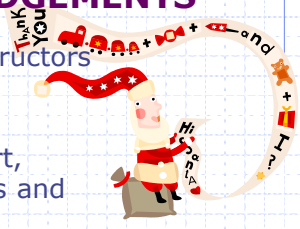
ESC101
Introduction to Computing

WELCOME

Amey Karkare
Dept of CSE
IIT Kanpur

ACKNOWLEDGEMENTS



- All previous instructors of Esc101 at IIT Kanpur.
- MS Office clip art, various websites and images
  - The images/contents are used for teaching purpose and for fun. The copyright remains with the original creator. If you suspect a copyright violation, bring it to my notice and I will remove that image/content.

The Course

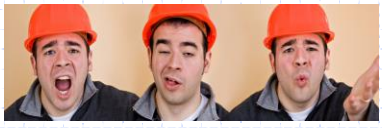
- The course teaches you how to solve problems using the computer.
- No prior exposure to programming is needed.

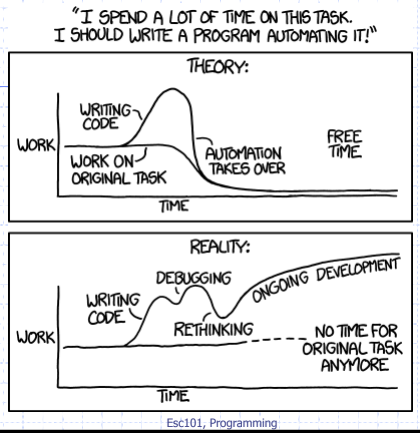
What is a Computer?

Almost all electronic gadgets today are Computers. They are everywhere!

Why am I doing this course?

- Every discipline uses computing: All branches of engineering, sciences, design and arts.
- Understand how computers work
- Write your own programs
  - Automate boring repetitive stuff!

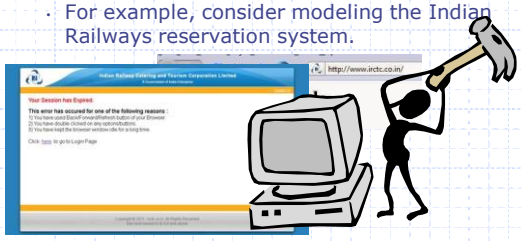




Source: http://xkcd.com/1319

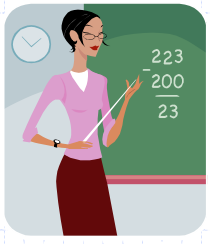
Process of Programming: Step 1

- Define and model the problem. In real-life this is important and complicated.
  - For example, consider modeling the Indian Railways reservation system.



Process of Programming

- In this course, all problems will be defined precisely and will be simple

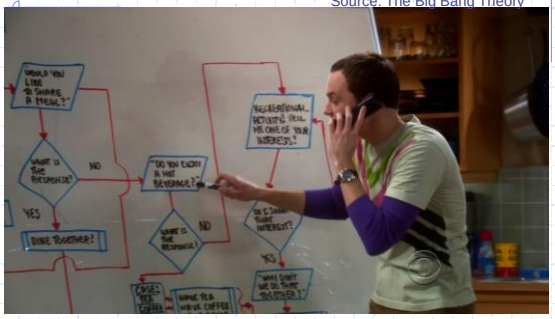


Process of Programming: Step 2

- Obtain a logical solution to your problem.
- A logical solution is a finite and clear step-by-step procedure to solve your problem.
- Also called an Algorithm.
  - We can visualize this using a Flowchart.
  - Very important step in the programming process.

Friendship Algorithm/Flowchart

Source: The Big Bang Theory



Algorithms in real-life

- Any step-by-step guide. e.g. Assembly instructions for a make-it-yourself kit.

http://www.gocomics.com/calvinandhobbes/2009/06/02

GCD

- An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.
- A naïve solution – Described *informally* as follows.
  1. Take the smaller number n.
  2. For each number k, n ≥k≥1, in descending order, do the following.
     a) If k divides m and n, then k is the gcd of m and n

GCD

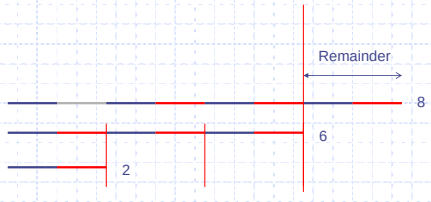
- This will compute gcd correctly, but is VERY slow (think about large numbers m and n=m-1).
- There is a faster way…

GCD Algorithm - Intuition

To find gcd of 8 and 6.

- Consider rods of length 8 and 6.
- Measure the longer with the shorter.
- Take the remainder **if any**.
- **Repeat** the process until the longer can be exactly measured as an integer multiple of the shorter.

GCD Algorithm - Intuition



Gcd(8, 6) = 2.

GCD Algorithm - Intuition

102

102 mod 21 = 18

21

21 mod 18 = 3

18

3

Gcd (102, 21) = 3

Euclid's method for gcd

- Suppose a > b. Then the gcd of a and b is the same as the gcd of b and the remainder of a when divided by b.

gcd(a,b) = gcd (b, a % b)

Proof:

Exercise

GCD Algorithm

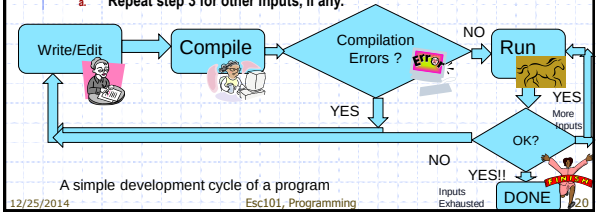
Data: Integers m and n
if n > m then Interchange m and n;
while n≠0 do
    g ← m%n;
    m ← n;
    n ← g;
end
return m;

Overview of Programming

Using C

The Programming Cycle

1. Write your program or edit (i.e., change or modify) your program.
2. Compile your program. If compilation fails, return to editing step.
3. Run your program on an input. If output is not correct, return to editing step.
   a. Repeat step 3 for other inputs, if any.



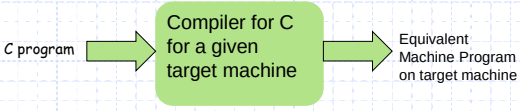
A simple development cycle of a program

IDE for Edit-Compile-Run cycle

- In this course, you will be using an Integrated Development Environment (IDE). IDE will be available through your browser.
- First login to the system.
- Type in your program in the editor of the IDE.
- Use the compile button to compile.
- Run button to run.
  - The labs in the first week will introduce you to the system in more detail.

Why program in high level languages like C

- Writing programs in machine language is long, tedious and error-prone.
- They are also not portable—meaning program written for one machine may not work on another machine.
- Compilers work as a bridge.
- Take as input a C program and produce an equivalent machine program.



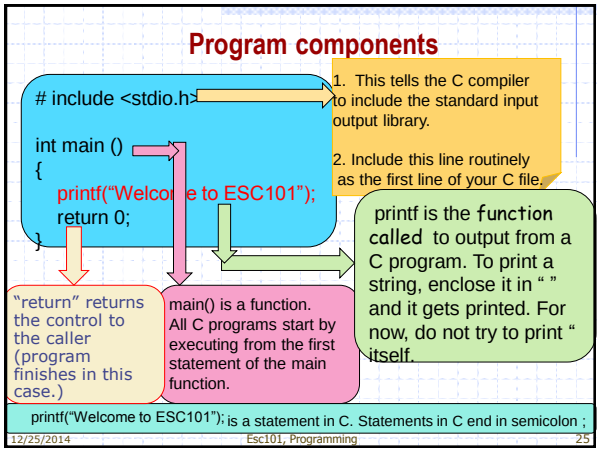
Simple! Program

- Today we will see some of the simplest C programs.

```
# include <stdio.h>
int main ()  {
   printf("Welcome to ESC101");
   return 0;
}
```

The program prints the message "Welcome to ESC101"

Program components

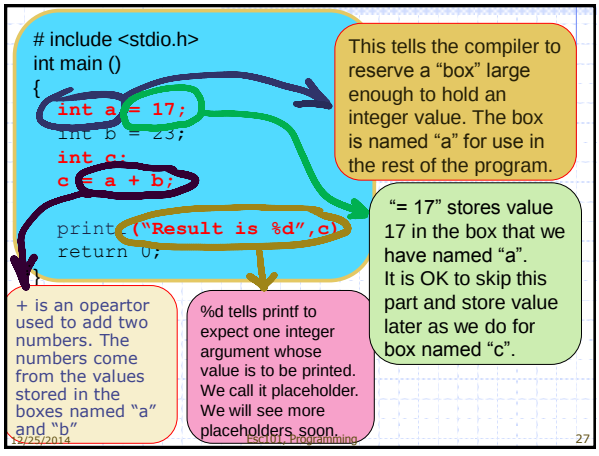


Another Simple Program

- Program to add two integers (17 and 23).

```
# include <stdio.h>
int main ()  {
   int a = 17;
   int b = 23;
   int c;
   c = a + b;
   printf("Result is %d", c);
   return 0;
}
```

The program prints the message "Result is 40"



Another Simple Program

- A smaller program to add two integers (17 and 23).

```
# include <stdio.h>
int main ()  {
  printf("Result is %d", 17+23);
  return 0;
}
```

The program prints the message "Result is 40"

In this case + is operating directly on two integer **constants**.

ESC101: Introduction to Computing

Overview of C Programming

Types

- **Type:**
  - A set of values
  - A set of operations on these values
- We have been using types
  - Natural numbers
    - 1, 2, 3, … values
    - +, - , *, >, <, … operations
  - Complex numbers
    - 5 + 3i, 7 + 2i, …
    - +, - , *, /, conjugate, …
    - **NO** >, < operations

Data Types in C

- **int**
  - Bounded integers, e.g. 732 or -5
- **float**
  - Real numbers, e.g. 3.14 or 2.0
- **double**
  - Real numbers with more precision
- **char**
  - Single character, e.g. a or C or 6 or $

Notes on Types

- Characters are written with ' ' (quotes)
  - 'a', 'A', '6', '$'
- Case sensitive
  - 'a' is not same as 'A'
- Types distinguish similar looking values
  - Integer 6 is not same as character '6'

More Notes on Types

- Integers (int) are bounded
  - Max value: INT_MAX
  - Min value: INT_MIN
    - These values are system specific
    - -2147483648 … 2147483647 on my machine
- So are other data types
  - Even some simple real values can not be represented by float and double
- Can surprise you sometime

```
#include <limits.h>
#include <stdio.h>
int main() {
    printf("Min=%d, Max=%d",
           INT_MIN, INT_MAX);
    return 0;
}
```

OUTPUT: Min=-2147483648, Max=2147483647

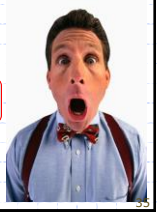
limits.h contains the definitions of INT_MAX and INT_MIN

1. A statement can span multiple lines.
2. printf can use multiple % placeholders.


```
#include <stdio.h>
int main() {
    float y = 100000009.0;
    printf("Value of y is %f", y);
    return 0;
}
```

%f is the placeholder for float.

OUTPUT: Value of y is 100000008.0



Function main

- The point at which C program begins its execution
- Every complete C program must have exactly one main

int main () { … }

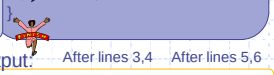
- Returns an int to its caller (Operating System)
  - Return value is generally used to distinguish successful execution (0) from an unsuccessful execution (non 0)

Function main

- Arguments: none ()
  - At least for now
- Body: C statements enclosed inside { and } (to solve the problem in hand)

Tracing the Execution

```
Line No.
1  # include <stdio.h>
2  int main()
3  {
4      printf("Welcome to ");
5      printf("C Programming");
6      return 0;
   }
```



Output: After lines 3,4 After lines 5,6

Welcome to C Programming

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.
- Let us run the program, one step at a time.
- Program terminates gracefully when main ``returns''.

Variables

- A name associated with memory cells (box-es) that store data
- Type of variable determines the size of the box.

int m = 64; 64

char c = 'X'; X

float f = 3.1416; 3.1416

- Variables can change their value during program

f = 2.7183; 2.7183

Variable: Box and Value

- Another analogy is that of Envelope and Letter
- Envelope must be big enough to hold the letter!

Variable Declaration

- To communicate to compiler the names and types of the variables used by the program
  - Type tells size of the box to store value
  - Variable must be declared before used
  - Optionally, declaration can be combined with definition (initialization)

```
int count;
int min = 5;
```

Declaration without initialization

Declaration with initialization

Identifiers

- Names given to different objects
  - Variable, Function etc.
- Consists of letters, digits and underscore (_) symbol
  - Must start with a letter or _

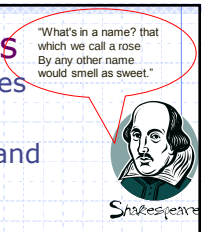
i, count, Lab5, max_Profit, _left, fUnNy

~~5j~~, ~~min Profit~~, ~~lab.7~~

Identifiers

- Certain names are reserved in C
  - Have special meaning
  - Can not be used as identifier
  - Some reserved words: int, float, void, break, switch, const, if, else, …
- Standard library names should be avoided
  - printf, scanf, strcmp, …
- Case sensitive
  - Esc101 ≠ esc101 ≠ ESC101

Choosing Identifiers



- Choose meaningful names
  - count vs c vs tmp1
- Should be easy to read and understand
  - count vs c_o_u_n_t
- Shorten only when no loss of meaning
  - Max vs Maximum
- Avoid unnecessary long names
  - a_loop_counter vs counter vs i

Comments

- Supplementary information in programs to make understanding easier
  - Only for Humans!
  - Ignored by compilers

Comments in C

- Anything written between /* and */ is considered a comment.

```
diameter = 2*radius; /* diameter of a circle */
```

- Comments can not be nested.

```
/* I am /* a comment */ but I am not */
```

First */ ends the effect of all unmatched start-of-comments (/*).

Comments in C

- Anything written after // up to the end of that line

```
diameter = 2*radius; // diameter of a circle
area = pi*radius*radius; // and its area
```

- Not all C compilers support this style of comments.
  - Our lab compiler does support it.

Assignment Statement

- A simple assignment statement

  Variable = Expression;

- Computes the value of the expression on the right hand side expression (RHS), and stores it in the "box" of left hand side (LHS) variable
- = is known as the assignment operator.

Assignment Statement

- Examples

```
x = 10;
ch = 'c';
disc_2 = b*b - 4*a*c;
count = count + 1;
```

If count was 5 before the assignment, it will become 6 after the assignment.

- Evaluation of an assignment stmt:
  - Expression on the RHS of the = operator is first evaluated.
  - Value of the expression is assigned to the variable on the LHS.

Input/Output

- Input: receive data from external sources (keyboard, mouse, sensors)
- Output: produce data (results of computations) (to monitor, printer, projector, …)

Input/Output

- printf function is used to display results to the user. (output)
- scanf function is used to read data from the user. (input)
- Both of these are provided as library functions.
  - #include <stdio.h> tells compiler that these (and some other) functions may be used by the programmer.

printf

string to be displayed, with placeholders

list of expressions (separated by comma)

\n is the newline character.

```
printf("%d kms is equal to %f miles.\n", km, mi);
```

The string contains placeholders (%d and %f). Exactly one for each expression in the list of expressions.

Placeholder and the corresponding expr must have compatible type.

While displaying the string, the placeholders are replaced with the value of the corresponding expression: first placeholder by value of first expression, second placeholder by value of second expression, and so on.

Using format string in printf

```
printf("a= %d, b= %d, hypotenuse squared =%d", a,b, csquare);
```

format string marker

Argument marker

1. format marker marks the first character of the format string.
2. Argument marker marks the first argument after the format string.
3. Print the character marked by format marker and advance format marker one character at a time until a %d is met or the format string finishes.
4. If format string finishes, we TERMINATE.
5. If control string marker reads %d, then printf takes the value of the variable at the argument marker and prints it as a decimal integer.
6. Advance argument marker; advance format marker past %d. Go to step 3.

a= 3, b=4, hypotenuse squared = 25

scanf

Similar to printf: string with placeholders, followed by list of variables to read

& is the *addressof* operator. To Be Covered Later.

```
scanf("%d", &km);
```

Note the & before the variable name. DO NOT FORGET IT.

- String in " " contains only the placeholders corresponding to the list of variables after it.
- Best to use one scanf statement at a time to input value into one variable.

Some Placeholders

Placeholder	Type
%d	int
%f	float
%lf	double
%c	char
%%	literal percent sign (%)

If placeholder and expression/variable type do not match, you may get unexpected results.

Good and Not so good printf's

```
# include <stdio.h>
int main() {
    float x;
    x=5.67123;
    printf("%f", x);
    return 0;
}
```

Compiles ok

Output: 5.671230

```
# include <stdio.h>
int main() {
    float x;
    x=5.67123;
    printf("%d", x);
    return 0;
}
```

Compiles ok

-14227741

Printing a float using %d option is undefined. Result is machine dependent and can be unexpected. AVOID!

C often does not give compilation errors even when operations are undefined. But output may be unexpected!

ESC101: Introduction to Computing

Operators and Expressions

Binary Operations

- Operate on int, float, double (and char)

Op	Meaning	Example	Remarks
+	Addition	9+2 is 11	
		9.1+2.0 is 11.1	
-	Subtraction	9-2 is 7	
		9.1-2.0 is 7.1	
*	Multiplication	9*2 is 18	
		9.1*2.0 is 18.2	
/	Division	9/2 is 4	Integer div.
		9.1/2.0 is 4.55	Real div.
%	Remainder	9%2 is 1	Only for int

Unary Operators

- Operators that take only one argument (or operand)
  - -5
  - +3.0123
  - -b
- Observe that + and – have two purposes
  - Meaning depends on context
  - This is called overloading

The / operator

- When both (left and right) operand of / are of type int
  - The result is the integral part of the real division
  - The result is of type int
- Examples

  9/4 is 2

  1/2 is 0

The / operator

- When at least one (left or right or both) operands of / are of type float
  - The result is the real division
  - The result is of type float
- Examples

  9/4.0 is 2.25

  1.0/2 is 0.5,

  so is 1/2.0

  and 1.0/2.0

The % operator

- The remainder operator % returns the integer remainder of the result of dividing its first operand by its second.
- Both operands must be integers.
- Defined only for integers

  4%2 is 0

  31%4 is 3

Divison(/) and Remainder(%)

- Second argument can not be 0
  - Run time error
- For integers a and b (b≠0), / and % have the following relation

  a = (a/b)*b + (a%b)
- If a or b or both are negative, the result of / and % is system dependent.

Type of Arithmetic Expr

- Type of (result of) arithmetic expr depends on its arguments
- For binary operator
  - If both operands are int, the result is int
  - If one or both operand are float, the result is float
- For unary operator
  - Type of result is same as operand type

Operator Precedence

- More than one operator in an expression
  - Evaluation is based on precedence
- Parenthesis (…) have the highest precedence
- Precedence order for some common operators coming next

Operator Precedence

HIGH

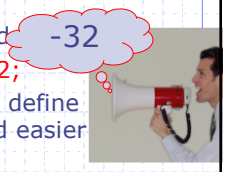
INCREASING

LOW

Operators	Description	Associativity
(unary) + -	Unary plus/minus	Right to left
* / %	Multiply, divide, remainder	Left to right
+ -	Add, subtract	Left to right
< > >= <=	less, greater comparison	Left to right
== !=	Equal, not equal	Left to right
=	Assignment	Right to left

Operator Precedence

- What is the value assigned?
  x = -5*4/2*3+-1*2;



- Always use parenthesis to define precedence. It is safer and easier to read.
- Avoid relying on operator precedence. Can give absurd results if not used correctly.
- Consult any textbook to know more about precedence.

Type Conversion (Type casting)

- Converting values of one type to another.
  - Example: int to float and float to int (also applies to other types)
- Can be implicit or explicit

```
int k =5;
float x = k;          // implicit conversion to 5.0
float y = k/10;       // y is assigned 0.0
float z = ((float) k)/10; // Explicit conversion
                      // z is assigned 0.5
```

Loss of Information!

- Type conversion may result in loss information.
- Larger sized type (e.g. float) converted to smaller sized type (e.g. int) is undefined.
- Smaller sized type (e.g. int) converted to larger sized type (e.g. float) may also give unexpected results. Take care!

float to int: type conversion (result ok)

```
#include<stdio.h>
int main() {
    float x;  int y;      /* define two variables */
    x = 5.67;
    y = (int) x;          /* convert float to int    */
    printf("%d", y);
    return 0;
}
```

Output : 5

```
float x;
…
(int) x;
```

converts the real value stored in x into an integer. Can be used anywhere an int can.

float to int type conversion (not ok!)

- float is a larger box, int is a smaller box. Assigning a float to an int may lead to loss of information and unexpected values.

The floating point number 1E50 is too large to fit in an integer box.

```
# include <stdio.h>
int main() {
    float x; int y;
    x  = 1.0E50; // 10^50
    y = (int) x;
    printf("%d", y);
    return 0;
```

Output: -2184748364

Careful when converting from a 'larger' type to 'smaller' type. Undefined.

int to float (take care!)

```
# include <stdio.h>
int main() {
    int y;
    y = 1000001;
    printf("%f", (float) y);
    return 0; }
```

Output: 1000001.000000

Result is correct

```
# include <stdio.h>
int main() {
    int y;
    y = 10000009;
    printf("%f\n", (float) y);
    printf(" %d", y);
    return 0; }
```

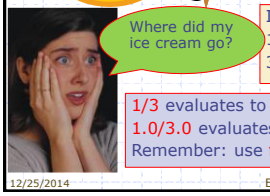
Output: 10000008.000000 10000009

Result is not correct. Information is lost.

Program Example

Volume of a cone = $\frac{1}{3} \times \pi \times radius^2 \times height$

```
float r,h;
scanf("%f", &r);
scanf("%f", &h);
printf("Volume is %f\n",
    1/3*3.14*r*r*h);
```



Input:
10
3

Output?
0

1/3 evaluates to 0
1.0/3.0 evaluates to 0.3333…
Remember: use floats for real division

temp_conversion.c

```
# include <stdio.h>
int main() {
    float Centigrades;
    float F;
    Centigrades=50;
    F = ((9*Centigrades)/5) + 32;
    printf("The temperature ");
    printf( "%f", Centigrades);
    printf("Celsius equals");
    printf("%f", F);
    printf("Fahrenheit");
    return 0;
}
```

- Microprocessors represent real numbers using ***finite precision***, i.e., using *limited number of digits after decimal point.*
- Typically uses scientific notation: 12.3456789 represented as 1.23456789E+1. Bit more later.

"%f" signifes that the corresponding variable is to be printed as a real number in decimal notation.

C 50.000000 122.000000 F

Compile and Run

```
The temperature 50.000000 Celsius equals 122.000000 Fahrenheit%
```

char data type in C

- Basic facts
  - Characters in C are encoded as numbers using the ASCII encoding
  - ASCII : American Standard Code for Information Interchange
- Encodings of some of the common characters:
  - 'A' is 65, 'B' is 66, 'C' is 67 ... 'Z' is 90
  - 'a' is 97, 'b' is 98 ... 'z' is 122
  - '0' is 48, '1' is 49 ... '9' is 57

char data type in C

- Range: 0 to 255
- You should NOT try to remember ASCII values
  - Encoding/programming languages provide alternatives to use them
- C treats characters as integers corresponding to their ASCII value.
- While displaying with %c placeholder, the ASCII value is converted to its corresponding character.

char data type in C

- Interconversion between character and integer datatypes can be exploited to write programs.

```
printf("%d\n", 'A');
printf("%d\n", '7');
printf("%c\n", 70);
printf("%c\n", 321);
```

Output:
65
55
F

321 is outside range! What do you think will be the output of printf("%c\n",321); Try it out

char data type in C

- Interconversion between character and integer datatypes can be exploited to write programs.

```
printf("%c\n", 'C'+5);
printf("%c\n", 'D' - 'A' + 'a' );
printf("%d\n", '3' + 2);
```

Output:
H
d
53

- Placeholder determines the output.
- Use with caution.
- Avoid arithmetic operation such as * and / on characters.
- Common Mistake: Incorrect data type of placeholder.

Formatting Output of a Program (int)

- When displaying an int value, place a number between the % and d which will specify the number of columns to use for displaying the int value (such as %5d).

```
int x = 2345, y=123;
printf("%d\n",x); //Usual

printf("%6d\n",x); //Display using 6 columns

printf("%6d\n",y); //Note: Right aligned

printf("%2d\n",x); //Less columns, same as %d
```

Output

```
2345
  2345
   123
2345
```


Formatting Output of a Program (float)

- Format placeholder id is %n.mf where
  - n is the total field width (both before and after the decimal point), and
  - m is the number of digits to be displayed after the decimal

```
float pi = 3.141592;
printf("%f\n",pi); //Usual

printf("%6.2f\n", pi); //2 decimal

printf("%.4f\n",pi); //4 decimal
                     // Note rounding off!
```

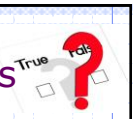
Output

```
3.141592
  3.14
3.1416
```


ESC101: Introduction to Computing

Conditional Expressions

Conditional Expressions



- An expression that evaluates to either true or false.
  - Often known as Boolean expression.
- C **does not** have a separate Boolean data type
  - Value 0 is treated as false.
  - Non-zero values are treated as true.

Conditional Expressions

- If an expression evaluates to true, we get a value 1
  - Think of 1 as *default* true value
- If an expression evaluates to false, we get a value 0

True	False
☑	☐

Relational Operators

- Compare two quantities

Operator	Function
>	Strictly greater than
>=	Greater than or equal to
<	Strictly less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

- Work on int, char, float, double…

Examples

Rel. Expr.	Result	Remark
3>2	1	
3>3	0	
'z' > 'a'	1	ASCII values used for char
2 == 3	0	
'A' <= 65	1	'A' has ASCII value 65
'A' == 'a'	0	Different ASCII values
('a' – 32) == 'A'	1	
5 != 10	1	
1.0 == 1	AVOID	May give unexpected result due to approximation

Avoid mixing int and float values while comparing. Comparison with floats is not exact!

Logical Operators

Logical Op	Function	Allowed Types
&&	Logical AND	char, int, float, double
||	Logical OR	char, int, float, double
!	Logical NOT	char, int, float, double

Remember
- value 0 represents false.
- any other value represents true.

Truth Tables

E1	E2	E1 && E2	E1 || E2
0	0	0	0
0	Non-0	0	1
Non-0	0	0	1
Non-0	Non-0	1	1

E	!E
0	1
Non-0	0

Examples

Expr	Result	Remark
2 && 3	1	
2 || 0	1	
'A' && '0'	1	ASCII value of '0' is non-0
'A' && 0	0	
'A' && 'b'	1	
! 0	1	
! 10	0	

Conditional Statements

- In daily routine
  - If Ishant Sharma takes 7 wickets, India will win.
  - If there is a quiz tomorrow, I will first study and then sleep. Otherwise I will sleep now.
  - If cost of tomatoes is more than 100Rs/kg, I will not buy tomatoes.
  - If I have 5 Rs, I will eat biscuits. If I have 20 Rs, I will eat maggi. If I have 500 Rs, I will order pizza.

Conditional statements in C

- 3 types of conditional statements in C
  - if (cond) action
    else some-other-action
  - if (cond) action
  - switch-case
- Each action is a sequence of one or more statements!

Statements and Blocks

- An expression such as x=0 or printf(…) becomes a statement when it is followed by a semi-colon, as in

```
x = 0;
printf( … );
5+2;
```

- Braces { and } are used to group variable declarations and statements together into a compound statement or a block
  - Syntactically equivalent to a single statement.
  - Can use it anywhere a single statement can be used

Statements and Blocks

```
{
    int x; float y; /* 2 statements
*/
    x = 10;
    printf("x = %d\n", x);
}
```

A single block

if-else statement

- Read two integers and print their minimum.

```
# include <stdio.h>
int main() {
    int x, y;
    scanf("%d%d", &x,&y);
    if (x < y)
        printf("%d", x);
    else
        printf("%d", y);
    return 0;
}
```

1. Checks if x is less than y.
2. If so, print x
3. Otherwise, print y.

Tracing Execution of if-else

```
# include <stdio.h>
int main() {
    int x; int y;
    scanf("%d%d", &x,&y);
    if (x < y) {
        printf("%d\n",x);
    }
    else {   printf("%d\n",y);}
    return 0;
}
```

6 < 10 so the if-branch is taken

Run the program

Input: 6 10

x: 6

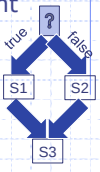
y: 10

Output: 6

if-else statement

- General form of the if-else statement

```
if (expression)
      statement S1
else
      statement S2
statement S3
```

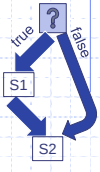


- Execution of if-else statement
  - First the expression is evaluated.
  - If it evaluates to a non-zero value, then S1 is executed and then control (program counter) moves to S3.
  - If expression evaluates to 0, then S2 is executed and then control moves to S3.
  - S1/S2 can be block of statements!

if statement (no else!)

- General form of the if statement

```
if (expression)
      statement S1
statement S2
```



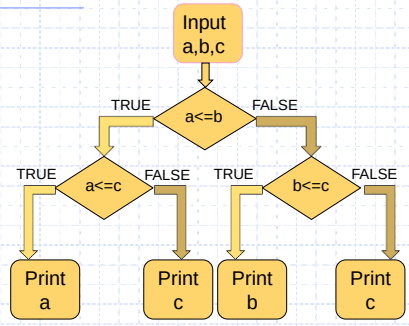
- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a non-zero value, then S1 is executed and then control (program counter) moves to the statement S2.
  - If expression evaluates to 0, then S2 is executed.

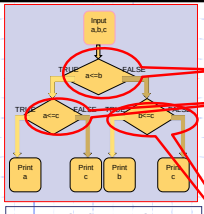
Example

- Problem: Input a, b, c are real positive numbers such that c is the largest of these numbers. Print ACUTE if the triangle formed by a, b, c is an acute angled triangle and print NOT ACUTE otherwise.

```
int main() {
    float a; float  b; float  c;
    scanf("%f%f%f", &a,&b,&c);        /* input a,b,c */

    if ( (a*a + b*b) >  (c*c) )  { /* expression*/
       printf("ACUTE");
    }
    else {
       printf("NOT ACUTE");
    }
    return 0;
}
```

Finding min of 3 numbers





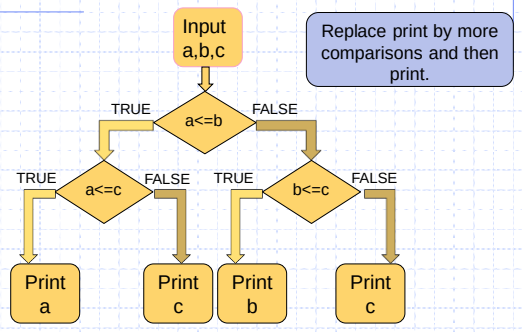
- Each branch translates to an if-else statement
- Hierarchical branches result in nested if-s

```
int a,b,c;
scanf("%d%d%d",&a,&b,&c);
if (a <= b) {
      if (a <= c) {
            printf("min = %d",a);
      }
      else {
            printf("min = %d", c);
      }
}
else {
      if (b <= c)  {
            printf("min = %d", b);
      }
      else {
            printf("min =%d", c);
      }
}
```

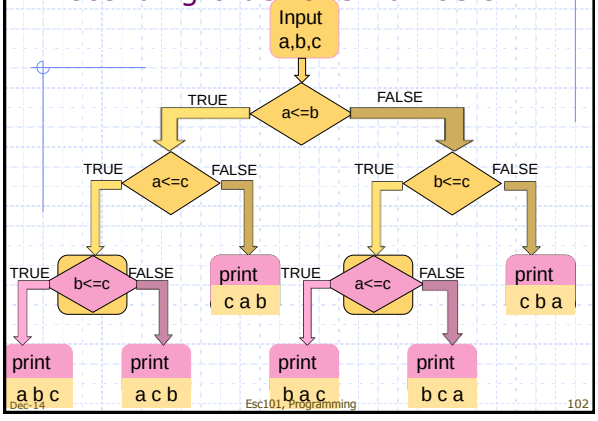
More Conditionals

- Sorting a sequence of numbers (i.e., arranging the numbers in ascending or descending order) is a basic primitive.
- Problem: read three numbers into a, b and c and print them in ascending order.
  - Start with the flowchart for finding minimum of three numbers and add one more level of conditional check.
  - Then translate the flowchart into C program.

Finding min of 3 numbers



Ascending order of 3 numbers



```
if (a <= b) {
    if (a <= c) {        /* a <= b and a <= c */
        if (b <= c) {    /* a <= b, a <= c, b <= c */
            printf("%d %d %d \n", a, b, c);
        } else {         /* a <= b, a <= c, c < b */
            printf("%d %d %d \n", a, c, b);
        }
    } else {             /* a <= b, c < a*/
        printf("%d %d %d \n", c, a, b) ;
    }
} else {                 /* b < a */
    if (b <= c) {        /* b < a and b <= c */
        if (a <= c) {    /* b < a, b <= c, a <= c */
            printf("%d %d %d\n", b, a, c);
        } else {         /* b < a, b <= c, c < a */
            printf("%d %d %d\n", b, c, a); }
        }
    } else {             /* b < a, c < b */
        printf("%d %d %d\n", c, b, a); }
    }
}
```

Nested if, if-else

- Earlier examples showed us *nested* if-else statements

```
if (a <= b) {
     if (a <= c) { … }  else {…}
} else {
      if (b <= c)  { … } else { … }
}
```

- Because if and if-else are also statements, they can be used anywhere a statement or block can be used.

Else if

- A special kind of nesting is the chain of if-else-if-else-… statements

```
if (cond1) {
      stmt1
} else {
     if (cond2)  {
        stmt2
     } else {
        if (cond3) {
           ….
        }
     }
}
```

General form of if-else-if-else…

```
if (cond1)
      stmt-block1
else if (cond2)
     stmt-block2
else if (cond3)
     stmt-block3
else if (cond4)
     stmt-block4
else if  …
else
     last-block-of-stmt
```

Example

- Given an integer day, where $1 \leq day \leq 7$, print the name of the weekday corresponding to day.
  - 1: Sunday
  - 2: Monday
  - …
  - 7: Saturday

Printing the day

```
int day;
scanf ("%d", &day);
if (day == 1) { printf("Sunday"); }
else if (day == 2) { printf ("Monday"); }
else if (day == 3) { printf ("Tuesday"); }
else if (day == 4) { printf ("Wednesday"); }
else if (day == 5) { printf ("Thursday"); }
else if (day == 6) { printf ("Friday"); }
else if (day == 7) { printf ("Saturday"); }
else { printf (" Illegal day %d", day); }
```


Example 2

- Given an integer day, where $1 \leq day \leq 7$, print Weekday, if the day corresponds to weekday, print Weekend otherwise.
  - 1, 7: Weekend
  - 2,3,4,5,6: Weekday

Weekday - version 1

```
int day;
scanf ("%d", &day);
if (day == 1) { printf("Weekend"); }
else if (day == 2) { printf ("Weekday"); }
else if (day == 3) { printf ("Weekday"); }
else if (day == 4) { printf ("Weekday"); }
else if (day == 5) { printf ("Weekday"); }
else if (day == 6) { printf ("Weekday"); }
else if (day == 7) { printf ("Weekend"); }
else { printf (" Illegal day %d", day); }
```


Weekday - version 2

```
int day;
scanf ("%d", &day);
if ((day == 1) || (day == 7)) {
      printf("Weekend");
} else if ( (day == 2) || (day == 3)
          || (day == 4) || (day == 5)
          || (day == 6)) {
      printf ("Weekday");
} else {
      printf (" Illegal day %d", day);
}
```


Weekday - version 3

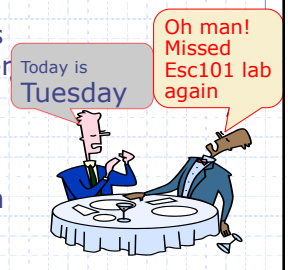
```
int day;
scanf ("%d", &day);
if ((day == 1) || (day == 7)) {
      printf("Weekend");
} else if ( (day >= 2) && (day <= 6) ) {
      printf ("Weekday");
} else {
      printf (" Illegal day %d", day);
}
```

Summary of if, if-else

- if-else, nested if's, else if.
- Braces {…} can be omitted if a block has only one statement.
- Multiple ways to solve a problem
  - issues of better readability
  - and efficiency.

Switch-case statement

- Multi-way decision
- Checks whether an expression matches one out of a number of constant integer (or char) values
- Execution *branches* based on the match found



Printing the day, version 2

```
switch (day) {
case 1: printf("Sunday"); break;
case 2: printf ("Monday"); break;
case 3: printf ("Tuesday"); break;
case 4: printf ("Wednesday"); break;
case 5: printf ("Thursday"); break;
case 6: printf ("Friday"); break;
case 7: printf ("Saturday"); break;
default: printf (" Illegal day %d", day);
}
```

Weekday, version 4

```
switch (day) {
case 1:
case 7: printf ("Weekend"); break;
case 2:
case 3:
case 4:
case 5:
case 6: printf ("Weekday"); break;
default: printf (" Illegal day %d", day);
}
```


General Form of switch-case

```
switch (selector-expr) {
case label1: s1; break;
case label2: s2; break;
...
case labelN: sN; break;
default : sD;
}
```


Surprise Quiz (not graded!)

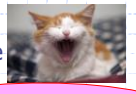
- What is the value of expression:

```
(5<2) && (3/0)
```

a) Compile time error

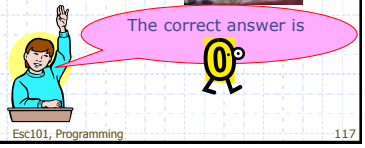
b) Run time crash

c) I don't know / I don't care



d) 0

e) 1



Short-circuit Evaluation

- Do not evaluate the second operand of binary logical operator if result can be deduced from first operand
  - Arguments of && and || are evaluated from left to right (in sequence)
  - Also applies to nested logical operators

1 0 0 1

!((2>5) && (3/0)) || (4/0)

Evaluates to 1

Surprise Quiz 2

- What is the value of expression:

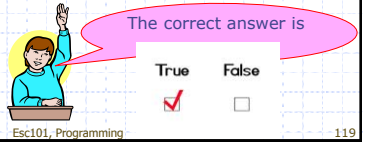
0 <= 10 <= 4

a) Compile time error

b) Run time crash

c) False (0)

d) True (1)



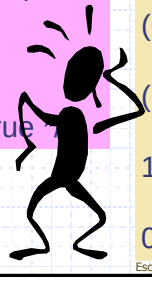
Evaluation

0 <= 10 <= 4

(0 <= 10) <= 4

1 <= 4

1 /* True */



Probably not what you intended. The intended expression is:

0 <= 10 && 10 <= 4

(0 <= 10) && (10 <= 4)

(1) && (10 <= 4)

1 && (0)

0 /*False*/

Unmatched if and else

```
if ((a != 0) && (b != 0))
   if (a * b >= 0)
      printf (“positive”);
else
   printf(“zero”);
```

OUTPUT for a = 5, b = 0
NO OUTPUT!!
OUTPUT for a = 5, b = -5
zero

```
if ((a != 0) && (b != 0))
   if (a * b >= 0)
      printf (“positive”);
   else
      printf(“negative”);
```

OUTPUT for a = 5, b = 0
NO OUTPUT!!
OUTPUT for a = 5, b = -5
negative

Unmatched if and else

- An else always matches closest unmatched if
  - Unless forced otherwise using { … }

```
if (cond1)
   if (cond2)
      …
   else
      …
```

```
if (cond1) {
   if (cond2)
      …
   else
      …
}
```


Unmatched if and else

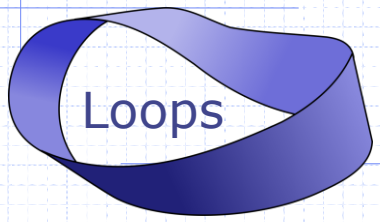
- An else always matches closest unmatched if
  - Unless forced otherwise using { … }

```
if (cond1)
   if (cond2)
      …
   else
      …
```

IS NOT SAME AS

```
if (cond1) {
   if (cond2)
      …
}
else
      …
```

ESC101: Introduction to Computing



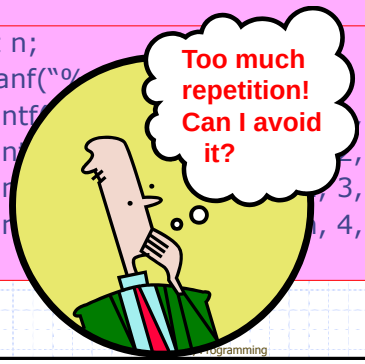
Loops

Printing Multiplication Table

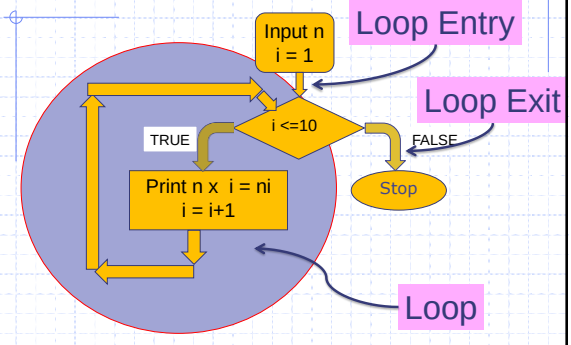
5	X	1	=	5
5	X	2	=	10
5	X	3	=	15
5	X	4	=	20
5	X	5	=	25
5	X	6	=	30
5	X	7	=	35
5	X	8	=	40
5	X	9	=	45
5	X	10	=	50

Program…

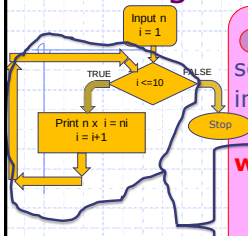
```
int n;
scanf("%
printf                    n*1);
prin                   , n*2);
prin                , 3, n*2);
prin                , 4, n*4);
....
```



Printing Multiplication Table



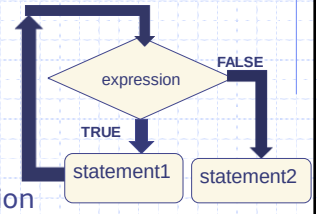
Printing Multiplication Table




```
scanf("%d", &n);
int i = 1;
while (i <= 10) {
  printf("%d X %d = %d",
         n, i, n*i);
  i = i + 1;
}

// loop exited!
```


While Statement

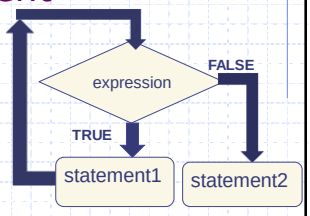
```
while (expression)
      statement1;
statement2;
```



1. Evaluate expression
2. If TRUE then
   a) execute statement1
   b) goto step 1.
3. If FALSE then execute statement2.

While Statement

```
while (expression)
        statement1;
statement2;
```



Read in English as:

As long as expression is TRUE execute statement1.

when expression becomes FALSE execute statement 2.

Example 1

1. Read a sequence of integers from the terminal until -1 is read.
2. Output sum of numbers read, not including the -1..

First, let us write the loop, then add code for sum.

```
int a;
scanf("%d", &a);              /* read into a */
while ( a != -1) {
        scanf("%d", &a);    /* read into a inside
loop*/
}
```


Tracing the loop

```
int a;
scanf("%d", &a);              /* read into a */
while ( a != -1) {
        scanf("%d", &a);  /*read into a inside
loop*/
}
```

INPUT
4
15
-5
-1

Trace of memory location a

-1

Add numbers until -1

- Keep an integer variable s. s is the sum of the numbers seen so far (except the -1).

```
int a;
int s;
s = 0;
scanf("%d", &a);      /* read into a */
while (a != -1) {
    s = s + a;
    scanf("%d", &a);  /*read into a inside loop*/
}
/* one could print s here etc. */
```


Terminology

- Iteration: Each run of the loop is called an iteration.
  - In example, the loop runs for 3 iterations, corresponding to inputs 4, 15 and -5.
  - For input -1, the loop is exited, so there is no iteration for input -1.
- 3 components of a while loop
  - Initialization
    - first reading of a in example
  - Condition (evaluates to a Boolean value)
    - a != -1
  - Update
    - another reading of a

```
scanf("%d", &a);      /* read into a */
while (a != -1) {
    s = s + a;
    scanf("%d", &a);  /*read into a inside loop*/
}
// INPUTS: 4  15  -5  -1
```


Common Mistakes

- Initialization is not done
  - Incorrect results. Might give error.
- Update step is skipped
  - Infinite loop: The loop goes on forever. Never terminates.
  - Our IDE will exit with "TLE" error (Time Limit Exceeded)
  - The update step must take the program towards the condition evaluating to false.
- Incorrect termination condition
  - Early or Late exit (even infinite loop).

Practice Problem

- Given a positive integer n, print all the integers less than or equal to n that are divisible by 3 or divisible by 5
- Hint: Two conditions will be used:
  - x <= n
  - (x%3 == 0) || (x%5 == 0)


```
int n; int x;
scanf("%d", &n);      // input n

x = 1;                // [while] initialization
while ( x <= n) {     // [while] cond

   if ((x%3 == 0) || (x%5 == 0)) { // [if] cond
      printf("%d\n", x);
   }

   x = x+1;           // [while] update
}
```

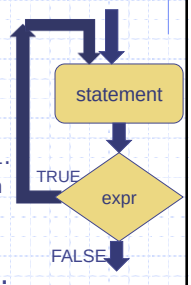

do-while loops

- do-while statement is a variant of while.

General form:

```
do
    statement
while (expr);
```

- Execution:
  1. First execute statement.
  2. Then evaluate expr.
  3. If expr is TRUE then go to step 1.
  4. If expr is FALSE then break from loop
- Continuation of loop is tested after the statement.



Comparing while and do-while

- In a while loop the body of the loop may not get executed even once, whereas, in a do-while loop the body of the loop gets executed at least once.
- In the do-while loop structure, there is a semicolon after the condition of the loop.
- Rest is similar to a while loop.

Comparative Example

- Problem: Read integers and output each integer until -1 is seen (include -1 in output).
- The program fragments using while and do-while.

Using do-while

```
int a; /*current int*/

do {
      scanf("%d", &a);
      printf("%d\n",
a);
} while (a != -1);
```

Using while

```
int a;/*current int*/

scanf("%d",&a);
while (a != -1) {
      printf("%d\n",
a);
      scanf("%d", &a);
}
printf("%d\n", a);
```

Comparative Example

- The while construct and do-while are equally expressive
  - whatever one does, the other can too.
  - but one may be *more readable* than other.

Using do-while

```
int a; /*current int*/

do {
      scanf("%d", &a);
      printf("%d\n",
a);
} while (a != -1);
```

Using while

```
int a;/*current int*/

scanf("%d",&a);
while (a != -1) {
      printf("%d\n",
a);
      scanf("%d", &a);
}
printf("%d\n", a);
```

For Loop

- Print the sum of the reciprocals of the first 100 natural numbers.

```
int i;              /* counter: runs from 1..100  */
float reciproc_sum = 0.0;  /* sum of reciprocals */

for (i=1; i<=100; i=i+1) {      /* the  for loop */
      reciproc_sum = reciproc_sum +  (1.0/i);
}

printf("sum of reciprocals of 1 to 100 is  %f ",
        reciproc_sum);
```

For loop in C

- General form

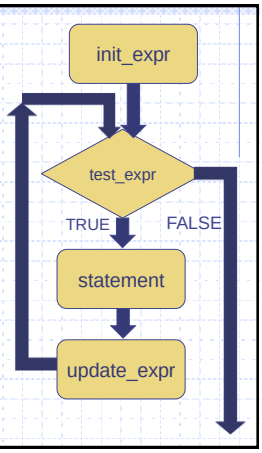
for (init_expr; test_expr; update_expr)
statement;

- init_expr is the initialization expression.
- update_expr is the update expression.
- test_expr is the expression that evaluates to either TRUE (non-zero) or FALSE (zero).

For loop in C

for (init_expr; test_expr; update_expr)
statement;

1. First evaluate **init_expr**;
2. Evaluate **test_expr**;
3. If **test_expr** is TRUE then
   a) execute **statement**;
   b) execute **update_expr**;
   c) go to Step 2.
4. if **test_expr** is FALSE then break from the loop



```
int i;
float reciproc_sum = 0.0;
for (i=1; i<=4; i=i+1) {
    reciproc_sum =
        reciproc_sum + (1.0/i);
}
printf("sum of reciprocals is %f",
        reciproc_sum);
```

i	reciproc_sum
5	2.0833333..

1. Evaluate init_expr; i.e., i=1;
2. Evaluate test_expr i.e., i<=4 TRUE
3. Enter body of loop and execute.
4. Execute update_expr; i=i+1; i is 2
5. Evaluate test_expr i<=4: TRUE
6. Enter body of loop and execute.
7. Execute i=i+1; i is 3
8. Evaluate test_expr i<=4: TRUE
9. Enter body of loop and execute.
10. Execute i=i+1; i is 4
11. Evaluate test_expr i<=4: TRUE
12. Enter body of loop and execute.
13. Execute i=i+1; i is 5
14. Evaluate test_expr i<=4: FALSE
15. Exit loop jump to printf statement

sum of reciprocals of is 2.083333

4 is 2.083333 $

For loop in terms of while

```
for (init_expr; test_expr; update_expr)
        statement;
```

- Execution is (almost) equivalent to

```
init_expr;
while (test_expr) {
        statement;
        update_expr;
}
```

- Almost? Exception if there is a continue; inside statement– this will be covered later.
- Both are equivalent in power.
- Which loop structure to use, depends on the convenience of the programmer.

Example: Geometric Progression

- Given positive real numbers r and a, and a positive integer, n, the n^{th} term of the geometric progression with a as the first term and r as the common ratio is ar^{n-1}.
- Write a program that given r, a, and n, displays the first n terms of the corresponding geometric progression.

```
#include<stdio.h>
int main(){
   int n, i;     float r, a, term;

   // Reading inputs from the user
   scanf("%f", &r);
   scanf("%f", &a);
   scanf("%d", &n);
   term = a;
   for (i=1; i<=n; i=i+1) {
      printf("%f\n", term); // Displaying i^th term
      term = term * r;        // Computing (i+1)^th term
   }
   return 0;
}
```


Nested Loops

- Loop with in a loop
- Many iterations of inner loop ⇒ One iteration of outer loop

Example

- Write a program that displays the following pattern

Integers are printed in 4 columns each

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25
6	12	18	24	30
7	14	21	28	35
8	16	24	32	40

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

   for (i=1; i<=8; i=i+1) {
      for (j=1; j<=5; j=j+1) {
         printf("%4d", i*j); // Displaying i, 2i, …, 5i
      }
      printf("\n"); // Move to the next line
   }

   return 0;
}
```

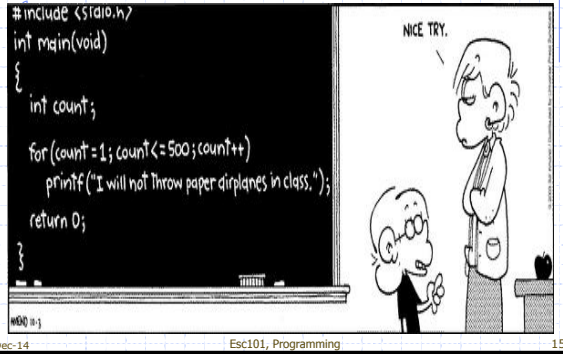

Displaying a pattern

```
#include <stdio.h>
int main(){
  int i,j;
  for (i=1; i<=5; i=i+1){
    for (j=i; j<2*i; j=j+1){
      printf("%d ",j);
    }
    printf("\n");
  }
  return 0;
}
```

- Output?

```
1
2 3
3 4 5
4 5 6 7
5 6 7 8 9
```


Not always advisable to avoid repetitive job ☺



increment/decrement operator

- Two very common actions in C

```
i = i + 1;
i = i – 1;
```

- These can be written in short as:

```
i++     // increment
i--     // decrement
```

- Complete semantics are bit involved
  - Not covered in this course
  - Advise: Do not use them other then in update_expr of for loops!

Continue and Break

Break

- Used for exiting a loop forcefully
- Example Program:

Read 100 integer inputs from a user. Print the sum of inputs until a negative input is found (Excluding the negative number) or all 100 inputs are exhausted.

```
int value;
int sum = 0;
int i;
for (i = 0; i < 100; i++) {
     scanf("%d", &value);
     if (value < 0) {
          //-ve number: no need to go
          // around the loop any more!!
          break;
     }
     sum = sum + value;
}
printf("%d", sum);
```


To break or not to!

- Use of break sometimes can simplify exit condition from loop.
- However, it can make the code a bit harder to read and understand.
- Tip: if the loop terminates in at least two ways which are sufficiently different and requires substantially different processing then consider the use of termination via break for one of them.

Continue

- Used for skipping an iteration of a loop
- The loop is NOT exited.
- Example Program:

Read 100 integer inputs from a user.
Print the sum of only positive inputs.

```
int sum = 0;
int i, value;
for (i = 0; i < 100; i++) {
     scanf("%d", &value);
     if (value < 0) {
          //-ve number: no need to add it
          // to the sum. Go ahead and
          // check the next input.
          continue;
     }
     sum = sum + value;
}
printf("%d", sum);
```


Break and Continue

- if there are nested loop, break and continue apply to the immediately enclosing loop only.

```
for (i = 0; i < 100; i++) {
  for (j = 0; j < 100; j++) {
     if (…) break;
  }
  …
}
```


Continue and Update Expr

- Make sure continue does not bypass update-expression for loops
  - Specially for while and do-while loops

```
i = 0;
while (i < 100) {
  scanf("%d", &value);
  if (value < 0) continue;
  sum = sum + value;
  i++;
}
```

i is never incremented
potentially infinite loop!!

Continue and Update Expr

- Correct Code:

```
i = 0;
while (i < 100) {
   i++;
   scanf("%d", &value);
   if (value < 0) continue;
   sum = sum + value;
}
```


Continue and Update Expr

- Correct Code:

```
i = 0;
while (i < 100) {
   scanf("%d", &value);
   if (value < 0) {
      i++;
      continue;
   }
   sum = sum + value;
   i++;
}
```


Quiz 3: Give output of prog.

- A common bug

```
int a =4;
while (a < 10) {
      if (a = 5) {
            printf("%d\n", a);
      }
      a=a+1;
}
```

5
5
5
5
...

Probable intention:

```
int a =0;
while (a < 10) {
      if (a == 5) {
            printf("%d", a);
      }
      a=a+1;
}
```

5

Ternary operator ?:

- Select among values of two expressions based on a condition

  condition ? true_expr : false_expr

- Both expressions must be of compatib type.
  - The expression is called ternary expression

Condition

value if condition is True

value if condition is False

```
int abs;
int val;
scanf ("%d", val);
if (val < 0)
    abs = -val;
else
    abs = val;
printf("%d", abs);
```

```
int abs;
int val;
scanf ("%d", val);
abs = (val < 0) ? –val : val;
printf("%d", abs);
```

```
int val;
scanf ("%d", val);
printf("%d", (val < 0) ? –val : val);
```


ESC101: Introduction to Computing

f(unction)

A Modern Smartphone

- Surf the net
  - Input: Web address
  - Output: Desired page
- Book tickets
  - Input: userid, password, booking info, bank info
  - Output: Ticket
- Send email
  - Input: email address of receiver, mail text
  - Output: --
- Take photos
  - Input: --
  - Output: Picture
- Talk (we can do that too!!)
  - Input: Phone number
  - Output: Conversation (if lucky)
- …

Lots of related/unrelated task to perform

- Divide and Conquer
  - Create well defined sub tasks
  - Work on each task independently
    - Development, Enhancements, Debugging
- Reuse of tasks.
  - Email and Chat apps can share spell checker.
  - Phone and SMS apps can share dialer
- Functions in C

Function

- An independent, self-contained entity of a C program that performs a well-defined task.
- It has
  - Name: for identification
  - Arguments: to pass information from outside world (rest of the program)
  - Body: processes the arguments do something useful
  - Return value: To communicate back to outside world
    - Sometimes not required

Why use functions?

Example : Maximum of 3 numbers

```
int main(){
   int a, b, c, m;

   /* code to read
    * a, b, c */

   if (a>b){
      if (a>c) m = a;
      else m = c;
   }
   else{
      if (b>c) m = b;
      else m = c;
   }

   /* print or use m */

   return 0;
}
```

```
int max(int a, int b){
   if (a>b)
     return a;
   else
     return b;
}

int main() {
   int a, b, c, m;

   /* code to read
    * a, b, c */

   m = max(a, b);
   m = max(m, c);
   /* print or use m */

   return 0;
}
```

This code can scale easily to handle large number of inputs (e.g.: max of 100 numbers!)

Why use functions?

- Break up complex problem into small sub-problems.
- Solve each of the sub-problems separately as a function, and combine them together in another function.
- The main tool in C for modular programming.

Modularity…

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Source: http://xkcd.com/221/

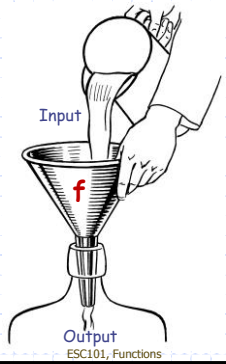
We have seen functions before

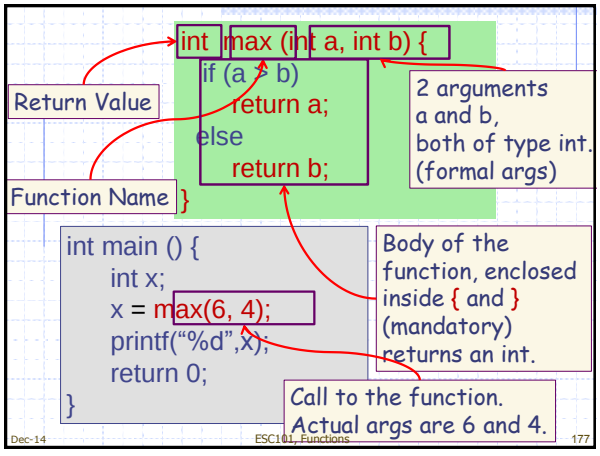
- main() is a special function. Execution of program starts from the beginning of main().
- scanf(…), printf(…) are standard input-output library functions.
- sqrt(…), pow(…) are math functions.

Advantages of using functions

- **Code Reuse**: Allows us to reuse a piece of code as many times as we want, without having to write it.
  - Think of the `printf` function!
- **Procedural Abstraction**: Different pieces of your algorithm can be implemented using different functions.
- **Distribution of Tasks**: A large project can be broken into components and distributed to multiple people.
- **Easier to debug**: If your task is divided into smaller subtasks, it is easier to find errors.
- **Easier to understand**: Code is better organized and hence easier for an outsider to understand it.

Parts of a function





Function Call

- A function call is an *expression*
  - feeds the necessary values to the function arguments,
  - directs a function to perform its task, and
  - receives the return value of the function.
- Similar to operator application

5 + 3 is an expression of type integer that evaluates to 8

max(5, 3) is an expression of type integer that evaluates to 5

Function Call

- Since a function call is an *expression*
  - it can be used anywhere an expression can be used
  - subject to type restrictions

```
printf("%d", max(5,3));
max(5,3) – min(5,3)
max(x, max(y, z)) == z

if (max(a, b)) printf("Y");
```

prints 5
evaluates to 2
checks if z is max of x, y, z
prints Y if max of a and b is not 0.

Returning from a function: Type

- Return type of a function tells the type of the result of function call
- Any valid C type
  - int, char, float, double, …
  - **void**
- Return type is void if the function is not supposed to return any value

```
void print_one_int(int n) {
    printf("%d", n);
}
```

Returning from a function: return statement

- If return type is not void, then the function MUST return a value:

  return return_expr;

- If return type is void, the function may *fall through* at the end of the body or use a return without return_expr:

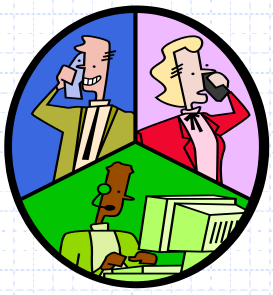
  return;

```
void print_positive(int n) {
    if (n <= 0) return;
    printf("%d", n);
}
```


Returning from a function: return statement

- When a return statement is encountered in a function definition
  - control is immediately transferred back to the statement making the function call in the parent function.
- A function in C can return only ONE value or NONE.
  - Only one return type (including void)

Execution of a Function: Steps



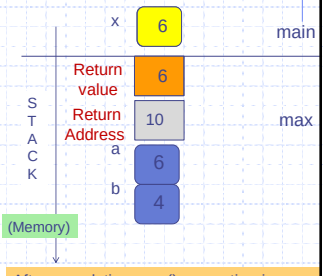
```
1  #include <stdio.h>
2  int max(int a, int b) {
3    if (a > b)
4      return a;
5    else
6      return b;
7  }
8  int main () {
9      int x;
10     x =  10a  x(6, 4);
11     printf("%d",x);
12     return 0;
13 }
```

- Steps when a function is called: max(6,4) in step 10a.
- Allocate space for (i) return value, (ii) store return address and (iii) pass parameters.

1. Create a box informally called ``Return value'' of same type as the return type of function.
2. Create a box and store the location of the next instruction in the calling function (main)—return address. Here it is 10 (why not 11?). Execution resumes from here once function terminates.
3. Parameter Passing- Create boxes for each formal parameter, a, b here. Initialize them using actual parameters, 6 and 4.


```
1  #include <stdio.h>
2  int max(int a, int b) {
3    if (a > b)
4      return a;
5    else
6      return b;
7  }
8  int main () {
9      int x = -1;
10     x = max(6, 4);
11     printf("%d",x);
12     return 0;
13 }
```

Calling max(6,4):
1. Allocate space for return value.
2. Store return address (10).
3. Pass parameters.



After completing max(), execution in main() will re-start from address 10.

Stack

- We referred to stack.
- A stack is just a part of the memory of the program that grows in one direction only.
- The memory (boxes) of all variables defined as actual parameters or local variables reside on the stack.
- The stack grows as functions call functions and shrinks as functions terminate.

Function Declaration

- A function declaration is a statement that tells the compiler about the different properties of that function
  - name, argument types and return type of the function
- Structure:

```
return_type function_name (list_of_args);
```

- Looks very similar to the first line of a function definition, but NOT the same
  - has semicolon at the end instead of BODY

Function Declaration

```
return_type function_name (list_of_args);
```

- Examples:
  - int max(int a, int b);
  - int max(int x, int y);
  - int max(int , int);

All 3 declarations are equivalent! Since there is no BODY here, argument names do not matter, and are optional.

- Position in program: Before the call to the function
  - allows compiler to detect inconsistencies
  - Header files (stdio.h, math.h,…) contain declarations of frequently used functions
  - #include <…> just copies the declarations
    - more on this later

Evaluating expressions

- Associativity and Precedence precisely define what an expression means, e.g.,

  a * b – c / d

  is same as:

  (a * b) – (c / d)

- But how exactly are expressions evaluated?

Evaluating expression (a*b) – (c/d)

- Computers evaluate one operator at a time.
- Above expr may be evaluated as

```
t1 = c/d;
t2 = a*b;
t3 = t2 – t1;
```

  - t1, t2, t3 are temporary variables, created by the compiler (not by programmer). They are temporary, meaning, their lifetime is only until use.

Evaluating expression (a*b) – (c/d)

- The order of evaluation of arguments for most C operators is not defined
  - Exception are && and ||
  - Remember: short circuit evaluation
- Above expr may also be evaluated as

```
t1 = a*b;
t2 = c/d;
t3 = t1 – t2;
```

  - Do not depend on order of evaluation of args, except for && and ||
  - Same holds true for function arguments


```
int main () {
    int a =4, b = 2, c = 5, d = 6;
    a = a + b *c-d/a;
    b = b-(c-d)/a;
    return 0;}
```

Compiler evaluates as follows. (Following is not a C program)

```
int main () {
    int a =4, b = 2, c = 5, d = 6;
    /* evaluate as  a = (a + (b *c)) - (d/a);
    create temporary variables t1, t2, t3
    of the right types: int   */
    t1 = b*c;
    t2 = d/a;
    t3 = a + t1;  /* t1 expires */
    a = t3 –t2;   /* t2,t3 expires */
    b = b-(c-d)/a;
    return 0; }
```

a	13
b	2
c	5
d	6

Temporary Variables

t1	10	expired
t2	1	expired
t3	14	expired

Even this statement is not atomic.

```
int main () {
    int a =4, b = 2, c = 5, d = 6;
    a = a + b *c-d/a;
    b = b-(c-d)*a;
    printf("%d %d",a,b);
    return 0;}
```

Evaluating statement 3.
(Following is not a C program)

```
int main () {
    int a =4, b = 2, c = 5, d = 6;
    a = a + b *c-d/a;
    /* evaluate b = b-((c-d)*a)
    create temporary variables t1, t2
    of the right types: int   */
    t1 =c-d;
    t2 = t1*a;  /* t1 expires */
    b  = b-t2;  /* t2 expires */
    printf("%d %d", a, b);
    return 0; }
```

State of the program just prior to second assignment

a	13
b	15
c	5
d	6

Temporary Variables

t1	-1	expired
t2	-13	expired


```
int main () {
    int a =4, b = 2, c = 5, d = 6;
    a = a + b *c-d/a;
    b = b-(c-d)*a;
    printf("%d %d",a,b);
    return 0; }
```

State of the program just prior to printf

a	13
b	15
c	5
d	6

Output

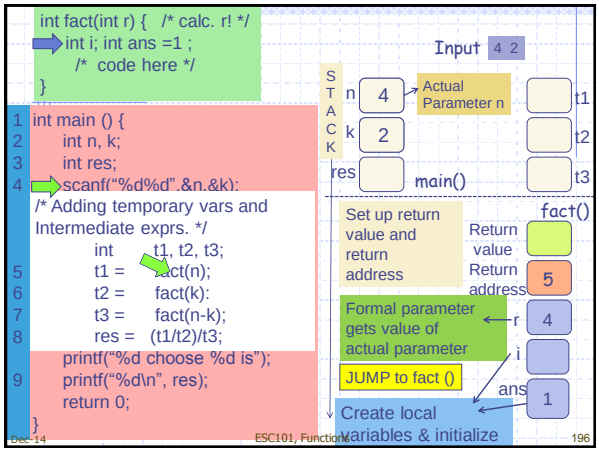
13 15

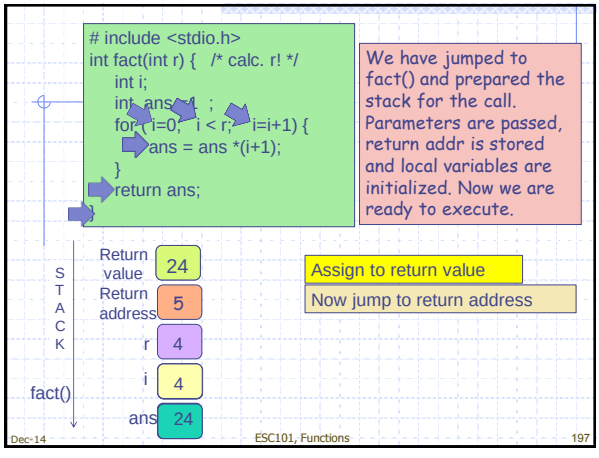
Since each function call is also an expression, statements having mix of function calls and operators are also evaluated in a similar fashion.

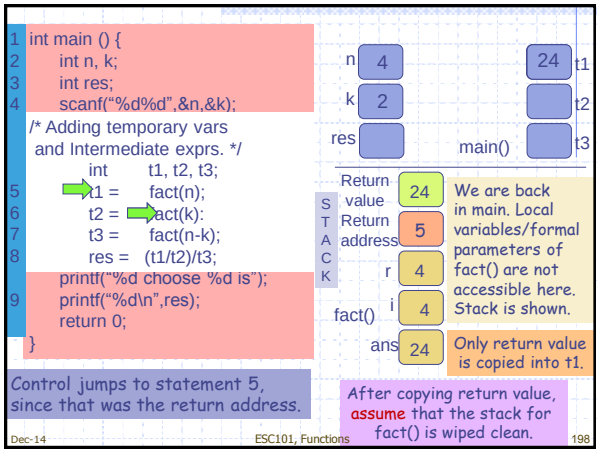

```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
    int i;
    int  ans=1;
    for (i=0; i < r; i=i+1) {
        ans = ans *(i+1);
    }
    return ans;
}
```

```
int main () {
    int n, k;
    int res;
    scanf("%d%d",&n,&k);
    res = (fact(n)/ fact(k))/fact(n-k);
    printf("%d choose %d is",n,k);
    printf("%d\n",res);
    return 0;
}
```

- Define a factorial function.
- Use to calculate ${}^{n}C_{k}$
- Let us trace the execution of main().
- Add temporary variables for expressions and intermediate expressions in main for clarity.


int fact(int r) { /* calc. r! */
int i; int ans =1 ;
/* code here */
}
1 int main () {
2 int n, k;
3 int res;
4 scanf("%d%d",&n,&k);
/* Adding temporary vars and
Intermediate exprs. */
int t1, t2, t3;
5 t1 = fact(n);
6 t2 = fact(k);
7 t3 = fact(n-k);
8 res = (t1/t2)/t3;
printf("%d choose %d is");
9 printf("%d\n", res);
return 0;
}
Input 4 2
STACK
n 4
Actual Parameter n
k 2
res
main()
t1
t2
t3
fact()
Set up return value and return address
Return value
Return address
5
Formal parameter gets value of actual parameter
r 4
i
JUMP to fact ()
ans 1
Create local variables & initialize
Dec-14
ESC101, Functions
196


include <stdio.h>
int fact(int r) { /* calc. r! */
int i;
int ans =1 ;
for (i=0; i < r; i=i+1) {
ans = ans *(i+1);
}
return ans;
}
We have jumped to fact() and prepared the stack for the call. Parameters are passed, return addr is stored and local variables are initialized. Now we are ready to execute.
STACK
Return value 24
Return address 5
r 4
i 4
fact()
ans 24
Assign to return value
Now jump to return address
Dec-14
ESC101, Functions
197


1 int main () {
2 int n, k;
3 int res;
4 scanf("%d%d",&n,&k);
/* Adding temporary vars and Intermediate exprs. */
int t1, t2, t3;
5 t1 = fact(n);
6 t2 = fact(k);
7 t3 = fact(n-k);
8 res = (t1/t2)/t3;
printf("%d choose %d is");
9 printf("%d\n",res);
return 0;
}
Control jumps to statement 5, since that was the return address.
n 4
k 2
res
main()
24 t1
t2
t3
STACK
Return value 24
Return address 5
r 4
i 4
fact()
ans 24
We are back in main. Local variables/formal parameters of fact() are not accessible here. Stack is shown.
Only return value is copied into t1.
After copying return value, assume that the stack for fact() is wiped clean.
Dec-14
ESC101, Functions
198

```
1  int main () {
2      int n, k;
3      int res;
4      scanf("%d%d",&n,&k);
   /* Adding temporary vars
    and Intermediate exprs. */
           int     t1, t2, t3;
5          t1 =    fact(n);
6          t2 =    fact(k):
7          t3 =    fact(n-k);
8          res =   (t1/t2)/t3;
       printf("%d choose %d is");
9      printf("%d\n",res);
       return 0;
   }
```

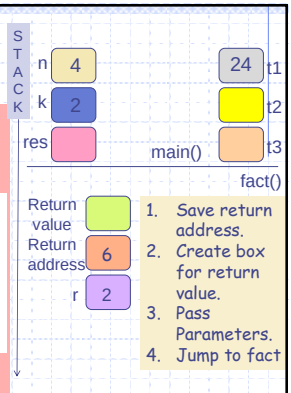
After copying return value, assume that the stack for fact() is wiped clean.

The next statement is another function call: fact(k). Prepare stack for new call.

1. Save return address.
2. Create box for return value.
3. Pass Parameters: Create boxes corresponding to formal parameters. Copy values from actual parameters.
4. Jump to called function.
5. Create/initialize local variables.

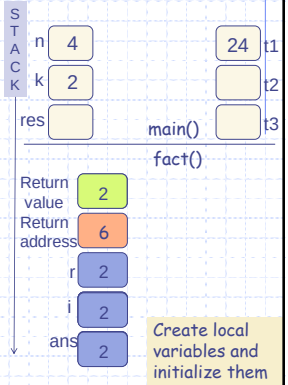

```
int fact(int r) {   /* calc. r! */
    int i; int ans =1 ;
      /*  code here */
}
```

```
1  int main () {
2      int n, k;
3      int res;
4      scanf("%d%d",&n,&k);
   /* Adding temporary vars and
    Intermediate exprs. */
           int     t1, t2, t3;
5          t1 =    fact(n);
6          t2 =    fact(k):
7          t3 =    fact(n-k);
8          res =   (t1/t2)/t3;
       printf("%d choose %d is");
9      printf("%d\n",res);
       return 0;
   }
```



1. Save return address.
2. Create box for return value.
3. Pass Parameters.
4. Jump to fact


```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
    int i;
    int  ans=1;
    for (i=0; i < r; i=i+1) {
        ans = ans *(i+1);
    }
    return ans;
}
```

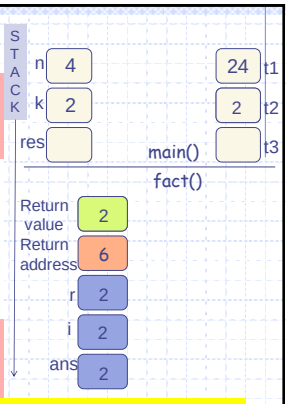


Assign value of ans to box for return value.

Create local variables and initialize them

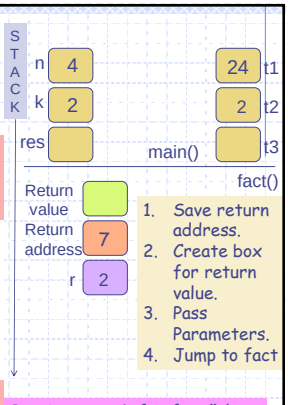
Now jump to return address

```
1  int main () {
2      int n, k;
3      int res;
4      scanf("%d%d",&n,&k);
       /* Adding temporary vars
       and Intermediate exprs. */
               int     t1, t2, t3;
5              t1 =    fact(n);
6              t2 =    fact(k):
7              t3 =    fact(n-k);
8              res =   (t1/t2)/t3;
9      printf("%d choose %d is");
       printf("%d\n",res);
       return 0;
   }
```



This is another function call. So we prepare stack. Earlier entries for fact() is erased.

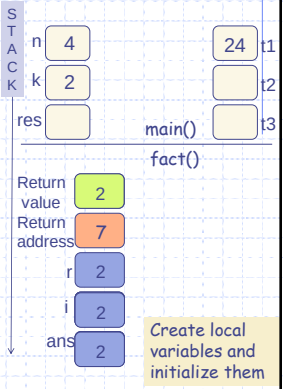

```
int fact(int r) {   /* calc. r! */
    int i; int ans =1 ;
      /*  code here */
}
1  int main () {
2      int n, k;
3      int res;
4      scanf("%d%d",&n,&k);
       /* Adding temporary vars
        and Intermediate exprs. */
               int     t1, t2, t3;
5              t1 =    fact(n);
6              t2 =    fact(k):
7              t3 =    fact(n-k);
8              res =   (t1/t2)/t3;
9      printf("%d choose %d is");
       printf("%d\n",res);
       return 0; }
```



1. Save return address.
2. Create box for return value.
3. Pass Parameters.
4. Jump to fact

Previous stack for fact() has been erased.


```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
    int i;
    int  ans=1;
    for (i=0; i < r; i=i+1) {
        ans = ans *(i+1);
    }
    return ans;
}
```

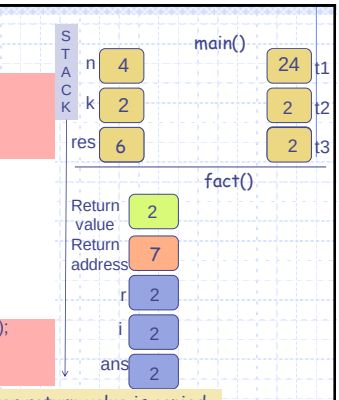


Assign value of ans to box for return value.

Create local variables and initialize them

Now jump to return address

```
1 int main () {
2     int n, k;
3     int res;
4     scanf("%d%d",&n,&k);
      /* Adding temporary vars
       and Intermediate exprs. */
            int     t1, t2, t3;
5           t1 =    fact(n);
6           t2 =    fact(k);
7           t3 =    fact(n-k);
8           res =   (t1/t2)/t3;
      printf("%d choose %d is");
9     printf("%d\n",res);
      return 0; }
```



Stack for fact() is erased after return value is copied.

4 choose 2 is 6

Nested Function Calls

- Functions can call each other
- A declaration or definition (or both) must be visible before the call
  - Help compiler detect any inconsistencies in function use
  - Compiler warning, if both (decl & def) are missing

```
#include<stdio.h>
int min(int, int); //declaration of
int max(int, int); //of max, min

int max(int a, int b) {
   return (a > b) ? a : b;
}

// a "cryptic" min, uses max
int min(int a, int b) {
   return a + b – max (a, b);
}

int main() {
  printf("%d", min(6, 4));
}
```


Predefined Functions

- C has many predefined functions. We have seen scanf, printf.
- To use a predefined function, the corresponding header file must be included.
  - Mathematical functions defined in the library math.h.
  - Input/Output functions in stdio.h
  - String functions in string.h

Some predefined math functions

Function	Description
double fabs(double x)	absolute value
double cos(double x)	sine
double sin(double x)	cosine
double tan(double x)	tan
double exp(double x)	e^x
double log(double x)	Natural log, x > 0
double log10(double x)	Log base 10, x > 0
double pow(double x, double y)	x^y
double sqrt(double x)	$\sqrt{x}$, x >= 0
double floor(double x)	largest integral value <= x
double ceil(double x)	smallest integral value >= x

Scope of a Name

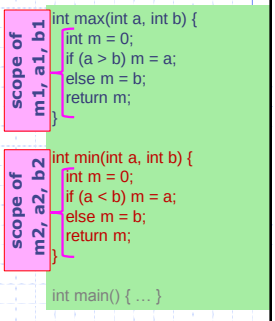
- Functions allow us to divide a program into smaller parts
  - each part does a well defined task
- There are other ways to *partition a program*
  - *Statement blocks, Files*
- Scope of a name is the part of the program in which the name can be used

Scope of a Name

- Two variables can have the same name only if they are declared in separate scopes.
- A variable can not be used outside its scope.
- C program has
  - function/block scope
  - file scope
  - global scope

Scope Rules: Functions

- The scope of the variables present in the argument list of a function's definition is the body of that function.
- The scope of any variable declared within a function is the body of the function.



Scope Rules : Blocks

- For an identifier declared at the head of a block
  - Scope begins at the end of declaration
  - Scope ends at the end of the block

```
int main() {
  int m = 5;
  {
      float m = 6.5; // shadows
      printf("%f", m);  // prints 6.5
  }
  printf("%d" , m); // prints 5
}
```


Scope of a Name

- Scopes can be nested
  - A name in inner scope "*shadows*" the same name, if present in outer scope

```
int main() {
  int m = 5;
  {
      float m = 6.5; // shadows
      printf("%f", m);  // prints 6.5
  }
  printf("%d" , m); // prints 5
}
```

Global Variable

- Variable declared outside every function definition
- Can be accessed by all functions in the program that follow the declaration
- Also called *External* variable
- What if a variable is declared inside a function that has the same name as a global variable?
  - The global variable is "shadowed" inside that particular function only.

Global Variables

```
#include<stdio.h>
int g=10, h=20;

int add(){
  return g+h;
}

void fun1(){
  int g=200;
  printf("%d\n",g);
}

int main(){
  fun1();
  printf("%d %d %d\n",
         g, h, add());
  return 0;
}
```

```
200
10 20 30
```

1. The variable g and h have been defined as global variables.
2. The use of global variables is normally discouraged. Use local variables of functions as much as possible.
3. Global variables are useful for defining constants that are used by different functions in the program.

Global Variables: example

```
const double PI = 3.14159;
double circum_of_circle(double r) {
  return 2 * PI * r; }
double area_of_circle (double r) {
  return PI * r * r;
}
```

defines PI to be of type double with value 3.14159. Qualified by const, which means that PI is a constant. The value inside the box associated with PI cannot be changed anywhere.

Static Variables

- We have seen two kinds of variables: local variables and global variables.
- There are static variables too.

```
int f () {
  static int ncalls = 0;
  ncalls  = ncalls + 1;
/* track the number of
times f() is called  */
  … body of f() …
}
```

- Use a local variable?
  - gets destroyed every time f returns
- Use a global variable?
  - other functions can change it! (dangerous)

GOAL: count number of calls to f()

SOLUTION: define **ncalls** as a **static** variable inside f().

It is created as an integer box the first time f() is called.

Once created, it **never** gets destroyed, and **retains its value across invocations of f()**.

It is like a global variable, but visible only within f().

Static variables are not allocated on stack. So they are not destroyed when f() returns.

Summary

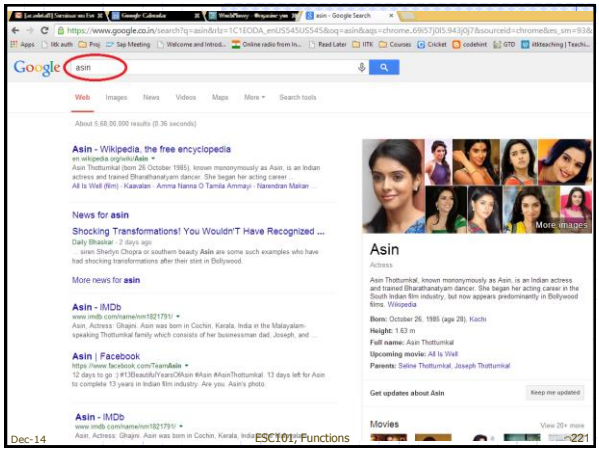
- Global Variables
  - Visible everywhere
  - Live everywhere (never destroyed)
- Local Variables
  - Visible in scope
  - Live in scope (destroyed at the point where we leave the scope)
- Static Variables
  - Visible in Scope
  - Live everywhere! (but can not be accessed outside scope)

Avoiding Common Errors

- Declare functions before use.
- Argument list of a function:
  - Provide the required number of arguments,
  - Check that each function argument has the correct type (or that conversion to the correct type will lose no information).

Avoiding Common Errors

- Return statement
  - value must match the return type
  - return statement must be encountered during execution for a function having non void return type
- Also be careful in using functions that are undefined on some values.
  - $\sin^{-1}(x)$ is defined only for $-1 \leq x \leq 1$
  - function in C : double asin(double x)
    - pronounced a-sine or arc-sine



Quiz 4: Argument Passing

```
// swapping a and b
void swap(int a, int b){
  int temp;
  temp = a;
  a = b;
  b = temp;
  printf("a=%d b=%d\n", a, b);
}
int main(){
  int  a=10, b=15;
  printf("a=%d b=%d\n", a, b);
  swap(a, b);
  printf("a=%d b=%d\n", a, b);
  return 0;
}
```

What is the output of the program? (fill the blanks)

OUTPUT

a=10 b=15

a=15 b=10

a=____ b=____

ESC101: Introduction to Computing

Defining arrays

Dictionary meaning of the word array

arr-ay: noun

1. a large and impressive grouping or organization of things: He couldn't dismiss the array of facts.
2. regular order or arrangement; series: an array of figures.

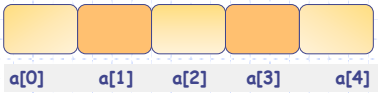
Arrays in C

An array in C is defined similar to defining a variable.

```
int a[5];
```

The square parenthesis [5] indicates that a is not a single integer but an array, that is a *consecutively allocated* group, of 5 integers.

It creates five integer boxes or variables



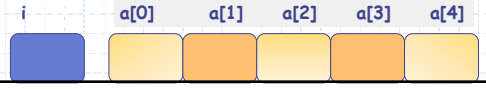
The boxes are addressed as a[0], a[1], a[2], a[3] and a[4]. These are called the **elements** of the array.

```
include <stdio.h>
int main () {
     int i;
     int a[5];

     for (i=0; i < 5; i= i+1) {
          a[i] = i;

          printf("%d", a[i] );
     }
     return 0;
}
```

The program defines an integer variable called i and an integer array with name a of size 5

This is the notation used to address the elements of the array.

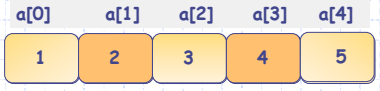
- The variable i is being used as an "index" for a.
- Similar to the math notation a_i




```
include <stdio.h>
int main () {
     int a[5];
     int i;
     for (i=0; i < 5; i= i+1) {
          a[i] = i+1;
     }
     return 0;
}
```

Let us trace through the execution of the program.

Fact : Array elements are consecutively allocated in memory.



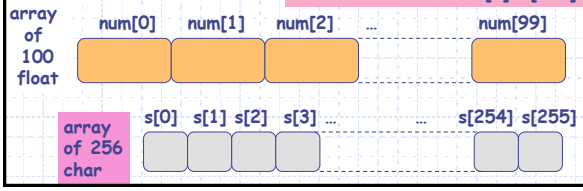
Statement becomes a[0] =0+1;
Statement becomes a[1] =1+1;
Statement becomes a[2] =2+1;
Statement becomes a[3] = 3+1;
Statement becomes a[4] = 4+1;

One can define an array of float or an array of char, or array of any data type of C. For example

```
int main() {
     float num[100];

     char s[256];

  /* some code here */
}
```

This defines an array called num of 100 floating point numbers indexed from 0 to 99 and named num[0]… num[99]

This defines an array called s of 256 characters indexed from 0 to 255 and named s[0]…s[255].



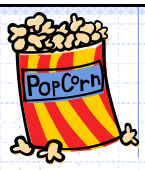
Mind the size(of array)

Consider program fragment:

```
int f() {
    int x[5];
    ...
}
```

This defines an integer array named x of size 5.

Five integer variables named x[0] x[1] … x[4] are allocated.



The variables x[0],x[1] … x[4] are integers, and can be assigned and operated upon like integers! OK, so far so good!

But what about x[5], x[6], … x[55]? Can I assign to x[5], increment it, etc.?

NO! Program may crash!

Why? x[5], x[6], and so on are undefined. These are names but no storage has been allocated. Shouldn't access them!

Q: Shouldn't I or couldn't I access array elements outside of the array range declared?

Ans: You can but shouldn't. Program may crash.


```
int f() {
    int x[5];
    x[0] =0;
    x[1] =1;
      ...
    x[4] =4;

    x[5] = 5;
    x[6] = 6;
}
```

All good

Both these statements are not recommended.

Will it compile? Yes, it will compile. C compiler may give a warning.

But, upon execution, the program may give "segmentation fault: core dumped" error or it may also run correctly and without error.

Array Example: Print backwards

Problem:

Define an character array of size 100 (upper limit) and read the input character by character and store in the array until either

- 100 characters are read or
- EOF (End Of File) is encountered

Now print the characters backward from the array.

Example Input 1	Output 1
Me or Moo	ooM ro eM
Example Input 2	**Output 2**
Eena Meena Dika	akiD aneeM aneE

Read and print in reverse

1. We will design the program in a top down fashion, using just main() function.
2. There will be two parts to main: read_into_array and print_reverse.
3. read_into_array will read the input character-by-character up to 100 characters or until a end of input.
4. print_reverse will print the characters in reverse.

Overall design

```
int main() {
        char s[100]; /* to hold the input */
        /* read_into_array */
        /* print_reverse */
        return 0;
}
```

Let us design the program fragment read_into_array.

Keep the following variables:
1. int count to count the number of characters read so far.
2. int ch to read the next character using getchar().

Note that getchar() has prototype int getchar() since getchar() returns all the 256 characters and the integer EOF

```
int ch;
int count = 0;
read the next character into ch using getchar();
while (ch is not EOF AND count < 100) {
        s[count] = ch;
        count = count + 1;
        read the next character into ch using getchar();
}
```

An initial design (pseudo-code)

```
int ch;
int count = 0;
read the next character into ch using getchar();
while (ch is not EOF AND count < 100) {
        s[count] = ch;
        count = count + 1;
        read the next character into ch using getchar();
}
```

initial design pseudo-code

```
int ch;
int count = 0;
ch = getchar();
while ( ch != EOF && count < 100) {
        s[count] = ch;
        count = count + 1;
        ch = getchar();
}
```

Translating the read_into_array pseudo-code into code.

Overall design

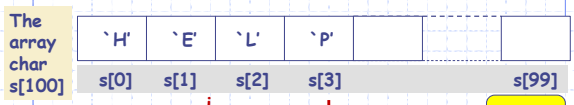
```
int main() {
    char s[100];
 /* read_into_array */
 /* print_reverse */
    return 0;
}
```

What is the value of count at the end of read_into_array?

Now let us design the code fragment print_reverse

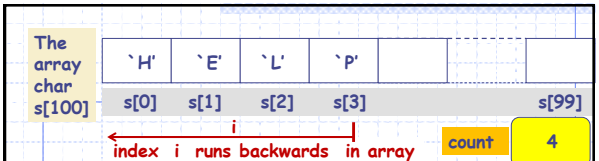
Suppose input is

HELP<eof>




```
int i;
set i to the index of last character read.

while (i >= 0) {
    print s[i]
    i = i-1;        /* shift array index one to left */
}
```

PSEUDO CODE




```
int i;
set i to index of the last character read.

while (i >= 0) {
    print s[i]
    i = i-1;
}
```

PSEUDO CODE

Translating pseudo code to C code: print_reverse

```
int i;
i = count-1;

while (i >=0) {
      putchar(s[i]);
      i=i-1;
}
```

Code for printing characters read in array in reverse

Putting it together

Overall design

```
int main() {
      char s[100];
      /* read_into_array */
      /* print_reverse */
      return 0;
}
```

The code fragments we have written so far.

```
int count = 0;
int ch;
ch = getchar();
while ( ch != EOF && count < 100) {
      s[count] = ch;
      count = count + 1;
      ch = getchar();
}
```

read_into_array code.

```
int i;
i = count-1;
while (i >=0) {
      putchar(s[i]);
      i=i-1;
}
```

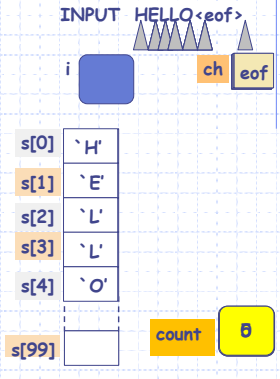
print_reverse code

```
#include <stdio.h>
int main() {
    char s[100];        /* the array of 100 char */
    int count = 0;      /* counts number of input chars read */
    int ch;             /* current character read  */
    int i;              /* index for printing array backwards */
    ch = getchar();                          /*read_into_array */
    while ( ch != EOF && count < 100) {
        s[count] = ch;
        count = count + 1;
        ch = getchar();
    }
    i = count-1;
    while (i >=0) {
        putchar(s[i]);
        i=i-1;
    }                                        /*print_in_reverse */
    return 0;
}
```

Putting code together

Let us trace the execution.
We will do this for part
read_into_array

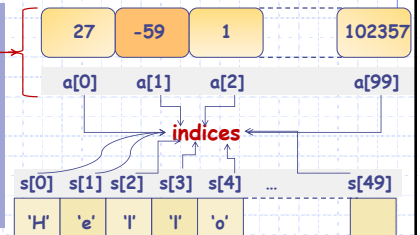
```
#include <stdio.h>
int main() {
    char s[100];
    int count = 0;
    int ch, i;
    ch = getchar();
    while ch != EOF &&
          count < 100) {
        s[count] = ch;
        count = count + 1;
        ch = getchar();
    }
```



Arrays: Recap

Arrays are a consecutively allocated group of variables whose names are indexed.

```
int a[100];
a[0]=27;
a[1]=-59;
a[2]=1;
...
char s[50];
s[0]='H';
s[1]='e';
...
```



Indices always start at 0 in C

Reading directly into array

Read N numbers from user directly into an array

```
#include <stdio.h>
int main() {
  int num[10];
  for (i=0; i<10; i=i+1) {
      scanf("%d", &num[i]);
  }
  return 0;
}
```

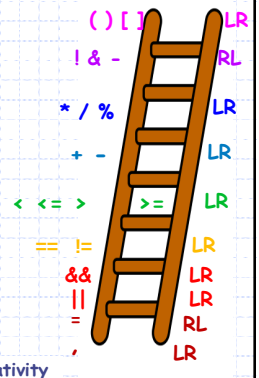
scanf can be used to read directly into array by treating an array element like any other variable of the same data type.

1. For integers, read as scanf("%d", &num[i]);
2. For reading elements of a char array s[], use scanf("%c", &s[j]).

We have seen that &num[i] is evaluated by applying the indexing operator first and the address-of operator second.

More formally, the precedence of the operators in C reflects this.

1. The array indexing operator [] is given higher precedence than the address-of operator &.
2. So &num[i] is evaluated by applying the array operator first and the address-of operator next.



Legend LR: Left-to-Right associativity
RL: Right -to-Left associativity

Passing arrays to functions

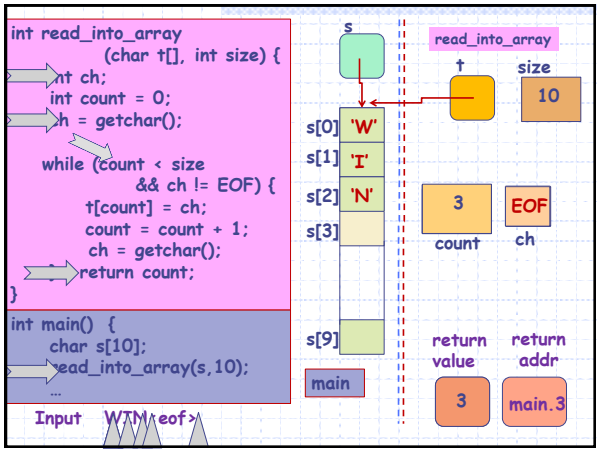
Write a function that reads input into an array of characters until EOF is seen or array is full.

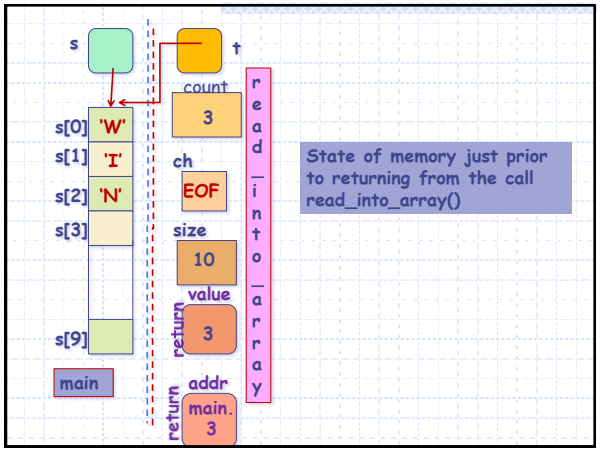
```
int read_into_array
        (char t[], int size);
/* returns number of chars
   read  */
```

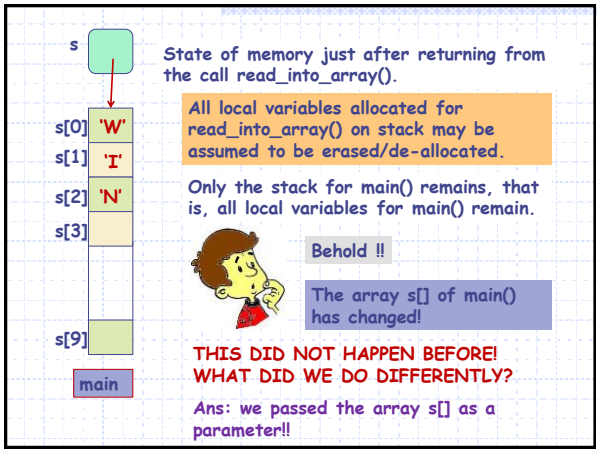
read_into_array takes an array t as an argument and size of the array and reads the input into array.

```
int main()  {
  char s[100];
  read_into_array(s,100);
  /* process */
}
```

```
int read_into_array
        (char t[], int size) {
    int ch;
    int count = 0;
    ch = getchar();
    while (count < size
            && ch != EOF) {
        t[count] = ch;
        count = count + 1;
        ch = getchar();
    }
    return count;
}
```


int read_into_array
(char t[], int size) {
nt ch;
int count = 0;
h = getchar();
while (count < size
&& ch != EOF) {
t[count] = ch;
count = count + 1;
ch = getchar();
return count;
}
int main() {
char s[10];
ead_into_array(s,10);
...
Input WIN\r\neof>
s
s[0] 'W'
s[1] 'I'
s[2] 'N'
s[3]
s[9]
main
read_into_array
t
size
10
3
count
EOF
ch
return value
3
return addr
main.3


s
t
s[0] 'W'
s[1] 'I'
s[2] 'N'
s[3]
s[9]
main
count
3
ch
EOF
size
10
return value
3
return addr
main.3
read_into_array
State of memory just prior to returning from the call read_into_array()


s
s[0] 'W'
s[1] 'I'
s[2] 'N'
s[3]
s[9]
main
State of memory just after returning from the call read_into_array().
All local variables allocated for read_into_array() on stack may be assumed to be erased/de-allocated.
Only the stack for main() remains, that is, all local variables for main() remain.
Behold !!
The array s[] of main() has changed!
THIS DID NOT HAPPEN BEFORE!
WHAT DID WE DO DIFFERENTLY?
Ans: we passed the array s[] as a parameter!!

Parameter Passing

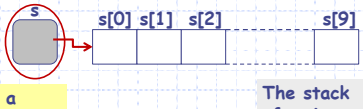
Basic steps:

1. Create new variables (boxes) for each of the formal parameters allocated on a fresh stack area created for this function call.
2. Copy values from actual parameters to the newly created formal parameters.
3. Create new variables (boxes) for each local variable in the called procedure. Initialize them as given.

Let us look at parameter passing more carefully.

```
int main()  {
  int s[10];
  read_into_array(s,10);
  …
```

```
int read_into_array
        (char t[], int size) {
    int ch;
    int count = 0;
    /* …   */
}
```



Array variables store address!!

s is an array. It is a variable and it has a box.

The value of this box is the address of the first element of the array.

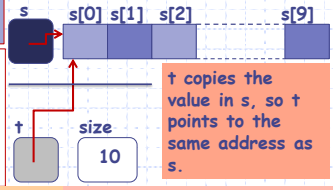
The stack of main just prior to call

Parameter Passing: Arrays

1. Create new variables (boxes) for each of the formal parameters allocated on a fresh stack created for this function call.
2. Copy values from actual parameters to the newly created formal paramters.

```
int main()  {
  char s[10];
  read_into_array(s,10);
  …
```

```
int read_into_array
        (char t[], int size) {
    int ch;
    int count = 0;
    /* …   */
}
```

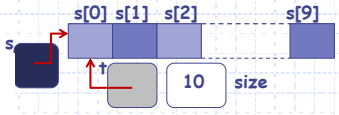


t copies the value in s, so t points to the same address as s.

s and t are the same array now, with two different names!!

s[0] and t[0] refer to the same variable, etc..

Implications of copying content of array variable during parameter passing



s is an array. In C an array is identified with a box whose value is the address of the first element of the array.

The value of s is copied into t. So the box corresponding to t has the same value as the box corresponding to s.

They both now contain the address of the first element of the array.

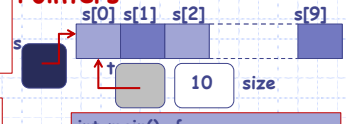
1. In the computer, an address is simply the value of a memory location.
2. For e.g., the value in the box for s would be the memory location of s[0].
3. When we draw figures, we will show this by an arrow.

Pointers

The arrow from inside box s to s[0] indicates that s stores address of s[0].

Referred to as :

s points to s[0], or,
s is a pointer to s[0].



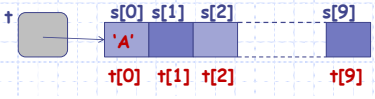

```
int main() {
  int s[10];
  read_into_array(s,10); }
```

```
int read_into_array
        (char t[], int size);
```

Passing an actual parameter array s to a formal parameter array t[] makes t now point to the first element of array s.

Since t is declared as char t[], t[0] is the box pointed to by t, t[1] refers to the box one char further from the box t[0], t[2] refers to the box that is 2 chars further from the box t[0] and so on…

Let us see this now.



- t[0] is the box whose address is stored in t. This is same as s[0].
- t[1] is the box next to (successor to) the box whose address is stored in t. This is the same as s[1].
- t[2] is the box 2 steps next to the box whose address is stored in t; this is same as s[2], etc..

Now suppose we change t[0] using
`t[0] = 'A';`
Later on, in main(), when we access s[0], we see that s[0] is 'A'.

The box is the same, but it has two names, s[0] in main() and t[0] in read_into_array()

Argument Passing: Array vs Simple Type

- When a basic datatype (such as int, char, float, etc) is passed to a function
  - a copy of the value is created in the memory space for that function,
  - after the function completes its execution, these values are lost.
- When an array is passed to a function
  - the address of the first element is copied,
  - any changes to the array elements are visible to the caller of the function.

Example: Dot Product

- Problem: write a function dot_product that takes as argument two integer arrays, a and b, and an integer, size, and computes the dot product of first size elements of a and b.
- Declaration of dot_product

```
int dot_product(int a[], int b[], int);
```

OR

```
int dot_product(int [], int [], int);
```

```
#include<stdio.h>
int dot_product (int[], int[], int);
int main(){
  int vec1[] = {2,4,1,7,-5,0, 3, 1};
  int vec2[] = {5,7,1,0,-3,8,-1,-2};
  printf("%d\n", dot_product(vec1, vec1, 8));
  printf("%d\n", dot_product(vec1, vec2, 8));
  return 0;
}
int dot_product (int a[], int b[], int size){
```

$$p = \sum_{i=1}^{size} (a_i \times b_i)$$

Convert to C

```
}
```

OUTPUT
105
49

```
#include<stdio.h>
int dot_product (int[], int[], int);
int main(){
  int vec1[] = {2,4,1,7,-5,0, 3, 1};
  int vec2[] = {5,7,1,0,-3,8,-1,-2};
  printf("%d\n", dot_product(vec1, vec1, 8));
  printf("%d\n", dot_product(vec1, vec2, 8));
  return 0;
}
int dot_product (int a[], int b[], int size){
  int p = 0, i;
  for(i=0;i<size; i++)
      p = p + (a[i]*b[i]);
  return p;
}
```

OUTPUT
105
49

Generating Prime Numbers

- Problem: Given a positive integer N, generate all prime numbers up to N.
- A Greek mathematician Eratosthenes came up with a simple but fast algorithm
- Sieve of Eratosthenes

Sieve of Eratosthenes

- On a piece of paper, write down all the integers starting from 2 till N.
- Starting from 2 strike off all multiples of 2, except 2.
- Next, find the first number that has not been struck and strike off all its multiples, except the number.
- Continue until you cannot strike out any more numbers.
- The numbers that have not been struck, are PRIMES.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

	2	3	4	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~

	2	3	4	~~5~~	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

Generating Prime Numbers using Sieve of Eratosthenes

- No more numbers can be marked. Algorithm terminates.
- Primes up to 100 are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.

Sieve of Eratosthenes: Program

```
int comp[10000]; // global array
void sieve(int n) {
  int i, j = 2;
  comp[0]=0; comp[1]=0;
  for (i=2; i<=n; i++)  comp[i] = 1;

  while (j <= n) {
     if (comp[j] == 0) { // composite
       j++; continue;
     }
     for (i= j*j; i<=n; i=i+j)
           comp[i] = 0;
     j++;
  }
}
```

```
int main() {
  int i, n;
  scanf("%d", &n);
  // check n < 10000

  sieve(n); // set primes

  for (i=2; i<=n; i++) {
      if (comp[i] == 1)
          printf("%d\n", i);
  }

  return 0;
}
```

In the previous slide, we had the statement:

`scanf("%d", &num[i]);`

What does &num[i] mean?

& is the "address-of" operator.

1. It can be applied to any defined variable.
2. It returns the location (i.e., address) of this variable in the program's memory.

[ ] is the array indexing operator, e.g., num[i].

&num[i] is made of two operators & and []. & num [i] gives the address of the array element num[i].

&num[i] is evaluated as **&(num[i])**.

& is applied to the result of applying the indexing operator [i] to num.

NOT as

(&num)[i] which would mean that first & is applied to num and [] operator is applied to &num

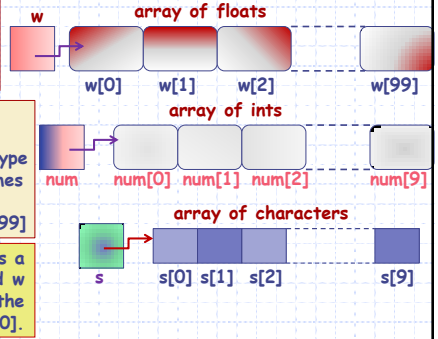
ESC101: Introduction to Computing

Strings

Recap about arrays

Basics: Arrays are defined as follows.

```
float w[100];
int num[10];
char s[10];
....
```



1. float w[100] defines 100 variables of type float. Its names are indexed: w[0],w[2],…w[99]
2. Also it defines a variable called w which stores the address of w[0].

How can we create an int array num[] and initialize it to:

Method 1 `int num[] = {-2,3,5,-7,19, 103, 11};`

1. Initial values are placed within curly braces separated by commas.
2. The size of the array **need not be specified**. It is set to the number of initial values provided.
3. Array elements are assigned in sequence in the index order. First constant is assigned to array element [0], second constant to [1], etc..

Method 2 `int num[10] = {-2,3,5, -7, 19, 103, 11};`

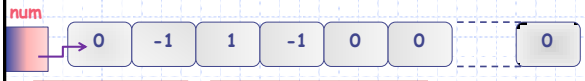
Specify the array size. **size must be at least equal to the number of initialized values.** Array elements assigned in index order. Remaining elements are set to 0.

Recommended method: array size determined from the number of initialization values.

```
int num[] = {-2,3,5,-7,19,103,11};
```

Is this correct? `int num[100] ={0,-1,1,-1};`

YES! Creates num as an array of size 100. First 4 entries are initialized as given. num[4] … num[99] are set to 0.



Is this correct? NO! it won't compile!

```
int num[6] = {-2,3,5,-7,19,103,11};
```

Why?

1. num is declared to be an int array of size 6 but 7 values have been initialized.
2. Number of initial values must be less than equal to the size specified.

Initialization values could be constants or **constant expressions**. Constant expressions are expressions built out of constants.

```
int num[] = { 109, 7+3, 25*1023 };
```

Type of each initialization constant should be promotable/demote-able to array element type.

E.g., `int num[] = { 1.09, 'A', 25.05};`

Float constants 1.09 and 25.05 downgraded to int

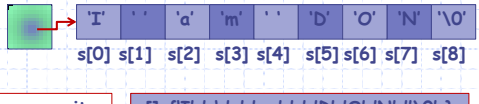
Would this work?

```
int curr = 5;
int num[] = { 2, curr*curr+5};
```

YES! ANSI C allows constant expressions **AND simple expressions** for initialization values. "Simple" is compiler dependent.

Character array initialization

Character arrays may be initialized like arrays of any other type. Suppose we want the following char array.



We can write: `s[]={'I',' ','a',' m',' ','D','O','N',"\0' };`

BUT! C allows us to define **string constants**. We can also write: `s[] ="I am DON";`

1. "I am DON" is a **string** constant. The '\0' character (also called NULL char) is automatically added to the end.
2. Strings constants in C are specified by enclosing in double quotes e.g. "I am a string".

Printing strings

We have used string constants many times. Can you recall?

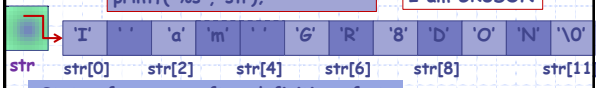
printf and scanf: the first argument is always a string.

1. printf("The value is %d\n", value);
2. scanf("%d", &value);

Strings are printed using %s option.

E.g. 1
```
printf("%s", "I am DON");
```
Output
I am DON

E.g. 2
```
char str[]="I am GR8DON";
printf("%s", str);
```
Output
I am GR8DON



State of memory after definition of str in E.g. 2. Note the NULL char added in the end.

This NULL char is not printed.

Strings

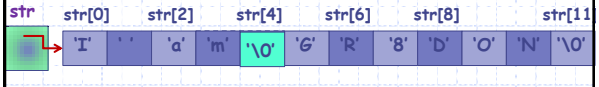
Consider the fragment.

```
char str[]="I am GR8DON";
str[4] = '\0';
printf("%s", str);
```

This defines a constant string, i.e., character array terminated by, but not including, '\0'.

What is printed?

Let us trace the memory state of str[].



Output
I am

1. A string is a sequence of characters terminated by '\0'. This '\0' is not part of the string.
2. There may be non-null characters after the first occurrence of '\0' in str[]. They are not part of the string str[] and don't get printed by printf("%s", str);

So do we lose the chars after the first '\0' ? Where did they go?

Of course not, they remain right where they were. They were not printed because we used %s in printf. Let's take a look.


```
char str[]="I am GR8DON";
str[4]='\0';
printf("%s", str);
```

Output
I am

```
int i;
for (i=0; i < 11; i++) {
    putchar(str[i]);
}
```

Output
I amGR8DON

The character '\0' may be printed differently on screen depending on terminal settings.

Reading a String (scanf)

- Placeholder: %s
- Argument: Name of character array.
- No & sign before character array name.
- Input taken in a manner similar to numeric input.
- scanf skips whitespaces.
  - There are three basic whitespace characters in C : space, newline ('\n') and tab ('\t').
  - Any combination of the three basic whitespace characters is a whitespace.

Reading a String (scanf)

- Starts with the first non-whitespace character.
- Copies the characters into successive memory cells of the character array variable.
- When a whitespace character is reached, scanning stops.
- scanf places the null character at the end of the string in the array variable.

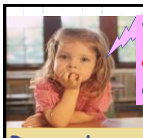

```
#include <stdio.h>

int main() {
 char str1[20], str2[20];

 scanf("%s",str1);
 scanf("%s",str2);

 printf("%s + %s\n",
              str1, str2);

 return 0;
}
```

INPUT
IIT Kanpur

OUTPUT
IIT + Kanpur

INPUT
I am DON

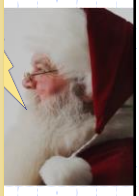
OUTPUT
I + am



Why is there no & when we read character array?

Remember parameter passing?

- A simple variable can not be modified from inside a function call (Recall **swap()** function)
- However, Arrays can be modified from inside a function call (Recall **read_into_array()** function)
- Similarly, **scanf** can also "modify" arrays directly. Since string is just an array of chars, **&** is not required.
- More on this when we do **pointers**



NULL character '\0'

- ASCII value 0.
- Marks the end of the string.
- C needs this to be present in every string in order to differentiate between a character array and a string.
- Size of char array holding the string ≥ 1 + length of string
  - Buffer overflow otherwise!

NULL character '\0'

- What happens if no '\0' is kept at the end of string?
  - '\0' is used to detect end of string, for example in printf("%s", str).
  - Without '\0', such functions will keep reading array elements beyond the array bound (out of bound access).
  - We can get an incorrect result or a Runtime Error.

Reading a line as an input

- scanf, when used with the %s placeholder, reads a block of non-whitespace characters as a string.
- What if we want to read a line as a string?
- We will define our own function to read a line.
- **EXERCISE:** Take as input a line (that ends with the newline character) into a character array as a string.


```
#include <stdio.h>
// read a line into str, return length
int read_line(char str[]) {
   int c, i=0;
   c = getchar();
   while (c != '\n' && c != EOF) {
      str[i] = c;
      c = getchar();
      i++;
   }
   str[i] = '\0'; // we want a string!
   return i; // i is the length of the string
}
```

Buffer overflow possible


```
#include <stdio.h>
// read a line into str, return length.
//   maximum allowed length is limit
int read_line(char str[], int limit ) {
   int c, i=0;
   c = getchar();
   while (c != '\n' && c != EOF) {
      str[i] = c;
      c = getchar();
      i++;
      if (i == limit-1) break;
   }
   str[i] = '\0'; // we want a string!
   return i; // i is the length of the string
}
```

Safer version!

Copying one String to Other

- We can not copy content of one string variable to other using assignment operator

```
char str1[] = "Hello";
char str2[] = str1;
```

WRONG

  - This is true for any array variable.
- We need to do element-wise copying

String Copy

str_copy(char dest[], char src[]);

- Arguments: Two strings: dest and src.
- Copy contents of src into dest.
- We assume that dest is declared with size at least as large as src.
- Note the use of '\0' for loop termination

```
void str_copy(char dest[], char src[]) {
   int i;
   for (i = 0; src[i] != '\0'; i++)
   { dest[i] = src[i]; }
   dest[i] = '\0';
}
```


Comparing Two Strings

- Lexicographical Ordering
  - A string str1 is said to be lexicographically smaller than another string str2 if the first character, where the strings differ, is smaller in str1.
- Examples:
  - "cap" is smaller than "cat".
  - "mat" is smaller than "matter".
- Order of words in a Dictionary

String Comparison

- We will write a function that compares two strings lexicographically:
  str_compare (char str1[], char str2[])
- Arguments: Two strings str1 and str2
- Return value:
  - 0 if the strings are equal,
  - -1 if str1 is "smaller",
  - 1 if str2 is "smaller".
- Assumption: The strings contain letters of one case (either capital or small).

Code for str_compare

```
int str_compare(char str1[], char str2[]){
  int i=0;
  while (str1[i]==str2[i] && str1[i]!='\0')
    i++;
  if (str1[i] == str2[i])
    return 0;
  else if (str1[i] < str2[i])
    return -1;
  else //str2 < str1
    return 1;
}
```

At this point, only way str1[i] and str2[i] can be same is if both are '\0' => strings are same

At this point, **i** is either
- the **first** index where str1 and str2 **differ**, or
- the **end** of str1

At this point, since the first differing characters are such that str1[i] < str2[i], => str1 is smaller

Other string functions

- Return length of a string.
- Concatenates one string with another.
- Search for a substring in a given string.
- Reverse a string
- Find first/last/k-th occurrence of a character in a string
  - … and more
- Case sensitive/insensitive versions

string.h

- Header File with Functions on Strings
- strlen(s): returns length of string s (without '\0')
- strcpy(d, s): copies s into d
- strcmp(s1, s2): return an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.
- strcat(d, s): appends s at the end of d ('\0' is moved to the end of result)

string.h

- strncpy(d, s, n)
- strncmp(s1, s2, n)
- strncat(d, s, n)
  - restrict the function to "n" characters at most (argument n is an integer)
- strcasecmp, strncasecmp: case insensitive comparison
- Many more utility functions.
- All functions depend on '\0' as the end-of-string marker.

ESC101: Introduction to Computing

Multidimensional Arrays

Why Multidimensional Arrays?

- Marks of 800 students in 5 subjects each.
- Distance between cities
- Sudoku
  - All the above require 2D arrays
- Properties of points in space (Temperature, Pressure etc.)
- Mathematical Plots
  - > 2D arrays

Multidimensional Arrays

Multidimensional arrays are defined like this:

```
double mat[5][6];
```
OR
```
int mat[5][6];
```
OR
```
float mat[5][6];
```
etc.

The definition states that mat is a 5 X 6 matrix of doubles (or ints or floats). It has 5 rows, each row has 6 columns, each entry is of type double.

mat

2.1	1.0	-0.11	-0.87	31.5	11.4
-3.2	-2.5	1.678	4.5	0.001	1.89
7.889	3.333	0.667	1.1	1.0	-1.0
-4.56	-21.5	1.0e7	-1.0e-9	1.0e-15	-5.78
45.7	26.9	-0.001	1000.09	1.0e15	1.0

Accessing matrix elements-I

1. The (i,j) th member of mat is accessed as mat[i][j]. Note the slight difference from the matrix notation in maths.
2. The row and column numbering each start at 0 (not 1).
3. The following program prints the input matrix.

```
void print_matrix(double mat[5][6]) {
    int i,j;
    for (i=0; i < 5; i=i+1) {    /* prints the ith row i = 0..4. */
      for (j=0; j < 6; j = j+1) {
         printf("%f ", mat[i][j]);   /* In each row, prints each of the six columns  j=0..5 */
      }
      printf("\n");  /* prints a newline after each row */
    }
}
```

Accessing matrix elements-II

1. Code for reading the matrix from the terminal.
2. The address of the i,j th matrix element is &mat[i][j].
3. This works without parentheses since the array indexing operator [] has higher precedence than &.

```
void read_matrix(double mat[5][6]) {
   int i,j;
   for (i=0; i < 5; i=i+1) {
      for (j=0; j < 6; j = j+1) {
        scanf("%f ", &mat[i][j]);
      }
   }
}
```

/* read the ith row i = 0..4. */

/* In each row, read each of the six columns j=0..5 */

scanf with %f option will skip over whitespace.

So it really doesn't matter whether the entire input is given in 5 rows of 6 doubles in a row or all 30 doubles in a single line, etc..

Initializing 2 dimensional arrays

We want a[4][3] to be this 4 X 3 int matrix.

1	2	3
4	5	6
7	8	9
0	1	2

Initialize as

```
int  a[][3] = {
            {1,2,3},
            {4,5,6},
            {7,8,9},
            {0,1,2}
    };
```

Initialization rules:
1. Most important: values are given row-wise, first row, then second row, so on.
2. Number of columns must be specified.
3. Values in each row is enclosed in braces {…}.
4. Number of values in a row may be less than the number of columns specified. Remaining col values set to 0 (or 0.0 for double, '\0' for char, etc.)

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

gives this matrix for a:

1	0	0
2	3	0
3	4	5

Accessing matrix elements

```
void read_matrix(double mat[5][6]) {
   int i,j;
   for (i=0; i < 5; i=i+1) {
      for (j=0; j < 6; j = j+1) {
        scanf("%f ", &mat[i][j]);
      }
   }
}
```

/* read the ith row i = 0..4. */

/* In each row, read each of the six columns j=0..5 */

Could I change the formal parameter to mat[6][5]? Would it mean the same? Or mat[10][3]?

That would not be correct. It would change the way elements of mat are addressed. Let us see some examples.

Passing two dimensional arrays as parameters

Write a program that takes a two dimensional array of type double [5][6] and prints the sum of entries in each row.

```
void marginals(double mat[5][6]) {
   int i,j; double rowsum;
   for (i=0; i < 5; i=i+1) {
       rowsum = 0.0;
       for (j=0; j < 6; j = j+1) {
         rowsum = rowsum+mat[i][j];
       }
      printf("%f ", rowsum);
    }
}
```

Question?

But suppose we have only read the first 3 rows out of the 5 rows of mat. And we would like to find the marginal sum of the first 3 rows.

Answer:

That's easy, we can take an additional parameter nrows and run the loop for i=0..(nrows-1) instead of 0..5.

The slightly generalized program would be:

```
void marginals(double mat[5][6], int nrows) {
   int i,j; double rowsum;
   for (i=0; i < nrows; i=i+1) {
       rowsum = 0.0;
       for (j=0; j < 6; j = j+1) {
         rowsum = rowsum+mat[i][j];
       }
      printf("%f ", rowsum);
    }
}
```

In parameter double mat[5][6], *C completely ignores the number of rows* 5. It is only interested in the number of cols: 6.

We declared mat to be of type double [5][6]. Does this mean that nrows should be <= 5? We are not checking for it!

Let's see more examples…

The following program is exactly identical to the previous one.

```
void marginals(double mat[ ][6], int nrows)
 {
   int i,j; int rowsum;
   for (i=0; i < nrows; i=i+1) {
       rowsum = 0.0;
       for (j=0; j < 6; j = j+1) {
         rowsum = rowsum+mat[i][j];
       }
      printf("%f ", rowsum);
    }
}
```

1. Why? because *C does not care about the number of rows, only the number of cols.*
2. And why is that? We'll have to understand 2-dim array addressing.

This means that the above program works with a k X 6 matrix where k could be passed for nrows.

Example…

```
void marginals(double mat[ ][6], int nrows);
void main() {
     double mat[9][6];
    /* read the first 8 rows into mat */
     marginals(mat,8);
}
```

Example calls for marginals

```
void marginals(double mat[ ][6], int nrows);
void main() {
    double mat[9][6];
   /* read 9 rows into mat */
    marginals(mat,10);
}
```

UNSAFE

The 10th row of mat[9][6] is not defined. So we may get a segmentation fault when marginals() processes the 10th row, i.e., i becomes 9.

As with 1 dim arrays, allocate your array and stay within the limits allocated.

Why is # of columns required?

- The memory of a computer is a 1D array!
- 2D (or >2D) arrays are "flattened" into 1D to be stored in memory
- In C (and most other languages), arrays are flattened using Row-Major order
  - In case of 2D arrays, knowledge of number of columns is required to figure out where the next row starts.
  - Last n-1 dimensions required for n-D arrays

Row Major Layout

mat[3][5]

Layout of mat[3][5] in memory

- for a 2D array declared as mat[M][N], cell [i][j] is stored in memory at location i*N + j from start of mat.
- for k-D array arr[N_1][N_2]…[N_k], cell [i_1][i_2]…[i_k] will be stored at location

  $i_k + N_k*(i_{k-1} + N_{k-1}*(i_{k-2} + (\dots + N_2*i_1) \dots))$

Coin Collection: Practice Problem

You have an $n \times n$ grid with a certain number of coins in each cell of the grid. The grid cells are indexed by (i, j) where $0 \leq i, j \leq n-1$.

For example, here is a 3x3 grid of coins:

	0	1	2
0	5	8	2
1	3	6	9
2	10	15	2

Coin Collection: Problem Statement

- You have to go from cell (0, 0) to (n-1, n-1).
- Whenever you pass through a cell, you collect all the coins in that cell.
- You can only move right or down from your current cell.

Goal: Collect the maximum number of coins.

Consider the example grid

5	8	2
3	6	9
10	15	2

In an $n \times n$ grid, how many such paths are possible?

There are many ways to go from (0,0) to (n-1,n-1)

5	8	2
3	6	9
10	15	2

Total = 35

5	8	2
3	6	9
10	15	2

Total = 25

5	8	2
3	6	9
10	15	2

Total = 31

5	8	2
3	6	9
10	15	2

Total = 30

5	8	2
3	6	9
10	15	2

Total = 23

5	8	2
3	6	9
10	15	2

Total = 36

Max = 36

Building a Solution

- We cannot afford to check every possible path and find the maximum.
  - Why?
- Instead we will iteratively try to build a solution.

Solution Idea

- Consider a portion of some matrix

- What is the maximum number of coins that I can collect when I reach the blue cell?
  - This number depends only on the maximum number of coins that I can collect when I reach the two green cells!
  - Why? Because I can only come to the blue cell via one of the two green cells.

Solution Idea

```
Max-coins (bluecell) =
        max(Max-coins (greencell-1),
            Max-coins (greencell-2))
          + No. of coins (bluecell))
```

Solution Idea

- Let a(i,j) be the number of coins in cell(i,j)
- Let coin(i,j) be the maximum number of coins collected when travelling from (0,0) to (i,j).
- Then,

coin(i,j) = max(coin(i,j-1), coin(i-1,j)) + a(i,j))

Implementation

- Use an additional two dimensional array, whose (i,j)-th cell will store the maximum number of coins collected when travelling from (0,0) to (i,j).
- Fill this array one row at a time, from left to right.
- When the array is completely filled, return the (n-1, n-1)-th element.

Implementation: Boundary Cases

- To fill a cell of this array, we need to know the information of the cell above and to the left of the cell.
- What about elements in the top most row and left most column?
  - Cell in top row: no cell above
  - Cell in leftmost column: no cell on left
- Before starting with the other elements, we will fill these first.

```
int coin_collect(int a[][100], int n){
  int i,j, coins[100][100];

  coins[0][0] = a[0][0]; //initial cell

  for (i=1; i<n; i++) //first row
    coins[0][i] = coins[0][i-1] + a[0][i];

  for (i=1; i<n; i++) //first column
    coins[i][0] = coins[i-1][0] + a[i][0];

  for (i=1; i<n; i++) //filling up the rest of the array
    for (j=1; j<n; j++)
      coins[i][j] = max(coins[i-1][j], coins[i][j-1])
                    + a[i][j];

  return coins[n-1][n-1]; //value of last cell
}
```



```
int max(int a, int b){
  if (a>b) return a;
  else return b;
}

int main(){
  int m[100][100],i,j,n;

  scanf("%d", &n);
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      scanf("%d", &m[i][j]);

  printf("%d\n", coin_collect(m,n));
  return 0;
}
```


Array of Strings

- 2D array of char.
- Recall
  - Strings are character arrays that end with a '\0'
  - To display a string we can use printf with the %s placeholder.
  - To input a string we can use scanf with %s. Only reads non-whitespace characters.

Array of Strings: Example

- Write a program that reads and displays the name of few cities of India

```
int main(){
  const int ncity = 4;
  const int lencity = 10;
  char city[ncity][lencity];
  int i;

  for (i=0; i<ncity; i++){
    scanf("%s", city[i]);
  }

  for (i=0; i<ncity; i++){
    printf("%d %s\n", i, city[i]);
  }
  return 0;
}
```

INPUT
Delhi
Mumbai
Kolkata
Chennai

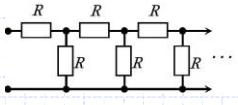
city

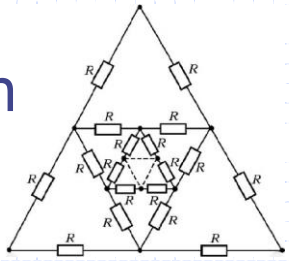
D	e	l	h	i	\0				
M	u	m	b	a	i	\0			
K	o	l	k	a	t	a	\0		
C	h	e	n	n	a	i	\0		

OUTPUT
0 Delhi
1 Mumbai
2 Kolkata
3 Chennai

ESC101: Introduction to Computing

Recursion





Recursion

- A function calling itself, *directly* or *indirectly,* is called a *recursive function.*
  - The phenomenon itself is called recursion
- Examples:
  - Factorial:

```
0! = 1
n! = n * (n-1)!
```

  - Even and Odd:

```
Even(n) = (n == 0) || Odd(n-1)
Odd(n)  = (n != 0) && Even(n-1)
```

Recursive Functions: Properties

- The arguments change between the recursive calls

  5! = 5 * 4! = 5 * 4 * 3! = …

- Change is towards a case for which solution is known (base case)
- There must be one or more base cases:

  0! is 1

  Odd(0) is false

  Even(0) is true

Recursion and Induction

When programming recursively, think inductively

- Mathematical induction for the natural numbers
- Structural induction for other recursively-defined types (to be covered later!)

Recursion and Induction

When writing a recursive function,

- Write down a clear, concise *specification* of its behavior,
- Give an *inductive proof* that your code satisfies the specification.

Constructing Recursive functions: Examples

Write a function search(int a[], int n, key) that performs a sequential search of the array a[0..n-1] of int. Returns 1 if the key is found, otherwise returns 0.

How should we start? We have to think of the function search() in terms of search applied to a smaller array. Don't think in terms of loops…think recursion.

Here's a possibility ….

search(a,n,key)

Base case: If n is 0, then, return 0.

Otherwise: /* n > 0 */
1. compare last item, a[n-1], with key.
2. if a[n-1] == key, return 1.
3. search in array a, up to size n-1.
4. return the result of this "smaller" search.

a → search(a,10,3)

31	4	10	35	59	31	3	25	35	11

Either 3 is a[9]; or search(a,10,3) is same as the result of search for 3 in the array starting at a and of size 9.


```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[n-1] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

Let us do a quick trace.

a → E.g., (0) search(a,5,10)

31	4	10	35	59

a[4] is 59, not 10. call search(a,4,10)

a → (1) search(a,4,10)

31	4	10	35	59

a[3] is 35, calls search(a,3,10)

a → (2) search(a,3,10)

31	4	10	35	59

a[2] is 10, return 1

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[n-1] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

Let us do another quick trace.

E.g., (0) search(a,5,3)

31	4	10	35	59

a[4] is 59, not 3. call search(a,4,3)

(1) search(a,4,3)

31	4	10	35	59

a[3] is 35, calls search(a,3,3)

(2) search(a,3,3)

31	4	10	35	59

a[2] is 10, calls search(a,2,3)

(3) search(a,2,3)

31	4	10	35	59

a[1] is 4, calls search(a,1,3)

(4) search(a,1,3)

31	4	10	35	59

a[0] is 31, calls search(a,0,3)

(5) search(a,0,3) returns 0


```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

search(a,5,3)

31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	
search(a,1,3)	search(a,2,3)	search.5	
search(a,0,3)	search(a,1,3)	search.5	

recursion exits here

A state of the stack


```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

search(a,5,3)

31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	
search(a,1,3)	search(a,2,3)	search.5	
search(a,0,3)	search(a,1,3)	search.5	0

recursion exits here

A state of the stack

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

a → search(a,5,3)

31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	
search(a,1,3)	search(a,2,3)	search.5	0

state of the stack


```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

a → search(a,5,3)

31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.5	
search(a,2,3)	search(a,3,3)	search.5	0

A state of the stack


```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

a → search(a,5,3)

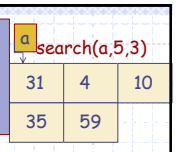
31	4	10
35	59	

Stack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	0

A state of the stack

```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```

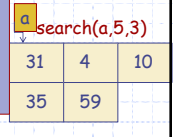


Stack

function	called by	return address	return value
search(a,5,3)	main()	---	0

A state of the stack


```
1. int search(int a[], int n, int key) {
2.     if (n==0) return 0;
3.     if (a[0] == key)  return 1;
4.     return search(a,n-1,key);
5. }
```



search(a,5,3) returns 0. Recursion call stack terminates.

Searching in an Array

- We can have other recursive formulations
- Search1: search (a, start, end, key)
  - Search key between a[start]…a[end]

```
if start > end, return 0;
if a[start] == key, return 1;
return search(a, start+1, end, key);
```

Disclaimer: Algorithm not tested for boundary cases

Searching in an Array

- One more recursive formulations
- Search2: search (a, start, end, key)
  - Search key between a[start]…a[end]

```
if start > end, return 0;
mid = (start + end)/2
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
          || search(a, mid+1, end, key);
```

Disclaimer: Algorithm not tested for boundary cases

Estimating the Time taken

- Two types of operations
  - Function calls
  - Other operations (call them simple operations)
- Assume each simple operation takes fixed amount of time (1 unit) to execute
  - Really a very crude assumption, but will simplify calculations
- Time taken by a function call is proportional to the number of operations performed by the call before returning.

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - Let T(n) denote the time taken by search on an array of size n.
  - Line 1 takes 1 unit (or 2 units if you consider if check and return as two operations)
  - Line 2 takes 1 unit (or 3 units if you consider if check, array access and return as three operations)
  - But what about line 3?

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - What about line 3?
  - Remember the assumption: *Let T(n) denote the time taken by search on an array of size n.*
  - Line 3 is searching in n-1 sized array => takes T(n-1) units
  - But what about the value of T(n) ?

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - But what about the value of T(n) ?
  - Looking at the body of search, and the information we gathered on previous slides, we can come up with a recurrence relation:

$$T(n) = T(n-1) + C$$

  - We need to solve the recurrence to get the estimate of time

Estimating the Time taken

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - Solution to the recurrence?

$$T(n) = T(n-1) + C$$

$$T(n) = Cn$$

  - The worst case run time of Search1 is proportional to the size of array
    - Bigger the array, slower the search
  - What is the best case run time?
  - Which one is more important to consider?

Estimating the Time taken

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
        || search(a, mid+1, end, key);
```

- Search2
  - Recurrence?

$$T(n) = T(n/2) + T(n/2) + C$$

  - Solution?

$$T(n) \propto n$$

  - The worst case run time of Search2 is also proportional to the size of array
    - Can we do better?

Can we search Faster?

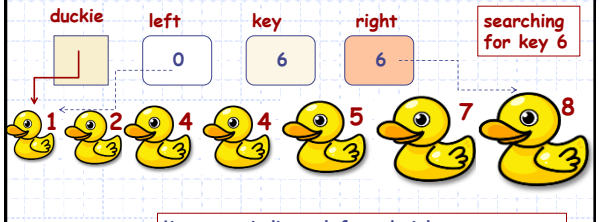
- Yes, provided the elements in the array are sorted
  - in either ascending or descending order

Let us take an example. We have an array of numbers, sorted in non-descending order.

```
int duckie [] = {1,2,4,4,5,6,7};
```

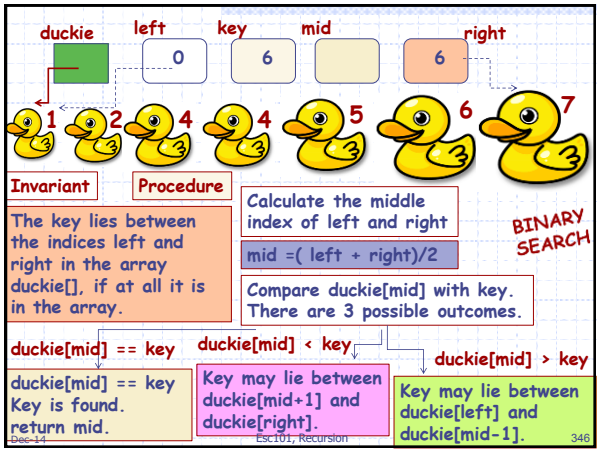
some numbers can be repeated, like 4 in duckie[]

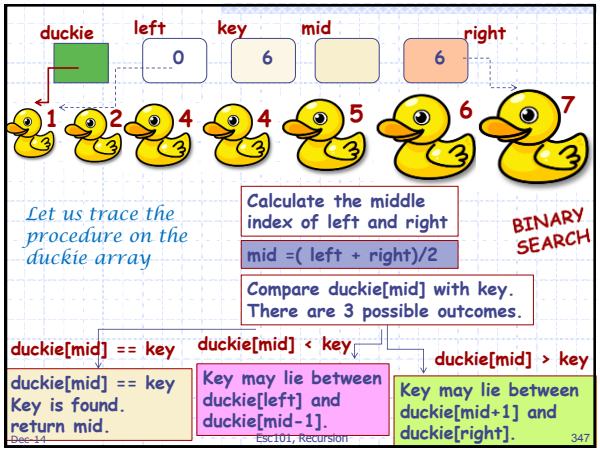
To illustrate the idea, consider searching for the number **6** in the array.

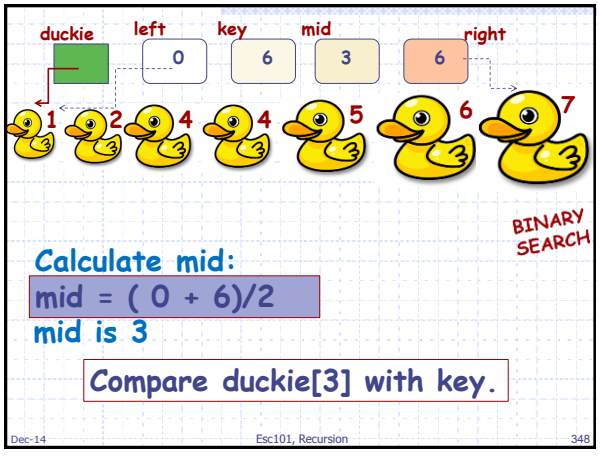


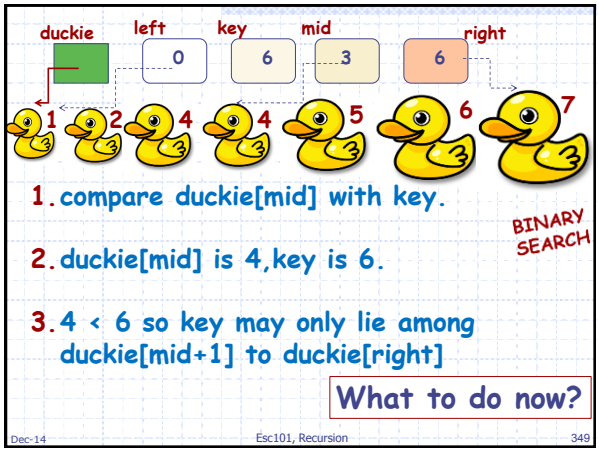
Initialization Keep two indices, left and right. Initially left is 0 and right is the rightmost index in the array. Here right is 6.

Invariant The key that is being searched for lies in between the indices left and right in the array duckie[] (both ends included), if at all it is in the array.


duckie
left
key
mid
right
0
6
6
1
2
4
4
5
6
7
Invariant
Procedure
The key lies between the indices left and right in the array duckie[], if at all it is in the array.
Calculate the middle index of left and right
mid =(left + right)/2
BINARY SEARCH
Compare duckie[mid] with key. There are 3 possible outcomes.
duckie[mid] == key
duckie[mid] < key
duckie[mid] > key
duckie[mid] == key
Key is found.
return mid.
Key may lie between duckie[mid+1] and duckie[right].
Key may lie between duckie[left] and duckie[mid-1].
Dec-14
Esc101, Recursion
346


duckie
left
key
mid
right
0
6
6
1
2
4
4
5
6
7
Let us trace the procedure on the duckie array
Calculate the middle index of left and right
mid =(left + right)/2
BINARY SEARCH
Compare duckie[mid] with key. There are 3 possible outcomes.
duckie[mid] == key
duckie[mid] < key
duckie[mid] > key
duckie[mid] == key
Key is found.
return mid.
Key may lie between duckie[left] and duckie[mid-1].
Key may lie between duckie[mid+1] and duckie[right].
Dec-14
Esc101, Recursion
347


duckie
left
key
mid
right
0
6
3
6
1
2
4
4
5
6
7
BINARY SEARCH
Calculate mid:
mid = (0 + 6)/2
mid is 3
Compare duckie[3] with key.
Dec-14
Esc101, Recursion
348



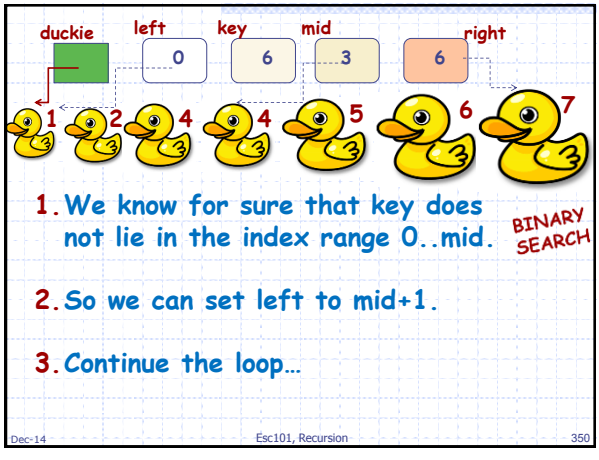
1. compare duckie[mid] with key.

BINARY SEARCH

2. duckie[mid] is 4, key is 6.

3. 4 < 6 so key may only lie among duckie[mid+1] to duckie[right]

What to do now?

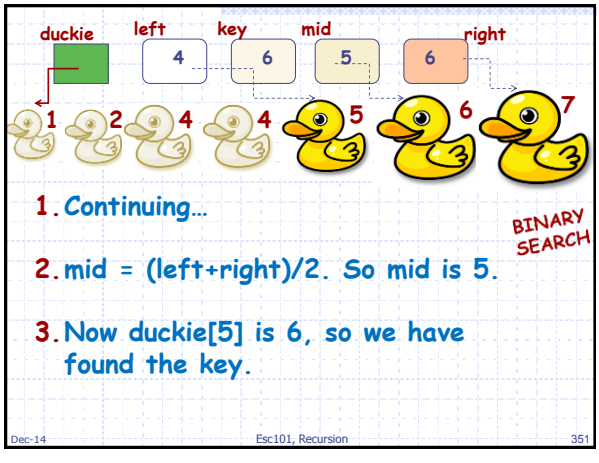


1. We know for sure that key does not lie in the index range 0..mid.

BINARY SEARCH

2. So we can set left to mid+1.

3. Continue the loop…

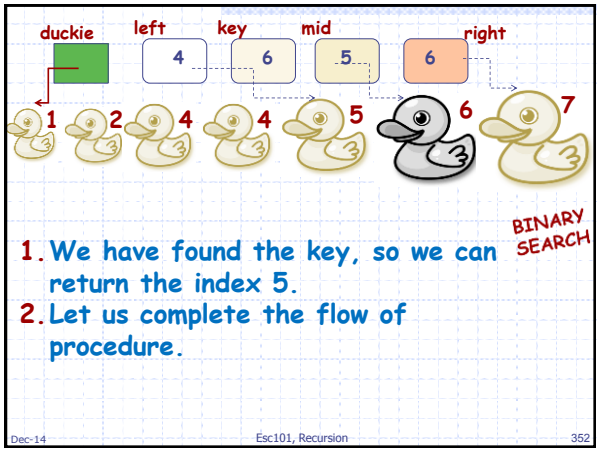


1. Continuing…

BINARY SEARCH

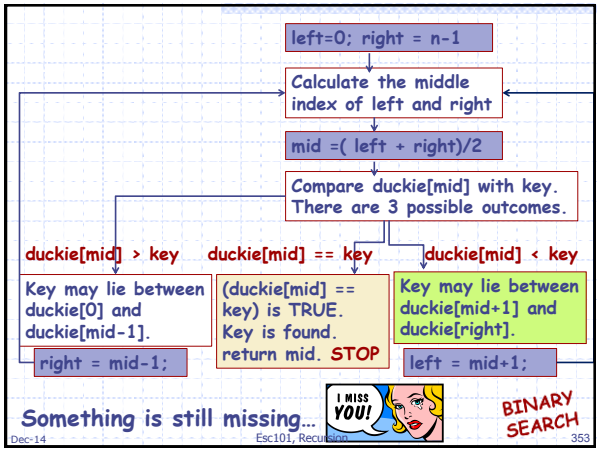
2. mid = (left+right)/2. So mid is 5.

3. Now duckie[5] is 6, so we have found the key.



BINARY SEARCH

1. We have found the key, so we can return the index 5.
2. Let us complete the flow of procedure.

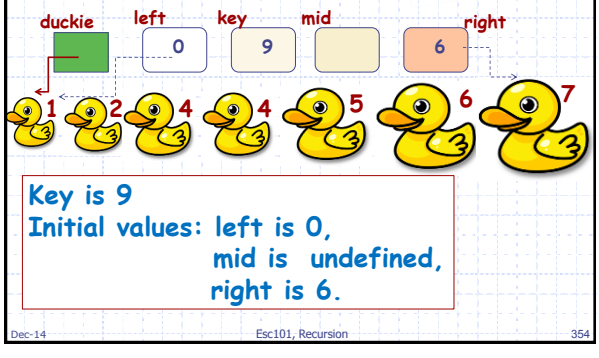


Something is still missing...

BINARY SEARCH

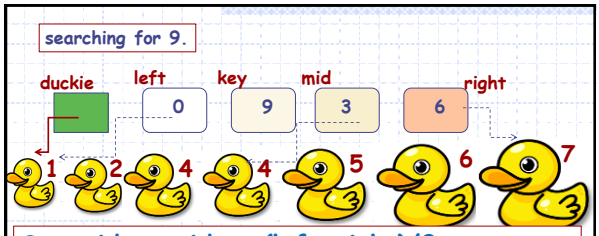
Let us search for 9 in this array.

an example of a search that does not succeed



Key is 9
Initial values: left is 0,
mid is undefined,
right is 6.

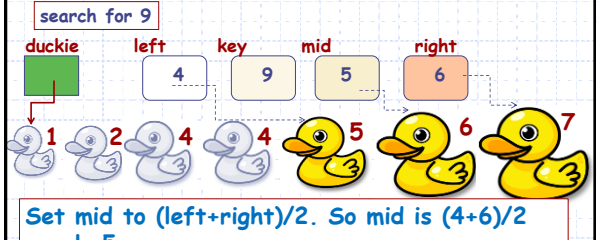
searching for 9.



Set mid to mid = (left+right)/2.
mid is 3. Compare duckie[mid] with key.
duckie[mid] is 4, key is 9 and 4<9.

So we have to move right, meaning left is set to mid+1.
So left will be 4.

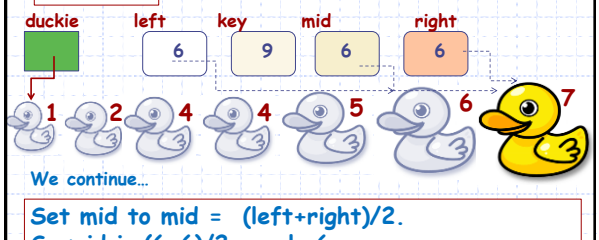
search for 9



Set mid to (left+right)/2. So mid is (4+6)/2 equals 5.
Compare duckie[mid] with key.
duckie[mid] is 6, key is 9 and 6<9.

So, we have to move right again.
Set left to mid +1, so left becomes 6.

search for 9

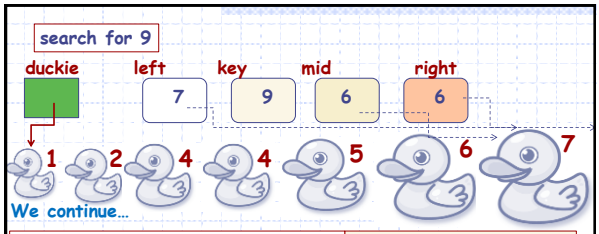


We continue…

Set mid to mid = (left+right)/2.
So mid is (6+6)/2 equals 6.
Compare duckie[mid] with key. duckie[mid] is 7, key is 9 and 7<9.

So, we have to move right again.
Set left to mid +1, so left becomes 7.

search for 9



We continue…

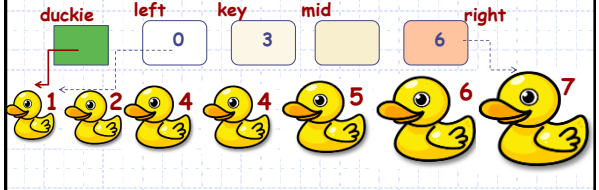
left is 7, right is 6.
The two ends have crossed over.
So the item is not there in the array!

NOT FOUND!

By invariant, item is there between duckie[left] and duckie [right] so long as left <= right.

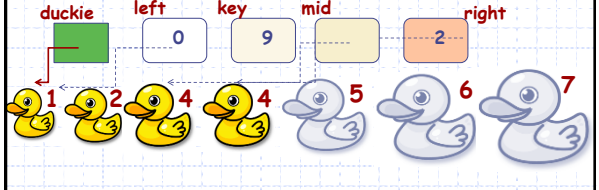
OK, so another condition when the loop terminates is left > right. Is there any other termination condition? Can we search for 3?

Searching for 3 in this array.



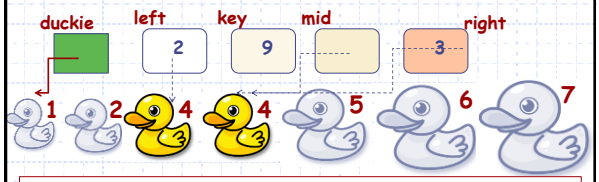
1. left is 0, right is 6.
2. mid is (0+6)/2 which is 3.
3. duckie[mid] is 4, key is 3, so we have to move left.
4. right will be set to mid-1.

Searching for 3 in this array.



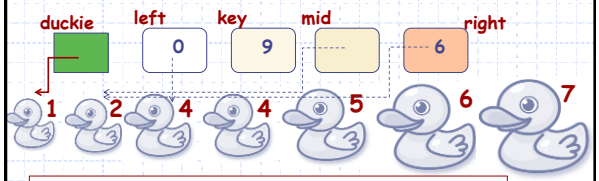
1. left is 0, right is 2.
2. mid is (0+2)/2 which is 1.
3. duckie[mid] is 2 , key is 3, so we have to move right.
4. left will be set to mid+1.

Searching for 3 in this array.



1. previously mid was 1, left is mid+1 which is 2, right is 2.
2. Now mid is (2+2)/2 which is 2.
3. duckie[mid] is 4 , key is 3, so we have to move left.
4. right will be set to mid-1,So right will be 1.

Searching for 3 in this array.



1. left is 2, right is 1.
2. Left and right have crossed over, NOT FOUND!

Binary Search for Sorted Arrays

- binsearch(a, start, end, key)
  - Search key between a[start]…a[end], where a is a sorted (non-decresing) array

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
if (a[mid] > key)
        binsearch(a, start, mid-1, key);
else  binsearch(a, mid+1, end, key);
```

Wait, isn't this same as search2?

- Lets look closely

```
int search2(a, start, end, key) {
  if start > end, return 0;
  mid = (start + end)/2 ;
  if a[mid]==key, return 1;
  return search2(a, start, mid-1, key)
                || search2(a, mid+1, end, key);
}
```

In worst case, **Both** search2 may fire. But, **only ONE** of the two binsearch will fire.

```
int binsearch(a, start, end, key) {
  if start > end, return 0;
  mid = (start + end)/2 ;
  if a[mid]==key, return 1;
  if (a[mid] > key)
        binsearch(a, start, mid-1, key);
  else binsearch(a, mid+1, end, key);
}
```


Estimating the Time taken

- binsearch
  - Recurrence?

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
if (a[mid] > key)
      binsearch(a, start, mid-1, key);
else  binsearch(a, mid+1, end, key);
```

$$T(n) = T(n/2) + C$$

  - Solution? $T(n) \propto \log n$

  - The worst case run time of binsearch is proportional to the log of the size of array
    - Much faster than Search/Search1/Search2 for large arrays
    - Remember: It works for SORTED arrays only

Some problems related to binary search

Given a sorted array, find the left-most (right-most) occurrence of a key.

Given a key, find its **successor (predecessor)** in the array. That is, find the smallest (largest) value larger (smaller) than the given key that occurs in the array.

You are not allowed to:

1. Find an occurrence of the key and then sequentially go left (for predecessor) or go right (for successor).
2. **Why?**
3. because this may have linear complexity. Solve the problem as efficiently as binary search, that is, number of comparisons is bounded by constant times log(n).
4. Also the given key may not exist in the array.

Recursion vs Iteration

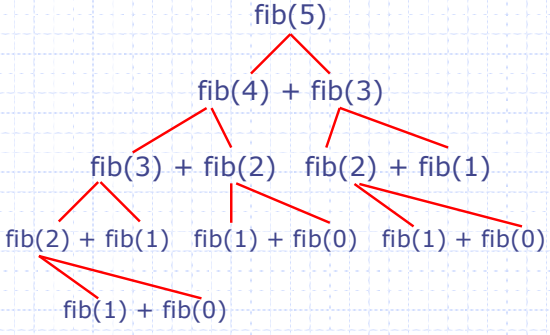
```
int fib(int n)
{
  int first = 0, second = 1;
  int next, c;
  if (n <= 1)
     return n;
  for ( c = 1; c < n ; c++ ) {
     next = first + second;
     first = second;
     second = next;
  }
  return next;
}
```

The recursive program is closer to the definition and easier to read.

But very very inefficient

```
int fib(int n)
{
  if ( n <= 1 )
     return n;
  else
     return fib(n-1) + fib(n-2);
}
```

Recursive fib



Rec. fib: How fast #calls grow

```
#include<stdio.h>
int count = 0; /*Global: #fib calls */

int fib(int n) {
  count = count+1;
  if (n<=1) return n;
  else return fib(n-1) + fib(n-2);
}

int main() {
  int num, res;
  for (num=5; num<=30; num=num+5) {
    count = 0; /* reset the count*/
    res = fib(num);
    printf("%d, %d\n", res, count);
  }
  return 0;
}
```

num	fib(num)	Count
5	5	15
10	55	177
15	610	1973
20	6765	21891
25	75025	242785
30	832040	2692537

Recursion: Summary

- Advantages
  - Elegant. Solution is cleaner.
  - Fewer variables.
  - Once the recursive definition is figured out, program is easy to implement.
- Disadvantages
  - Debugging can be considerably more difficult.
  - Figuring out the logic of the recursive function is not easy sometimes.
  - Can be inefficient (requires more time and space), if not implemented carefully.

Around Easter 1961, a course on ALGOL 60 was offered … It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who **included recursion in their language** and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.

- **The Emperor's Old Clothes, C. A. R. Hoare, ACM Turing Award Lecture, 1980**

Puzzle: Tower of Hanoi

- We have three poles: **A**, **B** and **C**; and **N** discs that fit onto the poles.
- The discs differ in size and are initially arranged on pole **A** in order of size
  - **largest** disc at the **bottom** to **smallest** disc at the **top**.
- The task is to move the stack of discs to pole **C** while obeying the following rules:
  - Move **only one disc** at a time.
  - **Never** place a disc on a smaller one.

Tower of Hanoi

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

Tower of Hanoi ..2

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

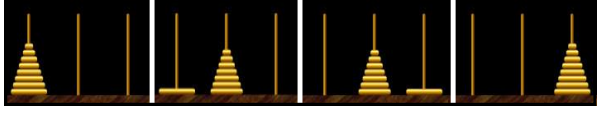
Tower of Hanoi ..3

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

Tower of Hanoi ..4

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html


```
// move n disks From A to C using B as auxx
void hanoi(int n, char A, char C, char B) {
  if (n==0) { return; } // nothing to move!!
   // recursively move n-1 disks
   // from A to B using C as auxx
   hanoi(n-1, A, B, C);
   // atomic move of a single disk
   printf("Move 1 disk from %c to %c\n", A, C);
   // recursively move n-1 disks
   // from B to C using A as auxx
   hanoi(n-1, B, C, A);
}
```



OUTPUT for hanoi(4, 'A', 'C', 'B')

Move 1 disk from A to B
Move 1 disk from A to C
Move 1 disk from B to C
Move 1 disk from A to B
Move 1 disk from C to A
Move 1 disk from C to B
Move 1 disk from A to B
Move 1 disk from A to C
Move 1 disk from B to C
Move 1 disk from B to A
Move 1 disk from C to A
Move 1 disk from B to C
Move 1 disk from A to B
Move 1 disk from A to C
Move 1 disk from B to C

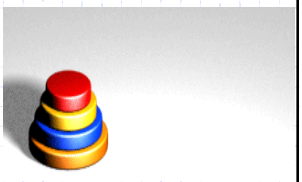


Image Source: http://upload.wikimedia.org/wikipedia/commons/6/60/Tower_of_Hanoi_4.gif

ESC101: Introduction to Computing

Sorting

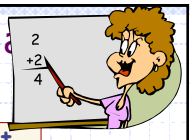
Sorting

- Given a list of integers (in an array), arrange them in ascending order.
  - Or descending order

INPUT ARRAY	5	6	2	3	1	4
OUTPUT ARRAY	1	2	3	4	5	6

- Sorting is an extremely important problem in computer science.
  - A common problem in everyday life.
  - Example:
    - Contact list on your phone.
    - Ordering marks before assignment of grades.

What's easy to do in a Sorted Array?



Clearly, searching for a key is fast.

Rank Queries: find the k^{th} largest/smallest value.
Quantile: 90%ile—the key value in the array such that 10% of the numbers are larger than it.

40	50	55	60	70	75	80	85	90	92

Marks in an exam: sorted

90 percentile : 90
80 percentile : 85
10 percentile: 40
50 percentile: 70
(also called median)

with

- inserting a new element while preserving the sorted structure.
- deleting an existing element (while preserving the sorted structure.
- In both cases, there may be need to shift elements to the right or left of the index corresponding to insertion or deletion.

40	50	55	60	70	75	80	85	90	92

Example: Insert 65.
1. Find index where 65 needs to be inserted

40	50	55	60	65	70	75	80	85	90	92

2. Shift right from index 5 to create space.

3. Insert 65

May have to shift n-1 elements in the worst case.

Sorting

- Many well known sorting Algorithms
  - Selection sort
  - Quick sort
  - Merge sort
  - Bubble sort
  - …
- Special cases also exist for specific problems/data sets
- Different runtime
- Different memory requirements

Selection Sort

- Select the largest element in your array and swap it with the first element of the array.
- Consider the sub-array from the second element to the last, as your current array and repeat Step 1.
- Stop when the array has only one element.
  - Base case, trivially sorted

Selection Sort: Pseudo code

```
selection_sort(a, start, end) {
   if (start == end) /* base case, one elt => sorted */
       return;

   idx_max = find_idx_of_max_elt(a, start, end);
   swap(a, idx_max, start);
   selection_sort(a, start+1, end);
}
```

```
swap(a, i, j) {
   tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
}
```

```
main() {
   arr[] = { 5, 6, 2, 3, 1, 4 };
   selection_sort(arr, 0, 5);
   /* print arr */
}
```


Selection Sort: Properties

- Is the pseudo code iterative o recursive?
- What is the estimated run tim when input array has **n** eleme
  - for swap **Constant**
  - for find_idx_of_max_elt $\propto n$
  - for selection_sort **On next slide**
- **Practice**: Write C code for iterative and recursive versions of selection sort.

Selection Sort: Time Estimate

- Recurrence

$$T(n) = T(n-1) + k_1 \times n + k_2$$

- Solution

$$T(n) \propto n(n+1)$$

- Or simply

$$T(n) \propto n^2$$

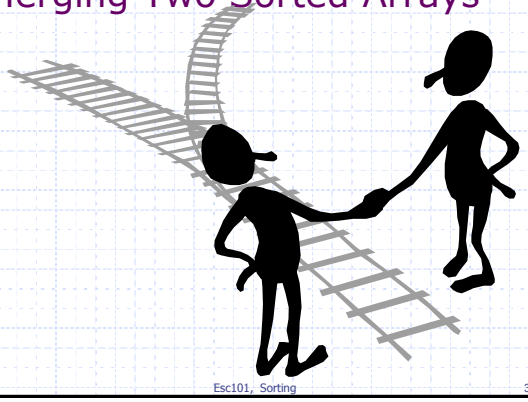
Selection sort runs in time proportional to the square of the size of the array to be sorted.

Can we do better? YES WE CAN

Merging Two Sorted Arrays

- Input: Array A of size n & array B of size m.
- Create an empty array C of size n + m.
- Variables i , j and k
  - array variables for the arrays A, B and C resp.
- At each iteration
  - compare the i^{th} element of A (say u) with the j^{th} element of B (say v)
  - if u is smaller, copy u to C; increment i and k,
  - otherwise, copy v to C; increment j and k,

Merging Two Sorted Arrays



Time Estimate

- Number of steps ∝ 3(n + m).
  - The constant 3 is not very important as it does not vary with different sized arrays.
- Now suppose A and B are halves of an array of size n (both have size n/2).
- Number of steps = 3n.

$$T(n) \propto n$$

MergeSort

- Merge function can be used to sort an array
  - recursively!
- Given an array C of size n to sort
  - Divide it into Arrays A and B of size n/2 each (approx.)
  - Sort A into A' using MergeSort
  - Sort B into B' using MergeSort
  - Merge A' and B' to give C' ≡ C sorted

Recursive calls. Base case? n <= 1

- Can we reduce #of extra arrays (A', B', C')?


```
/*Sort ar[start, …, start+n-1] in place */
void merge_sort(int ar[], int start, int n) {
   if (n>1) {
      int half = n/2;
      merge_sort(ar, start, half);
      merge_sort(ar, start+half, n-half);
      merge(ar, start, n);
   }
}
```

```
int main() {
  int arr[]={2,5,4,8,6,9,8,6,1,4,7};
  merge_sort(arr,0,11);
  /* print array */
  return 0;
}
```

```
void merge(int ar[], int start, int n) {
  int temp[MAX_SZ], k, i=start, j=start+n/2;
  int lim_i = start+n/2, lim_j = start+n;
  for(k=0; k<n; k++) {
     if ((i < lim_i) && (j < lim_j)) {// both active
        if (ar[i] <= ar[j]) { temp[k] = ar[i]; i++; }
        else { temp[k] = ar[j]; j++; }
     } else if (i == lim_i) // 1st half done
        { temp[k] = ar[j]; j++; } // copy 2nd half
     else // 2nd half done
        { temp[k] = ar[i]; i++; } // copy 1st half
  }
  for (k=0; k<n; k++)
     ar[start+k]=temp[k]; // in-place
}
```

Time Estimate

`void merge_sort(int a[], int s, int n) {`	T(n)
`  if (n>1) {`	C
`    int h = n/2;`	C
`    merge_sort(a, s, h);`	T(n/2)
`    merge_sort(a, s+h, n-h);`	T(n-n/2)≈T(n/2)
`    merge(a, s, n);`	≈ 4n
`  }`	
`}`	

Time Estimate

T(n) = 2T(n/2) + 4n

= 2(2T(n/4) + 4n/2) + 4n = 2^2T(n/4) + 8n

= 2^2(2T(n/8) + 4n/4) + 4n = 2^3T(n/8) + 12n

= … // keep going for k steps

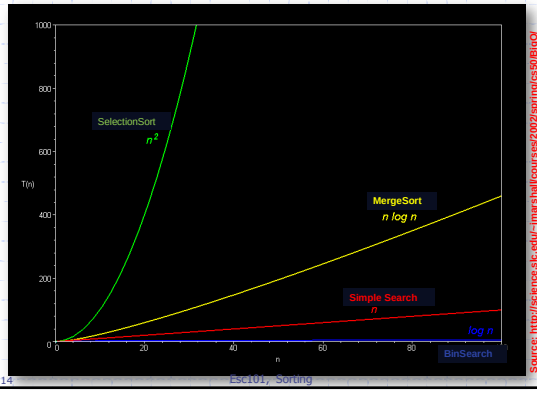
= 2^kT(n/2^k) + k*4n

Assume n = 2^k for some k. Then,

T(n) = n*T(1) + 4n*$\log_2$n

$$T(n) \propto n \log_2 n$$

Time Estimates…



ESC101: Introduction to Computing

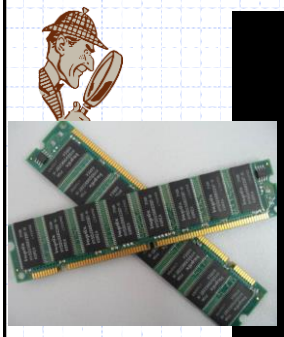
Pointers

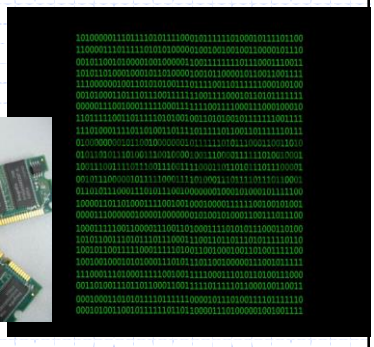
Pointer: Dictionary Definition

point·er (poin'tər)
n.

1. One that directs, indicates, or points.
2. A scale indicator on a watch, balance, or other measuring instrument.
3. A long tapered stick for indicating objects, as on a chart or blackboard.
4. Any of a breed of hunting dogs that points game, typically having a smooth, short-haired coat that is usually white with black or brownish spots.
5.
   a. A piece of advice; a suggestion.
   b. A piece of indicative information: *interest rates and other pointers in the economic forecast.*
6. *Computer Science* A variable that holds the address of a core storage location.
7. *Computer Science* A symbol appearing on a display screen in a GUI that lets the user select a command by clicking with a pointing device or pressing the enter key when the pointer symbol is positioned on the appropriate button or icon.
8. Either of the two stars in the Big Dipper that are aligned so as to point to Polaris.

Pointers and





Simplified View of Memory

- "Array" of blocks
- Each block can hold a byte (8-bits)
- "char" stored in 1 block
- "int" (32-bit) stored in 4 consecutive blocks
- Finite number of blocks
  - Limited by the capacity of (Virtual) Memory
  - Blocks are addressable – $[0 \ldots 2^N-1]$

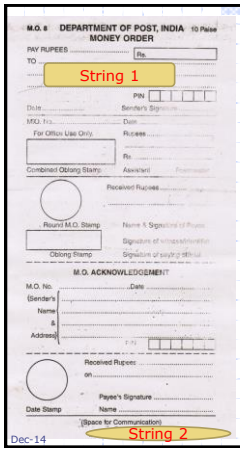
1004000	'A'
1004001	'E'
1004002	'I'
1004003	'O'
1004004	'U'
1004005	
1004006	
1004007	
1004008	1024
1004009	
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	

Simplified View of Memory

- Content of the 4-blocks starting at address 1004012
  - ✓1004001
- Without knowing the context it is not possible to determine the significance of number 1004001
  - ✓It could be an integer value 1004001
  - ✓It could be the "location" of the block that stores 'E'

How do we decide what it is?

1004000	'A'
1004001	'E'
1004002	'I'
1004003	'O'
1004004	'U'
1004005	
1004006	
1004007	
1004008	1024
1004009	
1004010	
1004011	
1004012	1004001
1004013	
1004014	
1004015	



String 1: Address of the recipient

String 2: Message from sender to recipient

- Sender can send own address in String 2, but it is still treated as message
- Postman doesn't get confused (generally ☺)

"Type" helps us disambiguate.

What if there is a room number Y9391 in a BIIIIIIG Hostel (26 wings, 10000 rooms in each wing)?

FIRST HALL OF R

R THE MONTH OF SEPTEMBER

		Y 9373	1415	516
		Y 9374	1591	518
		Y 9378	1592	520
		Y 9379	3774	522
262	A 159	Y 9380	1624	524
263	E 117	Y 93		
264	A 207	Y 93		
265	E 212	Y 93		
266	D 314	Y 93		
267	A 311	Y 9386	1475	534
268	E 112	Y 9388	3530	536
269	E 117	Y 9391	3622	538
270	D 203	Y 9394	3667	540
271	C 223	Y 9395	1592	542
272	G 108	Y 9398	1401	544
273	D 318	Y 9399	1766	546

Room Address

Roll Number

"Type" helps us disambiguate.

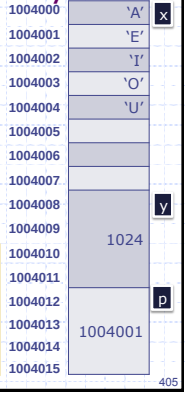
Simplified View of Memory

- In programming also, "Type" helps us decide whether 1004001 is an integer or a pointer to block containing 'E' (or something else)

```
#include<stdio.h>
int main() {
        char x[5] = {'A', 'E', 'I', 'O', 'U'};
        int y = 1024;
        char *p = x+1;
        ...
}
```

```
#include<stdio.h>
int main() {
        char x[5] = {'A', 'E', 'I', 'O', 'U'};
        int y = 1024;
        int p = 1004001;
        ...
}
```

Declaration of a pointer to char box



What is a Pointer

- **Pointer**: A **special type** of variable that contains an address of a memory location.
- Think of a pointer as a new data type (a new kind of box) that holds memory addresses.
- Pointers are almost always associated with the type of data that is contained in the memory location.
  - For example, an integer pointer is a memory location that contains an integer.
  - Character pointer, float pointer
  - Even pointer to pointer (more on this later …)

too much theory. Give examples please

Banti

OK.
Let me take you on a journey of Pointers in ~~SEA~~ C

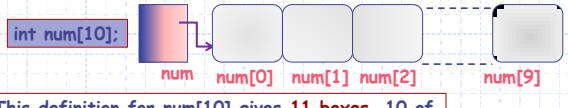
Owlie

Remember Arrays?

The memory allocated to array has two components:

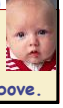
A consecutively allocated segment of memory boxes of the same type, and

A box with the same name as the array. This box holds the address of the base (i.e., first) element of the array.

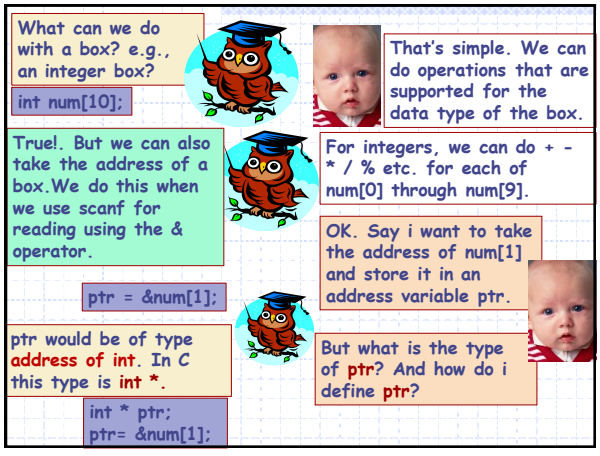


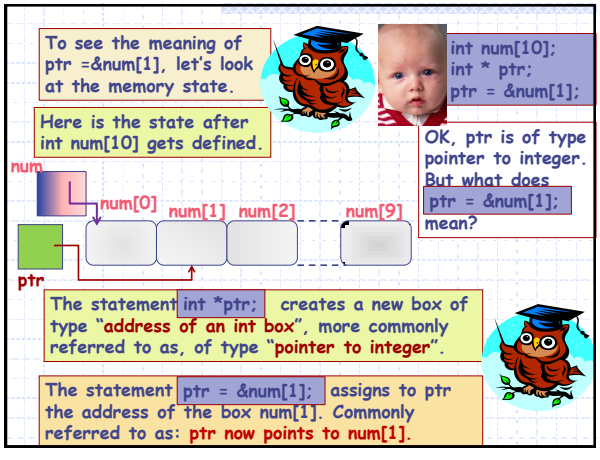
This definition for num[10] gives **11 boxes**, 10 of type int, and 1 of type address of an int box.

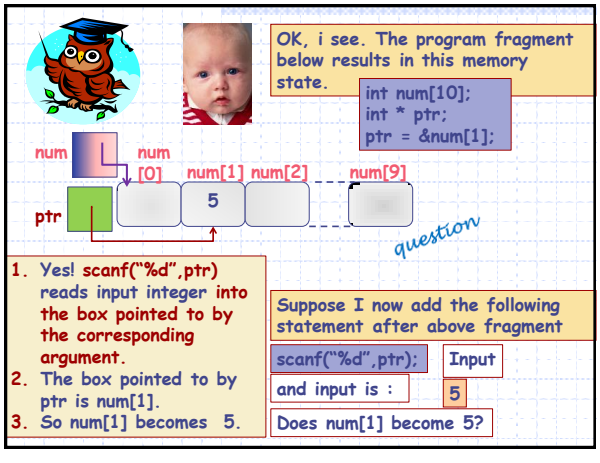
Hmm…

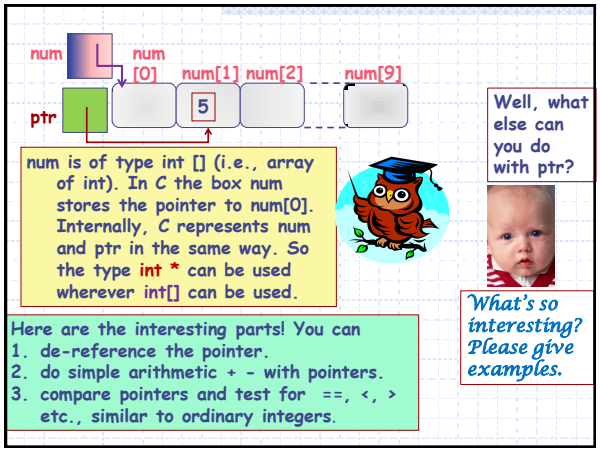


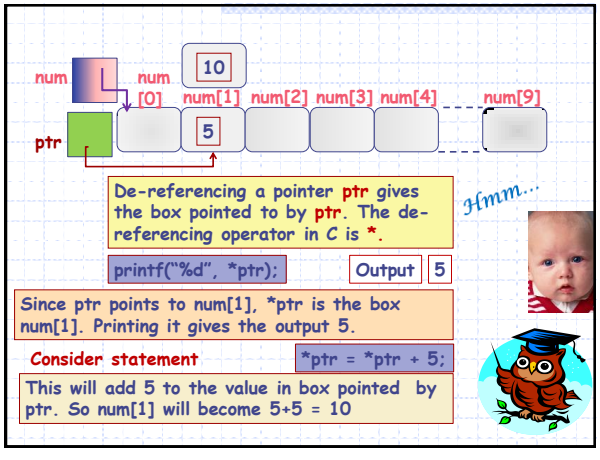
1. We represent the **address of a box x** by an **arrow to the box x**. So **addresses** are referred to as **pointers**.
2. The contents of an address box is a pointer to the box whose address it contains. e.g., num points to num[0] above.

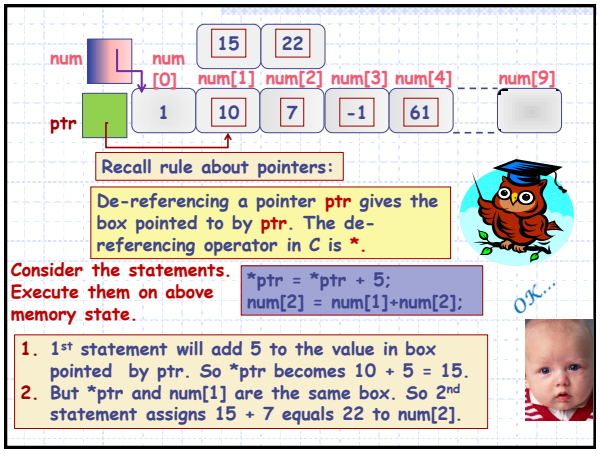

What can we do with a box? e.g., an integer box?
int num[10];
That's simple. We can do operations that are supported for the data type of the box.
True!. But we can also take the address of a box.We do this when we use scanf for reading using the & operator.
For integers, we can do + - * / % etc. for each of num[0] through num[9].
OK. Say i want to take the address of num[1] and store it in an address variable ptr.
ptr = &num[1];
But what is the type of ptr? And how do i define ptr?
ptr would be of type address of int. In C this type is int *.
int * ptr;
ptr= &num[1];


To see the meaning of ptr =&num[1], let's look at the memory state.
int num[10];
int * ptr;
ptr = &num[1];
Here is the state after int num[10] gets defined.
OK, ptr is of type pointer to integer. But what does ptr = &num[1]; mean?
num
num[0]
num[1]
num[2]
num[9]
ptr
The statement int *ptr; creates a new box of type "address of an int box", more commonly referred to as, of type "pointer to integer".
The statement ptr = &num[1]; assigns to ptr the address of the box num[1]. Commonly referred to as: ptr now points to num[1].


OK, i see. The program fragment below results in this memory state.
int num[10];
int * ptr;
ptr = &num[1];
num
num [0]
num[1]
num[2]
num[9]
5
ptr
question
1. Yes! scanf("%d",ptr) reads input integer into the box pointed to by the corresponding argument.
2. The box pointed to by ptr is num[1].
3. So num[1] becomes 5.
Suppose I now add the following statement after above fragment
scanf("%d",ptr);
Input
and input is :
5
Does num[1] become 5?


num
num [0]
num[1] num[2]
num[9]
ptr
5
num is of type int [] (i.e., array of int). In C the box num stores the pointer to num[0]. Internally, C represents num and ptr in the same way. So the type int * can be used wherever int[] can be used.
Well, what else can you do with ptr?
What's so interesting? Please give examples.
Here are the interesting parts! You can
1. de-reference the pointer.
2. do simple arithmetic + - with pointers.
3. compare pointers and test for ==, <, > etc., similar to ordinary integers.


num
num [0]
10
num[1] num[2] num[3] num[4]
num[9]
ptr
5
De-referencing a pointer ptr gives the box pointed to by ptr. The de-referencing operator in C is *.
Hmm...
printf("%d", *ptr);
Output 5
Since ptr points to num[1], *ptr is the box num[1]. Printing it gives the output 5.
Consider statement
*ptr = *ptr + 5;
This will add 5 to the value in box pointed by ptr. So num[1] will become 5+5 = 10


num
num [0]
15
22
num[1] num[2] num[3] num[4]
num[9]
ptr
1
10
7
-1
61
Recall rule about pointers:
De-referencing a pointer ptr gives the box pointed to by ptr. The de-referencing operator in C is *.
Consider the statements. Execute them on above memory state.
*ptr = *ptr + 5;
num[2] = num[1]+num[2];
OK...
1. 1st statement will add 5 to the value in box pointed by ptr. So *ptr becomes 10 + 5 = 15.
2. But *ptr and num[1] are the same box. So 2nd statement assigns 15 + 7 equals 22 to num[2].

num num[0] num[1] num[2] num[9]

1 15 22

16

ptr

Consider the statement

num[2]= *num + *ptr;

Is it a legal statement? What would be the result?

Hmm… seems legal— 'cause

1. num can be thought to be of type int *, and, ptr is of type int *.
2. So *num is of type int, which is 1 and *ptr is of type int with value 15
3. So num[2] is set to 16.

Let me show you some cool stuff: pointer arithmetic.

Let int num[] = {1,22,16,-1,23};

num num[0] num[1] num[2] num[3] num[4]

1 22 16 -1 23

num+1 points to integer box just next to the integer box pointed to by num. Since arrays were consecutively allocated, the integer box just next to num[0] is num[1].

So num+1 points to num[1]. Similarly, num+2 points to num[2], num + 3 points to num[3], and so on.

Can you tell me the output of this printf statement?

printf("%d %d %d", *(num+1), *(num+2),*(num+3));

Okay, What's cool?

Hmm.. Output would be

22 16 -1

Let us predict the output of some simple code fragments.

owlie

char str[] ="BANTI is a nice girl";
char *ptr = str + 6; /*initialize*/
printf("%s",ptr);

What is printed by the above program?

First let us draw the state of memory.

str[0] str[5] str[10] str[15]

'B' 'A' 'N' 'T' 'I' ' ' 'i' 's' ' ' 'a' ' ' 'n' 'i' 'c' 'e' ' '

str ptr

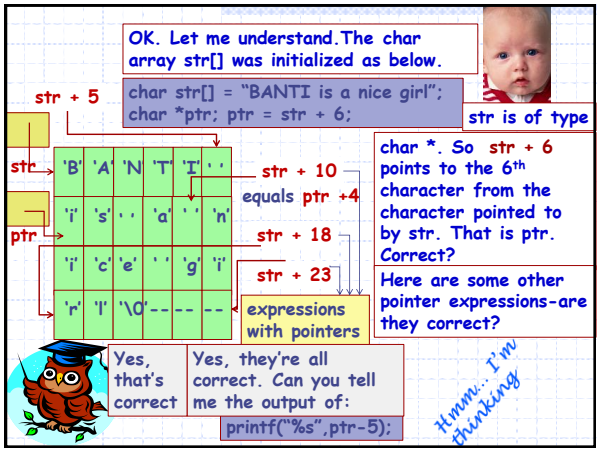
'g' 'i' 'r' 'l' '\0'

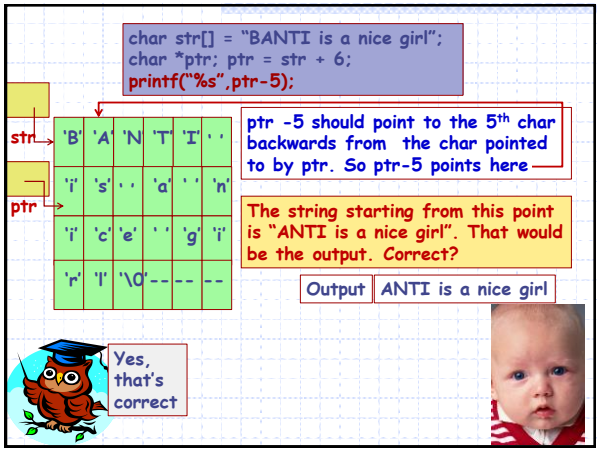
str[16] str[20]

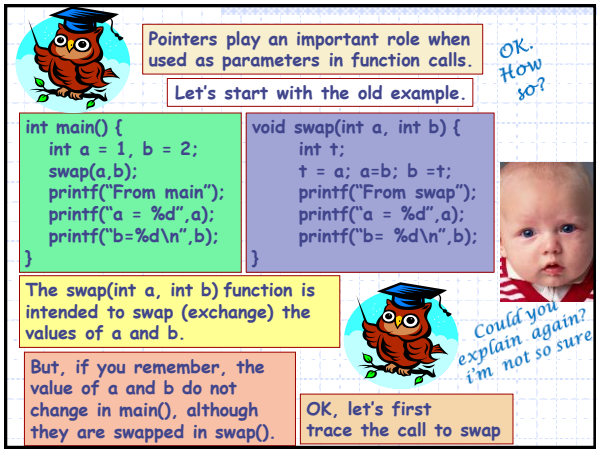
ptr points to str[6]. printf prints the string starting from str[6], which is

Output is a nice girl

Hmm.. OK


OK. Let me understand.The char array str[] was initialized as below.
char str[] = "BANTI is a nice girl";
char *ptr; ptr = str + 6;
str is of type
str + 5
str
ptr
'B' 'A' 'N' 'T' 'I' ' '
'i' 's' ' ' 'a' ' ' 'n'
'i' 'c' 'e' ' ' 'g' 'i'
'r' 'l' '\0'
str + 10
equals ptr +4
str + 18
str + 23
expressions with pointers
char *. So str + 6 points to the 6th character from the character pointed to by str. That is ptr. Correct?
Here are some other pointer expressions-are they correct?
Yes, that's correct
Yes, they're all correct. Can you tell me the output of:
printf("%s",ptr-5);
Hmm... I'm thinking


char str[] = "BANTI is a nice girl";
char *ptr; ptr = str + 6;
printf("%s",ptr-5);
str
ptr
'B' 'A' 'N' 'T' 'I' ' '
'i' 's' ' ' 'a' ' ' 'n'
'i' 'c' 'e' ' ' 'g' 'i'
'r' 'l' '\0'
ptr -5 should point to the 5th char backwards from the char pointed to by ptr. So ptr-5 points here
The string starting from this point is "ANTI is a nice girl". That would be the output. Correct?
Output
ANTI is a nice girl
Yes, that's correct


Pointers play an important role when used as parameters in function calls.
Let's start with the old example.
OK. How so?
int main() {
int a = 1, b = 2;
swap(a,b);
printf("From main");
printf("a = %d",a);
printf("b=%d\n",b);
}
void swap(int a, int b) {
int t;
t = a; a=b; b =t;
printf("From swap");
printf("a = %d",a);
printf("b= %d\n",b);
}
The swap(int a, int b) function is intended to swap (exchange) the values of a and b.
Could you explain again? i'm not so sure
But, if you remember, the value of a and b do not change in main(), although they are swapped in swap().
OK, let's first trace the call to swap

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
    int t;
    t = a; a=b; b =t;
    printf("From swap ");
    printf("a= %d",a);
    printf("b= %d\n",b);
}
```

STACK

a 1

b 2

main()

return address main.3

swap()

a 2

b 1

1 t

Output: From swap a= 2 b= 1

Now swap() returns:
1. Return address is line 3 of main(). Program counter is set to this location.
2. Stack for swap() is deleted.

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
    int t;
    t = a; a=b; b =t;
    printf("From swap ");
    printf("a = %d",a);
    printf("b = %d\n",b);
}
```

STACK

a 1

b 2

main()

Output: From swap a = 2 b = 1

Returning back to main(),we resume execution from line 3.

Hmm...

But the variables a and b of main() are unchanged from what they were before the call to swap(). They are printed as is.

Changes made by swap() remained local to the variables of swap(). They did not propagate back to main().

```
int main() {
    int a = 1, b = 2;
    swap(a,b);
    printf("From main");
    printf(" a = %d",a);
    printf("b = %d",b);
}
```

```
void swap(int a, int b) {
    int t;
    t = a; a=b; b =t;
    printf("From swap ");
    printf("a = %d",a);
    printf("b = %d\n",b);
}
```

a 1

b 2

Output: From swap a = 2 b = 1
From main a = 1 b = 2

OK, i remember now

1. Passing int/float/char as parameters does not allow passing "back" to calling function.
2. Any changes made to these variables are lost once the function returns.

Pointers will help us solve this problem!

Here is the changed program.

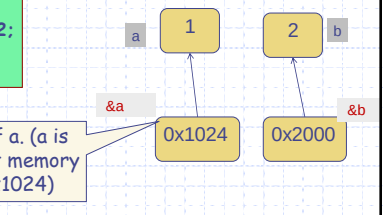
```
void
swap(int *ptra, int *ptrb)
{
    int t;
    t = *ptra;
    *ptra= *ptrb;
    *ptrb =t;
}
```

```
int main() {
   int a = 1, b = 2;
   swap(&a, &b);
   printf("a=%d, b=%d",
          a, b);
   return 0;
}
```

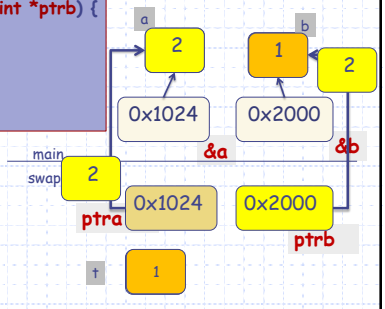
1. The function swap() uses pointer to integer arguments, int *a and int *b.
2. The main() function calls swap(&a,&b), i.e., passes the addresses of the ints it wishes to swap.

Tracing the swap function

```
int main() {
   int a = 1, b = 2;
   swap(&a, &b);
}
```




```
void swap(int *ptra, int *ptrb) {
   int t;
   t = *ptra;
   *ptra= *ptrb;
   *ptrb =t;
}
```



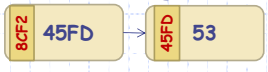
Homework ☺

Will the following code perform swap correctly?

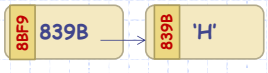
```
void swap(int *ptra, int *ptrb) {
    int *ptrt;
    ptrt = ptra;
    ptra= ptrb;
    ptrb =ptrt;
}
```

Pointers: Visual Representation

- Typically represented by box and arrow diagram



- x is an int variable that contains the value 53.
- Address of x is 45FD.
- px is a pointer to int that contains address of x.



- y is an char variable that contains the character 'H'.
- Address of y is 839B.
- py is a pointer to char that contains address of y.

We are showing addresses for explanation only.
Ideally, the program should not depend on actual addresses.

Pointer to a pointer

- If we have a pointer P to some memory cell, P is also stored somewhere in the memory.
- So, we can also talk about address of block that stores P.


```
int x = 53;
int *px = &x;
int **ppx = &px;
```

Size of Datatypes

- The smallest unit of data in your computer's memory is one bit. A bit is either 0 or 1.
- 8 bits make up a byte.
- 2^{10} bytes is 1 kilobyte (KB). 2^{10} KB is 1 megabyte (MB). 2^{10} MB is 1 gigabyte (GB).
- Every data type occupies a fixed amount space in your computer's memory.

Size of Datatypes

- There is an operator in C that takes as argument the name of a data type and returns the number of bytes the data type takes
  - the sizeof operator.
- For example, sizeof(int) return the number of bytes a variable of type int uses up in your computer's memory.

sizeof Examples

```
printf("int: %d\n", sizeof(int));                int: 4
printf("float: %d\n", sizeof(float));            float: 4
printf("long int: %d\n", sizeof(long int));      long int: 8
printf("double: %d\n", sizeof(double));          double: 8
printf("char: %d\n", sizeof(char));              char: 1
printf("int ptr: %d\n", sizeof(int *));          int ptr: 8
printf("double ptr: %d\n", sizeof(double*));     double ptr: 8
printf("char ptr: %d\n", sizeof(char *));        char ptr: 8
```

- The values can vary from computer to computer.
- Note that all pointer types occupy the same number of bytes (8 bytes in this case).
  - Depends only on total # of memory blocks (RAM/Virtual Memory) and not on data type

Static Memory Allocation

- When we declare an array, size has to be specified before hand
- During compilation, the C compiler knows how much space to allocate to the program
  - Space for each variable.
  - Space for an array depending on the size.
- This memory is allocated in a part of the memory known as the stack.
- Need to assume worst case scenario
  - May result in wastage of Memory

Dynamic Memory Allocation

- There is way of allocating memory to a program during runtime.
- This is known as dynamic memory allocation.
- Dynamic allocation is done in a part of the memory called the heap.
- You can control the memory allocated depending on the actual input(s)
  - Less wastage

Memory allocation: malloc

- The malloc function is declared in stdlib.h
- Takes as argument a **+ve** integer (say **n**),
- Allocates **n** consecutive bytes of memory space, and
- returns the address of the first cell of this memory space
- The return type is void*

WAIT!! Doesn't void means "nothing" in C? What is the meaning of void*? Pointer to nothing!

void* is NOT pointer to nothing!

- malloc knows *nothing* about the use of the memory blocks it has allocated
- void* is used to convey this message
  - Does not mean pointer to nothing, but means pointer to *something* about which *nothing is known*
- The blocks allocated by malloc can be used to store "anything" provided we allocate enough of them

malloc: Example

A pointer to float

```
float *f;
f= (float*) malloc(10 * sizeof(float));
```

Size big enough to hold 10 floats.

Explicit type casting to convey users intent

Note the use of **sizeof** to keep it machine independent

malloc evaluates its arguments at *runtime* to allocate (reserve) space. Returns a **void***, pointer to first address of allocated space.

malloc: Example

Key Point: The size argument can be a variable or non-constant expression!

After memory is allocated, pointer variable behaves as if it is an array!

```
float *f; int n;
scanf("%d", &n);
f= (float*) malloc(n * sizeof(float));

f[0] = 0.52;
scanf("%f", &f[3]); // OOB if n<=3
printf("%f", *f + f[0]);
```

Exercise

- Write a program that reads two integers, n and m, and stores powers of n from 0 up to m (n^0, n^1, …, n^m)

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *pow, i, n, m;
    scanf("%d %d", &n, &m); // m>= 0
    pow = (int *) malloc ((m+1) * sizeof(int));
    pow[0] = 1;
    for (i=1; i<=m; i++)
        pow[i] = pow[i-1]*n;
    for (i=0; i<=m; i++)
        printf("%d\n",pow[i]);
    return 0;
}
```

Note that instead of writing **pow[i]**, we can also write ***(pow + i)**

NULL

- A special pointer value to denote "points-to-nothing"
- C uses the value 0 or name NULL
- In Boolean context, NULL is equivalent to false, any other pointer value is equivalent to true
- A malloc call can return NULL if it is not possible to satisfy memory request
  - negative or ZERO size argument
  - TOO BIG size argument

Pointers and Initialization

- Uninitialized pointer has GARBAGE value, NOT NULL
- Memory returned by malloc is not initialized.
- Brothers of malloc
  - calloc(n, size): allocates memory for n-element array of size bytes each. Memory is initialized to ZERO.
  - realloc(ptr, size): changes the size of the memory block pointed to by ptr to size bytes.
    - Complicated semantics, try to avoid.

With great power comes great responsibility

- Power to allocate memory when needed must be complimented by the responsibility to de-allocate memory when no longer needed!
  - free unused pointers
- Be prepared to face rejection of demand
  - Check the return value of malloc (and its variants)

Freeing the memory: free

void free(void *ptr)

- Takes as argument a pointer that points to a dynamically allocated memory block and releases it.
- Does NOT require the size argument
- Can free only dynamically allocated pointers (malloc'ed/calloc'ed/realloc'ed)
- malloc and free need not be part of same function
  - memory allocated is not part of function's stack

Typical dynamic allocation

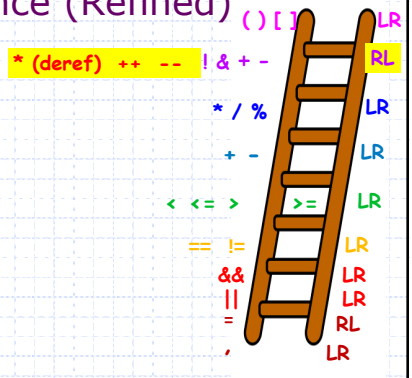
```
int *ar;
…
ar = (int*) malloc(…);
if (ar == NULL) {  // ≡ if (!ar)
  // take corrective measures OR
  // return failure
}
…
…ar[i]… // use of ar
…
free(ar); // free after last use of ar
```

Arrays and Pointers

- In C, array names are nothing but pointers.
  - Can be used interchangeably in most cases
- However, array names can not be assigned, but pointer variables can be.

```
int ar[10], *b;
ar = ar + 2;   ✗
ar = b;        ✗
b = ar;        ✓
b = b + 1;     ✓
b = ar + 2;    ✓
b++;           ✓
```


Precedence (Refined)



Array of Pointers

- Consider the following declaration

  int *arr[10];

- arr is a 10-sized array of pointers to integers
- How can we have equivalent dynamic array?

```
int **arr;
arr = (int **)malloc ( 10 * sizeof(int *) );
```

Array of Pointers

```
int **arr;
arr = (int **)malloc ( 10 *  sizeof(int *)  );
```

- Note that individual elements in the array arr (arr[0], … arr[9]) are NOT allocated any space.
- We need to do it (directly or indirectly) before using them.

```
int j;
for (j = 0; j < 10; j++)
    arr[j] = (int*) malloc (sizeof(int));
```


Multi-dimensional Array vs. Multi-level pointer

- Are these two equivalent?

```
int a[2][3];
```

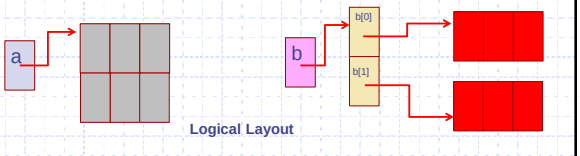
```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```

- Both a and b can hold 6 integer in a 2x3 grid like structure.
- In case of a all 6 cells are consecutively allocated. For b, we have 2 blocks of 3 consecutive cells each.

Memory layout

```
int a[2][3];
```

```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```



Logical Layout

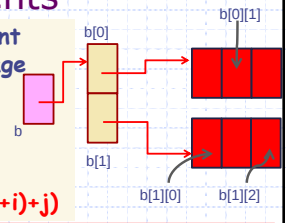
Possible Physical Layout

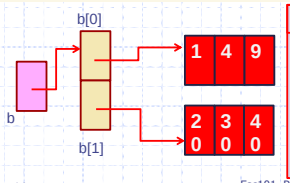
Indexing Elements

How do refer to an element of the array in the language of pointers?

- b[0][1] is *(*b+1)
- b[1][0] is **(b+1)
- b[1][2] is *(*(b+1)+2)

In general, **b[i][j] is *(*(b+i)+j)**





Expression	Value
**(b+1)	20
*(*b+1)	4
(**b+1)	2
*(*b+2)+2	11
((b+1)+2)+2	42

Pointers vs. Arrays: Indexing

- Matrix style notation A[i][j] is easier for humans to read
- Computers understand pointer style notation *(*(p + i) + j)
  - More efficient in some cases
- Be extremely careful with brackets
  - **(p + i +j) ≠ *(*(p + i) + j) ≠ *(*p + i + j)


```
int a[3][3], i, j, *b, *c;

for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        a[i][j] = pow((i+3),(j+1));

b = *a;
c = *(a+2);
for (i=0; i<3; i++)
    printf("%d ", b[i]);
printf("\n");

for (i=0; i<3; i++)
    printf("%d ", *(c+i));
```

At this point, array **a** is:

3	9	27
4	16	64
5	25	125

What do **b** and **c** point-to here?

b is a pointer to a[0][0]?
c is a pointer to a[2][0]?

OUTPUT
3 9 27
5 25 125

```
int a[3][3], i, j, *b, *c;

for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        a[i][j] = pow((i+3),(j+1));

b = *a;
c = *(a+2) + 1;
for (i=0; i<3; i++)
    printf("%d ", b[i]);
printf("\n");

for (i=0; i<2; i++)
    printf("%d ", *(c+i));
```

At this point, array **a** is:

3	9	27
4	16	64
5	25	125

What do **b** and **c** point-to here?

b is a pointer to a[0][0]?
c is a pointer to a[2][1]?

OUTPUT
3 9 27
25 125

note the change

Exercise: All Substrings

- Read a string and create an array containing all its substrings.
- Display the substrings.

Input: ESC

Output:
E
ES
ESC
S
SC
C

All Substrings: Solution Strategy

- What are the possible substrings for a string having length len?
- For $0 \leq i < len$ and for every $i \leq j < len$, consider the substring between the i^{th} and j^{th} index.
- Allocate a 2D array having $\frac{len \times (len+1)}{2}$ rows (Why ?)
- Copy the substrings into different rows of this array.

Solution: Version 1

```
int len, i, j, k=0, nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);
for (i=0; i<nsubstr; i++)
    substrs[i] = (char*)malloc(sizeof(char) * (len+1));

for (i=0; i<len; i++){
    for (j=i; j<len; j++){
        strncpy(substrs[k], st+i, j-i+1);
        k++;
    }
}
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```


Solution: Version 2

```
int len, i, j, k=0,nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);

for (i=0; i<len; i++){
   for (j=i; j<len; j++){
      substrs[k] = (char*)malloc(sizeof(char) * (j-i+2));
      strncpy(substrs[k], st+i, j-i+1);
      k++;
    }
}
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```

This version uses much less memory compared to version 1

Solution: Version 3

```
int len, i, j, k=0,nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);

for (i=0; i<len; i++){
   for (j=i; j<len; j++){
      substrs[k] = strndup(st+i, j-i+1);
      k++;
    }
}
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```

Less code => more readable, fewer bugs!
possibly faster!

Returning Pointers

Source: http://www.xkcd.com/138

Example Function that Returns Pointer

char *strdup(const char *s);

- strdup creates a copy of the string (char array) passed as arguments
  - copy is created in dynamically allocated memory block of sufficient size
- returns a pointer to the copy created
- C does not allow returning an Array of any type from a function
  - But we can use a pointer to simulate return of an array (or multiple values of same type)

Returning Pointer: Beware

```
#include<stdio.h>
int *fun();
int main() {
  printf("%d",*fun());
}

int *fun() {
  int *p, i;
  p = &i;
  i = 10;
  return p;
}
```

OUTPUT

```
#include<stdio.h>
int *fun();
int main() {
  printf("%d",*fun());
}

int *fun() {
  int *p;
  p = (int*)malloc(sizeof(int));
  *p = 10;
  return p;
}
```

OUTPUT: 10

Returning Pointer: Beware

- The function stack (except for the return value) is gone once the function completes its execution.
  - All addresses of local variables and formal arguments become invalid
  - available for "reuse"
- But the heap memory, once allocated, remains until it is explicitly "freed"
  - even beyond the function that allocated it.
- addresses of static and global variables remain valid throughout the program.

An Intuition

- Think of executing a function as writing on a classroom blackboard.
- Once the function finishes execution (the class is over), everything on the blackboard is erased.
- What if we want to retain a message, after class is over?
- Solution could be to post essential information on a "notice board", which is globally accessible to all classrooms.
- The blackboard of a class is like the stack (possibly erased/overwritten in the next class), and the notice board is like the heap.

Common Issues and Errors

Source: http://www.xkcd.com/371

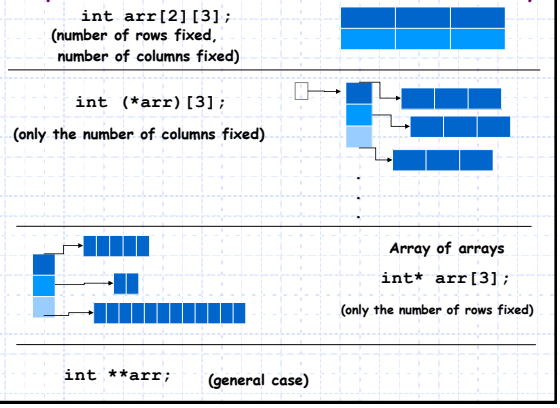
Common Issues and Errors

- Forgetting to malloc, forgetting to initialize allocated memory
- Not allocating enough space in malloc (e.g. Allocating 4 characters instead of 5 to store the string "IITK".)
- Returning pointers to temporaries (called dangling pointers)
- Forgetting to free memory after use (called a memory leak.)
- Freeing the same memory more than once (runtime error), using free-d memory

Memory Leaks

- Consider code:
```
1. int *a;
2. a = (int *)malloc(5*sizeof(int));
3. a = NULL;
```
- Memory is allocated to a at line 2.
- However, at line 3, a is reassigned NULL
- No way to refer to allocated memory!!
  - We can not even free it, as free-ing requires passing address of allocated block
- This memory is practically lost for the program (Leaked)
  - Ideally, memory should be freed before losing last reference to it

Array of Pointers vs. Pointer to an Array



```
int **a; *b[2]; (*c)[3]; d[2][3];

c = d;  /* Fine, matches the column size */
c = b;  /* Warning: incompatible pointer type */

printf("sizeof(a) = %3d, sizeof(*a) = %3d, sizeof(**a) = %3d\n",
          sizeof(a), sizeof(*a), sizeof(**a));
```

sizeof(a) = 8, sizeof(*a) = 8, sizeof(**a) = 4

```
printf("sizeof(b) = %3d, sizeof(*b) = %3d, sizeof(**b) = %3d\n",
          sizeof(b), sizeof(*b), sizeof(**b));
```

sizeof(b) = 16, sizeof(*b) = 8, sizeof(**b) = 4

```
printf("sizeof(c) = %3d, sizeof(*c) = %3d, sizeof(**c) = %3d\n",
          sizeof(c), sizeof(*c), sizeof(**c));
```

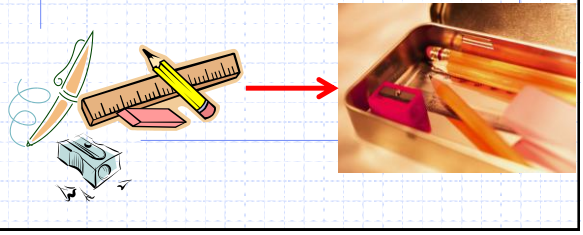
sizeof(c) = 8, sizeof(*c) = 12, sizeof(**c) = 4

```
printf("sizeof(d) = %3d, sizeof(*d) = %3d, sizeof(**d) = %3d\n",
          sizeof(d), sizeof(*d), sizeof(**d));
```

sizeof(d) = 24, sizeof(*d) = 12, sizeof(**d) = 4

ESC101: Introduction to Computing

Structures



Motivation

- Till now, we have used data types int, float, char, arrays (1D, 2D,...) and pointers.
- What if we want to define our own data types based on these?
- Say a geometry package – we want to define a point as having an x coordinate, and a y coordinate.
  - array of size 2?
  - we keep two variables point_x and point_y? (But there is no way to indicate that they are part of the same point! - requires a disciplined use of variable names)
- Is there any better way ?

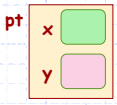
Structures

- A structure is a collection of variables with a common name.
- The variables can be of different types (including arrays, pointers or structures themselves!).
- Structure variables are called fields.

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

This defines a structure called point containing two integer variables (fields), called x and y.

struct point pt defines a variable pt to be of type struct point.

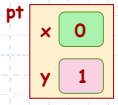


memory depiction of pt

Structures

- The x field of pt is accessed as pt.x.
- Field pt.x is an int and can be used as any other int.
- Similarly the y field of pt is accessed as pt.y

```
struct point {
      int x;
      int y;
};
struct point pt;
pt.x = 0;
pt.y = 1;
```



memory depiction of pt

Structures

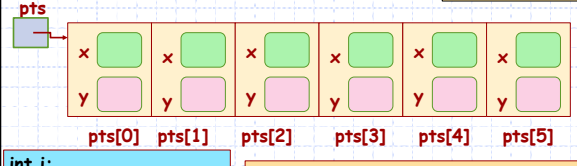
```
struct point {
      int x; int y;
}
```

struct point is a type. It can be used just like int, char etc..

For now, define structs in the beginning of the file, after #include.

```
struct point pt1,pt2;
struct point pts[6];
```

We can define array of struct point also.



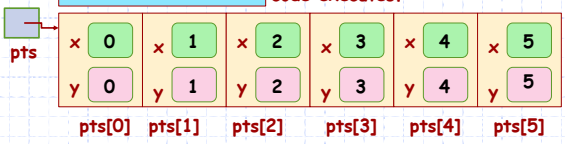

```
int i;
for (i=0; i < 6; i=i+1) {
      pts[i].x = i;
      pts[i].y = i;
}
```

Read pts[i].x as (pts[i]).x
The . and [] operators have same precedence. Associativity: left-right.

Structures

```
struct point {
      int x; int y;
};
struct point pts[6];
int i;
for (i=0; i < 6; i=i+1)
{
      pts[i].x = i;
      pts[i].y = i;
}
```

State of memory after the code executes.



pts[0] pts[1] pts[2] pts[3] pts[4] pts[5]

Functions returning structures

```
struct point {
      int x; int y;
};
struct point make_point
              (int x, int y)
{
      struct point temp;
      temp.x = x;
      temp.y = y;
      return temp;
}
int main() {
      int x, y;
      struct point pt;
      scanf("%d%d", &x,&y);
      pt = make_point(x,y);
      return 0;
}
```

1. make_point(x,y) creates a struct point given coordinates (x,y).
2. Note: make_point(x,y) returns struct point.
3. Functions can return structures just like int, char, int *, etc..
4. We can also pass struct parameters. struct are passed by copying the values.

Given int coordinates x,y, make_point(x,y) creates and returns a struct point with these coordinates.

Functions with structures as parameters

```
# include <stdio.h>
# include <math.h>
struct point {
      int x; int y;
};
double norm2( struct point p) {
   return sqrt ( p.x*p.x + p.y*p.y);
}
int main() {
      int x, y;
      struct point pt;
      scanf("%d%d", &x,&y);
      pt = make_point(x,y);
      printf("distance from origin
            is %f ", norm2(pt) );
      return 0;
}
```

The norm2 or Euclidean norm of point (x,y) is

$$\sqrt{x^2+y^2}$$

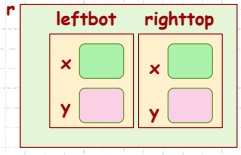
norm2(struct point p) returns Euclidean norm of point p.

Structures inside structures

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

```
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct rect r;
```

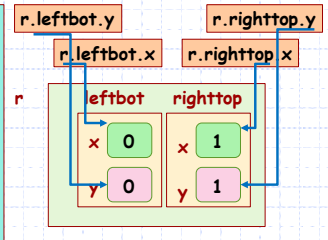
1. Recall, a structure definition defines a type.
2. Once a type is defined, it can be used in the definition of new types.
3. struct point is used to define struct rect. Each struct rect has two instances of struct point.



r is a variable of type struct rect. It has two struct point structures as fields.

So how do we refer to the x of leftbot point structure of r?

```
struct point {
      int x;
      int y;
};
struct rect {
   struct point leftbot;
   struct point righttop;
};
int main() {
      rect r;
      r.leftbot.x = 0;
      r.leftbot.y = 0;
      r.righttop.x = 1;
      r.righttop.y = 1;
      return 0;
}
```



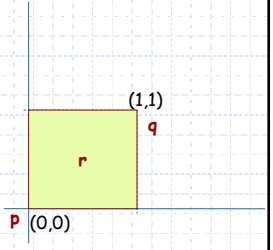
Addressing nested fields unambiguously

Initializing structures

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

1. Initializing structures is very similar to initializing arrays.
2. Enclose the values of all the fields in braces.
3. Values of different fields are separated by commas.

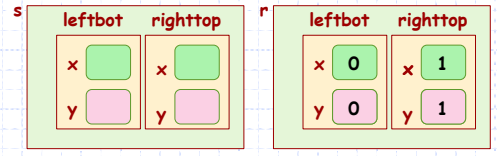
```
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct point p = {0,0};
struct point q = {1,1};
struct rect r = {{0,0}, {1,1}};
```



Assigning structure variables

```
struct rect r,s;
r.leftbot.x = 0;
r.leftbot.y = 0;
r.righttop.x = 1;
r.righttop.y = 1;
s=r;
```

1. We can assign a structure variable to another structure variable
2. The statement s=r; does this

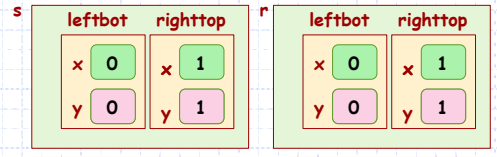


Before the assignment

Assigning structure variables

```
struct rect r,s;
r.leftbot.x = 0;
r.leftbot.y = 0;
r.righttop.x = 1;
r.righttop.y = 1;
s=r;
```

1. We can assign a structure variable to another structure variable
2. The statement s=r; does this



After the assignment

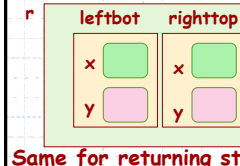
```
struct rect { struct point leftbot;
              struct point righttop; };
int area(struct rect r) {
   return
       (r.righttop.x – r.leftbot.x) *
       (r.righttop.y – r.leftbot.y);
}
void fun() {
   struct rect r ={{0,0}, {1,1}};
   area(r);
}
```

Passing structures..?

We can pass structures as parameters, and return structures from functions, like the basic types int, char, double etc..

But is it efficient to pass structures or to return structures?

Usually NO. E.g., to pass struct rect as parameter, 4 integers are copied. This is expensive.



Same for returning structures

So what should be done to pass structures to functions?

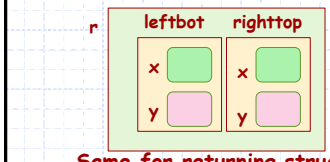
```
struct rect { struct point leftbot;
              struct point righttop;};
int area(struct rect *pr) {
  return
 ((*pr).righttop.x – (*pr).leftbot.x) *
 ((*pr).righttop.y – (*pr).leftbot.y);
}
void fun() {
   struct rect r ={{0,0}, {1,1}};
   area (&r);
 }
```

Passing structures..?

Instead of passing structures, pass pointers to structures.

area() uses a pointer to struct rect pr as a parameter, instead of struct rect itself.

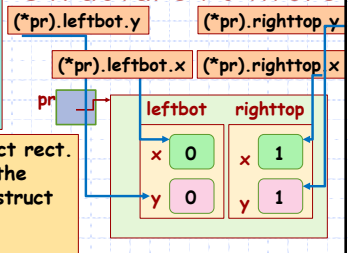
Now only one pointer is passed instead of a large struct.



Same for returning structures

```
struct point {
      int x; int y;};
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct rect *pr;
```

Structure Pointers

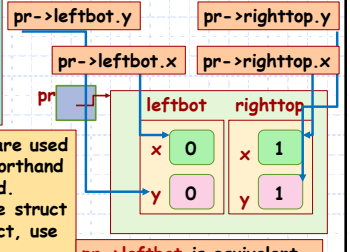


1. *pr is pointer to struct rect.
2. To access a field of the struct pointed to by struct rect, use
   (*pr).leftbot
   (*pr).righttop
3. Bracketing (*pr) is essential here. * has lower precedence than .
4. To access the x field of leftbot, use (*pr).leftbot.x

Addressing fields via the structure's pointer

```
struct point {
      int x; int y;};
struct rect {
   struct point leftbot;
   struct point righttop;
};
struct rect *pr;
```

Structure Pointers



1. Pointers to structures are used so frequently that a shorthand notation (->) is provided.
2. To access a field of the struct pointed to by struct rect, use
   pr->leftbot
3. -> is one operator. To access the x field of leftbot, use
   pr->leftbot.x
3. -> and . have same precedence and are left-associative. Equivalent to (pr->leftbot).x

pr->leftbot is equivalent to (*pr).leftbot

Addressing fields via the structure's pointer (shorthand)

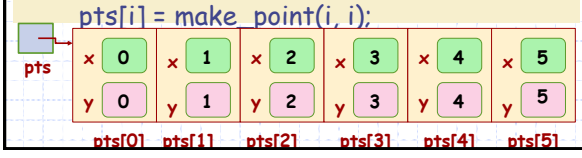
Passing struct to functions

- When a struct is passed directly, it is passed by copying its contents
  - Any changes made inside the called function are lost on return
  - This is same as that for simple variables
- When a struct is passed using pointer,
  - Change made to the contents using pointer dereference are visible outside the called function

Dynamic Allocation of struct

- Similar to other data types
- sizeof(…) works for struct-s too

```
struct point* pts;
int i;
pts = (struct point*) malloc(6 * sizeof(struct point));
for (i = 0; i < 6; i++)
    pts[i] = make_point(i, i);
```



(Re)defining a Type - typedef

- When using a structure data type, it gets a bit cumbersome to write struct followed by the structure name every time.
- Alternatively, we can use the typedef command to set an alias (or shortcut).

```
struct point {
      int x; int y;
};
typedef struct point Point;
struct rect {
   Point leftbot;
   Point righttop;
};
```

- We can merge struct definition and typedef:

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

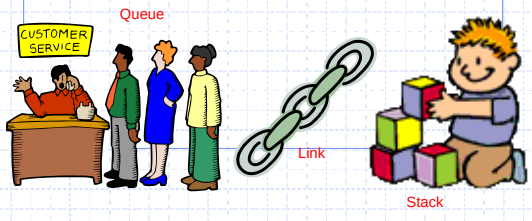
More on typedef

- `typedef` may be used to rename *any* type
  - Convenience in naming
  - Clarifies purpose of the type
  - Cleaner, more readable code
  - Portability across platforms

```
typedef char *String; // String: a new
                      // name to char pointer
typedef int size_t; // Improved
                    // Readability
typedef struct point *PointPtr;
typedef long long int64; // Portability
```

ESC101: Introduction to Computing

Data Structures



Data Structure

- What is a data structure?
- According to Wikipedia:
  - … a particular way of storing and organizing data in a computer so that it can be used efficiently...
  - … highly specialized to specific tasks.
- Examples: array, a dictionary, a set, etc.

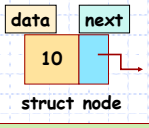
Linked List

- A linear, dynamic data structure, consisting of nodes. Each node consists of two parts:
  - a "data" component, and
  - a "next" component, which is a pointer to the next node (the last node points to nothing).

Linked List : A Self-referential structure

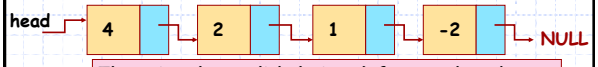
Example:

```
struct node {
    int data;
    struct node *next;
};
```



1. Defines the structure struct node, which will be used as a node in a "linked list" of nodes.
2. Note that the field next is of type struct node *
3. If it was of type struct node, it could not be permitted (recursive definition, of unknown or infinite size).

An example of a (singly) linked list structure is:

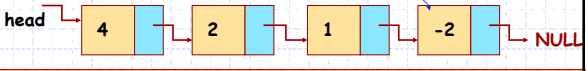


There is only one link (pointer) from each node, hence, it is also called "singly linked list".

Linked Lists

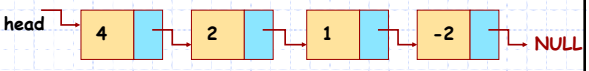
List starts at node pointed to by head

next field == NULL pointer indicates the last node of the list



1. The list is modeled by a variable called head that points to the first node of the list.
2. head == NULL implies empty list.
3. The next field of the last node is NULL.
4. Note that the name head is just a convention – it is possible to give any name to the pointer to first node, but head is used most often.

Displaying a Linked List

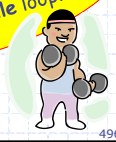



```
void display_list(struct node *head)
{
  struct node *cur = head;
  while (cur != NULL) {
    printf("%d ", cur->data);
    cur = cur->next;
  }
  printf("\n");
}
```

OUTPUT

4 2 1 -2

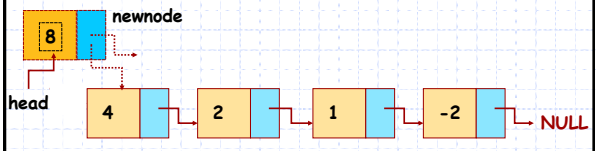
Exercise: Rewrite the code using for loop instead of while loop.



Insert at Front

Inserting at the front of the list.

1. Create a new node of type struct node. Set its data field to the value given.
2. "Add" it to the front of the list: Make its next pointer point to target of head.
3. Adjust head correctly to point to newnode.

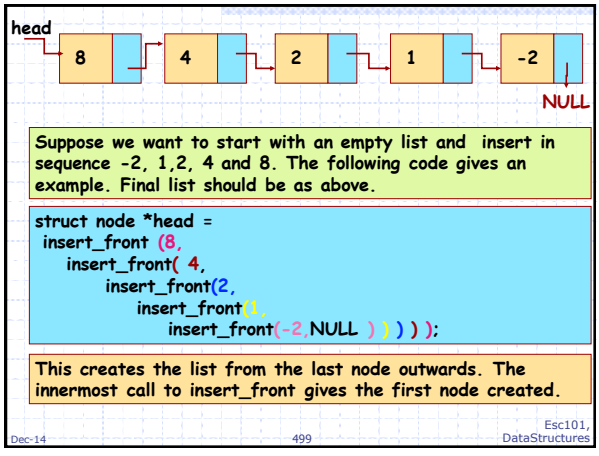


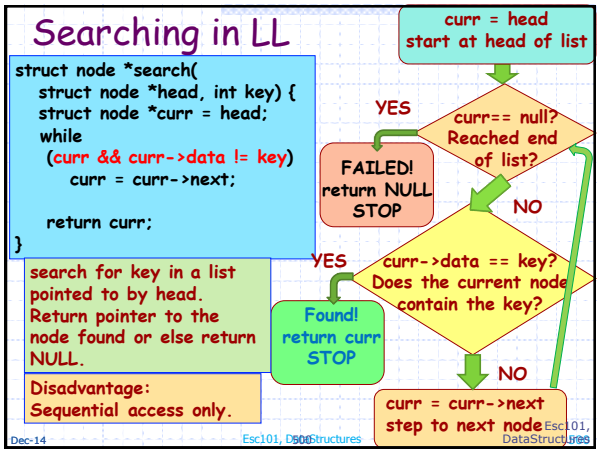

```
struct node * make_node(int val) {
    struct node *nd;
    nd = calloc(1, sizeof(struct node));
    nd->data = val;
    return nd;
}
```

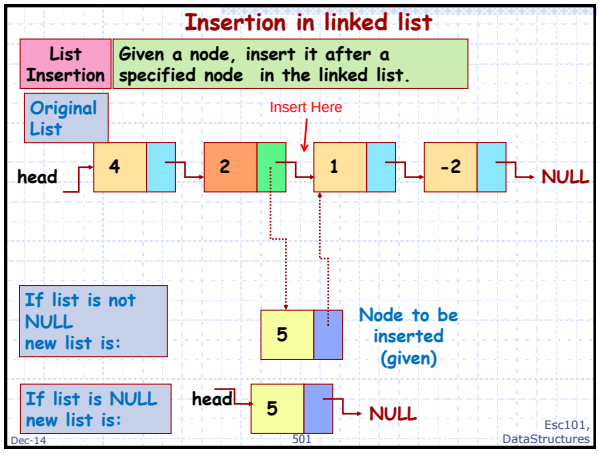
/* Allocates new node pointer and sets the data field to val, next field initialized to NULL */

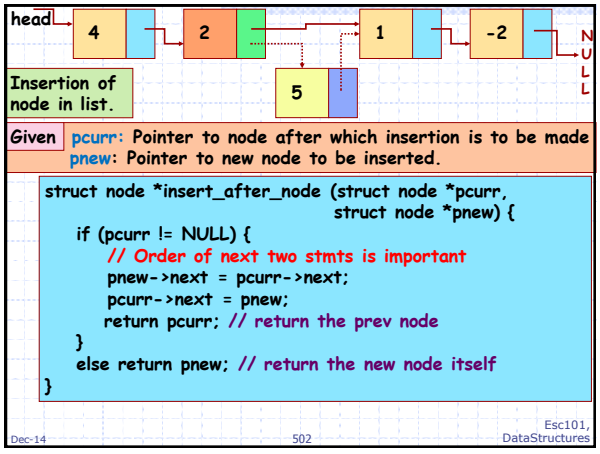
```
struct node *insert_front(int val, struct node *head) {
    struct node *newnode= make_node(val);
    newnode->next = head;
    head = newnode;
    return head;
}
```

/* Inserts a node with data field val at the head of the list currently pointed to by head. Returns pointer to the head of new list. Works even when the original list is empty, i.e. head == NULL */


head
8
4
2
1
-2
NULL
Suppose we want to start with an empty list and insert in sequence -2, 1,2, 4 and 8. The following code gives an example. Final list should be as above.
struct node *head =
insert_front (8,
insert_front(4,
insert_front(2,
insert_front(1,
insert_front(-2,NULL)))));
This creates the list from the last node outwards. The innermost call to insert_front gives the first node created.
Dec-14
499
Esc101, DataStructures


Searching in LL
struct node *search(
struct node *head, int key) {
struct node *curr = head;
while
(curr && curr->data != key)
curr = curr->next;
return curr;
}
search for key in a list pointed to by head. Return pointer to the node found or else return NULL.
Disadvantage: Sequential access only.
curr = head
start at head of list
curr== null? Reached end of list?
YES
FAILED! return NULL STOP
NO
curr->data == key? Does the current node contain the key?
YES
Found! return curr STOP
NO
curr = curr->next
step to next node
Dec-14
Esc101, DataStructures


Insertion in linked list
List Insertion
Given a node, insert it after a specified node in the linked list.
Original List
Insert Here
head
4
2
1
-2
NULL
If list is not NULL new list is:
5
Node to be inserted (given)
If list is NULL new list is:
head
5
NULL
Dec-14
501
Esc101, DataStructures



Given pcurr: Pointer to node after which insertion is to be made
pnew: Pointer to new node to be inserted.

```
struct node *insert_after_node (struct node *pcurr,
                                struct node *pnew) {
    if (pcurr != NULL) {
        // Order of next two stmts is important
        pnew->next = pcurr->next;
        pcurr->next = pnew;
        return pcurr; // return the prev node
    }
    else return pnew; // return the new node itself
}
```


Recap: typedef in C

- Repetitive to keep writing the type struct node for parameters, variables etc.
- C allows naming types— the typedef statement.

Defines a new type Listnode as struct node *

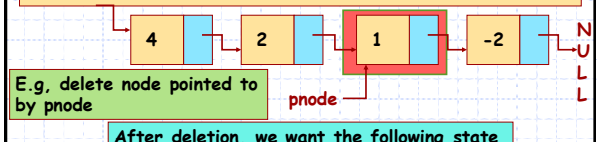
```
typedef struct node * Listnode;
```

Listnode is a type. It can now be used in place of struct node * for variables, parameters, etc..

```
Listnode head, curr;
 /* search in list for key */
Listnode search(Listnode list, int key);
/* insert the listnode n in front of listnode list */
Listnode insert_front(Listnode list, Listnode n);
 /* insert the listnode n after the listnode curr */
Listnode insert_after(Listnode curr, Listnode n);
```

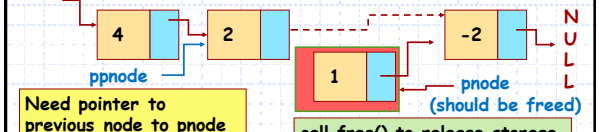

Deletion in linked list

Given a pointer to a node pnode that has to be deleted. Can we delete the node?



E.g, delete node pointed to by pnode

After deletion, we want the following state



Need pointer to previous node to pnode to adjust pointers.

call free() to release storage for deleted node.

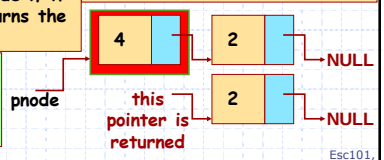
prototype `delete(Listnode pnode, Listnode ppnode)`

```
Listnode delete(Listnode pnode, Listnode ppnode)
{
    Listnode t;
    if (ppnode)
        ppnode->next = pnode->next;
    t = ppnode ? ppnode : pnode->next;
    free (pnode);
    return t;
}
```

Delete the node pointed to by pnode. ppnode is pointer to the node previous to pnode in the list, if such a node exists, otherwise it is NULL.

Function returns ppnode if it is non-null, else returns the successor of pnode.

The case when pnode is the head of a list. Then ppnode == NULL.



Why linked lists

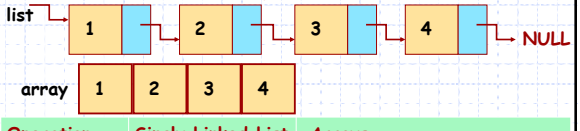
- The same numbers can be represented in an array. So, where is the advantage?

1. Insertion and deletion are inexpensive, only a few "pointer changes".
2. To insert an element at position k in array: create space in position k by shifting elements in positions k or higher one to the right.
3. To delete element in position k in array: compact array by shifting elements in positions k or higher one to the left.

Disadvantages of Linked List

- Direct access to kth position in a list is expensive (time proportional to k) but is fast in arrays (constant time).

Linked Lists: the pros and the cons



Operation	Singly Linked List	Arrays
Arbitrary Searching.	sequential search (linear)	sequential search (linear)
Sorted structure.	Still sequential search. Cannot take advantage.	Binary search possible (logarithmic)
Insert key after a given point in structure.	Very quick (constant number of operations)	Shift all array elements at insertion index and later one position to right. Make room, then insert. (linear time)

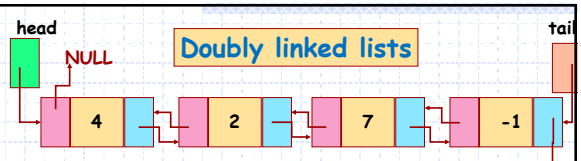
Singly Linked Lists

Operations on a linked list. For each operation, we are given a pointer to a current node in the list.

Operation	Singly Linked List
Find next node	Follow next field
Find previous node	Can't do !!
Insert before a node	Can't do !!
Insert in front	Easy, since there is a pointer to head.

Principal Inadequacy: Navigation is one-way only from a node to the next node.

Doubly linked lists



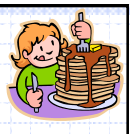
Each node has 3 fields: (i) pointer to previous node; (ii) data; (iii) pointer to next node

Defining node of Doubly linked list and the Dllist itself.

```
struct dlnode {
      int data;
      struct dlnode *next;
      struct dlnode *prev;
};
typedef struct dlnode *Ndptr;
```

```
struct dlList {
   Ndptr head;/*first node */
   Ndptr tail; /* last node */
};
typedef struct dlList *DlList;
```

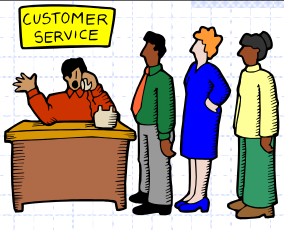

Stack

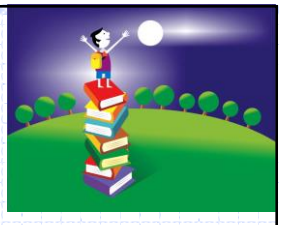


- A linear data structure where addition and deletion of elements can happen at one end of the data structure only.
  - Last-in-first-out.
  - Only the top most element is accessible at any point of time.
- Operations:
  - Push: Add an element to the top of the stack.
  - Pop: Remove the topmost element.
  - IsEmpty: Checks whether the

Queue

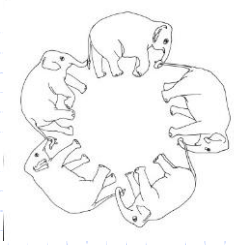
- A linear data structure where addition happens at one end (`back') and deletion happens at the other end (`front')
  - First-in-first-out
  - Only the element at the front of the queue is accessible at any point of time
- Operations:
  - Enqueue: Add an element to the back
  - Dequeue: Remove the element from the
  - IsEmpty: Checks whether the queue is empty or not.





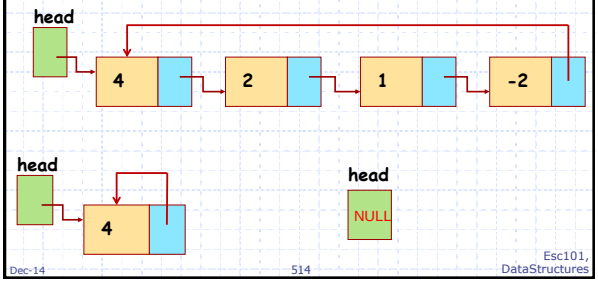
Both Queue and Stack data structures can be implemented using Linked Lists!

Circular Linked List

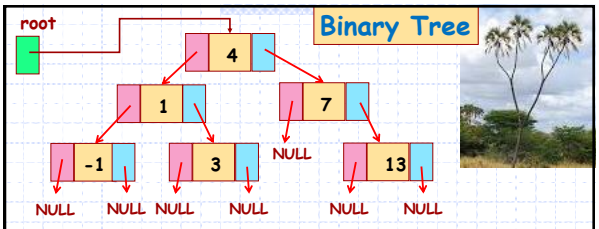


So far, we were modeling a singly linked list by a pointer to the first node of the list.
Let us make the following change:

Make the list circular: next pointer of last node is not NULL, it points to the head node.



Binary Tree



Each node has 3 fields: (i) pointer to left child node | (ii) data | (iii) pointer to right child node

Defining Binary Tree

```
typedef struct _btnode *Btree;
struct _btnode {
        int data;
        Btree left;
        Btree right;
};
```

```
Btree root;
```


Traversing a Binary Tree

- Visit each node in the binary tree exactly once
- Easy to traverse recursively
- Three common ways of visit
  - inorder: left, root, right
  - preorder: root, left, right
  - postorder: left, right

```
void inorder(tree t)
{
   if (t == NULL) return;
   inorder(t->left);
   process(t->data);
   inorder(t->right);
}
```

Recursion vs Iteration

```
void inorder(tree t)
{
    stack s;
    push(s,t);
    while (!empty(s)) {
        curr = top(s);
        if (curr != null) {
            if (!curr->visited) {
                push(s,curr->left);
            } else {
                process(curr->data);
                pop(s);
                push(s,curr->right);
            }
        } else {
            pop(s);
            if (!empty(s))
                top(s)->visited = true;
        }
    }
}
```

```
void inorder(tree  t)
{
    if (t == null) return;

    inorder(t->left);
    process (t->data);
    inorder(t->right);
}
```

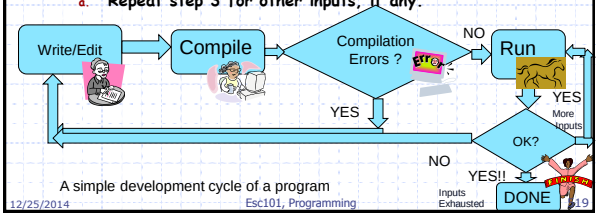
* Disclaimer: Code not tested!

ESC101: Introduction to Computing

Command Line & File Handling

The Programming Cycle

1. Write your program or **edit** (i.e., change or modify) your program.
2. **Compile** your program. If compilation fails, return to editing step.
3. **Run** your program on an input. If output is not correct, return to editing step.
   a. Repeat step 3 for other inputs, if any.



A simple development cycle of a program

Edit

- First login to the system.
- Now open an editor. An editor is a system program that lets you type in text, modify and update it.
  - Some popular editors are: vim, emacs, gedit, notepad
  - Use an editor that provides syntax highlighting and auto-indent
  - My personal favorites are emacs and vim – many powerful features
- Type in your code in the editor. Save what you type into a file.
  - Give meaningful names to your files.

Compile

- After editing, you have to COMPILE the program.
- The computer cannot execute a C program or the individual statements of a C program directly.
  - For example, in C you can write `g = a + b * c`
  - The microprocessor cannot execute this statement. It translates it into an equivalent piece of code consisting of even more basic statements.
- Some error checking is also done as part of compilation process.

How do you compile?

- On Unix/Linux systems you can COMPILE the program using the gcc command.

```
gcc sample.c
```

- If there are no errors, then the system silently shows the prompt ($).
- If there are errors, the system will list the errors and line numbers. Then you can edit (change) your file, fix the errors and recompile.
  - Warnings may also be produced.

Compile…

- As long as there are compilation errors, the EXECUTABLE file is not created.
- If there are no errors then gcc places the machine program in an executable format for your machine and calls it a.out
- The file a.out is placed in your current working directory.

Simple! Program

- Lets compile some of the simplest C programs.
- Login, then open an editor and type in the following lines. Save the program as sample.c

```
# include <stdio.h>
int main ()  {
    printf("Welcome to C");
    return 0;
}
```

sample.c: The program prints the message "Welcome to C"

Compile and Run

- Now compile the program. System compiles without errors.

```
$ gcc sample.c
$
```

- Compilation creates the executable file a.out by default.
- Now run the program. The screen looks like this:

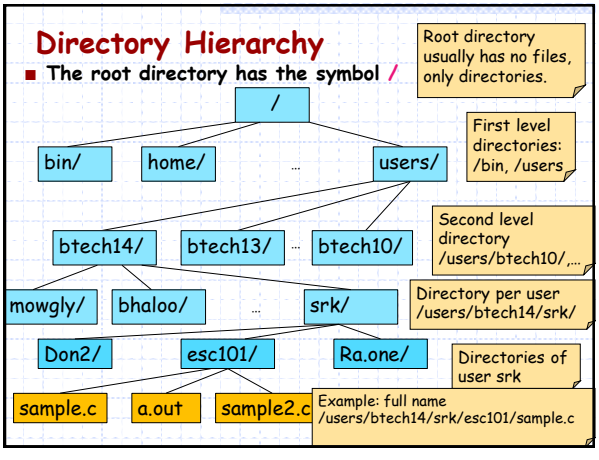
```
$ ./a.out
Welcome to C$
```

Introduction to Files and Directory

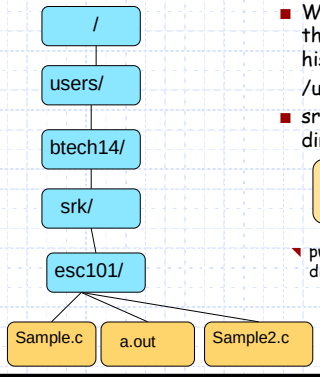
- Compiling using gcc by default produces the file a.out in your current working directory.
- Let us understand the notion of directory and current working directory.
- The unit of data in a system is a file.
- Files are organized into directories, also called folders. Each directory may have many files inside it and also many directories inside it.
- Having files and directories inside directories gives it a hierarchical structure.

Directory Hierarchy

- **The root directory has the symbol /**



Directory commands

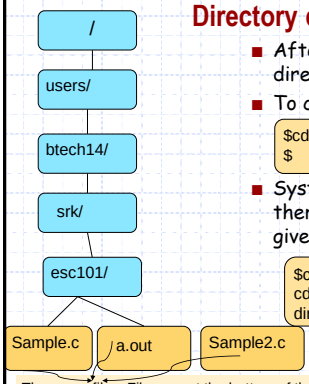


- When user srk logs in, the system places him in his home directory:
  /users/btech14/srk/
- srk can find his current directory by typing

```
$pwd
/usrs/btech14/srk
```

- pwd stands for print working directory

Directory commands



- After login srk is in home directory /users/btech14/srk
- To change directory to esc101

```
$cd esc101/
$
```

- System returns silently. If there is spelling error, system gives a message. For example,

```
$cd esc101a/
cd: esc101a: no such file or
directory
```

These are files. Files are at the bottom of the directory hierarchy. Files do not contain Files or directories. Only directories contain files or other directories (or both).

Arguments on the Command Line

- Typically when using commands we provide arguments to the command in the same line.
  - cd my_dir
  - gcc my_file.c
  - cp file1.c file2.c
- In each case, stuff in red is the command line argument
- In the third example, cp is the command name and file1.c and file2.c are its two arguments.

Batch mode vs. Interactive mode

- Interactive mode:
  1. first you enter command (say mkdir)
  2. then you get prompted and you enter an arg (the directory name, say esc101)
  3. mkdir creates the directory esc101, and asks if you want to create more directories. If you say yes, it goes to step 2. Else, it exits.
- This is cumbersome.
- Batch Mode: If the arguments are standard, we prefer entering them along with the command (Also called command-line mode):
  - mkdir esc101 phy102 chm_lab
  - 3 Directories created: esc101, phy102 and chm_lab

Command Line Args in C

- Write a program to read a name from command line, and say "Hello" to it.
- Some Example Interaction (Output in red):

```
$ ./a.out Amey
Hello Amey
$ ./a.out World
Hello World
$ ./a.out ESC101
Hello ESC101
```

Note that the program really has no sense of what is a name. It just prints the argument provided.

Command Line Args = Args to main

- So far we used the following signature for main

  int main()

- But main can take arguments. The modified prototype of main is

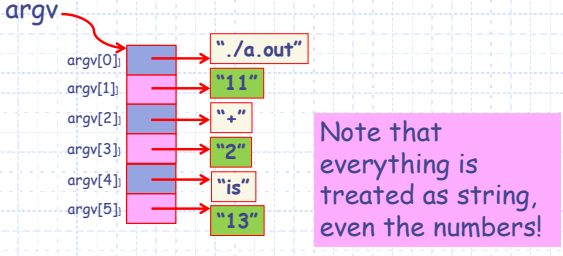
  int main(int argc, char **argv)

  - Argument Count (argc): An int that tells the number of arguments passed on command line
  - Argument Values (argv): Array of strings. argv[i] is the i-th argument as string.

Args to main

./a.out 11 + 2 is 13

argc = 6 ./a.out is included in arguments



Note that everything is treated as string, even the numbers!

Example

NOTE: char **argv is same as char *argv[]

```
#include<stdio.h>

int main(int argc, char *argv[]) {
   if (argc<2)
      printf ("Too few args!\n");
   else if (argc == 2)
      printf ("Hello %s\n",argv[1]);
   else
      printf("Too many args!\n");
   return 0;
}
```

```
$ ./a.out
Too few args!
```

```
$ ./a.out Amey
Hello Amey
$ ./a.out World
Hello World
$ ./a.out ESC101
Hello ESC101
```

```
$ ./a.out Hey There
Too many args!
```


What about Other Types?

- Write a program that takes two numbers (integers) on command line and prints their sum.
- Problem:
  - Everything on command line is read as string!
  - How do I convert string to int?
- Solution: Library functions in stdlib.h
  - atoi: takes a string and converts to int

```
atoi("1234") is 1234, atoi("123ab") is 123, atoi("ab") is 0
```

  - atof: converts a string to double
- Other variations : atol, atoll

Adding 2 Numbers

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[]) {
   if (argc != 3)
      printf ("Bad args!\n");
   else {
      int a = atoi(argv[1]);
      int b = atoi(argv[2]);
      printf ("%d\n",a+b);
   }
   return 0;
}
```

```
$ ./a.out
Bad args!
```

```
$ ./a.out 3 4
7
$ ./a.out 3 -4
-1
$ ./a.out 3 four
3
```

```
$ ./a.out 3 4 5
Bad args!
```

Command Line Sorting

```
int main(int argc, char *argv[]) {
  int *ar, n;

  n = argc - 1;
  ar = (int *)malloc(sizeof(int) * n);
  for (i=0; i<n; i++)
    ar[i] = atoi(argv[i+1]);

  merge_sort(ar, n); // or any other sort

  for (i=0; i<n; i++)
    printf("%d ",ar[i]);
  return 0;
}
```

```
void merge_sort (
      int *arr, int n)
{
   ...
}
```

```
$ ./a.out 1 4 2 5 3 9 -1 6 -10 10
-10 -1 1 2 3 4 5 6 9 10
```


Renaming Executable

```
int main(int argc, char *argv[]) {
  int *ar, n;

  n = argc - 1;
  ar = (int *)malloc(sizeof(int) * n);
  for (i=0; i<n; i++)
    ar[i] = atoi(argv[i+1]);

  merge_sort(ar, n); // or any other

  for (i=0; i<n; i++)
    printf("%d ",ar[i]);
  return 0;
}
```

```
void merge_sort (
      int *arr, int n)
{
   ...
}
```

The flag "-o" of gcc can be used to give user-defined name to the executable, e.g.

$ gcc -o sort myfile.c

```
$ ./sort 1 4 2 5 3 9 -1 6 -10 10
-10 -1 1 2 3 4 5 6 9 10
```


Reading from and Writing to a File from C Program

Files

- What is a file?
  - Collection of bytes stored on secondary storage like hard disks.
- Any addressable part of the file system in an Operating system can be a file.
  - includes such strange things as /dev/null (nothing), /dev/urand (random data device), /dev/audio (speakers), and of course, files that a user creates (/home/don/input.txt, /home/don/Esc101/lab12.c)

File Access

- 3 files are always connected to a C program :
  - stdin : the standard input, from where scanf, getchar(), gets() etc. read input from
  - stdout : the standard output, to where printf(), putchar(), puts() etc. output to.
  - stderr : standard error console.

File handling in C

1. Open the file for reading/writing etc.: fopen
   - return a *file pointer*
   - pointer points to an internal structure containing information about the file:
     - location of a file
     - the current position being read in the file
     - and so on.

```
FILE* fopen (char *name, char *mode)
```

2. Read/Write to the file

```
int fscanf(FILE *fp, char *format, …)
int fprintf(FILE *fp, char *forma
```

3. Close the File.

```
int fclose(FILE *fp)
```

Compared to scanf and printf – a new (first) argument fp is added

Opening Files

```
FILE* fopen (char *name, char *mode)
```

- The first argument is the name of the file
  - can be given in short form (e.g. "inputfile") or the full path name (e.g. "/home/don/inputfile")
- The second argument is the mode in which we want to open the file. Common modes include:
  - "r" : read-only. Any write to the file will fail. File must exist.
  - "w" : write. The first write happens at the beginning of the file, by default. Thus, may overwrite the current content. A new file is created if it does not exist.
  - "a" : append. The first write is to the end of the current content. File is created if it does not exist.

Opening Files

- If successful, fopen returns a *file pointer* – this is later used for fprintf, fscanf etc.
- If unsuccessful, fopen returns a NULL.
- It is a good idea to check for errors (e.g. Opening a file on a CDROM using "w" mode etc.)

Closing Files

- An open file must be closed after last use
  - allows reuse of FILE* resources
  - flushing of ***buffered*** data

File I/O: Example

- Write a program that will take two filenames, and print contents to the standard output. The contents of the first file should be printed first, and then the contents of the second.
- The algorithm:
  1. Read the file names.
  2. Open file 1. If open failed, we exit
  3. Print the contents of file 1 to stdout
  4. Close file 1
  5. Open file 2. If open failed, we exit
  6. Print the contents of file 2 to stdout
  7. Close file 2

The Program: main

```
int main()
{
  FILE *fp;  char filename1[128], filename2[128];
  scanf("%s", filename1);
  scanf("%s", filename2);
  fp = fopen( filename1, "r" );
  if(fp == NULL) {
    fprintf(stderr, "Opening File %s failed\n", filename1);
    return -1;
  }
  copy_file(fp, stdout);
  fclose(fp);
  fp = fopen( filename2, "r" );
  if (fp == NULL) {
    fprintf(stderr, "Opening File %s failed\n", filename2);
    return -1;
  }
  copy_file (fp, stdout);
  fclose(fp);
  return 0;
}
```


The Program: copy_file

```
void copy_file(FILE *fromfp, FILE *tofp)
{
  char ch;

  while ( !feof ( fromfp ) ) {
    fscanf  ( fromfp, "%c", &ch );
    fprintf ( tofp, "%c", ch );
  }
}
```


Some other file handling functions

- `int feof ( FILE* fp );`
  - Checks whether the EOF is set for fp – that is, the EOF has been encountered. If EOF is set, it returns nonzero. Otherwise, returns 0.
- `int ferror ( FILE *fp );`
  - Checks whether the error indicator has been set for fp. (for example, write errors to the file.)

Some other file handling functions

- `int fseek(FILE *fp, long int offset, int origin);`
  - To set the current position associated with fp, to a new position = origin+offset.
  - Origin can be:
    - SEEK_SET: beginning of file
    - SEEK_CURR: current position of file pointer
    - SEEK_END: End of file
- `int ftell(FILE *fp)`
  - Returns the current value of the position indicator of the stream.

Opening Files: More modes

- There are other modes for opening files, as well.
  - "r+" : open a file for read and write (update). The file must be present.
  - "w+" : write/update. Create an empty file and open it both for input and output.
  - "a+" : append/update. Repositioning operations (fseek etc.) affect next read. Output is always at the end of file.

FileI/O: stdout vs stderr

- What is the output of following program when run on a terminal:

```
#include <stdio.h>
int main()
{
  int input;
  scanf("%d", &input);
  fprintf(stdout, "Printing to STDOUT %d\n", input);
  fprintf(stderr, "Printing to STDERR %d\n", input);
  return 0;
}
```

INPUT
5

```
Printing to STDOUT 5
Printing to STDERR 5
```

FileI/O: stdout vs stderr

- What is the output of following program when run on a terminal:

```
#include <stdio.h>
int main()
{
  int input;
  scanf("%d", &input);
  fprintf(stdout, "Printing to STDOUT %d", input);
  fprintf(stderr, "Printing to STDERR %d", input);
  return 0;
}
```

INPUT
5

~~Printing to STDOUT 5Printing to STDERR 5~~

Printing to STDERR 5Printing to STDOUT 5

Stdout vs. Stderr (Intuition)

VS

.