

Spring 2018

[illegible]

TAs: Selim Yılmaz, Bahar Gezici, Cemil Zalluhoğlu

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

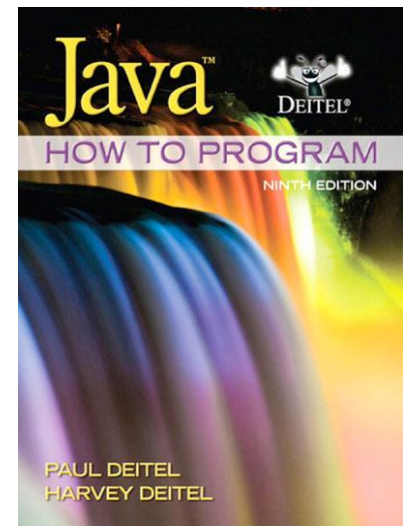
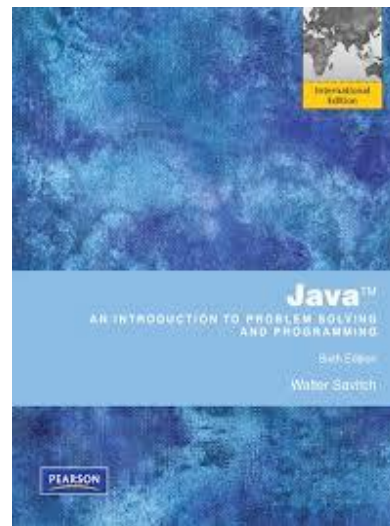
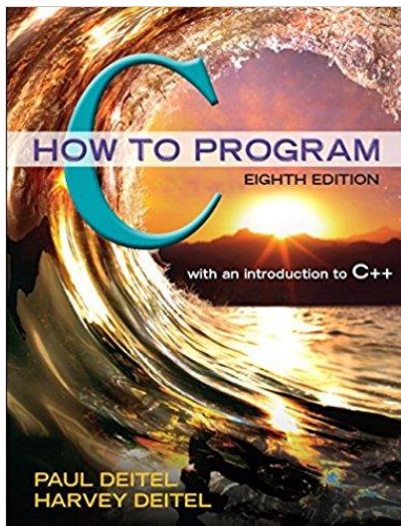
- We will start the term by recapping what you have learned about the C language in the previous term.
- We will also be covering some additional subjects that are;
 - Dynamic Memory Management
 - Structs
 - File I/O

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 Lab.:** The students will gain programming experience via a set of programming assignments and quizzes.
- **Requirements:** You must know basic programming (i.e. BBM101).

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**

- Assignments and quizzes.
- No recitations for this semester.

- **Communication**

- Announcements and course related discussions through

BBM 102: <http://piazza.com/hacettepe.edu.tr/spring2019/bbm102/home>

BBM 104: <http://piazza.com/hacettepe.edu.tr/spring2019/bbm104/home>

Course Work and Grading

- **Two Midterm Exams (25 + 25 = 50%)**
 - Closed book and notes
- **Final Exam (50%)**
 - Closed book
 - To be scheduled by the registrar.

Course Work and Grading

Week	Date	Title
1	27-Feb	C Review & Pointers
2	6-Mar	Arrays & Dynamic Memory
3	13-Mar	Structs & File Input/Output
4	20-Mar	Java Introduction
5	27-Mar	Classes & Objects, Encapsulation
6	3-Apr	Inheritance
7	10-Apr	Midterm Exam 1
8	17-Apr	Polymorphism
9	24-Apr	Abstract Classes & Interfaces
10	1-May	- May 1 st
11	8-May	Collections
12	15-May	Exceptions
13	22-May	Midterm Exam 2
14	29-May	Streams & Input Output

BBM 104 Introduction to Programming Laboratory

- **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.
- **Quizzes**
 - Five quizzes throughout the semester.
- **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!)

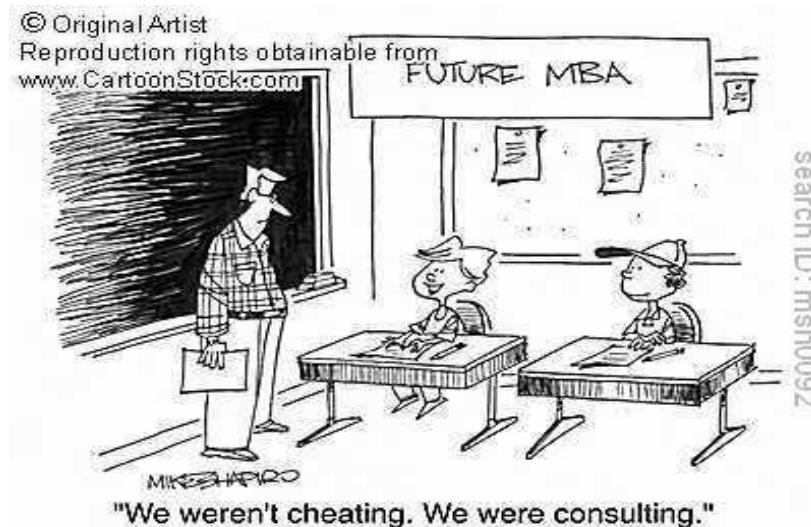
- **Lateness Penalties**

- 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

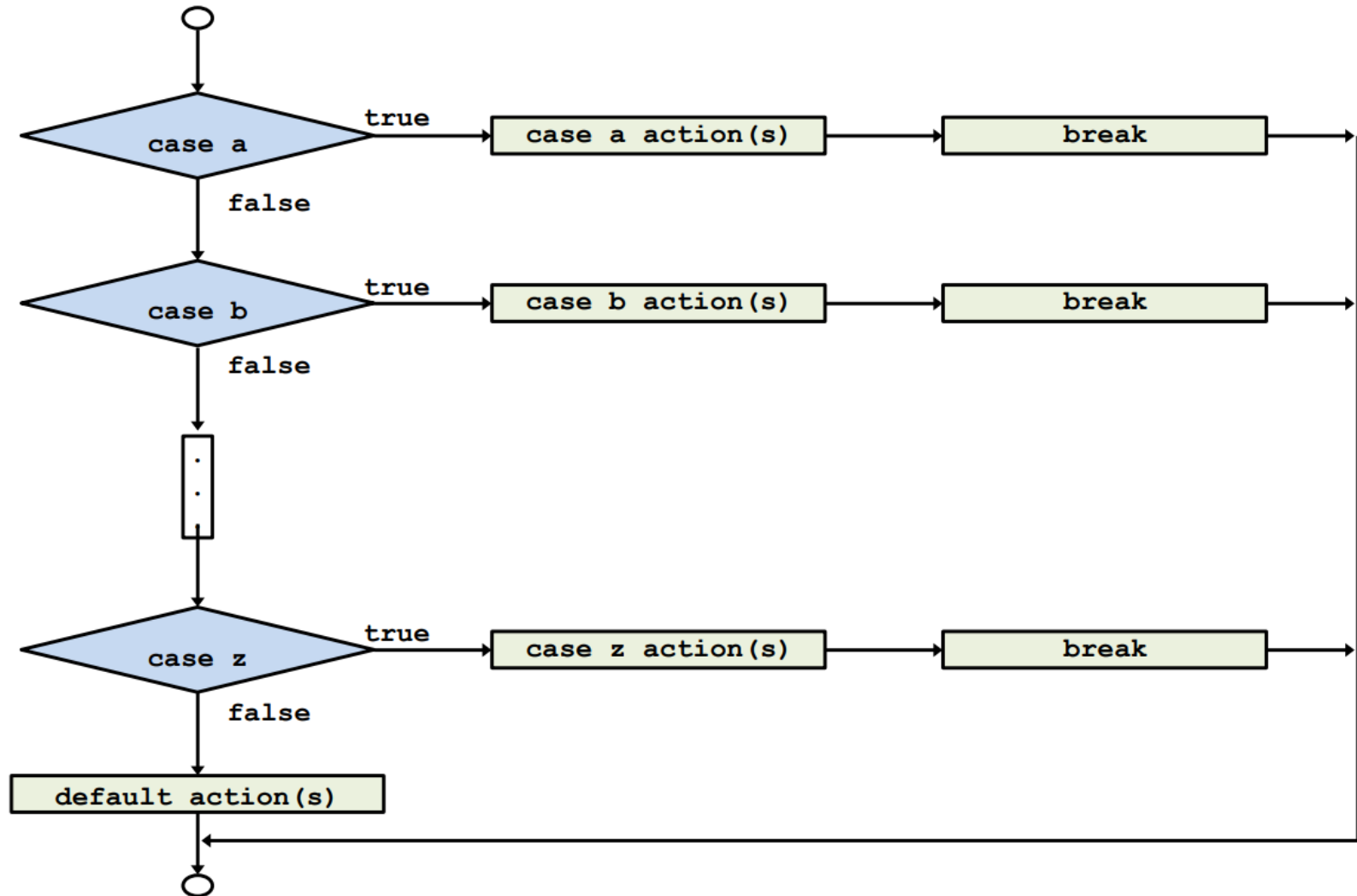
- **Switch Statement**

- 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



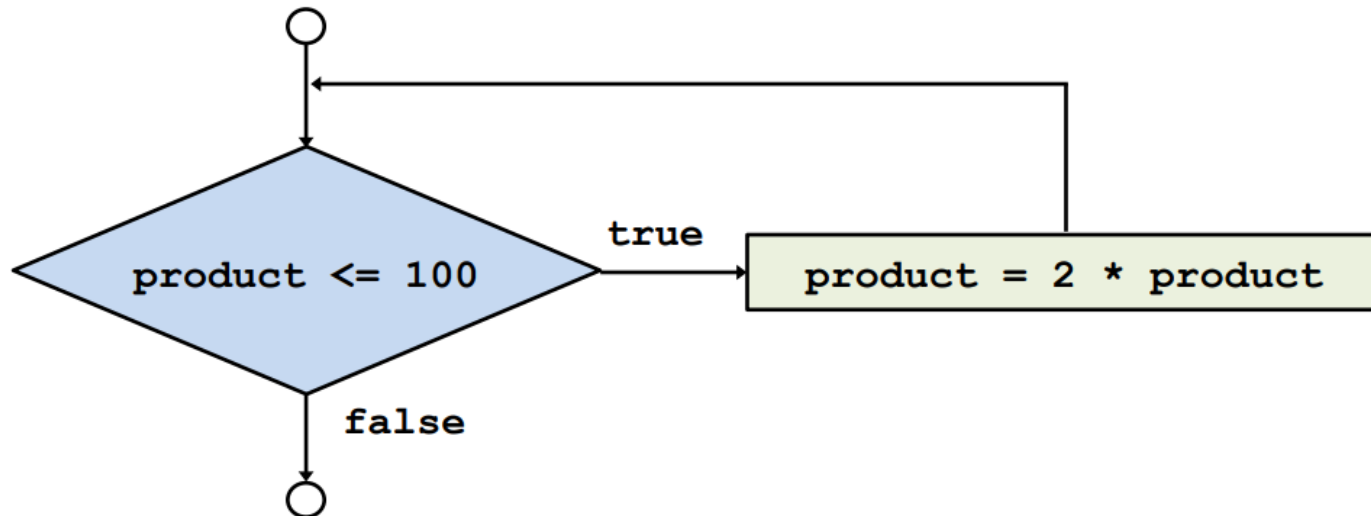
Control Structures

```
#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>
```

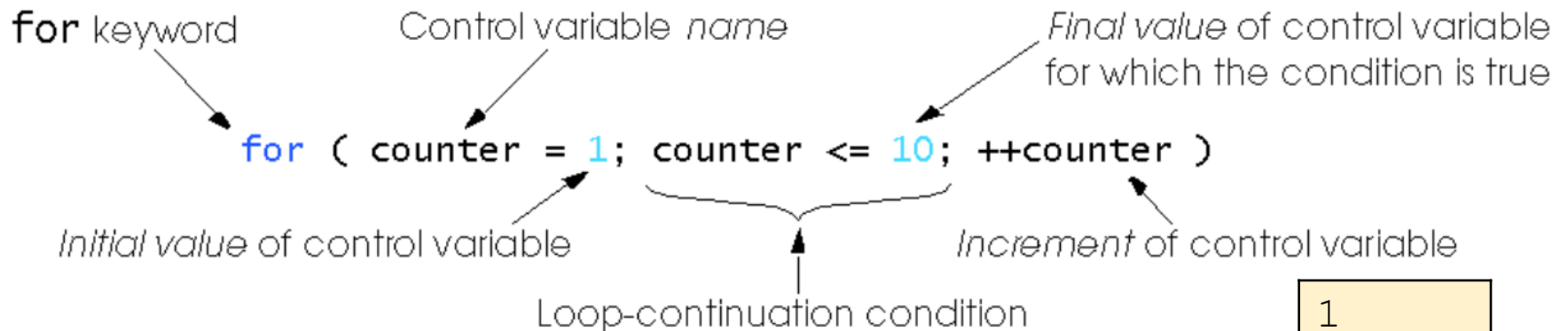
```
int main()  
{  
    int counter, grade, total, average;  
    /* initialization phase */  
    total = 0;  
    counter = 1;  
    /* processing phase */  
    while (counter <= 10) {  
        printf("Enter grade: ");  
        scanf("%d", &grade);  
        total = total + grade;  
        counter = 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

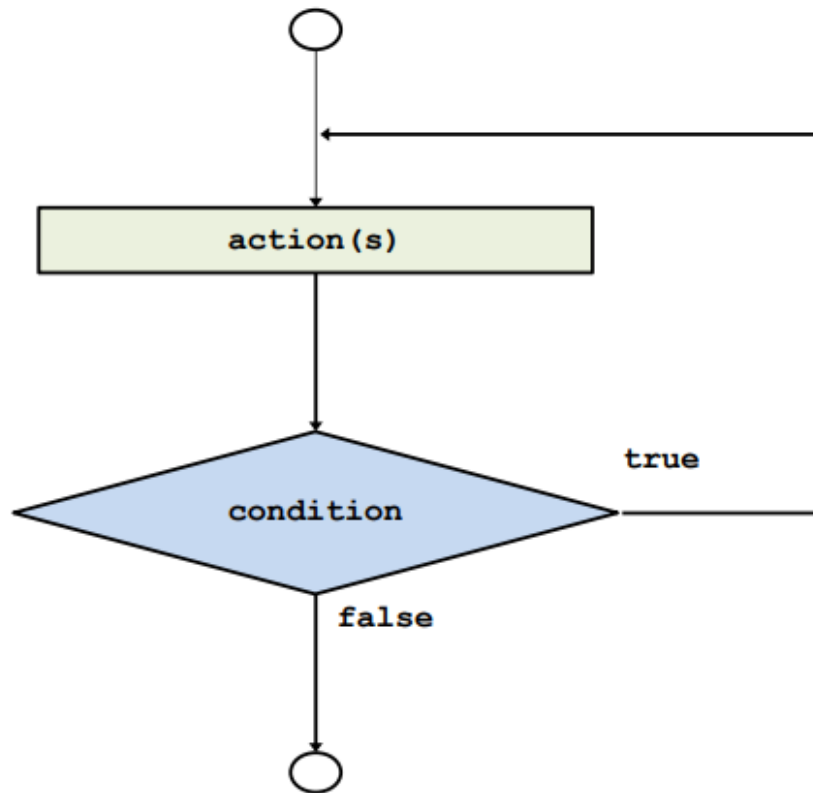
Control Structures

- 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;  
}
```

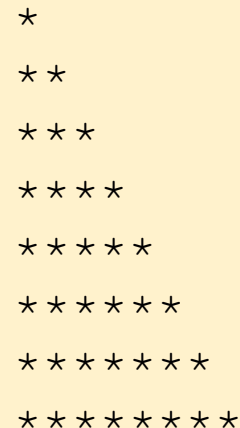
1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

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

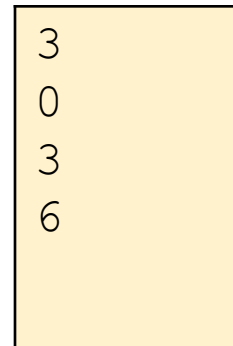
- Let's write a function that computes sum from 1 to n

```
#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

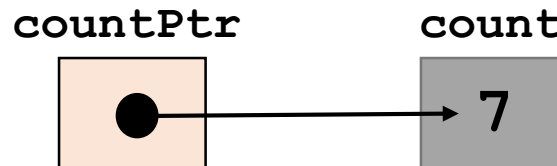
- Pointers allow you to reference a large data structure in a compact way.
- Pointers facilitate sharing data between different parts of a program.
 - Call-by-Reference
- **Dynamic memory allocation**: Pointers make it possible to reserve new memory during program execution.

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

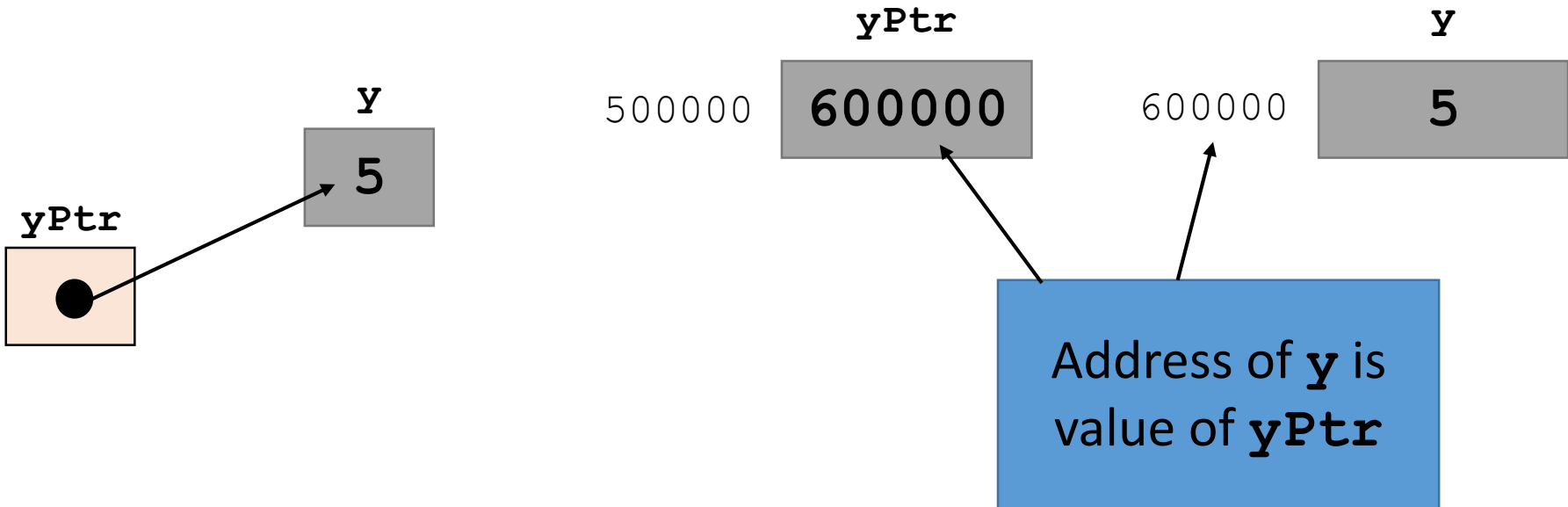


Pointers

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

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

- *yPtr points to y*



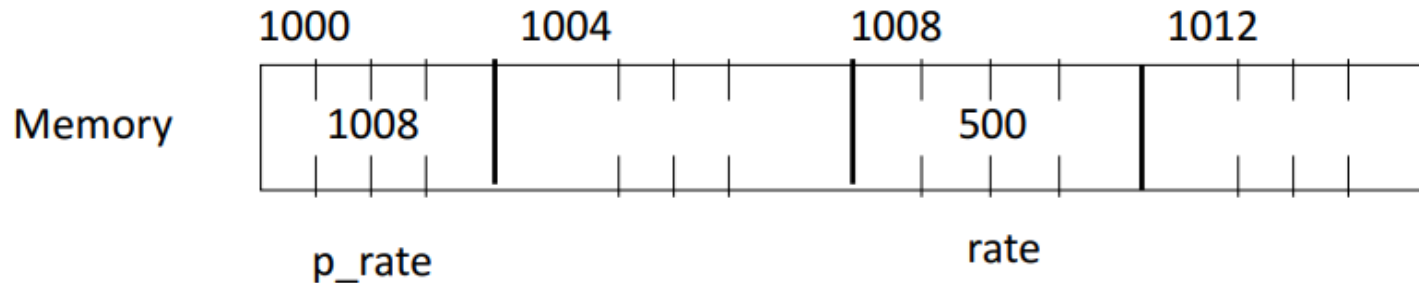
Pointers

- `*` (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;
```



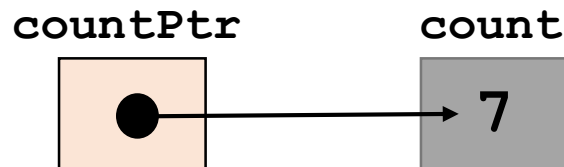
All equivalent

Pointers

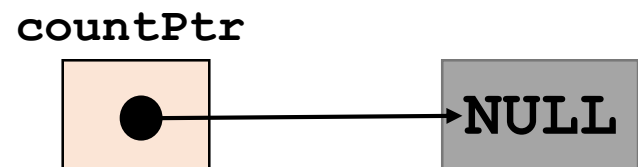
- The **NULL** Value

- 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;
}
```

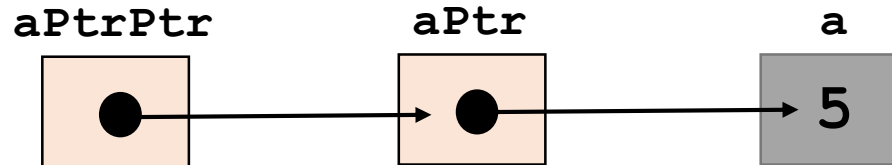
```
a: 3
b: 5
In swap
a: 5
b: 3
After swap
a: 5
b: 3
```



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 **bPtrPtr = &bPtr;
```



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 = &a, *bPtr = &b;  
    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