# Warning

# CEng-140

Pointers

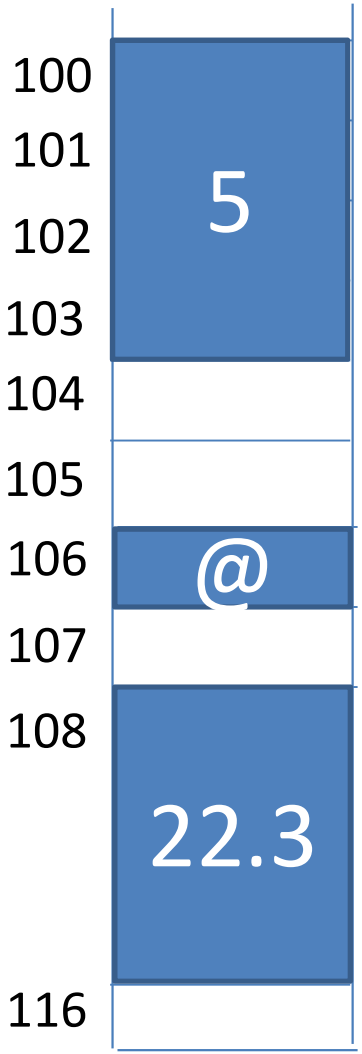
**Dr. Ismail Sengor ALTINGOVDE**

**METU**

# Recall

- Memory can be visualized as an ordered sequence of consecutively numbered storage locations (bytes).
- A data item is stored in one or more adjacent storage locations depending upon its type.
- The address of a data item is the address of its **first** storage location.

- The address of a data item is the address of its **first** storage location.




x

c

grade

Suppose: int 4 bytes

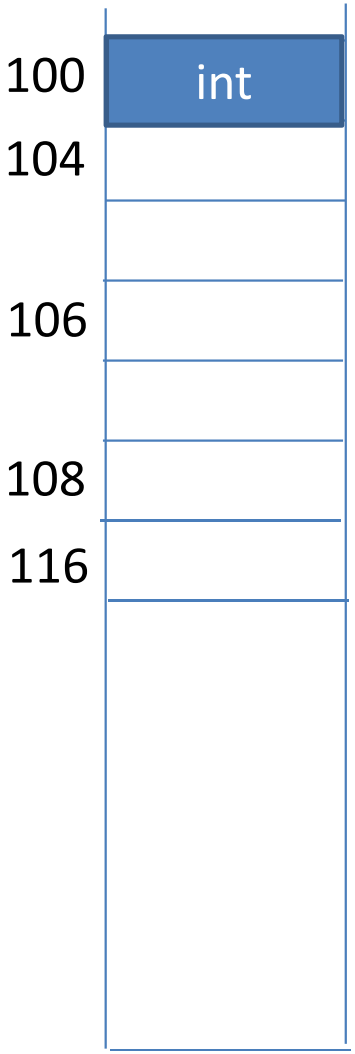
double 8 bytes

char 1 byte

int x = 5;

char c = '@';

double grade = 22.3;




x

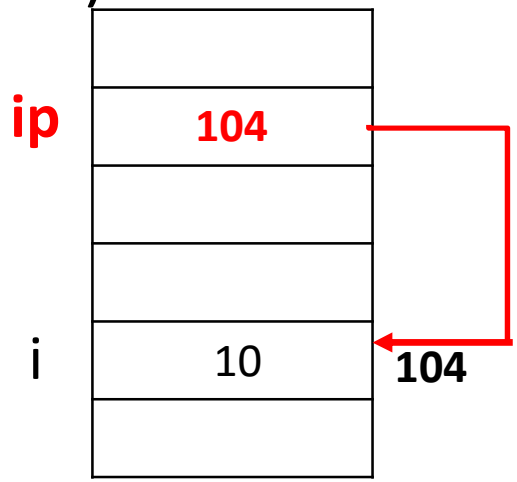
c

grade

# Pointers

- The address of a data item can be stored in another data item and manipulated
  - The **address of a data item** is called a **pointer** to the data item
  - A **variable** that **holds an address** is called **a pointer variable** (Note: a pointer to a **certain type** of data item - variable)

```
/* typical int var */
int i = 10;
```




ip is a pointer variable:
it holds a pointer to
an (data item of type) integer
(it stores the address of an
(data item of type) integer)

# Why do we need pointers?

- To overcome pass-by-value
- To handle dynamic memory and to implement dynamic data structures

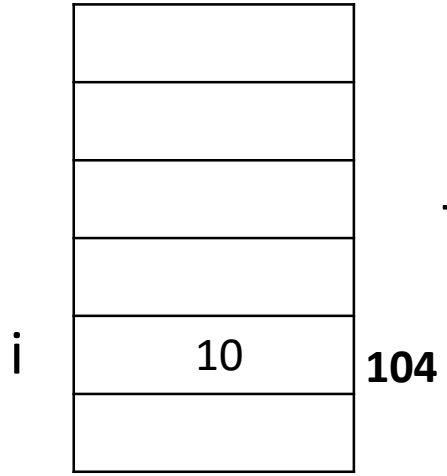
# Operators

- Two unary operators to manipulate data using pointers
- **& (the address of)** → when applied to a variable, gives its address (pointer to var)
- *** (the value at the address)** → when applied to a pointer, fetches the value at that address

# & operator

- & (the address of) → when applied to a variable, gives its address (pointer to var)
- Can **only** be applied to an **l-value**
  - &10  &(x+3)  &'C' → all **wrong!**
- If the type of the operand (of &) is T, then type of the result is **"pointer to T"**

```
/* typical int var */
int i = 10;
```



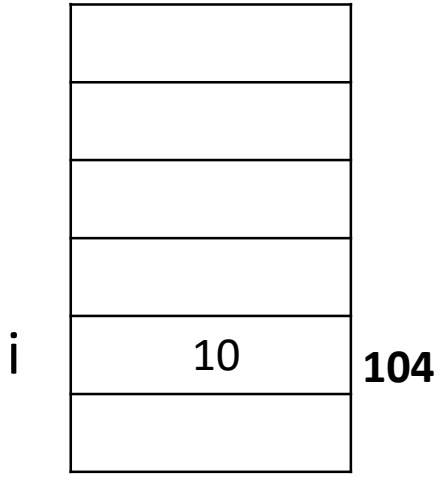

&i → **104**

Type is: **pointer to integer**

# * Operator

- * (the value at the address) → when applied to a pointer, fetches the value at that address
- The * operator can only be applied to a pointer
- If the type of the operand (of *) is "pointer to T", then type of the result is **T**

```
/* typical int var */
int i = 10;
```




&i → **104**

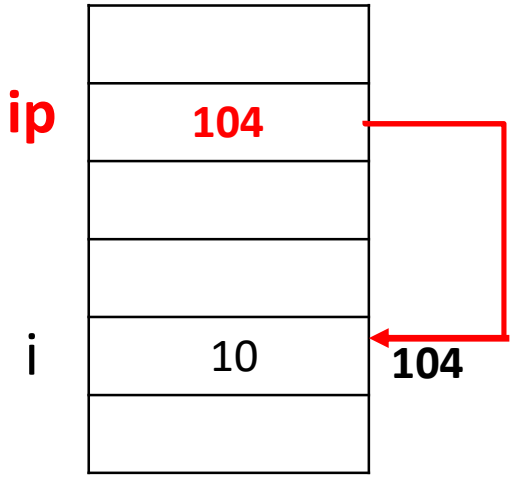
*(&i) → 10

Type is: **integer**

# * Operator

- Accessing an object **via** a pointer is called **dereferencing**
  - If **ip** is a pointer to integer i, *ip is of type integer, and it can be used in every exp instead of i

```
/* typical int var */
int i = 10;
```




```
printf("%d", *ip);

*ip = 20;
```

# Precedence & Associativity

| Operator | Type | Associativity |
|---|---|---|
| Function call: () | | **Left to right** |
| (type) + - ++ -- **! & *** | Unary | **Right to left** |
| * / % | Binary | Left to right |
| + - | Binary | Left to right |
| < <= > >= | Binary | Left to right |
| == != | Binary | Left to right |
| && | Binary | Left to right |
| \|\| | Binary | Left to right |
| = *= /= %= += -= | Binary | Right to left |
| , | | Left to right |

# Declaration

- For each type of object that can be declared in C, a corresponding type of pointer can be declared.
- To indicate a variable contains a pointer to a specified type of object (rather than the object itself), we use an asterisk before the name of object (i.e., var):

  type ***identifier***;

  declares *identifier* to be of type "pointer to type"
- Decl allocates space for the named pointer variable, but not for what it points to!

# Declaration

int *ip, i, j;

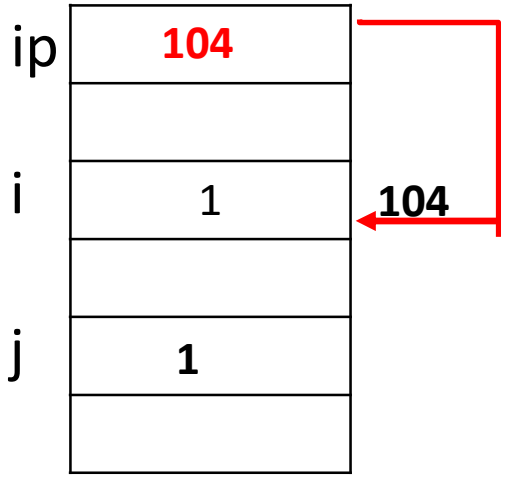
C's philosophy: Decl of a var should follow the form of use

- *ip is of type "int", just like variables i and j
- ip: is of type "pointer to int"

# Assignment

- A pointer value may be assigned to another pointer of the same type

```
int  i=1, j, *ip;
ip = &i;
j = *ip;
```

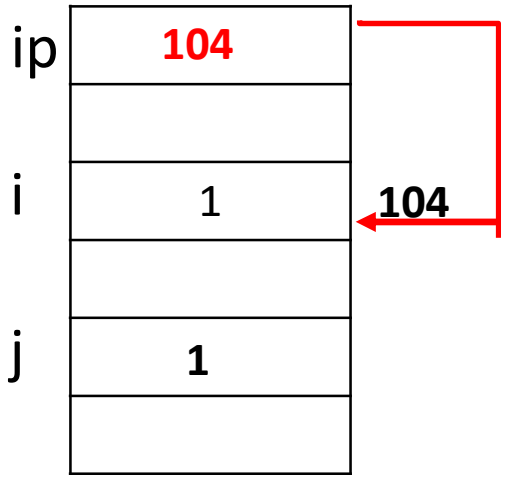



/* j is assigned to the value of i, which is 1*/

# Assignment

- A pointer value may be assigned to another pointer of the same type

```
int  i=1, j, *ip;
ip = &i;
j = *ip;
```




/* j is assigned to the value of i, which is 1*/

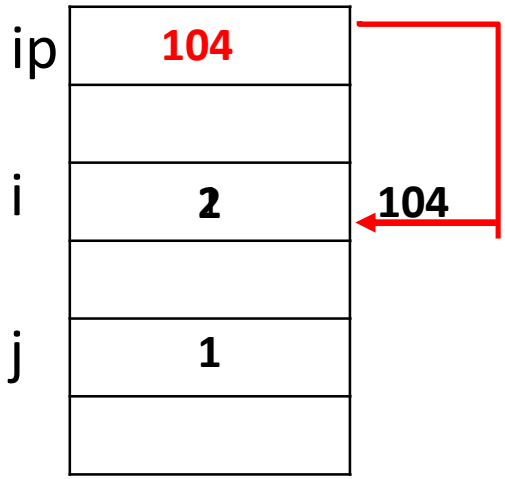
Note: j = *ip; → j = *(&i); → j = i;

**Remark**: The address of (&) operator is the inverse of the dereferencing operator (*).

# Assignment

- A pointer value may be assigned to another pointer of the same type

```
int  i=1, j, *ip;
ip = &i;
j = *ip;
```




```
*ip = 2;  /* value of i becomes 2 */
 /* I modified the value of i without directly
      mentioning i → *ip is an L-VALUE!
```

# Assignment

- A pointer value may be assigned to another pointer of the same type
- REMARK: Pointers and integers are NOT interchangeable.
  - Exception: Constant zero (0) can be assigned to a pointer of any type.
  - A pointer value of 0 is known as **NULL pointer** (stdio.h includes its definition)

# Initialization

- Important as they can initially point to an arbitrary position

  type ***identifier*** = initializer;

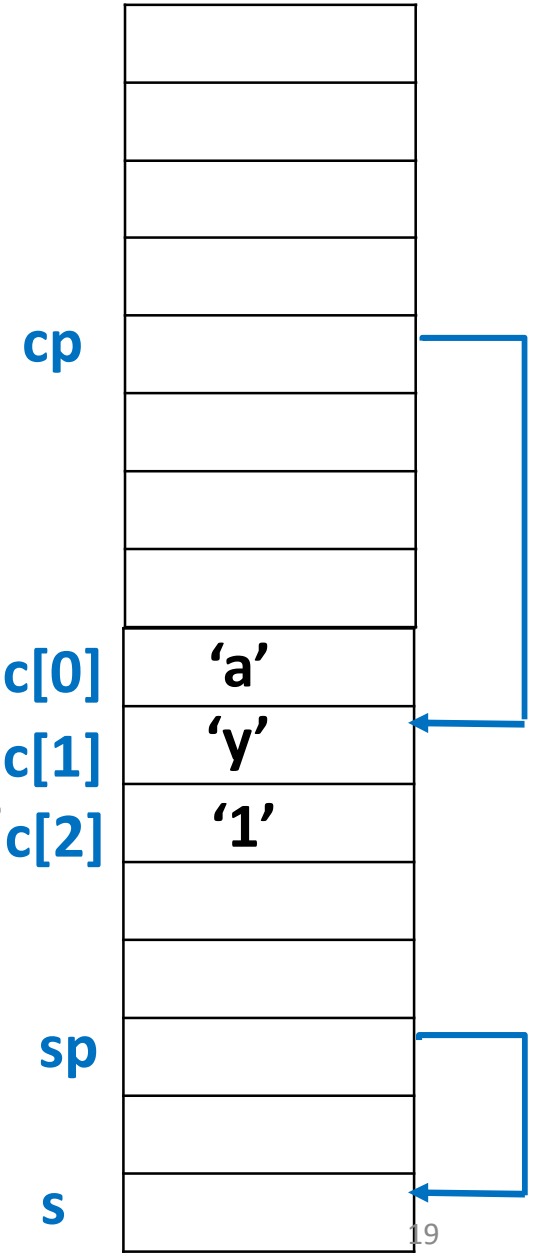
- Initializer.. must either evaluate to an address of previously defined data of appropriate type, or,
- it can be the NULL pointer

# Initialization

```
#include <stdio.h>
float *fp = NULL;
short s;
short *sp = &s;
char c[3];
char *cp = &c[1];
```

**Equivalent to:** ???

```
char *cp;
cp = &c[1];
```

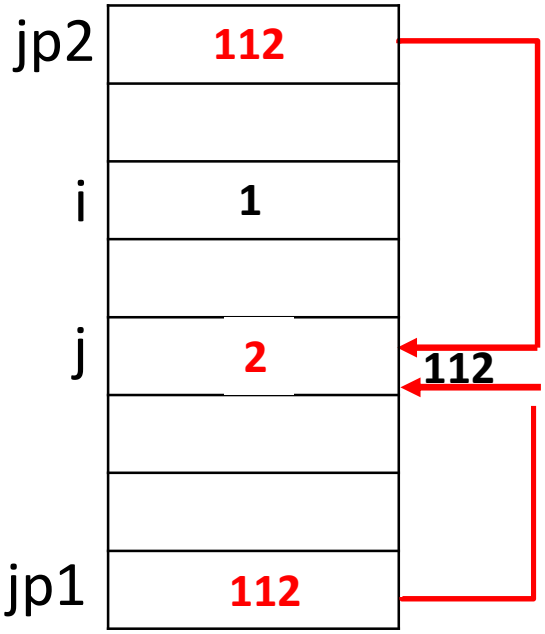





19

# Initialization

```
int i, j = 1;
int *jp1, *jp2 = &j;
jp1 = jp2;
i = *jp1;
*jp2 = *jp1 + 1;
```




```
printf ("%d %d %d %d", i, j, *jp1, *jp2);
```

# Use of Pointers

- A function can take a pointer to any type of data as argument, and can return a pointer to any data type.
- Recall: Parameters are passed by value
  - Values of the arguments are used to initialize the parameters of the called function
  - Any change in the value of a parameter in the called function is NOT reflected in the argument variable in the caller
- Pass by reference: addresses of arguments are supplied to the called function

# Use of Pointers

- We can have the effect of the "call by reference" by passing pointers to variables as arguments to the function

# Example

```
void exchange( int *ip, int *jp)
{ int t;
   t = *ip, *ip = *jp, *jp = t; }

int main(void)
{ int i = 10, j = 20;
   exchange(&i, &j);
   printf ("%d %d", i, j); }
```

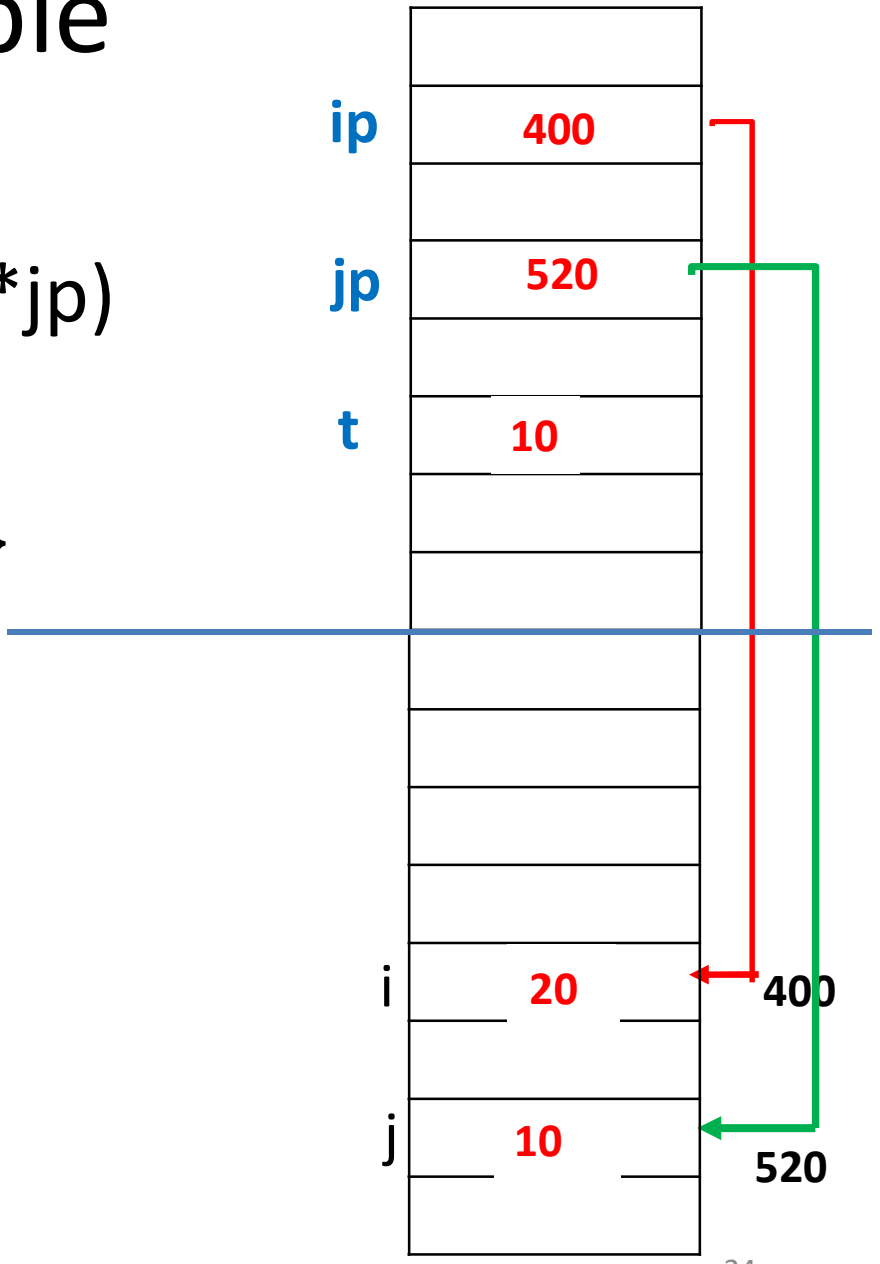
What is the output?

# Example

```
void exchange( int *ip, int *jp)
{ int t;
   t = *ip, *ip = *jp, *jp = t; }

int main(void)
{ int i = 10, j = 20;
   exchange(&i, &j);
   printf ("%d %d", i, j); }
```

What is the output?



# Remark

- When a pointer is passed to a function as arg, the pointer is copied (pass by value, as usual), but the pointed object is not copied.
  - Thus, using the pointer, the called function can access and change the pointed object (in the calling function) (this mimics pass by reference)
  - However, any change to the pointer parameter itself does **not** change the argument pointer, which is consistent with **pass by value** logic

# Example

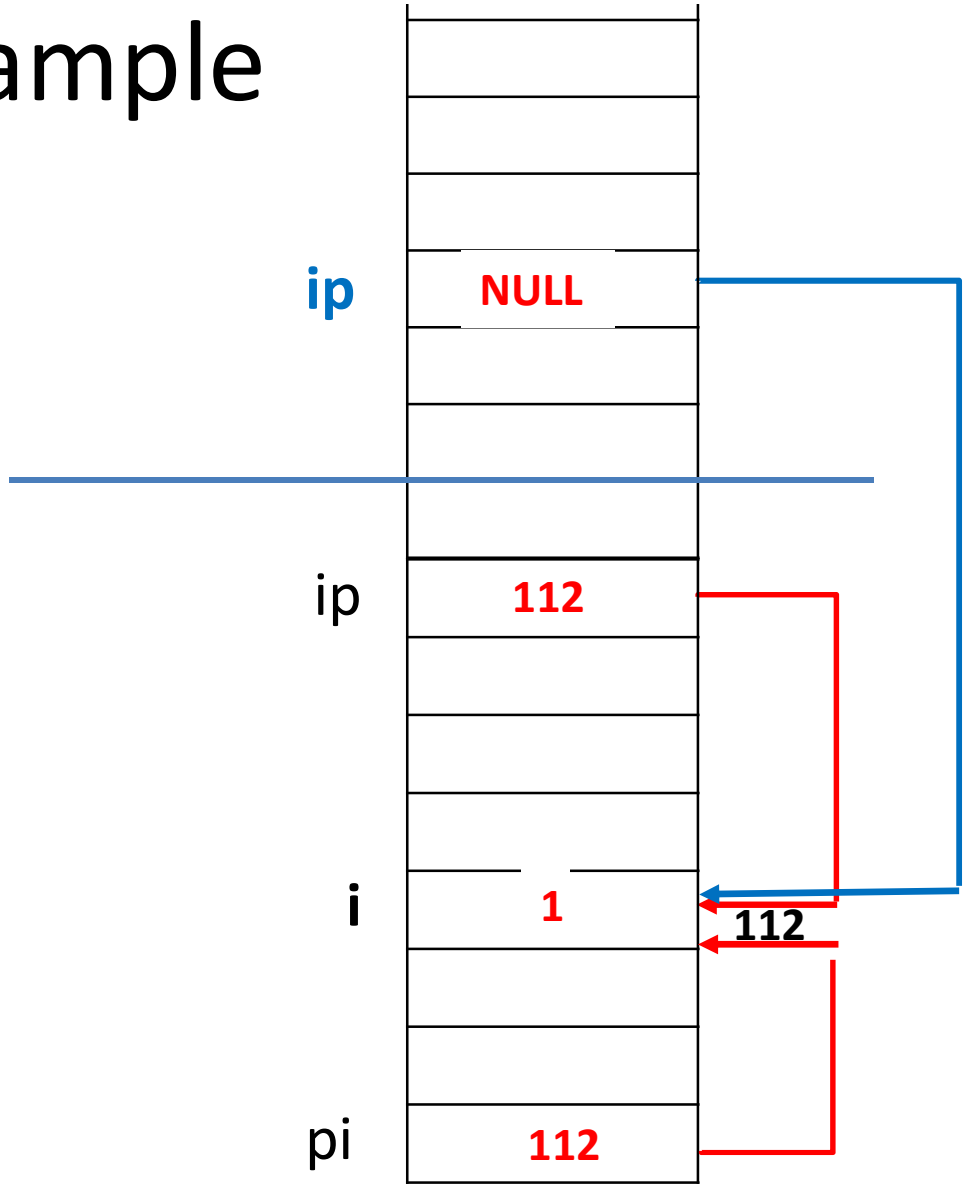
```
void change(int *ip)
{ *ip = *ip +1;
   ip = NULL;}

int main(void)
{  int i=0, *ip, *pi;
   ip = pi = &i;
   change(ip);
   printf ("%d ", i);
   if (ip == pi) printf("NO change in argument ip ");
   else printf("change in argument ip "); }
```

# Example

```
void change(int *ip)
{ *ip = *ip +1;
    ip = NULL;}

int main(void)
{  int i=0, *ip, *pi;
    ip = pi = &i;
    change(ip);
    printf ("%d ", i);
    if (ip == pi) printf("no change in argument ip "); }
```

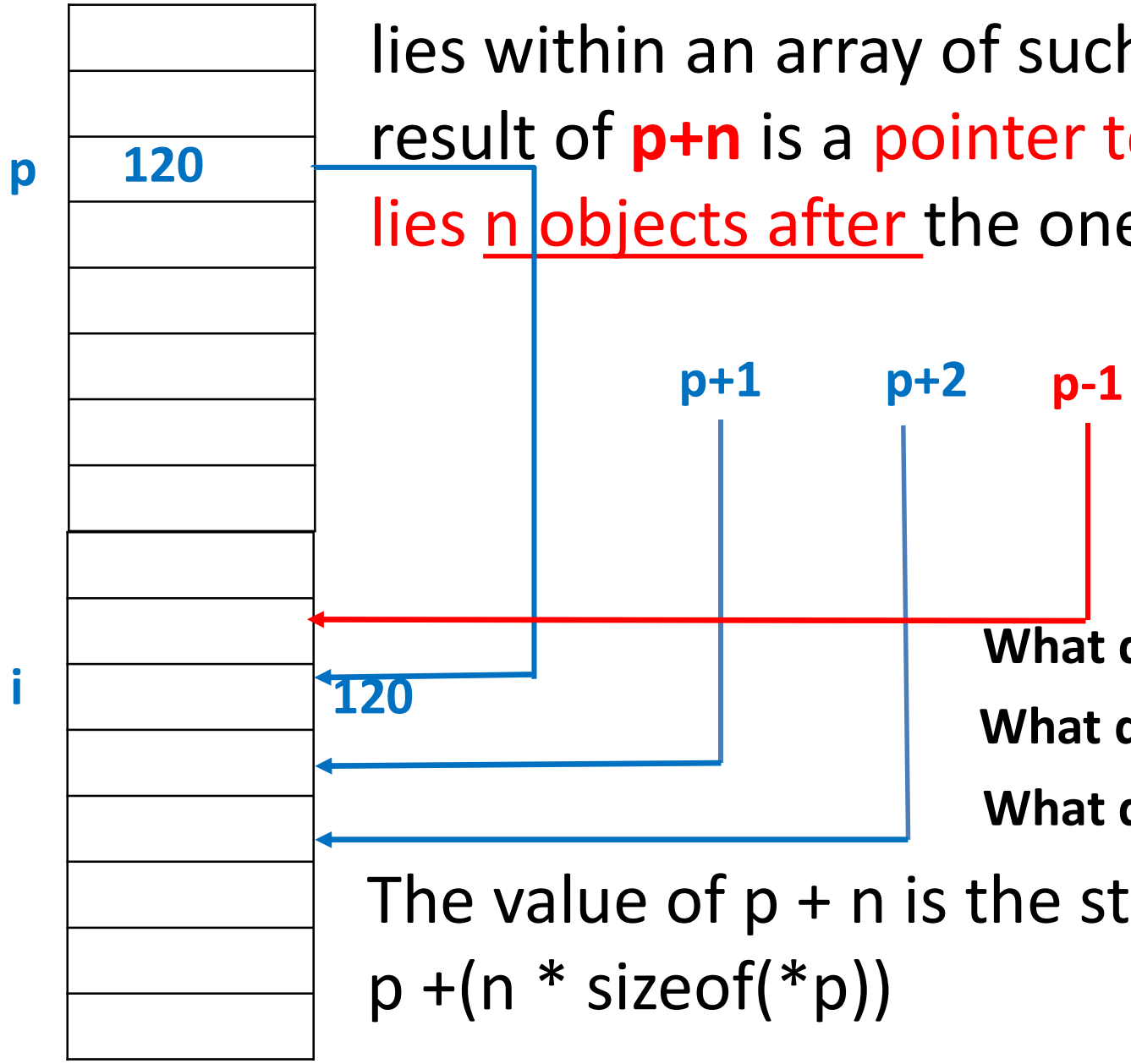


# Pointer Arithmetic

- Adding (subtracting) an **integral n** to a **pointer p**:
- **Adding:** Assuming that the object that **p** points to lies within an array of such objects,

  the result of **p+n** is a pointer to object that lies n objects after the one **p** points to.

Assuming that the object that **p** points to lies within an array of such objects, the result of **p+n** is a pointer to object that lies n objects after the one **p** points to.




```
int  i=1,  *p;
p = &i;
```

**What does p+1 evaluate to?**

**What does p+2 evaluate to?**

**What does p-1 evaluate to?**

The value of p + n is the storage location:

p +(n * sizeof(*p))

# Pointer Arithmetic

- Adding (subtracting) an **integral n** to a **pointer p**:
- Assuming that the object that **p** points to lies within an array of such objects,

  the result of **p+n** is a pointer to object that lies

  n objects after the one **p** points to.
- The value of p + n is the storage location:

  p +(n * sizeof(*p))

# Pointer Arithmetic

- Two pointer of the same type can be subtracted:
- When p1 is subtracted from p2, the result is the number of objects that can fit in between the two pointers
  - result is a signed value (yet type is impl. dependent)
  - result is undefined if the pointers do not point to objects within the same array
- How can we find no of bytes between two pointers p1 and p2?

  (p2 –p1) * sizeof (*p1) or

  (int) p2 – (int) p1

# Remark

- The rules of **pointer arithmetics** apply regardless of how this is written:
- if p is a pointer, p+1 points to the next object of the same type
- Then p++;
  - p = p+1; → now p points to the next object of the same type

# Reminder

| Operator | Type | Associativity |
|---|---|---|
| Function call: () | | **Left to right** |
| (type) + - ++ -- **! & *** | Unary | **Right to left** |
| * / % | Binary | Left to right |
| + - | Binary | Left to right |
| < <= > >= | Binary | Left to right |
| == != | Binary | Left to right |
| && | Binary | Left to right |
| \|\| | Binary | Left to right |
| = *= /= %= += -= | Binary | Right to left |
| , | | Left to right |

# Precedence

- Pointer operators **&** and * have the same precedence with other unary ops (++, --, etc)
- They associate from right to left

char *cp;

*++cp → *(++cp) → increment pointer and dereference (i.e. fetch the pointed value)

*cp++ → *(cp++) → deref. cp, and then increment cp

++*cp → ++(*cp) → increment the value pointed by cp, cp remains unchanged

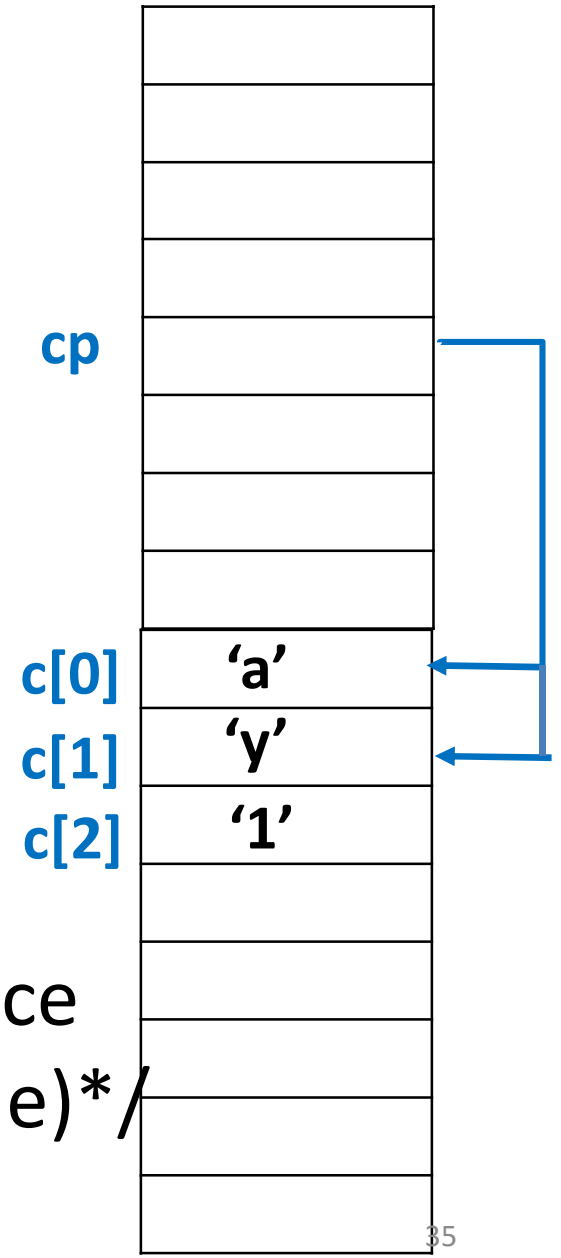
**C idioms (Book: 7.1.6)**

# C idioms-1

```
char c[3] = {‘a’, ‘y’, ‘1’}, *cp = &c[0];
char v;
v = *++cp;

printf("%c", v);

/* increment pointer and dereference
            (i.e. fetch the pointed value)*/
```

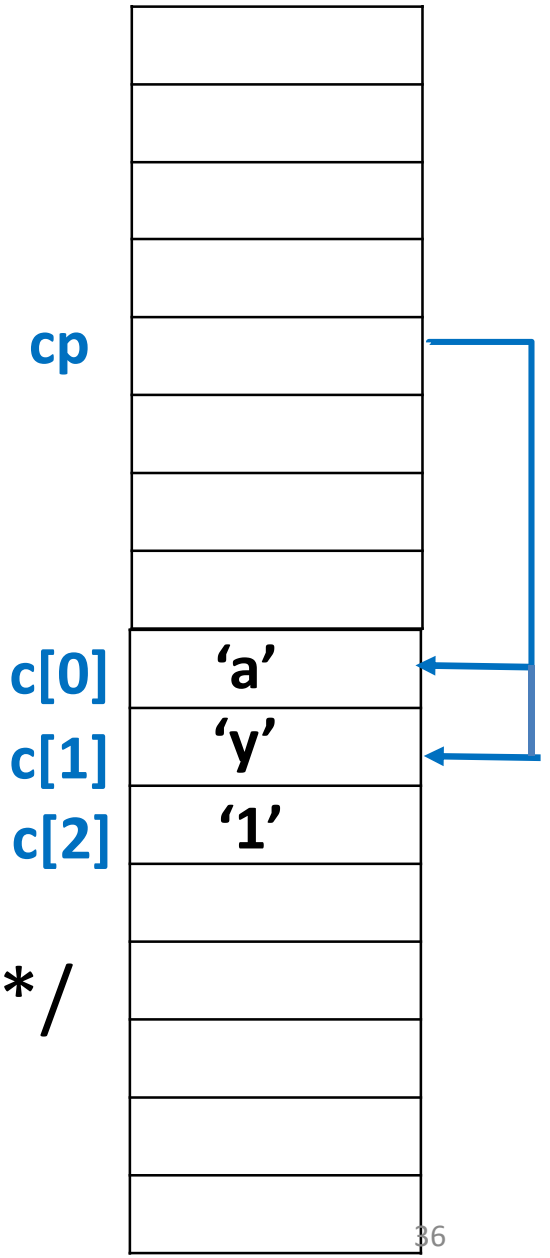


# C idioms-2

```
char c[3] = {‘a’, ‘y’, ‘1’}, *cp = &c[0];
char v;
v = *cp++ ;

printf("%c", v);

/* deref. cp, and then increment cp */
```

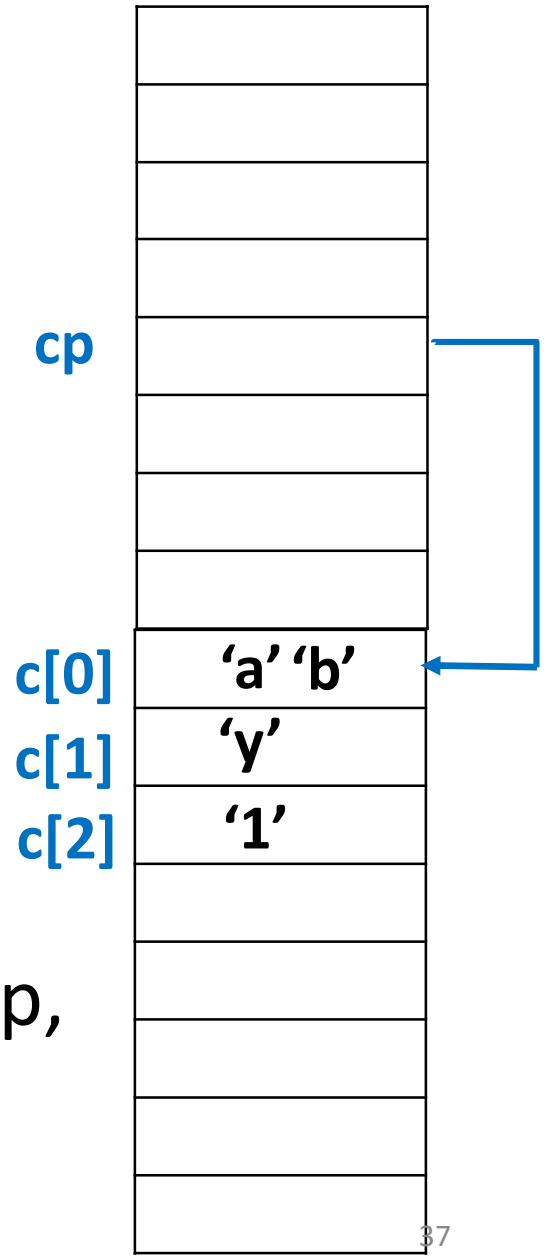


# C idioms-3

```
char c[3] = {‘a’, ‘y’, ‘1’}, *cp = &c[0];
char v;
v = ++*cp;

printf("%c", v);

/* increment the value pointed by cp,
   cp   remains unchanged */
```

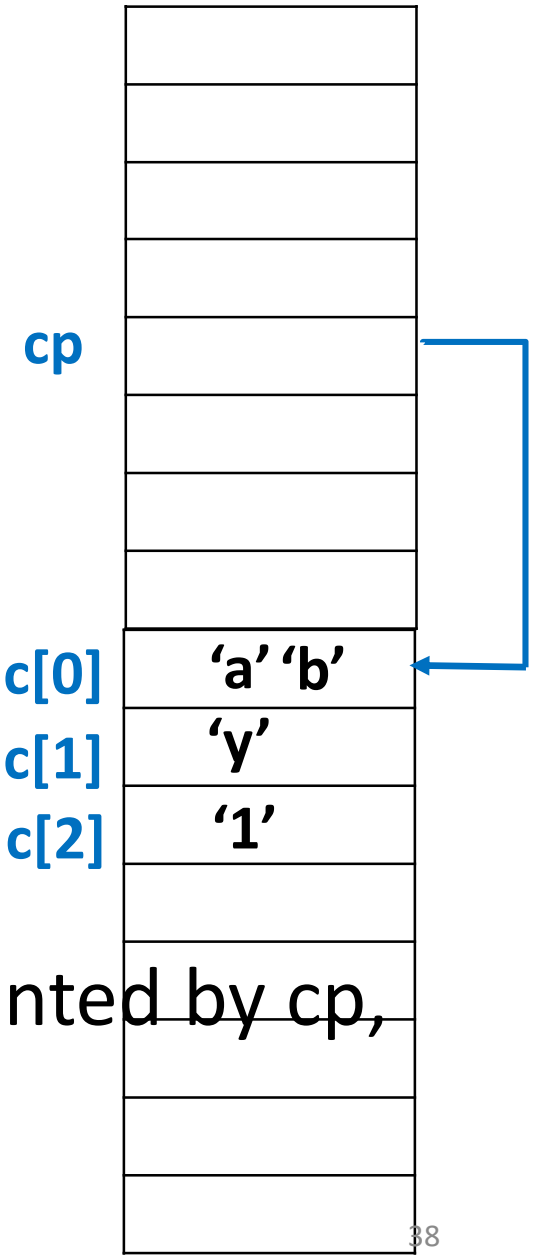


# C idioms-4

```
char c[3] = {‘a’, ‘y’, ‘1’}, *cp = &c[0];
char v;
v = (*cp)++;

printf("%c", v);

/* use & then increment the value pointed by cp,
   cp   remains unchanged */
```



# Pointer Comparison

- void * → generic pointer, we can use if we don't know the type of the data beforehand
  - Any pointer can be converted to type void * and back w/o loss of information (i.e., an explicit cast may be added for clarity; but not necessary)
  - Good for functions with parameters that can be pointers of any type (e.g., free, qsort, etc…)
  - You **cannot** dereference a void * or do pointer arithmetic with it; you must **convert** it to a pointer to a complete data type first.

# Pointer Comparison

- == and != allowed between:
  - Pointers of the same type
  - Pointer of type void * and any other pointer
  - NULL and any other pointer
- Pointer operands are considered equal only if:
  - they point to the same object or
  - they are both NULL

# Common Usage

```
if (ip != NULL)
    j += *ip;
```

```
if (ip)
    j += *ip;
```

```
if (ip == NULL)
    printf("error");
```

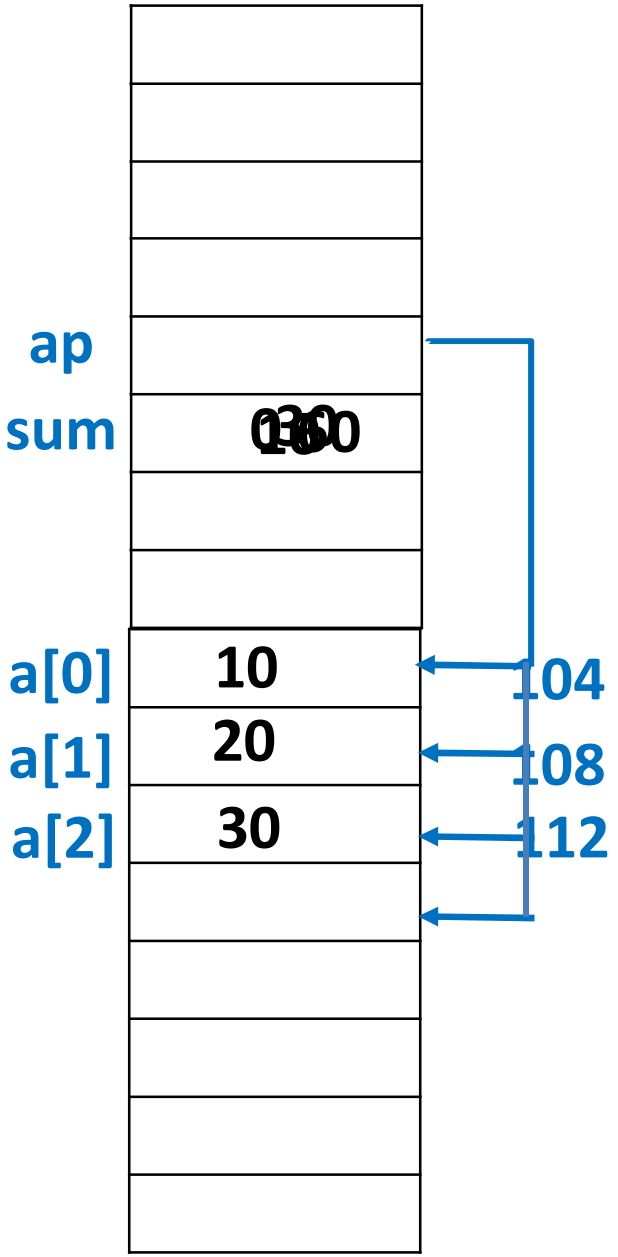
```
if (!ip)
    printf("error");
```

# Pointer Comparison

- < <= > >= allowed between:
  - Pointers of the same type
- Result depends on the relative location of the to objects pointed to (result is portable only if the objects are in the same array)
- The pointer to the first element beyond the array is well-represented to permit relational comparisons with a pointer in array

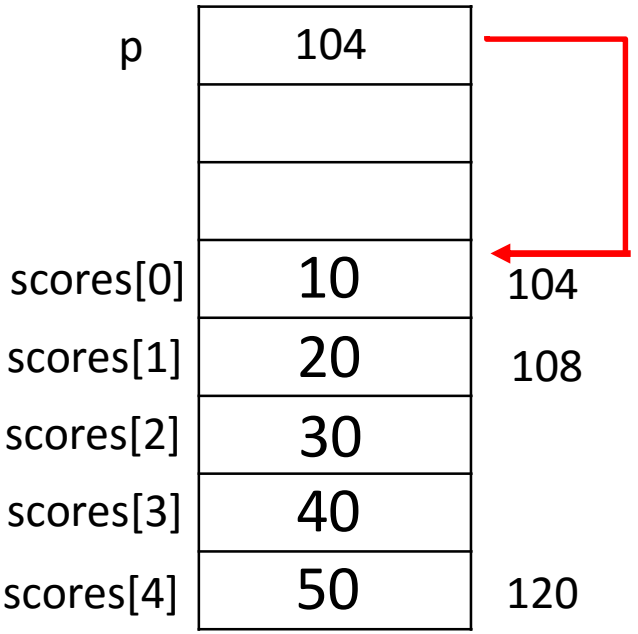
# Example

```
int main(void)
{ int sum = 0, a[3]={10,20,30}, *ap;
  ap = &a[0];
  while (ap < &a[3])
      sum += *ap++;
  return sum;
}
```



# Arrays vs Pointers…?

```
int scores[] = {10, 20, 30, 40, 50};
int *p = &scores[0];
```




value of scores[0]: 10

value of scores[1]: 20

..

value of scores[4]: 50

value of *p: 10

value of *(p+1): 20

..

value of *(p+4): 50

# Bitter Truth

- C treats a variable of type **"array of T"** as **"pointer to T"**, whose value is the **address of the first element of the array**

# So

Given the declaration int scores[5];

- The **array name** scores is a **SYNONYM** for the **pointer to** the **first element** of the array!
- The values of **scores** is the same as **&scores[0]**
- **BUT;**
- Array name is like a constant pointer, i.e., can **not** be changed
  - ~~int *p; scores = p;~~
  - ~~scores++;~~

The following are OK and equivalent:
p = &scores[0];
p = scores;

# Array Subscripting

- Array subscripting has also been defined in terms of the pointer arithmetics:

  **scores[i]** is same as ***(scores+i)**

(this is how C really implements array access)

- This equivalence means that the pointers may also be subscripted

int *p;

p = scores;

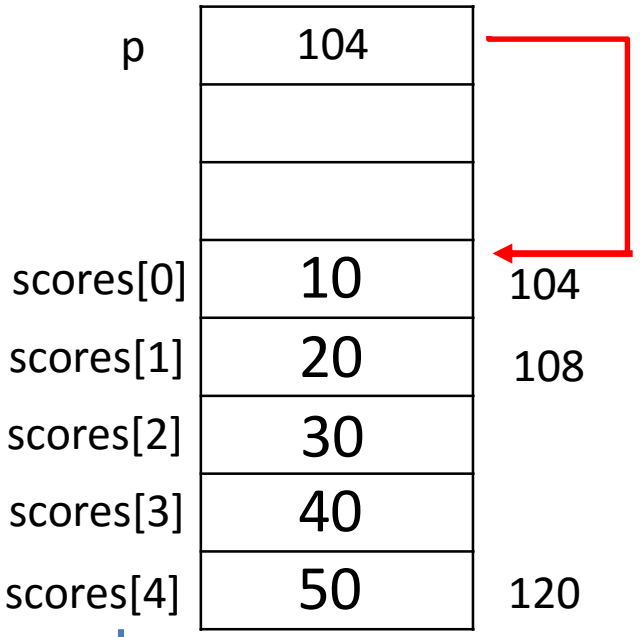
*(p+i) → p[i]

# Reminder

| Operator | Type | Associativity |
|---|---|---|
| Function call: () **Array subscript: []** | | **Left to right** |
| (type) + - ++ -- **! & *** | Unary | **Right to left** |
| * / % | Binary | Left to right |
| + - | Binary | Left to right |
| < <= > >= | Binary | Left to right |
| == != | Binary | Left to right |
| && | Binary | Left to right |
| \|\| | Binary | Left to right |
| = *= /= %= += -= | Binary | Right to left |
| , | | Left to right |

# Arrays vs Pointers…?

```
int scores[] = {10, 20, 30, 40, 50};
int *p = &scores[0];
```




| | | |
|---|---|---|
| value of scores[0]: 10 | value of *p: 10 | value of **p[0]**: 10 |
| value of scores[1]: 20 | value of *(p+1): 20 | value of **p[1]**: 20 |
| .. | .. | .. |
| value of scores[4]: 40 | value of *(p+4): 40 | value of **p[4]**: 40 |

# Arrays vs Pointers

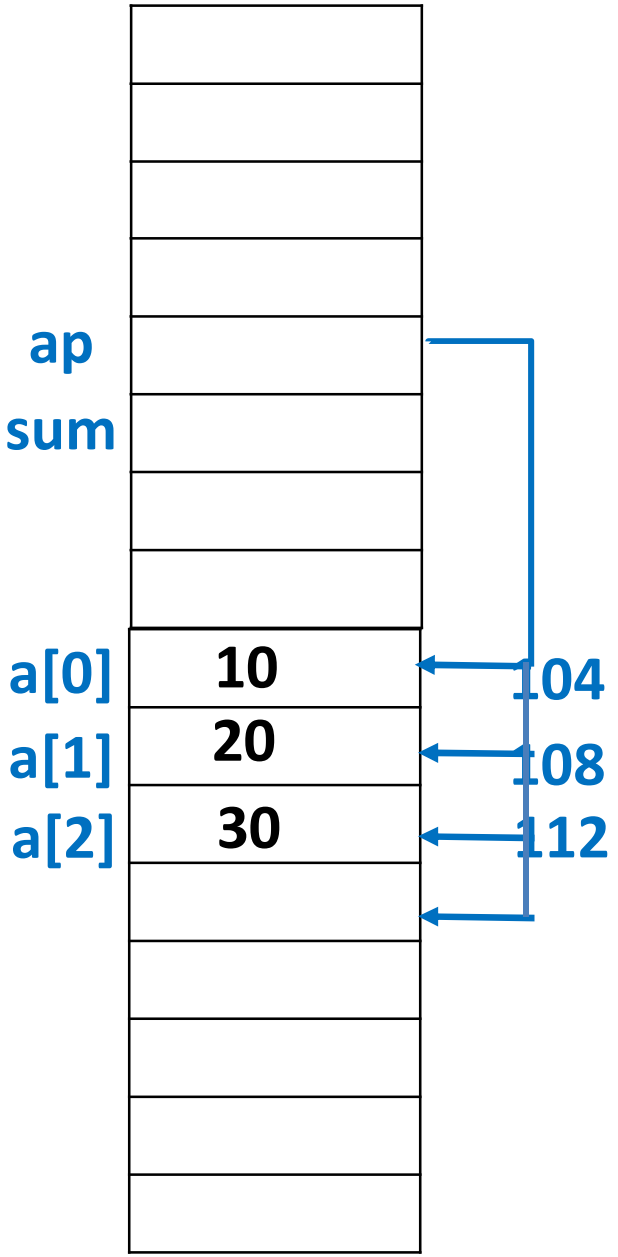
int scores[5], *p;

| ARRAY NOTATION | POINTER NOTATION |
|---|---|
| &scores[0] | |
| scores[i] | |
| &scores[i] | |
| p[i] | |

# Revisit

```
int main(void)
{ int sum = 0, a[3]={10,20,30}, *ap;
  ap = &a[0];
  while (ap < &a[3])
      sum += *ap++;
  return sum;
}
```

a+3

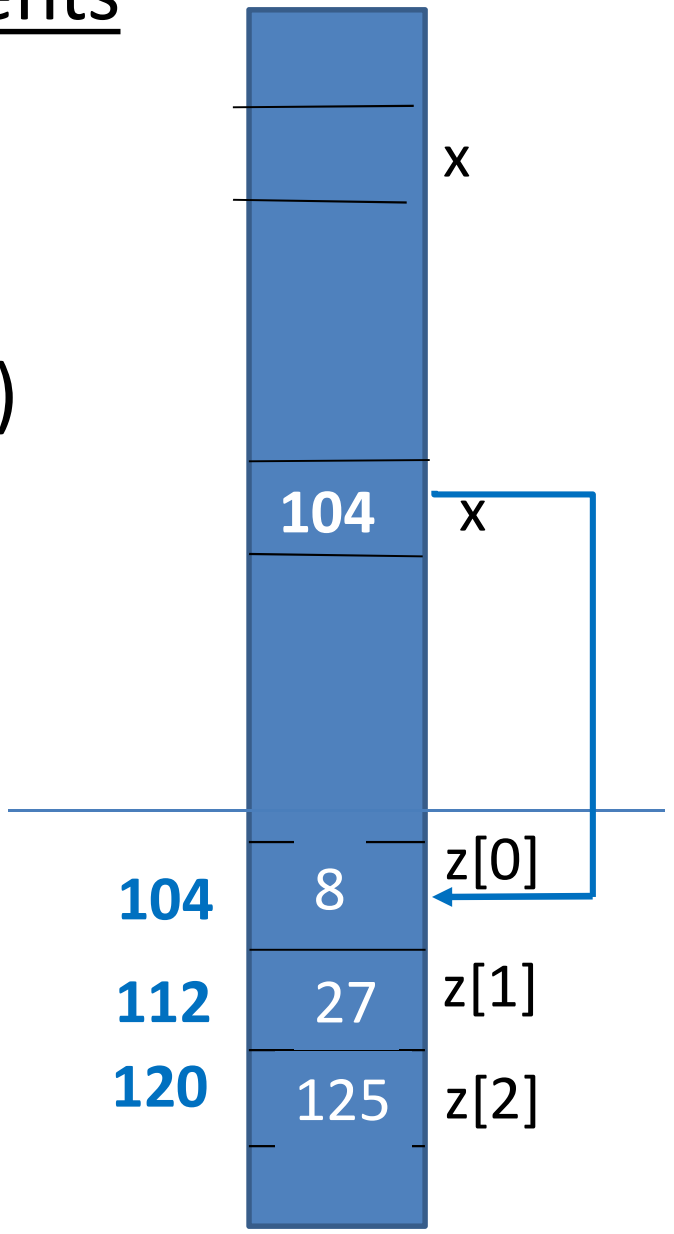


# Revisit: Passing Arrays as Arguments

```
double cuberoot(double x)
{ return pow(x, 1.0/3.0); }

void array_cuberoot(double x[3])
{ int i;

   for (i=0; i<3; i++)
      x[i] = cuberoot(x[i]); }
  *(x+i)                  *(x+i)
int main(void)
{ double z[3] = {8, 27, 125};
  array_cuberoot(z);
   /* Output if we print elements of array z here?*/ }
```



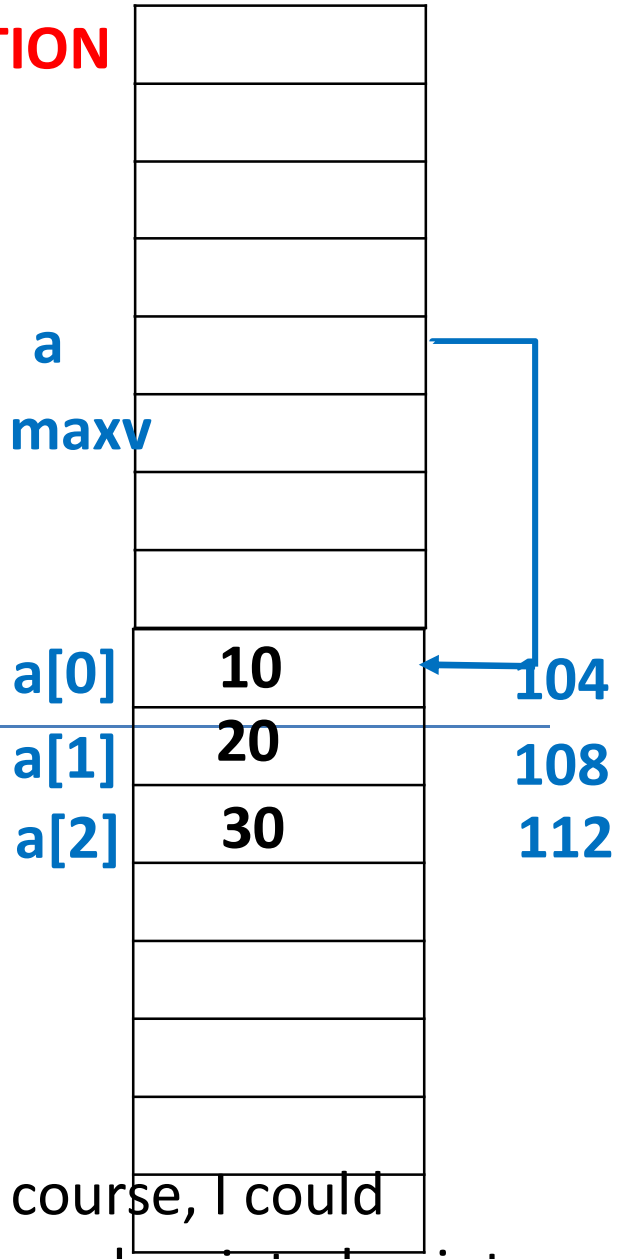
- Now that we just learned that the **array name** is a **SYNONYM** for the **pointer to** the **first element** of the array
  - So what we pass as parameter is **the address** of the **first element** of the array
  - Hence the formal parameter declared to be of type **"array of T"** is treated as of type **"pointer to T"**
  - So, we can equivalently write:
  - void **array_cuberoot**(double x[], int **length**)
  - void **array_cuberoot**(double *x, int **length**)

```
int max (int a[], int length)
{ int i, maxv;
  for (i=1, maxv=a[0]; i < length; i++)
      if (a[i] > maxv)
          maxv = a[i];
  return maxv; }
```

**ARRAY NOTATION**

```
int max (int *a, int length)
{ int i, maxv;
  for (i=1, maxv=*a; i < length; i++)
      if (*(a+i) > maxv)
          maxv = *(a+i);
  return maxv; }
```

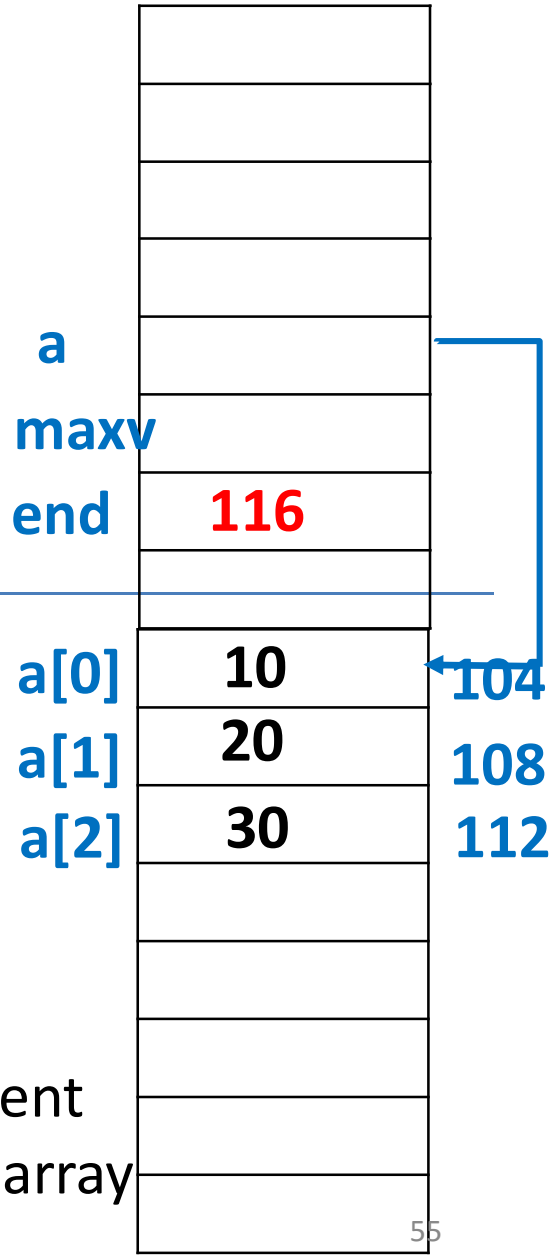
**POINTER NOTATION**




Of course, I could use subscripted pointers like a[i] here, too…

```
int max (int *a, int length)
{ int i, maxv;
   for (i=1, maxv=*a; i < length; i++)
       if (*(a+i) > maxv)
           maxv = *(a+i);
   return maxv; }
```

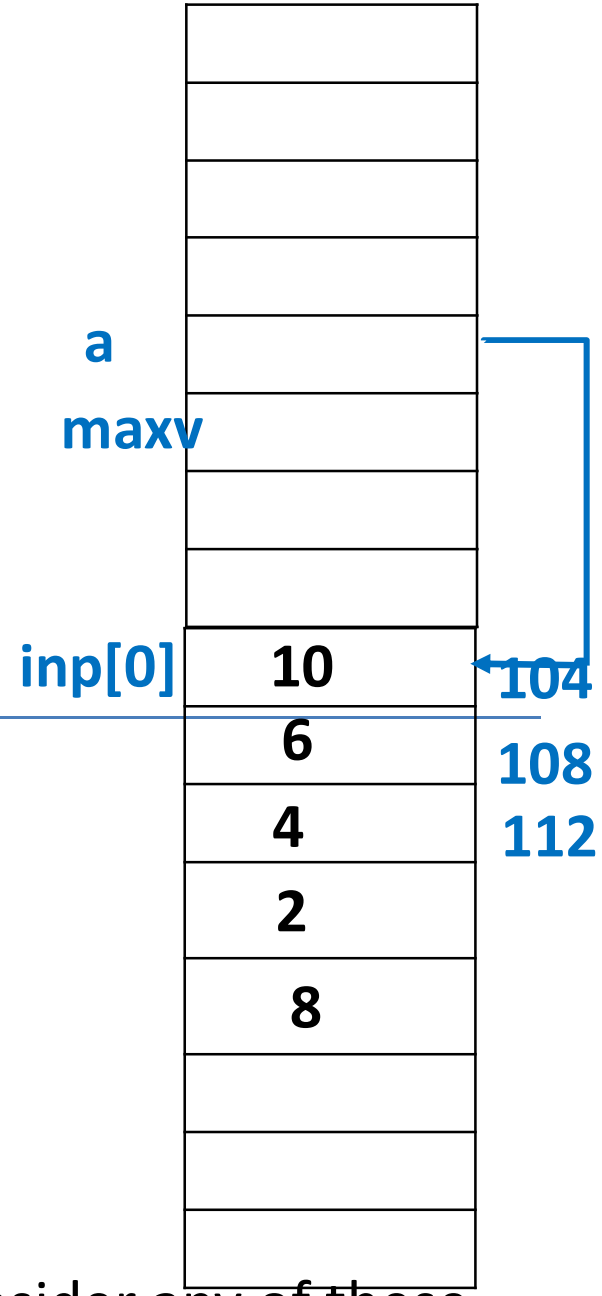
**POINTER NOTATION**




```
int max (int *a, int length)
{ int maxv, *end = a +length;
   for (maxv=*a; a < end ; a++)
       if (*a > maxv)
           maxv = *a;
   return maxv; }
```

Points to the first element beyond the end of the array

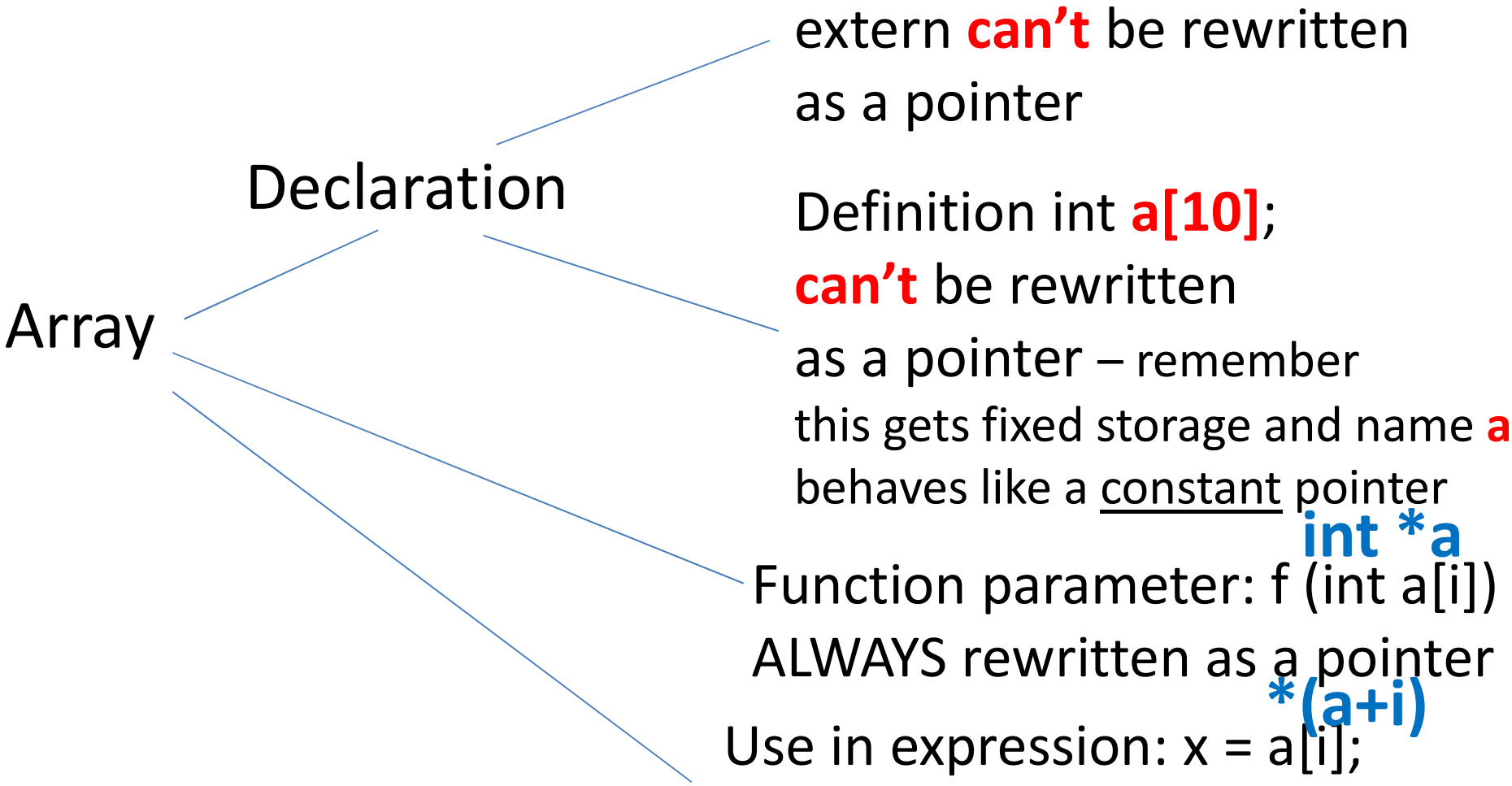
```
int max (int *a, int length)
{ int maxv, *end = a +length;
  for (maxv=*a; a < end ; a++)
      if (*a > maxv)
          maxv = *a;
  return maxv; }
```




```
int inp[5] = {10, 6, 4, 2, 8};
printf("%d\n", max(inp, 5) );
printf("%d\n", max(&inp[2], 3) );
printf("%d\n", max(inp+2, 3) );
printf("%d\n", max(inp+2, 2) );
```

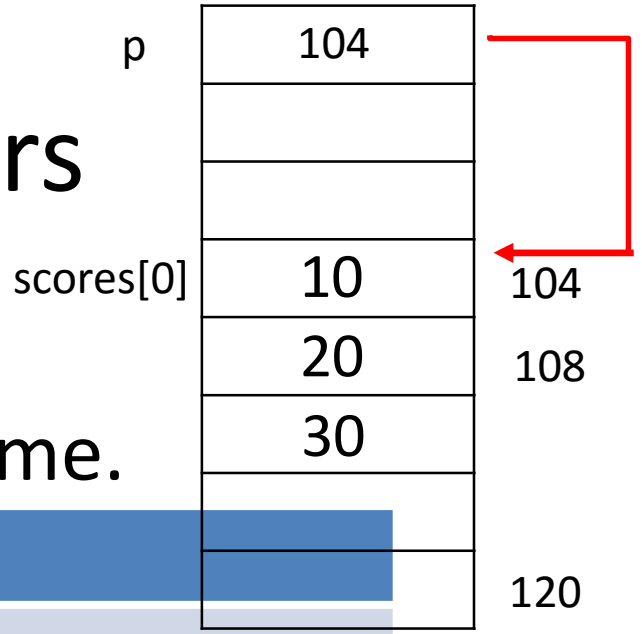
Consider any of these max functions

# When arrays are treated as pointers, and when not?




[Deep C Secrets]

# Arrays & Pointers




- Arrays and pointers are related,

  but the are NOT the same.

| Pointer | Array |
|---|---|
| Holds pointer to data | Holds data |
| Data accessed indirectly | Data accessed directly |
| Commonly used for dynamic data structures | Commonly used for storing fixed no of elements of the same type |
| Allocated via malloc etc | Implicitly allocated and deallocated |
| Typically points to anonymous data | Is a named variable on its own right |

[Deep C Secrets]

# Warning