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

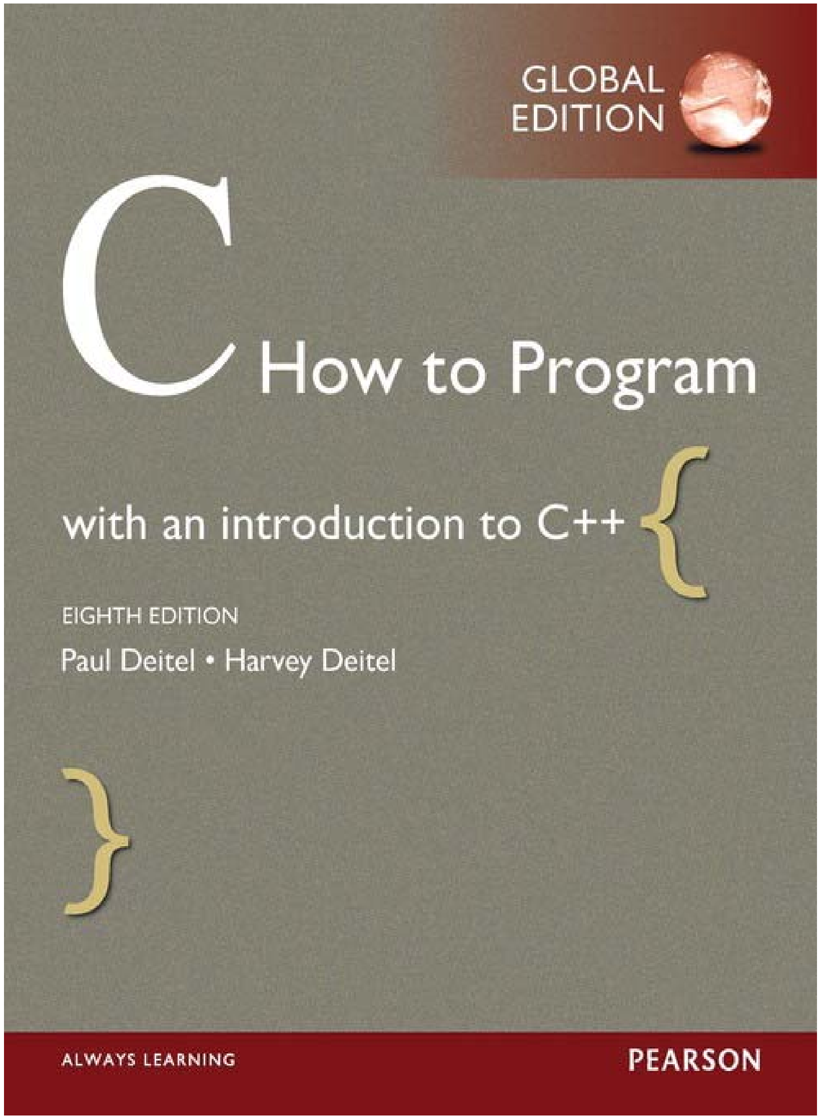
ICT1002 – Programming Fundamentals

Week 10

Agenda

1. Pointers
2. Arrays and pointers
3. User-defined data types
4. Call-by-reference

RECOMMENDED READING



Paul Deitel and Harvey Deitel, *C: How to Program*, 8th Edition, Prentice Hall, 2016

- Chapter 7: *C Pointers*
- Chapter 10: *C Structures, Unions, Bit Manipulation and Enumerations*

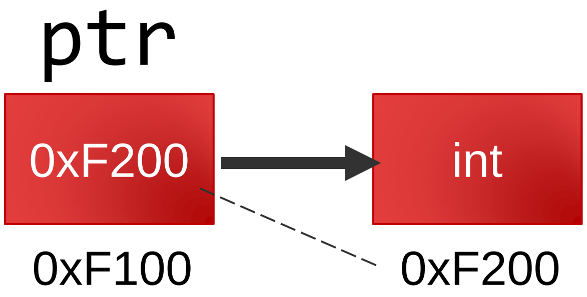
WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

POINTERS

SINGAPORE INSTITUTE OF TECHNOLOGY

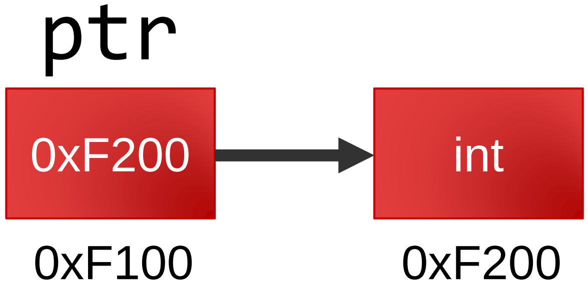
POINTER VARIABLES

Pointers are variables whose values are memory addresses.



POINTER VARIABLE DEFINITION

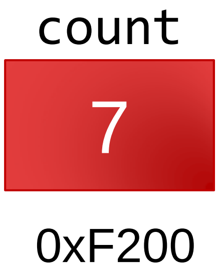
```
int *ptr;
```



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

POINTER VARIABLES

```
int count = 7;
```



count

7

0xF200

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

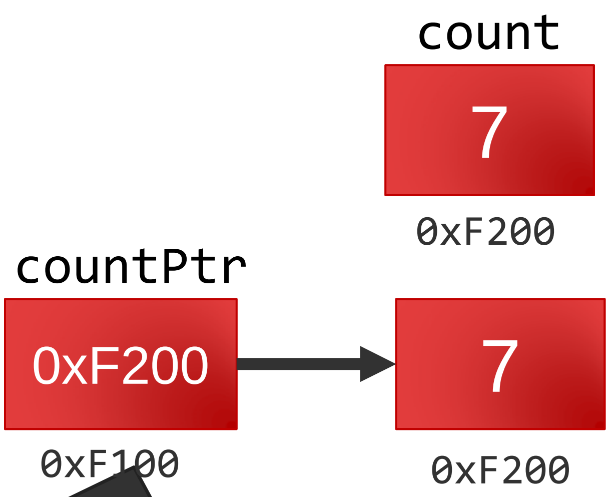
A `variable` directly contains a specific value

POINTER VARIABLES

```
int count = 7;
```

count directly references a variable that contain the value 7

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



countPtr indirectly references a variable that contains the value 7

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

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 OPERATORS

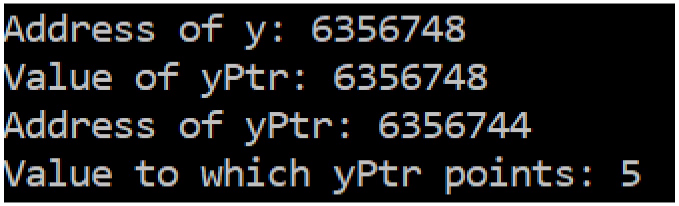
```
#include <stdio.h>

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

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

    return 0;
}
```

The & operator returns the address of its operand.

POINTER OPERATORS

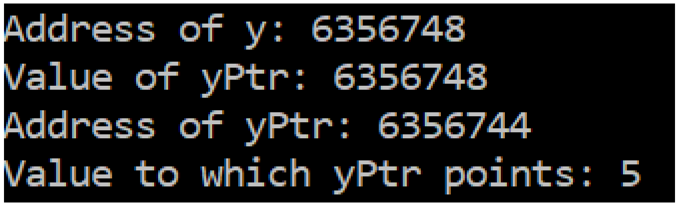
Assign the address of y to yPtr

```
#include <stdio.h>

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

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

    return 0;
}
```



```
Address of y: 6356748
Value of yPtr: 6356748
Address of yPtr: 6356744
Value to which yPtr points: 5
```

POINTER OPERATORS



```
Address of y: 6356748
Value of yPtr: 6356748
Address of yPtr: 6356744
Value to which yPtr points: 5
```

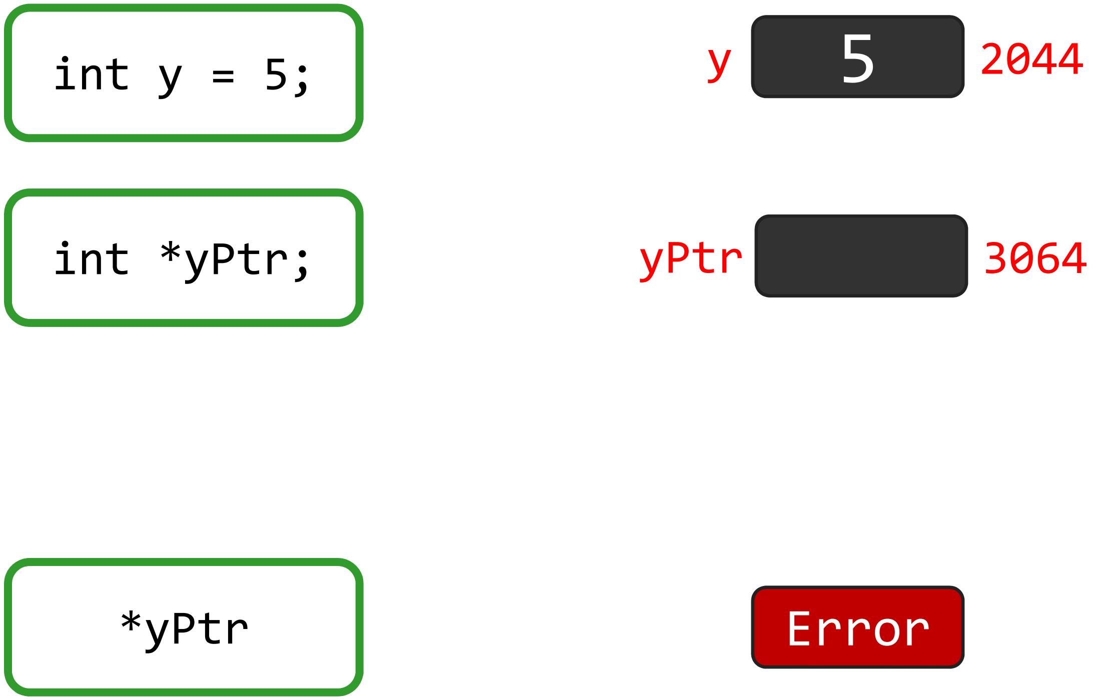
```c
#include <stdio.h>

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

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

    return 0;
}
```

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

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.



```
int y = 5;
```

y 5 2044

```
int *yPtr;
```

yPtr 3064

```
*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;           //b = ?
 p  = &b;
 b  = *p + 3;           //b = ?
```

CONFUSED NOW?

- Easier way to remember: DID

  - D: Declaration
    - int variable;
    - int *ptr;

  - I: Initialization (assignment)
    - int variable = 10;
    - ptr = &variable;

  - D: Dereference
    - *ptr = 20;
    - int a = *ptr;

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

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.

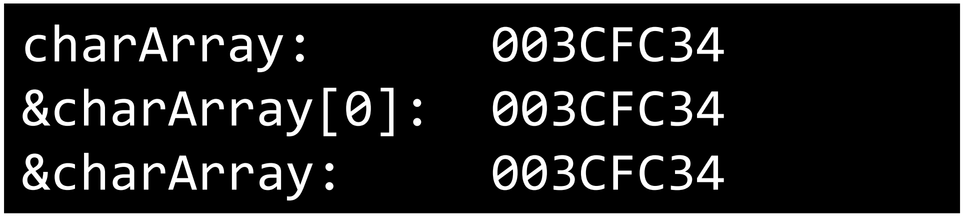
- Pointer arithmetic can be used to navigate arrays.

POINTERS & ARRAYS

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

```
int main() {

        char charArray[5];

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

        return 0;
}
```

Output



```
charArray:      003CFC34
&charArray[0]:  003CFC34
&charArray:     003CFC34
```

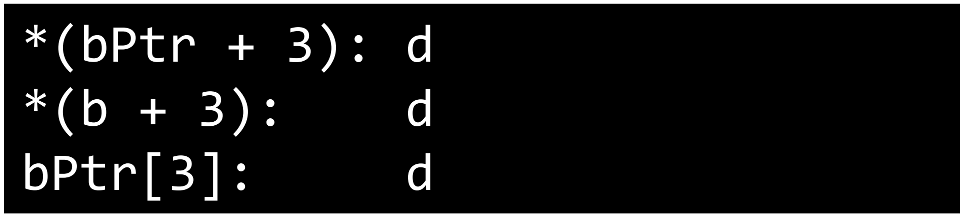
array VS &array: https://www.geeksforgeeks.org/whats-difference-between-array-and-array-for-int-array5/

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.

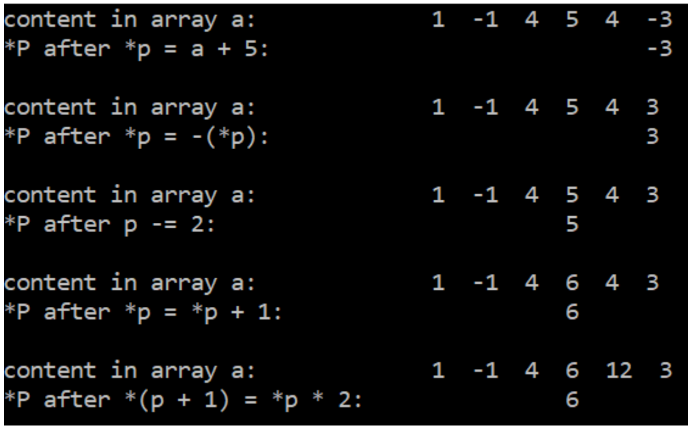
```
bPtr[3]
```

pointers can be subscripted exactly as arrays can.

EXERCISE

What are the contents of the array a and the position of the pointer p 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

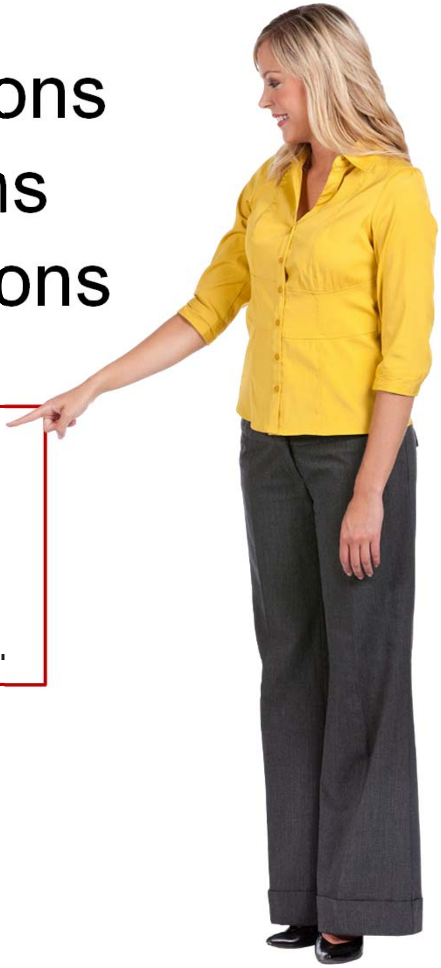
SiT SINGAPORE INSTITUTE OF TECHNOLOGY

POINTER EXPRESSIONS & ARITHMETIC

In general, pointers are valid operands in

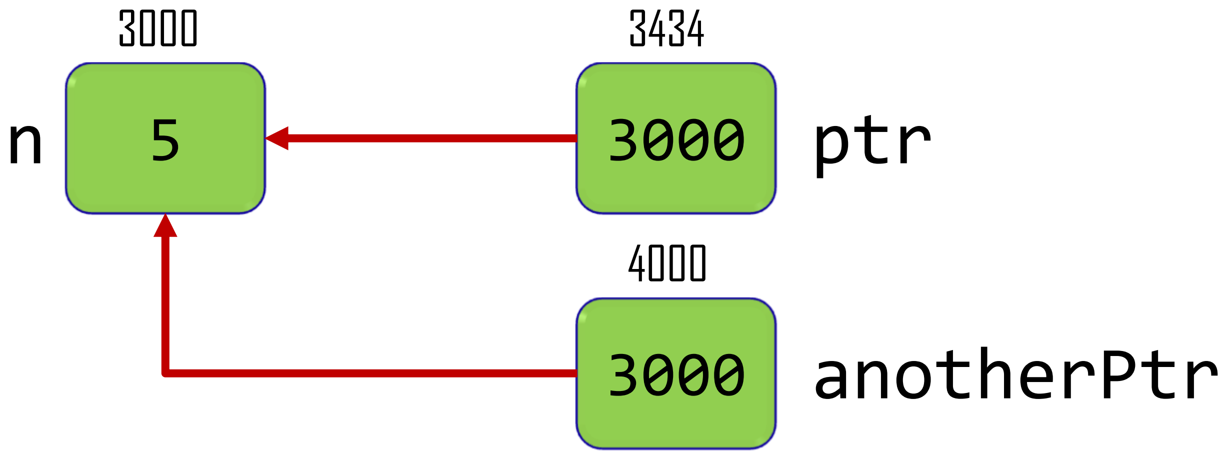
- assignment expressions
- arithmetic expressions
- comparison expressions

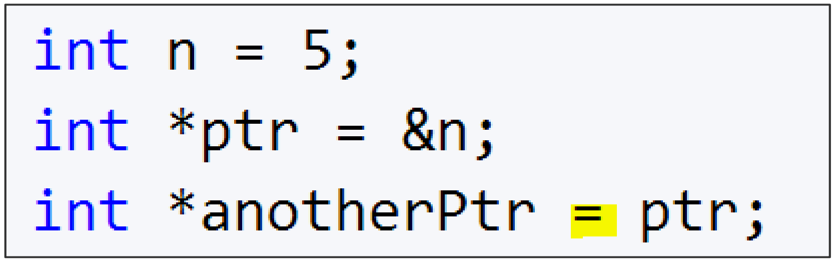
However, not all the operators normally used in these expressions are valid in conjunction with pointer variables.



POINTER ASSIGNMENT

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

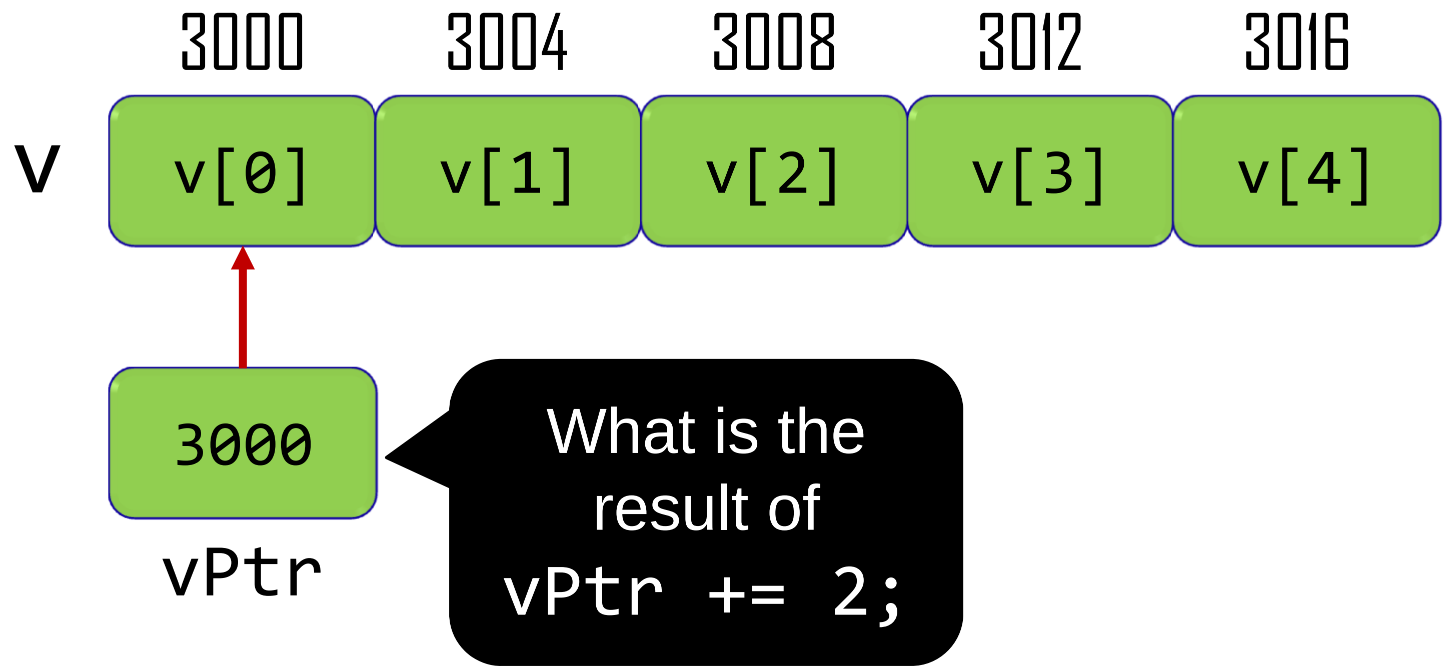




```
int n = 5;
int *ptr = &n;
int *anotherPtr = ptr;
```

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

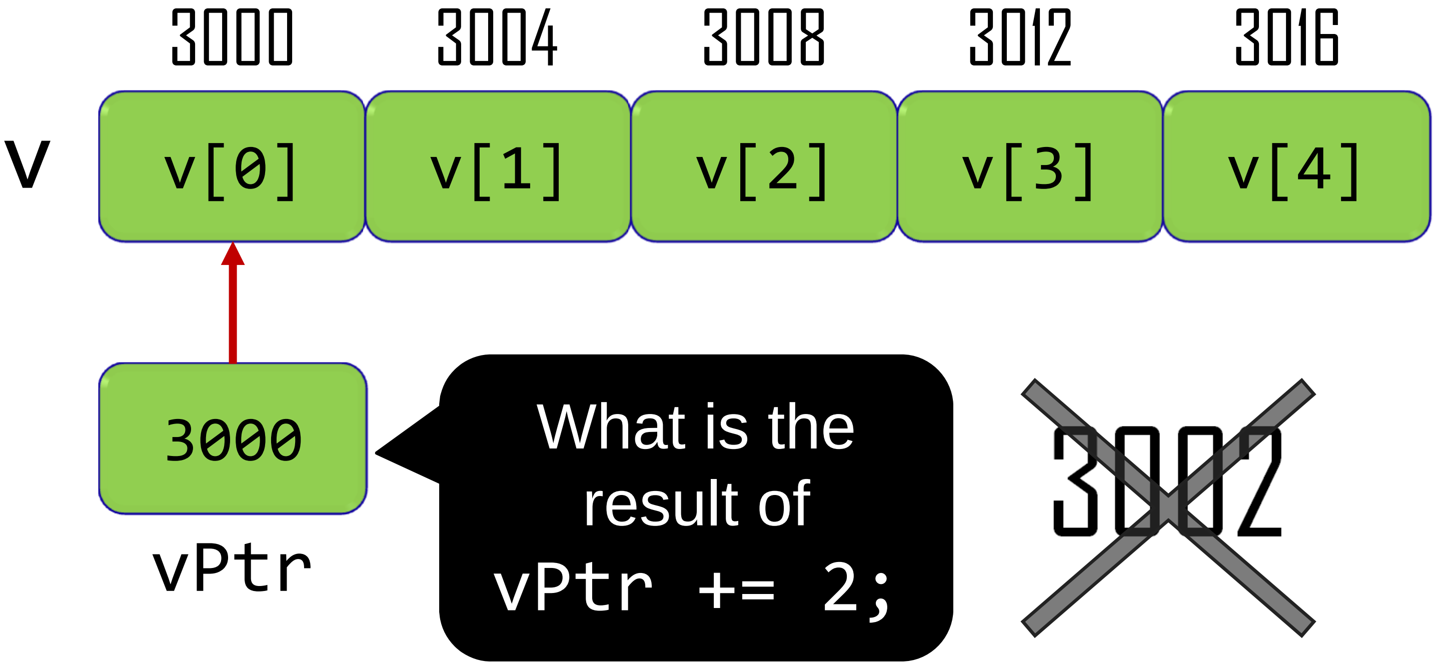
POINTER EXPRESSIONS & ARITHMETIC



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

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

POINTER EXPRESSIONS & ARITHMETIC



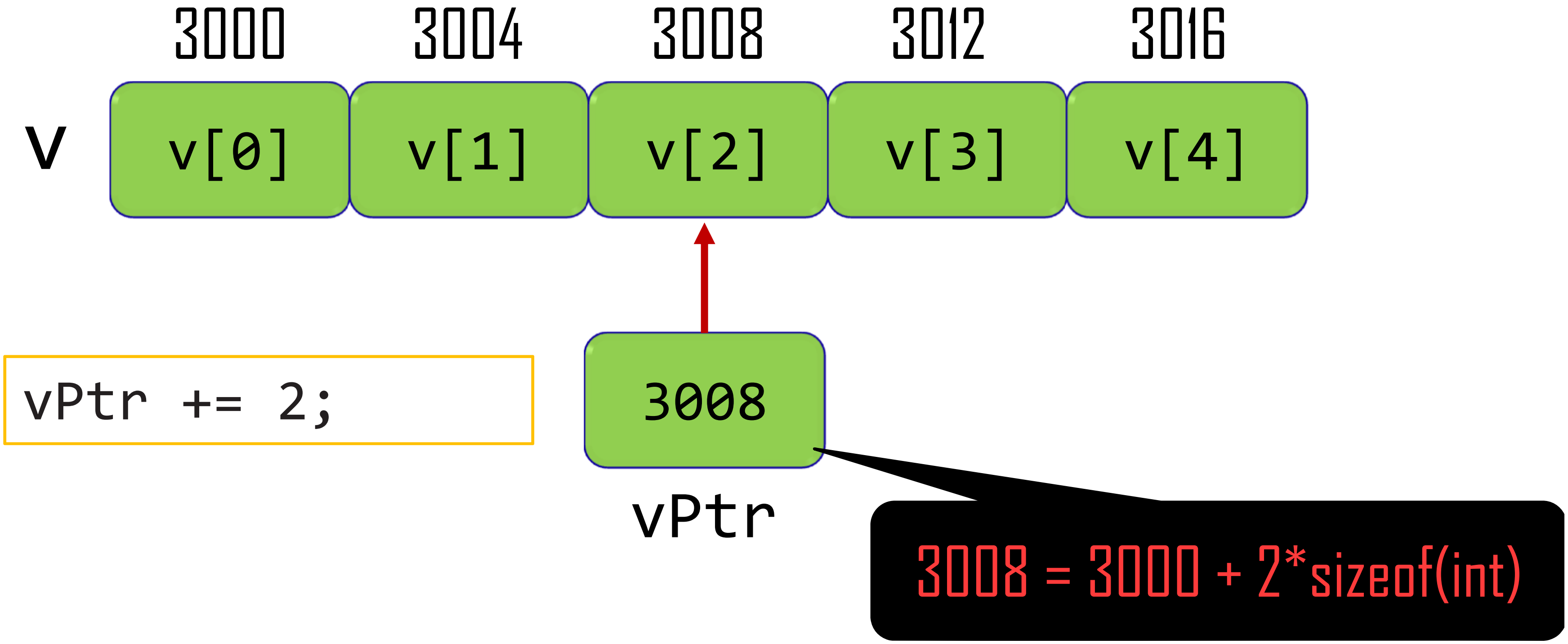
What is the result of `vPtr += 2;`

~~3002~~

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

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

POINTER EXPRESSIONS & ARITHMETIC



	3000	3004	3008	3012	3016
v	v[0]	v[1]	v[2]	v[3]	v[4]

```
vPtr += 2;
```

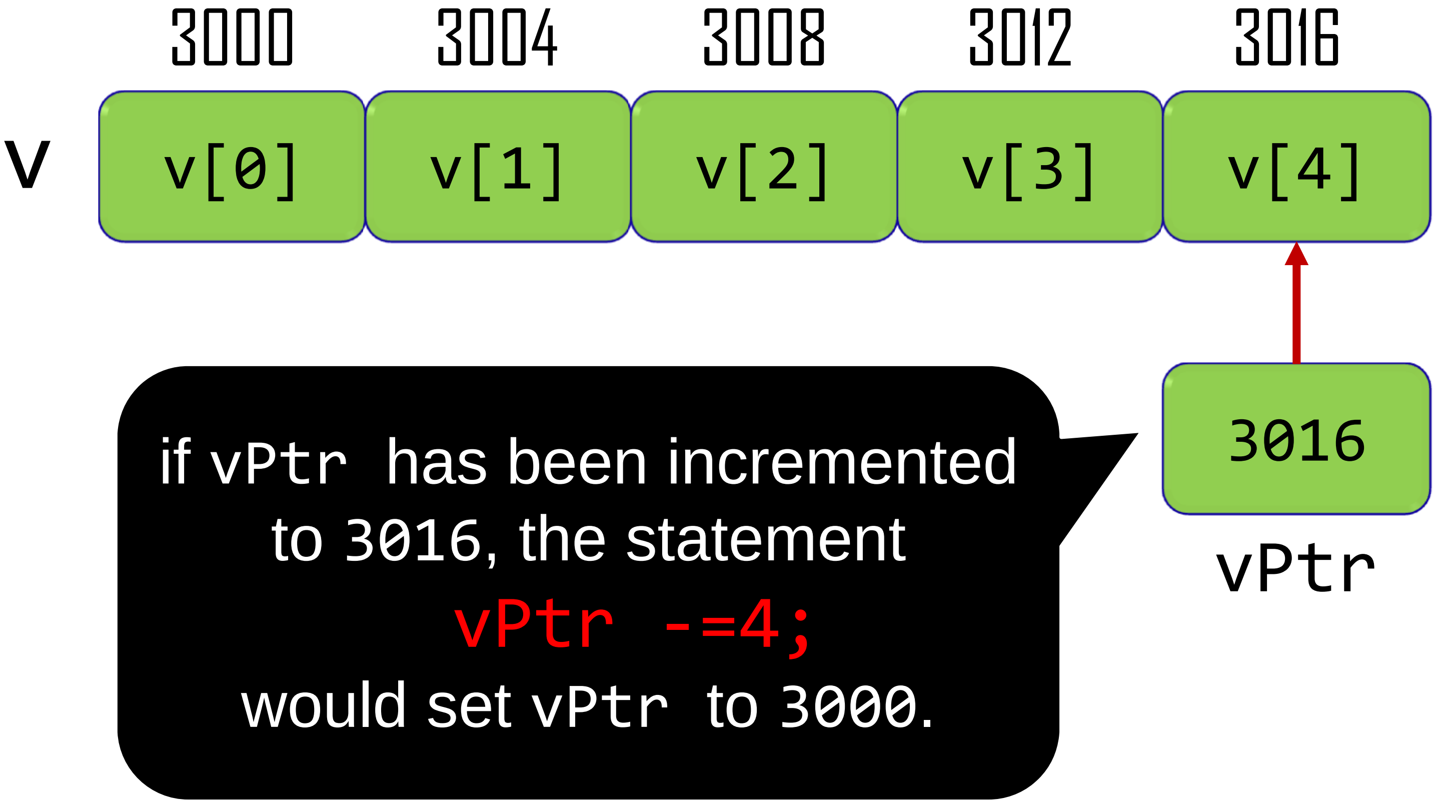
3008

vPtr

3008 = 3000 + 2*sizeof(int)

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

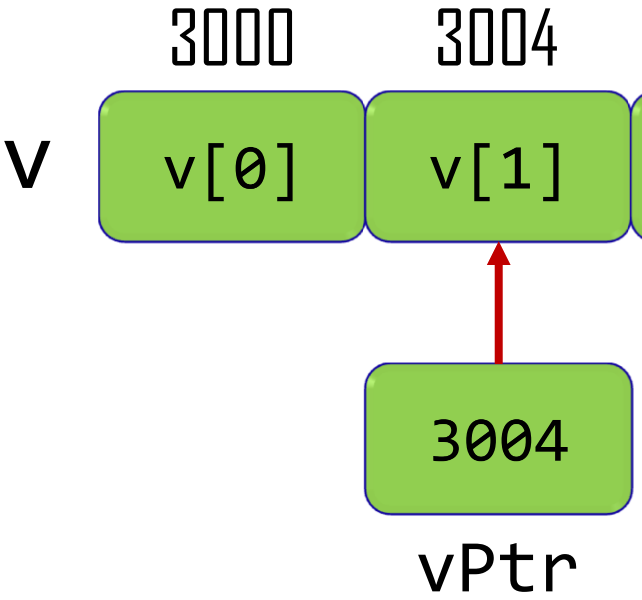


if vPtr has been incremented to 3016, the statement

`vPtr -=4;`

would set vPtr to 3000.

POINTER EXPRESSIONS & ARITHMETIC



If vPtr has been reset to 3000, the statement

`vPtr++;`

would set vPtr to 3004.

CONFUSED NOW?

- Easier way to remember: DID
  - D: Declaration
    - int *ptr;
  - I: Initialization (assignment)
    - int variable = 10;
    - ptr = &variable;
  - D: Dereference
    - *ptr = 20;

EXERCISE

– DID for the following statements

```
*abc = 100;

float *xyz;

ptr = &va;

int *a = &vb;
```

10 Minutes Break

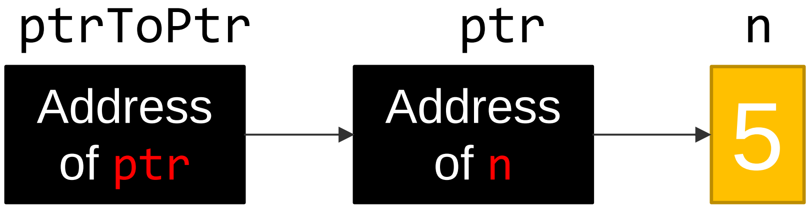
WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

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

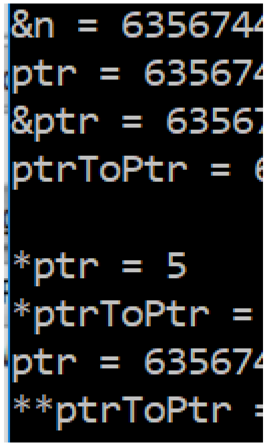
Many uses in C:

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

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

         //printf("**ptrToPtr = %d\n", *(*ptrToPtr));

         return 0;

}
```

Output



```
&n = 6356744
ptr = 635674
&ptr = 63567
ptrToPtr = 6

*ptr = 5
*ptrToPtr =
ptr = 635674
**ptrToPtr =
```

POINTERS TO POINTERS

EXAMPLE

What does `swapPointer` do?

How is it different from `swapValue`?

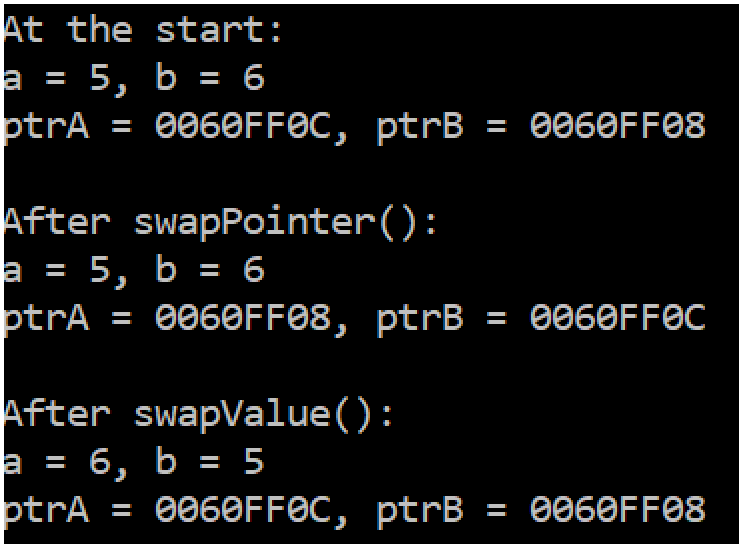
```
void swapPointer(int **a, int **b) {
        int *temp = *a;
        *a = *b;
        *b = temp;
}
```

```
void swapValue(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

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
```

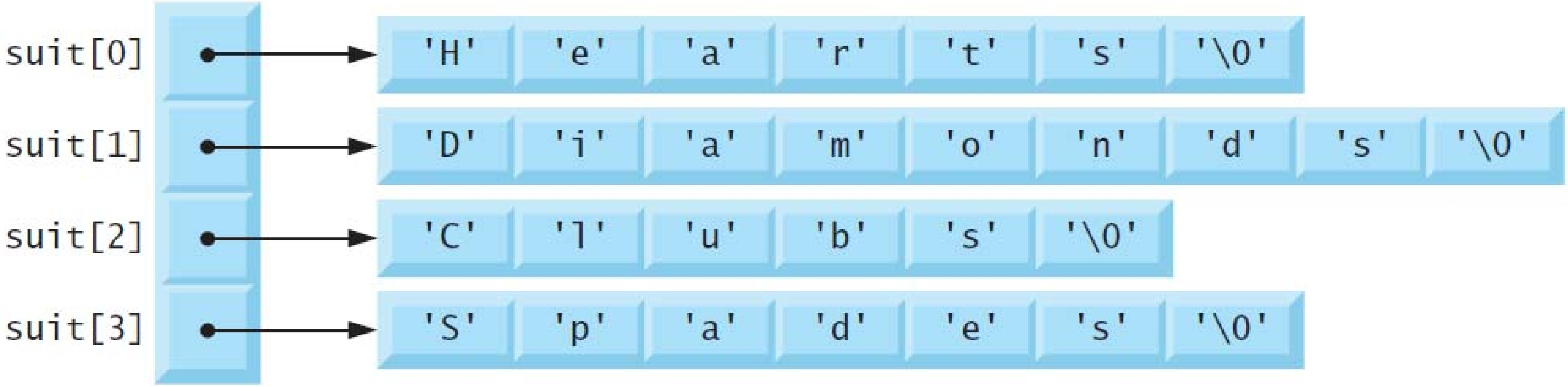
ARRAYS OF POINTERS

Arrays may contain pointers.

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

each element is of type “pointer to char”

“an array of 4 elements”

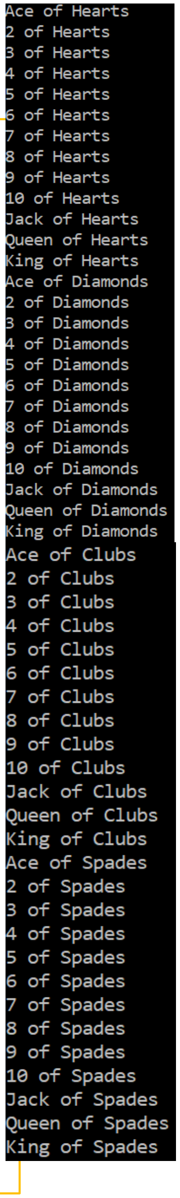


suit[0]
'H' 'e' 'a' 'r' 't' 's' '\0'
suit[1]
'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'
suit[2]
'C' 'l' 'u' 'b' 's' '\0'
suit[3]
'S' 'p' 'a' 'd' 'e' 's' '\0'

*illustration from Deitel & Deitel

ARRAYS OF POINTERS - EXAMPLE

```
#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
```

VOID POINTERS

```
int x;
void *xPtr = &x;
printf("xPtr: %p\n", xPtr);

float f;
void *fPtr = &f;
printf("fPtr: %p\n", fPtr);
```

All pointers can be assigned to a pointer to `void`.

A pointer to `void` can point to a variable of any type.

VOID POINTERS

```c
float f = 123.45;

/* incorrect */
void *fPtr = &f;
printf("*fPtr: %f\n", *fPtr);

/* correct */
float *fPtr2 = (float *)fPtr;
printf("*fPtr2: %0.2f\n", *fPtr2);
```

The compiler says:

```
void_pointers.c
void_pointers.c(16): error C2100:
illegal indirection
```

A pointer to void `cannot be dereferenced`. Void pointers should always be `cast` before dereferencing.

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

USER-DEFINED DATA TYPES

SINGAPORE INSTITUTE OF TECHNOLOGY

STRUCTURES

Suppose you want to represent this information about a `student`.

Name	Sachin Kumar
Roll	101
Age	16
Class	ICT1002

STRUCTURES

C allows structured collections of information to be defined using the struct keyword.

```
struct <name> {
   member 1
   member 2
   :
   member n
};
```

STRUCTURES - EXAMPLE

```
struct student {
   char name[20];
   int roll;
   int age;
   char class[12];
} student_x, student_y;

struct student student_z;
```

The code above declares three variables of type `struct student`, called `student_x`, `student_y`, and `student_z`

STRUCTURES – EXAMPLE

```
/*
 * struct example from Sharma
 */
#include <stdio.h>

struct student {
         char name[20];
         int roll;
         int age;
         char class[12];
};

int main() {

         /* initialise a variable of type student */
         struct student stud1 = { "Sachin Kumar", 101, 16, "ICT1002" };

         /* display contents of stud1 */
         printf("\n Name : %s", stud1.name);
         printf("\n Roll : %d", stud1.roll);
         printf("\n Age  : %d", stud1.age);
         printf("\n Class: %s", stud1.class);

         return 0;

}
```

Output

```
Name : Sachin Kumar
Roll : 101
Age  : 16
Class: ICT1002
```

Structures can be initialised similar to arrays.

Use the dot operator to refer to members of a structure.

USER-DEFINED DATA TYPES - TYPEDEF

```
typedef <type> <new_type>
```

User-defined data types can be declared using `typedef`:

`<type>` can be a basic data type or struct

`<new_type>` is the user-defined data type

USER-DEFINED DATA TYPES - TYPEDEF

```
typedef float salary;
salary wages_of_month;
```

In this example wages_of_month is of type salary which is a float by itself.
This enhances the readability of the program.

USER-DEFINED DATA TYPES - TYPEDEF

```
struct student {
   char name[20];
   int roll;
   int age;
   char class[12];
};

typedef (struct student) Student;

/* initialise a variable of type Student */
Student stud1 = { "Sachin Kumar", 101, 16, "ICT1002" };
```

USER-DEFINED DATA TYPES - TYPEDEF

```
typedef struct {
   char name[20];
   int roll;
   int age;
   char class[12];
} Student;

/* initialise a variable of type student */
Student stud1 = { "Sachin Kumar", 101, 16, "ICT1002" };
```

USER-DEFINED DATA VALUES

Many programmers use `#define` to give symbolic names to numeric codes.

```
#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
...
#define EDOM        33  /* Math argument out of domain */
#define ERANGE      34  /* Math result not representable */
```

<errno.h> (gcc)

```
double r = sqrt(n);
if (errno == EDOM)
    printf("%f does not have a square root.\n", n);
```

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

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:

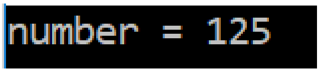
- The caller allows the called function to modify the original value.
- Call-by-reference can be simulated using a pointer in C.

pass by reference
pass by value
cup =
cup =
fillCup( )
fillCup( )
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

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

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 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.

```
int main() {

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

        return 0;

}
```

Output:

```
[Array] = 0 1 2 3 4
[Array] = 0 2 4 6 8
```

```
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

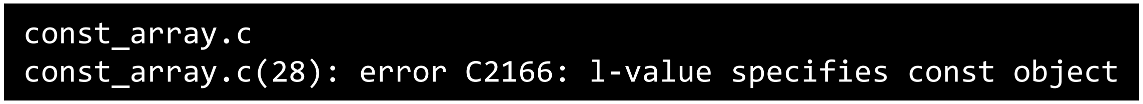
Many times, you do not want a function to change the contents of the original array, so what do you do in this case?

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;

}
```

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.

Compiler Output



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

PASSING STRUCTURES TO FUNCTIONS

```
/*
 * struct example with functions
 */
#include <stdio.h>

void print_student(Student *s);

int main() {

        /* initialise a variable of type Student */
        Student stud1 = { "Sachin Kumar", 101, 16, "ICT1002" };

        /* display contents of stud1 */
        print_student(&stud1);

        return 0;
}

void print_student(Student *s) {

        printf("\n Name : %s", s->name);
        printf("\n Roll : %d", s->roll);
        printf("\n Age  : %d", s->age);
        printf("\n Class: %s", s->class);

}
```

Structures can be passed by reference.

Use the arrow operator to de-reference a pointer to a structure.

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

□ User-defined data types

□ Dot and arrow operators

Self-assessment (for practice only):

Socrative: https://b.socrative.com/login/student
Room: ICT1002