[Γα=Ω5]

Punteros y Memoria

Por Ariel Parra

[Γα=Ω5]

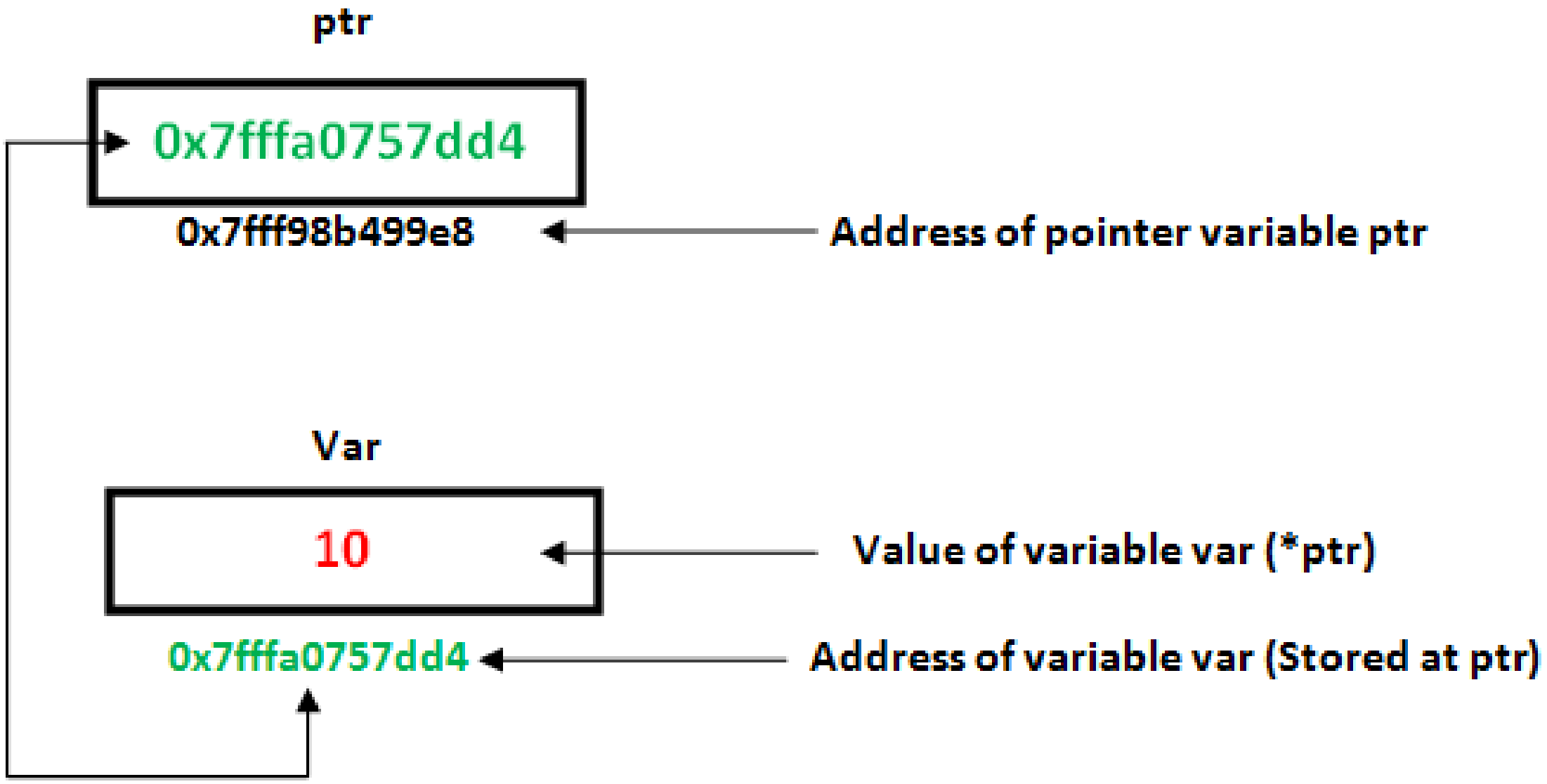
Punteros (Pointers)

Un puntero también conocido como apuntador es una variable que almacena la dirección de memoria de una variable u objeto. Los 3 propósitos principales son:

1. para asignar nuevos objetos en el heap
2. pasar funciones a otras funciones
3. para iterar sobre elementos en matrices u otras estructuras de datos.

Declaración de punteros:

```
int n = 5;
int *p = nullptr; // se lee como: la variable 'p' es puntero de tipo entero inicializado en nulo
p = &n; //variable 'p' almacena la dirección de un entero 'n'
cout << *p; //accedemos a su valor con el operador de desrreferencia '*'
cout << p; // mostrara su dirección de memoria
```


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

Referencias (References)

Una referencia, como un puntero, es una expresión que almacena la dirección de un objeto que se encuentra en la memoria. A diferencia de un puntero, una referencia después de su inicialización no puede hacer referencia a un objeto diferente ni establecerse como nula.

Declaración de referencias (operador ampersand):

```
int n = 5;
int &n1 = n; //alias
```

¿Qué salida tendra el cout?

```
int n = 5;
int *p = &n;
cout<< &n << n << p << *p;
```

Vectores con punteros

los vectores apuntan a una dirección de memoria que es el inicio de este.

```
int vec[3] = { 10, 20, 30 };
int* ptr;
ptr = vec; // ptr=&vec[0];
for(int i=0;i<3;i++){
    cout<< ptr[i];
}
```

Los strings al ser un vector de caracteres es lo mismo

```
char *s = "ola";
cout<< *s << s[1] << *(s+2);
```

Paso de Parametros

Con Punteros:

```
void squarePtr(int *n){
    *n *= *n;
}
int n = 5; int *ptr = &n;
squarePtr(&n); squarePtr(ptr);
```

Por referencia:

```
void squareRef(int& n){
    n *= n;
}
squareRef(n);
```

Memoria dinámica

Permite asignar y liberar memoria en ejecución, se usa cuando no se conoce desde antes cuánto espacio de memoria se necesitará, como cuando el tamaño de una estructura depende de la entrada del usuario.

```
int main(){
    int n; cin >> n;

    int* vecC = (int*)malloc(n * sizeof(int)); // en C
    int* vecCpp = new int[n]; // en C++

    int size = sizeof(vecC) / sizeof(vecC[0]); // tamaño del vector en C

    free(vecC); // en C
    delete[] vecCpp;// en C++, también se puede: delete []vec;
return 0;
}
```

Funciones de tipo puntero

```
int *crearVec(int tam){
    int* aux = new int[n];
    for(int i=0;i<tam;i++)
        aux[i]=i;
    return aux;
}
int main(){
    int n; cin>>n;
    int* vec = crearVec(n);
    for(int i=0;i<n;i++)
        cout << vec[i];
    delete[] vec;
return 0;
}
```

Aritmética de Punteros

La memoria al ser declarada contigua permite el uso de operaciones aritméticas con punteros.

Ejemplo:

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;  // Apunta a arr[0]
ptr++;  // Ahora apunta al segundo elemento arr[1] = (20)
cout << *(ptr++); // ¿Qué mostrara?
```

Esto se puede expandir a un uso de iteradores en vectores:

```
int vec[] = {10, 20, 30};
int *ptr = vec;
for(int i=0; i < 3;++i)
    cout << *(ptr+i); // ptr[i];
```

Aritmética de Punteros en **matrices**:

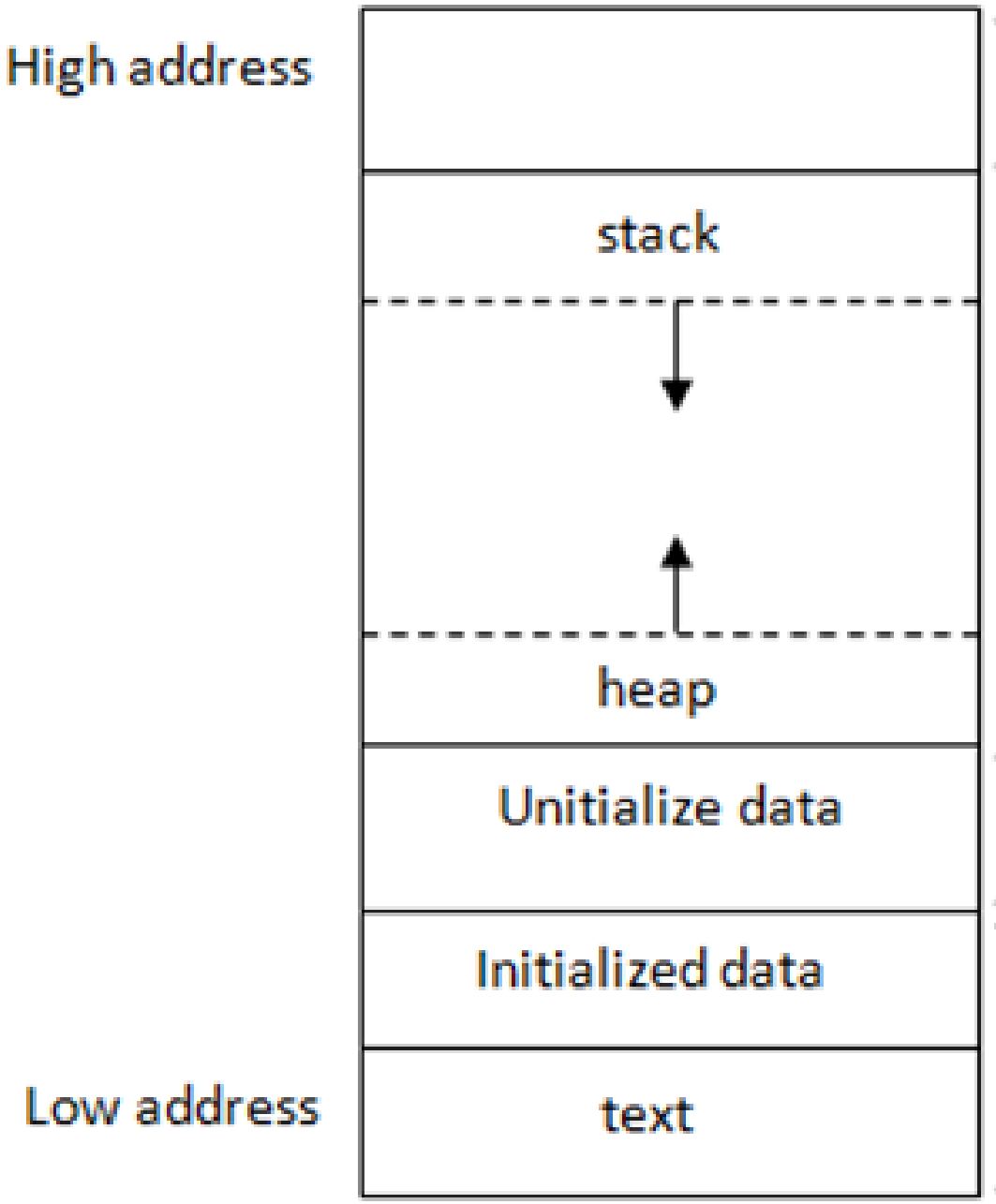
```
const int REN=3, COL=3;
int mat[REN][COL]={ { 10, 20, 30 },
                    { 40, 50, 60 },
                    { 70, 80, 90 } };
int* ptr = &mat[0][0];
for(int i=0; i< REN ;i++){
    for(int j=0; j< COL ;j++){
        cout << *(ptr + i * COL+ j) ;
    }
}
```

¿Creen que este código funcione?

```
int vec[] = {10, 20, 30}; int *ptr = vec;
for(int i=0; i < 3; ++i) cout << i[ptr];
```

Distribución de la memoria

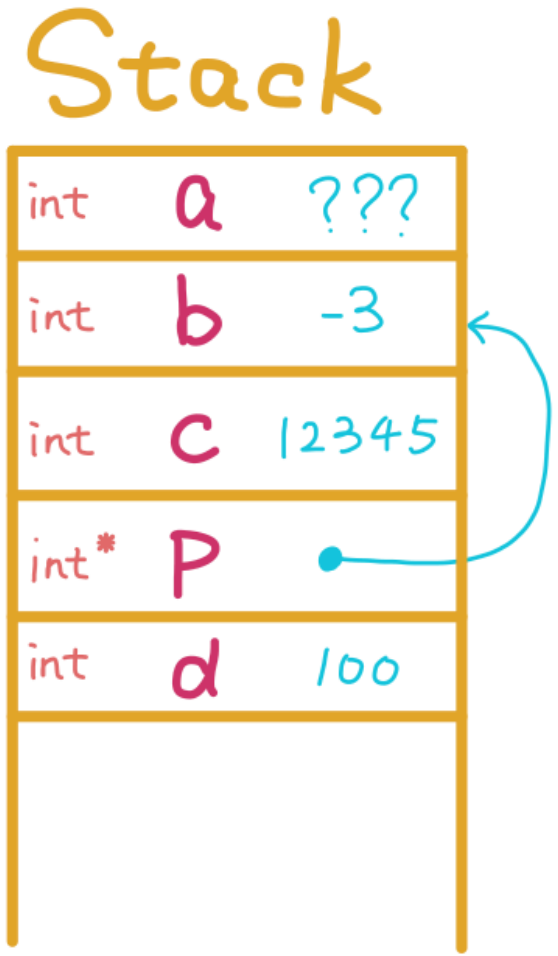
- **stack**: almacena variables locales
- **heap**: memoria dinámica para que la asigne el programador
- **data**: almacena variables globales, separadas en inicializadas y no inicializadas
- **text**: almacena las funciones y el código que se está ejecutando



El Stack

El segmento de stack en la parte superior de la memoria. Cada vez que se llama a una función, el CPU le asigna algo de memoria al stack. Cuando se declara una nueva variable local, se le asigna más memoria de stack a esa función para almacenar la variable. Estas asignaciones hacen que la stack crezca hacia abajo. Después de que la función retorna, la memoria de stack de esta función se desasigna, lo que significa que todas las variables locales dejan de ser válidas. **La asignación y desasignación de memoria de stack se realiza automáticamente**.

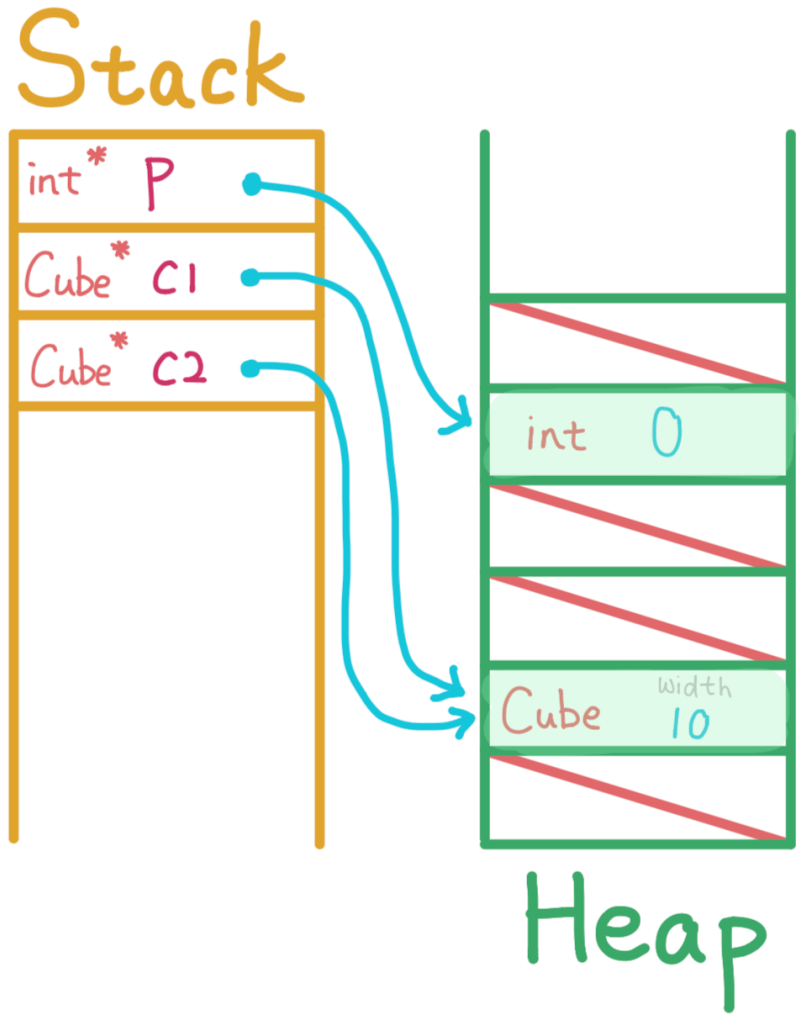
Ejemplo de cómo se ve la memoria de stack cuando se ejecuta el código correspondiente:




```c
int hello() {
  int a = 100;
  return a;
}
int main() {
  int a;
  int b = -3;
  int c = 12345;
  int *p = &b;
  int d = hello();
  return 0;
}
```

El Heap

A diferencia de la memoria de stack, la memoria del heap es asignada explícitamente por los programadores y no se desasignará hasta que se libere explícitamente. (Memoria dinámica).




```
int main() {
  int *p = new int;
  Cube *c1 = new Cube();
  Cube *c2 = c1;
  c2->setLength( 10 );
  return 0;
}
```

Cuando usar el Heap o Stack

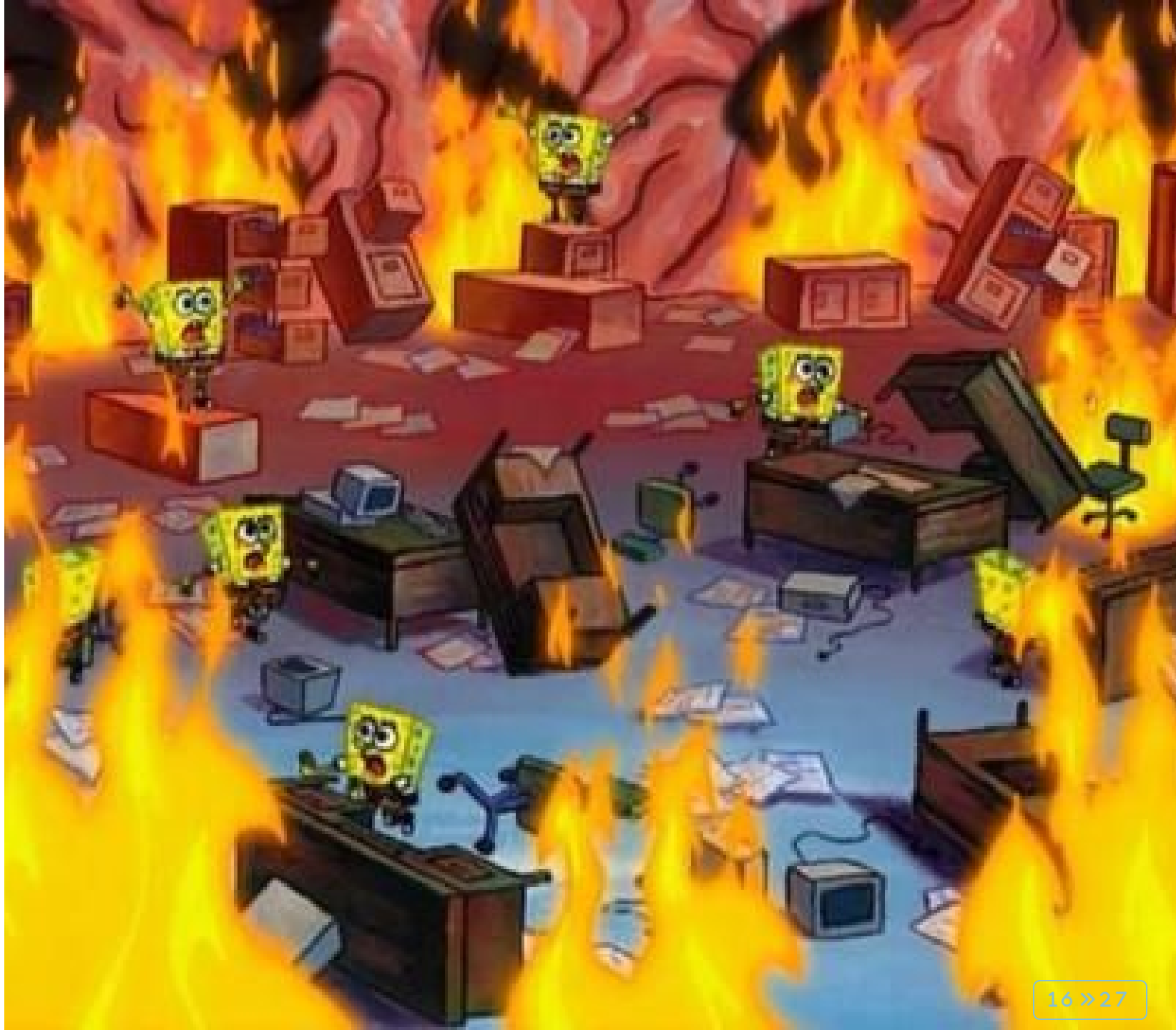
Usa el stack cuando:

- No deseas desasignar variables usted mismo.
- Necesitas velocidad (la CPU gestiona eficientemente el espacio).
- El tamaño variable es estático.

Usa el Heap cuando:

- Necesitas una gran cantidad de espacio (prácticamente sin límite de memoria).
- No le importa un acceso un poco más lento (pueden ocurrir problemas de fragmentación).
- Quiere pasar objetos (globales) entre funciones.
- Te gusta gestionar las cosas tú mismo.
- El tamaño variable podría ser dinámico.

Complicaciones



Colisión entre Heap y Stack

```c
void recursion(int depth) {
    int largeArray[10000];  // Asignación grande en el stack
    recursiveFunction(depth + 1);
}

int main() {
    // Asignación de un bloque grande de memoria en el heap
    int *heapMemory = (int*)malloc(100000000 * sizeof(int));

    // Iniciar la función recursiva que consumirá el stack
    recursiveFunction(1);

    // Liberar la memoria en el heap (aunque no se alcanzará si ocurre una colisión)
    free(heapMemory);
    return 0;
}
```

Pointer to a Constant (const int* ptr)

No puedes cambiar el valor apuntado, pero puedes cambiar a qué dirección apunta el puntero.

```
int x = 10;
const int* ptr = &x; // ptr es un puntero a una constante int

// *ptr = 20;  // Error: no se puede modificar el valor a través de ptr
x = 20;        // Esto es válido, el valor de 'x' se puede modificar directamente

int y = 30;
ptr = &y;      // Esto es válido, ptr puede apuntar a otro entero
```

Constant Pointer (int* const ptr)

Puedes cambiar el valor apuntado, pero no puedes cambiar a qué dirección apunta el puntero.

```
int x = 10;
int* const ptr = &x; // ptr es un puntero constante

*ptr = 20;   // Esto es válido, el valor de 'x' se puede modificar a través de ptr

// int y = 30;
// ptr = &y;  // Error: no se puede cambiar la dirección a la que apunta ptr
```

void*

```
int n = 42;
void* ptr = &n;  // ptr es un puntero genérico que apunta a un int

// Necesitas hacer un cast para desreferenciar:
int* intPtr = (int*)ptr;
```

Memory leaks

```
int main() {
    int *ptr = new int[100];
    return 0;
}
```

Conclusión

Problemas

Resuelve los siguientes problemas usando aritmetica de punteros, memoria dinámica, etc.

- **734A** Anton and Danik
- **300A** Array

Referencias

- Alex Hyett. (2022). *Stack vs Heap Memory - Simple Explanation* [video]. Recuperado de https://youtu.be/5OJRqkYbK-4?si=luRFJ6hTUlxJpWi6
- Aspnes, J. (2014). *C/Pointers*. Recuperado de https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)Pointers.html
- Bro Code. (2021). *Learn C memory addresses in 7 minutes* 📬 [video]. Recuperado de https://youtu.be/1KVpi0VN82E?si=N44zYykfuip9x8UJ
- Bro Code. (2024). *C++ pointers explained easy* 👈 [video] . Recuperado de https://www.youtube.com/watch?v=slzcWKWCMBg
- Chase, J. (2016). *Notes on Heap Manager in C*. Recuperado de https://courses.cs.duke.edu/spring16/cps110/slides/heap.pdf
- Chen, J. & Guo, R. (2022). *Stack and Heap Memory*. Recuperado de https://courses.grainger.illinois.edu/cs225/fa2022/resources/stack-heap/

- cplusplus. (s.f.).*Pointers*. Recuperado https://cplusplus.com/doc/tutorial/pointers/
- CS50. (2017). *Pointers - CS50 Shorts* [video]. Recuperado de https://youtu.be/XISnO2YhnsY?si=dR_Px2OfnFPVlpOs
- CS50. (2023). *CS50x 2024 - Lecture 4 - Memory* [video] . Recuperado de https://youtu.be/F9-yqoS7b8w?si=WAQ3xs808ncA3el5
- Cusimano, R. (2022). *Pointers vs References*. Recuperado de https://cppbyexample.com/pointers_vs_references.html
- Cusimano, R. (2022). *What Are Pointers*. Recuperado de https://cppbyexample.com/pointer.html
- Cusimano, R. (2022). *What Are References*. Recuperado de https://cppbyexample.com/references.html
- Cusimano, R. (2022). *What is the Heap*. Recuperado de https://cppbyexample.com/what_is_the_heap.html
- Cusimano, R. (2022). *What is the Stack*. Recuperado de https://cppbyexample.com/what_is_the_stack.html
- Cusimano, R. (2022). *Why Use Pointers*. Recuperado de https://cppbyexample.com/why_pointer.html
- Cusimano, R. (2022). *Why Use References*. Recuperado de https://cppbyexample.com/why_references.html
- Dave's Garage. (2023). *Master Pointers in C: 10X Your C Coding!* [video] . Recuperado de https://youtu.be/IrGjyfBC-u0?si=8BlMpHTvQRkXTwW1

- freeCodeCamp.org. (2020). *Pointers in C / C++ [Full Course]* [video] . Recuperado de https://youtu.be/zuegQmMdy8M?si=21whSJBB0wItx625
- freeCodeCamp.org. (2023). *Pointers in C for Absolute Beginners – Full Course* [video] . Recuperado de https://www.youtube.com/watch?v=MIL2BK02X8A
- GeeksforGeeks. (2024). *References in C++*. Recuperado de https://www.geeksforgeeks.org/references-in-cpp/
- Kariya, A. (2024). *C++ Pointers*. Recuperado de https://www.geeksforgeeks.org/cpp-pointers/
- Low Level Lerning. (2022). *what even is a "reference"?* [video] . Recuperado de https://youtu.be/wro8Bb6JnwU?si=w5oB1hUxNzEpcH8P
- Low Level Lerning. (2022). *you will never ask about pointer arithmetic after watching this video* [video] . Recuperado de https://youtu.be/q24-QTbKQS8?si=3N5Cy1G_AaObg_JO
- Low Level Lerning. (2022). *you will never ask about pointers again after watching this video* [video] . Recuperado de https://youtu.be/2ybLD6_2gKM?si=WsGGHYCRYycqyHNR
- Low Level Lerning. (2023). *The Origins of Process Memory | Exploring the Use of Various Memory Allocators in Linux C* pointers even exist?* [video] . Recuperado de https://youtu.be/c7xf5dvUb_Q?si=liTAHayttd4YWN93
- Low Level Lerning. (2023). *why do void* pointers even exist?* [video] . Recuperado de https://youtu.be/t7CUti_7d7c?si=WNufRDLbkIPsSyzv

- Microsoft. (2021). *Pointers (C++)*. Recuperado de https://learn.microsoft.com/en-us/cpp/cpp/pointers-cpp?view=msvc-170
- Microsoft. (2024). *References (C++)*. Recuperado de https://learn.microsoft.com/en-us/cpp/cpp/references-cpp?view=msvc-170
- MIT. (2011). *Lecture 5 Notes: Pointers*. Recuperado de https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/0240aeefb6d5fb9c0a20587ed98fa7ca_MIT6_096IAP11_lec05.pdf
- mycodeschool. (2013). *Pointers and dynamic memory - stack vs heap* [video]. Recuperado de https://youtu.be/_8-ht2AKyH4?si=fbCY_IeJD_dIQgm9
- Portfolio Courses. (2021).*Introduction to Pointers | C Programming Tutorial* [video]. Recuperado de https://youtu.be/2GDiXG5RfNE?si=qEWVhJ3cs--SsYkU
- Portfolio Courses. (2022). *Constant Pointer VS. Pointer To A Constant | C Programming Tutorial* [video]. Recuperado de https://youtu.be/egvGq3WSF9Y?si=EjCk9hBtGSK0zaLh
- Portfolio Courses. (2022). *int p vs int p Pointer Declarations | C Programming Tutorial* [video]. Recuperado de https://youtu.be/H5MlTh5cBOg?si=SSy-8eLBmbjGZ3Gj

- The Cherno. (2017). *POINTERS in C++* [video] . Recuperado de https://youtu.be/DTxHyVn0ODg?si=kdi8CpqxyL-ct6H_
- The Cherno. (2017). *Stack vs Heap Memory in C++* [video]. Recuperado de <http s://youtu.be/wJ1L2nSIV1s?si=Zb7nvQelG4Jz6Tz1>
- The Cherno. (2023). *Should I pass by const reference or by value?* [video]. Recuperado de https://youtu.be/lj1O_GH7SGs?si=65CFz0ojo9JxNp9X
- Tortellini Soup. (2024). *Why This Works* [video] . Recuperado de https://www.youtube.com/watch?v=lFRp0tZKosc
- W3Schools. (s.f.). *C++ Pointers*. Recuperado de https://www.w3schools.com/cpp/cpp_pointers.asp