WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

POINTERS

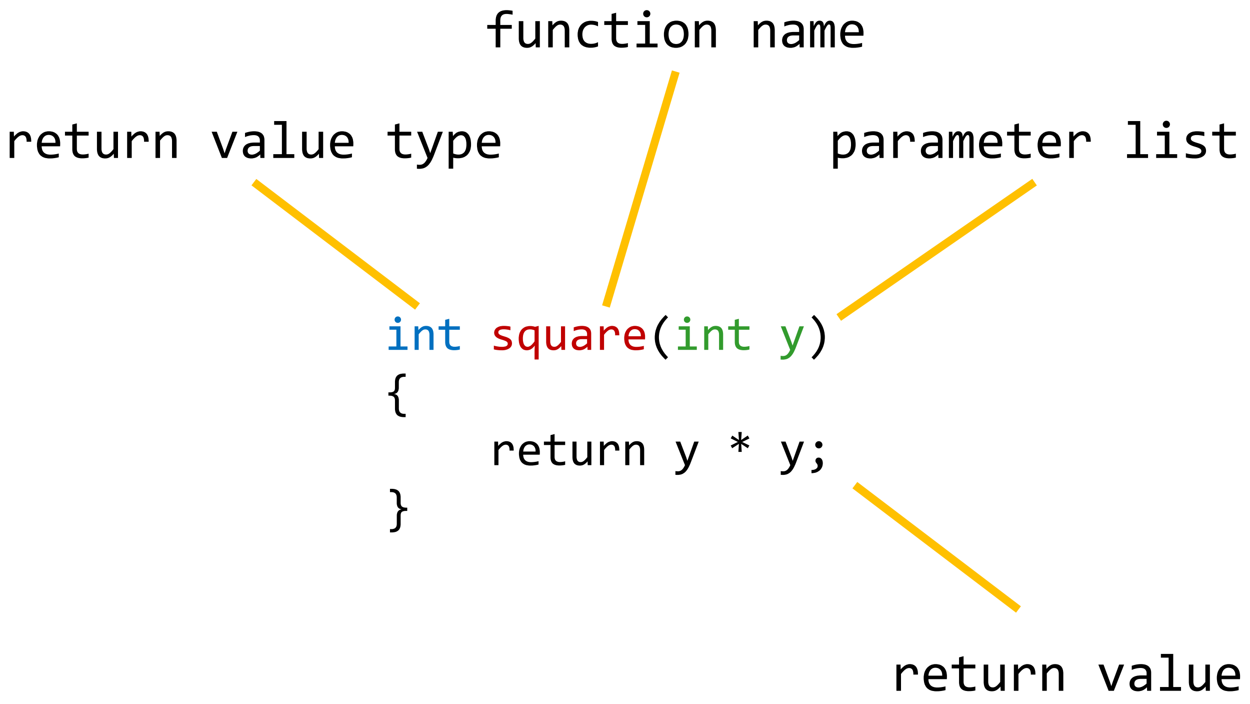
Dr Frank guan

INF1002 – Programming Fundamentals

Week 10

RECAP FROM LAST WEEK

- Function



function name

return value type

parameter list

```
int square(int y)
{
    return y * y;
}
```

return value

- A function prototype is a function definition without a body, e.g.
  - `int square(int);`

CALL BY VALUE VS CALL BY REFERENCE

Call by Reference

Call by Value

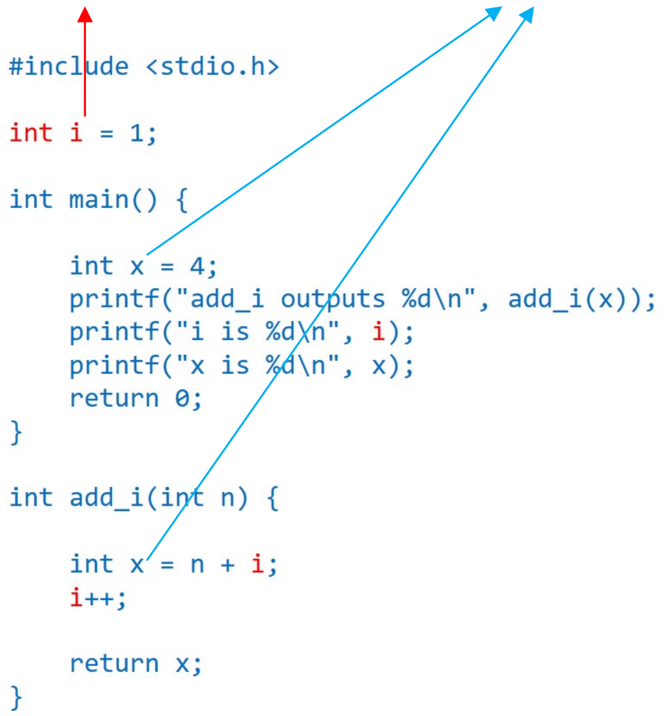
```
cup =

fillCup(        )
```

```
cup =

fillCup(        )
```

www.mathwarehouse.com

SCOPE

- The scope of an identifier is the portion of the program in which the identifier can be referenced.
- Global VS Local variable

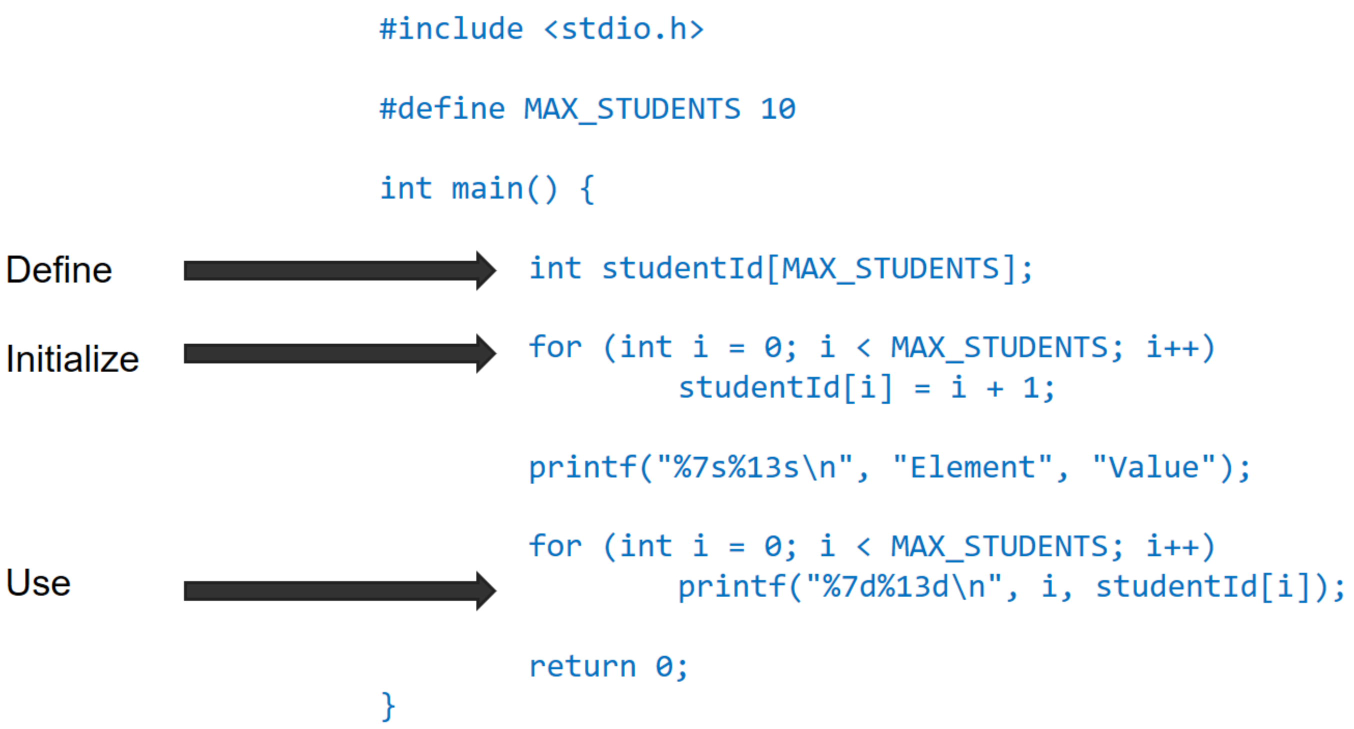


```
#include <stdio.h>

int i = 1;

int main() {

    int x = 4;
    printf("add_i outputs %d\n", add_i(x));
    printf("i is %d\n", i);
    printf("x is %d\n", x);
    return 0;
}

int add_i(int n) {

    int x = n + i;
    i++;

    return x;
}
```

ARRAY

An array is a group of memory locations that all have

- the same **name**
- the same **type**



Define

Initialize

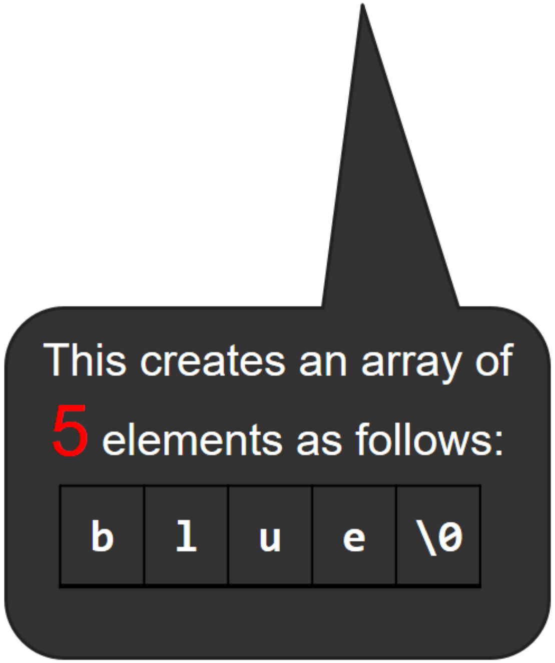
Use

```c
#include <stdio.h>

#define MAX_STUDENTS 10

int main() {

        int studentId[MAX_STUDENTS];

        for (int i = 0; i < MAX_STUDENTS; i++)
                studentId[i] = i + 1;

        printf("%7s%13s\n", "Element", "Value");

        for (int i = 0; i < MAX_STUDENTS; i++)
                printf("%7d%13d\n", i, studentId[i]);

        return 0;
}
```

STRING

A string in C is an **array** of characters **ending in the null character (`'\0'`)**.

```
char colour[] = "blue";
```



This creates an array of 5 elements as follows:

b	l	u	e	\0

BEFORE WE START

- Project related matters
- Specs uploaded
- Grouping uploaded
  - Double check your name, ID and Email
  - Start discussing with your teammates

- Alert
  - Be focused in this lecture

Agenda

1. Pointers
2. Arrays and pointers
3. Call-by-reference

MORE ABOUT VARIABLES

For ALL variables:

- name, type, value
- How to use:
  - Declare
  - Initialize
  - Use (assign new value/retrieve value)

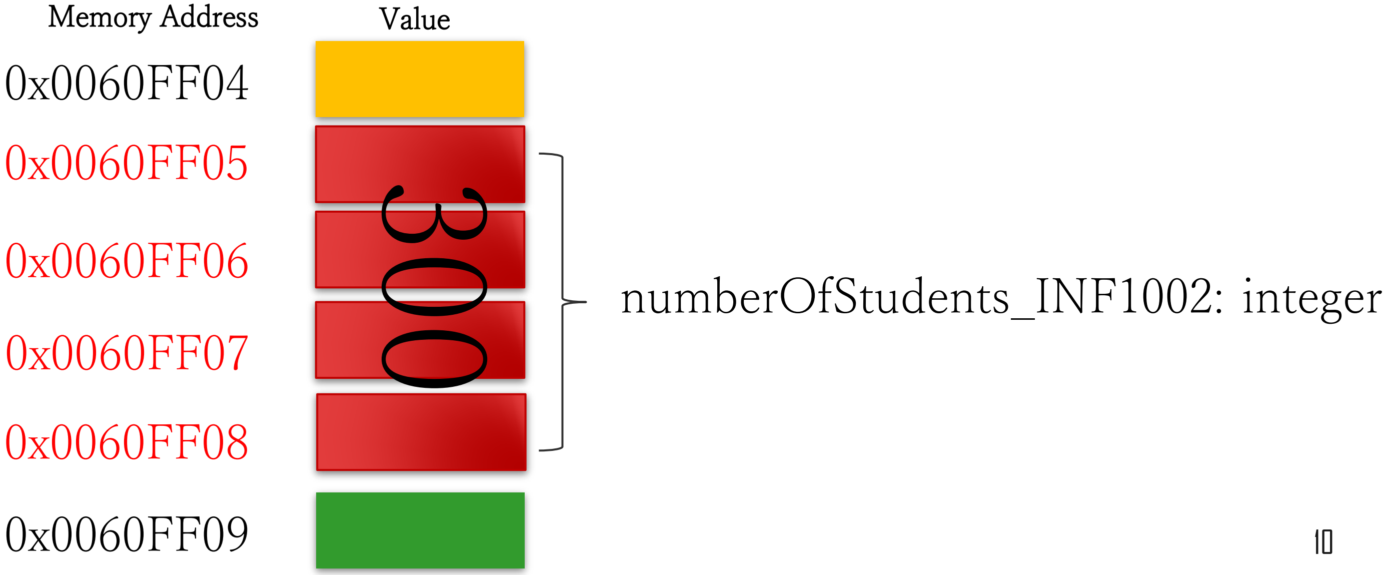
```
int numberOfStudents_INF1002 = 300;
```

- Two steps happened for the above code:
  - Step 1 – declare :
    - int numberOfStudents_INF1002;
    - A random value will be assigned to numberOfStudents_INF1002
  - Step 2 – initialize (assign an initial value):
    - numberOfStudents_INF1002 = 300;

MORE ABOUT VARIABLES

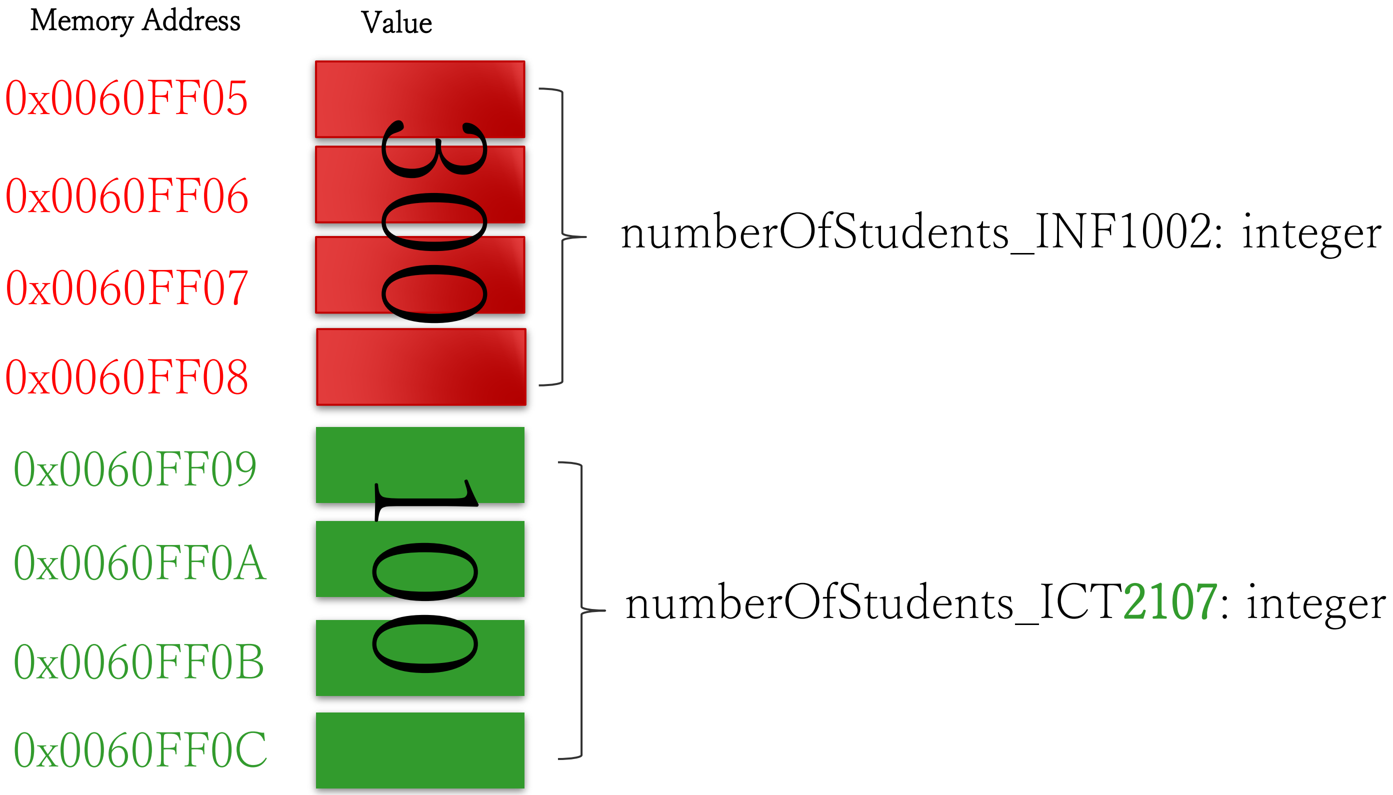
- When a variable is declared:
  - Memory space is automatically allocated for the variable
- When a variable is initialized/assigned with a value:
  - The data in the allocated memory space will be updated

```
int numberOfStudents_INF1002 = 300;
printf("Memory starting address: %p", &numberOfStudents_INF1002);
printf("Memory length: %d", sizeof(numberOfStudents_INF1002);

Result:
Memory starting address: 0x0060FF05
Memory length: 4
```



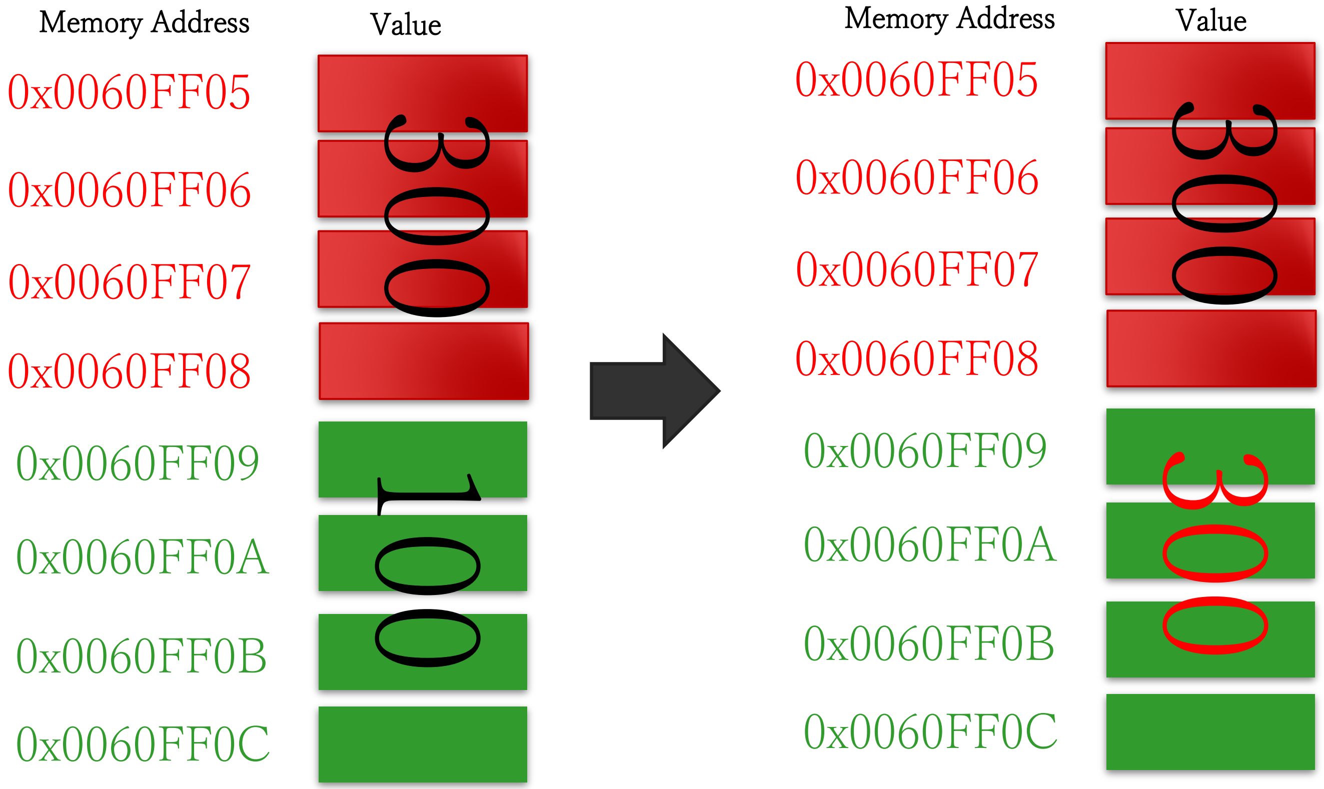
MORE ABOUT VARIABLES

```
int numberOfStudents_INF1002 = 300;
int numberOfStudents_ICT2107 = 100;
```



MORE ABOUT VARIABLES

```
numberOfStudents_ICT2107 = numberOfStudents_INF1002;
```



Memory Address	Value
0x0060FF05	300
0x0060FF06	
0x0060FF07	
0x0060FF08	
0x0060FF09	100
0x0060FF0A	
0x0060FF0B	
0x0060FF0C	

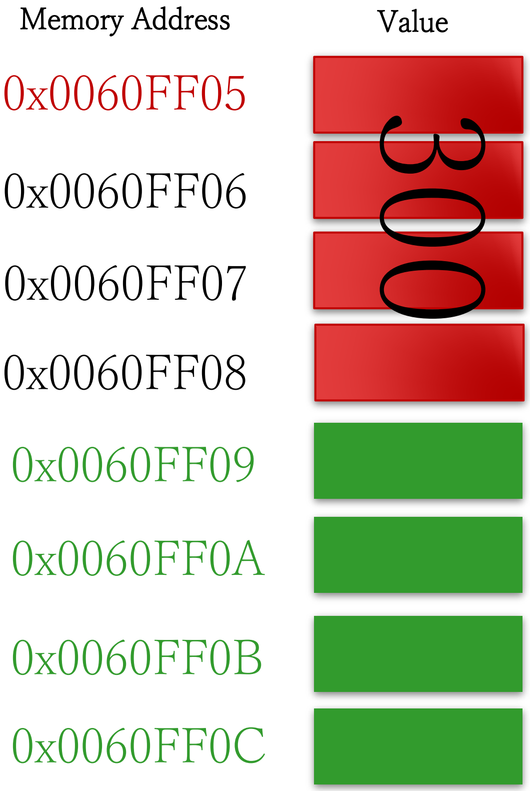
Memory Address	Value
0x0060FF05	300
0x0060FF06	
0x0060FF07	
0x0060FF08	
0x0060FF09	300
0x0060FF0A	
0x0060FF0B	
0x0060FF0C	

HOW TO OBTAIN MEMORY ADDRESS ALLOCATED FOR A VARIABLE

- The addressof & operator returns the **address** of its operand.

```
int numberOfStudents_INF1002 = 300;
```

Starting memory address: *&numberOfStudents_INF1002*



Memory Address	Value
0x0060FF05	300
0x0060FF06	
0x0060FF07	
0x0060FF08	
0x0060FF09	
0x0060FF0A	
0x0060FF0B	
0x0060FF0C	

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

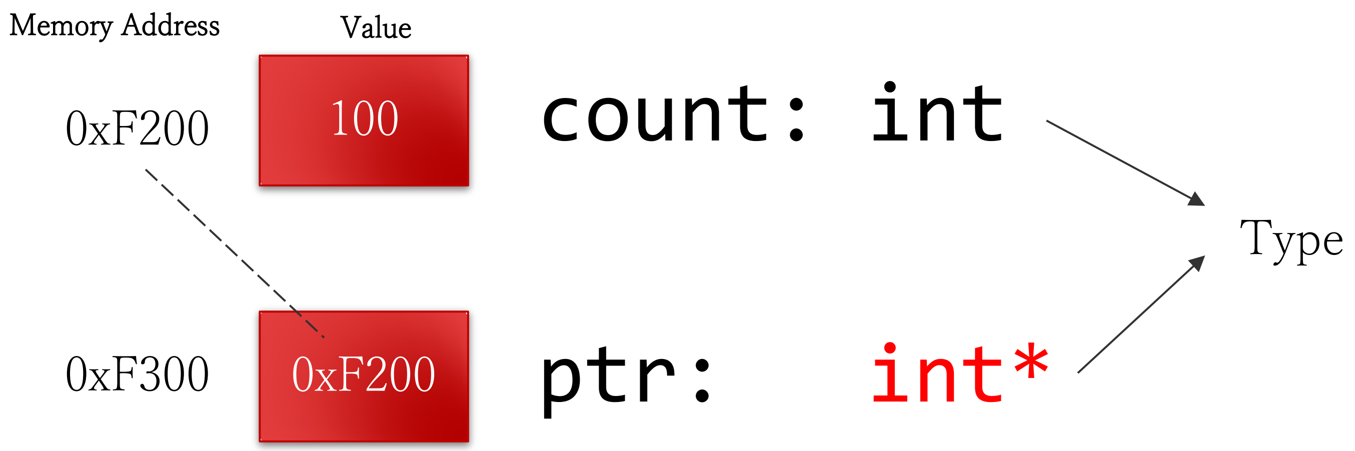
POINTERS

SiT SINGAPORE INSTITUTE OF TECHNOLOGY

POINTER VARIABLES

Pointers are variables whose values are memory addresses.

```
int count = 100;

int* ptr = &count;
```



Memory Address

Value

0xF200

100

count: int

0xF300

0xF200

ptr: int*

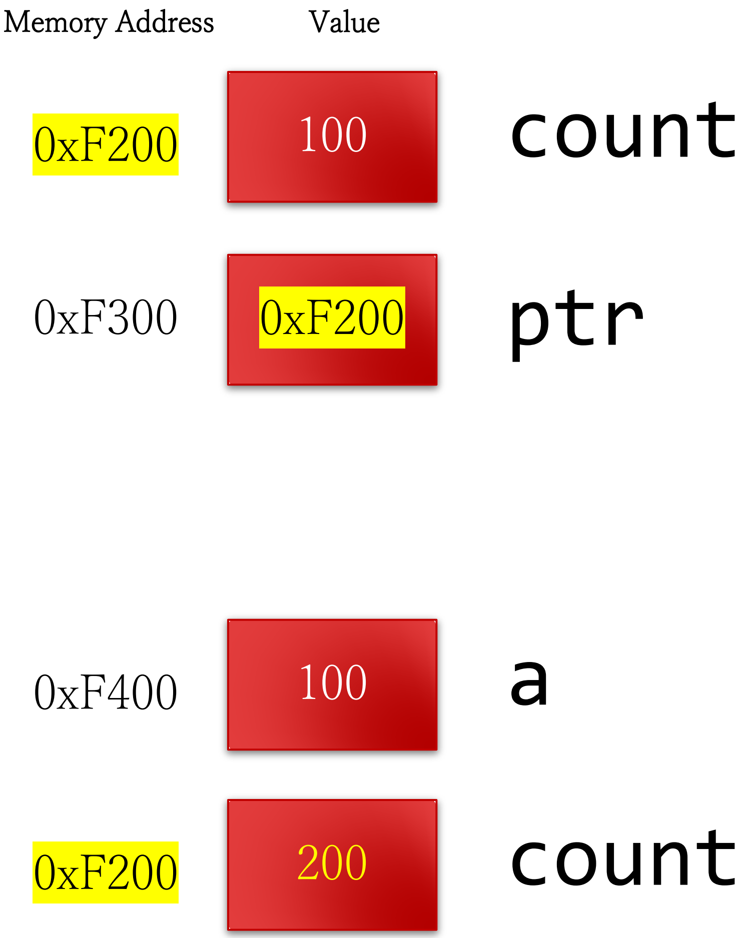
Type

HOW TO USE?

- Declare
  - int count;
  - int *ptr;

- Initialize (initial value assignment)
  - int count = 100;
  - ptr = &count;

- Use: dereference (retrieve/update the value in the memory space it points to)
  - int a = *ptr;
  - *ptr = 200;



Memory Address	Value	
0xF200	100	count
0xF300	0xF200	ptr
0xF400	100	a
0xF200	200	count

POINTER VARIABLE DEFINITION

```
int *ptr;
```

* indicates that the variable being defined is a pointer: “ptr is a pointer to an int”

POINTER VARIABLE DEFINITION

```
int *ptr1, *ptr2;
int a, b;
```

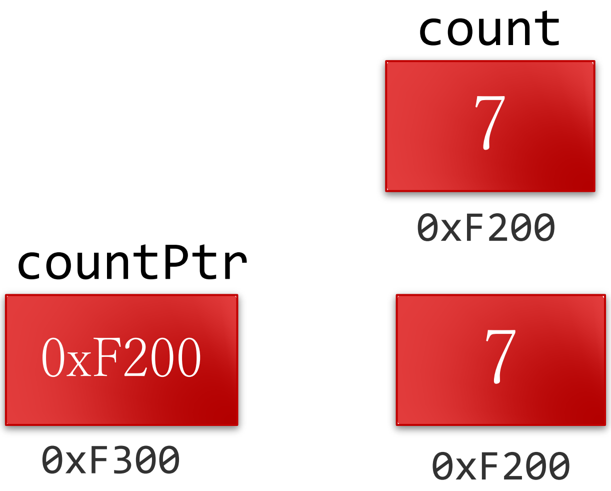
Note: The asterisk (*) does not distribute to all variable names in a declaration.

Each pointer must be declared with the * prefixed to the name.

POINTER INITIALIZATION

```
int count = 7;
```

```
int *countPtr = &count;
```



`count` directly references a variable that contain the value 7

`countPtr` indirectly references a variable that contains the value 7

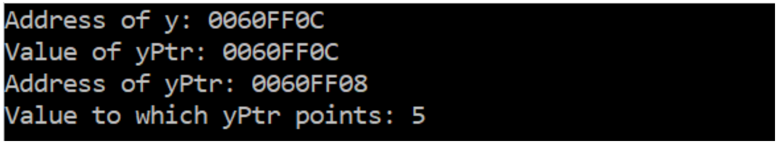
A **pointer** contains an address of a variable that contains a specific value

POINTER INITIALIZATION

The addressof & operator returns the **address** of its operand.

Assign the address of y to yPtr

```
#include <stdio.h>

int main() {
    int y = 5;
    int *yPtr;

    yPtr = &y;
    printf("Address of y: %p\n", &y);
    printf("Value of yPtr: %p\n", yPtr);
    printf("Address of yPtr: %p\n", &yPtr);
    printf("Value to which yPtr points: %d\n",
        *yPtr);

    return 0;
}
```



```
Address of y: 0060FF0C
Value of yPtr: 0060FF0C
Address of yPtr: 0060FF08
Value to which yPtr points: 5
```

QUESTION

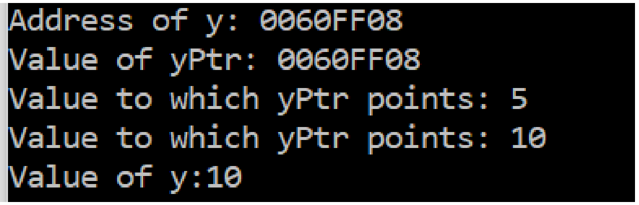
- Why not initialize a pointer variable with a direct value?
- i.e.
- `int *yPtr;`
- `yPtr = 0060FF0C;`

POINTER OPERATORS

The de-referencing operator returns the value of the object to which its operand points.

```
int main() {
    int y = 5;
    int *yPtr;

    yPtr = &y;
    printf("Address of y: %p\n", &y);
    printf("Value of yPtr: %p\n", yPtr);

    printf("Value to which yPtr points: %d\n", *yPtr);

    *yPtr = 10;

    printf("Value to which yPtr points: %d\n", *yPtr);

    printf("Value of y:%d\n", y);

    return 0;
}
```

The de-referencing operator is used to update the value of the object to which its operand points.

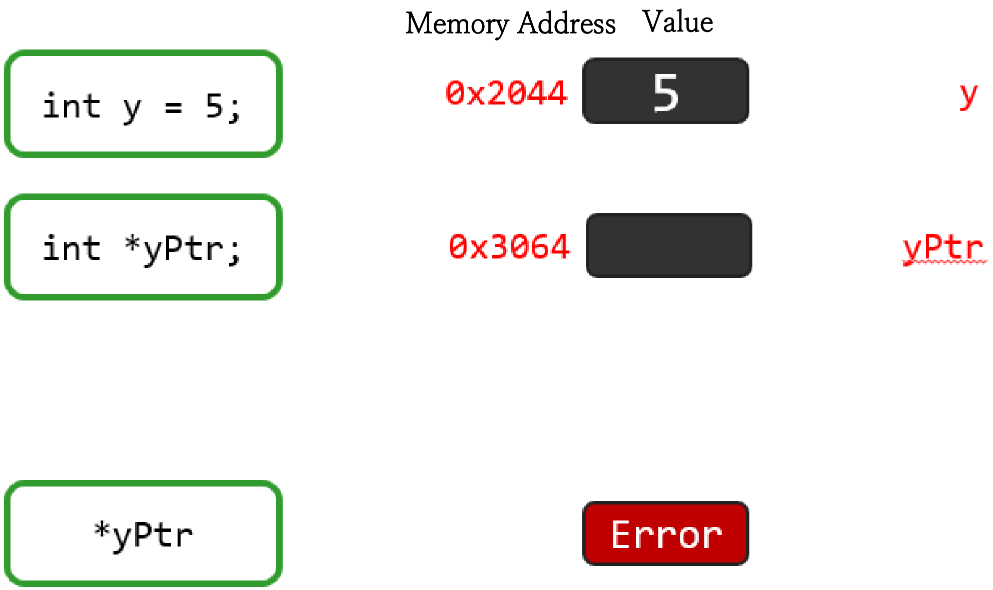


```
Address of y: 0060FF08
Value of yPtr: 0060FF08
Value to which yPtr points: 5
Value to which yPtr points: 10
Value of y:10
```

POINTER OPERATORS

Dereferencing a pointer which has not been properly initialised or that has not been assigned to point to a specific location in memory is an error.

This could cause a fatal run time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.



Memory Address Value

`int y = 5;` 0x2044 5 y

`int *yPtr;` 0x3064 yPtr

`*yPtr` Error

Dereferencing a pointer that has not been properly initialised.

EXERCISE

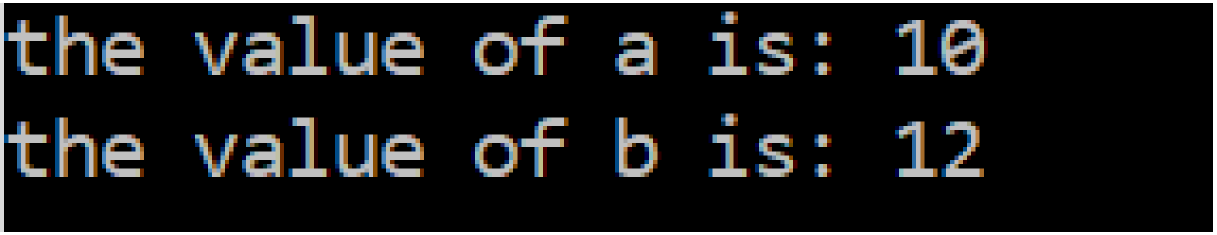
What are the values of ***a*** and ***b*** after each line of the following program?

```
int a = 5, b = 2;
int *p = &a, *q = &b;

(*p) *= 2;            //a = ?, (*p) = (*p) * 2
*q  = *p – 1;
 p  = &b;
 b  = *p + 3;         //b = ?
```

EXERCISE

What are the values of ***a*** and ***b*** after each line of the following program?

```
int a = 5, b = 2;
int *p = &a, *q = &b;

(*p) *= 2;                  //a = ?, (*p) = (*p) * 2
*q  = *p – 1;
 p  = &b;
 b  = *p + 3;               //b = ?
```



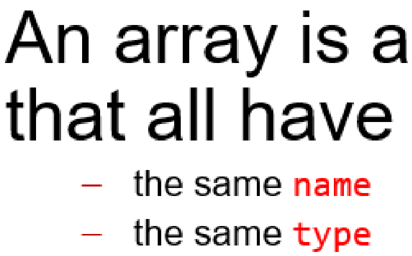
```
the value of a is: 10
the value of b is: 12
```

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

POINTERS & ARRAYS

SINGAPORE INSTITUTE OF TECHNOLOGY

POINTERS & ARRAYS



An array is a group of memory locations that all have

- the same `name`
- the same `type`

Pointers and arrays are intimately related in C.

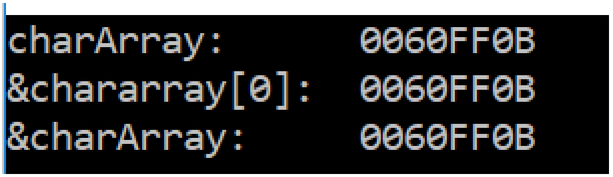
- An array name can be thought of as a `constant pointer` to the start of the array.
- Array subscripts can be applied to pointers.
- Pointer arithmetic can be used to navigate arrays.

POINTERS & ARRAYS

The name of the array evaluates to the address of the first element of the array.

```
int main() {

        char charArray[] = {'a', 'b', 'c', 'd', 'e' };;

        printf("charArray: \t%p\n", charArray);
        printf("&chararray[0]: \t%p\n", &(charArray[0]));
        printf(“&charArray: \t%p\n", &charArray);


        return 0;
}
```

Output



```
charArray:      0060FF0B
&chararray[0]:  0060FF0B
&charArray:     0060FF0B
```

POINTERS & ARRAYS

Subscripting and pointer arithmetic can be used interchangeably.

```
int main() {

    char b[] = {'a', 'b', 'c', 'd', 'e' };
    char *bPtr = b;

    printf("*(bPtr + 3): \t%c\n", *(bPtr + 3));
    printf("*(b + 3): \t%c\n", *(b + 3));
    printf("bPtr[3]: \t%c\n", bPtr[3]);

        return 0;
}
```

Output

```
*(bPtr + 3): d
*(b + 3):    d
bPtr[3]:     d
```

POINTERS & ARRAYS

The fourth element of b can be referenced using any of the following statements:

```
*(bPtr + 3)
```

3 is the offset to the pointer indicates which element of the array should be referenced

```
*(b+3)
```

The array itself can be treated as a pointer to the first element of the array.

```
bPtr[3]
```

pointers can be subscripted exactly as arrays can.

EXERCISE

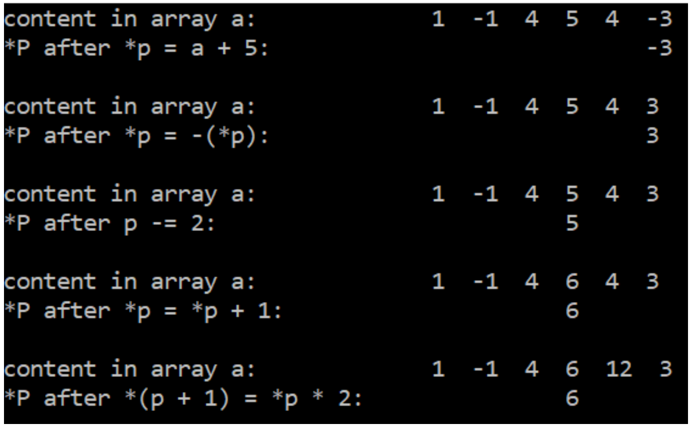
What are the contents of the array `a` after each line of the following program?

```
int a[] = { 1, -1, 4, 5, 4, -3 };
int *p = a + 5;

*p = -(*p);
 p -= 2;
*p = *p + 1;
*(p + 1) = *p * 2;
```

EXERCISE

What are the contents of the array `a` after each line of the following program?

```
int a[] = { 1, -1, 4, 5, 4, -3 };
int *p = a + 5;

*p = -(*p);
 p -= 2;
*p = *p + 1;
*(p + 1) = *p * 2;
```



```
content in array a:             1  -1  4  5  4  -3
*P after *p = a + 5:                             -3

content in array a:             1  -1  4  5  4  3
*P after *p = -(*p):                             3

content in array a:             1  -1  4  5  4  3
*P after p -= 2:                         5

content in array a:             1  -1  4  6  4  3
*P after *p = *p + 1:                    6

content in array a:             1  -1  4  6  12  3
*P after *(p + 1) = *p * 2:              6
```

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

POINTER EXPRESSIONS & ARITHMETIC

SINGAPORE
INSTITUTE OF
TECHNOLOGY

THE SIZEOF OPERATOR

The `sizeof` operator returns the number of bytes required to hold a type.

- E.g.
  - `sizeof(char)` evaluates to 1
  - `sizeof(int)` evaluates to 2, 4 or 8 depending on the word size of the compiler

THE SIZEOF OPERATOR

This gives the size of an integer.

```
int size = sizeof(int) * 4;
printf("size of 4 integers is:
    %d bytes\n", size);
```

THE SIZEOF OPERATOR

This gives the size of 4 integers.

```c
int size = sizeof(int) * 4;
printf("size of 4 integers is:
     %d bytes\n", size);
```

```
Output:
size of 4 integers is: 16 bytes
```

POINTER EXPRESSIONS & ARITHMETIC

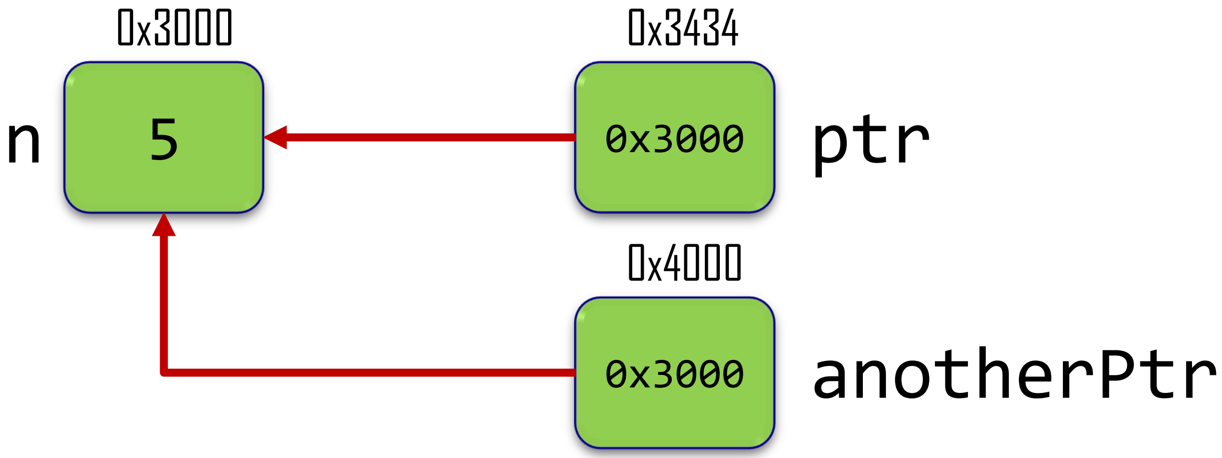
In general, pointers are valid operands in

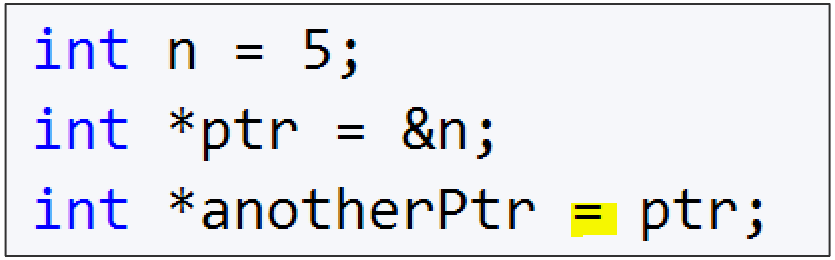
- assignment expressions
- arithmetic expressions
- comparison expressions

However, not all the operators normally used in these expressions are valid in conjunction with pointer variables.

POINTER ASSIGNMENT

A pointer can be assigned to another pointer if they have the same type.

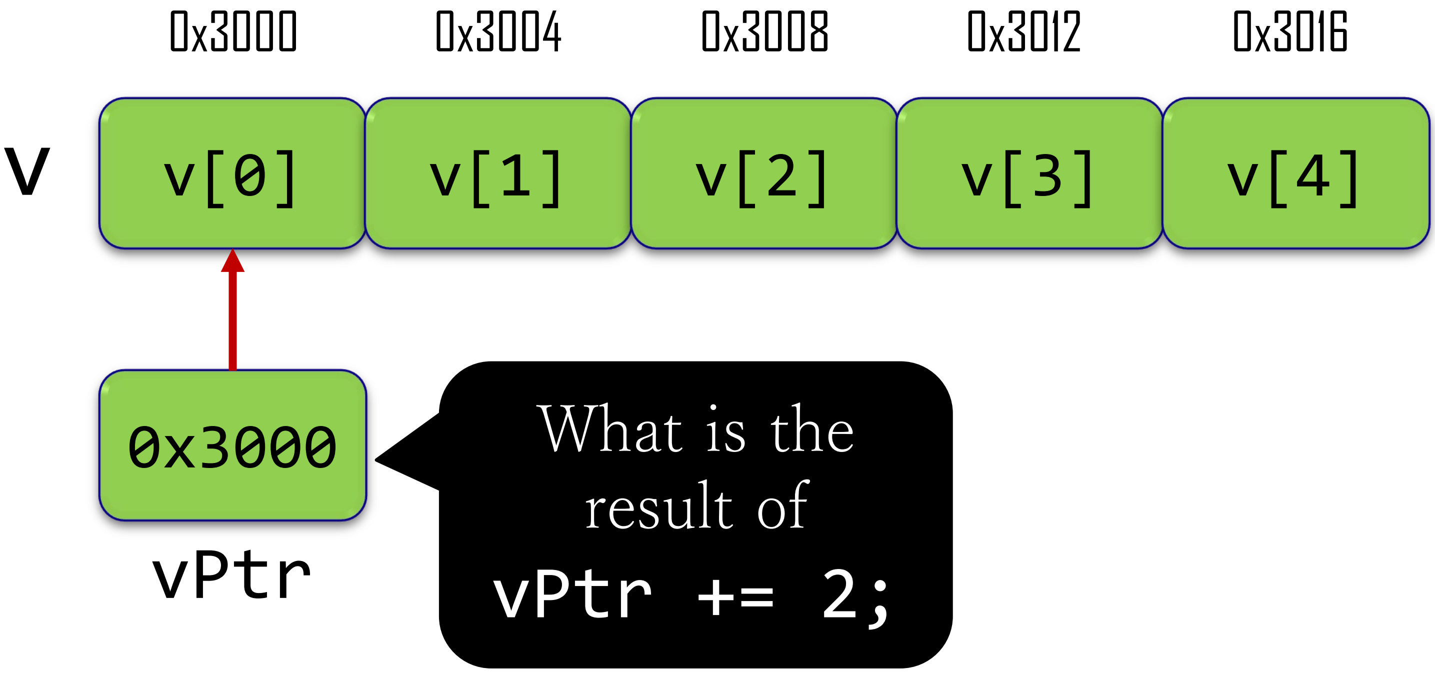




```
int n = 5;
int *ptr = &n;
int *anotherPtr = ptr;
```

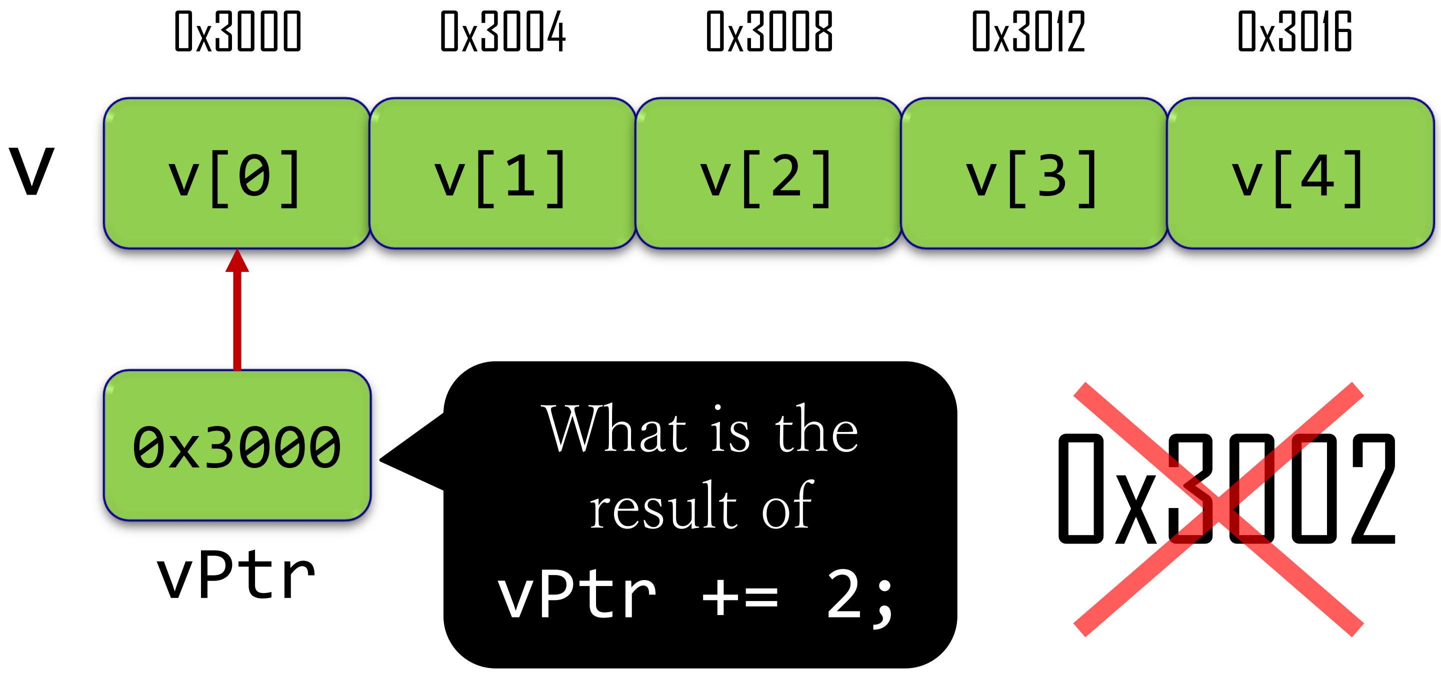
`anotherPtr` will point to whatever memory location that `ptr` is pointing to

POINTER EXPRESSIONS & ARITHMETIC



```
int v[5] = {0};
int *vPtr = v;
```

POINTER EXPRESSIONS & ARITHMETIC



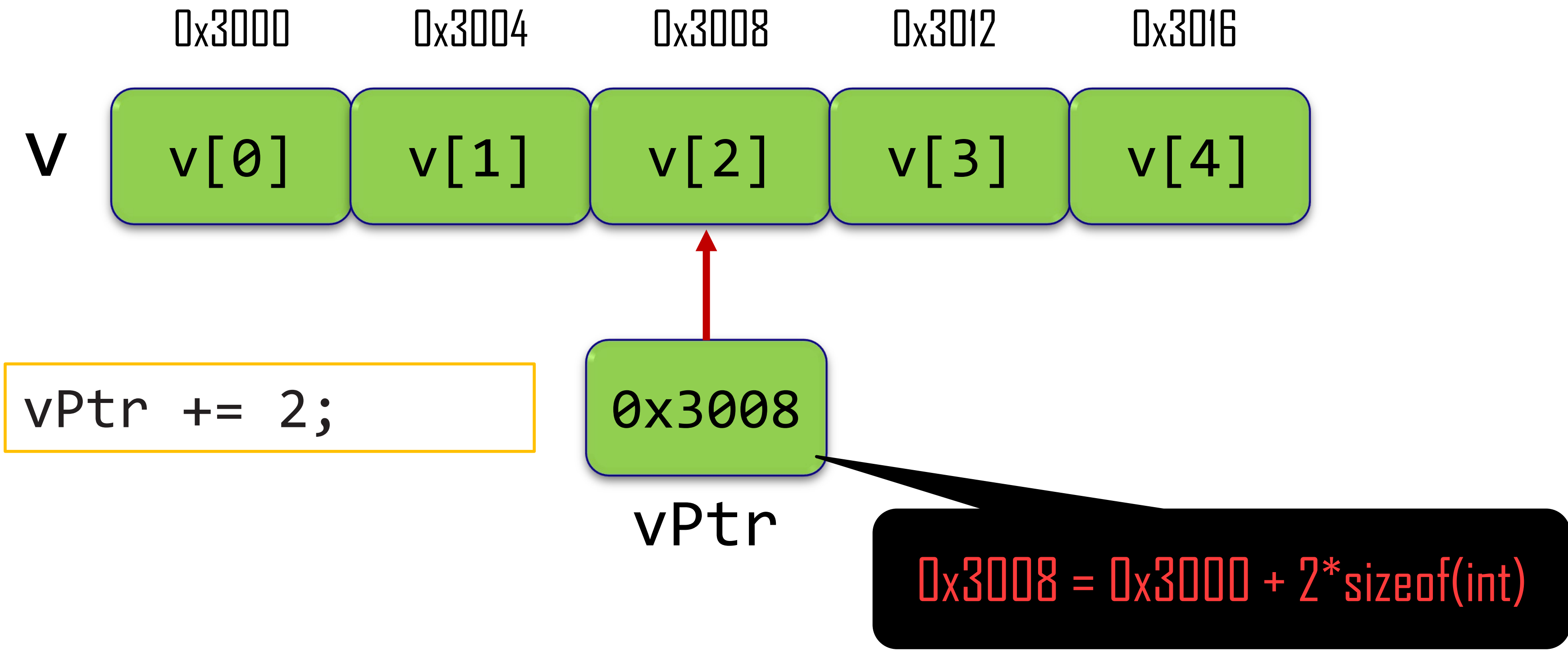
What is the result of
`vPtr += 2;`

~~0x3002~~

```
int v[5] = {0};
int *vPtr = v;
```

In conventional arithmetic, 3000+2 = 3002. However, this is `not the case with pointer arithmetic`.

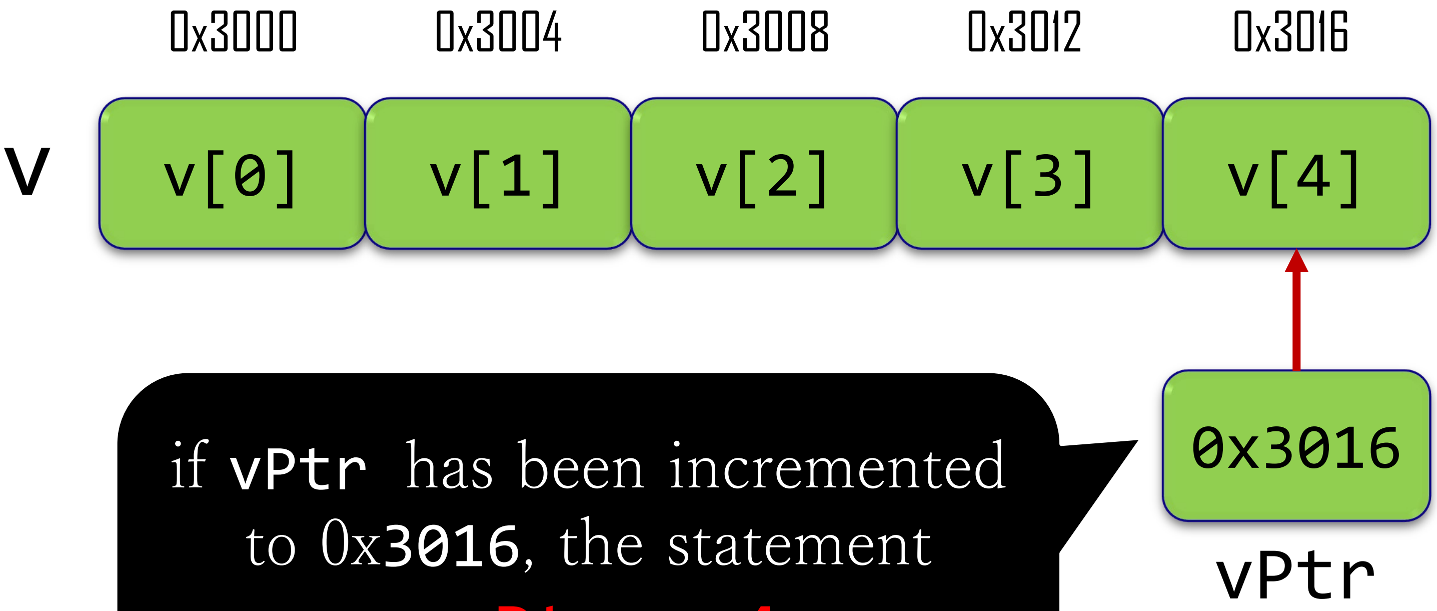
POINTER EXPRESSIONS & ARITHMETIC



```
vPtr += 2;
```

When an integer is added or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.

POINTER EXPRESSIONS & ARITHMETIC

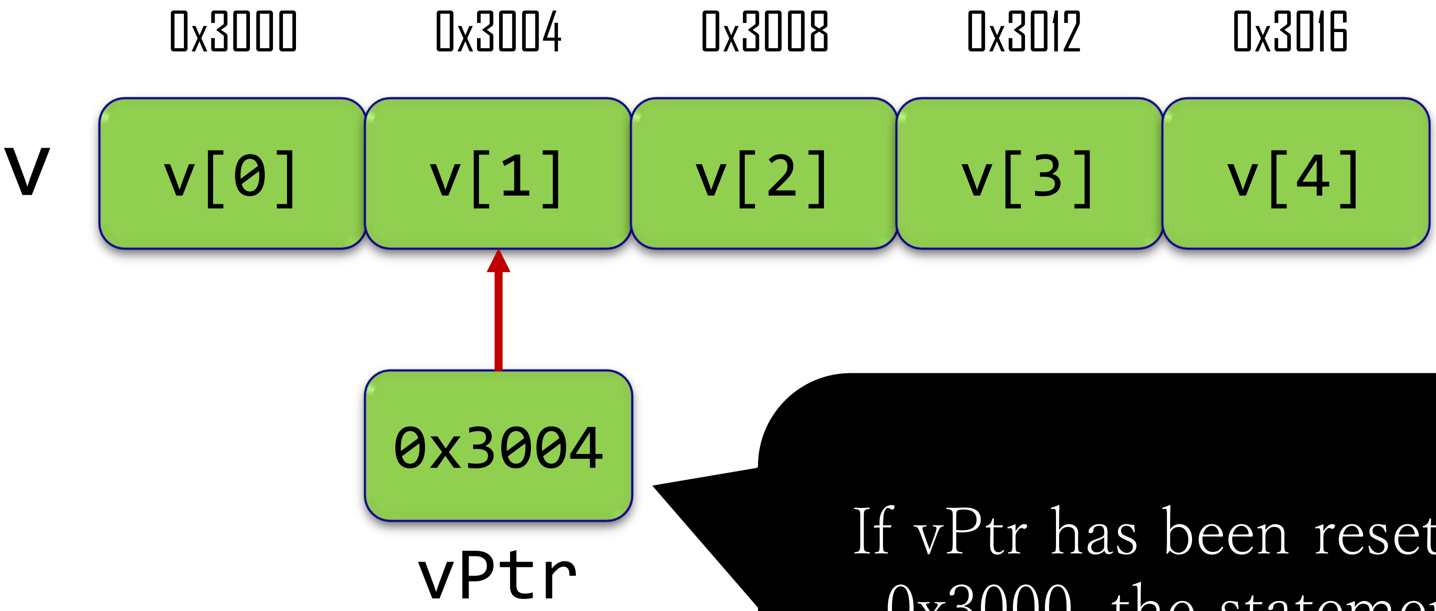


if **vPtr** has been incremented to 0x**3016**, the statement

vPtr -=4;

would set **vPtr** to 0x**3000**.

POINTER EXPRESSIONS & ARITHMETIC



If vPtr has been reset to 0x3000, the statement

`vPtr++;`

would set vPtr to 0x3004.

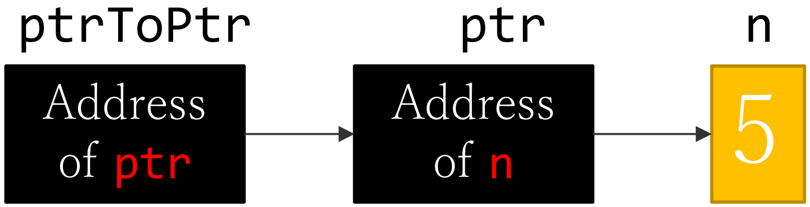
WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

POINTERS TO POINTERS

SINGAPORE INSTITUTE OF TECHNOLOGY

POINTERS TO POINTERS

```
int n = 5;
int *ptr = &n;
int **ptrToPtr = &ptr;
```



`(int *)`: pointer to an integer

`(int *)*`: pointer to a pointer which points to an integer

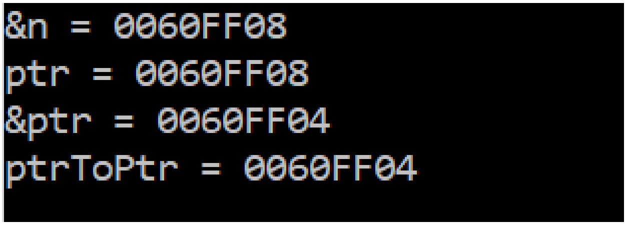
Many uses in C:

- Arrays of pointers
- Arrays of strings

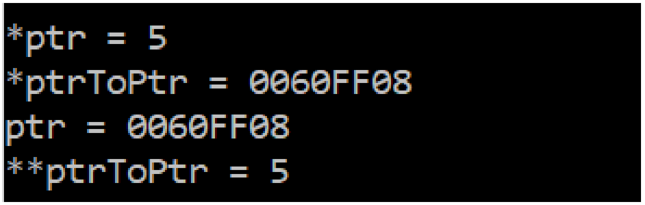
POINTERS TO POINTERS

```
#include <stdio.h>

int main() {

        int n = 5;
        int *ptr = &n;
        int **ptrToPtr = &ptr;

        printf("&n = %p\n", &n);
        printf("ptr = %p\n", ptr);
        printf("&ptr = %p\n", &ptr);
        printf("ptrToPtr = %p\n", ptrToPtr);

        /* illustrating the dereferencing operator * */
        printf("*ptr = %d\n", *ptr);
        printf("*ptrToPtr = %p\n", *ptrToPtr);
        printf("ptr = %p\n", ptr);
        printf("**ptrToPtr = %d\n", **ptrToPtr);


        return 0;

}
```

Output



```
&n = 0060FF08
ptr = 0060FF08
&ptr = 0060FF04
ptrToPtr = 0060FF04
```

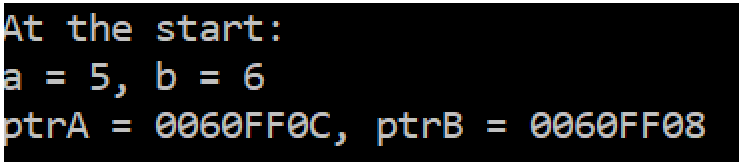


```
*ptr = 5
*ptrToPtr = 0060FF08
ptr = 0060FF08
**ptrToPtr = 5
```

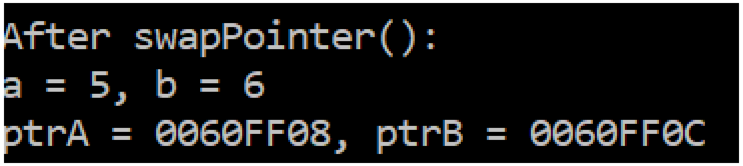
POINTERS TO POINTERS

EXAMPLE – WHAT IS THE OUTPUT OF THIS PROGRAM?

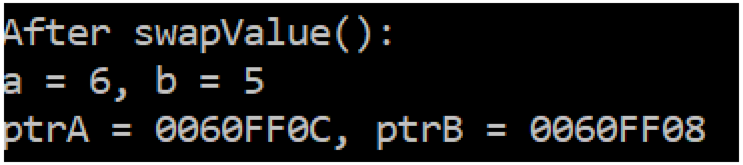
```
int main() {

           int a = 5;
           int b = 6;
           int *ptrA = &a;
           int *ptrB = &b;

           printf("At the start:\n");
           printf("a = %d, b = %d\n", a, b);
           printf("ptrA = %p, ptrB = %p\n\n", ptrA, ptrB);

           /* test swapPointer() */
           ptrA = &a;
           ptrB = &b;
           swapPointer(&ptrA, &ptrB);
           printf("After swapPointer():\n");
           printf("a = %d, b = %d\n", a, b);
           printf("ptrA = %p, ptrB = %p\n\n", ptrA, ptrB);

           /* test swapValue() */
           ptrA = &a;
           ptrB = &b;
           swapValue(ptrA, ptrB);
           printf("After swapValue():\n");
           printf("a = %d, b = %d\n", a, b);
           printf("ptrA = %p, ptrB = %p\n\n", ptrA, ptrB);

           return 0;

}
```



```
At the start:
a = 5, b = 6
ptrA = 0060FF0C, ptrB = 0060FF08
```



```
After swapPointer():
a = 5, b = 6
ptrA = 0060FF08, ptrB = 0060FF0C
```



```
After swapValue():
a = 6, b = 5
ptrA = 0060FF0C, ptrB = 0060FF08
```

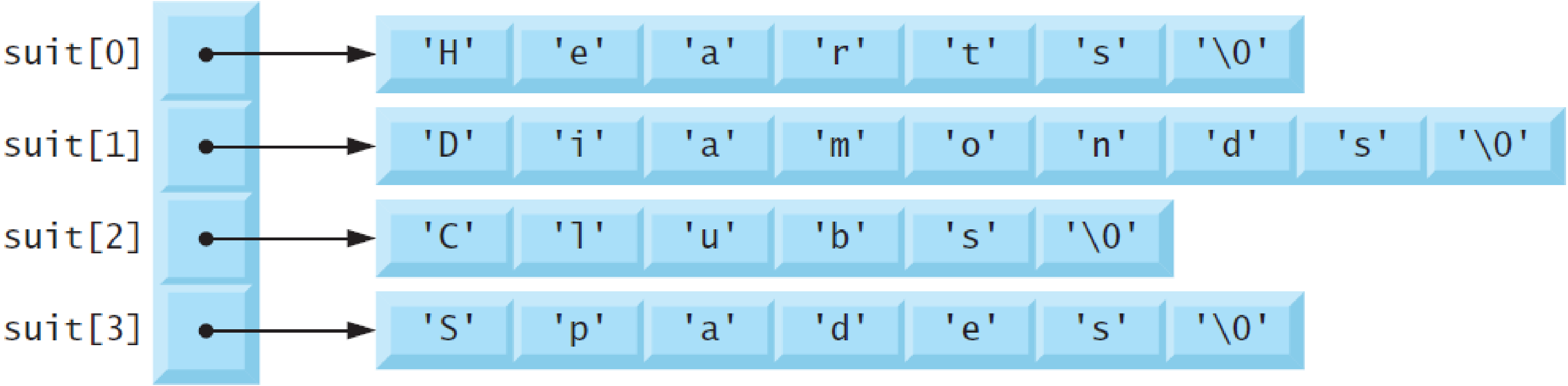
ARRAYS OF POINTERS

Arrays may contain pointers.

```
char * suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

each element is of type "pointer to char"

"an array of 4 elements"



suit[0] → 'H' 'e' 'a' 'r' 't' 's' '\0'

suit[1] → 'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'

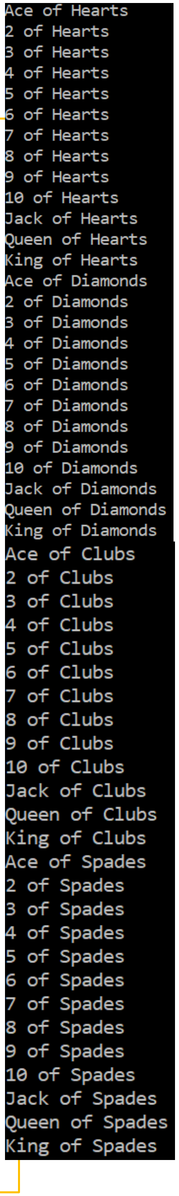
suit[2] → 'C' 'l' 'u' 'b' 's' '\0'

suit[3] → 'S' 'p' 'a' 'd' 'e' 's' '\0'

*illustration from Deitel & Deitel

ARRAYS OF POINTERS - EXAMPLE

```c
#include <stdio.h>

int main() {
    char *suit[4] = { "Hearts", "Diamonds", "Clubs", "Spades" };
    char *face[13] = {
        "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10",
        "Jack", "Queen", "King"
    };

    for (int i = 0; i < 4; i++) {
        char *card_suit = suit[i];
        for (int j = 0; j < 13; j++) {
            printf("%s of %s\n", face[j], card_suit);
        }
    }

    return 0;
}
```



```
Ace of Hearts
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Diamonds
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

CALL-BY-REFERENCE

SINGAPORE
INSTITUTE OF
TECHNOLOGY

CALLING FUNCTIONS BY VALUE

Recall call-by-value:

- A copy of the argument's value is made and passed to the called function.
- Changes to the copy do not affect the original variable's value in the caller.
- By default, all calls in C are by value.

www.mathwarehouse.com

CALLING FUNCTIONS BY REFERENCE

In call-by-reference:

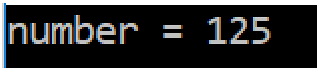
- The caller allows the called function to modify the original value.
- Call-by-reference can be simulated using a pointer in C.

www.mathwarehouse.com

FUNCTIONS – CALL-BY-REFERENCE

```
#include <stdio.h>

/* cube a number in-place */
void cubeByReference(int *);

int main() {

         int number = 5;
         cubeByReference(&number);
         printf("number = %d\n", number);

         return 0;
}

void cubeByReference(int *ptr) {

         *ptr = (*ptr) * (*ptr) * (*ptr);

}
```

Output



```
number = 125
```

Simulating call-by-reference

When calling a function with arguments that should be modified, the **addresses for the arguments** are passed.

PASSING ARRAYS TO FUNCTIONS

Suppose we have this array:

```
int a[5] = { 0, 1, 2, 3, 4 };
```

To pass an array argument to a function, specify the name of the array without any brackets:

```
modifyArray(a, 5);
```

This function call passes array `a` and its size to function `modifyArray`.

PASSING ARRAYS TO FUNCTIONS

The square brackets tell the compiler that the function expects an array.

```
void modifyArray(int b[], int size)
```

The size of the array is not required between the array brackets `[]`.

PASSING ARRAYS TO FUNCTIONS

```
/* the first argument of this function is an array of integers */
void modifyArray(int [], int);

int main() {

        int a[5] = {0, 1, 2, 3, 4};
        modifyArray(a, 5);

        return 0;

}

/* double every element of an array */
void modifyArray(int b[], int size) {

        int j;
        for (j = 0; j < size; j++)
                b[j] *= 2;

}
```

This function doubles the value of each element in the array. Will the contents of array `a` in `main` change after this function returns?

C automatically passes arrays to functions by reference.

The called function can modify the element values in the callers’ original arrays.

Output:
[Array] = 0 1 2 3 4
[Array] = 0 2 4 6 8

```
int main() {
        int a[5] = {0, 1, 2, 3, 4};
        printArray(a, 5);
        modifyArray(a, 5);
        printArray(a, 5);

        return 0;
}
/* double every element of an array */
void modifyArray(int b[], int size) {

        int j;
        for (j = 0; j < size; j++)
                b[j] *= 2;
}
```

This function doubles every element of an arry

```
void printArray(int b[], int size) {

        int j;
        printf("[Array] = ");
        for (j = 0; j < size; j++)
                printf("%d ", b[j]);
        printf("\n");
}
```

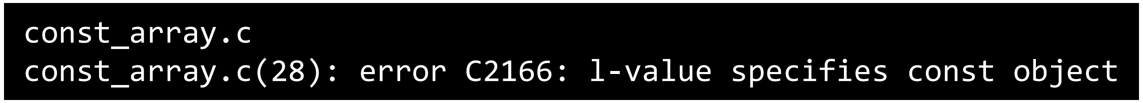
This function prints the contents of the array

Many times, you do not want a function to change the contents of the original array, so what do you do in this case?

Use `const` to prevent modification of values in an array in a functions.

```
void tryToModifyArray(const int b[], int size) {

        int j;
        for (j = 0; j < size; j++)
                b[j] *= 2;

}
```

Compiler Output



```
const_array.c
const_array.c(28): error C2166: l-value specifies const object
```

When an array parameter is preceded by the `const` qualifier, the array elements become constant in the function body, and any attempt to modify an element of the array in the function body results in a compile-time error.

END-OF-WEEK 10 CHECKLIST

- □ Pointer declarations
- □ Address operator
- □ Pointer dereferencing
- □ Pointer assignment
- □ Void pointers
- □ Pointers to pointers
- □ Pointer arithmetic
- □ Arrays & pointers
- □ Arrays of pointers
- □ Call by reference
- □ Passing arrays to functions
- □ Using const