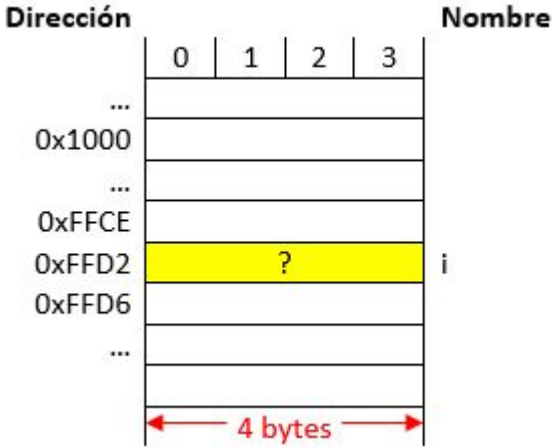
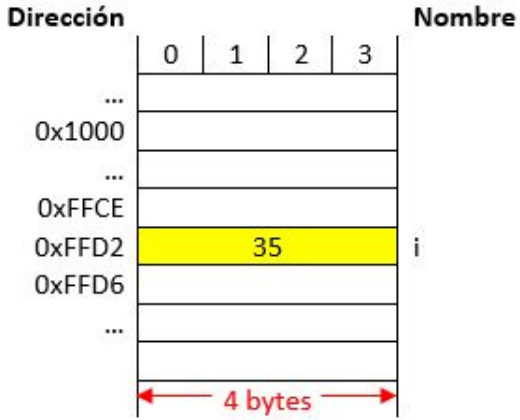
ACTIVIDAD PREVIA

Leer y simular los temas de los enlaces apuntadores y arreglos y estructuras en C, traer dudas para discutir, entre mas pregunten mejor pues habrá quiz en la plataforma de este tema y sera habilitado al otro dia durante 24 horas (de ser posible con 2 intentos como máximo).

VARIABLES Y MEMORIA



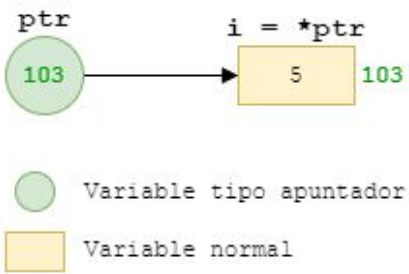


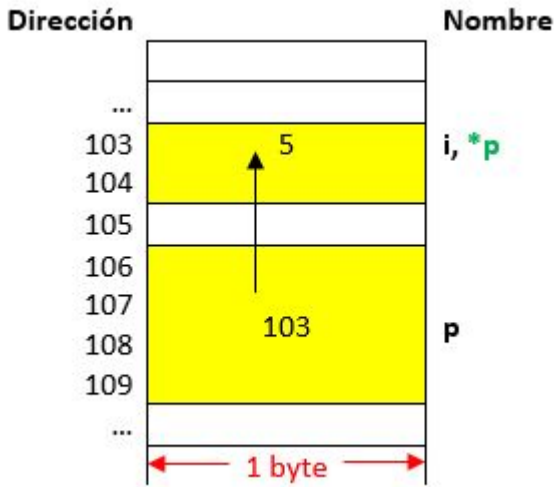
APUNTADORES

- Almacena una posicion de memoria.
- Operadores:
  - **&**: Direccion (obtiene dirección)
  - *: Desreferencia (declaración, accede al lugar apuntado).

```
short i = 5;
short *ptr = &i;
```

```
//instrucciones
short i = 5;
short *ptr = &i;
```





APUNTADORES - CAPTANDO EL CONCEPTO

Ejemplo: Dado el fragmento de código mostrado a continuación, y teniendo en cuenta que:

- Suponga que i y j son de 4 bytes y ocupan las direcciones base 1000 y 1004.
- El apuntador p ocupa las direccione base 2000.
- Así mismo la arquitectura es de 64 bits por lo que el espacio ocupado por el apuntador sera de 8 bytes.

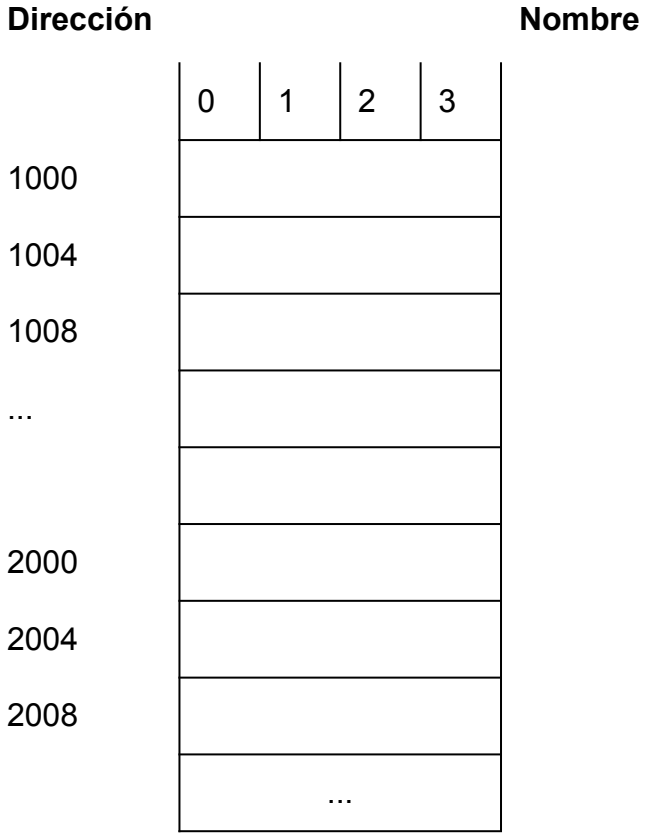
```
int main() {
  int i,j;
  int *p;
  p = &i;
 *p = 5;
  return 0;
}
```

Muestre la ejecución paso a paso del código anterior resaltando la evolución en memoria.

CAPTANDO EL CONCEPTO

- Suponga que i y j son de 4 bytes y ocupan las direcciones base 1000 y 1004.
- El apuntador p ocupa las direccione base 2000.
- Así mismo la arquitectura es de 64 bits por lo que el espacio ocupado por el apuntador sera de 8 bytes.

```
int main() {
  int i,j;
  int *p;
  p = &i;
 *p = 5;
  return 0;
}
```



Solución online

APUNTADORES - CAPTANDO EL CONCEPTO

Ejemplo: Para el siguiente código suponga lo siguiente:

- Suponga que i y j son de 4 bytes y ocupan las direcciones base 1000 y 1008.
- Los apuntadores p, q y r ocupan las direcciones base 2000, 3000 y 4000.
- Así mismo la arquitectura es de 32 bits por lo que el espacio ocupado por el apuntador será de 4 bytes.

```
int main() {
  int i;
  int *p,*q,*r;
  p = &i;
  q = &i;
  r = p;
  return 0;
}
```

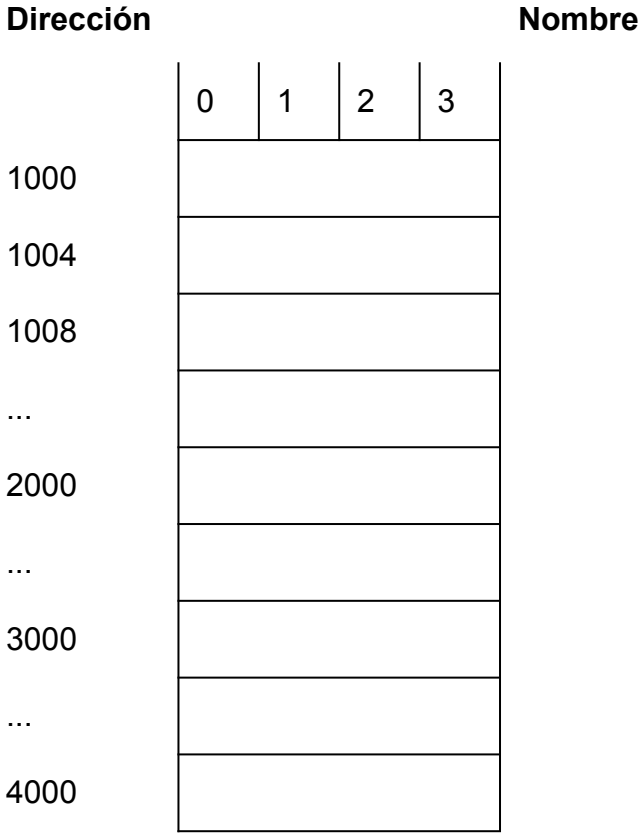
Muestre la ejecución paso a paso del código anterior resaltando la evolución en memoria.

CAPTANDO EL CONCEPTO

- Suponga que i y j son de 4 bytes y ocupan las direcciones base 1000 y 1008.
- Los apuntadores p, q y r ocupan las direcciones base 2000, 3000 y 4000.
- Así mismo la arquitectura es de 32 bits por lo que el espacio ocupado por el apuntador será de 4 bytes

```
int main() {
  int i;
  int *p,*q,*r;
  p = &i;
  q = &i;
  r = p;
  return 0;
}
```

Solución online



NOTAS IMPORTANTES SOBRE APUNTADORES

- Los apuntadores deben ser inicializados a una posición de memoria antes de ser usado, sino habrá error.

```
1  int main() {
2    int *p;
3    int a = 4;
4    *p = 8;
5    return 0;
6  }
```

ERROR: Use of uninitialised value of size 8
(Stopped running after the first error. Please fix your code.)

Ejcución online

```
1  int main() {
2    int *p = 8;
3    int a = 4;
4    *p = 3;
5    return 0;
6  }
```

ERROR: Invalid write of size 4
(Stopped running after the first error. Please fix your code.)

Ejcución online

NOTAS IMPORTANTES SOBRE APUNTADORES

- Los apuntadores deben ser inicializados a una posición de memoria antes de ser usado, sino habrá error.

```
1  int main() {
2    int a;
3    int *p = &a;
4    *p = 6;          // a = 6
5    *p = a + *p;     // a = a + a;
6    return 0;
7  }
```

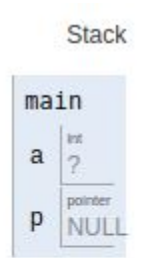
Ejcución online

```
1  int main() {
2    int a;
3    int *p;
4    p = &a;
5    *p = 6;          // a = 6
6    *p = a + *p;     // a = a + a;
7    return 0;
8  }
```

Ejcución online

```
1  #define NULL 0
2
3  int main() {
4    int a;
5    int *p = NULL;
6    p = &a;
7    *p = 6;          // a = 6
8    *p = a + *p;     // a = a + a;
9    return 0;
10 }
```

Ejcución online



FUNCIONES POR REFERENCIA Y POR VALOR

- **Por valor**: Sobre copias de las variables pasadas como parámetros a la función.

```
tipo_retorno f(tipo_1 vName_1,...,tipo_N vName_N)
```

```
void swap(int i, int j) {
 int k = i;
 i = j;
 j = k;
}
```

- **Por referencia**: Directamente sobre la variable pasada como parámetro haciendo uso de apuntadores.

```
tipo_retorno f(tipo_1 *pName_1,...,tipo_N *pName_N)
```

```
void swap(int *i, int *j) {
 int k = *i;
 *i = *j;
 *j = k;
}
```

FUNCIONES POR REFERENCIA Y POR VALOR

Analizando las diferencias

Item analizado	Llamada por valor	Llamada por referencia
Declaración	`void swap(int i, int j);`	`void swap(int *i, int *j);`

Definición	`void swap(int i, int j) {`
	`  int t = i;`
	`  i = j;`
	`  j = t;`
	`}`
		`void swap(int *i, int *j) {`
		`  int t = *i;`
		`  *i = *j;`
		`  *j = t;`
		`}`

Invocación	`int v1 = 1, v1 = 2;` `swap(v1, v2);`	`int v1 = 1, v2 = 2;` `swap(&v1, &v2);`

```c
void swap1(int, int);
void swap2(int *, int *);

int main() {
  int a = 3, b = -5;
  // a = 3, b = -5
  swap1(a, b);
  // a = 3, b = -5
  swap2(&a, &b);
  // a = -5, b = 3
  return 0;
}

void swap1(int a, int b) {
  int c;
  c = a;
  a = b;
  b = c;
}

void swap2(int *a, int *b) {
  int c;
  c = *a;
  *a = *b;
  *b = c;

}
```

Ejecución online

FUNCIONES POR REFERENCIA Y POR VALOR

- Analizar el siguiente código:

```
void A(int, int *);

int main() {
  int a1, a2, a3;
  int *p1 = &a1;
  int *p2 = &a2;
  int *p3 = p1;
  *p3 = a3;
  // a1 = ______ , a2 = ______, a3 = ______
  A(*p3, &a3);
  // a1 = ______ , a2 = ______, a3 = ______
  A(a1, p2);
  // a1 = ______ , a2 = ______, a3 = ______
  A(*p1, p2);
  // a1 = ______ , a2 = ______, a3 = ______
}

void A(int a, int *b) {
  a += 1;
  *b = 2*a;
}
```

Ejcución online

FUNCIONES POR REFERENCIA Y POR VALOR

Comprendiendo lo anterior responda falso o verdadero para cada uno de los casos siguientes en los que se invoca la función mostrada a continuación:

```
void f(int *p1, int *p2, int z) {
 ...
}
```

Caso 1: ____

```
int x = 3, y = 4, z = 1;
f(&x, &y, z);
```

Caso 2: ____

```
int x = 3, y = 4, z = 1;
f(&x, &y, &z);
```

FUNCIONES POR REFERENCIA Y POR VALOR

```
void f(int *p1, int *p2, int z) {
 ...
}
```

Caso 3: ____

```
int x = 3, y = 4, z = 1;
int *xp = &y;
f(xp, &y, z);
```

Caso 4: ____

```
int x = 3, *y, z = 1;
y = &z;
f(&x, &y, y);
```

Caso 5: ____

```
int x = 3, *y, z = 1;
y = &z;
f(&x, &y, y);
```

Caso 6: ____

```
int x = 3, *y, z = 1;
y = &z;
int *zz = y;
f(zz, y, *y);
```

VECTORES Y APUNTADORES

- Los vectores manejan el mismo método de indexación que java u otros lenguajes. Pero a diferencia de java (que necesita el new), es posible asignar en la declaración el espacio de memoria. Esto dependerá del problema.
- Formas de declaración:
  - Forma 1:

```
tipo arrayName[TAM] = {valor1, valor2, ...};
```

  - Forma 2:

```
tipo arrayName[] = {valor1, valor2, ...};
```

  - Forma 3;

```
tipo arrayName[TAM];
```

VECTORES Y APUNTADORES

- El acceso para procesar los elementos del array es similar al de cualquier lenguaje como java:
  - **Ejemplo**: Hacer un programa que llene un arreglo de 10 elementos con los multiplos del 10 (1, 10, 20, etc.).

```c
#include <stdio.h>

#define TAM 10

int main() {
  int A[TAM];
  int num = 1;
  // Inicializando el arreglo
  for(int i = 0; i < TAM; i++) {
    A[i] = 10*num;
    num++;
  }
  // Imprimiendo el arreglo
  printf("A = [ ");
  for(int i = 0; i < TAM; i++) {
    printf("%d ", A[i]);
  }
  printf("]");
  return 0;
}
```

FUNCIONES Y ARRAYS

- Es posible pasar arreglos como argumentos de funciones.

Item	Forma

Definición de la función de la función	`return_type function_name (data type array[],...) {`
	`  local declarations;`
	`  function statements;`
	`}`

Declaración de la función	`return_type function_name (data type arrayParam[],...);`
Invocación de la función	`[return_type var = ] function_name (arrayArg[],...);`

Item	Forma

Definición de la función de la función	`return_type function_name (data type array[],...) {`
	`  local declarations;`
	`  function statements;`
	`}`

Declaración de la función	`return_type function_name (data type arrayParam[],...);`
Invocación de la función	`[return_type var = ] function_name (arrayArg[],...);`

Ejemplo: Caso con una función para imprimir un vector.

Declación	Declaración	Invocación

`void imprimirVector(int V[],int tam)`
`{`
`  printf("[ ");`
`  for(int i = 0; i < tam; i++) {`
`  printf("%d ", V[i]);`
`  }`
`  printf("]\n");`
`}`
	`void imprimirVector(int [],int);`	`int A[5] = {-1,10,4,8,33};`
		`imprimirVector(A,5);`

APUNTADORES Y ARREGLOS

- Como las variables sencillas, los apuntadores también pueden ser empleados para manipular arreglos

```c
#include <stdio.h>

int V[4];
int main() {
  printf("%d\n",sizeof(int));
  printf("%p\n",V);
  int *p1 = V;
  *p1 = 3;
  int *p2 = &V[0];
  p2 += 2;
  *p2 = 1;
  p1 = p2 - 1;
  *p1 = -(*p2);
  *(p2 + 1) = 2;
  return 0;
}
```

Codigo online

APUNTADORES Y FUNCIONES

- Un array como tal es un apuntador. Por lo tanto es posible hablar de una equivalencia entre la forma de acceder a los elementos de un vector por medio de apuntadores:
- Para el caso, sea **A** un vector, la siguiente tabla muestra la relación entre la forma de acceso usando notación apuntador o notación vector.

Notación subíndice	Notación apuntador
`&A[0]`	`A`
`&A[i]`	`A + i`
`A[0]`	`*A`
`A[i]`	`*(A + i)`

NOTACIÓN SUBÍNDICE .VS. NOTACIÓN APUNTADOR

Item	Empleando subindices	Empleando apuntadores
Declaración	`void imprimirVector(int V[],int tam);`	`void imprimirVector(int *V,int tam);`

Definición	`void imprimirVector(int V[],int tam) {`
	`  printf("[ ");`
	`  for(int i = 0; i < tam; i++) {`
	`  printf("%d ", V[i]);`
	`  }`
	`  printf("]\n");`
	`}`
		`void imprimirVector(int *V,int tam) {`
		`  printf("[ ");`
		`  for(int i = 0; i < tam; i++) {`
		`  printf("%d ", *(V + i));`
		`  }`
		`  printf("]\n");`
		`}`

Invocación	`int A[] = {1, 2, 3};` `imprimirVector(A, 3);`	`int A[] = {1, 2, 3};` `imprimirVector(A, 3);`

NOTACIÓN SUBÍNDICE .VS. NOTACIÓN APUNTADOR

- Hacer una función que sume todos los elementos de un vector:
  a. Empleando la notación subíndices.
  b. Empleando la notación apuntador.
- Pruebe el correcto funcionamiento de está pasando un vector arbitrario.
- Para imprimir puede emplear la función imprimir Vector previamente analizada.

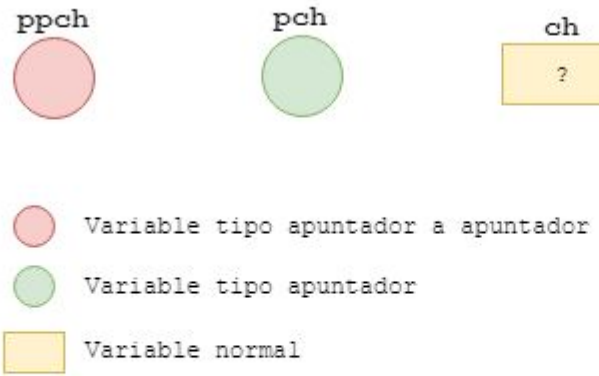
APUNTADORES A APUNTADORES

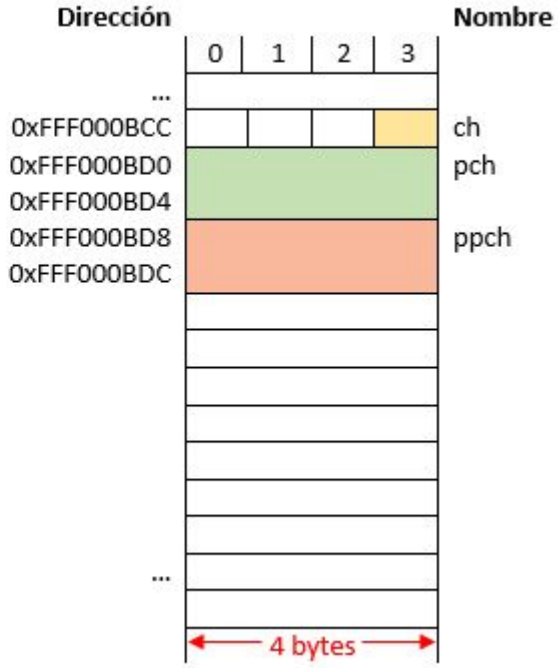
- Es posible poner apuntar un apuntador a un apuntador, lo cual se indica con la cantidad de asteriscos colocados en la declaración del apuntador, así la declaración realizada en las siguientes líneas de código:

```
char ch; /*Un caracter*/
char *pch; /*Un apuntado a un dato tipo caracter*/
char **pch; /*Un apuntador a un apuntador a un caracter*/
```

APUNTADORES A APUNTADORES

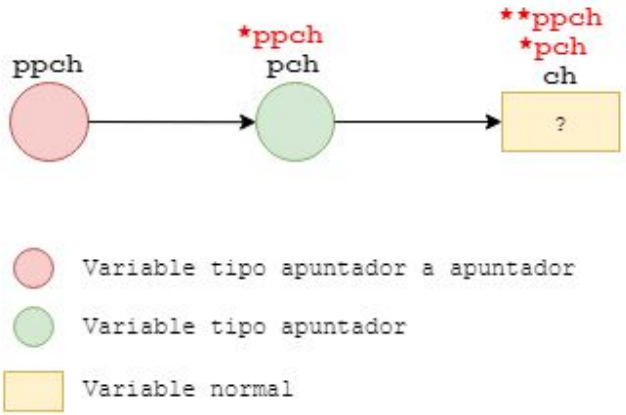
```
//instrucciones
char ch;
char *pch;
char **ppch;
```

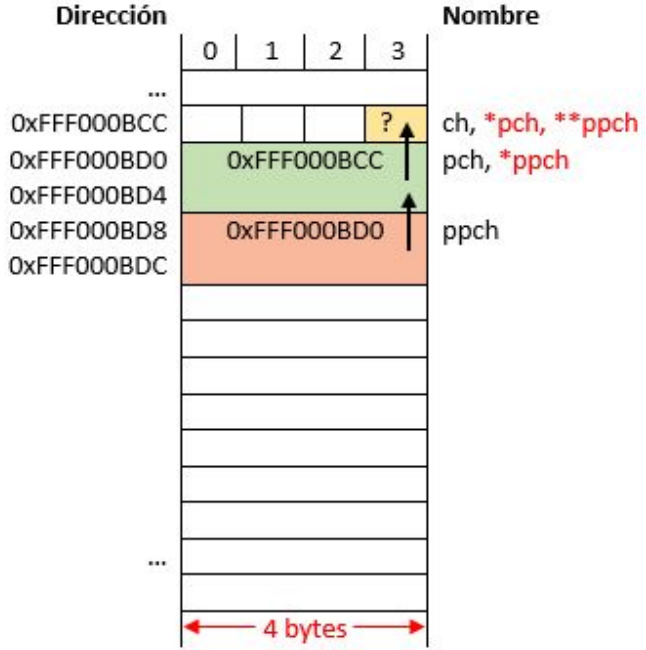




APUNTADORES A APUNTADORES

```
//instrucciones
...
pch = &ch;
ppch = &pch;
```





APUNTADORES A APUNTADORES

```c
#include <stdio.h>

int main() {
  char ch;
  char *pch, **ppch;
  char ***pppch = &ppch;
  pch = &ch;
  ppch = &pch;
  ***pppch = 'A';
  **ppch = *pch + 1;
  ch = **ppch + 3;
  return 0;
}
```

Ejecución online

APUNTADORES A VOID

- Un apuntador genérico o void pointer es un tipo especial de apuntador que puede apuntar a cualquier tipo de dato.
- No puede ser desreferenciado directamente por lo que es necesario un casting para hacer que el apuntador generico pueda apuntar a un tipo de dato concreto (el cual si puede ser referenciado).
- En si el cast es de la forma:

```
((tipo *)ptr)
```

```
/* Declaracion de variables*/
  tipo1 var1_1, var1_2,...;
  ...
  tipoN varN 1, varN 2,...;
  /* Declaracion apuntador generico*/
  void *ptr;
  /* Referencia a una variable tipo tipo1 */
  ptr = &var1 1; // var es una variable
                 // de cualquier tipo
  /* Desreferencia a una variable tipo tipo1*/
  var1 2 = *((tipo1 *)ptr); // cast
  /* Referencia a una variable tipo tipoN */
  ptr = &varN 1; // var es una variable
                 // de cualquier tipo
  /* Desreferencia a una variable tipo tipo1*/
  varN_2 = *((tipoN *)ptr); // cast
  ...
```

APUNTADORES A VOID

```c
#include <stdio.h>

int main() {
  int a = 5;
  double b = 3.1415;
  void *vp;
  vp = &a;
  printf("a = %d\n", *((int *)vp)); // Cast a (int *)
  vp = &b;
  printf("b = %lf\n", *((double *)vp)); // Cast (double *)
  return 0;
}
```

Ejecución online