C programming Language

Chapter 3

1. Pointers

What is a Pointer?

- A pointer is a variable that contains the address of a variable.
- A Pointer (to type) can be associated with a **type**.
- We are familiar with: **`int`**, **`float`**, **`char`**, … types.
- Each type has a comparable pointer (to) type:

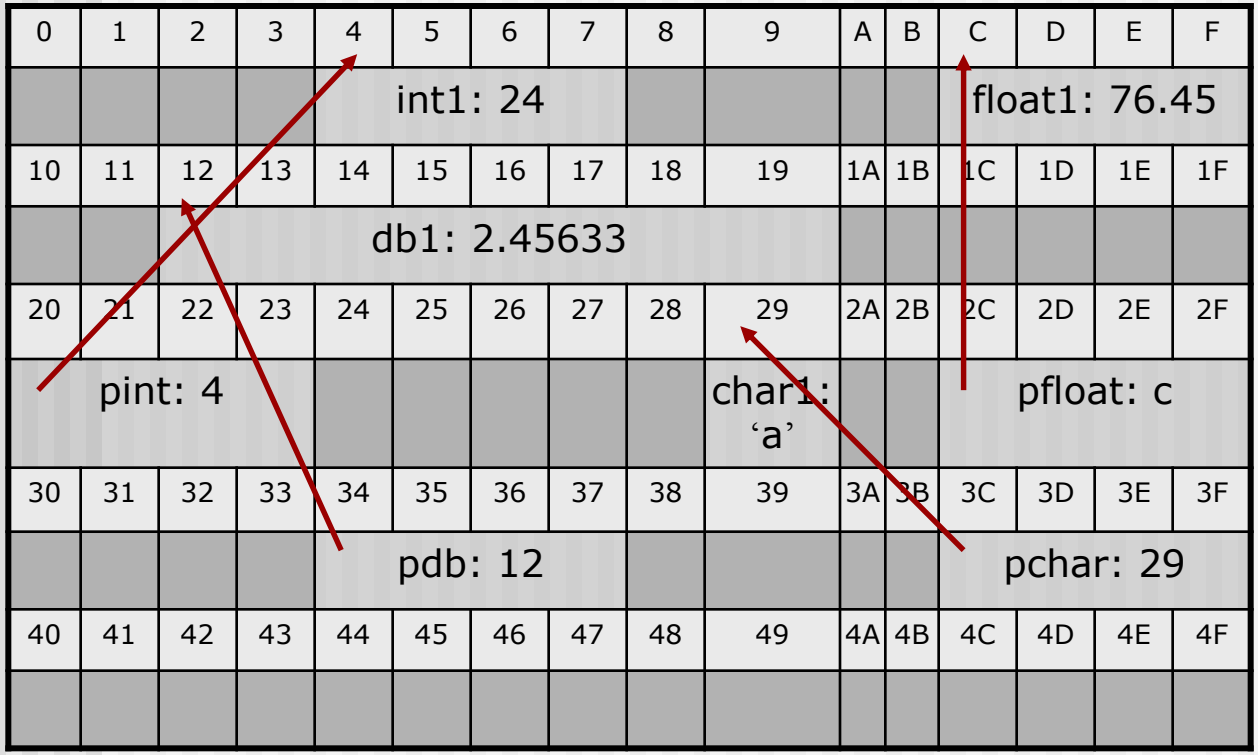
  `int *`, **`float *`**, **`char *`**, etc.
- The difference: while primitive types store values, the pointer types store addresses. For example:
  - int * stores an address of int.
  - char * stores an address of char.
  - float * stores an address of float.
- Pointer size is 4 bytes (independently of its type).

Operators * and &

C defines two operators: `&` (address of) and `*` (indirection) for use with pointers:

- **Indirection Operator** `*` has 2 different meanings:
  - Upon declaration – "I am a pointer".
  - After declaration – access the variable whose address is held by the pointer.
- **Address Operator** `&` provides the address of a variable.

Pointers Example




```
void main()
{
  int int1=24;
  float float1=76.45;
  double db1=2.45633;
  char char1='a';
  double *pdb=&db1;
  char *pchar=&char1;
  int *pint;
  float *pfloat;
  pint = &int1;
  pfloat = &float1;
}
```

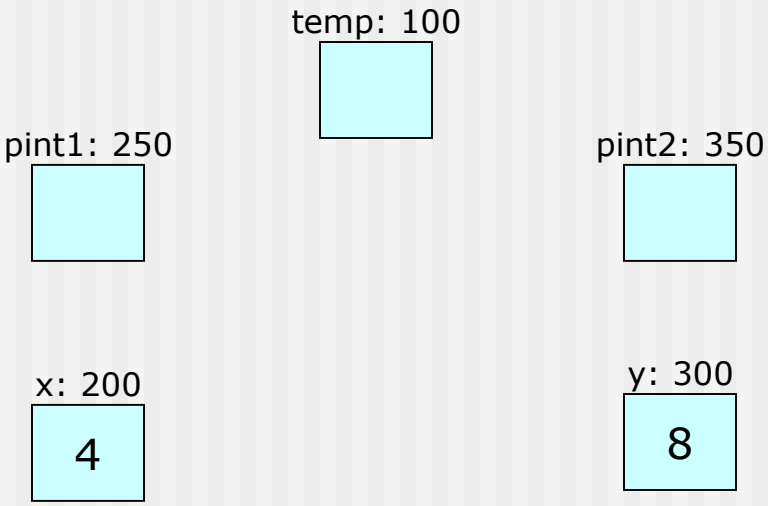
Example

Swap two integer values through use of pointers

```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  *pint1 = *pint2;
  *pint2 = temp;
}
```

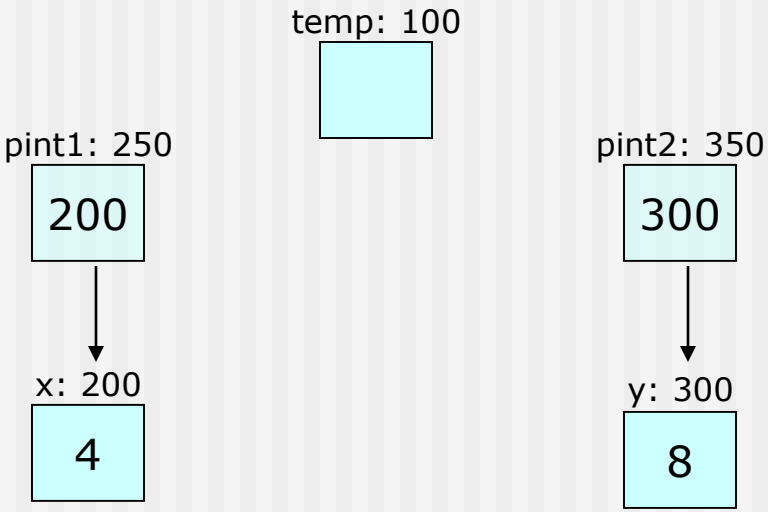
Example




```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  *pint1 = *pint2;
  *pint2 = temp;
}
```

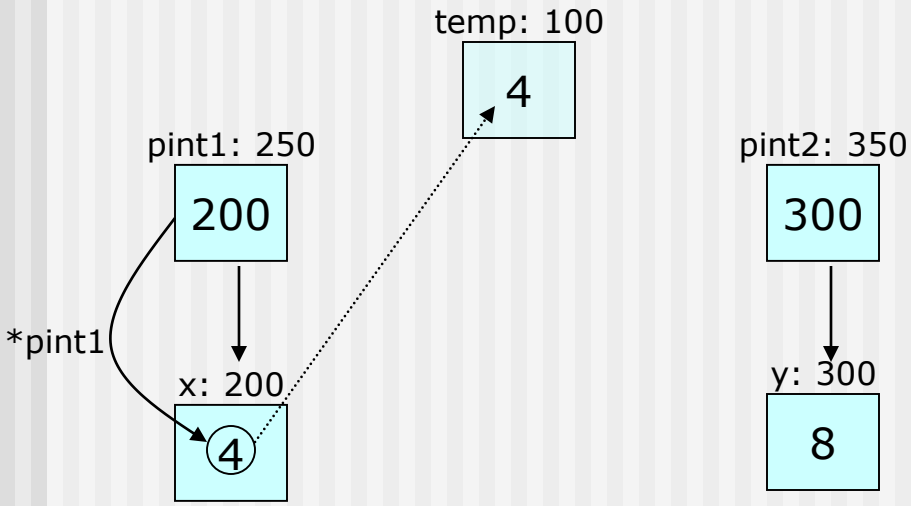
Example




```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  *pint1 = *pint2;
  *pint2 = temp;
}
```

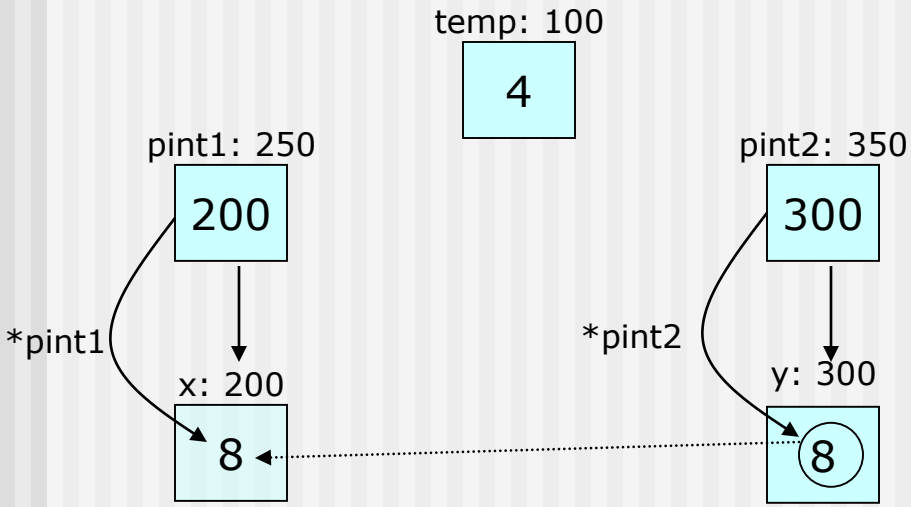
Example




```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  *pint1 = *pint2;
  *pint2 = temp;
}
```

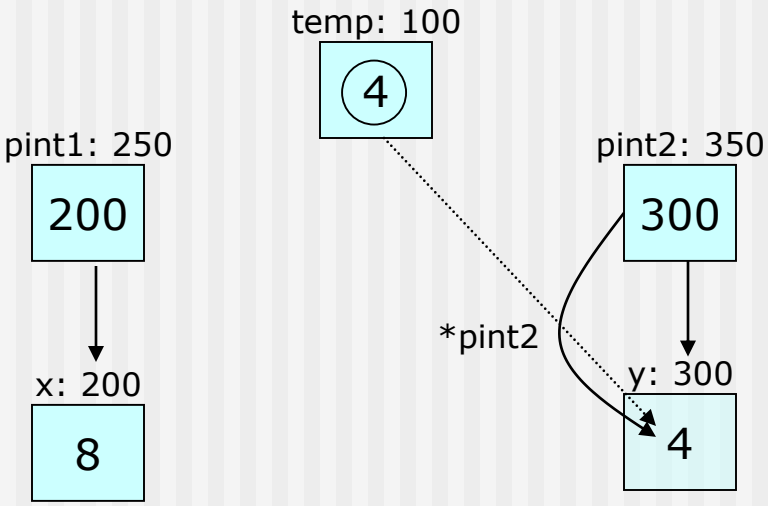
Example




```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  *pint1 = *pint2;
  *pint2 = temp;
}
```

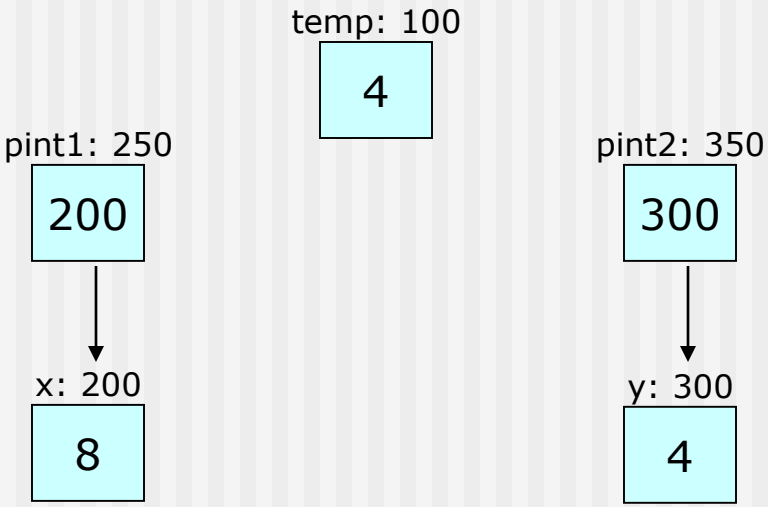
Example




```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  *pint1 = *pint2;
  *pint2 = temp;
}
```

Example

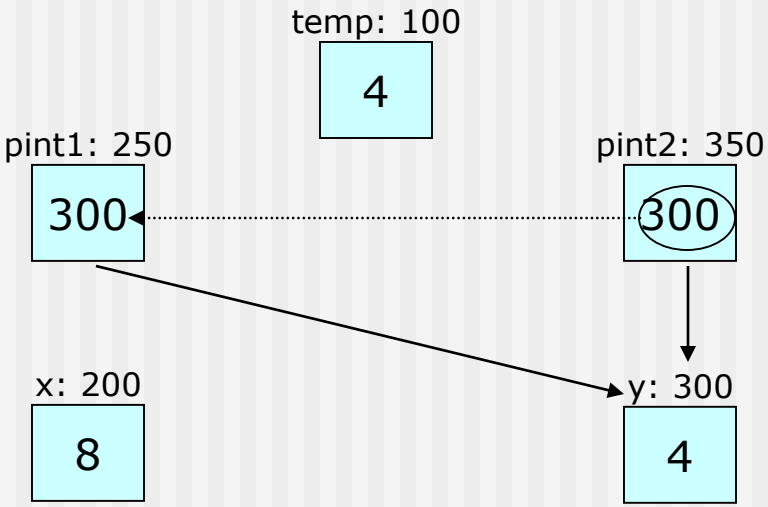



```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  pint1 = pint2;
  *pint2 = temp;
}
```

What happens in this case?

Example




```
void main()
{
  int x=4, y=8, temp;
  int *pint1,*pint2;
  pint1 = &x;
  pint2 = &y;

  temp = *pint1;
  pint1 = pint2;
  *pint2 = temp;
}
```

This is a pointer assignment!

Be careful

```
int x = 4;
int *px;
px = x;
```

Error compilation:
Cannot convert from int to int *

```
int x = 4;
int *px;
float *pf;
pf = px;
```

BUG!
Cannot convert from int * to float *

```
int *px;
*px = 9;
```

BUG!!!
px points to garbage - trying to assign a value to memory not allocated

Call by Address

- Passing argument by value:

  the argument is copied to the local parameter of the function.
- Passing argument by address:
  only the address of the argument is passed (a local parameter stores the address). So the operator * enables access to the value of the argument.

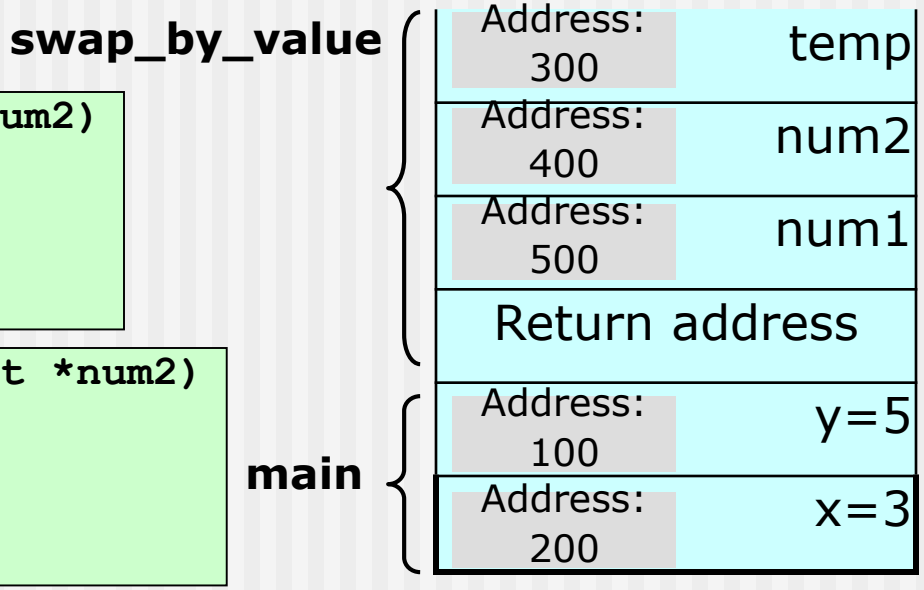
Call by Value Example

swap gets x and y
and swaps between them

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
    swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



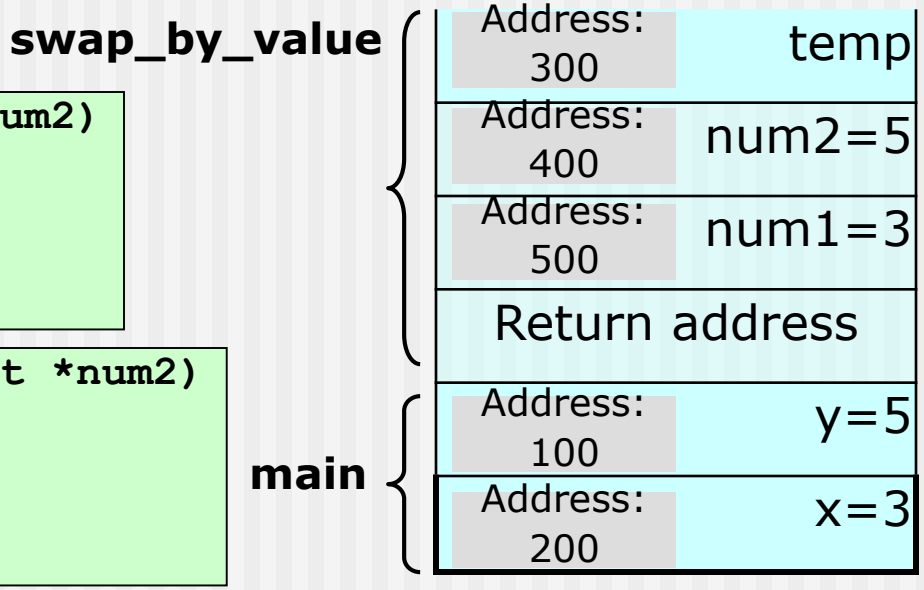
Call by Value Example

Arguments passing

```
void main()
{
    int x=3, y=5;
➔   swap_by_value(x, y);
    swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



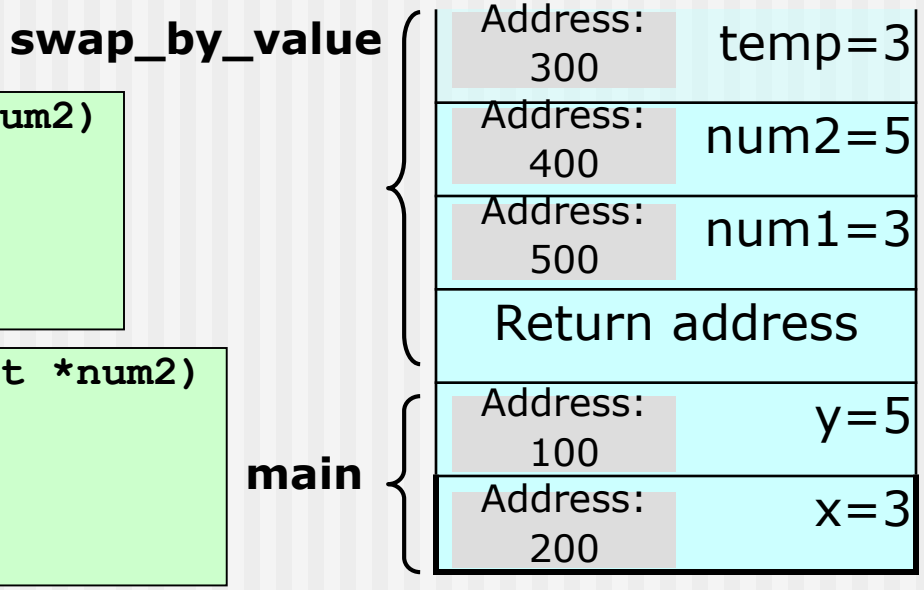
Call by Value Example

temp = num1;

```
void main()
{
    int x=3, y=5;
➔   swap_by_value(x, y);
    swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
➔   int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



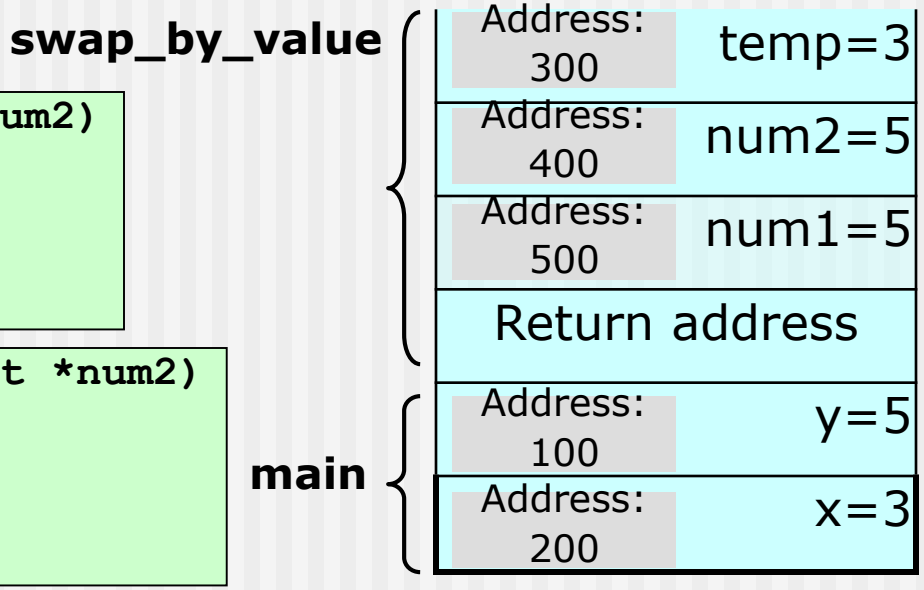
Call by Value Example

num1 = num2;

```
void main()
{
    int x=3, y=5;
➔   swap_by_value(x, y);
    swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
➔   num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



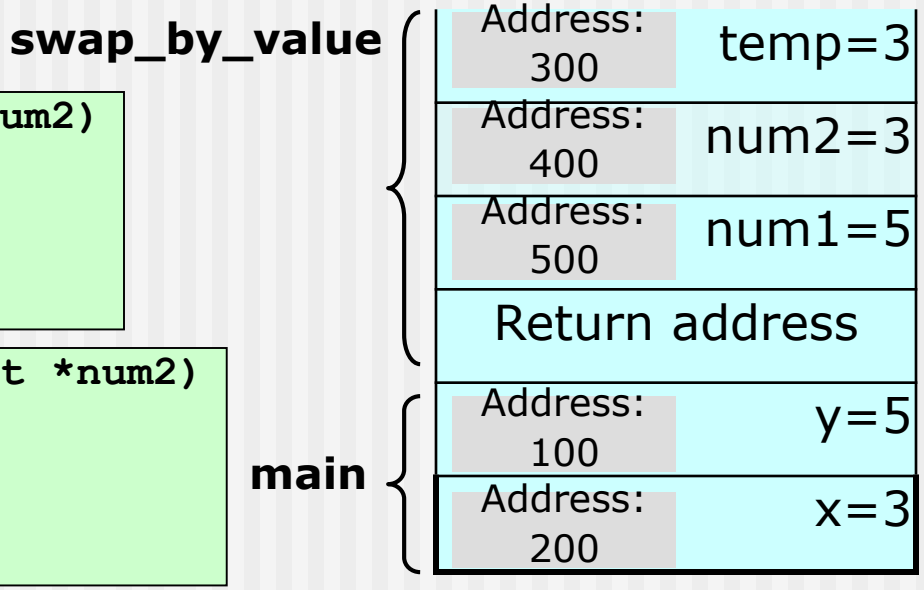
Call by Value Example

num2 = temp;

```
void main()
{
    int x=3, y=5;
➔   swap_by_value(x, y);
    swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
➔   num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



Call by Value Example

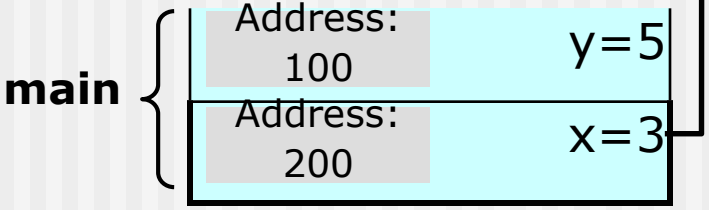
After swap_by_value has returned

Note that x and y **weren't** changed

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
➔   swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



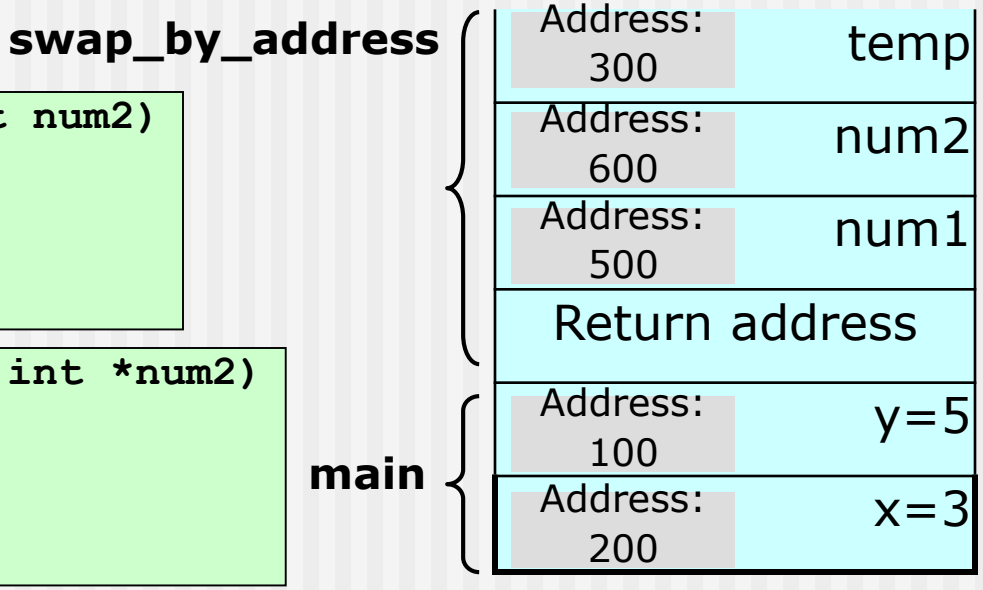
Call by Address Example

swap gets x and y
and swaps between them

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
➔   swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



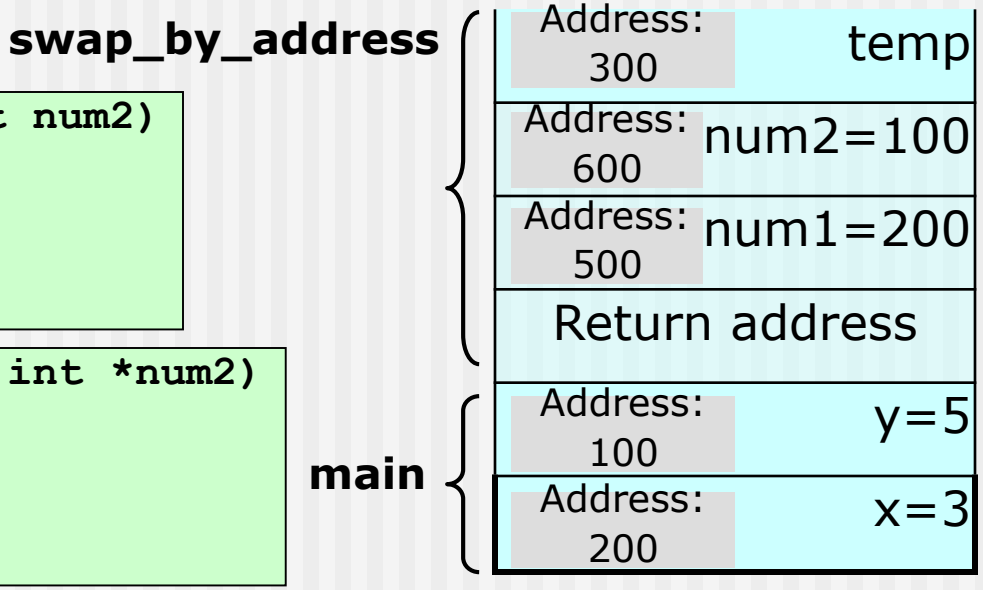
Call by Address Example

Arguments passing

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
➔   swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



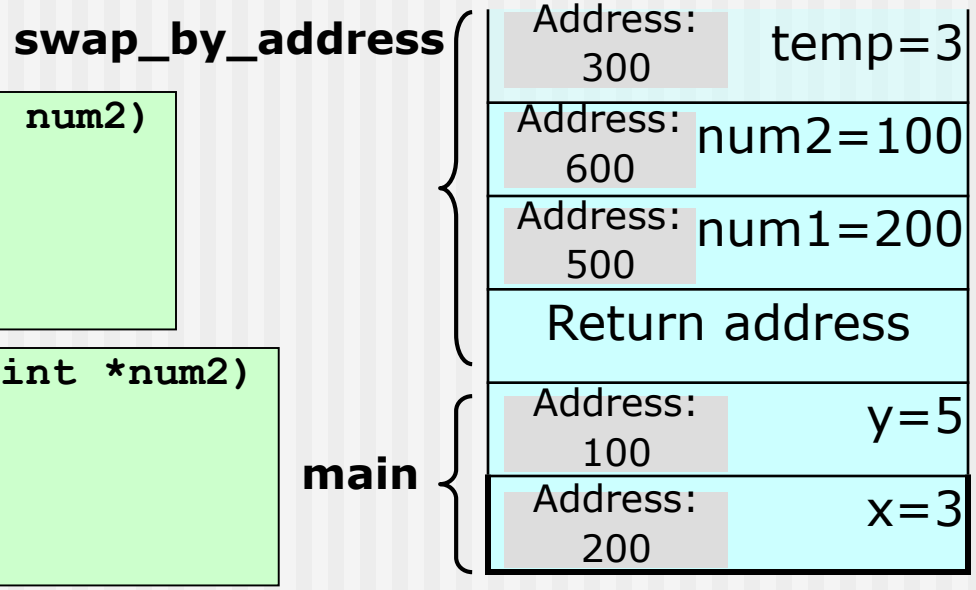
Call by Address Example

temp = *num1;

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
➔   swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
➔   int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



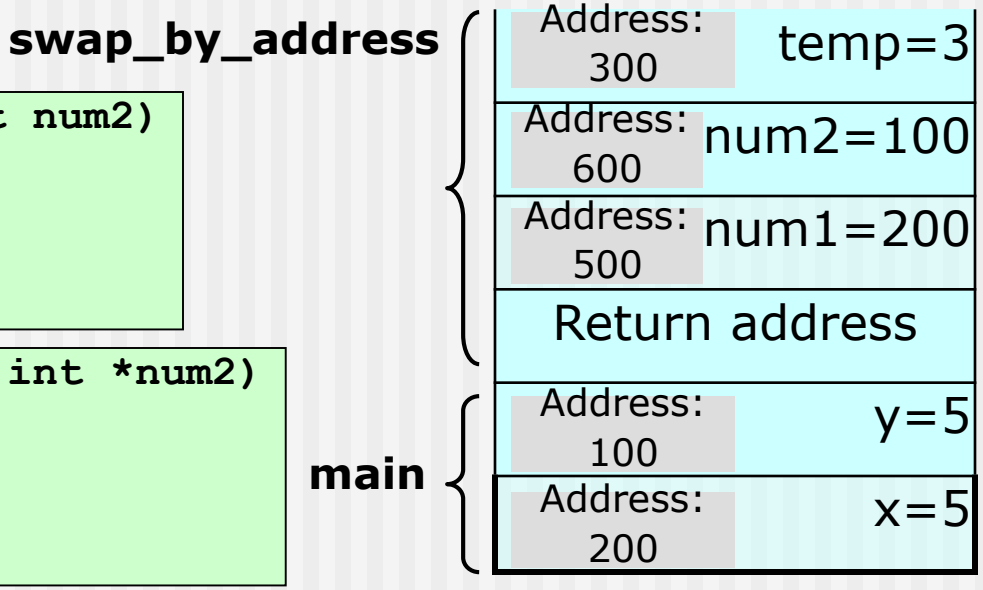
Call by Address Example

*num1 = *num2;

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
➔   swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
➔   *num1 = *num2;
    *num2 = temp;
}
```



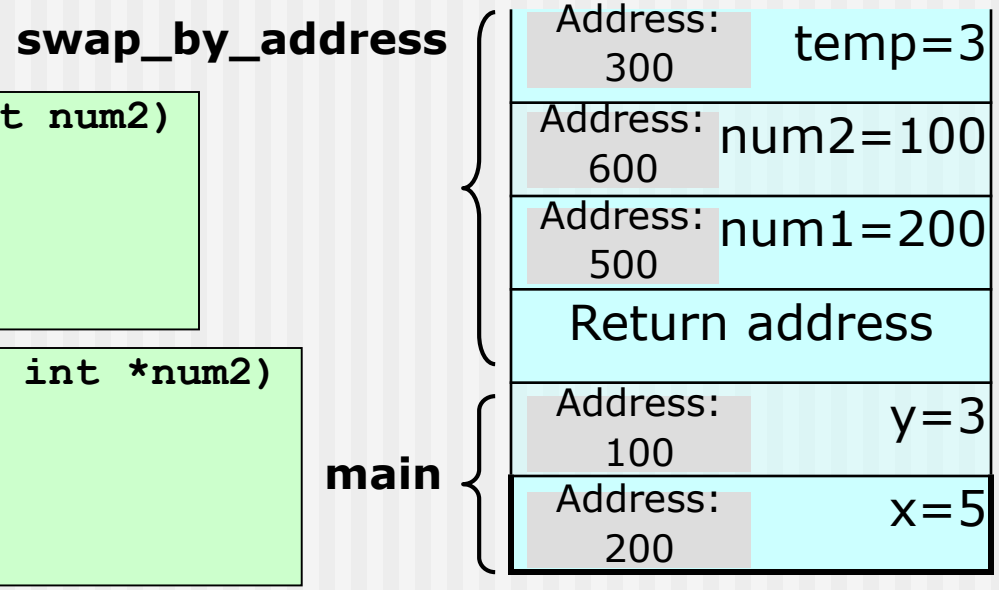
Call by Address Example

*num2 = temp;

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
➔   swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
➔   *num2 = temp;
}
```



Call by Address Example

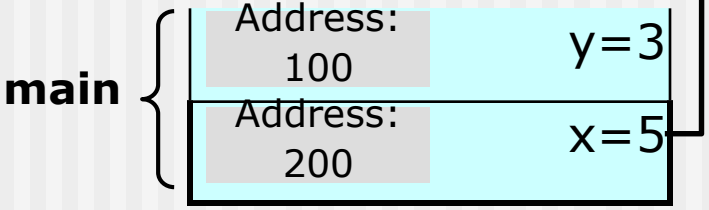
After swap_by_value has returned

Note that x and y were **changed**

```
void main()
{
    int x=3, y=5;
    swap_by_value(x, y);
    swap_by_address(&x, &y);
}
```

```
void swap_by_value(int num1, int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

```
void swap_by_address(int *num1, int *num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```



Passing Arguments - Summary Table

	By value	By address (of array)
Declaration	func(type name)	func(type name[]) func(type *name)
Scope	In function	In function
Lifetime	In function	Address: lifetime only in function Array's elements: original lifetime
Argument changeable	No	Address: no Array's elements: yes

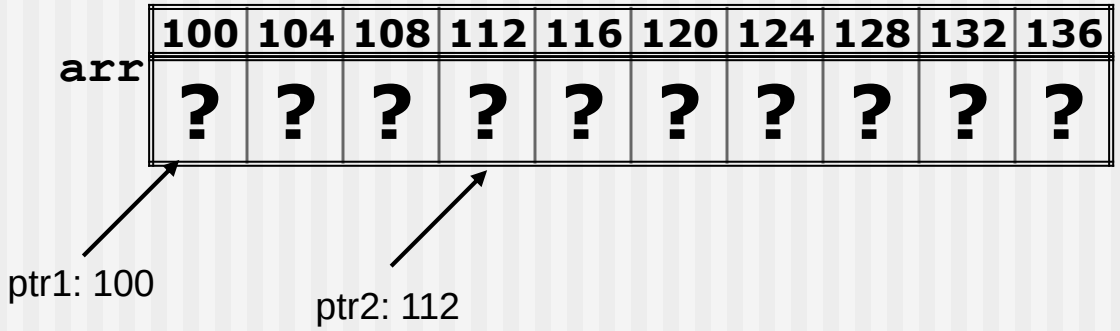
Pointers and Arrays

- The meaning of assigning an array to a pointer is assigning the first array address (by array name) into the pointer:
  - **int array[5];**
  - **int *parr;**
  - **parr = array; // same as parr = &array[0];**
- Scanning the array is possible by using:
  - Operator []
  - Operator *
- But first, which pointer operators available?

Pointer Arithmetic Operators

- There are some arithmetic operations that can be performed on pointers.
- Add/subtract an integer to/from a pointer:

```
int arr[10];
int *ptr1 = arr, *ptr2;
ptr2 = ptr1 + 3; //ptr1+3*sizeof(int)
```



Conclusion:
On adding an integer to a pointer, the new address is determined according to sizeof of the pointed type.

Pointer Relational Operators

- Comparing pointers:
  - Two pointers can be compared if they point to the same type.
  - Any pointer may be compared to the NULL (zero) pointer.

    Examples:

```
int *p1 = NULL, *p2 = NULL;
if (p1 == p2) …
if (p1 < p2) …
if (p1 != NULL) …
```

- **A good programmer verifies the validity of a pointer before using the indirection operator * on it.**

Operator [] and Operator *

- As seen before, we may declare:

```
int a[100];
int *p = a;
```

- The following expressions (in each line) have the same meaning:
  - `a,&a[0],p,&p[0]` address of the 1st element
  - `*a,a[0],*p,p[0]` value of the 1st element
  - `a+1,&a[1],p+1,&p[1]` address of the 2nd element
  - `*(a+1),a[1],*(p+1),p[1]` value of the 2nd element
  - `a+i,&a[i],p+i,&p[i]` address of the i-th element
  - `*(a+i),a[i],*(p+i),p[i]` value of the i-th element

Operator [] and Operator *

So what is the difference between operator [] and operator * ???

```
int arr[100];
int *parr = arr;
arr++;
arr += 5;

parr++;
parr += 5;
```

Error in compilation

Perfectly legal

Note: The array name isn't a variable!!!

Operator ++

- Given the declarations:

```
int k, *ptr = NULL;
int Arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
ptr = Arr;
```

What is the meaning of `k = *ptr++;` ?

- `k = *ptr++` is equivalent to the following 2 statements:
  - `k = *ptr; ptr++;`
- What is the meaning of :
  - `k = (*ptr)++;`
  - `k = *(ptr++);`
  - `k = *++ptr;`
  - `k = ++*ptr;`
- Try to avoid such unreadable code, even it works!

Scanning Arrays using Pointers

- Scanning arrays using pointers is faster than scanning them using indices. Here are 2 functions that perform the same job.

1. Uses indices to sum the n first elements of array v:

```
long scan_with_indices(int v[], int n)
{
    int i = 0;                  // The operations
    long total = 0;

    while(i < n)                // <
    {
        total += v[i];          // += *(v + i*sizeof(int))
        ++i;                    // ++
    }
    return total;
}
```

Scanning Arrays using Pointers

2. Uses pointers to sum n first elements of array v.

```
long scan_with_pointers(int *v, int n)
{
        long total = 0;          // The operations
        int *save_end = v+n;
        while(v < save_end)      // <
        {
                total += *v;     // += *(indirection)
                ++v;             // ++
        }
        return total;
}
```

- **Conclusion**: Each iteration of scanning with indices involves a multiplying action and an addition action that are not necessary when scanning with pointers - more efficient!

Array of Pointers

- Array of pointers is an array whose elements are pointer types.
- Definition:

```
int *arr[5];        // array of 5 int pointers
```

- This is necessary for 2D-array handling by pointers.
- For instance:
  - Assume array of strings is defined. Manipulations on the array is preferable by pointers than by array indexing itself.
  - For example: reverse the order of the strings.

Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```

pnames: 200

200	?
204	?
208	?

names: 100

D	A	N	A	\0
R	A	N	I	\0
S	H	I	R	\0

?

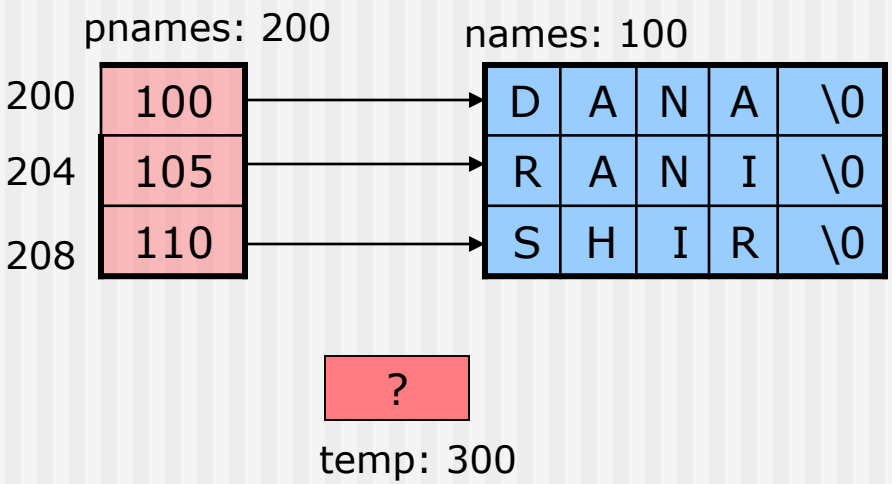
temp: 300

Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```

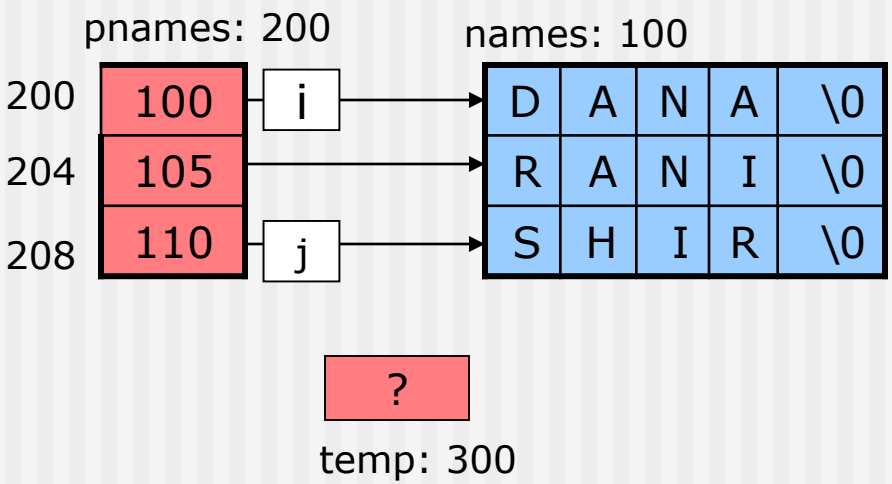


Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```

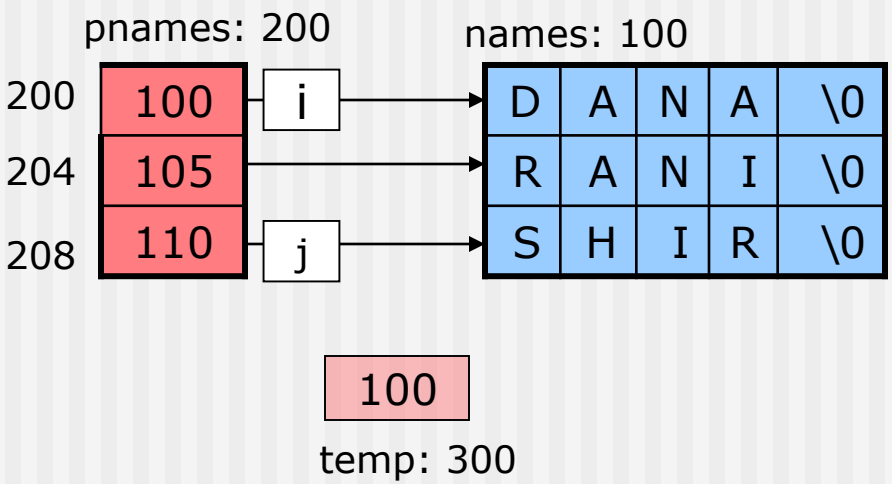


Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```

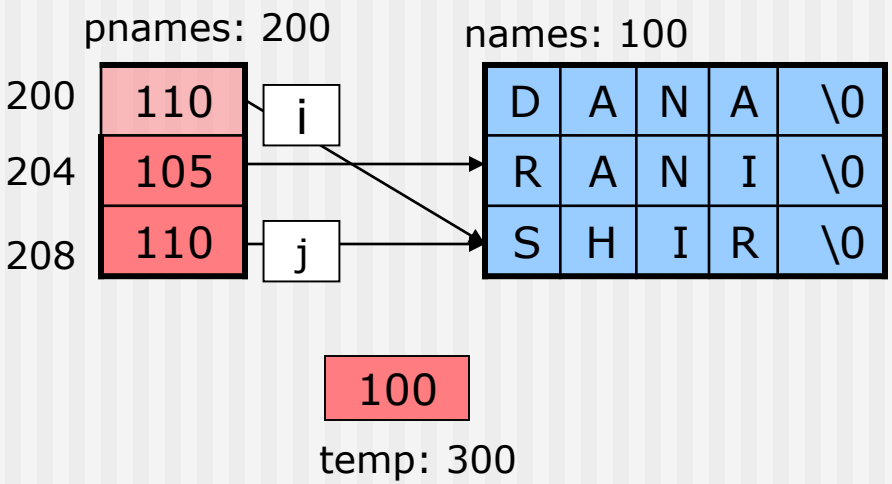


Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```

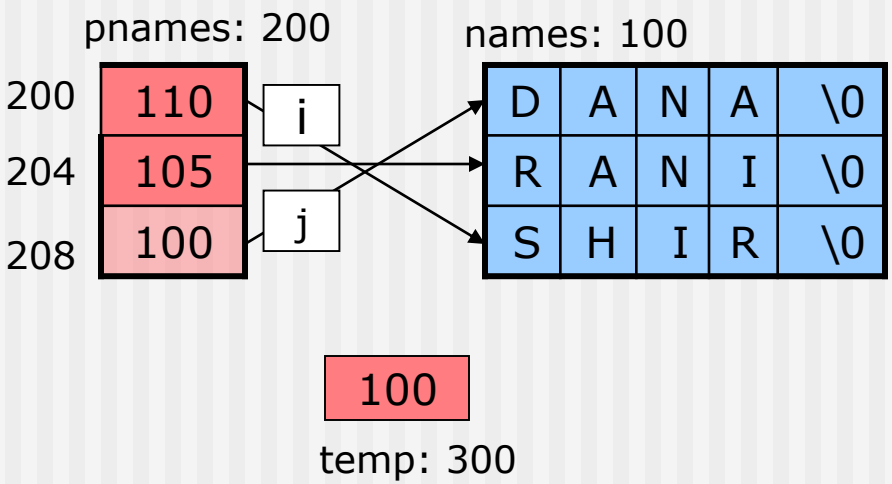


Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```

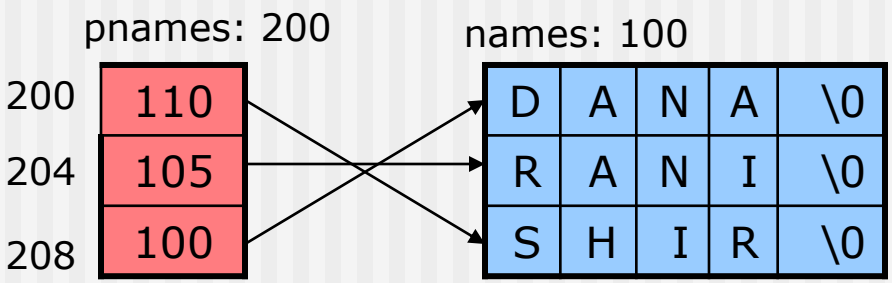


Array of Pointers Example

```
void main()
{
  char names[3][5] = {"DANA",
                      "RANI","SHIR"};
  char *pnames[3], *temp;
  int i, j;
  for(i=0; i<3; i++)
    pnames[i] = names[i];

  for(i=0,j=3-1; i<j; i++,j--)
  {
    temp = pnames[i];
    pnames[i] = pnames[j];
    pnames[j] = temp;
  }

  for(i=0; i<3; i++)
    printf("%s\n", pnames[i]);
}
```



Output:
SHIR
RANI
DANA