PUNTEROS

CADA APARTADO PRINCIPAL CORRESPONDE CON EL NOMBRE DEL PROYECTO CON EJEMPLOS DE LO APUNTADO AQUÍ

1. INTRODUCCIÓN A LOS PUNTEROS

- Los punteros son una de los temas importantes en C++, te permiten hacer ciertas cosas de forma muy conveniente.
- Hasta ahora usábamos variables, variables que tienen una dirección en memoria, algo de lo que no habíamos hablado mucho hasta ahora. Todas las variables tienen una dirección de memoria, y dichas direcciones podemos almacenarlas, lo que en esencia es un puntero.
- Por lo tanto, **un puntero es un tipo especial de variable que almacena las direcciones de memoria de otras variables.**
- Supongamos que tenemos una variable de tipo `int` llamada 'var' que almacena el valor 22:
  `int` var = 22
  Como antes decía, todas las variables tienen una dirección de memoria, digamos que en este caso la variable 'var' se almacena en la dirección de memoria: 1008
  Pues nuestro puntero (una variable especial) que en este caso apunta a una variable de tipo `int`, tendrá como valor 1008

2. DECLARACIÓN Y USO DE PUNTEROS

- Para denotar una variable como puntero, se le añade un * tras el tipo de variable, por lo que siguiendo el ejemplo del apartado anterior:
  Variable 'var' → `int` x = 22
  Dirección de memoria de 'var' → 1008
  Puntero → `int *` p_var = 1008

- No importa si la posición del * está pegado al tipo de dato, al nombre de la variable o entre ambos.
  `int*` p_var = 1008
  `int *` p_var = 1008
  `int  *`p_var = 1008
  Todas las opciones son válidas.
  Es bueno declarar cada puntero y cada variable en general en una línea distinto para evitar dar lugar a confusiones.
- Puedes hacer punteros a variables de cualquier tipo de dato, no solo `int`, incluso tipos personalizados, que más adelante veremos en el curso.
- **Para que el puntero pueda almacenar las direcciones de las variables tanto el puntero como la variable deben tener el mismo tipo de dato.** En caso contrario la compilación del programa fallará.
- Para indicar que nuestros punteros no están apuntando nada al declararlos se usan `{}`, también puedes indicarlo de forma explícita añadiendo `nullptr` dentro de `{}` pero no podrás emplear los valores de las direcciones. **No hay que usar punteros que contengan `nullptr`.**
  Es necesario inicializar los punteros de alguna forma para usarlos (ya sea con `{}` o con `nullptr`), en caso contrario pueden darse errores graves. También hay que asegurarse que estás modificando direcciones válidas en dichos punteros. **Si no sabes lo que hay en un puntero, no lo uses.**
- **Los punteros y las variables ocuparán lo mismo en bytes al tener el mismo tipo de dato, sin embargo los punteros y las direcciones de memoria, que son las que se almacenan en el puntero no.**
- Para asignar una dirección de memoria a un puntero se usa & delante del nombre de la variable de la cual queremos guardar su dirección:
  `int` var = 45;
  `int *`p_var = &`var`;
  También puedes modificar más tarde el valor del puntero con otra dirección de memoria.
- Al mostrar por pantalla un puntero, verás la dirección de memoria de la variable.

- Los punteros también nos permiten "**desreferenciarlos**", es decir, **la capacidad de acceder al valor de la variable alojada en la dirección de memoria almacenada en el puntero.**

3. PUNTEROS DE TIPO CHAR

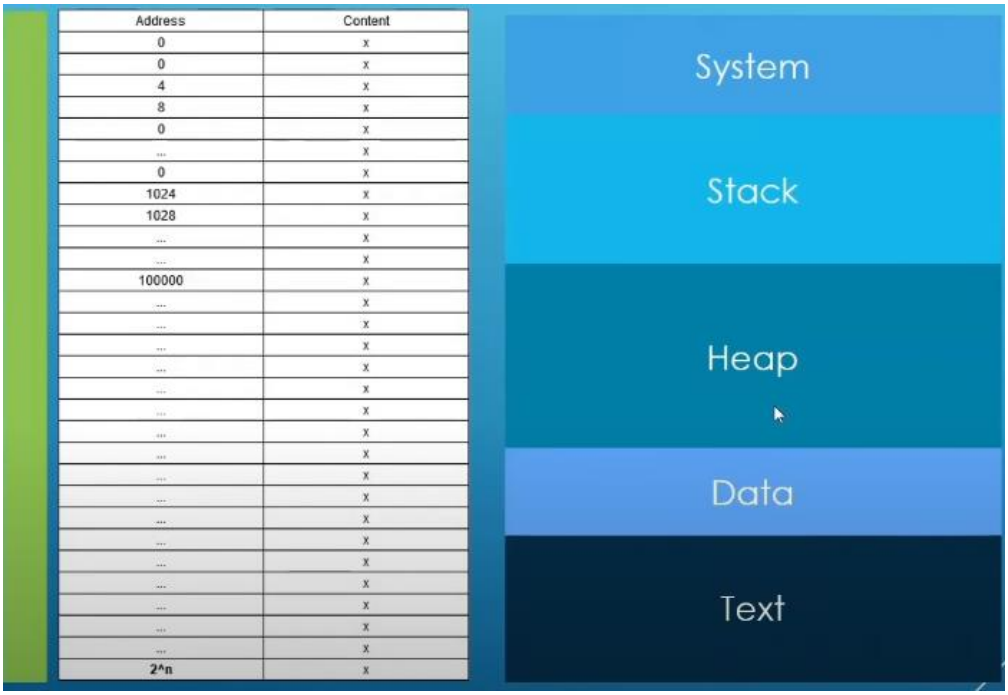
- La forma de declarar e inicializar un puntero que apunta a un `char` es la misma que para declarar un puntero que apunta a un `integer`.
- La diferencia entre los punteros es que un puntero de `char` puedes inicializarlo directamente con un `string` literal (String de C). Ya que se creará un `array` de `char` y el puntero apuntará al primer elemento del `array` creado (el primer carácter, es decir la primera letra del `string`).
  Esto nos otorga la flexibilidad de poder tratar `strings` como punteros de `chars`.
- Algunos compiladores se pueden negar a compilar este tipo de código (el compilador de Visual Studio por ejemplo) ya que el `array` de `char` creado a partir del `string`, es un `array` de tipo `const char`, es decir un array de `char` **CONSTANTES** y que impiden su modificación. Por lo que algunos compiladores te forzarán a usar punteros de tipo `const char`.
- A pesar de que el puntero de tipo char solo apunta al primer elemento creado a partir del `string` (la primera letra), al desreferenciar el puntero para imprimir el valor por pantalla, se mostrará el `string` al completo. Es una de las cualidades especiales que diferencia a los punteros de tipo `char` con respecto a punteros de otros tipos de dato.
- Si queremos modificar el `string` mediante punteros, es más conveniente crear directamente un `array` de `char` y modificarlo, ya que como se indica previamente, los `strings` se transforman en `arrays` de tipo `const char` que **NO** se pueden modificar.
  Sin embargo el principal uso de los punteros de tipo `char` es almacenar `strings` e imprimirlos por pantalla, no modificarlos por lo que nos seguirán siendo útiles. Aun así, si quieres modificar los `strings` crea un `array` de `char` directamente.

4. MAPA DE MEMORIA DE UN PROGRAMA

- Ya sabemos cómo funciona el flujo de desarrollo en C++, tenemos nuestro código de C++, que se compila mediante el compilador (valga la redundancia), el cual genera un binario ejecutable, que representa nuestro código de forma binaria. Y que al abrirlo, este, se carga en una sección especial de la memoria RAM (la encargada de ejecutar continuamente los procesos activos del PC) llamada "**Área de Programa**".
  Hasta ahora, pensábamos que el programa se cargaba en memoria real en nuestro PC, pero no es así, el programa piensa que sí, pero no es real. A continuación vamos a profundizar en ello, pero antes vamos a explicar porque esto es así.
  En nuestro PC hay muchos procesos ejecutándose constantemente, cientos, incluso miles de programas que se están ejecutando en segundo plano y de los cuales ni siquiera somos conscientes, por lo que si todos los procesos utilizasen memoria real de nuestro PC, nos quedaríamos sin RAM muy rápido o directamente nunca tendríamos suficiente. Para dar solución a esto, surgió la idea de lo que llamamos "**Memoria Virtual**", un truco que engaña al programa y le hace pensar que él es el único proceso ejecutándose en nuestro sistema operativo y que él es el dueño de toda la memoria de nuestro ordenador. Y lo que ve nuestro proceso de esta memoria es el llamado "**Mapa de Memoria**". Cada proceso tiene acceso a un rango de memoria entre 0 y $(2^{n}) - 1$, donde ***n*** es el número de bits de nuestro sistema operativo. El 1 se resta ya que la primera dirección de memoria es 0 y no 1.
  En mi caso al estar en un sistema operativo de 64 bits, el rango de mi memoria virtual va a estar entre 0 y $(2^{64}) - 1$, muchísima memoria, por lo que nuestro proceso va a pensar que al ser el dueño de dicha memoria nunca se va a acabar y va a poder hacer lo que quiera.
  Ya conocíamos la memoria real (RAM) y ahora hemos conocido la memoria virtual (entre 0 y $(2^{n}) - 1$), así que vamos a actualizar la idea que teníamos sobre el modelo de ejecución de programas ahora que sabemos más.
- Como decíamos, tenemos nuestro código de C++, que al compilarlo genera un binario, que al abrirlo carga nuestro proceso en el Área

de Programa de la RAM. Es entonces cuando comienza la ejecución del programa, nuestro proceso pasa por una sección de la CPU, llamada "**Unidad de Gestión de Memoria"** (MMU, Memory Management Unit), que se encargará de convertir la representación que tenemos en el Mapa de Memoria en memoria RAM real, sin embargo la CPU y la MMU solo cargan en la RAM las partes del programa que van a ser ejecutadas y el resto de programa se almacena en el disco duro, haciendo así un uso eficaz de la memoria real que como hemos dicho es un recurso valioso y limitado. Todo este proceso es mucho más complejo, pero sirve como idea general para saber cómo funciona todo por dentro. Supongamos que tenemos varios programas cargados en memoria cada uno con su propio Mapa de Memoria, se ejecutan y entran a la MMU, que se encargará de asignar a los procesos secciones en la memoria real de nuestro PC (RAM). Dichas secciones serán usadas por los programas.

- El Mapa de Memoria es un formato estándar definido por el sistema operativo, es por ello por lo que por ejemplo no puedes coger un ejecutable creado en Windows y ejecutarlo en Linux.
- Esta es la apariencia del Mapa de Memoria:



Está conformado por varias secciones como se puede apreciar en la imagen y cada una tiene su función pero las que nos importan por el momento son "**Stack**" y "**Heap**":

- **Stack**: Sección que almacena nuestras variables locales de nuestro programa, la llamada de funciones…
- **Heap**: Sección donde podemos conseguir memoria adicional si nos quedamos sin la memoria de la sección de Stack y que podemos consultar en tiempo de ejecución para facilitarle las cosas al programa.
- **Text**: Sección que carga el binario de nuestro programa para que la CPU pueda ejecutarlo. No vamos a hablar mucho de esta sección ya que se sale fuera del alcance del curso, es algo avanzado.

5. ASIGNACIÓN DINÁMICA DE MEMORIA (DYNAMIC MEMORY ALLOCATION)

- La asignación dinámica de memoria es una técnica empleada para poder usar memoria de la sección **Heap** que tenemos en el Mapa de Memoria de nuestro programa de C++. Como decía en el apartado anterior, esta memoria adicional de Heap, puedes ser usada si la memoria de **Stack** no es suficiente para la tarea que queremos llevar a cabo.
- Hasta ahora usábamos memoria que se encontraba en la sección de Stack, usábamos variables declaradas en el *main* o en otras funciones, ahora vamos a ver cómo podemos usar la memoria de la sección Heap.
- Las diferencias entre la memoria de Stack y la memoria de Heap son las siguientes:

Stack	**Heap**
La memoria es **finita**	La memoria es **finita**
El desarrollador **NO** tiene control total del tiempo de vida de la memoria	El desarrollador **SÍ** tiene control total del tiempo de vida de la memoria
El tiempo de vida de la memoria es **controlado por el ámbito** (*scope*) **en el que se encuentre**	El tiempo de vida de la memoria es **controlado explícitamente mediante las palabras reservadas** `new` y `delete`

- Variable alojada en **Stack** VS. Variable alojada en **Heap**:

```
1   #include <iostream>
2   int main()
3   {
4       {
5           int varAlojadaEnStack = 33;
6           int *pVarAlojadaEnHeap = nullptr;
7           pVarAlojadaEnHeap = new int;
8       }
9
10      return 0;
11  }
```

  En el ejemplo de arriba, `varAlojadaEnStack` no se podrá usar fuera de las líneas 5 y 7, ya que su tiempo de vida está delimitado por el ámbito en el que se encuentra, en este caso los `{}` de las líneas 4 y 8. Sin embargo, `*pVarAlojadaEnHeap` al ser un puntero que vive en Heap, y ser nosotros los dueños de la memoria, vamos a poder seguir usándolo fuera de los `{}`, ya que somos nosotros los que controlamos explícitamente cuando muere la porción de memoria empleada y es devuelta al sistema.
- Para usar la memoria alojada en la sección Heap, en primer lugar se crea un puntero (que va a apuntar a la memoria alojada en Heap) y se inicializa como `null` y a continuación se asigna al puntero la palabra reservada `new`, seguida del tipo de dato del puntero que hemos creado:

  ```
  int *pNumero1 = nullptr;
  pNumero1 = new int;
  ```

  Los punteros también se pueden inicializar directamente con memoria dinámica al declararlos:

  ```
  int *pNumero2 = new int;
  ```

  Una vez ejecutada la sentencia, el sistema operativo va a proceder a alojar memoria suficiente para nuestra variable en la sección Heap y esta parte de memoria pasará a ser de nuestra propiedad de tal manera que ningún otro programa del sistema va a poder usar esta porción de memoria. La memoria va a ser nuestra y podremos hacer lo que queramos con ella hasta que explícitamente

indiquemos lo contrario, es entonces cuando esta porción de memoria será devuelta al sistema.
Para devolver la memoria al sistema se usa la palabra reservada `delete` seguida del nombre de nuestro puntero. Una vez hecho esto, es importante "reiniciar" el puntero a `null` asignándolo a `nullptr` para indicar que ya no está apuntando a nada que sea válido. Ya que si intentas usar una sección de memoria que ha sido eliminada, el programa *crasheará*, además estarás intentando usar memoria que no ha sido inicializada. Es muy importante **NO** usar `delete` más de una vez en un mismo puntero, ya que el programa *crasheará.*

- **Es importante recordar que mientras usemos la memoria Heap lo que hagamos se almacenará en Heap y NO en Stack.**

6. PUNTEROS COLGANTES (DANGLING POINTERS)

- Son punteros que no apuntan a una dirección de memoria válida, intentar desreferenciarlos y usarlos resultará en un comportamiento no definido, que por lo general hará que tu programa *crashee* aunque puede ir a peor.
- Hay 3 tipos de referencias colgantes:
  - Cuando un puntero no ha sido inicializado.
  - Cuando un puntero ha sido eliminado.
  - Cuando varios punteros apuntan a la misma dirección de memoria.
- Para evitar punteros colgantes, hay que:
  - Inicializar los punteros tras declararlos.
  - Reiniciar los punteros a `nullptr` tras liberar la memoria.
  - Elegir un puntero y hacerlo el *master* (maestro) que controle la memoria. El resto de punteros serán los *slaves* (esclavo) por lo que no podrán liberar la memoria, pero si leerla y trabajar con ella.

7. CUANDO 'NEW' FALLA

- Como hemos visto hasta ahora, para asignar memoria de forma dinámica en el Heap se usa el operador `new`, por lo general este operador va a funcionar siempre, pero puede darse el caso de que el operador falle y si no lo solucionas el programa *crasheará*. Para evitar esto es importante asegurarse de que la asignación de memoria se ha realizado correctamente.
- Hay 2 métodos para comprobar que la asignación de memoria se ha realizado correctamente:
  - El mecanismo para tratar las excepciones proporcionado por C++ (Try - Catch):
    Las excepciones sirven para manejar un error que ocurra durante la ejecución, de tal manera que haga que el programa no *crashee*.
    Para ello se envuelve el código en un bloque `try`, seguido de un bloque `catch`, para que en caso de error, en vez de *crashear*, el programa ejecutará la parte de código del `catch`, que puede tener un mensaje avisando del error o incluso una posible solución al propio error.
  - Usar `std::nothrow` en el operador `new` para forzar que no lance una excepción si la asignación de memoria falla. Simplemente devolverá un puntero inicializado a `nullptr`, lo cual nos indica que la asignación de memoria ha fallado.

8. NULL POINTER SAFETY

- El concepto **Null Pointer Safety** son una serie de medidas tomadas para asegurarse de que cuando usas un puntero, este contiene una dirección de memoria válida. Si la dirección de memoria no es válida, no hay que usar el puntero.
- Para evitar el `null pointer` se llevan a cabo una serie de comprobaciones:
  - Un `if` que compare el valor de nuestro puntero con `nullptr` antes de empezar a usarlo, para saber si la dirección de memoria que tiene es válida.

Incluso no es necesario una comparación con `nullptr` ya que los punteros se pueden convertir implícitamente en `booleans` si nuestro `if` es lo que espera.
Puedes llamar a `delete` en un puntero que contenga `nullptr`, no hay problema con eso, por lo que no es necesaria una comprobación.

9. FUGAS DE MEMORIA (MEMORY LEAKS)

- Una fuga de memoria se produce cuando pierdes acceso a una parte de memoria que estaba asignada dinámicamente a nuestro programa. Básicamente pierdes el puntero que apuntaba a esa porción de memoria asignada.
- Al producirse una fuga de memoria, el sistema no puede emplear la parte de memoria asignada dinámicamente porque no ha sido liberada por nuestro programa, pero es que nuestro programa tampoco puede usarla porque nuestro puntero ya no apunta a esa dirección. Obviamente esto es un problema pues se está asignando una memoria que no está pudiendo ser usada ni por nuestro programa ni por el sistema, y cuando dicha memoria de Heap se agote el programa *crasheará*.
- Casos en los que se dan lugar fugas de memoria:
  - Reasignación de una dirección de memoria de Stack en un puntero activo que contiene una dirección de memoria dinámica (Heap).
  - Asignación de dos direcciones de memoria dinámica en nuestro puntero activo.
  - Declaración y asignación de memoria dinámica de un puntero en un ámbito local.

  Estos ejemplos se van a ver muy claramente en los proyectos con código.
- Es muy importante evitar a toda costa la fuga de memoria, para ello hay que cerciorarse de liberar la memoria y dársela de vuelta al sistema, llamando a `nullptr` en el puntero y reiniciándolo a `nullptr`, como hemos ido diciendo en todo este capítulo.

10. ARRAYS DINÁMICOS

- Son arrays que se alojan en Heap, y no en Stack como habíamos visto hasta ahora.
- Para alojar arrays en Heap se usa el operador `new`, también se puede usar el modificador `std::nothrow` para evitar excepciones.
- La declaración e inicialización es la misma que la de cualquier otro puntero solo que esta vez el tipo de dato es un array.
  `int *pEstudiantes = new int[10];` //Array con valores basura, no inicializado
- Puedes realizar operaciones corrientes de un array, aunque sea un array dinámico.
- La memoria se libera de una forma muy similar al resto de punteros:
  ```
  delete[] pEstudiantes;
  pEstudiantes = nullptr;
  ```
- Los arrays dinámicos **NO** requieren de un tamaño constante.
- Los arrays dinámicos tienen limitaciones:
  - `std::size` no funciona con los arrays dinámicos, ya que el puntero apunta al primer elemento del array y la información sobre el tamaño no se almacena junto al puntero, solo su dirección de memoria. `std::size` no puede determinar el tamaño de un array dinámico porque desconoce el tamaño original de la memoria dinámica asignada.
  - Los bucles for in range no son compatibles con los arrays dinámicos, porque necesitan el primer y el último elemento para saber cuándo parar de iterar, y en este caso esa información es desconocida, ya que el puntero apunta al primer elemento del array y la información sobre el tamaño no se almacena junto al puntero, solo se almacena su dirección de memoria.