Punteros

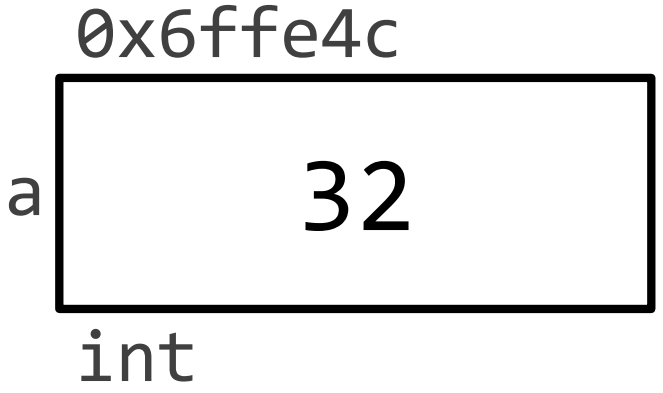
PROF. ÁLVARO SALINAS

Dirección de Memoria

Cuando definimos una variable, se reserva automáticamente un espacio de memoria para almacenar los valores que le demos a esa variable.

Nosotros podemos conocer la dirección de memoria de una variable con el operador &.

```
int a = 32;
std::cout << a; // 32
std::cout << &a; // 0x6ffe4c
```

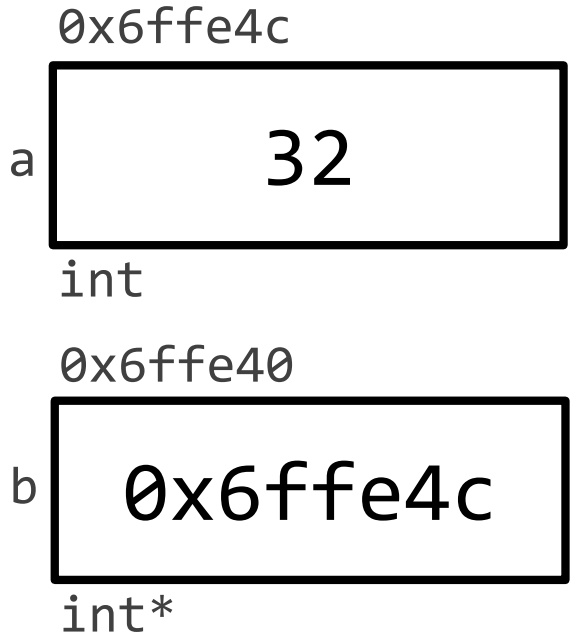


Puntero

Un puntero es una variable donde se almacena la dirección de memoria de un dato.

Un puntero está asociado a un tipo de dato al igual que las variables normales.

```
int a = 32;
std::cout << a; // 32
std::cout << &a; // 0x6ffe4c
int *b = &a;
std::cout << b; // 0x6ffe4c
std::cout << &b; // 0x6ffe40
```

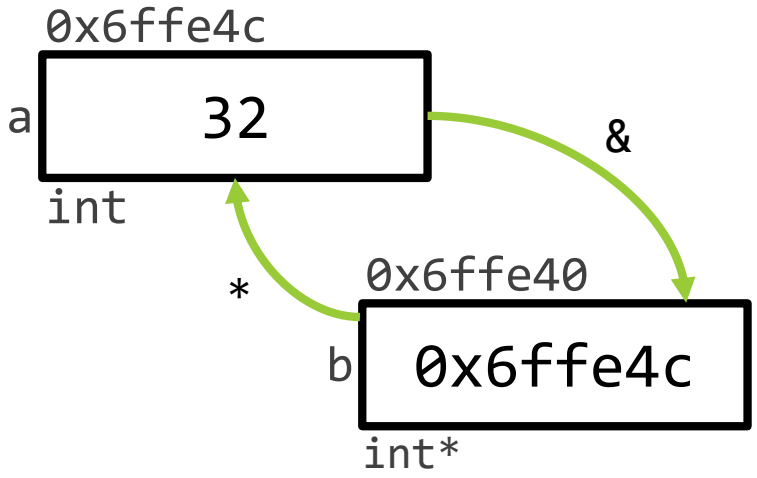


Puntero

El operador * no solo nos permite definir un puntero, sino que también es el operador inverso a &.

En otras palabras, si el operador & nos entrega la dirección de memoria de una variable que almacena un dato, el operador * nos entrega el dato al que “apunta” la dirección de memoria almacenada en un puntero.

```
int a = 32;
int *b = &a;
std::cout << a; // 32
std::cout << *(&a); // 32
std::cout << *b; // 32
```



Puntero a Puntero

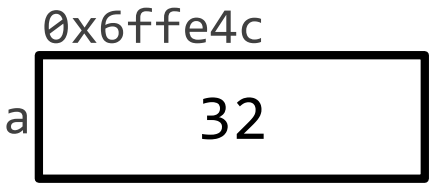
Podemos tener punteros que apunten a otros punteros, es decir, podemos almacenar las direcciones de memoria donde se encuentran otras direcciones de memoria.

```
int a = 32;
int *b = &a;
int **c = &b;
int ***d = &c;
std::cout << a; // 32
std::cout << *b; // 32
std::cout << **c; // 32
std::cout << ***d; // 32
```

Puntero a Puntero

Podemos tener punteros que apunten a otros punteros, es decir, podemos almacenar las direcciones de memoria donde se encuentran otras direcciones de memoria.

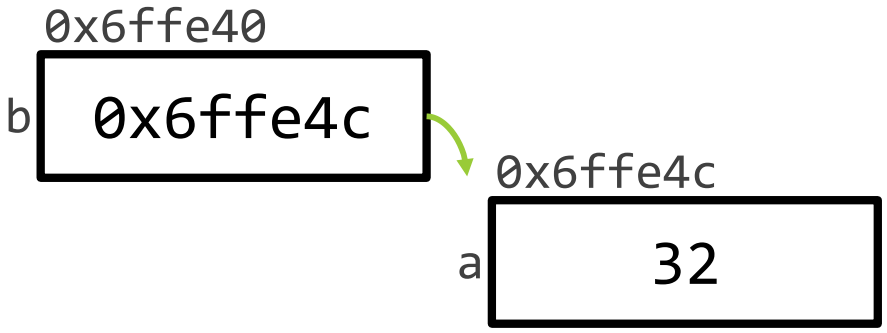
```
int a = 32;
int *b = &a;
int **c = &b;
int ***d = &c;
std::cout << a; // 32
std::cout << *b; // 32
std::cout << **c; // 32
std::cout << ***d; // 32
```



Puntero a Puntero

Podemos tener punteros que apunten a otros punteros, es decir, podemos almacenar las direcciones de memoria donde se encuentran otras direcciones de memoria.

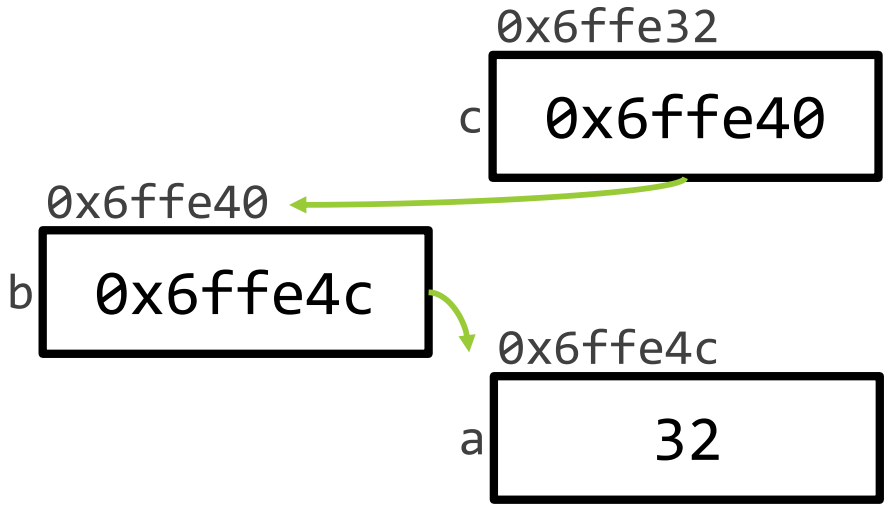
```
int a = 32;
int *b = &a;
int **c = &b;
int ***d = &c;
std::cout << a; // 32
std::cout << *b; // 32
std::cout << **c; // 32
std::cout << ***d; // 32
```



Puntero a Puntero

Podemos tener punteros que apunten a otros punteros, es decir, podemos almacenar las direcciones de memoria donde se encuentran otras direcciones de memoria.

```
int a = 32;
int *b = &a;
int **c = &b;
int ***d = &c;
std::cout << a; // 32
std::cout << *b; // 32
std::cout << **c; // 32
std::cout << ***d; // 32
```



Puntero a Puntero

Podemos tener punteros que apunten a otros punteros, es decir, podemos almacenar las direcciones de memoria donde se encuentran otras direcciones de memoria.

```
int a = 32;
int *b = &a;
int **c = &b;
int ***d = &c;
std::cout << a; // 32
std::cout << *b; // 32
std::cout << **c; // 32
std::cout << ***d; // 32
```

Modificar Valores

Modificar el valor de una variable puede realizarse utilizando el puntero que apunta a su dirección de memoria.

```
int a = 32;
int *b = &a;
std::cout << a; // 32
*b = 24;
std::cout << a; // 24
(*b)++;
std::cout << a; // 25
```

Paso por Referencia

Si “pasamos por valor” una variable como parámetro a una función, en realidad estamos pasando una copia de su valor y las modificaciones que dicha variable sufra dentro de la función no serán realizadas sobre la variable original.

```
void f(int x){ x++; }

int main(){
	int a = 2;
	f(a);
	std::cout << a; // 2
	return 0;
}
```

Paso por Referencia

El “paso por referencia” por el contrario, permite que los cambios realizados se mantengan en la variable original. Aquí, el operador & no se utiliza para obtener la dirección de memoria, sino para crear una referencia de la variable.

```
void f(int &x){ x++; }

int main(){
	int a = 2;
	f(a);
	std::cout << a; // 3
	return 0;
}
```

Paso por Referencia

Esto mismo lo podemos hacer utilizando punteros. Lo que queremos, es pasarle a la función la dirección de memoria de la variable, por lo que los cambios que realicemos no los estaremos realizando sobre una copia del valor, sino sobre la ubicación del valor original.

```
void f(int *x){ (*x)++; }

int main(){
	int a = 2;
	f(&a);
	std::cout << a; // 3
	return 0;
}
```

Punteros y Arreglos

Podemos acceder a los elementos de un arreglo utilizando aritmética de punteros luego de hacer que un puntero apunte al primer elemento de dicho arreglo.

```
int a[4] = {1,2,3,4};
int *b = &a[0];
std::cout << *b; // 1
std::cout << *(++b); // 2
std::cout << *(b+1); // 3
std::cout << a[3]; // 4
```

Arreglos Dinamicos

Un arreglo dinámico permite utilizar un espacio de memoria cuyo tamaño será conocido en tiempo de ejecución. Para esto, solo debemos crear un puntero que apuntará a la dirección de memoria en donde comenzará dicho arreglo y posteriormente, reservar la memoria que utilizaremos. La reserva de memoria se realiza con la función malloc.

```
int *a;
int n;
std::cin >> n;
a = malloc(n * sizeof(int));
for(int i = 0; i < n; i++)
       a[i] = i;
```

Arreglos Dinamicos

Una vez, hayamos reservado la memoria con el uso de malloc, debemos liberarla con la función free. De esta manera indicamos que dicho espacio de memoria se encuentra disponible para futuras llamadas a malloc.

```
int *a;
int n;
std::cin >> n;
a = malloc(n * sizeof(int));
for(int i = 0; i < n; i++)
       a[i] = i;
free(a);
```