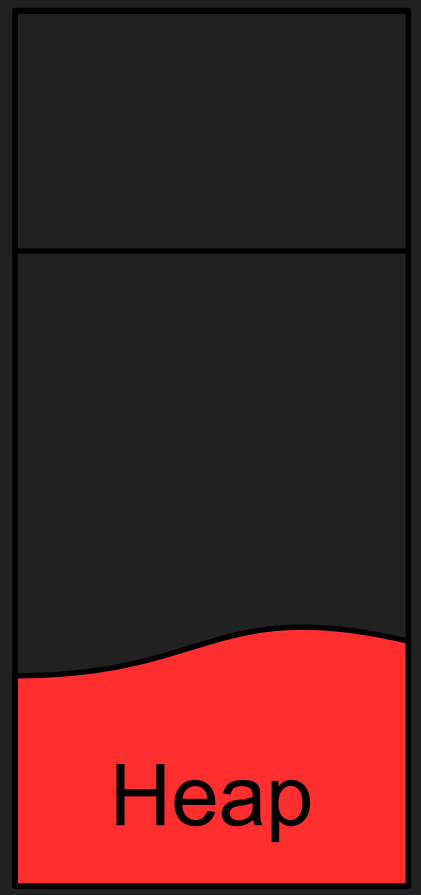
Memoria Heap (Montón)



- La memoria heap funciona como la estática, pero la declaración debe ser explícita: usando malloc() o `new`.
- En esta asignatura, solamente utilizaremos `new`.
- La palabra reservada `new` **devuelve siempre un puntero**: pero la variable siempre se aloja en la heap.
- Por lo que para utilizar la variable se deben utilizar operadores de dirección e indirección:

```
int* var_en_heap = new int; // crear en heap
*var_en_heap = 1; // asignar un valor
cout << &var_en_heap; // direccion de la variable
```

Memoria Heap (Montón) . ¡IMPORTANTE!



- Al utilizar `new` para crear variables en la memoria heap siempre hay que eliminar la variable creada.
- La eliminación se realiza con la palabra reservada `delete`
- Es importante siempre realizar este proceso de la forma más ordenada posible: Siempre que se escriba new, inmediatamente debajo se debe eliminar dicha variable y agregar instrucciones o líneas entre estas dos instrucciones

```
int* var_en_heap = new int;

   ... Instrucciones que usan var_en_heap

delete var_en_heap; // liberar memoria
```

Memorias Static-Stack-Heap

- Se puede observar que las variables y funciones están en sitios diferentes de la memoria al observar sus direcciones, así en:

```
static int var1;
int var3;
int* var5 = new int;
static float var2;
float var4;
float* var6 = new float;
```

- Hay variables que están más cerca de otras, ¿cuáles?

Parámetros y Argumentos de Funciones

- Argumento != Parámetro

```
1   int factorial_pv(int num){
2       int resultado = num;
3       for (int i = 1; i < num; ++i){
4           resultado *= i;
5       }
6       return resultado;
7   }
8
9   int main(){
10      int numero = 5;
11      cout << "Factorial de "<< numero;
12      int resultado = factorial_pv(numero);
13      cout << " es " << resultado << endl;
```

Entonces, ¿en qué región de la memoria se aloja?

- `num` es el parámetro de entrada de esta función.

  Es declarado en la definición de la función y se utiliza localmente: si la función factorial( ) no se llama o ejecuta, int `num` no existe.

- `numero` contiene un valor que es proporcionado como argumento de entrada de la función.

  `numero` existe dentro de la función `main()`

Parámetros y Argumentos de Funciones

```
1   int factorial_pv(int num){
2       int resultado = num;
3       for (int i = 1; i < num; ++i){
4           resultado *= i;
5       }
6       return resultado;
7   }
8
9   int main(){
10      int numero = 5;
11      cout << "Factorial de "<< numero;
12      int resultado = factorial_pv(numero);
13      cout << " es " << resultado << endl;
```

- El proceso principal comienza en la línea 12 obteniendo el valor en numero y **copia** su **contenido** al espacio correspondiente al primer y único parámetro de la función: **`num: int`**, la variable que toma el valor copiado (5).
- Con esto se ejecuta la función factorial.
- La variable "`resultado`" dentro de la función toma distintos valores antes de volver al main.

El mensaje ent
funciones es el v
no la variab

Pass-by-value

```
1   int factorial_pv(int num){
2       int resultado = num;
3       for (int i = 1; i < num; ++i){
4           resultado *= i;
5       }
6       return resultado;
7   }
8
9   int main(){
10      int numero = 5;
11      cout << "Factorial de "<< numero;
12      int resultado = factorial_pv(numero);
13      cout << " es " << resultado << endl;
```

- Si observamos las direcciones de numero y num podemos observar que son diferentes.
- Este proceso se denomina ***pass-by-value***.
- Las ventajas incluyen:
  - No se modifica el **argumento** de entrada.
  - Los datos tienen mejor seguridad.
- La desventaja:
  - Se crean copias de los valores:
    - Más memoria utilizada
    - Más tiempo de ejecución

Pass-by-reference

- En lugar de hablar (y operar) sobre los valores y copias de ellos, podríamos referimos a las mismas variables ya definidas y no crear copias, ya que así podríamos tener ventajas de memoria y tiempo de ejecución.
- Al inicializar y asignar `int numero = 5;` sabemos que existe un espacio de memoria asignado a la variable `numero` del tipo `int`
- En C++,podemoa hacer uso del operador `&`(ampersand) para crear referencias, así `int &num` es una referencia, no una variable
- Al utilizar en funciones, esto se denomina *pass-by-reference*.

Pass-by-reference

```
1   void factorial_pr(int& num){
2       int limite = num;
3       for (int i = 1; i < limite; ++i){
4           num *= i;
5       }
6   }
7
8   int main()
9   {
10      int numero = 5;
11      cout << "Factorial de "<< numero;
12      factorial_pr(numero);
13      cout << " es " << numero << endl;
```

- Para utilizar la referencia a la variable en lugar de el dato en la variable, se debe cambiar el **parámetro** de la función. Es decir que el parámetro debe convertirse en una referencia en lugar de una variable nueva.
- El **argumento** no cambia.
- Se pueden corroborar que las direcciones de numero y num son las mismas.

Pass-by-reference

```
1   void factorial_pr(int& num){
2       int limite = num;
3       for (int i = 1; i < limite; ++i){
4           num *= i;
5       }
6   }
7
8   int main()
9   {
10      int numero = 5;
11      cout << "Factorial de "<< numero;
12      factorial_pr(numero);
13      cout << " es " << numero << endl;
```

- **Importante**: El entero numero con el que se opera ya existe en *main*. Pero la referencia &num se crea en el momento de ejecutar la función.
- La función ya no necesita devolver ningún valor ya que el valor importante en el main ya se está modificando dentro de otra función

El mensaje entre funciones es la variable

Pass-by-pointer

- Por último, el parámetro de la función también puede ser un puntero, evitando así pasar variables y cambiando sus aliases.
- En este caso, el argumento puede ser o bien el valor de un puntero o una dirección de una variable (que son definiciones equivalentes).
- Recordemos que un puntero almacena direcciones de memoria y se declaran de la siguiente forma:

```
int *p_numero;
```

Pass-by-pointer

```
1   void factorial_pr(int& num){
2       int limite = num;
3       for (int i = 1; i < limite; ++i){
4           num *= i;
5       }
6   }
7
8   int main()
9   {
10      int numero = 5;
11      cout << "Factorial de "<< numero;
12      factorial_pr(numero);
13      cout << " es " << numero << endl;
```

- **Importante**: El entero numero con el que se opera ya existe en *main*. Pero la referencia &num se crea en el momento de ejecutar la función.
- La función ya no necesita devolver ningún valor ya que el valor importante en el main ya se está modificando dentro de otra función

El mensaje entre funciones es la variable

Pass-by-pointer

- La función debe declarar un nuevo puntero que almacene direcciones.
- Luego, al utilizarlo, podemos utilizar *punt para recuperar el valor, si se utiliza en la parte derecha de una asignación o bien asignar un valor en dicha dirección, si se utiliza en la parte izquierda.

```
1  void factorial_puntero(int * punt_a_num){
2      int limite = *punt_a_num;
3      for (int i = 1; i < limite; ++i){
4          *punt_a_num *= i;
5      }
6  }
```

Pass-by-pointer

- En cuanto al uso de la función, podemos considerar utilizar los argumentos como simples direcciones de memoria (&variable) o copiar los valores almacenados dentro de punteros (puntero)

```
1   int numero = 5;
2   cout << "Factorial de "<< numero;
3   factorial_puntero(&numero);
4   cout << " es " << numero << endl;
```

```
1   int numero2 = 5;
2   cout << "Factorial de "<< numero2;
3   int *p_numero = &numero2;
4   factorial_puntero(p_numero);
5   cout << " es " << *p_numero << endl;
```

- En definitiva, el mensaje será siempre una dirección de memoria

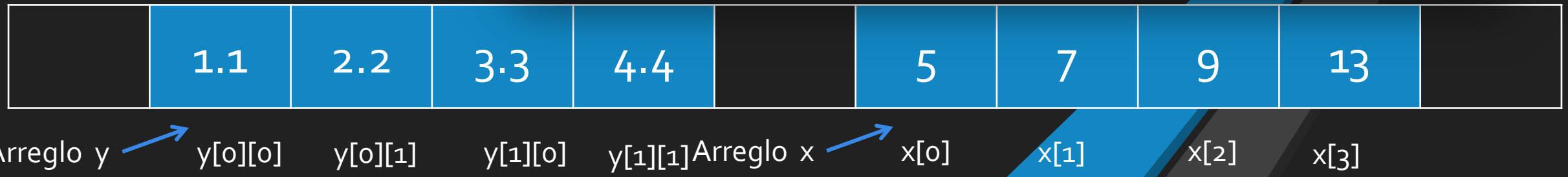
Memorias Static-Stack-Heap

- La memoria Static almacena todo lo que sea de uso global o que sea *estático*
- La memoria Stack almacena las variables en el orden que aparecen y luego van desapareciendo en el mismo orden. Se consideran como variables locales.
- La memoria Heap no guarda en algún orden especifico, se utiliza para almacenar de forma preventiva varios espacios de memoria.

Aritmética de punteros. Arreglos de Variables

- Al definir un arreglo de cualquier tipo, C++ reserva automáticamente cada elemento del arreglo de forma contigua. No importa si el arreglo es multidimensional.
- Entonces al definir:

```
1   int x[] = {5, 7, 9, 13};
2   double y[][2] = {{1.1, 2.2}, {3.3, 4.4}};
```



- **Esto no significa que las variables x e y se almacenen una después de la otra.**

Aritmética de punteros. Variables

- Esto permite definir y utilizar una aritmética de punteros:

- Por ejemplo, la suma de un puntero con un entero:

```
1   int *p_x = &x[0];
2   cout << "El primer elemento es " << *p_x << endl;
3   cout << "El segundo elemento es " << *(p_x + 1) << endl;
4       cout << "El elemento " << i << " es " << *(p_x + i) << endl;
```

Aritmética de punteros. Arreglos de Variables

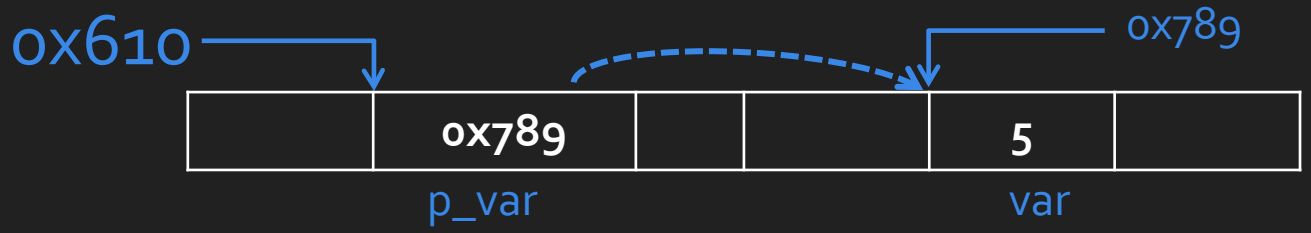
- Toda suma aritmética entre un puntero y un entero está definida, el resto de las operaciones no son útiles o no están definidas.

```
p_x + i
```

- Con esto, se definen las operaciones útiles más útiles como:
  - Apuntar al siguiente elemento de un vector: p_x++;
  - Apuntar al elemento anterior de un vector: p_x--;
  - Obtener las direcciones pares de un vector: (p_x + 2*i);
  - Obtener los elementos pares de un vector: *(p_x + 2*i);

Puntero a ...

- Toda variable declarada siempre tiene una dirección de memoria, por lo que siempre es posible crear un puntero de cualquier tipo de variable:
  - float,double, int, ... --> float*, double*, int*, ...
- Pero una variable tipo puntero también es otra variable, por lo que también tiene una dirección de memoria.



- ¿Como guardamos esta dirección de memoria?

Puntero a puntero

- Si para crear punteros a cualquier tipo se utiliza **tipo ***, para crear una variable que apunte a un puntero de cualquier tipo (un puntero a un puntero) basta con utilizar el operador dos veces el operador de indirección **tipo ****

```
int var, * punt1, **punt2;
var = 10; // variable del tipo entero
punt1 = &var; // puntero al entero var
punt2 = &punt1; // puntero al puntero punt1
```

Puntero a puntero

- El ejemplo más práctico del uso de puntero a punteros es el almacenaje de múltiples arreglos de datos de diferentes dimensiones:

```
1   int arreglo1[6] = {0, 1, 2, 3, 4, 5};
2   int arreglo2[3] = {6, 7, 8};
3   int arreglo3[4] = {9, 10, 11, 12};
4   int *arreglos[3] = {&arreglo1[0], &arreglo2[0], &arreglo3[0]};
5   int **punt_a_arreglos = &arreglos[0];
```

```
**punt_a_arreglos
```

```
**(punt_a_arreglos+i)
```

Indirección de arreglos

```
*(*(punt_a_arreglos)+j)
```

Indirección de elementos

Puntero a puntero

- El ejemplo más práctico del uso de puntero a punteros es el almacenaje de múltiples arreglos de datos de diferentes dimensiones:

```
1   int arreglo1[6] = {0, 1, 2, 3, 4, 5};
2   int arreglo2[3] = {6, 7, 8};
3   int arreglo3[4] = {9, 10, 11, 12};
4   int *arreglos[3] = {&arreglo1[0], &arreglo2[0], &arreglo3[0]};
5   int **punt_a_arreglos = &arreglos[0];
```

Mediante esta formulación es sencillo acceder a cada elemento de cada arreglo que se almacena mediante un puntero a puntero

```
*(*(punt_a_arreglos+i)+j)
```

Problema de repaso

- Crear una función que sume 2 “vectores” y guarde el resultado en un tercer “vector”. La función debe tener los siguientes parámetros: 3 punteros a double y un entero. Los argumentos corresponden a las direcciones de los primeros elementos de los dos arreglos a sumar, el arreglo en donde se debe almacenar el resultado y la dimensión de los arreglos.
- Repita el ejercicio con otra función que reciba un puntero a puntero y la dimensión de los arreglos.