

*Spring 2018*

[illegible]

**TAs:** Feyza Nur Kılıçaslan, Nebi Yılmaz, Özge Yalçinkaya, Gültekin Işık

# Today

- **Introduction**

- About the class
- Organization of this course

- **C Review & Pointers**

- Variables
- Operators
- Control Structures
- Functions
- Pointers

# Today

- **Introduction**

- About the class
- Organization of this course

- **C Review & Pointers**

- Variables
- Operators
- Control Structures
- Functions
- Pointers

# About the Course

- We will start the term by recapping what you have learned about the C language in the previous term.
- We will also be covering some additional subjects that are;
  - Dynamic Memory Management
  - Structs
  - File I/O

# About the Course

- This course will help students understand object-oriented programming principles and apply them in the construction of Java programs.
  - The course is structured around basic topics such as **classes, objects, encapsulation, inheritance, polymorphism, abstract classes and interfaces and exception handling.**
- **BBM 104 Introduction to Programming Practicum:** The students will gain hands-on experience via a set of programming assignments supplied as complementary.
- **Requirements:** You must know basic programming (i.e. BBM101).

# About the Course

## Instructors



Gönenç Ercan  
(Section 1)



Öner Barut  
(Section 2)



Cumhuriyet Yiğit ÖZCAN  
(Section 3)



Ali Seydi Keçeli  
(Section 4)

## TAs



Gültekin Işık



Nebi Yılmaz



Feyza Nur Kılıçaslan

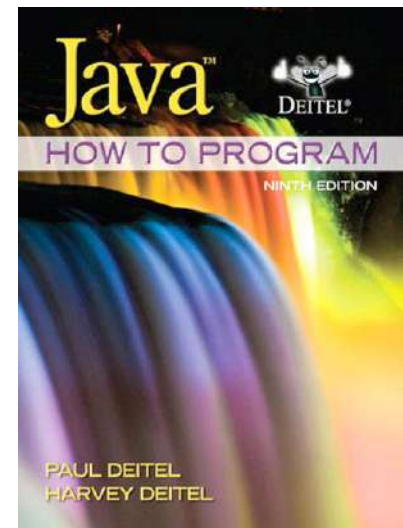
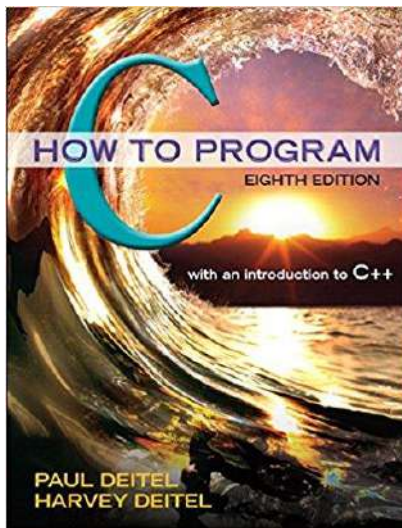


Özge Yalçınkaya

- Office Hours: See the web page

# Reference Book

- C – How to Program, Paul Deitel and Harvey Deitel, Pearson, 2016
- Java – An Introduction to Problem Solving And Programming, Walter Savitch, Pearson, 2012
- Java – How to Program, Paul Deitel and Harvey Deitel Prentice Hall, 2012



# Communication



- The course web page will be updated regularly throughout the semester with lecture notes, programming assignments, announcements and important deadlines.

<http://web.cs.hacettepe.edu.tr/~bbm102>



# Getting Help

- **Office Hours**

- See the web page for details

- **BBM 104 Introduction to Programming Practicum**

- Course related recitations, practice with example codes, etc.

- **Communication**

- Announcements and course related discussions through

BBM 102: <http://piazza.com/hacettepe.edu.tr/spring2018/bbm102/home>

BBM 104: <http://piazza.com/hacettepe.edu.tr/spring2018/bbm104/home>

# Course Work and Grading

- **Two Midterm Exams (20 + 30 = 50%)**
  - Closed book and notes
- **Final Exam (50%)**
  - Closed book
  - To be scheduled by the registrar.
- **Class Attendance (5%)**
  - Attempting to create false attendance (e.g., singing in the attendance list on behalf of someone else) will be punished.

# Course Work and Grading

Week	Date	Title
1	14-Feb	C Review & Pointers
2	21-Feb	Arrays & Dynamic Memory
3	28-Feb	Structs & File Input/Output
4	7-Mar	C Wrap Up
5	14-Mar	Java Introduction
6	21-Mar	Classes & Objects, Encapsulation
7	28-Mar	Inheritance
8	4-Apr	Midterm Exam 1
9	11-Apr	Polymorphism
10	18-Apr	Abstract Classes & Interfaces
11	25-Apr	Collections
12	2-May	Exceptions
13	9-May	Midterm Exam 2
14	16-May	Streams & Input Output

# **BBM 104 Introduction to Programming Practicum**

- **Programming Assignments (PAs)**
  - Four assignments throughout the semester.
  - Each assignment has a well-defined goal such as solving a specific problem.
  - You must work alone on all assignments stated unless otherwise.
- **Important Dates**
  - See the course web page for the schedule.

# Policies

- **Work Groups**

- You must work alone on all assignments unless stated otherwise.

- **Submission**

- Assignments due at 23.59 (no extensions!)
- Electronic submissions (no exceptions!)

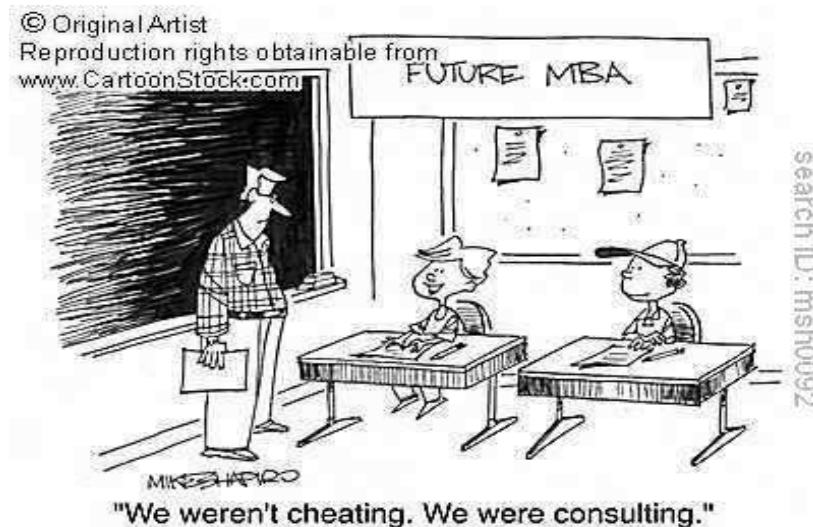
- **Lateness Penalties**

- Get penalized 10% per day
- No late submission is accepted 3 days after due date

# Cheating

- **What is cheating?**

- Sharing code: by copying, retyping, looking at, or supplying a file
- Coaching: helping your friend to write a programming assignment, line by line
- Copying code from previous course or from elsewhere on WWW



- **What is NOT cheating?**

- Explaining how to use systems or tools
- Helping others with high-level design issues

# Cheating

- **Penalty for cheating:**

- Removal from the course with failing grade.

- **Detection of Cheating:**

- We do check: Our tools for doing this are much better than most cheaters think!



# Today

- **Introduction**

- About the class
- Organization of this course

- **C Review & Pointers**

- Variables
- Operators
- Control Structures
- Functions
- Pointers



# A Simple C Program

```
/* Hello World! Example */
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("Hello World!\n");
```

```
    return 0;
```

```
}
```

```
Hello world!
```

# Variable Declarations

- A declaration consists of a data type name followed by a list of (one or more) variables of that type:

```
char c;  
int ali, bora;  
float rate;  
double trouble;
```

- A variable may be initialized in its declaration:

```
char c = 'a';  
int a = 220, b = 448;  
float x = 1.23e-6;          /*0.00000123*/  
double y = 27e3;           /*27,000*/
```

# Variable Declarations

- Variables that are not initialized may have garbage values.

```
#include <stdio.h>
```

```
int main()
{
    int a;
    double b;
    char c;
    float d;

    printf("int: %d \n", a);
    printf("double: %lf \n", b);
    printf("char: %c \n", c);
    printf("float: %f \n", d);
    return 0;
}
```

```
int: 2
double: 0.000000
Char: a
float: 0.000000
```

# Operators

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int a = 5, b = 6, c;
```

```
    float d;
```

```
    a = a - 1;
```

```
    b += 3;
```

```
    c = b / a;
```

```
    d = b / (float)a;
```

```
    printf("a: %d \n", a);
```

```
    printf("b: %d \n", b);
```

```
    printf("c: %d \n", c);
```

```
    printf("d: %f \n", d);
```

```
    return 0;
```

```
}
```

a: 4

b: 9

c: 2

d: 2.250000

# Control Structures

- **Selection Statements**

- C provides three types of selection structures in the form of statements
- The **if** selection statement either selects (performs) an action if a condition is true or skips the action if the condition is false.
- The **if..else** selection statement either performs an action if a condition is true and performs a different action if the condition is false.
- The **switch** selection statement performs one of many different actions depending on the value of an expression.

# Control Structures

- **Iteration Statements**

- C provides three types of iteration structures in the form of statements
- The **while** iteration statement keeps performing an action while a condition is true.
- The **do..while** iteration statement keeps performing an action while a condition is true. In contrast to while, it will perform the action at least once even if the condition is false initially.
- The **for** iteration statement keeps performing an action while a condition is true. In contrast to while it involves two special parts, one for an action to be performed initially, and the other for to be performed at the end of every iteration.

# Control Structures

- if Statement

```
if (x < 0)
{
    printf("negative\n");
}
```

- if..else Statement

```
if (first > second)
    max = first;
else
    max = second;
```

# Control Structures

- Multiple Conditions

```
disc = b * b - 4 * a * c;
if (disc < 0)
{
    num_sol = 0;
}
else
{
    if (disc == 0)
    {
        num_sol = 1;
    }
    else
    {
        num_sol = 2;
    }
}
```

Notice that the **else** clause here holds just one statement (an **if..else** statement), so we can omit the braces around it.



# Control Structures

- Multiple Conditions

```
disc = b * b - 4 * a * c;  
if (disc < 0)  
{  
    num_sol = 0;  
}  
else  
    if (disc == 0)  
    {  
        num_sol = 1;  
    }  
    else  
    {  
        num_sol = 2;  
    }
```

Actually all of the other if and else clauses also hold just one statement. Therefore we can get rid of all the braces.

# Control Structures

- Multiple Conditions

```
disc = b * b - 4 * a * c;  
if(disc < 0)  
    num_sol = 0;  
else  
    if(disc == 0)  
        num_sol = 1;  
    else  
        num_sol = 2;
```

```
disc = b * b - 4 * a * c;  
if(disc < 0)  
    num_sol = 0;  
else if(disc == 0)  
    num_sol = 1;  
else  
    num_sol = 2;
```

We can remove unnecessary whitespaces (space, tab, newline etc.) to re-arrange the code into a form which is easier to read.

# Control Structures

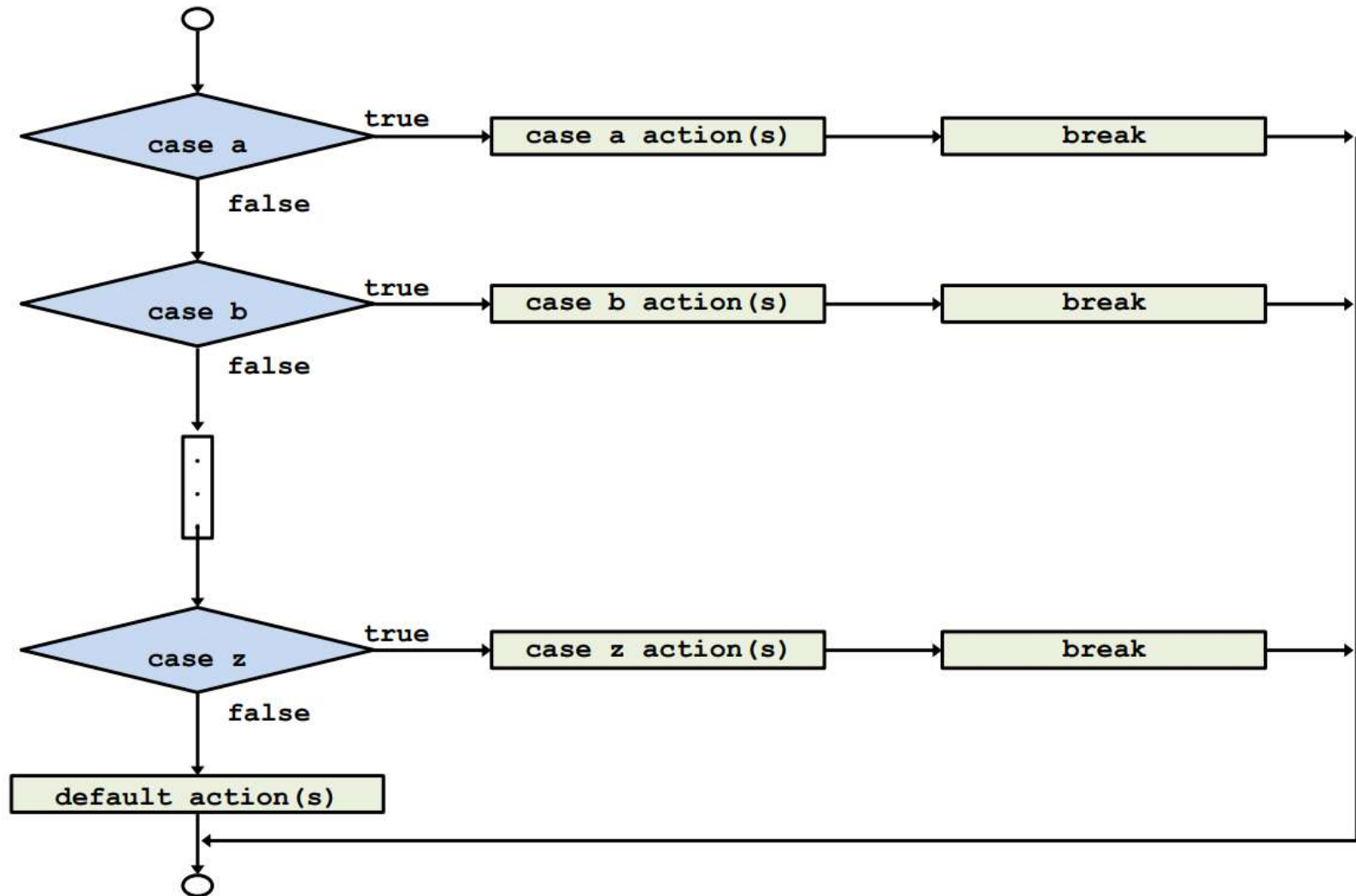
- **Switch Statement**

- Useful when a variable or expression is tested for all the values it can assume and different actions are taken
- Series of case labels and an optional default case

```
switch (a_variable or expr)
{
    case value1: actions
    case value2 : actions
    ...
    default: actions
}
```

- **break;** exits from the structure

# Control Structures



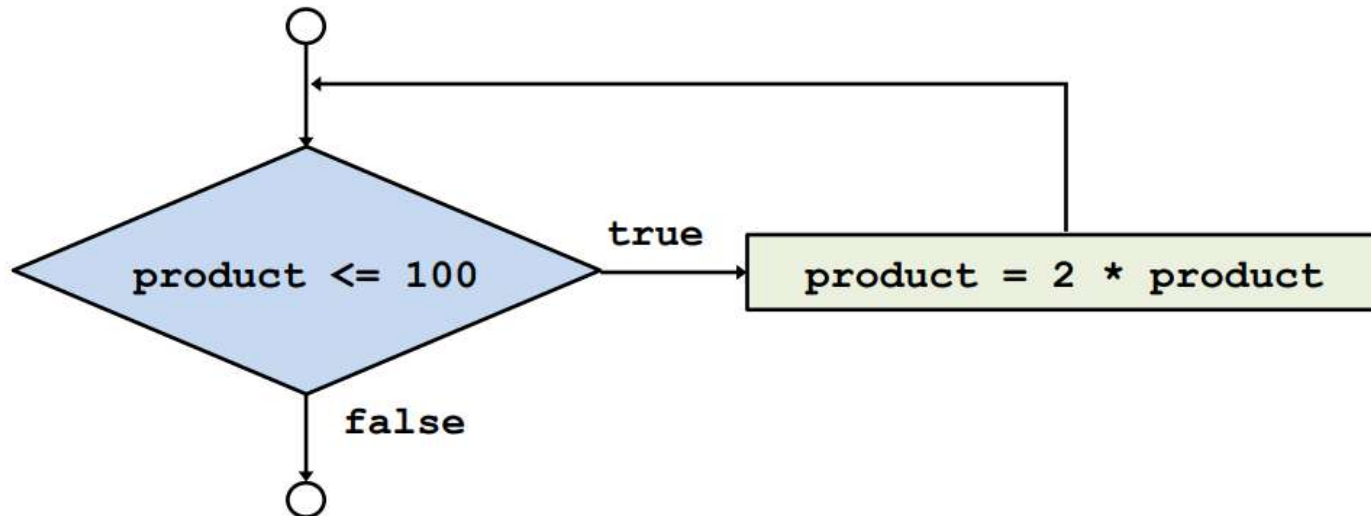
# Control Structures

```
#include <stdio.h>
int main()
{
    int month, year, days, leapyear;
    printf("Enter a month and a year:");
    scanf("%d %d", &month, &year);
    if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))
        leapyear = 1;
    else
        leapyear = 0;
    switch(month) {
        case 4 :
        case 6 :
        case 9 :
        case 11: days = 30; break;
        case 2 :
            if(leapyear == 1)
                days = 29;
            else
                days = 28;
            break;
        default :
            days = 31;
    }
    printf("There are %d days in that month in that year.\n", days);
    return 0;
}
```

# Control Structures

- While Loop

```
int product = 2;  
while (product <= 100)  
    product = 2 * product;
```



# Control Structures

- A class of 10 students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz
- The algorithm
  1. Set total to 0
  2. Set counter to 1
  3. If the counter is less than or equal to 10 go to 4, else go to 8
  4. Input the next grade
  5. Add the grade into the total
  6. Add 1 to the counter
  7. Go to 3
  8. Set the class average to the total divided by ten
  9. Print the class average

# Control Structures

```
/* Class average program with counter-controlled repetition */  
#include <stdio.h>
```

```
int main()  
{  
    int counter, grade, total, average;  
    /* initialization phase */  
    total = 0;  
    counter = 1;  
    /* processing phase */  
    while (counter <= 10) {  
        printf("Enter grade: ");  
        scanf("%d", &grade);  
        total = total + grade;  
        counter = counter + 1;  
    }  
  
    /* termination phase */  
    average = total / 10.0;  
    printf("Class average is %d\n", average);  
  
    return 0; /* indicate program ended successfully */  
}
```

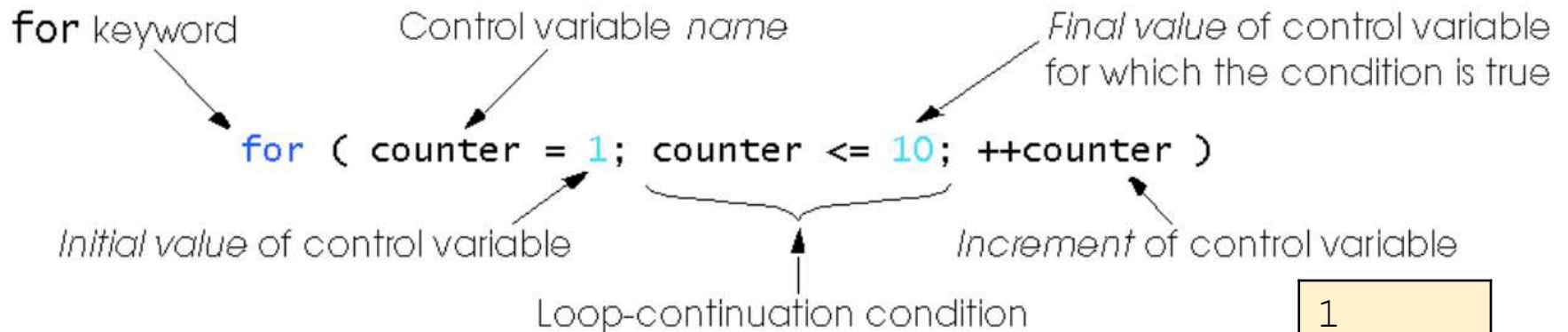
```
Enter grade: 98  
Enter grade: 76  
Enter grade: 71  
Enter grade: 87  
Enter grade: 83  
Enter grade: 90  
Enter grade: 57  
Enter grade: 79  
Enter grade: 82  
Enter grade: 94  
Class average is 81
```



# Control Structures

- For Loop

- Format when using **for** loops



- Example

```
for(counter = 1; counter <= 10; counter++)  
    printf("%d\n", counter);
```

1
2
3
4
5
6
7
8
9
10

# Control Structures

- Initialization and increment can be comma-separated lists

```
for (i = 0, j = 0; j + i <= 10; j++, i++)  
    printf("%d\n", j + i);
```

- Example: Print the sum of all numbers from 2 to 100

```
/*Summation with for */  
#include <stdio.h>  
  
int main()  
{  
    int sum = 0, number;  
    for (number = 2; number <= 100; number += 1)  
        sum += number;  
    printf("Sum is %d\n", sum);  
    return 0;  
}
```

Sum is 5049

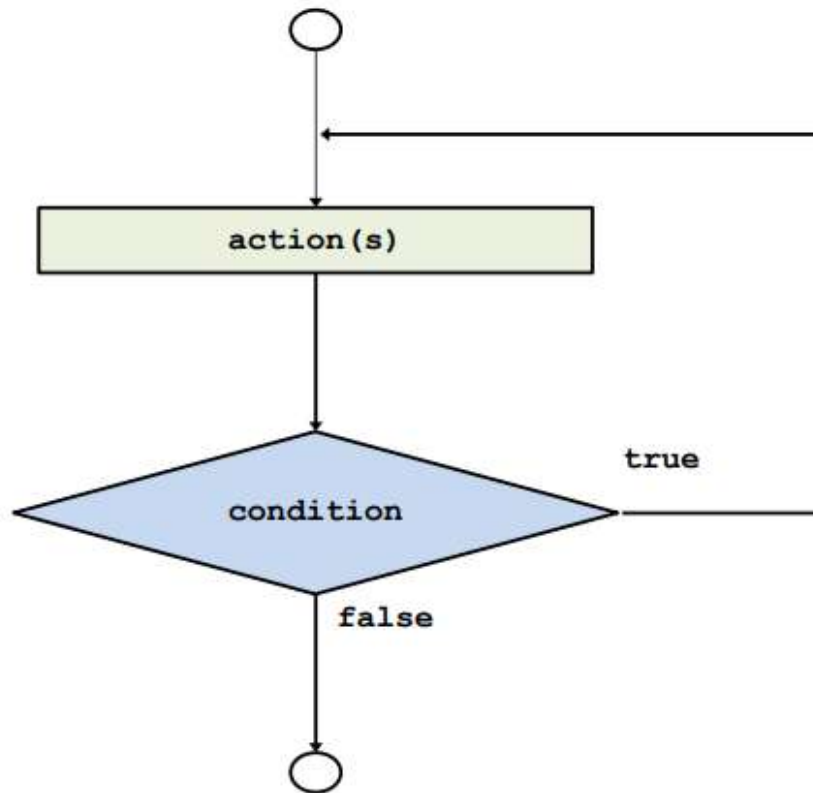
# Control Structures

- The `do..while` repetition structure
  - Similar to the `while` structure
  - Condition for repetition is tested after the body of the loop is performed
    - All actions are performed at least once.
  - Format:

```
do {  
    statement;  
} while (condition);
```

# Control Structures

- Do/While Loop



# Control Structures

```
/*Using the do/while repetition structure */  
#include <stdio.h>
```

```
int main()  
{  
    int counter = 1;  
  
    do {  
        printf("%d ", counter);  
        counter = counter + 1;  
    } while (counter <= 10);  
  
    return 0;  
}
```

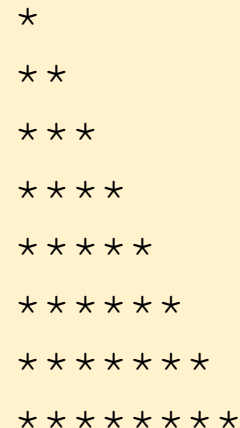
1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

# Control Structures

- When a loop body includes another loop construct this is called a *nested loop*.
- In a nested loop structure the inner loop is executed from the beginning every time the body of the outer loop is executed.

```
for (i = 1; i <= num_lines; ++i)
{
    for (j = 1; j <= i; ++j)
        printf("*");

    printf("\n");
}
```



```
*
**
***
****
*****
*****
*****
*****
*****
```

# Control Structures

- **break**
  - Causes immediate exit from a **while**, **for**, **do..while** or **switch** statement
  - Program execution continues with the first statement after the structure.
  - Common uses of break statement
    - Escape early from a loop
    - Skip the remainder of a switch statement

# Control Structures

- **break** example

```
#include <stdio.h>
int main()
{
    int x;

    for(x = 1; x <= 10 ; x++)
    {
        if(x == 5)
            break;
        printf("%d ", x);
    }

    printf("\nBroke out of the loop at x = %d ", x);
    return 0;
}
```

```
1 2 3 4
Broke out of loop at x = 5
```



# Control Structures

- `continue`
  - Skips the remaining statements in the body of a `while`, `for` or `do..while` statement
    - Proceeds with the next iteration of the loop
- `while` and `do..while`
  - Loop-continuation test is evaluated immediately after the `continue` statement is executed
- `for`
  - Increment expression is executed, then the loop-continuation test is evaluated

# Control Structures

- `continue` example

```
#include <stdio.h>
```

```
int main()
{
    int x;
    for(x = 1; x <= 10 ; x++)
    {
        if( x == 5) {
            continue;

            printf("%d ", x);
        }
        printf("\nUsed continue to skip printing the value 5");
    }
    return 0;
}
```

```
1 2 3 4 6 7 8 9 10
```

```
Used continue to skip printing the value 5
```

# Functions

- Functions are used for packaging a set of actions
- It may be easier to design a complex program starting with small working modules (i.e. functions)
  - And then we can combine these modules to form the software
- Functions are re-useable
  - Shorter
  - Easier to understand
  - Less error-prone
  - Easier to manage (e.g. easier to fix the bugs or add new features)

# Functions

- A function definition has the following form:

```
return_type function_name (parameter-declarations)
{
    variable-declarations
    function-statements
}
```

- **return\_type** : specifies the type of the function and corresponds to the type of value returned by the function
  - **void**: indicates that the function returns nothing
- **function\_name** : name of the function being defined
- **parameter-declarations** : specify the types and names of the parameters of the function, separated by commas

# Functions

- When a return statement is executed, the execution of the function is terminated and the program control is immediately passed back to the calling environment.
- If an expression follows the keyword return, the value of the expression is returned to the calling environment as well
- A return statement can be one of the following two forms:
  - `return;`
  - `return expression;`

# Functions

- Let's define a function to compute the cube of a number:

```
int cube (int num) {  
    int result;  
    result = num * num * num;  
    return result;  
}
```

- This function can be called as:

```
int n = cube(5) ;
```

# Functions

- Example: void function

```
void prn_message(void) /* function definition */
{
    printf("A message for you: ");
    printf("Have a nice day!\n");
}

int main (void)
{
    prn_message (); /* function invocation */
    return 0;
}
```

A message for you: Have a nice day!

# Functions

- A function can have zero or more parameters.
- In declaration:

```
int f (int x, double y, char c);
```



the *formal parameter list*  
(parameter variables and their  
types are declared here)

- In function calling:

```
value = f (age, 100 * score, initial);
```



actual parameter list (cannot tell  
what their type are from here)



# Functions

- The number of parameters in the actual and formal parameter lists must be consistent
- Parameter association is positional: the first actual parameter matches the first formal parameter, the second matches the second, and so on
- Actual parameters and formal parameters must be of compatible data types
- Actual parameters may be a variable, constant, any expression matching the type of the corresponding formal parameter

# Block Structure

- A block is a sequence of variable declarations and statements enclosed within braces.
- Block structure and the scope of a variable

```
int factorial(int n)
{
    if (n < 0) return -1;
    else if (n == 0) return 1;
    else
    {
        int i, result = 1;
        for (i = 1; i <= n; i++)
            result *= i;
        return result;
    }
}
```

# External Variables

- Local variables can only be accessed in the function in which they are defined.
- If a variable is defined outside any function at the same level as function definitions, it is available to all the functions defined below in the same source file  
→ external variable
- Global variables: external variables defined before any function definition
  - Their scope will be the whole program
  - Avoid using external variables as much as you can

# Example

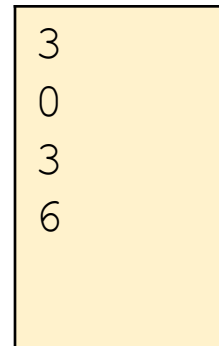
- Let's write a function that computes sum from 1 to n

```
#include <stdio.h>
```

```
int compute_sum (int n)
{
    int sum;
    sum = 0;
    for ( ; n > 0; --n)
        sum += n;

    printf("%d\n", n);
    return sum;
}
```

```
int main (void)
{
    int n, sum;
    n = 3;
    printf("%d\n", n);
    sum=compute_sum(n);
    printf("%d\n", n);
    printf("%d\n", sum);
    return 0;
}
```



```
3
0
3
6
```

# Example

- Let's write a function that swaps two variables

```
#include <stdio.h>
```

```
void swap (int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;

    printf("In swap\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
}
```

```
int main (void)
{
    int a = 3, b = 5;
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    swap(a, b);
    printf("After swap\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    return 0;
}
```

```
a: 3
b: 5
In swap
a: 5
b: 3
After swap
a: 3
b: 5
```



# Pointers

- What actually happens when we declare variables?

```
char a;  
int b;
```

- C reserves a byte in memory to store a, four bytes to store b.
- Where is that memory? At an **address**.
- Under the hood, C has been keeping track of variables and their addresses.

# Pointers

- We can work with memory addresses too. We can use variables called pointers.
- A pointer is a variable that contains the address of a variable.
- Pointers provide a powerful and flexible method for manipulating data in your programs; but they are difficult to master.
  - Close relationship with arrays and strings.

# Pointers

- Pointers allow you to reference a large data structure in a compact way.
- Pointers facilitate sharing data between different parts of a program.
  - Call-by-Reference
- **Dynamic memory allocation**: Pointers make it possible to reserve new memory during program execution.

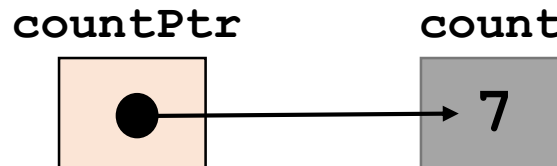


# Pointers

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



- Pointers contain address of a variable that has a specific value (indirect reference)
- Indirection – referencing a pointer value

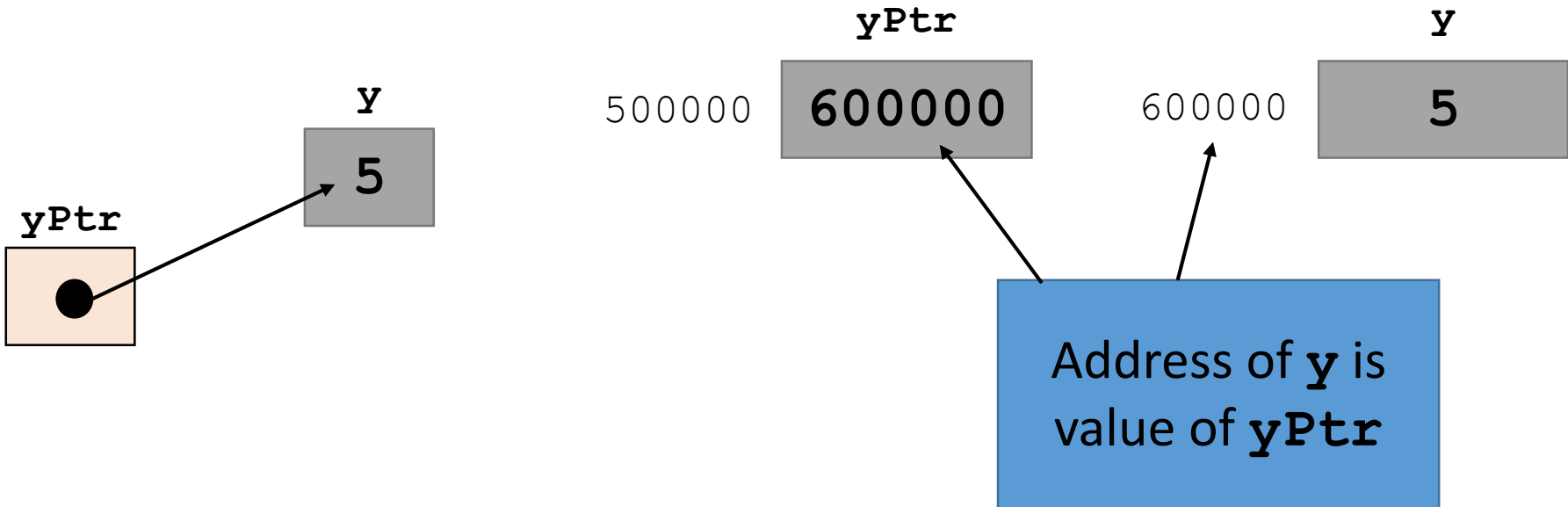


# Pointers

- **&** (address operator)
  - Returns the address of operand

```
int y = 5;  
int *yPtr;  
yPtr = &y;    /* yPtr gets address of y */
```

- *yPtr points to y*



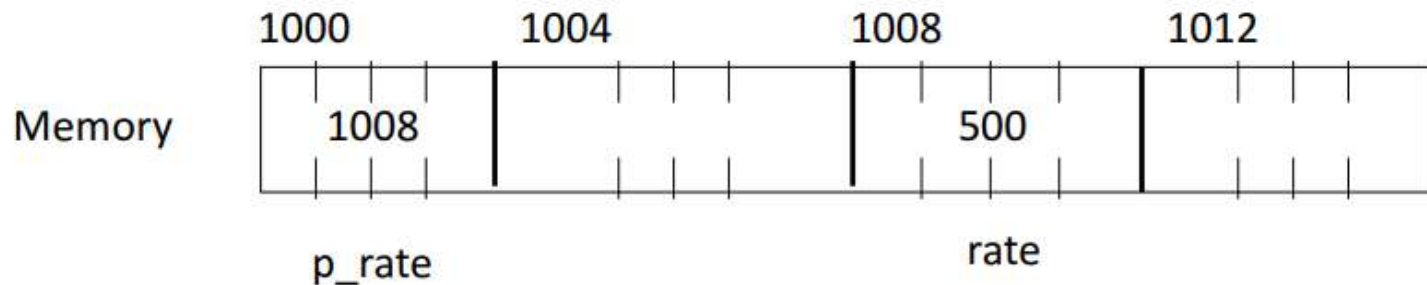
# Pointers

- `*` (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - `*yPtr` returns `y` (because `yPtr` points to `y`)
  - `*` can be used for assignment
    - `*yPtr = 7; /* changes value of y to 7 */`
- `*` and `&` are inverses
  - They cancel each other out

# Pointers

```
int rate;  
int *p_rate;
```

```
rate = 500;  
p_rate = &rate;
```



```
/* Print the values */  
printf("rate = %d\n", rate); /* direct access */  
printf("rate = %d\n", *p_rate); /* indirect access */
```

# Pointers

The address of **a** is  
the value of **aPtr**

The **\*** operator returns an  
alias to what its operand  
points to. **aPtr** points to **a**,  
so **\*aPtr** returns **a**.

```
#include <stdio.h>
int main()
{
    int a; /* a is an integer */
    int *aPtr; /* aPtr is a pointer to an integer */

    a = 7;
    aPtr = &a; /* aPtr set to address of a */

    printf("The address of a is %p\nThe value of aPtr is %p", &a, aPtr);

    printf("\n\nThe value of a is %d\nThe value of *aPtr is %d", a, *aPtr);

    printf("\n\nShowing that * and & are inverses of each other.\n&*aPtr = %p\n*&aPtr = %p\n", &*aPtr, *&aPtr);

    return 0;
}
```

Notice how **\*** and **&**  
are inverses

The address of a is 0012FF88  
The value of aPtr is 0012FF88

The value of a is 7  
The value of \*aPtr is 7  
Showing that \* and & are inverses of each other.  
&\*aPtr = 0012FF88  
\*&aPtr = 0012FF88

# Pointers

```
int a, b, *p;
```

```
a = b = 7;
```

```
p = &a;
```

```
printf("*p = %d\n", *p);
```

```
*p = 3;
```

```
printf("a = %d\n", a);
```

```
p = &b;
```

```
*p = 2 * *p - a;
```

```
printf("b = %d \n", b);
```

```
*p = 7
```

```
a = 3
```

```
b = 11
```

# Pointers

```
float x, y, *p;
```

```
x = 5;  
y = 7;  
p = &x;  
y = *p;
```

Thus,

```
y = *p;  
y = *&x;  
y = x;
```



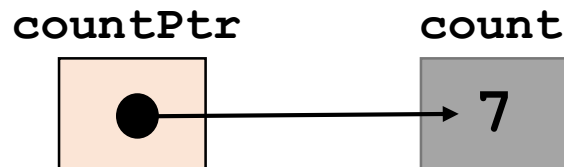
All equivalent

# Pointers

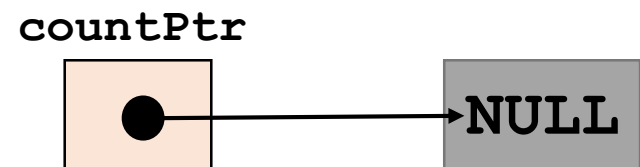
- The **NULL** Value

- The value null means that no value exists.
- The null pointer is a pointer that ‘intentionally’ points to nothing.
- If you don’t have an address to assign to a pointer, you can use NULL
- NULL value is actually 0 integer value, if the compiler does not provide any special pattern.
  - Do not use NULL value as integer!

```
countPtr = &count;
```



```
countPtr = NULL;
```





# Pointers

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change actual location in memory
- \* operator
  - Used as alias/nickname for variable inside of function

```
void double_it (int *number)
{
    *number = 2 * (*number);
}
```
- **\*number** used as nickname for the variable passed

# Pointers

```
void set_to_zero(int var)
{
    var = 0;
}
```

- You would make the following call:

```
set_to_zero(x);
```

- This function has no effect whatsoever. Instead, pass a pointer:

```
void set_to_zero(int *var)
{
    *var = 0;
}
```

- You would make the following call:

```
set_to_zero(&x);
```

- This is referred to as *call-by-reference*.

# Example

- Let's write a function that swaps two variables

```
#include <stdio.h>
```

```
void swap (int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;

    printf("In swap\n");
    printf("a: %d\n", *a);
    printf("b: %d\n", *b);
}
```

```
int main (void)
{
    int a = 3, b = 5;
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    swap(&a, &b);
    printf("After swap\n");
    printf("a: %d\n", a);
    printf("b: %d\n", b);
    return 0;
}
```

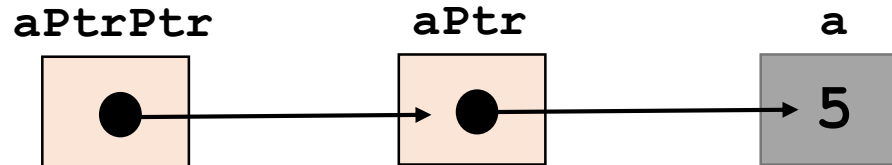
```
a: 3
b: 5
In swap
a: 5
b: 3
After swap
a: 5
b: 3
```



# Pointers

- A pointer type is also a data type and we can create pointers of all types
- Therefore we can define pointers to pointers, or pointers to pointers of pointers, ...

```
int a = 5, b = 8;  
int *aPtr = &a, *bPtr = &b;  
int **aPtrPtr = &aPtr;  
int **bPtr = &b;
```



# Example

- Let's write a function that swaps two pointers

```
#include <stdio.h>
```

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

```
int main (void){  
    int a = 3, b = 5;  
    int *aPtr, *bPtr;  
    swap_ptr(&aPtr, &bPtr);  
    printf("aPtr: %d\n", *aPtr);  
    printf("bPtr: %d\n", *bPtr);  
    return 0;  
}
```

```
aPtr: 5  
bPtr: 3
```

# Example

- Using call-by-reference to return more than one value

```
#include <stdio.h>
```

```
void divide(int a, int b, int *division, int *remainder){  
    *division = 0;  
  
    while(a > b){  
        a -= b;  
        (*division)++;  
    }  
    *remainder = a;  
    return;  
}
```

```
int main (void){  
    int a = 17, b = 5;  
    int division, remainder;  
    divide(a, b, &division, &remainder);  
    printf("division: %d\n", division);  
    printf("remainder: %d\n", remainder);  
    return 0;  
}
```

division: 3  
remainder: 2

# STRUCTURES & FILE IO

---

# Structures

- Collections of related variables (aggregates) under one name
  - Can contain variables of different data types
- Commonly used to define records to be stored in files
- Combined with pointers, can create linked lists, stacks, queues, and trees



# Structure Definitions

## Example 1:

```
struct card {  
    char *face;  
    char *suit;  
};
```

- `struct` introduces the definition for structure `card`
- `card` is the structure name and is used to declare variables of the structure type
- `card` contains two members of type `char *`
  - These members are `face` and `suit`

# Structure Definitions

- A structure definition does not reserve space in memory
  - Instead creates a new data type used to define structure variables

- Variables can be defined as below:

```
struct card {  
    char *face;  
    char *suit;  
} oneCard, deck[ 52 ], *cPtr;
```

- Or defined like other variables:

```
struct card {  
    char *face;  
    char *suit;  
};  
struct card oneCard, deck[ 52 ], *cPtr;
```

# Structure Definitions

Example 2:

```
struct point {  
    int x;  
    int y;  
};
```

```
struct point pt; /* defines a variable pt which  
                  is a structure of type  
                  struct point */
```

```
pt.x = 15;  
pt.y = 30;  
printf("%d, %d", pt.x, pt.y);
```

# Structure Definitions

```
/* Structures can be nested. One representation of  
a rectangle is a pair of points that denote the  
diagonally opposite corners. */
```

```
struct rect {  
    struct point pt1;  
    struct point pt2;  
};
```

```
struct rect screen;
```

```
/* Print the pt1 field of screen */  
printf("%d, %d", screen.pt1.x, screen.pt1.y);
```

```
/* Print the pt2 field of screen */  
printf("%d, %d", screen.pt2.x, screen.pt2.y);
```

# Structure Definitions

## Valid Operations

- Assigning a structure to a structure of the same type
- Taking the address (&) of a structure
- Accessing the members of a structure
- Using the sizeof operator to determine the size of a structure

# Initializing Structures

- **Initializer lists**

- Example:

```
card oneCard = { "Three", "Hearts" };
```

- **Assignment statements**

- Example:

```
card threeHearts = oneCard;
```

- Could also define and initialize threeHearts as follows:

```
card threeHearts;
```

```
threeHearts.face = "Three";
```

```
threeHearts.suit = "Hearts";
```

# Accessing Members of Structures

- Accessing structure members
  - Dot operator (.) used with structure variables

```
card myCard;  
printf( "%s", myCard.suit );
```
  - Arrow operator (->) used with pointers to structure variables

```
card *myCardPtr = &myCard;  
printf( "%s", myCardPtr->suit );
```
  - `myCardPtr->suit` is equivalent to  
`( *myCardPtr ).suit`

```
1 // Fig. 10.2: fig10_02.c
2 // Structure member operator and
3 // structure pointer operator
4 #include <stdio.h>
5
6 // card structure definition
7 struct card {
8     char *face; // define pointer face
9     char *suit; // define pointer suit
10 };
11
12 int main(void)
13 {
14     struct card aCard; // define one struct card variable
15
16     // place strings into aCard
17     aCard.face = "Ace";
18     aCard.suit = "Spades";
19
20     struct card *cardPtr = &aCard; // assign address of aCard to cardPtr
21 }
```

---

**Fig. 10.2** | Structure member operator and structure pointer operator. (Part 1 of 2.)



```
22 printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,  
23     cardPtr->face, " of ", cardPtr->suit,  
24     (*cardPtr).face, " of ", (*cardPtr).suit);  
25 }
```

```
Ace of Spades  
Ace of Spades  
Ace of Spades
```

**Fig. 10.2** | Structure member operator and structure pointer operator. (Part 2 of 2.)

# typedef

## typedef

- Creates synonyms (aliases) for previously defined data types
- Use **typedef** to create shorter type names

Example:

```
typedef struct point pixel;
```

- Defines a new type name **pixel** as a synonym for type **struct point**

```
typedef struct Card *CardPtr;
```

- Defines a new type name **CardPtr** as a synonym for type **struct Card \***
- **typedef** does not create a new data type
  - Only creates an alias

# Using Structures With Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

# Using Structures with Functions 1

```
#include<stdio.h> /* Demonstrates passing a structure to a  
function */
```

```
struct data{  
    int amount;  
    char fname[30];  
    char lname[30];  
}rec;
```

```
void printRecord(struct data x){  
    printf("\nDonor %s %s gave $%d", x.fname, x.lname, x.amount);  
}
```

```
int main(void){  
    printf("Enter the donor's first and last names\n");  
    printf("separated by a space:  ");  
    scanf("%s %s",rec.fname, rec.lname);  
    printf("Enter the donation amount:  ");  
    scanf("%d",&rec.amount);  
    printRecord(rec);  
    return 0;  
}
```

# Using Structures with Functions 2

```
/* Make a point from x and y components. */
struct point makepoint (int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return (temp);
}
```

```
/* makepoint can now be used to initialize a structure */
struct rect screen;
struct point middle;

screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(50,100);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                  (screen.pt1.y + screen.pt2.y)/2);
```

```
/* add two points */  
  
struct point addpoint (struct point p1, struct point p2)  
{  
    p1.x += p2.x;  
    p1.y += p2.y;  
    return p1;  
}
```

Both arguments and the return value are structures in the function addpoint.

# Structures and Pointers

```
struct point *p; /* p is a pointer to a structure  
                  of type struct point */  
struct point origin;
```

```
p = &origin;  
printf("Origin is (%d, %d)\n", (*p).x, (*p).y);
```

- Parenthesis are necessary in  $(*p).x$  because the precedence of the structure member operator (dot) is higher than  $*$ .
- The expression  $*p.x \equiv *(p.x)$  which is illegal because  $x$  is not a pointer.

# Structures and Pointers

- Pointers to structures are so frequently used that an alternative is provided as a shorthand.
- If `p` is a pointer to a structure, then

`p -> field_of_structure`

refers to a particular field.

- We could write

```
printf("Origin is (%d %d)\n", p->x, p->y);
```



# Assignments

```
struct student {  
    char *last_name;  
    int student_id;  
    char grade;  
};  
struct student temp, *p = &temp;
```

```
temp.grade = 'A';  
temp.last_name = "Casanova";  
temp.student_id = 590017;
```

<u>Expression</u>	<u>Equiv. Expression</u>	<u>Value</u>
temp.grade	p -> grade	A
temp.last_name	p -> last_name	Casanova
temp.student_id	p -> student_id	590017
(*p).student_id	p -> student_id	590017

# Structures and Pointers

- Both `.` and `->` associate from left to right
- Consider

```
struct rect r, *rp = &r;
```

- The following 4 expressions are equivalent.

```
r.pt1.x
```

```
rp -> pt1.x
```

```
(r.pt1).x
```

```
(rp->pt1).x
```

```
struct rect {  
    struct point pt1;  
    struct point pt2;  
};
```

# Arrays of Structures

- Usually a program needs to work with more than one instance of data.
- For example, to maintain a list of phone #s in a program, you can define a structure to hold each person's name and number.

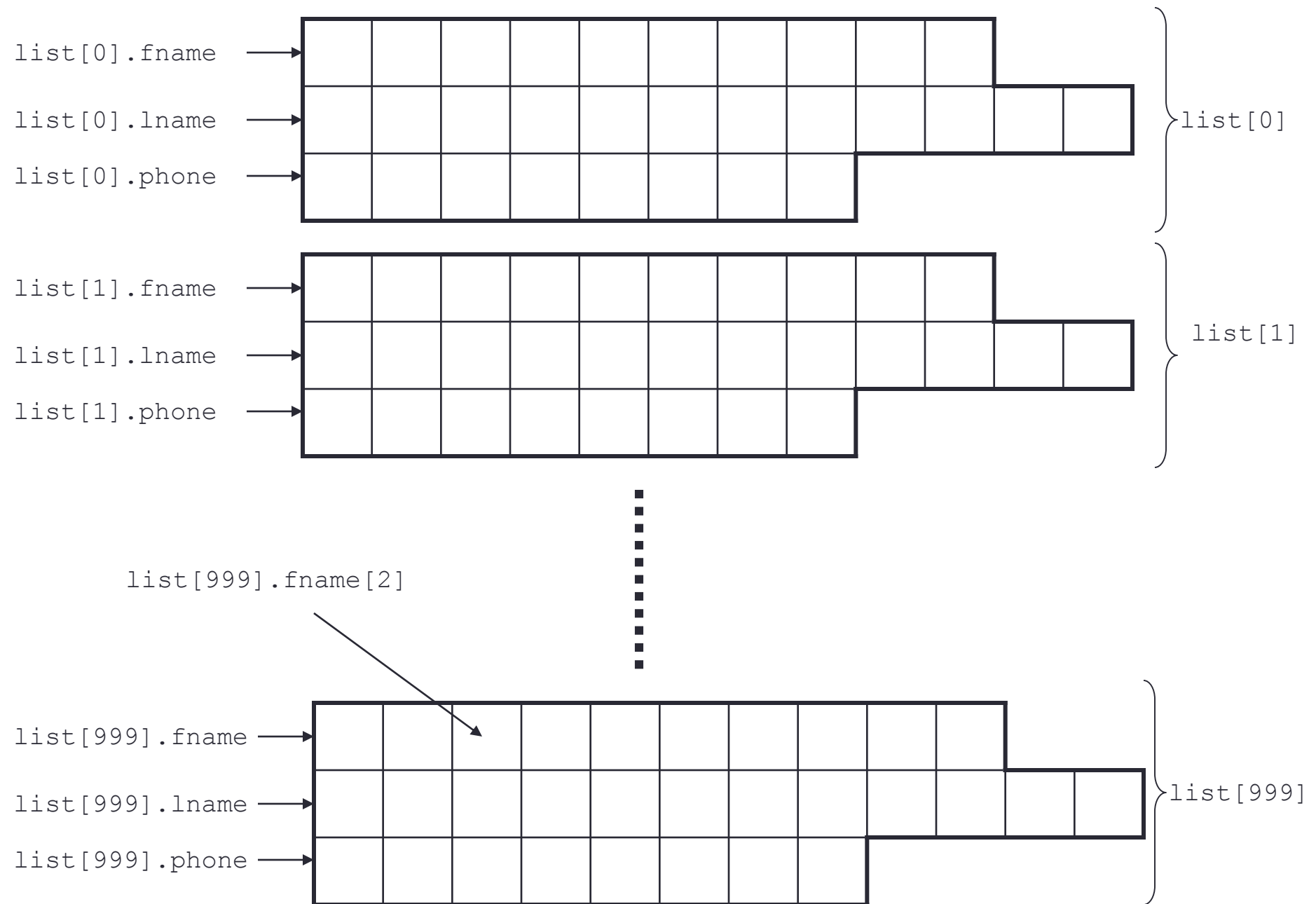
```
struct entry {  
    char fname[10];  
    char lname[12];  
    char phone[8];  
};
```

# Arrays of Structures

- A phone list has to hold many entries, so a single instance of the entry structure isn't of much use. What we need is an array of structures of type entry.
- After the structure has been defined, you can define the array as follows:

```
struct entry list[1000];
```

# struct entry list[1000]



- To assign data in one element to another array element, you write

```
list[1] = list[5];
```

- To move data between individual structure fields, you write

```
strcpy(list[1].phone, list[5].phone);
```

- To move data between individual elements of structure field arrays, you write

```
list[5].phone[1] = list[2].phone[3];
```

```
#define CLASS_SIZE 100
struct student {
    char *last_name;
    int student_id;
    char grade;
};

int main(void)
{
    struct student temp,
        class[CLASS_SIZE];

    ... /*Do some operation to fill class structure*/

    printf ("Number of A's in class: %d\n", countA(class));
}

int countA(struct student class[])
{
    int i, cnt = 0;
    for (i = 0; i < CLASS_SIZE; ++i)
        cnt += class[i].grade == 'A';
    return cnt;
}
```

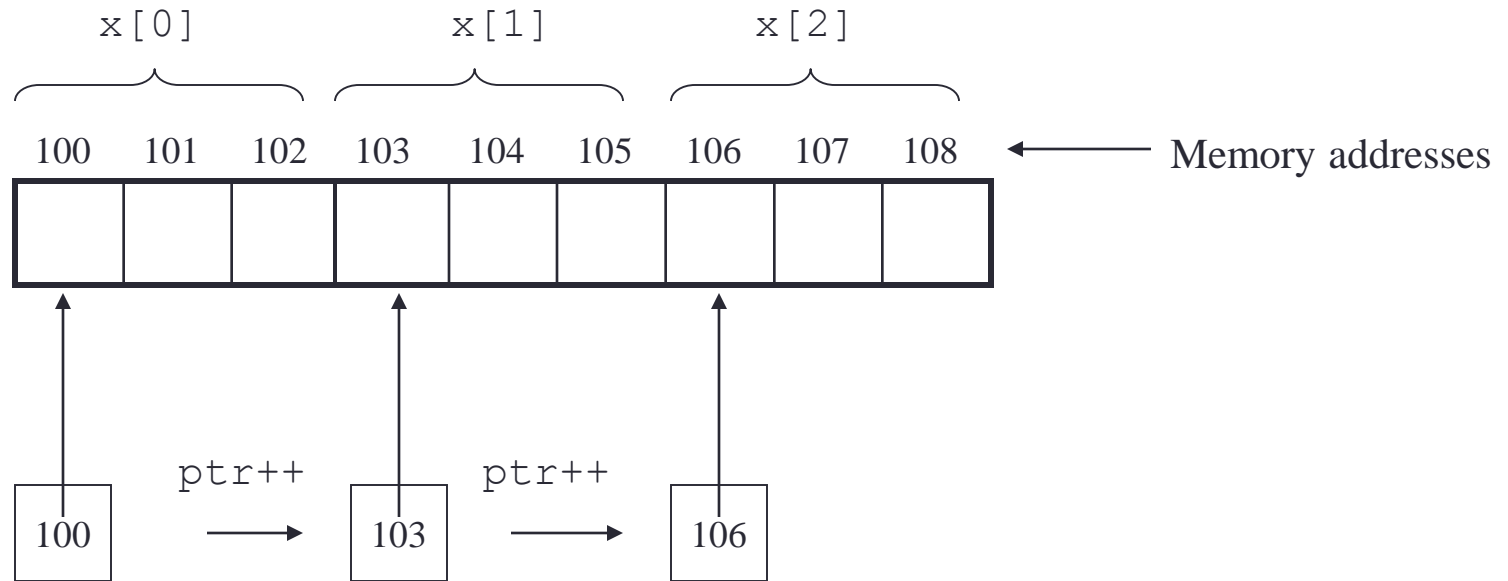
- Arrays of structures can be very powerful programming tools, as can pointers to structures.

```
struct part {  
    int number;  
    char name [10];  
};
```

```
struct part data[100];  
struct part *p_part;
```

```
p_part = data;  
printf("%d %s", p_part->number, p_part -> name);
```





- The above diagram shows an array named `x` that consists of 3 elements. The pointer `ptr` was initialized to point at `x[0]`. Each time `ptr` is incremented, it points at the next array element.

```
/* Array of structures */
#include <stdio.h>
#define MAX 4

struct part {
    int number;
    char name[10];
};

struct part data[MAX]= {1, "Smith", 2, "Jones", 3, "Adams", 4, "Will"};

int main (void)
{
    struct part *p_part;
    int count;

    p_part = data;
    for (count = 0; count < MAX; count++) {
        printf("\n %d %s", p_part -> number, p_part -> name);
        p_part++;
    }
    return 0;
}
```

## Example: High-Performance Card Shuffling and Dealing Simulation

- The program in Fig. 10.3 is based on the card shuffling and dealing simulation discussed in Chapter 7.
- The program represents the deck of cards as an array of structures and uses high-performance shuffling and dealing algorithms.

```
1 // Fig. 10.3: fig10_03.c
2 // Card shuffling and dealing program using structures
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <time.h>
6
7 #define CARDS 52
8 #define FACES 13
9
10 // card structure definition
11 struct card {
12     const char *face; // define pointer face
13     const char *suit; // define pointer suit
14 };
15
16 typedef struct card Card; // new type name for struct card
17
18 // prototypes
19 void fillDeck(Card * const wDeck, const char * wFace[],
20             const char * wSuit[]);
21 void shuffle(Card * const wDeck);
22 void deal(const Card * const wDeck);
23
```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part I of 4.)

```
24 int main(void)
25 {
26     Card deck[CARDS]; // define array of Cards
27
28     // initialize array of pointers
29     const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
30         "Six", "Seven", "Eight", "Nine", "Ten",
31         "Jack", "Queen", "King"};
32
33     // initialize array of pointers
34     const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
35
36     srand(time(NULL)); // randomize
37
38     fillDeck(deck, face, suit); // load the deck with Cards
39     shuffle(deck); // put Cards in random order
40     deal(deck); // deal all 52 Cards
41 }
42
```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 2 of 4.)

```
43 // place strings into Card structures
44 void fillDeck(Card * const wDeck, const char * wFace[],
45             const char * wSuit[])
46 {
47     // loop through wDeck
48     for (size_t i = 0; i < CARDS; ++i) {
49         wDeck[i].face = wFace[i % FACES];
50         wDeck[i].suit = wSuit[i / FACES];
51     }
52 }
53
54 // shuffle cards
55 void shuffle(Card * const wDeck)
56 {
57     // loop through wDeck randomly swapping Cards
58     for (size_t i = 0; i < CARDS; ++i) {
59         size_t j = rand() % CARDS;
60         Card temp = wDeck[i];
61         wDeck[i] = wDeck[j];
62         wDeck[j] = temp;
63     }
64 }
65
```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 3 of 4.)

---

```
66 // deal cards
67 void deal(const Card * const wDeck)
68 {
69     // loop through wDeck
70     for (size_t i = 0; i < CARDS; ++i) {
71         printf("%5s of %-8s%s", wDeck[i].face, wDeck[i].suit,
72             (i + 1) % 4 ? " " : "\n");
73     }
74 }
```

---

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 4 of 4.)



Three of Hearts	Jack of Clubs	Three of Spades	Six of Diamonds
Five of Hearts	Eight of Spades	Three of Clubs	Deuce of Spades
Jack of Spades	Four of Hearts	Deuce of Hearts	Six of Clubs
Queen of Clubs	Three of Diamonds	Eight of Diamonds	King of Clubs
King of Hearts	Eight of Hearts	Queen of Hearts	Seven of Clubs
Seven of Diamonds	Nine of Spades	Five of Clubs	Eight of Clubs
Six of Hearts	Deuce of Diamonds	Five of Spades	Four of Clubs
Deuce of Clubs	Nine of Hearts	Seven of Hearts	Four of Spades
Ten of Spades	King of Diamonds	Ten of Hearts	Jack of Diamonds
Four of Diamonds	Six of Spades	Five of Diamonds	Ace of Diamonds
Ace of Clubs	Jack of Hearts	Ten of Clubs	Queen of Diamonds
Ace of Hearts	Ten of Diamonds	Nine of Clubs	King of Spades
Ace of Spades	Nine of Diamonds	Seven of Spades	Queen of Spades

**Fig. 10.4** | Output for the high-performance card shuffling and dealing simulation.



# Unions

- **union**

- Memory that contains a variety of objects over time
- Only contains one data member at a time
- Members of a union share space
- Conserves storage
- Only the last data member defined can be accessed

- **union definitions**

- Same as struct

```
union Number {  
    int x;  
    float y;  
};  
union Number value;
```

# Unions

- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->

```
/* number union definition */
union number {
    int x;    /* define int x */
    double y; /* define double y */
}; /* end union number */

int main(){
    union number value; /* define union value */

    value.x = 100; /* put an integer into the union */

    printf("Put a value in the integer member.\n");
    printf(" int: %d\n double:%f\n\n", value.x, value.y );

    value.y = 100.0; /* put a double into the same union */

    printf("Put a value in the floating member.\n");
    printf(" int: %d\n double:%f\n\n", value.x, value.y );

    return 0; /* indicates successful termination */
} /* end main */
```

## Put a value in the integer member.

```
int: 100
```

## double:-

[illegible]

## Put a value in the floating member.

```
int: 0
```

double: 100.000000

# FILE INPUT/OUTPUT

- In this chapter, you will learn:
  - To be able to create, read, write and update files.
  - To become familiar with sequential access file processing.
  - To become familiar with random-access file processing.

# Introduction

- Data files
  - Can be created, updated, and processed by C programs
  - Are used for permanent storage of large amounts of data
    - Storage of data in variables and arrays is only temporary
- When you use a file to store data for use by a program, that file usually consists of text (alphanumeric data) and is therefore called a **text file**.

# The Data Hierarchy

- Data Hierarchy:
  - Bit – smallest data item
    - Value of 0 or 1
  - Byte – 8 bits
    - Used to store a character
      - Decimal digits, letters, and special symbols
  - Field – group of characters conveying meaning
    - Example: your name
  - Record – group of related fields
    - Represented by a `struct` or a `class`
    - Example: In a payroll system, a record for a particular employee that contained his/her identification number, name, address, etc.

# The Data Hierarchy

- Data Hierarchy (continued):
  - File – group of related records
    - Example: payroll file
  - Database – group of related files

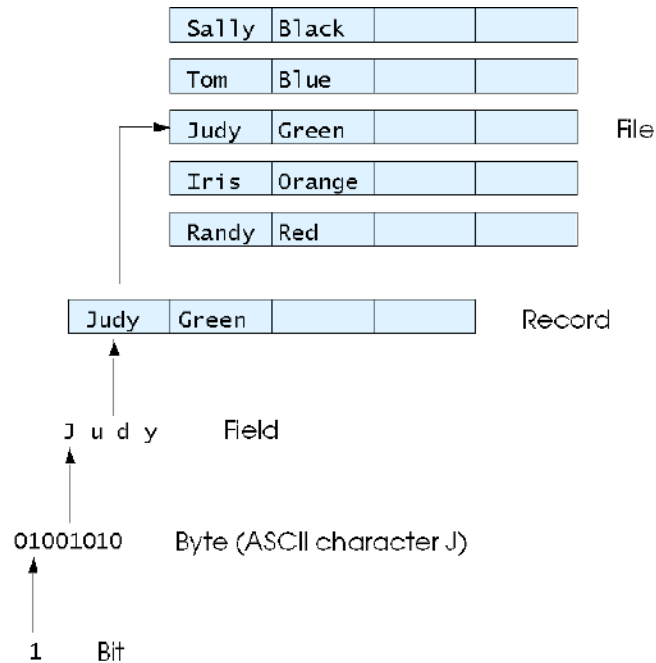


Fig. 11.1 The data hierarchy.



# Files and Streams

- C views each file as a sequence of bytes
  - File ends with the *end-of-file marker*
    - Or, file ends at a specified byte
- Stream created when a file is opened
  - Provide communication channel between files and programs
  - Opening a file returns a pointer to a FILE structure
    - Example file pointers:
      - `stdin` - standard input (keyboard)
      - `stdout` - standard output (screen)
      - `stderr` - standard error (screen)

# Files and Streams

- FILE structure
  - File descriptor
    - Index into operating system array called the open file table
  - File Control Block (FCB)
    - Found in every array element, system uses it to administer the file



Fig. 11.2 C's view of a file of  $n$  bytes.

# Files and Streams

- Read/Write functions in standard library
  - `fscanf / fprintf`
    - File processing equivalents of `scanf` and `printf`
  - `fgetc`
    - Reads one character from a file
    - Takes a `FILE` pointer as an argument
    - `fgetc( stdin )` equivalent to `getchar()`
  - `fputc`
    - Writes one character to a file
    - Takes a `FILE` pointer and a character to write as an argument
    - `fputc( 'a', stdout )` equivalent to `putchar( 'a' )`
  - `fgets`
    - Reads a line from a file
  - `fputs`
    - Writes a line to a file

```
1  /*
2      Create a sequential file */
3  #include <stdio.h>
4
5  int main()
6  {
7      int account;
8      char name[ 30 ];
9      double balance;
10     FILE *cfPtr;    /* cfPtr = clients.dat file pointer */
11
12     if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL )
13         printf( "File could not be opened\n" );
14     else {
15         printf( "Enter the account, name, and balance.\n" );
16         printf( "Enter EOF to end input.\n" );
17         printf( "? " );
18         scanf( "%d%s%lf", &account, name, &balance );
19
20         while ( !feof( stdin ) ) {
21             fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
22
23             printf( "? " );
24             scanf( "%d%s%lf", &account, name, &balance );
25         }
26
27         fclose( cfPtr );
28     }
29
30     return 0;
31 }
```

## Program Output

```
Enter the account, name, and balance.  
Enter EOF to end input.  
? 100 Jones 24.98  
? 200 Doe 345.67  
? 300 White 0.00  
? 400 Stone -42.16  
? 500 Rich 224.62  
? ^Z
```

# Creating a Sequential Access File

- Creating a File
  - `FILE *myPtr;`
    - Creates a FILE pointer called myPtr
  - `myPtr = fopen(filename, openmode);`
    - Function fopen returns a FILE pointer to file specified
    - Takes two arguments – file to open and file open mode
    - If open fails, NULL returned

Computer system	Key combination
UNIX systems	<i>&lt;return&gt; &lt;ctrl&gt; d</i>
IBM PC and compatibles	<i>&lt;ctrl&gt; z</i>
Macintosh	<i>&lt;ctrl&gt; d</i>

Fig. 11.4      End-of-file key combinations for various popular computer systems.

# Creating a Sequential Access File

Mode	Description
r	Open a file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at end of file.
r+	Open a file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append; open or create a file for update; writing is done at the end of the file.
rb	Open a file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append; open or create a file for writing at end of file in binary mode.
rb+	Open a file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append; open or create a file for update in binary mode; writing is done at the end of the file.

Fig. 11.6 File open modes.

# Creating a Sequential Access File

- `fprintf`
  - Used to print to a file
  - Like `printf`, except first argument is a `FILE` pointer (pointer to the file you want to print in)
- `feof( FILE pointer )`
  - Returns true if end-of-file indicator (no more data to process) is set for the specified file
- `fclose( FILE pointer )`
  - Closes specified file
  - Performed automatically when program ends
  - Good practice to close files explicitly
- **Details**
  - Programs may process no files, one file, or many files
  - Each file must have a unique name and should have its own pointer



# Reading Data from a File

- Reading a sequential access file
  - Create a FILE pointer, link it to the file to read  
`myPtr = fopen( "myfile.dat", "r" );`
  - Use `fscanf` to read from the file
    - Like `scanf`, except first argument is a FILE pointer  
`fscanf( myPtr, "%d%s%f", &account, name, &balance );`
  - Data read from beginning to end
  - File position pointer
    - Indicates number of next byte to be read / written
    - Not really a pointer, but an integer value (specifies byte location)
    - Also called byte offset
  - `rewind( myPtr )`
    - Repositions file position pointer to beginning of file (byte 0)

```

1
2  /* Reading and printing a sequential file */
3  #include <stdio.h>
4
5  int main()
6  {
7      int account;
8      char name[ 30 ];
9      double balance;
10     FILE *cfPtr;    /* cfPtr = clients.dat file pointer */
11
12     if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL )
13         printf( "File could not be opened\n" );
14     else {
15         printf( "%-10s%-13s%\n", "Account", "Name", "Balance" );
16         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
17
18         while ( !feof( cfPtr ) ) {
19             printf( "%-10d%-13s%7.2f\n", account, name, balance );
20             fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
21         }
22
23         fclose( cfPtr );
24     }
25
26     return 0;
27 }

```

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

## Example: Merge two files

```
#include <stdio.h>

int main()
{
    FILE *fileA, /* first input file */
          *fileB, /* second input file */
          *fileC; /* output file to be created */
    int num1, /* number to be read from first file */
        num2; /* number to be read from second file */
    int f1, f2;

    /* Open files for processing */
    fileA = fopen("class1.txt", "r");
    fileB = fopen("class2.txt", "r");
    fileC = fopen("class.txt", "w");
```

```
/* As long as there are numbers in both files, read and compare numbers one
by one. Write the smaller number to the output file and read the next number
in the file from which the smaller number is read. */
```

```
f1 = fscanf(fileA, "%d", &num1);
f2 = fscanf(fileB, "%d", &num2);

while ((f1!=EOF) && (f2!=EOF)) {
    if (num1 < num2) {
        fprintf(fileC, "%d\n", num1);
        f1 = fscanf(fileA, "%d", &num1);
    }
    else if (num2 < num1) {
        fprintf(fileC, "%d\n", num2);
        f2 = fscanf(fileB, "%d", &num2);
    }
    else { /* numbs are equal:read from both files */
        fprintf(fileC, "%d\n", num1);
        f1 = fscanf(fileA, "%d", &num1);
        f2 = fscanf(fileB, "%d", &num2);
    }
}
```

```
while (f1!=EOF){/* if reached end of second file, read
    the remaining numbers from first file and write to
    output file */
    fprintf(fileC,"%d\n", num1);
    f1 = fscanf(fileA, "%d", &num1);
}
while (f2!=EOF){ if reached the end of first file, read
    the remaining numbers from second file and write
    to output file */
    fprintf(fileC,"%d\n", num2);
    f2 = fscanf(fileB, "%d", &num2);
}

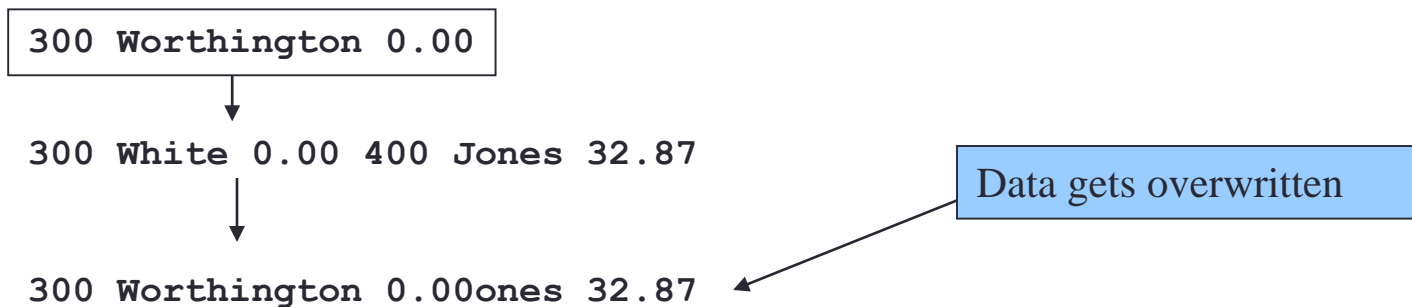
/* close files */
fclose(fileA);
fclose(fileB);
fclose(fileC);
return 0;
} /* end of main */
```

# Reading Data from a Sequential Access File

- Sequential access file
  - Cannot be modified without the risk of destroying other data
  - Fields can vary in size
    - Different representation in files and screen than internal representation
    - 1, 34, -890 are all `ints`, but have different sizes on disk

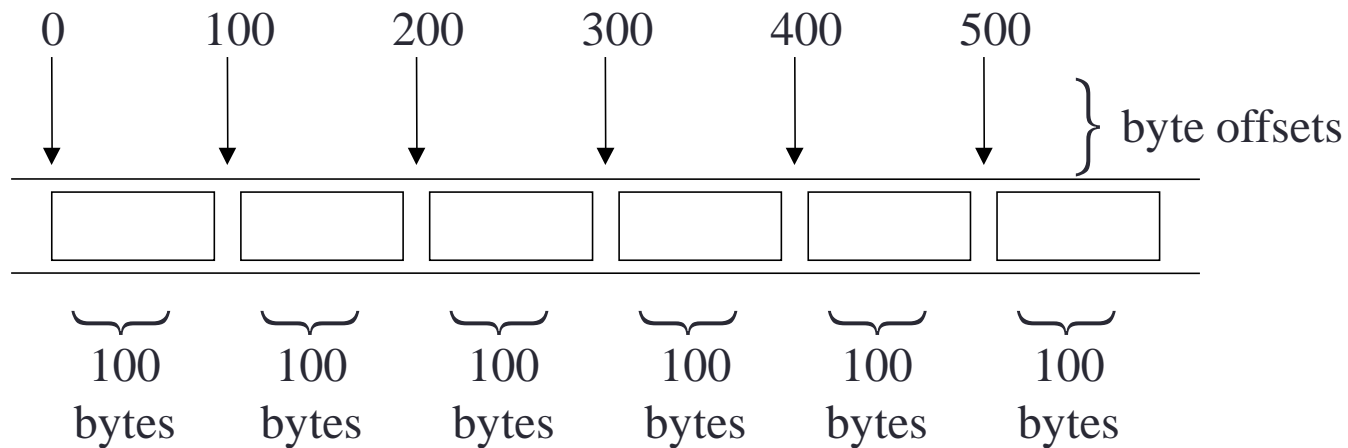
300 White 0.00 400 Jones 32.87 (old data in file)

If we want to change White's name to Worthington,



# Random-Access Files

- Random access files
  - Access individual records without searching through other records
  - Instant access to records in a file
  - Data can be inserted without destroying other data
  - Data previously stored can be updated or deleted without overwriting
- Implemented using fixed length records
  - Sequential files do not have fixed length records



# Creating a Randomly Accessed File

- Data in random access files
  - Unformatted (stored as "raw bytes")
    - All data of the same type (`ints`, for example) uses the same amount of memory
    - All records of the same type have a fixed length
    - Data not human readable



# Creating a Randomly Accessed File

- Unformatted I/O functions

- `fwrite`

- Transfer bytes from a location in memory to a file

- `fread`

- Transfer bytes from a file to a location in memory

- Example:

- ```
fwrite( &number, sizeof( int ), 1, myPtr );
```

- `&number` – Location to transfer bytes from
      - `sizeof( int )` – Number of bytes to transfer
      - `1` – For arrays, number of elements to transfer
        - In this case, "one element" of an array is being transferred
      - `myPtr` – File to transfer to or from

# Creating a Randomly Accessed File

- Writing structs

```
fwrite( &myObject, sizeof (struct myStruct), 1, myPtr );
```

- `sizeof` – returns size in bytes of object in parentheses

- To write several array elements

- Pointer to array as first argument
- Number of elements to write as third argument

```
1  /* Fig. 11.11: fig11_11.c
2      Creating a randomly accessed file sequentially */
3  #include <stdio.h>
4
5  /* clientData structure definition */
6  struct clientData {
7      int acctNum;          /* account number */
8      char lastName[ 15 ]; /* account last name */
9      char firstName[ 10 ]; /* account first name */
10     double balance;       /* account balance */
11 }; /* end structure clientData */
12
13 int main()
14 {
15     int i; /* counter */
16
17     /* create clientData with no information */
18     struct clientData blankClient = { 0, "sevil", "sen", 5000.0 };
19
20     FILE *cfPtr; /* credit.dat file pointer */
```

```
22  /* fopen opens the file; exits if file cannot be opened */
23  if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
24      printf( "File could not be opened.\n" );
25  } /* end if */
26  else {
27
28      /* output 100 blank records to file */
29      for ( i = 1; i <= 100; i++ ) {
30          fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
31      } /* end for */
32
33      fclose ( cfPtr ); /* fclose closes the file */
34  } /* end else */
35
36  return 0; /* indicates successful termination */
37
38 } /* end main */
```

---

# Writing Data Randomly to a Randomly Accessed File

- `fseek`
  - Sets file position pointer to a specific position
  - `fseek( pointer, offset, symbolic_constant );`
    - *pointer* – pointer to file
    - *offset* – file position pointer (0 is first location)
    - *symbolic\_constant* – specifies where in file we are reading from
    - `SEEK_SET` – seek starts at beginning of file
    - `SEEK_CUR` – seek starts at current location in file
    - `SEEK_END` – seek starts at end of file

```
1  /* Fig. 11.12: fig11_12.c
2      Writing to a random access file */
3  #include <stdio.h>
4
5  /* clientData structure definition */
6  struct clientData {
7      int acctNum;          /* account number */
8      char lastName[ 15 ]; /* account last name */
9      char firstName[ 10 ]; /* account first name */
10     double balance;       /* account balance */
11 }; /* end structure clientData */
12
13 int main()
14 {
15     FILE *cfPtr; /* credit.dat file pointer */
16
17     /* create clientData with no information */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen opens the file; exits if file cannot be opened */
21     if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
22         printf( "File could not be opened.\n" );
23     } /* end if */
24     else {
25
```

```
26      /* require user to specify account number */
27      printf( "Enter account number"
28              " ( 1 to 100, 0 to end input )\n? " );
29      scanf( "%d", &client.acctNum );
30
31      /* user enters information, which is copied into file */
32      while ( client.acctNum != 0 ) {
33
34          /* user enters last name, first name and balance */
35          printf( "Enter lastname, firstname, balance\n? " );
36
37          /* set record lastName, firstName and balance value */
38          fscanf( stdin, "%s%s%lf", client.lastName,
39                  client.firstName, &client.balance );
40
41          /* seek position in file of user-specified record */
42          fseek( cfPtr, ( client.acctNum - 1 ) *
43                  sizeof( struct clientData ), SEEK_SET );
44
45          /* write user-specified information in file */
46          fwrite( &client, sizeof( struct clientData ), 1, cfPtr );
47
48          /* enable user to specify another account number */
49          printf( "Enter account number\n? " );
50          scanf( "%d", &client.acctNum );
```

```
51     } /* end while */
52
53     fclose( cfPtr ); /* fclose closes the file */
54 } /* end else */
55
56 return 0; /* indicates successful termination */
57
58 } /* end main */
```

---

```
Enter account number ( 1 to 100, 0 to end input )
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```



# Writing Data Randomly to a Randomly Accessed File

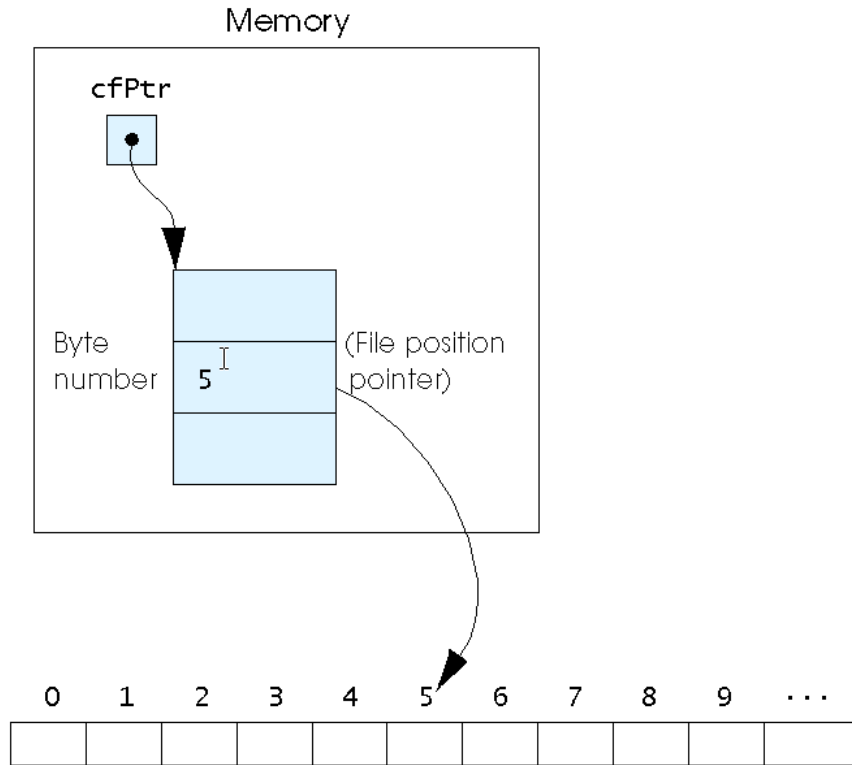


Fig. 11.14 The file position pointer indicating an offset of 5 bytes from the beginning of the file.

# Reading Data Randomly from a Randomly Accessed File

- **fread**

- Reads a specified number of bytes from a file into memory  
`fread( &client, sizeof (struct clientData), 1, myPtr );`
- Can read several fixed-size array elements
  - Provide pointer to array
  - Indicate number of elements to read
- To read multiple elements, specify in third argument

---

```
1  /* Fig. 11.15: fig11_15.c
2     Reading a random access file sequentially */
3  #include <stdio.h>
4
5  /* clientData structure definition */
6  struct clientData {
7     int acctNum;          /* account number */
8     char lastName[ 15 ]; /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance;        /* account balance */
11 }; /* end structure clientData */
12
13 int main()
14 {
15     FILE *cfPtr; /* credit.dat file pointer */
16
17     /* create clientData with no information */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen opens the file; exits if file cannot be opened */
21     if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
22         printf( "File could not be opened.\n" );
23     } /* end if */
```

```

24  else {
25      printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
26          "First Name", "Balance" );
27
28      /* read all records from file (until eof) */
29      while ( !feof( cfPtr ) ) {
30          fread( &client, sizeof( struct clientData ), 1, cfPtr );
31
32          /* display record */
33          if ( client.acctNum != 0 ) {
34              printf( "%-6d%-16s%-11s%10.2f\n",
35                  client.acctNum, client.lastName,
36                  client.firstName, client.balance );
37          } /* end if */
38
39      } /* end while */
40
41      fclose( cfPtr ); /* fclose closes the file*/
42  } /* end else */
43
44  return 0;
45
46 } /* end main */

```

| Acct | Last Name | First Name | Balance |
|------|-----------|------------|---------|
| 29   | Brown     | Nancy      | -24.54  |
| 33   | Dunn      | Stacey     | 314.33  |
| 37   | Barker    | Doug       | 0.00    |
| 88   | Smith     | Dave       | 258.34  |
| 96   | Stone     | Sam        | 34.98   |

# ARRAYS & HEAP

---

# Content

In this chapter, you will learn:

- To introduce the array data structure
- To understand the use of arrays
- To understand how to define an array, initialize an array and refer to individual elements of an array
- To be able to pass arrays to functions
- To be able to define and manipulate multi-dimensional arrays
- To be able to use arrays as pointers
- To be able to create variable-length arrays whose size is determined at execution time

# Introduction

## Arrays

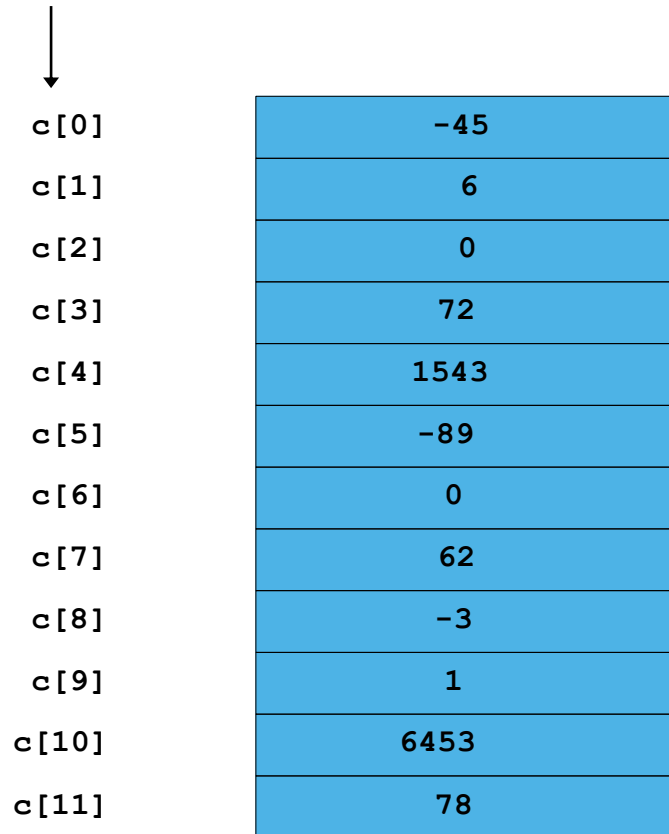
- Structures of related data items
- Static entity – same size throughout program
- Dynamic data structures will be discussed later

# Arrays

- Array
  - Group of consecutive memory locations
  - Same name and type
- To refer to an element, specify
  - Array name
  - Position number
- Format:
  - arrayname[ position number ]*
  - First element at position 0
  - n element array named c :
    - `c[ 0 ]`, `c[ 1 ]`...`c[ n - 1 ]`



Name of array (Note that all elements of this array have the same name, **c**)



|              |             |
|--------------|-------------|
| <b>c[0]</b>  | <b>-45</b>  |
| <b>c[1]</b>  | <b>6</b>    |
| <b>c[2]</b>  | <b>0</b>    |
| <b>c[3]</b>  | <b>72</b>   |
| <b>c[4]</b>  | <b>1543</b> |
| <b>c[5]</b>  | <b>-89</b>  |
| <b>c[6]</b>  | <b>0</b>    |
| <b>c[7]</b>  | <b>62</b>   |
| <b>c[8]</b>  | <b>-3</b>   |
| <b>c[9]</b>  | <b>1</b>    |
| <b>c[10]</b> | <b>6453</b> |
| <b>c[11]</b> | <b>78</b>   |

Position number of the element within array **c**

# Arrays

- Array elements are like normal variables

```
c[ 0 ] = 3;  
printf( "%d", c[ 0 ] );
```

- Perform operations in subscript. If x equals 3

```
c[ 5 - 2 ] == c[ 3 ] == c[ x ]
```

```
c[x+1] == c[4]
```

```
c[x-1] == c[2]
```

# Arrays

| Operators |    |    |        |    |    | Associativity | Type           |
|-----------|----|----|--------|----|----|---------------|----------------|
| []        | () |    |        |    |    | left to right | highest        |
| ++        | -- | !  | (type) |    |    | right to left | unary          |
| *         | /  | %  |        |    |    | left to right | multiplicative |
| +         | -  |    |        |    |    | left to right | additive       |
| <         | <= | >  | >=     |    |    | left to right | relational     |
| ==        | != |    |        |    |    | left to right | equality       |
| &&        |    |    |        |    |    | left to right | logical and    |
|           |    |    |        |    |    | left to right | logical or     |
| ?:        |    |    |        |    |    | right to left | conditional    |
| =         | += | -= | *=     | /= | %= | right to left | assignment     |
| ,         |    |    |        |    |    | left to right | comma          |

**Fig. 6.2** Operator precedence.

double x[8];

| x[0] | x[1] | x[2] | x[3] | x[4] | x[5] | x[6] | x[7]  |
|------|------|------|------|------|------|------|-------|
| 16.0 | 12.0 | 6.0  | 8.0  | 2.5  | 12.0 | 14.0 | -54.5 |

i=5

printf("%d %.1f", 4, x[4]);

4 2.5

printf("%d %.1f", i, x[i]);

5 12.0

printf("%.1f", x[i]+1);

13.0

printf("%.1f", x[i]+i);

17.0

printf("%.1f", x[i+1]);

14.0

printf("%.1f", x[i+i]);

invalid

printf("%.1f", x[2\*i]);

invalid

printf("%.1f", x[2\*i-3]);

-54.5

printf("%.1f", x[(int)x[4]]);

6.0

printf("%.1f", x[i++]);

12.0

printf("%.1f", x[--i]);

12.0

May result in a run-time error  
Display incorrect results

```
double x[8];
```

| x[0] | x[1] | x[2] | x[3] | x[4] | x[5] | x[6] | x[7]  |
|------|------|------|------|------|------|------|-------|
| 16.0 | 12.0 | 6.0  | 8.0  | 2.5  | 12.0 | 14.0 | -54.5 |

```
i=5
```

```
x[i-1] = x[i]
```

```
x[i] = x[i+1]
```

```
x[i]-1 = x[i]
```

Illegal assignment statement!

# Defining Arrays

- When defining arrays, specify

- Type of array
- Name
- Number of elements

```
arrayType arrayName[ numberOfElements ];
```

Examples:

```
int c[ 10 ];  
float myArray[ 3284 ];
```

- Defining multiple arrays of same type

- Format similar to regular variables
- Example:

```
int b[ 100 ], x[ 27 ];
```

# Examples Using Arrays

- Initializers

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

- If not enough initializers, elements with missing values become 0

```
int n[ 5 ] = { 0 }
```

All elements 0

- C arrays have no bounds checking

- If size omitted, initializers determine it

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

- 5 initializers, therefore 5 element array

# Initializing an Array

```
1  /* Fig. 6.3: fig06_03.c
2     initializing an array */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     int n[ 10 ]; /* n is an array of 10 integers */
9     int i;       /* counter */
10
11     /* initialize elements of array n to 0 */
12     for ( i = 0; i < 10; i++ ) {
13         n[ i ] = 0; /* set element at location i to 0 */
14     } /* end for */
15
16     printf( "%s%13s\n", "Element", "Value" );
17
18     /* output contents of array n in tabular format */
19     for ( i = 0; i < 10; i++ ) {
20         printf( "%7d%13d\n", i, n[ i ] );
21     } /* end for */
22
23     return 0; /* indicates successful termination */
24
25 } /* end main */
```



## Program Output

| Element | value |
|---------|-------|
| 0       | 0     |
| 1       | 0     |
| 2       | 0     |
| 3       | 0     |
| 4       | 0     |
| 5       | 0     |
| 6       | 0     |
| 7       | 0     |
| 8       | 0     |
| 9       | 0     |

```
1  /* Fig. 6.4: fig06_04.c
2      Initializing an array with an initializer list */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8      /* use initializer list to initialize array n */
9      int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10     int i; /* counter */
11
12     printf( "%s%13s\n", "Element", "Value" );
13
14     /* output contents of array in tabular format */
15     for ( i = 0; i < 10; i++ ) {
16         printf( "%7d%13d\n", i, n[ i ] );
17     } /* end for */
18
19     return 0; /* indicates successful termination */
20
21 } /* end main */
```

## Program Output

| Element | Value |
|---------|-------|
| 0       | 32    |
| 1       | 27    |
| 2       | 64    |
| 3       | 18    |
| 4       | 95    |
| 5       | 14    |
| 6       | 90    |
| 7       | 70    |
| 8       | 60    |
| 9       | 37    |

```
1  /* Fig. 6.5: fig06_05.c
2     Initialize the elements of array s to the even integers from 2 to 20 */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* function main begins program execution */
7  int main()
8  {
9     /* symbolic constant SIZE can be used to specify array size */
10    int s[ SIZE ]; /* array s has 10 elements */
11    int j;          /* counter */
12
13    for ( j = 0; j < SIZE; j++ ) { /* set the values */
14        s[ j ] = 2 + 2 * j;
15    } /* end for */
16
17    printf( "%s%13s\n", "Element", "value" );
18
19    /* output contents of array s in tabular format */
20    for ( j = 0; j < SIZE; j++ ) {
21        printf( "%7d%13d\n", j, s[ j ] );
22    } /* end for */
23
24    return 0; /* indicates successful termination */
25
26 } /* end main */
```

## Program Output

| Element | Value |
|---------|-------|
| 0       | 2     |
| 1       | 4     |
| 2       | 6     |
| 3       | 8     |
| 4       | 10    |
| 5       | 12    |
| 6       | 14    |
| 7       | 16    |
| 8       | 18    |
| 9       | 20    |

# Examples

- Reading values into an array

```
int i, x[100];
```

```
for (i=0; i < 100; i=i+1)
{
    printf("Enter an integer: ");
    scanf("%d",&x[i]);
}
```

- Summing up all elements in an array

```
int sum = 0;
for (i=0; i<=99; i=i+1)
    sum = sum + x[i];
```

# Examples

- Finding the location of a given value (`item`) in an array.

```
i = 0;
while ((i < 100) && (x[i] != item))
    i = i + 1;

if (i == 100)
    loc = -1; // not found
else
    loc = i;  // found in location i
```

# Examples

- Shifting the elements of an array to the left.

```
/* store the value of the first element in a
 * temporary variable
 */
temp = x[0];

for (i=0; i < 99; i=i+1)
    x[i] = x[i+1];

//The value stored in temp is going to be
the value of the last element:
x[99] = temp;
```



```

1  /* Fig. 6.6: fig06_06.c
2      Compute the sum of the elements of the array */
3  #include <stdio.h>
4  #define SIZE 12
5
6  /* function main begins program execution */
7  int main()
8  {
9      /* use initializer list to initialize array */
10     int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11     int i;          /* counter */
12     int total = 0; /* sum of array */
13
14     /* sum contents of array a */
15     for ( i = 0; i < SIZE; i++ ) {
16         total += a[ i ];
17     } /* end for */
18
19     printf( "Total of array element values is %d\n", total );
20
21     return 0; /* indicates successful termination */
22
23 } /* end main */

```

Total of array element values is 383

```
1  /* Fig. 6.7: fig06_07.c
2      Student poll program */
3  #include <stdio.h>
4  #define RESPONSE_SIZE 40 /* define array sizes */
5  #define FREQUENCY_SIZE 11
6
7  /* function main begins program execution */
8  int main()
9  {
10     int answer; /* counter */
11     int rating; /* counter */
12
13     /* initialize frequency counters to 0 */
14     int frequency[ FREQUENCY_SIZE ] = { 0 };
15
16     /* place survey responses in array responses */
17     int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18         1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19         5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
```

```

21  /* for each answer, select value of an element of array responses
22      and use that value as subscript in array frequency to
23      determine element to increment */
24  for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
25      ++frequency[ responses [ answer ] ];
26  } /* end for */
27
28  /* display results */
29  printf( "%s%17s\n", "Rating", "Frequency" );
30
31  /* output frequencies in tabular format */
32  for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
33      printf( "%6d%17d\n", rating, frequency[ rating ] );
34  } /* end for */
35
36  return 0; /* indicates successful termination */
37
38 } /* end main */

```

| Rating | Frequency |
|--------|-----------|
| 1      | 2         |
| 2      | 2         |
| 3      | 2         |
| 4      | 2         |
| 5      | 5         |
| 6      | 11        |
| 7      | 5         |
| 8      | 7         |
| 9      | 1         |
| 10     | 3         |

```
1  /* Histogram printing program */
2
3  #include <stdio.h>
4  #define SIZE 10
5
6  int main()
7  {
8      int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9      int i, j;
10
11     printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
12
13     for ( i = 0; i <= SIZE - 1; i++ ) {
14         printf( "%7d%13d", i, n[i] ) ;
15
16         for ( j = 1; j <= n[ i ]; j++ )    /* print one bar */
17             printf( "%c", '*' );
18
19         printf( "\n" );
20     }
21
22     return 0;
23 }
```

## Program Output

| Element | value | Histogram |
|---------|-------|-----------|
| 0       | 19    | *****     |
| 1       | 3     | ***       |
| 2       | 15    | *****     |
| 3       | 7     | *****     |
| 4       | 11    | *****     |
| 5       | 9     | *****     |
| 6       | 13    | *****     |
| 7       | 5     | *****     |
| 8       | 17    | *****     |
| 9       | 1     | *         |

```
1  /* Fig. 6.9: fig06_09.c
2      Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define SIZE 7
7
8  /* function main begins program execution */
9  int main()
10 {
11     int face;                /* random number with value 1 - 6 */
12     int roll;                /* roll counter */
13     int frequency[ SIZE ] = { 0 }; /* initialize array to 0 */
14
15     srand( time( NULL ) ); /* seed random-number generator */
16
17     /* roll die 6000 times */
18     for ( roll = 1; roll <= 6000; roll++ ) {
19         face = rand() % 6 + 1;
20         ++frequency[ face ]; /* replaces 26-line switch of Fig. 5.8 */
21     } /* end for */
22
23     printf( "%s%17s\n", "Face", "Frequency" );
24
```

```
25  /* output frequency elements 1-6 in tabular format */
26  for ( face = 1; face < SIZE; face++ ) {
27      printf( "%4d%17d\n", face, frequency[ face ] );
28  } /* end for */
29
30  return 0; /* indicates successful termination */
31
32 } /* end main */
```

## Program Output

| Face | Frequency |
|------|-----------|
| 1    | 1029      |
| 2    | 951       |
| 3    | 987       |
| 4    | 1033      |
| 5    | 1010      |
| 6    | 990       |

# Multi-Dimensional Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (m by n array)
  - Like matrices: specify row, then column

|       | Column 0    | Column 1    | Column 2    | Column 3    |
|-------|-------------|-------------|-------------|-------------|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

Diagram illustrating the structure of a multi-dimensional array (matrix) with row and column subscripts.

The array is represented as a table with rows and columns. The rows are labeled Row 0, Row 1, and Row 2. The columns are labeled Column 0, Column 1, Column 2, and Column 3.

The elements are indexed using the format `a[ row ][ column ]`. For example, the element at Row 0, Column 0 is `a[ 0 ][ 0 ]`.

Annotations:

- Array name: `a`
- Row subscript: The first index (e.g., `0` in `a[ 0 ][ 0 ]`)
- Column subscript: The second index (e.g., `0` in `a[ 0 ][ 0 ]`)



# Multi-Dimensional Arrays

- Initialization

- `int b[2][2] = { { 1, 2 }, { 3, 4 } };`
- Initializers grouped by row in braces
- If not enough, unspecified elements set to zero  
`int b[2][2] = { { 1 }, { 3, 4 } };`

|   |   |
|---|---|
| 1 | 2 |
| 3 | 4 |

|   |   |
|---|---|
| 1 | 0 |
| 3 | 4 |

- Referencing elements

- Specify row, then column  
`printf( "%d", b[0][1] );`

# Example: Multi-Dimensional Array

```
#include <stdio.h>
int main()
{
    int i; /* counter */
    int j; /* counter */

    /* initialize array1, array2, array3 */
    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

    printf( "Values in array1 by row are:\n" );

    for ( i = 0; i <= 1; i++ ) {
        for ( j = 0; j <= 2; j++ )
            printf( "%d ", array1[ i ][ j ] );
        printf( "\n" );
    }
}
```

/\* loop through rows \*/

/\* output column values \*/

# Example: Multi-Dimensional Array

```
printf( "Values in array2 by row are:\n" );
for ( i = 0; i <= 1; i++ ) {                               /* loop through rows */
    for ( j = 0; j <= 2; j++ )
        printf( "%d ", array2[ i ][ j ] );                 /* output column values */
    printf( "\n" );
}

printf( "Values in array3 by row are:\n" );
for ( i = 0; i <= 1; i++ ) {                               /* loop through rows */
    for ( j = 0; j <= 2; j++ )
        printf( "%d ", array3[ i ][ j ] );
    printf( "\n" );
}

return 0;
}
```

Values in array1 by row are:

1 2 3

4 5 6

Values in array2 by row are:

1 2 3

4 5 0

Values in array3 by row are:

1 2 0

4 0 0

# Pointers and Arrays

- Arrays are implemented as pointers.

- Consider:

```
double list[3];
```

`&list[1]` : is the address of the second element

`&list[i]` : the address of `list[i]` which is calculated by the formula

*base address of the array + i \* 8*

# The Relationship between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name is like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:  
`bPtr = b;`
    - The array name (`b`) is actually the address of first element of the array `b[ 5 ]`  
`bPtr = &b[ 0 ]`
  - Explicitly assigns `bPtr` to address of first element of `b`

# The Relationship between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `* ( bPtr + 3 )`
    - Where `n` is the offset. Called pointer/offset notation
  - Can be accessed by `bPtr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself  
`* ( b + 3 )`

# Example (cont.)

```
/* Using subscripting and pointer notations with  
   arrays */  
#include <stdio.h>  
int main(void)  
{  
    int i, offset, b[4]={10,20,30,40};  
    int *bPtr = b;  
  
    /* Array is printed with array subscript notation */  
  
    for (i=0; i < 4; i++)  
        printf("b[%d] = %d\n", i, b[i]);
```

# Example (cont.)

```
/* Pointer/offset notation where the pointer is  
the array name */
```

```
for (offset=0; offset < 4; offset++)  
    printf("* (b + %d) = %d\n", offset, * (b + offset));
```

```
/* Pointer subscript notation */
```

```
for (i=0; i < 4; i++)  
    printf("bPtr[%d] = %d\n", i, bPtr[i]);
```

```
/* Pointer offset notation */
```

```
for (offset = 0; offset < 4; offset++)  
    printf("* (bPtr + %d) = %d\n", offset"  
        " * (bPtr + offset)");
```

```
return 0;  
}
```



# Example (cont.)

b[ 0 ] = 10

b[ 1 ] = 20

b[ 2 ] = 30

b[ 3 ] = 40

\*( b + 0 ) = 10

\*( b + 1 ) = 20

\*( b + 2 ) = 30

\*( b + 3 ) = 40

bPtr[ 0 ] = 10

bPtr[ 1 ] = 20

bPtr[ 2 ] = 30

bPtr[ 3 ] = 40

\*( bPtr + 0 ) = 10

\*( bPtr + 1 ) = 20

\*( bPtr + 2 ) = 30

\*( bPtr + 3 ) = 40

# Passing Arrays as Parameters

- Arrays are passed to functions **by reference**
  - the called functions can modify the element values in the callers' original arrays
  - individual array elements are passed **by value** exactly as simple variables are
- The function's parameter list must specify that an array will be received
- `int SumIntegerArray(int a[], int n)`
- It indicates that `SumIntegerArray` expects to receive an array of integers in parameter `a` and the number of array elements in parameter `n`
- The size of the array is not required between the array brackets

# Passing Arrays as Parameters

```
int SumIntegerArray(int a[], int n)
{
    int i, sum;
    sum = 0;
    for (i=0; i < n; i++) {
        sum += a[i];
    }
    return sum;
}
```

**Assume**

```
int sum, list[5];
```

are declared in the main function. We can make the following function call:

```
sum = SumIntegerArray(list, 5);
```

# Strings

- Character arrays
  - String “first” is really a static array of characters
  - Character arrays can be initialized using string literals
    - `char string1[] = "first";`
    - Null character `'\0'` terminates strings
    - `string1` actually has 6 elements
      - equivalent to `char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };`
  - Can access individual characters
    - `string1[3]` is character ‘s’
- Array name is address of array, so `&` not needed for `scanf`
  - `scanf( "%s", string2 );`
  - Reads characters until whitespace encountered
  - Can write beyond end of array, be careful

```

1  /* Fig. 6.10: fig06_10.c
2     Treating character arrays as strings */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     char string1[ 20 ];           /* reserves 20 characters */
9     char string2[] = "string literal"; /* reserves 15 characters */
10    int i;                        /* counter */
11
12    /* read string from user into array string2 */
13    printf("Enter a string: ");
14    scanf( "%s", string1 );
15
16    /* output strings */
17    printf( "string1 is: %s\nstring2 is: %s\n"
18           "string1 with spaces between characters is:\n",
19           string1, string2 );
20
21    /* output characters until null character is reached */
22    for ( i = 0; string1[ i ] != '\0'; i++ ) {
23        printf( "%c ", string1[ i ] );
24    } /* end for */
25    printf( "\n" );
26
27
28    return 0; /* indicates successful termination */
29
30 } /* end main */

```

## Program Output

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

# Example: String Copy

```
/* Copying a string using array notation and pointer notation. */
#include <stdio.h>
void copy1( char *s1, const char *s2 );
void copy2( char *s1, const char *s2 );

int main()
{
    char string1[ 10 ];           /* create array string1 */
    char *string2 = "Hello";      /* create a pointer to a string */
    char string3[ 10 ];           /* create array string3 */
    char string4[] = "Good Bye"; /* create a pointer to a string */

    copy1( string1, string2 );
    printf( "string1 = %s\n", string1 );
    copy2( string3, string4 );
    printf( "string3 = %s\n", string3 );

    return 0;
}
```

# Example

```
/* copy s2 to s1 using array notation */
void copy1( char *s1, const char *s2 )
{
    int i;
    for ( i = 0; s2[ i ] != '\0'; i++ )
        s1[ i ] = s2[ i ];
    s1[ i ] = NULL;
} /* end function copy1 */

/* copy s2 to s1 using pointer notation */
void copy2( char *s1, const char *s2 )
{
    /* loop through strings */
    for ( ; *s2 != '\0'; s1++, s2++ )
        *s1 = *s2;
    *s1 = NULL;
} /* end function copy2 */
```

## Program Output

```
string1 = Hello
string3 = Good Bye
```

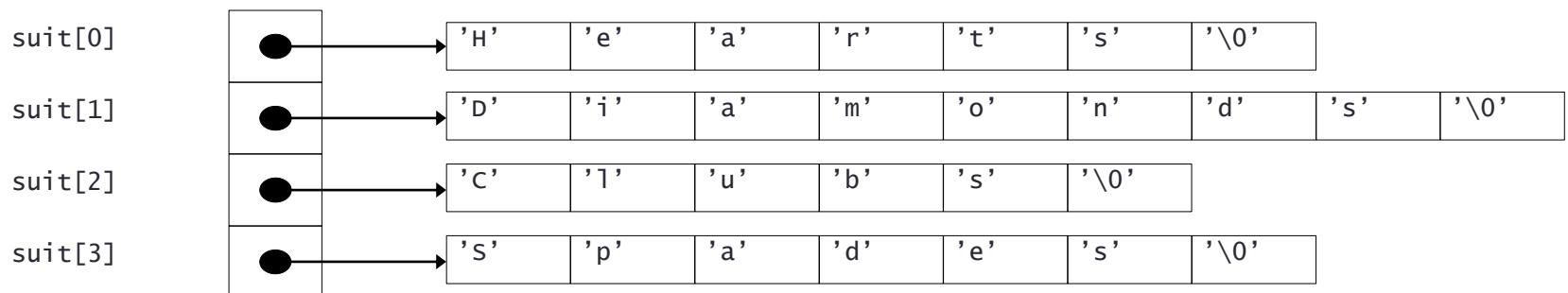


# Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

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

- Strings are pointers to the first character
- `char *` – each element of `suit` is a pointer to a char
- The strings are not actually stored in the array `suit`, only pointers to the strings are stored



- `suit` array has a fixed size, but strings can be of any size

# sizeof function

- **sizeof**
  - Returns size of operand in bytes
  - For arrays: size of 1 element \* number of elements
  - if `sizeof( int )` equals 4 bytes, then

```
int myArray[ 10 ];  
printf( "%d", sizeof( myArray ) );
```

    - will print 40
- **sizeof** can be used with
  - Variable names
  - Type name
  - Constant values

# Example

```
/* sizeof operator when used on an array name returns the number of
   bytes in the array. */
#include <stdio.h>
size_t getSize( float *ptr ); /* prototype */

int main(){
    float array[ 20 ]; /* create array */

    printf( "The number of bytes in the array is %d"
            "\nThe number of bytes returned by getSize is %d\n",
            sizeof( array ), getSize( array ) );

    return 0;
}

size_t getSize( float *ptr ) {
    return sizeof( ptr );
}
```

## Program Output

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

# Example

```
/* Demonstrating the sizeof operator */
#include <stdio.h>

int main()
{
    char c;           /* define c */
    short s;          /* define s */
    int i;             /* define i */
    long l;           /* define l */
    float f;          /* define f */
    double d;         /* define d */
    long double ld;    /* define ld */
    int array[ 20 ];   /* initialize array */
    int *ptr = array; /* create pointer to array */
}
```

# Example

```
printf( "      sizeof c = %d\\tsizeof(char)  = %d"
        "\\n      sizeof s = %d\\tsizeof(short) = %d"
        "\\n      sizeof i = %d\\tsizeof(int)  = %d"
        "\\n      sizeof l = %d\\tsizeof(long)  = %d"
        "\\n      sizeof f = %d\\tsizeof(float) = %d"
        "\\n      sizeof d = %d\\tsizeof(double) = %d"
        "\\n      sizeof ld = %d\\tsizeof(long double) = %d"
        "\\n      sizeof array = %d"
        "\\n      sizeof ptr = %d\\n",
        sizeof c, sizeof( char ), sizeof s,
        sizeof( short ), sizeof i, sizeof( int ),
        sizeof l, sizeof( long ), sizeof f,
        sizeof( float ), sizeof d, sizeof( double ),
        sizeof ld, sizeof( long double ),
        sizeof array, sizeof ptr );

return 0;
}
```

# Example

## Program Output

```
sizeof c = 1
sizeof s = 2
sizeof i = 4
sizeof l = 4
sizeof f = 4
sizeof d = 8
sizeof ld = 8
sizeof array = 80
sizeof ptr = 4
```

```
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 4
sizeof(long) = 4
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 8
```

# Dynamic Memory Management

- **Static memory allocation:** space for the object is provided in the binary at compile-time
- **Dynamic memory allocation:** blocks of memory of arbitrary size can be requested at run-time
- The four dynamic memory management functions are `malloc`, `calloc`, `realloc`, and `free`.
- These functions are included in the header file `<stdlib.h>`.

# Dynamic Memory Management

- `void *malloc(size_t size);`
- allocates storage for an object whose size is specified by `size`:
  - It returns a pointer to the allocated storage,
  - `NULL` if it is not possible to allocate the storage requested.
  - The allocated storage is not initialized in any way.
- **e.g.** `float *fp, fa[10];`  
`fp = (float *) malloc(sizeof(fa));`  
allocates the storage to hold an array of 10 floating-point elements, and assigns the pointer to this storage to `fp`.



# Dynamic Memory Management

- `void *calloc(size_t nobj, size_t size);`
- allocates the storage for an array of `nobj` objects, each of `size` `size`.
  - It returns a pointer to the allocated storage,
  - `NULL` if it is not possible to allocate the storage requested.
  - The allocated storage is initialized to zeros.
- **e.g.** `double *dp, da[10];`  
`dp=(double *) calloc(10,sizeof(double));`  
allocates the storage to hold an array of 10 double values, and assigns the pointer to this storage to `dp`.

# Dynamic Memory Management

- `void *realloc(void *p, size_t size);`
- changes the size of the object pointed to by `p` to `size`.
  - It returns a pointer to the new storage,
  - `NULL` if it is not possible to resize the object, in which case the object (`*p`) remains unchanged.
  - The new size may be larger (the original contents are preserved and the remaining space is uninitialized) or smaller (the contents are unchanged upto the new size) than the original size.

# Dynamic Memory Management

- **e.g.** `char *cp;`

```
cp =(char *) malloc(sizeof("computer"));
```

```
strcpy(cp, "computer");
```

`cp` points to an array of 9 characters containing the null-terminated string `computer`.

```
cp = (char *) realloc(cp, sizeof("compute"));
```

discards the trailing `'\0'` and makes `cp` point to an array of 8 characters containing the characters in `computer`

```
cp=(char *)realloc(cp,sizeof("computerization"));
```

`cp` points to an array of 16 characters, the first 9 of which contain the null-terminated string `computer` and the remaining 7 are uninitialized.

# Dynamic Memory Management

- `void *free(void *p);`
- deallocates the storage pointed to by p, where p is a pointer to the storage previously allocated by malloc, calloc, or realloc.
- **e.g.** `free(fp);`  
`free(dp);`  
`free(cp);`

# Dynamic Memory Management

- When using dynamic memory allocation, test for a `NULL` pointer return value, which indicates unsuccessful operation
- `free` dynamically allocated memory when it is no longer needed to avoid **memory leaks**
- Then set the pointer to `NULL` to eliminate possible further accesses to that memory
  - Referring to memory that has been deallocated is an error that typically results in the program crashing
- Trying to `free` memory not allocated dynamically is an error

# Example

```
#include<stdio.h>
#include<stdlib.h>

int main(void) {
    int *array, *p;
    int i,no_elements;
    printf("Enter number of elements: ");
    scanf("%d",&no_elements);

    printf("Enter the elements: ");
    array = ( int* )malloc( no_elements*sizeof( int ) );
    for(p=array,i=0; i<no_elements; i++, p++)
        scanf("%d",p);

    printf("Elements: ");
    for(p=array,i=0; i<no_elements; i++, p++)
        printf("%d ",*p);
    printf("\n");
}
```

# Example

```
array = ( int* )realloc(array, (no_elements+2)*sizeof( int ) );

printf("Enter two new elements: ");
for(p=array,i=0; i<no_elements; i++, p++) ;

for(; i<no_elements+2; i++, p++)
    scanf("%d",p);

printf("Elements: ");
for(p=array,i=0; i<no_elements+2; i++, p++)
    printf("%d ",*p);
printf("\n");

free(array);
return 0;
}
```

```
Enter number of elements: 4
Enter the elements: 2 3 4 5
Elements: 2 3 4 5
Enter two new elements: 6 7
Elements: 2 3 4 5 6 7
```

**Program Output**

# Example: Dynamic Allocation of 2D Array (1)

```
#include <stdio.h>
#include <stdlib.h>

#define ROW 3
#define COLUMN 4

/* Using a single pointer */
int main()
{
    int *arr = (int *)malloc(ROW * COLUMN * sizeof(int));

    int i, j, count = 0;
    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            *(arr + i*COLUMN + j) = ++count;

    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            printf("%d ", *(arr + i*COLUMN + j));

    return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12
```



# Example: Dynamic Allocation of 2D Array (2)

```
#include <stdio.h>
#include <stdlib.h>

#define ROW 3
#define COLUMN 4

/* Using an array of pointers */
int main()
{
    int i, j, count;

    int *arr[ROW];
    for (i=0; i<ROW; i++)
        arr[i] = (int *)malloc(COLUMN * sizeof(int));

    // Note that arr[i][j] is same as (*(arr+i)+j)
    count = 0;
    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            arr[i][j] = ++count; // OR (*(arr+i)+j) = ++count

    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            printf("%d ", arr[i][j]);

    return 0;
}
```

1 2 3 4 5 6 7 8 9 10 11 12

# Example: Dynamic Allocation of 2D Array (3)

```
#include <stdio.h>
#include <stdlib.h>

#define ROW 3
#define COLUMN 4

/* Using pointer to a pointer */
int main()
{
    int i, j, count;

    int **arr = (int **)malloc(ROW * sizeof(int *));
    for (i=0; i<ROW; i++)
        arr[i] = (int *)malloc(COLUMN * sizeof(int));

    // Note that arr[i][j] is same as *((arr+i)+j)
    count = 0;
    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            arr[i][j] = ++count; // OR *((arr+i)+j) = ++count

    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            printf("%d ", arr[i][j]);

    return 0;
}
```

1 2 3 4 5 6 7 8 9 10 11 12

# Example: Dynamic Allocation of 2D Array (4)

```
#include<stdio.h>
#include<stdlib.h>

#define ROW 3
#define COLUMN 4

/* Using double pointer and one malloc call for all rows */
int main()
{
    int **arr;
    int count = 0,i,j;

    arr = (int **)malloc(sizeof(int *) * ROW);
    arr[0] = (int *)malloc(sizeof(int) * COLUMN * ROW);

    for(i = 0; i < ROW; i++)
        arr[i] = (*arr + COLUMN * i);

    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            arr[i][j] = ++count; // OR (*(arr+i)+j) = ++count

    for (i = 0; i < ROW; i++)
        for (j = 0; j < COLUMN; j++)
            printf("%d ", arr[i][j]);

    return 0;
}
```

1 2 3 4 5 6 7 8 9 10 11 12

# Structured Programming with C

## Wrap-up

---

# Programming Paradigms

- Programming Paradigm : Defines how to think when you are programming. Way of approaching the task.
- Common Paradigms;
  - Imperative : Describe all the steps required to perform a task. Which variables to declare, what to assign etc. Describe how to do it!
  - Declarative : Specify the result you want, now how to do it. E.g. SQL for databases
  - Functional : Programming with function calls, avoid global state
  - Object-Oriented: Structure the program with Objects and relationships between these objects
- C is an Imperative programming language.

# Computing with Pen & Paper

- Think of computing by a person using pen and paper:
  - Task: Check if there are same number of 1s and 0s in a given String (e.g. 100101001110010100)
  - Go through the String till the end, for each 1, add a new mark to the paper
  - Go through the String till the end, for each 0, strike out the leftmost mark without a strike.
  - If we can't find a mark to strike-out for a 0 then there are more 0s
  - If after processing all zeros there are marks that are not striked-out, there are more 1s.
  - Otherwise there are same number of 1s and 0s

# States and Operations

- When designing a C program (applies to imperative programming in general) we think of the problem as a set of variables, and operations changing them
- Just like pen & paper example;
  - Paper stands for our variables (marking & striking are assignment)
  - The procedure executed by the person corresponds to our program
- Data types: To simplify our job, we define datatypes
  - Instead of a mark we can store integers, doubles or more complex datatypes (structs)
  - Decide on what data to store, and what type they should be
- Functions, Objects :
  - Simplify the algorithm description; break down the task into subtasks.

# Design of Program

- Given a problem;
  - Program flow : Roughly sketch the execution of the program, what are the decisions, loops and sequence of the operations.
  - Identify: Data types, data structures. Will using a struct make this task easier?
  - Top-down and/or Bottom-up design: How to structure the problem. (More on this later)
  - Coding: Use all tools provided by programming language to implement the idea. As you realize the problem, you can always change things in previous steps. Code changes design, design changes code (Refactoring).
  - Testing and Debugging : Will always have errors, establish habits and procedures to avoid them.



# Example: Tic-Tac-Toe Specs (1)

The program is to print a welcome message to introduce the game and its rules.

Welcome to TIC-TAC-TOE.

-----

The object of this game is to get a line of X's before the computer gets a line of O's. A line maybe across, down, or diagonal. The board is labelled from 1 to 9 as follows:

1|2|3

-----

4|5|6

-----

7|8|9

# Tic-tac-toe Specs (2)

This is followed by a request for whether the user wishes to start, with the integer 1 meaning 'yes' and 0 for 'no'.

Do you wish to go first (1-Yes, 0-No) ?

Each time the user is to make a move, a request is made for which location he wishes to choose, designated by a number between 1 and 9.

Your turn (1 - 9):

After the user or the computer has made a decision, the resulting table is printed in ASCII. For example, after 5 turns, the board might look like

```
x| |  
-----  
x|o|  
-----  
|x|o
```

# Tic-Tac-Toe Specs (3)

The game will terminate with a winner or a draw, and a comment is to be printed on account of the user's win, loss or draw, respectively.

44

You win. Congratulations!!

You lose. Better luck next time.

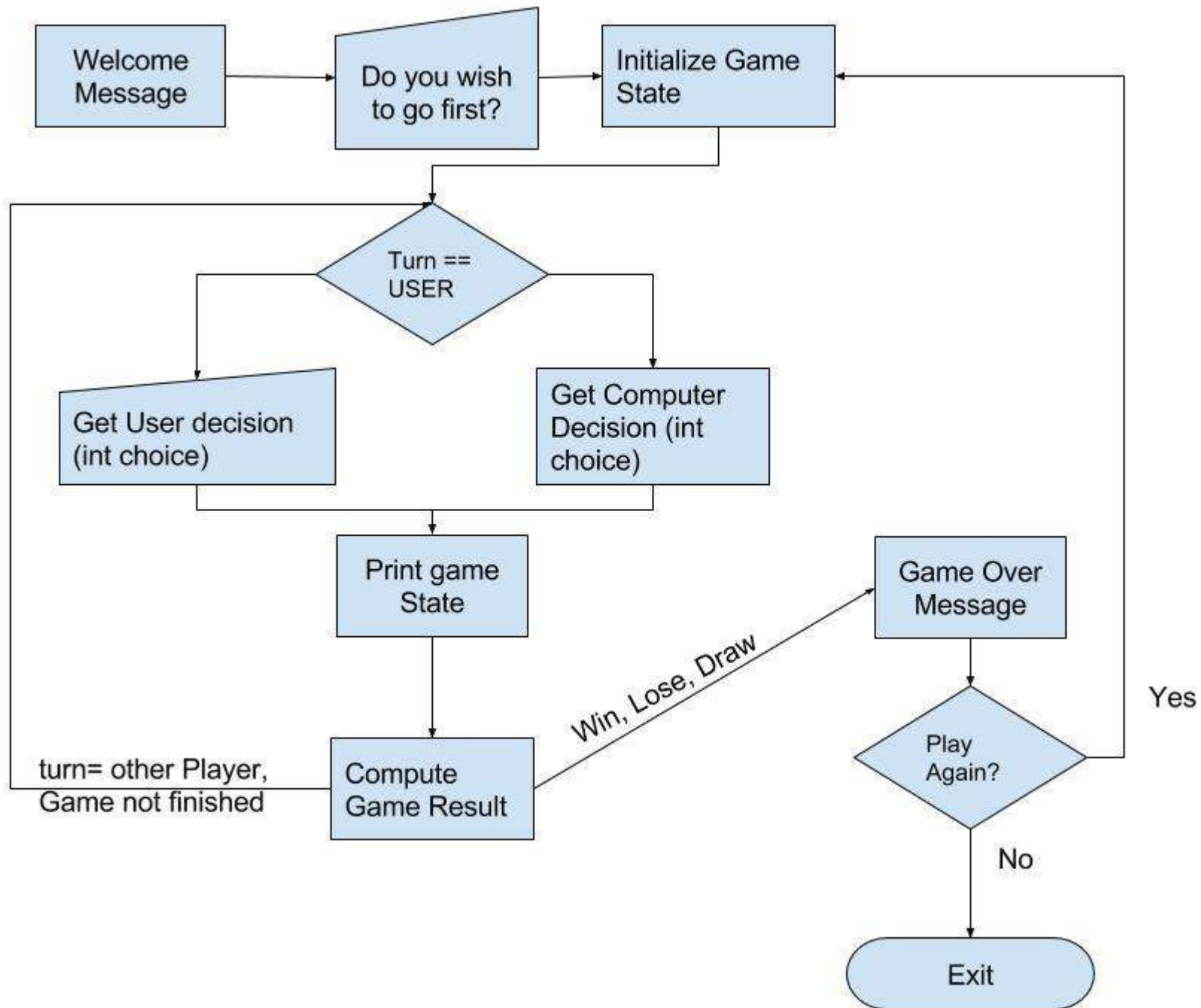
Its a draw. How dull.

Finally, the user is asked if he wishes to play again and, if so, the game returns to the request of whether he wishes to have first go. Otherwise, the program terminates.

Do you wish to play again (1-Yes, 0-No) ?

# Program Flow

- Model the flow of the program with respect to the specifications
- Flow will be a very simple flowchart showing the major tasks, decisions and loops
- You can probably spot the data needed and its types at this point.
- Try to picture where each variable is created and to which tasks it must be passed on to.
- You can always come-up with multiple flows and you can always revise your initial design.



# Top-down and Bottom-up Design

- Top-down: Start with a set of high level tasks (that will probably be called from main)
  - Split each task to manageable (you can code a function for it) recursively
  - Divide-and-conquer; will create a hierarchy of lower-level functions to implement
  - The functions you will identify might be too coupled with the task (not reusable)
- Bottom-up: identify various components that will be required
  - Implement first the required functions, more generically.

# Pseudocode Design

- Another good idea is to convert the flow diagram to simple pseudocode
- Write the pseudocode as C comments
- Start implementing each item in pseudocode, and leave the comments
- Example:

Loop number of times

Prompt user and get integer value

Calculate factorial

Print factorial

# Tic-tac-toe: Data structure

- Think of what represents a state of the program
    - Whose Turn is it?
    - The tic-tac-toe table, which cells are marked with X or O?
  - Types of the variables:
- another alternative is to define as struct:

```
#define NUMSTATES 9
enum { NOTHING, CROSS, NOUGHT };
int state[NUMSTATES];
```

```
struct tictactoe_state {
    int row; int col;
    int state = NOTHING; };
```



# Bottom-up Design

- Looking at the flow and specifications, we can determine the functions we need
- Print the message and get a value as input:  
`int getint_from_user(char* message);`
- Plot the tic-tac-toe table;  
`void plot_state(int state[]);`
- Let the computer decide on a play  
`void get_computer_decision(int state[]);`  
can initially implement it as purely random, but can also try and test more sophisticated things.

# Bottom-up Design

- Make your code more readable with enums, constants

```
enum Turn { USER, COMPUTER };  
enum Result { PLAYING, WIN, LOSE, DRAW };  
enum Turn turn;  
enum Result result;
```

- Check if the game is over or still continuing

```
enum Result compute_result(int state[], enum Turn turn);
```

# Bottom-up Design

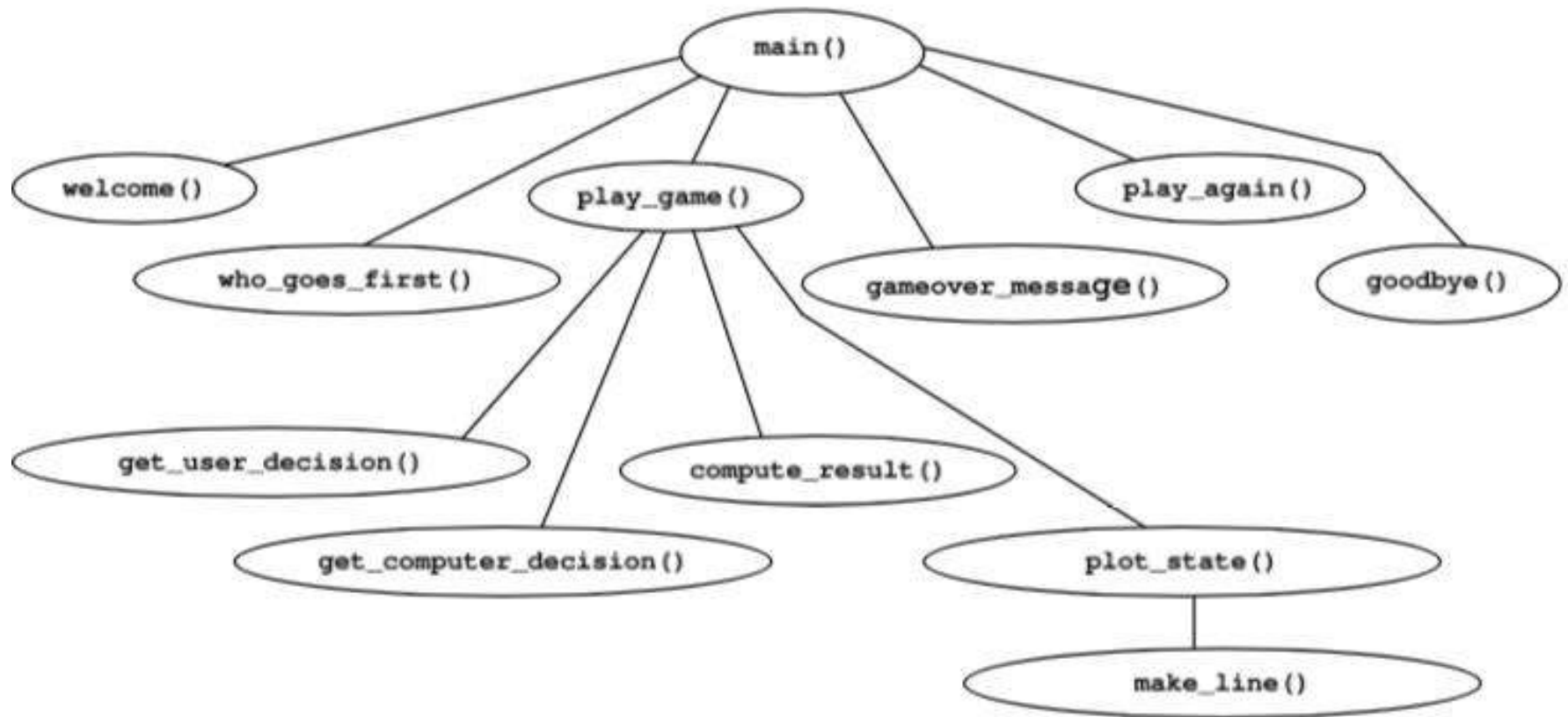
- Implement each function independently
- Write a driver method (main method) to test your function.
  - Make sure to test edge cases with multiple inputs. Did you check the diagonal?
- The function prototypes are your checklist, cross-out an item when you have tested your implementation thoroughly.
- Consider keeping your test codes around (configure multiple main methods in a target called test in your Makefile)

# Top-down Design

- Start coding the main function:

```
int main(void){
    enum Turn turn; enum Result result; int newgame = 1;
    welcome();
    while (newgame) {
        turn = who_goes_first();
        result = play_game(turn);
        gameover_message(result);
        newgame = play_again();
    };
    goodbye();
    return 0;
}
```

# Top-Down Hierarchy

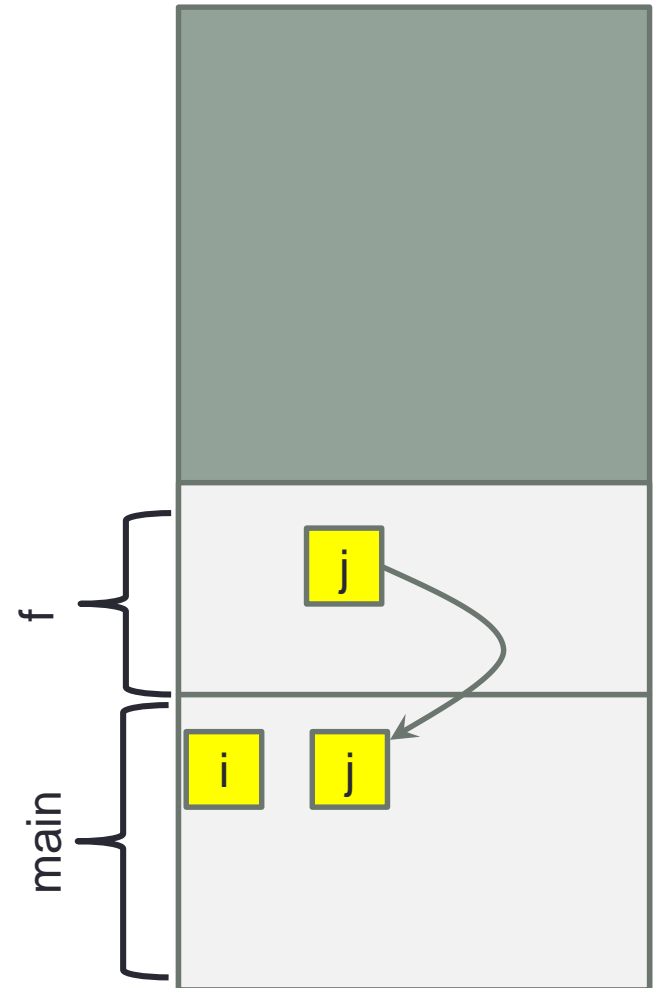


# Functions implementation

- It might be confusing at first to create functions and pass the required variables.
- Do not back down because of compiler errors, insist on correct modular design.
  - Never convert a variable to global to get rid of an error.
- We will see some example cases for function designs.

# Return multiple values

```
int f(int* j) {  
    *j=4;  
    return 5;  
}  
  
int main(void) {  
    int i=0;  
    int j=0;  
    i = f(&j);  
    printf("%d %d \n", i, j);  
  
    return EXIT_SUCCESS;  
}
```

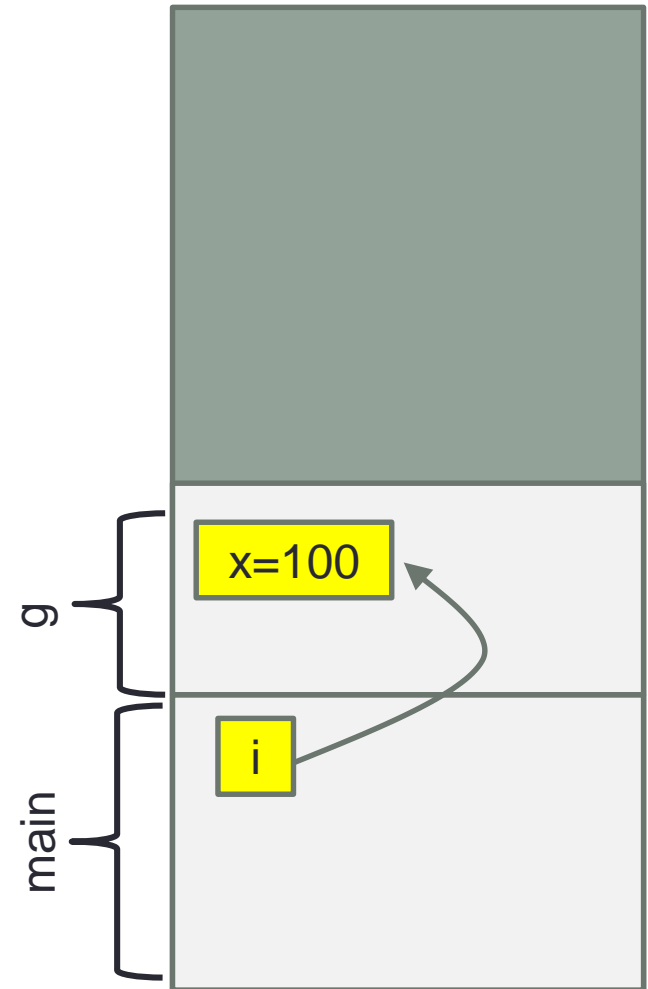


# Beware of the reverse

```
int* g() {  
    int x = 100;  
    return &x;  
}  
int main(void) {  
    int* i;  
    i = g();  
    printf("%d \n", *i);  
  
    return EXIT_SUCCESS;  
}
```

- Returning stack variables is never a good idea
- After g terminates, x will be removed.
- If you have checked the warnings;

Test.c:21:2: warning: function returns address of local variable [-Wreturn-local-addr]





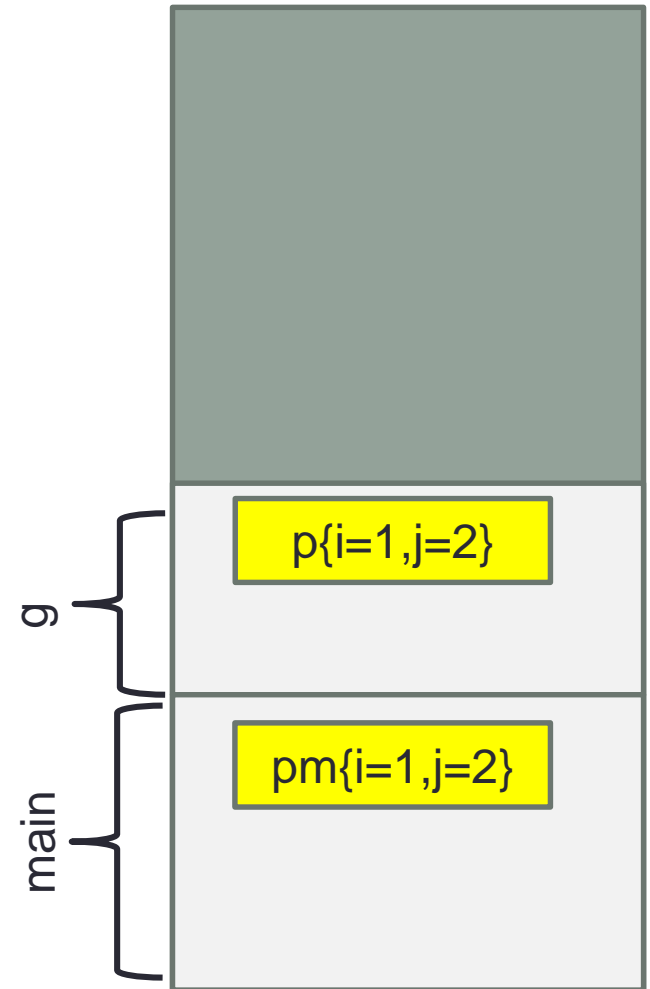
# Returning with structs or arrays

- Instead of using pass-by-reference, you can use an array or a struct to return multiple values from a function
- Array solution: You should allocate the array from Heap!
- Struct solution: Your struct will be copied to another struct in the caller function.

# Returning a struct

```
struct pair {  
    int i;  
    int j;  
};  
  
struct pair g() {  
    struct pair p = {.i=1, .j=2};  
    return p;  
}  
  
int main(void) {  
    struct pair pm = g();  
    printf("%d %d \n", pm.i, pm.j);  
    return 0;  
}
```

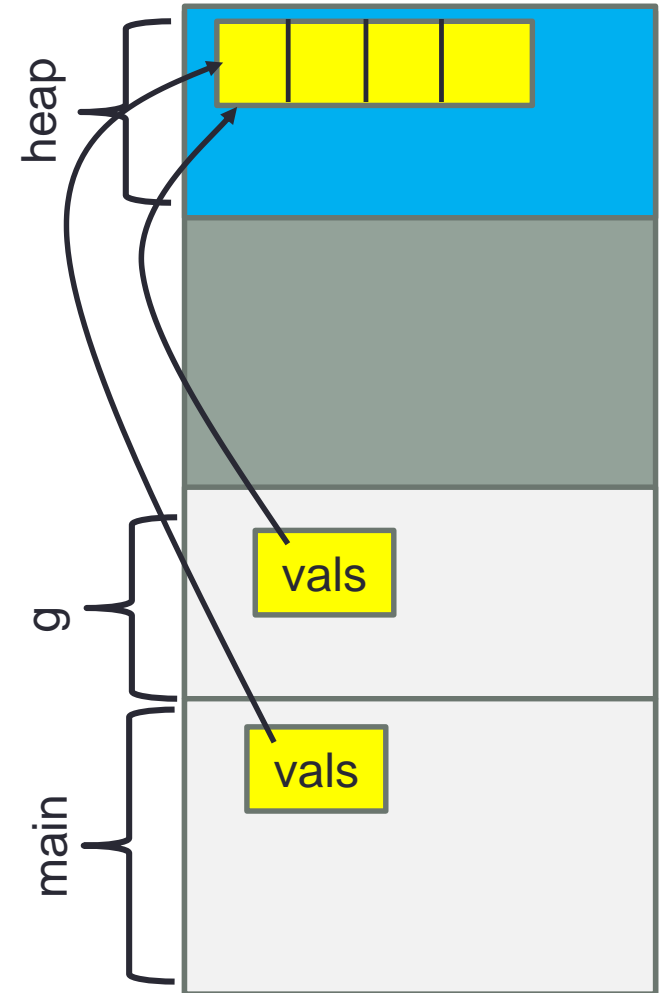
- The returned structs values are copied to variable in main. They have different addresses



# Returning an array

```
int* g() {  
    int* values = malloc(sizeof(int)*2);  
    values[0]=1;  
    values[1]=2;  
    return values;  
}  
  
int main(void) {  
    int* vals = g();  
    printf("%d %d \n", vals[0], vals[1]);  
    free(vals);  
    return 0;  
}
```

- You can also create the array in main and pass its pointer to the called function
- It is always your responsibility to free the memory allocated in Heap



# Laying traps for logical errors

- Logical errors are the worst type of errors
  - Program will not usually crash, but will not do what it is expected to
  - You don't know where the problem is, so finding that will require debugging.
- By being defensive and writing code to prevent these errors you can save significant amount of time.
- Do not cut down from lines of code, cut down from debugging time.

# Trick for comparisons

```
int main(void) {  
    int i=0;  
    if(i=0) {  
        printf("It is zero");  
    }  
    return 0;  
}
```

We wrote = instead of ==.  
C compiler will not give an error,  
but just a warning.  
You can trade this logical error  
with an compiler error.

- This program will not output anything.
- The problem is just one character (if(i=0)), C will return the value assigned to if statement
- One defensive programming strategy is to write it as;  
if(0=i)  
which will give an compiler error if you forget one = character.

# Asserts

```
double circle_area(double radius) {  
    double pi=3.14;  
    assert(radius>0);  
    return pi*radius*radius;  
}  
  
int main(void) {  
    printf("%f", circle_area(-1));  
    return 0;  
}
```

Assertion failed: radius>0, file  
../src/Test.c, line 46

- Protect against logical errors by placing assert statements in your code.
- It will show the failed assertion and the line number.

# Avoid overloading a line

```
int main(void) {  
    int array[4] = {1,2,3,4};  
    int i = 0;  
    int sum = 0;  
    while(i<4) {  
        sum += array[i++];  
    }  
  
    return 0;  
}
```

- It is tempting to use incrementation and other shortcuts to cut down the number of lines in your code
- Avoid this, keep your code simple and easy to read
- In the example above, increment i in a separate line.

# Memory Access Violations

- One of the most common error sources for C is related to memory access violations
  - Even worse, they sometimes work depending on memory usage.
  - Cause of most; «But it was working on my computer» errors.
- Some reasons:
  - Fault in array index. Lower bound inclusive, upper bound exclusive
  - Not providing the correct size in malloc
  - Not using malloc but just defining a pointer variable
  - Early or multiple frees on a single heap allocation
- Finding them out is hard because the executable files just crash without leaving any clues



# Memory Access Violations

```
struct point_3d {  
    double x;  
    double y;  
    double z;  
};  
  
int main(void) {  
    struct point_3d* points;  
    int num_points = 10;  
    points = malloc(num_points);  
    int i;  
    for (i = 0; i < num_points; ++i) {  
        points[i].x=1;  
        points[i].y=2;  
        points[i].z=3;  
    }  
    for (i = 0; i < num_points; ++i) {  
        printf("%f, %f, %f \n", points[i].x, points[i].y, points[i].z);  
    }  
  
    return 0;  
}
```

A ticking time bomb!!!

Might work for a while.

Some variable's values can change for no reason.

Why?

# Solution: Memory Debuggers

- Use a memory debugger.
- Valgrind is a great tool.

```
$ valgrind ./Test
==3300== Memcheck, a memory error detector
==3300== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
al.
==3300== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==3300== Command: ./Test
==3300==
==3300== Invalid write of size 8
==3300==    at 0x10862B: main (in Test)
==3300==   Address 0x521c048 is 8 bytes inside a block of size 10
alloc'd
==3300==    at 0x4C2FB0F: malloc (in vgpreload_memcheck-amd64-linux.so)
==3300==    by 0x108604: main (in /Test)
==3300==
```

# Recursive Algorithms

- Divide & Conquer methodology; divide the problem into sub-problems with the same structure.
- Nature of recursion:
  - One or more simple cases of the problem (the base case or stopping cases) have a trivial non-recursive solution
  - The cases of the problem can be reduced to sub-problems that are closer to stopping cases.
  - Eventually the problem can be reduced to stopping cases only.

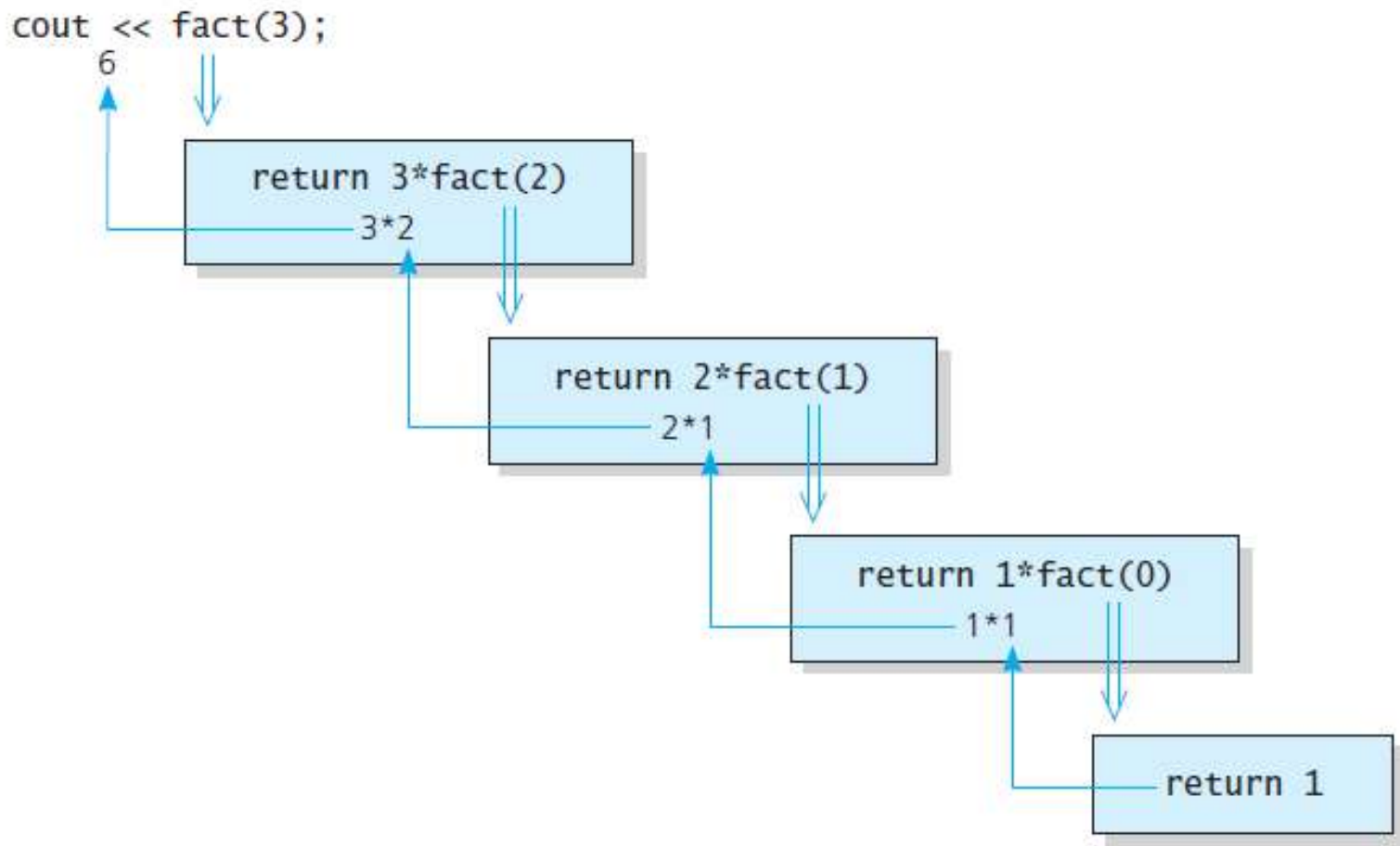
```
if (stopping case)
    solve it
else
    reduce the problem using recursion
```

# Example Problem: Factorial

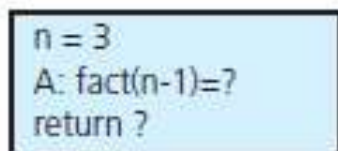
- $n! = n \times (n - 1) \dots 2 \times 1$
- $0! = 1$
- $$factorial(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times factorial(n - 1) & \text{if } n > 0 \end{cases}$$

```
int factorial(int n) {  
    if(n==0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

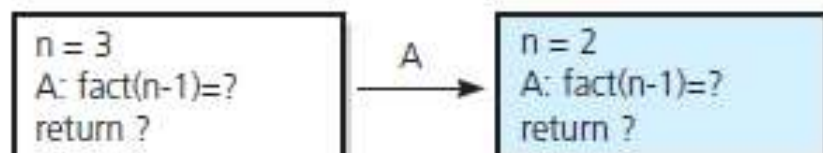
# Factorial Example



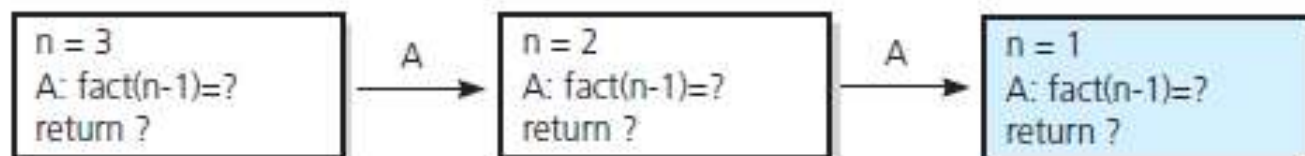
The initial call is made, and method **fact** begins execution:



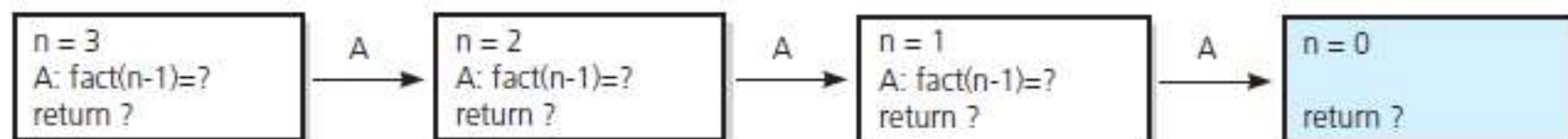
At point A a recursive call is made, and the new invocation of the method **fact** begins execution:



At point A a recursive call is made, and the new invocation of the method **fact** begins execution:

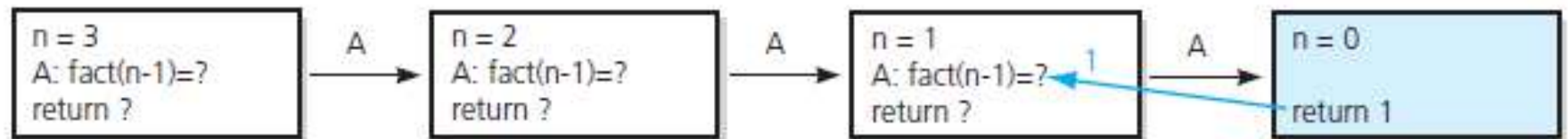


At point A a recursive call is made, and the new invocation of the method **fact** begins execution:

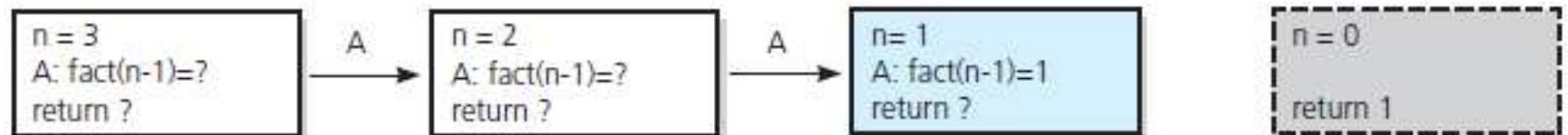


*(continues)*

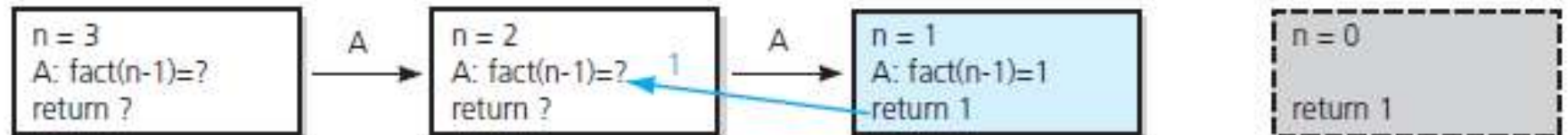
This is the base case, so this invocation of `fact` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:



The method value is returned to the calling box, which continues execution:



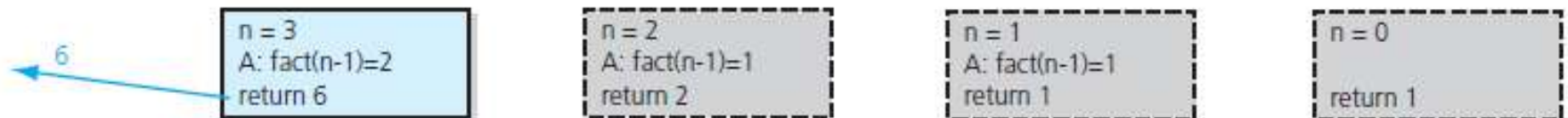
The current invocation of `fact` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes and returns a value to the caller:



The value 6 is returned to the initial call.



# Writing a String Backward

- Problem: Write the given string of characters in reverse order
- Recursive Solution:
  - Base case: Write the empty string backward
  - Recursive case: Print the last character, then solve the same problem for the remaining  $n-1$  characters

```
writeBackward(in s:string)
    if(the string is empty)
        Do nothing // base case
    else
        Write the last character of s
        writeBackward(s minus its last character)
```

# Write Backward

```
void writeBackward(char* s, int size) {  
    // Writes a character string backward.  
    // Precondition: The string s contains size characters,  
    // where size >= 0.  
    // Postcondition: s is written backward, but remains  
    // unchanged.  
    if (size > 0) {  
        printf("%c", s[size-1]);  
        // write the rest of the string backward  
        writeBackward(s, size-1); // Point A  
    }  
    // size == 0 is the base case - do nothing  
}
```

# Write Backward Trace

```
s = "cat"  
length = 3
```

Output line: **t**

Point A (`writeBackward(s)`) is reached, and the recursive call is made.

The new invocation begins execution:

```
s = "cat"  
length = 3
```

A

```
s = "ca"  
length = 2
```

Output line: **ta**

Point A is reached, and the recursive call is made.

The new invocation begins execution:

```
s = "cat"  
length = 3
```

A

```
s = "ca"  
length = 2
```

A

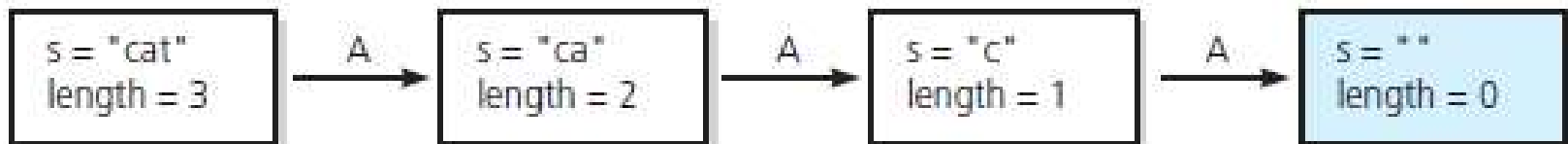
```
s = "c"  
length = 1
```

(continues)

Output line: `tac`

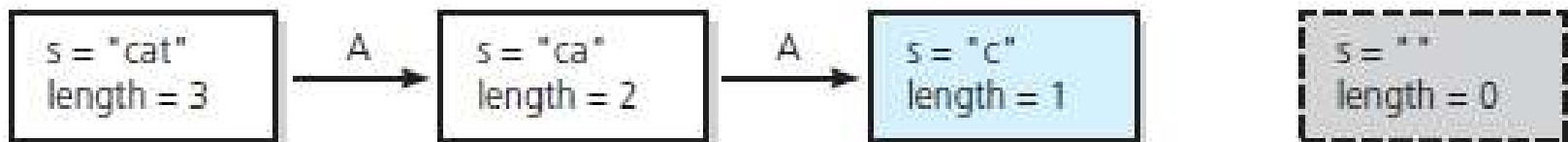
Point A is reached, and the recursive call is made.

The new invocation begins execution:



This is the base case, so this invocation completes.

Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the calling box, which continues execution:



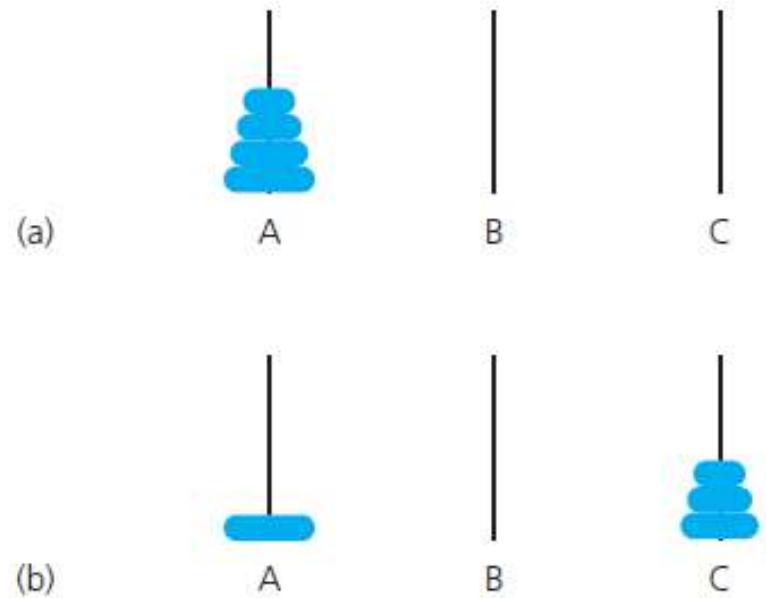
This invocation completes. Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the statement following the initial call.

# Towers of Hanoi

- There are  $n$  disks and three poles: A(source), B(destination), C(spare)
- Move all  $n$  disks from pole A to B.



# Towars of Hanoi

```
void solveTowers(int count, char source,
                 char destination, char spare) {
    if (count == 1) {
        printf("Move top disk from pole %c to pole %c nn",
               source, destination);
    } else {
        solveTowers(count-1, source, Spare , destination);
        solveTowers(1, source, destination, spare);
        solveTowers(count-1, spare,destination, source);
    }
}
```

# Towers of Hanoi

