C Program Structure: Statements, If else, Looping

IC100

7th November

Lecture 4

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 point

```
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
```

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!

Symbolic Constants

- #define name text

```
#include<stdio.h>
#define PI 3.141593

Int main (void){
    int radius = 4;
    float area;
    area = PI * radius * radius;
    printf("area = %f\n", area);
    return 0;
}
```

Likely printf Exam Questions

Print some pattern we give you, not necessarily these ones

Scanf Review

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

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

- 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.

Scanf Addendum

- Can input mutliple fields with one scanf
  - scanf("%d %d", &month, &day);
  - But can be frustrating to debug
- White space is skipped for consecutive numeric reads
  - But not skipped for character inputs
  - printf("Enter an integer and a float, then Y or N\n> ");
  - scanf("%d%f%c", &i, &f, &c);
  - Enter 12 23.4 N
  - i = 12, f = 23.4, c =
  - Enter 12 23.4N to get it right

Other basic I/O commands

- getchar()
  - Command to accept a single character as input
  - Syntax: *variable = getchar();*
  - Useful as an interactivity device
    - User prompt
    - Conditional behaviour
- putchar()
  - Command to print a single character as output
  - Syntax: *putchar(variable);*
  - Not as useful, limited applications

Reading and Writing Strings

- A string is an array of characters
  - E.g. “Hello, how are you?”
- Declared with syntax
  - *char x[LENGTH_OF_STRING];*
- C has special I/O functions for strings
  - Use gets() for string input
  - Use puts() for string output

```
int main(){
        char line[80];
        gets(line);
        puts(line);
}
```

Writing strings with printf

- Straightforward
  - Use %s instead of %c as a placeholder
  - *printf(“%s”, line)*

Reading strings with scanf

- Not straightforward
  - Has trouble handling white spaces
  - Remember to leave white space in format string for character inputs
  - *scanf(" %s", line)*
  - *Input "This is a line"*
  - *Stores "This"*
- Fixes
  - Use a custom placeholder
  - Defined using syntax *%[^\n]*

Debugging Workflow

- Fix syntax errors
- Fix execution errors
- Fix logic errors

Fixing Syntax Errors

Caught by the compiler

- Compiler will tell you what to fix

error.c:3:26: error: expected ';' after expression

Examples:

- Missing a semicolon
- Missing a closing quotation mark
- Unbalanced bracket
- Comment closes inappropriately
- Etc.

Fixing Execution Errors

- Runtime errors
  - Usually caused by bad syntax or poor exception handling
- Compiler not helpful
- No output to look at (Segmentation fault ☹)
- Examples
  - Numerical overflow or underflow
  - Division by zero
  - Attempting to compute logarithm or square root of a negative number
  - Array buffer overflow
  - Etc.

Use Comments

- Comment out all but the most innocuous code
- Program will now execute normally
  - Without accomplishing anything

```
#include <stdio.h>
int main(){
                // declarations

                //scanf("%d",amount);
                //printf("Input amount: Rs %d\n", amount);
                //printf("%d notes of Rs. %d\n", notes_1, N1); // likewise for others

                // calculations
    return 0;
}
```

Uncomment Statements One by One

- The statement that triggers the execution error is the problem
- Focus on that statement's syntax and fix it

```
#include <stdio.h>
int main(){
                // declarations

                scanf("%d",&amount);
                //printf("Input amount: Rs %d\n", amount);
                //printf("%d notes of Rs. %d\n", notes_1, N1); // likewise for others

                // calculations
    return 0;
}
```

Use Bisection Method

- Comment out the second half of the code
- An execution error indicates an error in the first half
- Successful execution indicates an error in the second half

Fixing Logical Error: Tracing

- Print out intermediate outputs where they are computed
- Differences from expected values show errors in logic

```
#include <stdio.h>
int main(){
        // declarations
        // I/O
         notes_1 = amount/N1;
         printf("%d", notes_1);
         amount = amount%N1;

         return 0;
}
```

Other Techniques

- Watch variables
  - Useful for programs with loops
  - Help identify when in the execution problems are occurring
- Breakpoints
  - Temporary stopping point within a program
- Stepping
  - Execution of one instruction at a time

Read about gdb

https://www.gnu.org/software/gdb/

Operators

- Operators are the building blocks of expressions
- Have their own syntax
- We will learn this syntax the next few lectures

Types of operators

- Arithmetic
- Unary
- Relational and logical
- Assignment
- Conditional

Arithmetic Operators

- 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 division
		9.1/2.0 is 4.55	Real division
%	Remainder	9%2 is 1	Only for int

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 (int and long)

  4%2 is 0

  31%4 is 3

Implicit Type Casting in C Arithmetic

- General rule for mixed type arithmetic
  - Final result will have highest possible precision consistent with operand types
  - Small boxes big boxes
- Examples
  - *float + double = double*
  - *float + long double = long double*
  - *int + float = float*
  - *char + float = float*
  - *…*

Explicit Type Casting in C Arithmetic

- Can cast an arithmetic expression to a specific data type
  - Syntax *(data type) expression;*
  - *E.g. ((int) f) % 2*

Precedence and Associativity for Arithmetic Expressions

Precedence	Operators
High	* / %
Low	+ -

e.g. a – b / c * d = a – ((b/c)*d)

Like in regular arithmetic, can use brackets to clarify correct order of operations, e.g. (a-b)/(c*d)

Associativity goes left □ right for arithmetic expressions

Types of operators

- Arithmetic
- **Unary**
- Relational and logical
- Assignment
- Conditional

Unary operators

Operator	Description
-	Negative of an expression
++/--	Increment/decrement a variable
sizeof	Output memory box size for a variable
type	Type-casting

Unary Operators - Negative

- Operators that take only one argument (or operand)
  - -5
  - -b
- Observe that – is both an arithmetic and unary operator
  - Meaning depends on context
  - This is called overloading

Unary Operators – Increment and Decrement

- Increment (++) increases a variable by 1
- Decrement (--) – decreases a variable by 1
- Both work on all variable types
- Syntax
  - *++variable* is the pre-increment operator
    - Means increment, then use
  - *variable++* is the post-increment operator
    - Means use, then increment

```
int main(){
      char a = 'A';    float b = 3.31;
      printf("%c\t%f\n",++a,b++);
            printf("%c\t%f",--a,b--);
     return 0;
}
```

```
B       3.31
A       4.31
```

Unary operators - sizeof

- Syntax
  - *sizeof var*
  - *sizeof(type)*
- Returns size of the operand in bytes
  - sizeof(char) will return 1
  - sizeof(float) will (mostly) return 4
- Very useful when you are porting programs across computers

Unary Operators - typecast

- Syntax
  - (type) var
- We have already seen this
- What will be the output of this program?

```
int main(){
        double a = 67.2;
        printf("size is %d\n", sizeof a);
        printf("size is %d\n", sizeof((char) a));
        printf("%c", (char) a);
    return 0;
}
```

```
Size is 8
Size is 1
C
```

Precedence

- Above arithmetic operators, only below brackets

- If a is 1 and b is 2, what will a + -b be evaluated as?
  - -1

Associativity

- From right to left
  - Important to remember this
  - Most other operators' associativity is left to right
- What will this program print?

```
int main(){
      int a = 1;
      printf("%d", - ++a);
     return 0;
}
```

-2

Types of operators

- Arithmetic
- Unary
- **Relational and logical**
- Assignment
- Conditional

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!

Example

- Problem: Input 3 positive integers. Print the count of inputs that are even and odd.
  - Do not use if-then-else

INPUT
10
5
3

OUTPUT
Even=1
Odd=2

```
int a; int b; int c;
int cEven; // count of even inputs
scanf("%d%d%d", &a,&b,&c); // input a,b,c

// (x%2 == 0) evaluates to 1 if x is Even,
//                          0 if x is Odd
cEven = (a%2 == 0) + (b%2 == 0) + (c%2 == 0);
printf("Even=%d\nOdd=%d", cEven, 3236+
-cEven);
0
```

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. Compiler returns 1 by default

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		
2 || 0		
‘A’ && ‘0’		
‘A’ && 0		
‘A’ && ‘b’		
! 0.0		
! 10.05		
(2<5) && (6>5)		

Examples

Expr	Result	Remark
2 && 3	1	
2 || 0	1	
‘A’ && ‘0’	1	ASCII value of ‘0’≠0
‘A’ && 0	0	
‘A’ && ‘b’	1	
! 0.0	1	0.0 == 0 is **guaranteed**
! 10.05	0	Any real ≠ 0.0
(2<5) && (6>5)	1	Compound expr

Precedence and associativity

- Not has same precedence as equality operator
- And and Or are lower than relational operators
- Or has lower precedence than And
- Associativity goes left to right
- 2 == 2 && 3 == 1 || 1==1 || 5==4 is true

Operator Precedence

HIGH

LOW

Operators	Description	Associativity
(unary) + - ++ -- (type) sizeof !	Unary operators (e.g. unary plus/minus), logical Not	Right to left
* / %	Multiply, divide, remainder	Left to right
+ -	Add, subtract	Left to right
< > >= <=	Relational operators	Left to right
== !=	Equal, not equal	Left to right
&&	And	Left to right
||	Or	Left to right

Practice Questions

```
int main(){
        int a = 7;
                printf("%d\n", a++ * a++);
        return 0;
}
```

56

```
int main(){
        int a = 10;
                printf("%d\n", ++a + ++a);
        return 0;
}
```

23

```
int main(){
        int a = 10;
                a = a++;
                printf("%d\n", a);
        return 0;
}
```

10

Types of Operators

- Arithmetic
- Unary
- Relational and logical
- **Assignment**
- Conditional

Assignment Operator

- Basic assignment
  - *variable = expression*

Variant	Meaning
Var += a	Var = Var + a
Var -= a	Var = Var – a
Var *=a	Var = Var *a
Var /=a	Var = Var/a
Var %=a	Var = Var%a

Precedence

- Always the last to be evaluated
  - x *= -2 *(y+z)/3
  - x = x*(-2*(y+z)/3)
- Seldom need to worry about it

Operator Precedence

HIGH

LOW

Operators	Description	Associativity
(unary) + - … !	Unary plus/minus etc, logical Not	Right to left
* / %	Multiply, divide, remainder	Left to right
+ -	Add, subtract	Left to right
< > >= <=	Relational operators	Left to right
== !=	Equal, not equal	Left to right
&&	And	Left to right
||	Or	Left to right
=	Assignment	Right to left

Class Quiz

- What is the value of expression:

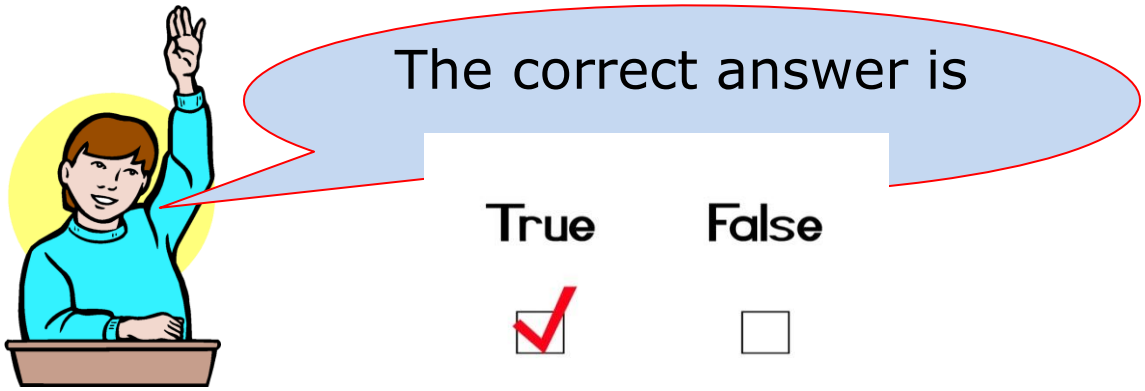
```
0 <= 10 <= 4
```

a) Compile time error

a) Run time crash

a) False (0)

a) True (1)



Types of Operators

- Arithmetic
- Unary
- Relational and logical
- Assignment
- **Conditional**

Conditional Operator

- Special form of assignment
- Syntax
  - *expression 1? expression 2 : expression 3*
  - 2 is executed if 1 is true, 3 is executed if 1 is false
  - Typically paired with assignment
- var min = (f<g) ? f : g

Read two integer number print user and show him the bigger number ?

Operator Precedence

HIGH

LOW

Operators	Description	Associativity
(unary) + - … !	Unary plus/minus etc., logical Not	Right to left
* / %	Multiply, divide, remainder	Left to right
+ -	Add, subtract	Left to right
< > >= <=	Relational operators	Left to right
== !=	Equal, not equal	Left to right
&&	And	Left to right
||	Or	Left to right
? :	Conditional	Right to left
=	Assignment	Right to left

Types of Operators

- Arithmetic
- Unary
- Relational and logical
- Assignment
- **Conditional**

Combine operators to form expressions

Arithmetic library functions

Function	Type	Purpose
abs(i)	int	Return the absolute value of an integer
ceil(d)	double	Round up to closest higher integer value
floor(d)	double	Round down to closest lower integer value
log(d)	double	Return natural log of a number
pow(d1,d2)	double	Return d1 raised to the power d2
rand()	int	Return a random integer
sqrt(d)	double	Return the square root of a number

Have to include the header files stdlib.h and/or math.h for several of these

Typical syntax : *variable = function(variable, variable ....);*
Compile the code as $ cc filec.c -lm

Sample Usage

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

int main() {
  int n;
  double m;
  n = rand() % 10 + 1;          // random number
  printf("%d\n", n);
  m = pow(n,2);                 // raised to 2
  printf("%f\n", m);
  m = sqrt(n);                              // natural log
  printf("%f\n", m);

  return 0;
}
```

```
4
16.00
2.00
```

Practice Program

- Write a C program to read a rupee amount (integer value) and break it up into the smallest possible number of bank notes
  - Assume bank notes are in the denominations 2000, 500, 100, 50, 20, 10 and 5

Do not use if else

Final Program

```
#include <stdio.h>
int main(){
  const int N1 = 2000, N2 = 500, N3 = 100, N4 = 50, N5 = 20, N6 = 10, N7 = 5;
  int notes_1, notes_2, notes_3, notes_4, notes_5, notes_6, notes_7;
  int amount;         // input amount

  scanf("%d",&amount);
  printf("Input amount: Rs %d\n\n", amount);

  // calculations
  notes_1 = amount/N1;    amount = amount%N1;
  notes_2 = amount/N2;    amount = amount%N2;
  notes_3 = amount/N3;    amount = amount%N3;
  notes_4 = amount/N4;    amount = amount%N4;
  notes_5 = amount/N5;    amount = amount%N5;
  notes_6 = amount/N6;    amount = amount%N6;
  notes_7 = amount/N7;    amount = amount%N7;

  printf("%d notes of Rs. %d\n", notes_1, N1);
  printf("%d notes of Rs. %d\n", notes_2, N2);
  printf("%d notes of Rs. %d\n", notes_3, N3);
  printf("%d notes of Rs. %d\n", notes_4, N4);
  printf("%d notes of Rs. %d\n", notes_5, N5);
  printf("%d notes of Rs. %d\n", notes_6, N6);
  printf("%d notes of Rs. %d\n", notes_7, N7);
  return 0;
}
```

Output

Input amount: Rs 4234

2 notes of Rs. 2000
0 notes of Rs. 500
2 notes of Rs. 100
0 notes of Rs. 50
1 notes of Rs. 20
1 notes of Rs. 10
0 notes of Rs. 5

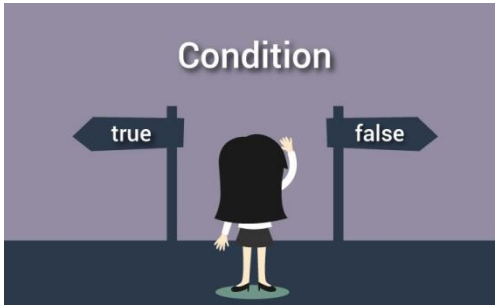
Limitations

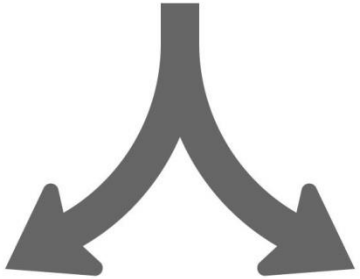
- Using only arithmetic expressions
  - Can’t handle exceptions
  - Can’t handle cases
  - Can’t run the program over and over
- Solutions
  - Conditional statements
  - Switch-case
  - Loops

- Using only unitary variable declarations
  - Have to write out expressions for all variables individually
- Solution
  - Arrays

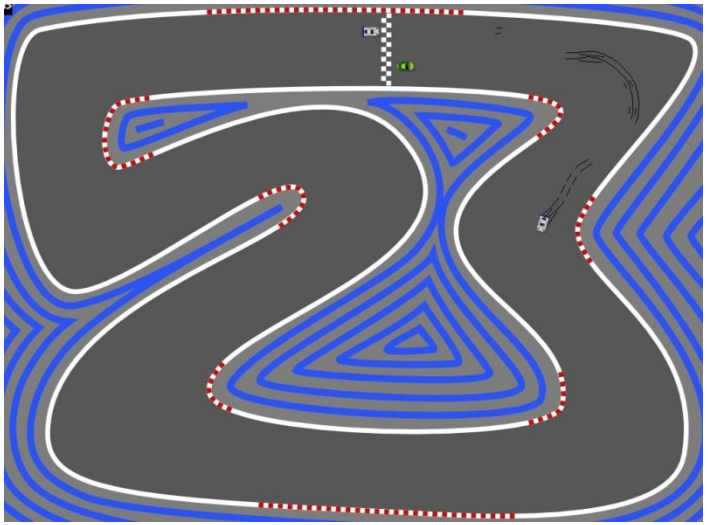
Control Statements

- Branching





- Looping



Control Statements

- Flow of control so far
  - Top to bottom

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

int main() {
  int n;
  double m;

  scanf(“%d”,&n);
  m = log(n);                          // natural log
  printf("%f\n", m);

  return 0;
}
```

Branching: Use Case

- What happens when n is negative?

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

int main() {
  int n;
  double m;

  scanf(“%d”,&n);
  m = log(n);                                  // natural log
  printf("%f\n", m);

  return 0;
}
```

Goto Statement

- Can ask program to skip to a particular arbitrary location in your code
- Syntax –
  - *goto label;*
  - *label: expression;*
- Program control goes to the statement beginning with the label in the goto statement

Solving the Use Case

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

int main() {
  int n;
  double m;
  printf(“Please enter a positive number: ”);
  scanf(“%d”,&n);
  if (n<0)
            goto error;
  m = log(n);                                  // natural log
  printf("%f\n", m);

  error:  printf(“Why can’t you follow instructions?");
  return 0;
}
```

Goto Statement

- A computer programmer is someone who, when asked to go to hell, objects to the use of the word ‘goto’
- (Almost) never use goto
  - Too arbitrary, makes code hard to understand

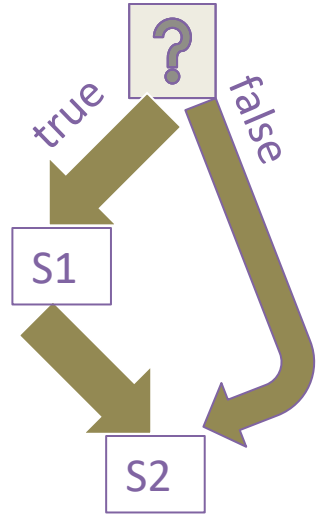
Branching Statements in C

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

If Statement

- General form of the if statement

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

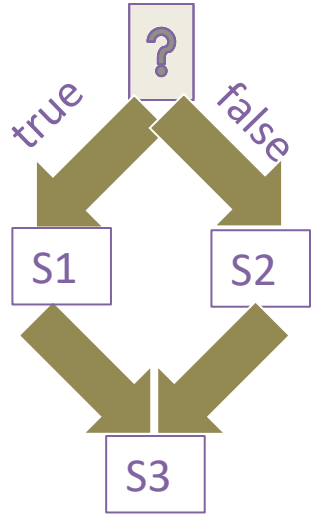


- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a non-zero value, then S1 is executed and then control (program counter) moves to the statement S2.
  - If expression evaluates to 0, then S2 is executed.

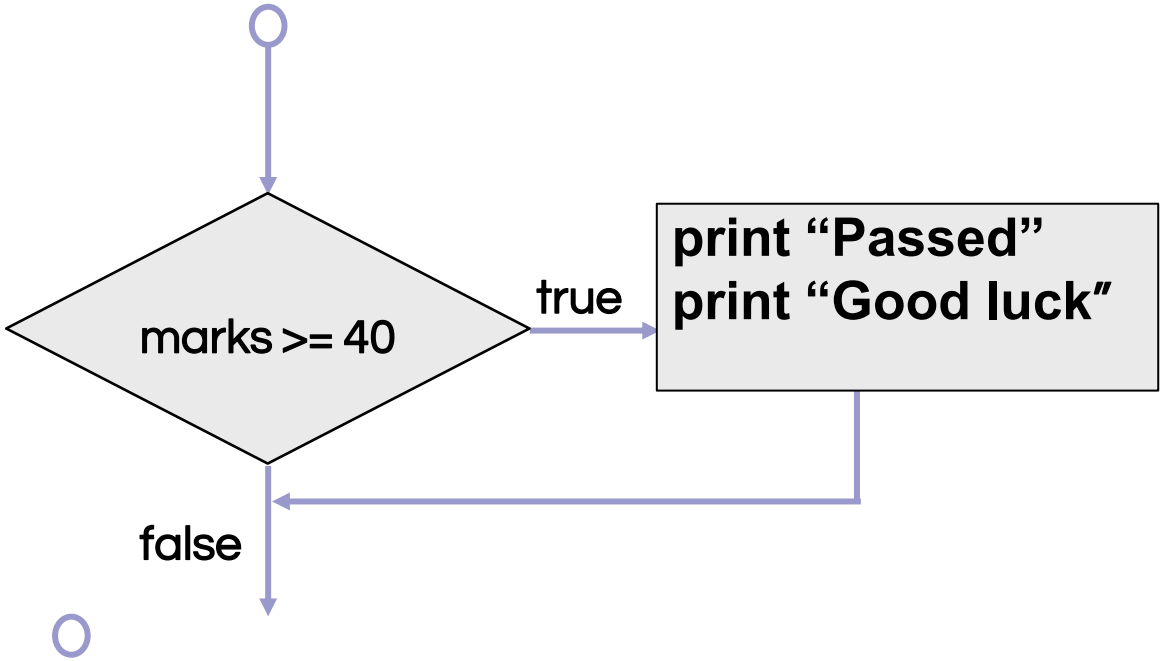
if-else Statement

- 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 (marks >= 40)  {
.    printf(“Passed \n”);
.    printf(“Good luck\n”);
. }
. printf (“End\n”) ;
```

Solving the Use Case with if-else

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

int main() {
  int n;
  double m;
  printf(“Please enter a positive number: ”);
  scanf(“%d”,&n);
  if (n>0)
            m = log(n);                                  // natural log
            printf("%f\n", m);
  else
            printf(“Why can’t you follow instructions?");

return 0;
}
```

Something’s not quite right in this code

Compound Statements

- A block of code containing zero or more statements
- Contained between { and }
- Format:

```
{
  Local Declarations

  Statements
}
```

Getting it Right

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

int main() {
  int n;
  double m;
  printf(“Please enter a positive number: ”);
  scanf(“%d”,&n);
  if (n>0){
            m = log(n);                                  // natural log
            printf("%f\n", m);
  }
  else
            printf(“Why can’t you follow instructions?");
  return 0;
}
```

Good Indentation Practice

- This is a main statement
  - This is a dependent statement
- Main statements are statements in the main control flow of your program
  - Dependent statements branch off from the main flow
  - **Indent them**, for easier understanding of code
  - Matters more in some languages, like python
- Pro-tip: use 4 spaces instead of tab to indent

if-else Statement

- Read two integers and print the min.

```
# 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. Check 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

1
0

Output

6

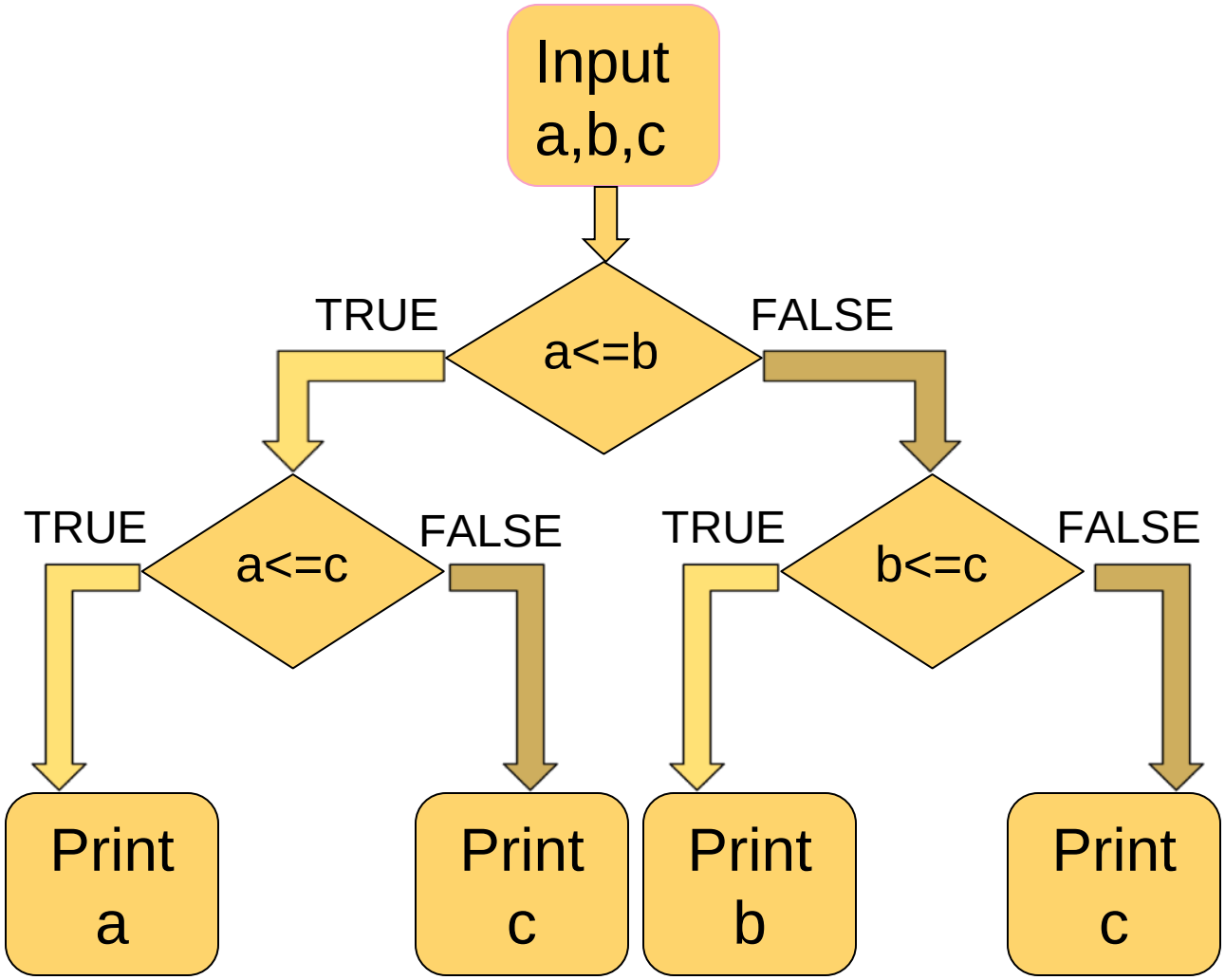
Nested if-else

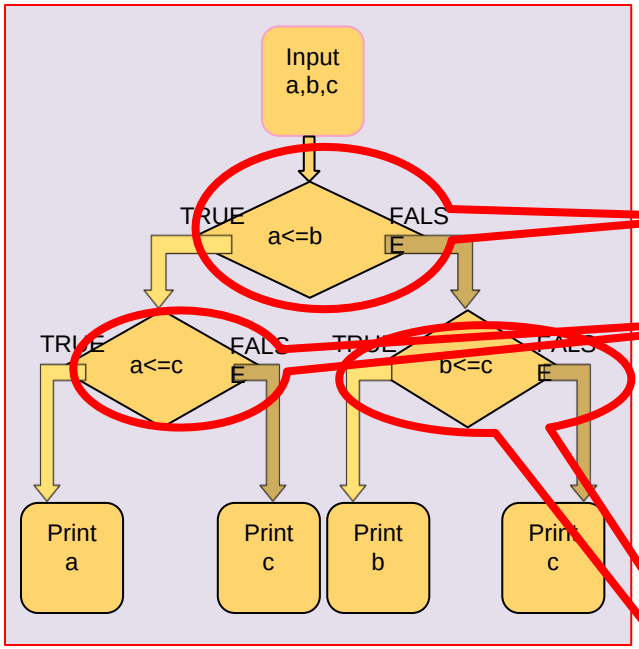
```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int c,d;
    printf("Do you like C? (0/1) ");
    scanf("%d",&c);
    if(c){
        printf("Do you really like C? (0/1) ");
        scanf("%d",&d);
        if(d)
            printf("You really like C!!");
        else
            printf("Not really");
    }
    else{
         printf("Ohh, You hate C!!");
    }


    return 0;
}
```

Nested if-else

- If e1 s1 else if e2 s2
- If e1 s1 else if e2 s2 else s3
- If e1 if e2 s1 else s2 else s3
- If e1 if e2 s1 else s2
- Rule of thumb: *else* clause looks for the closest previous *if* without an *else*

Finding min of 3 numbers





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

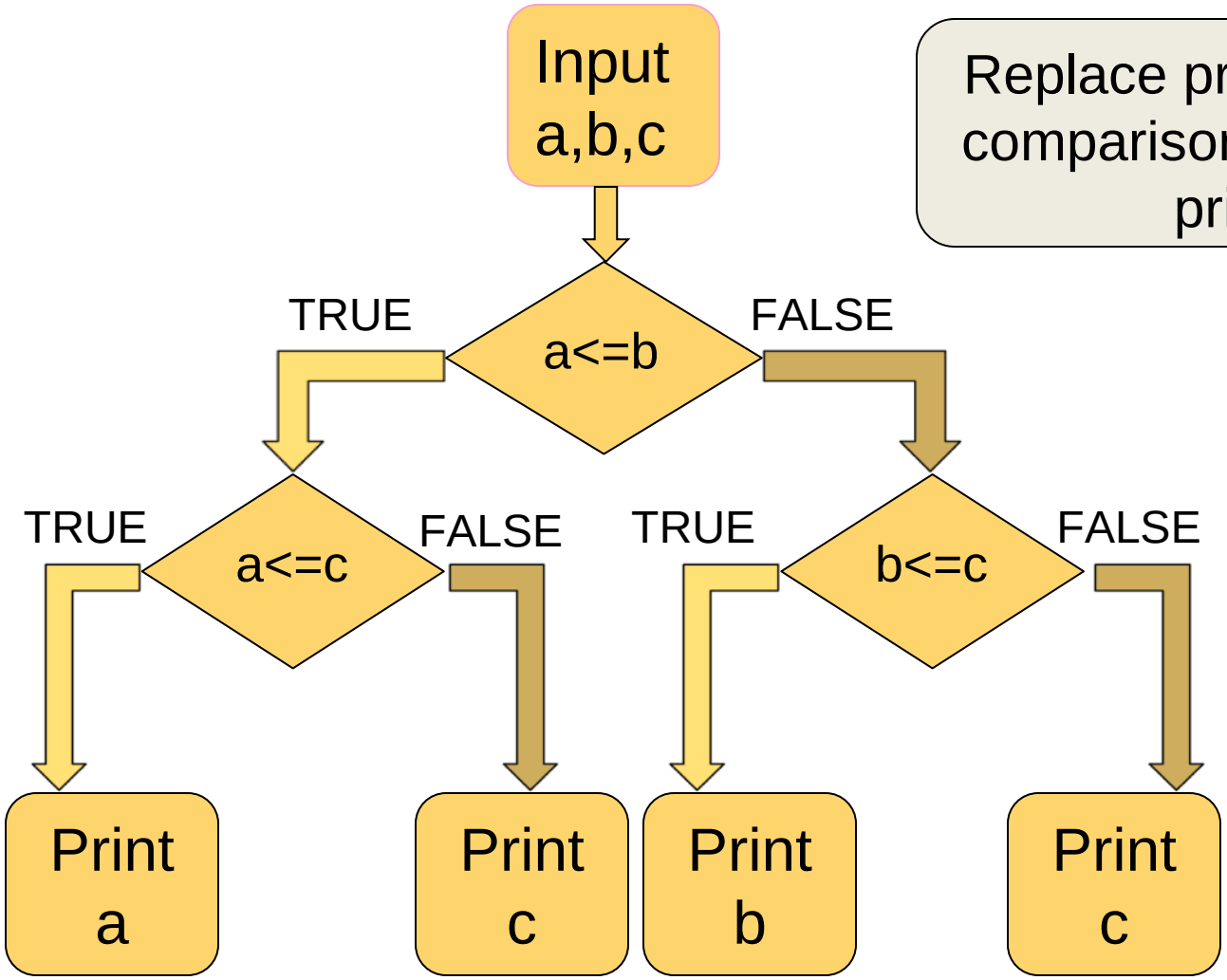
```
int a,b,c;
scanf("%d%d%d",&a,&b,&c);
if (a <= b) {
        if (a <= c) {
                printf("min = %d",a);
        }
        else {
                printf("min = %d", c);
        }
}
else {
        if (b <= c)  {
                printf("min = %d", b);
        }
        else {
               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

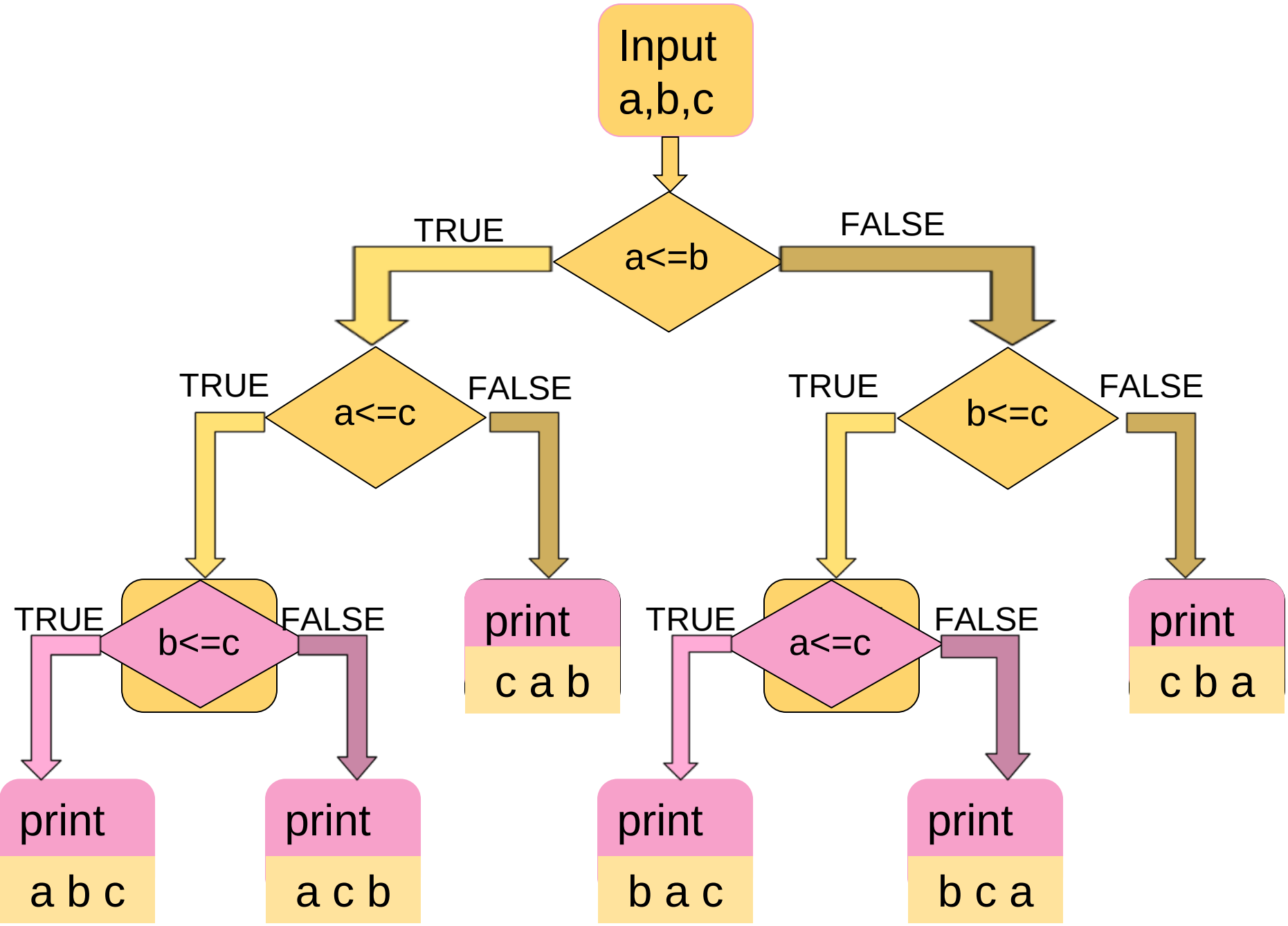
- Problem: read three numbers into a, b and c and print them in ascending order.
  - Start with the flowchart for finding minimum of three numbers and add one more level of conditional check.
  - Then translate the flowchart into C program.

Finding min of 3 numbers



Replace print by more comparisons and then print.

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.

Confusing Equality (==) and Assignment (=) Operators

- Dangerous error
  - Does not ordinarily cause syntax errors
  - Any expression that produces a value can be used in control structures
  - Nonzero values are true, zero values are false
- Example:

```
if ( payCode = 4 )
        printf( "You get a bonus!\n" );
```

If-else Review

```
#include<stdio.h>
 int main() {
            int i = 5, j = 6, k = 7;
            if(i > j == k)
                        printf("%d %d %d", i++, ++j, --k);
             else
                        printf("%d %d %d", i, j, k);
             return 0;
 }
```

5 6 7

```
#include<stdio.h>
 int main() {
             int i = 5;
             if(i = i - 5 > 4)
                        printf("inside if block");
             else
                        printf("inside else block");
            return 0;
 }
```

Inside else block

If-else Review

```
#include<stdio.h>
 int main() {
            if(sizeof(0))
                        printf(“Yes");
            else
                        printf(“No");
            return 0;
 }
```

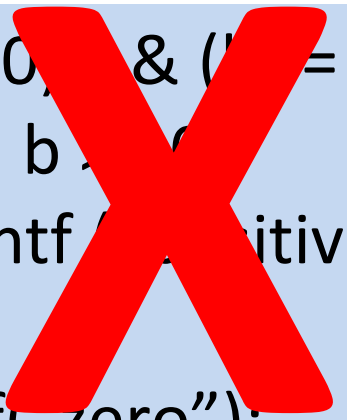
Yes

```
#include <stdio.h>
int main()
{
    char val=1;
    if(val---=0)
        printf("TRUE");
    else
        printf("FALSE");
    return 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

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

OUTPUT for a = 5, b = 0

NO OUTPUT!!

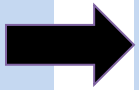
OUTPUT for a = 5, b = -5

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
        …
```

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 , where , print the name of the weekday corresponding to .

  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 , where , print Weekday, if the 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);
}
```

The Conditional Operator ?:

- This makes use of an expression that is either non-0 or 0
  An appropriate value is selected, depending on the valu
  of the expression
- Example: instead of writing

```
if (balance > 5000)
        interest = balance * 0.2;
else interest = balance * 0.1;
```

We can just write

```
interest = (balance > 5000) ? balance * 0.2 : balance * 0.1;
```

The Conditional Operator ?:

- This makes use of an expression that is either non-0 or 0
  An appropriate value is selected, depending on the valu
  of the expression
- Example: instead of writing

```
if (balance > 5000)
        interest = balance * 0.2;
else interest = balance * 0.1;
```

We can just write

```
interest = (balance > 5000) ? balance * 0.2 : balance * 0.1;
```

Switch-Case Statement

- Multi-way decision
- Checks whether an expression matches one out of a number of constant integer 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

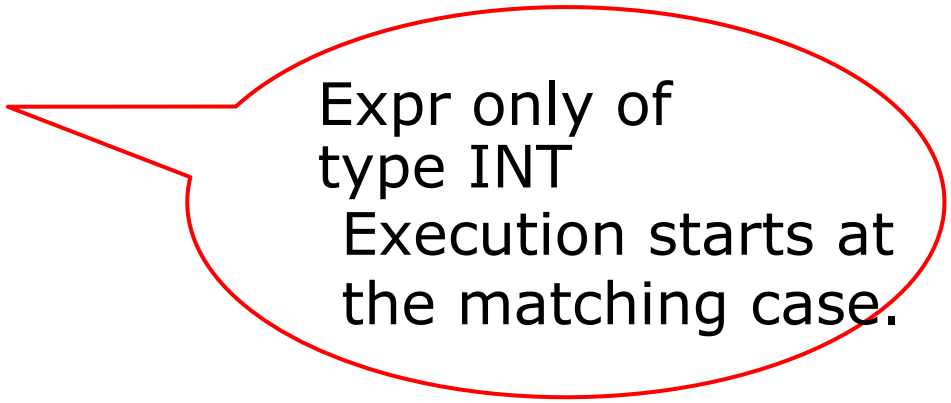
```
switch (day) {
case 1:
case 7: printf (“Weekend”); break;
case 2:
case 3:
case 4:
case 5:
case 6: printf (“Weekday”); break;
default: printf (“ Illegal day %d”, day);
}
```

The *break* Statement

- Syntax –
  - break;
- Used to break out of current code branch
  - Not to end the program

General Form of switch-case

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



- **default** is optional. (= *remaining cases*)
- The location of **default** does not matter.
- The statements following a case label are executed one after other until a **break** is encountered (**Fall Through**)

Example

- Switch expressions can only be of type *int*

```
#include<stdio.h>
 int main() {
            char ch = 65;
            switch(ch) {
                        case 'A': printf("Apple");
                        break;
                        case 'B': printf("Bing");
                        break;
                       default: printf("Bye");
                        break;
             }
            return 0;
 }
```

Apple

Fall Through…

```
int n = 100;
int digit = n%10; // last digit
switch (digit) {
default : printf("Not divisible by 5\n");
          break;
case 0: printf("Even\n");
case 5: printf("Divisible by 5\n");
        break;
}
```

What is printed by the program fragment?

Answer:

Even

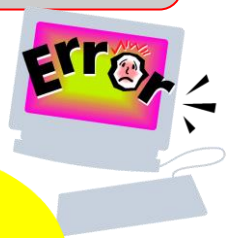
Divisible by 5;

Class Quiz

- What is the value of expression:

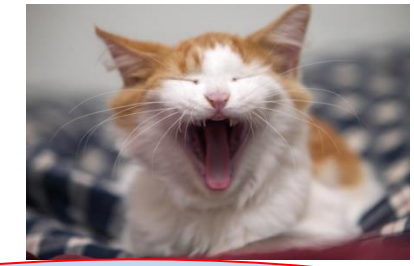
(5<2) && (3/0)

a) Compile time error



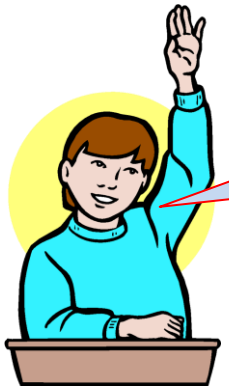
a) Run time crash

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



a) 0

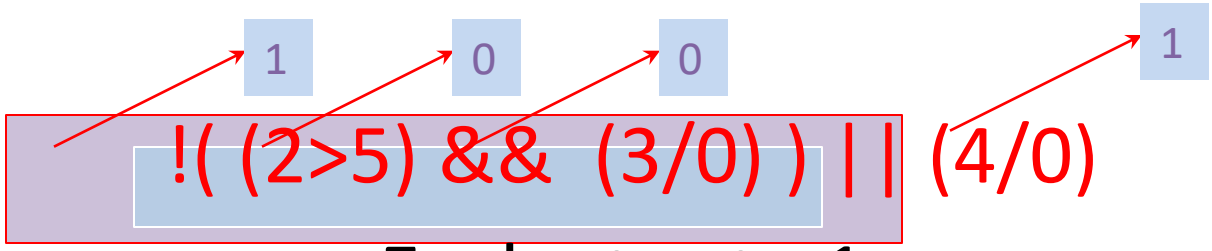
a) 1



The correct answer is

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



Evaluates to 1

3 Factors for Expr Evaluation

- Precedence
  - Applied to two different class of operators
  - + and *, - and *, && and >, + and &&, …
- Associativity
  - Applied to operators of same class
  - * and *, + and -, * and /, …
- Order of evaluation
  - Precedence and associativity identify the operands for each operator (Parenthesization)
  - Not which operand/expr is evaluated first
- In C, order of evaluation of operands is defined only for && and ||

Value Computation and Side Effect

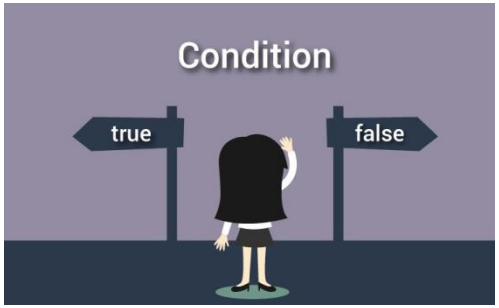
```
int main(){
	int a = 10;
		a = a++;
		printf("%d\n", a);
	return 0;
}
```

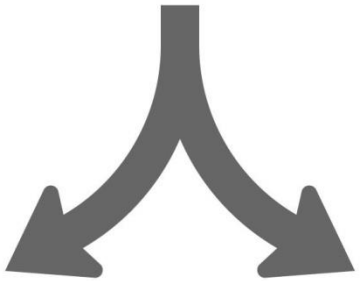
10

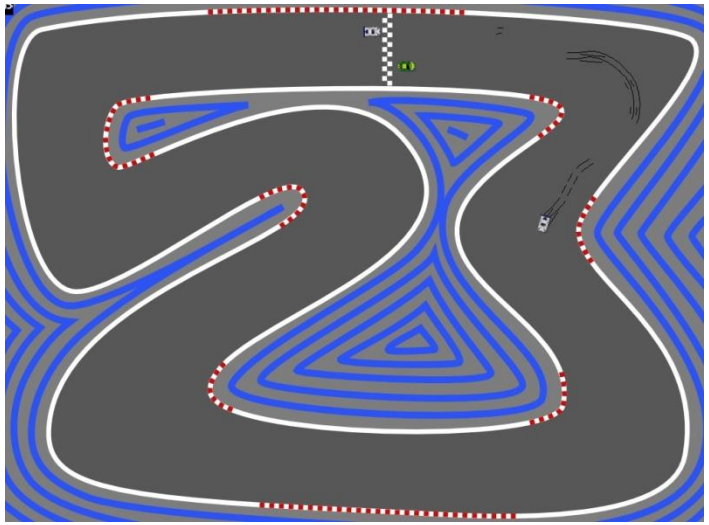
```
int main(){
	int a = 10;
	int temp;
	temp = a;
	a = a + 1
		a = temp;
		printf("%d\n", a);
	return 0;
}
```

Control Statements

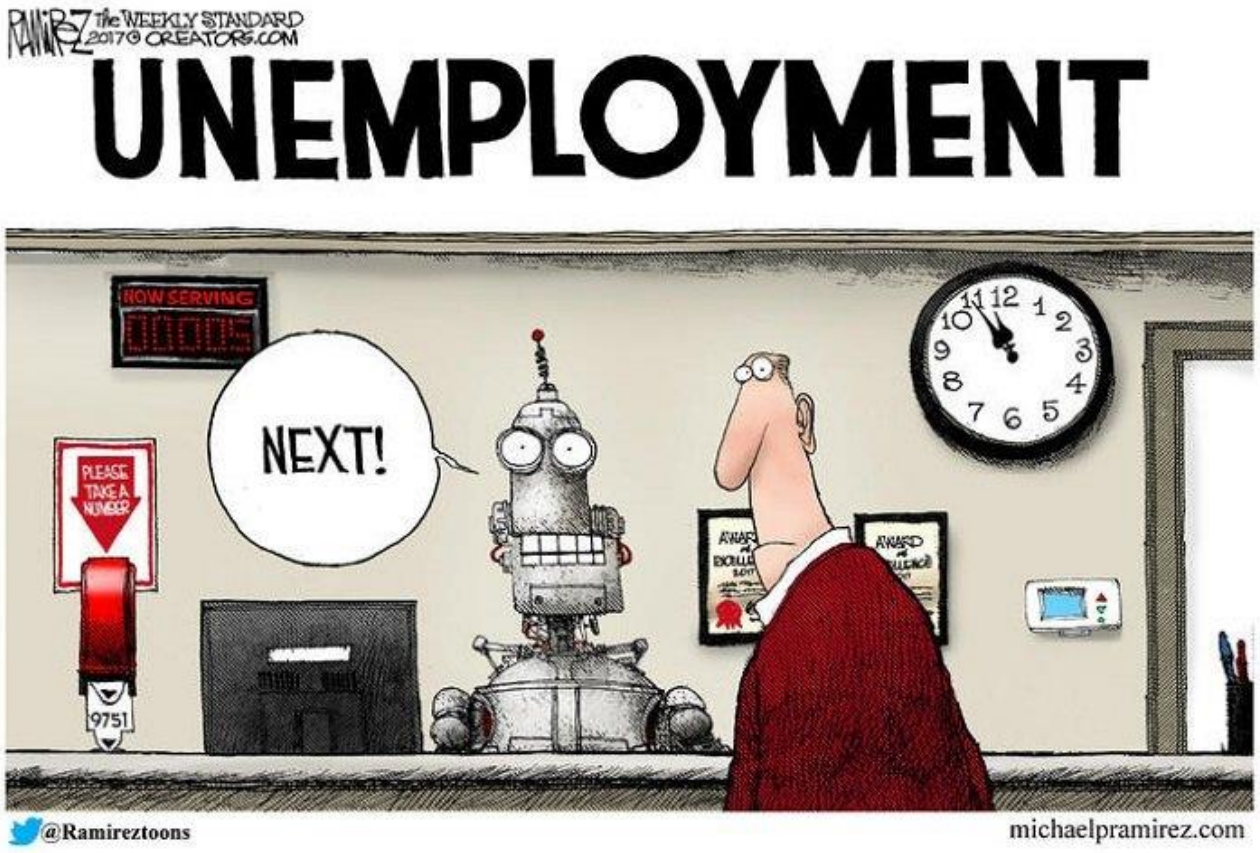
- Branching ✓
- Looping







The Computer's BIG Advantage



Once you've told a computer how to do something once, it can reproduce the exact same process a trillion times.

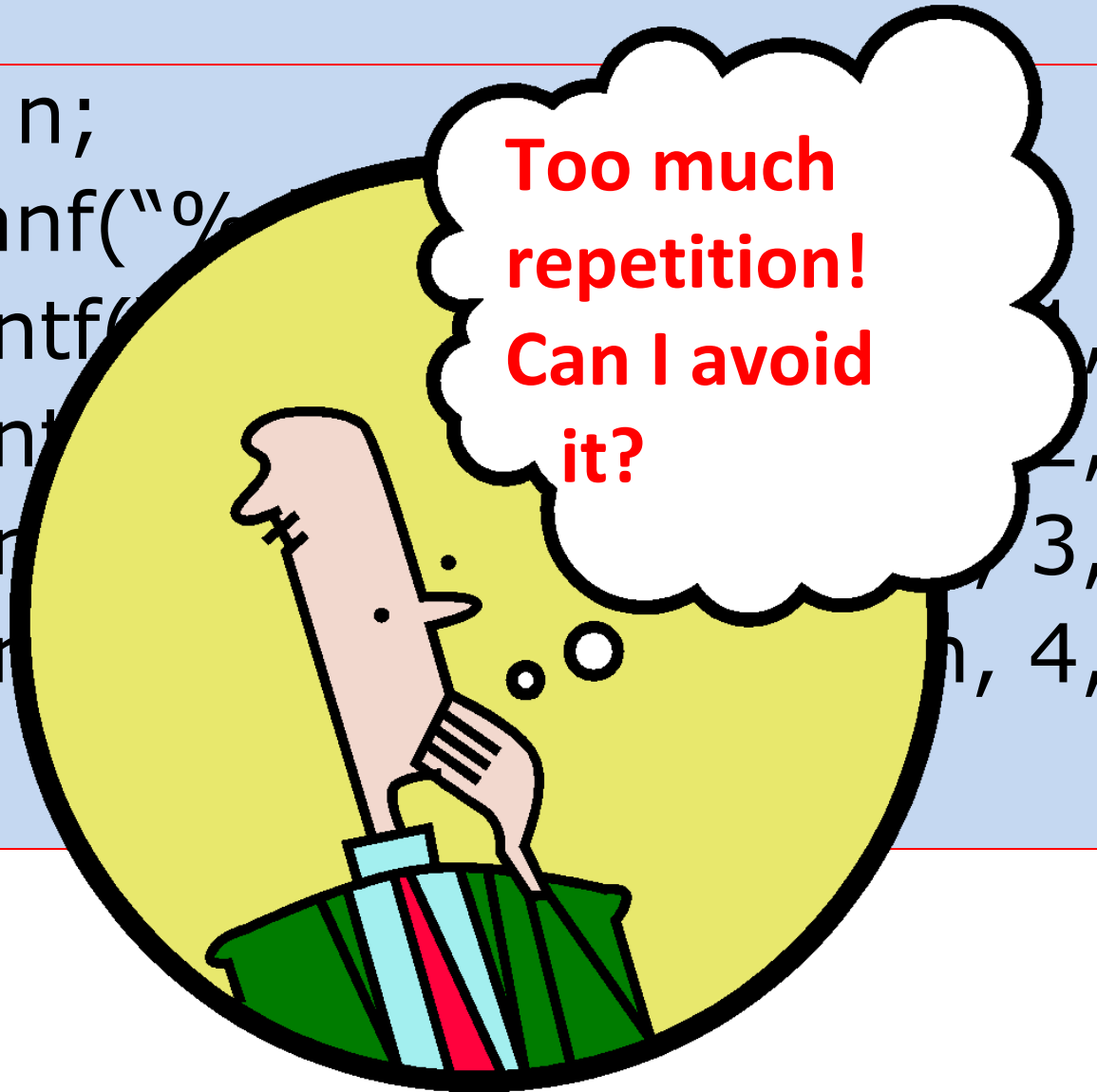
ATM

Printing Multiplication Table

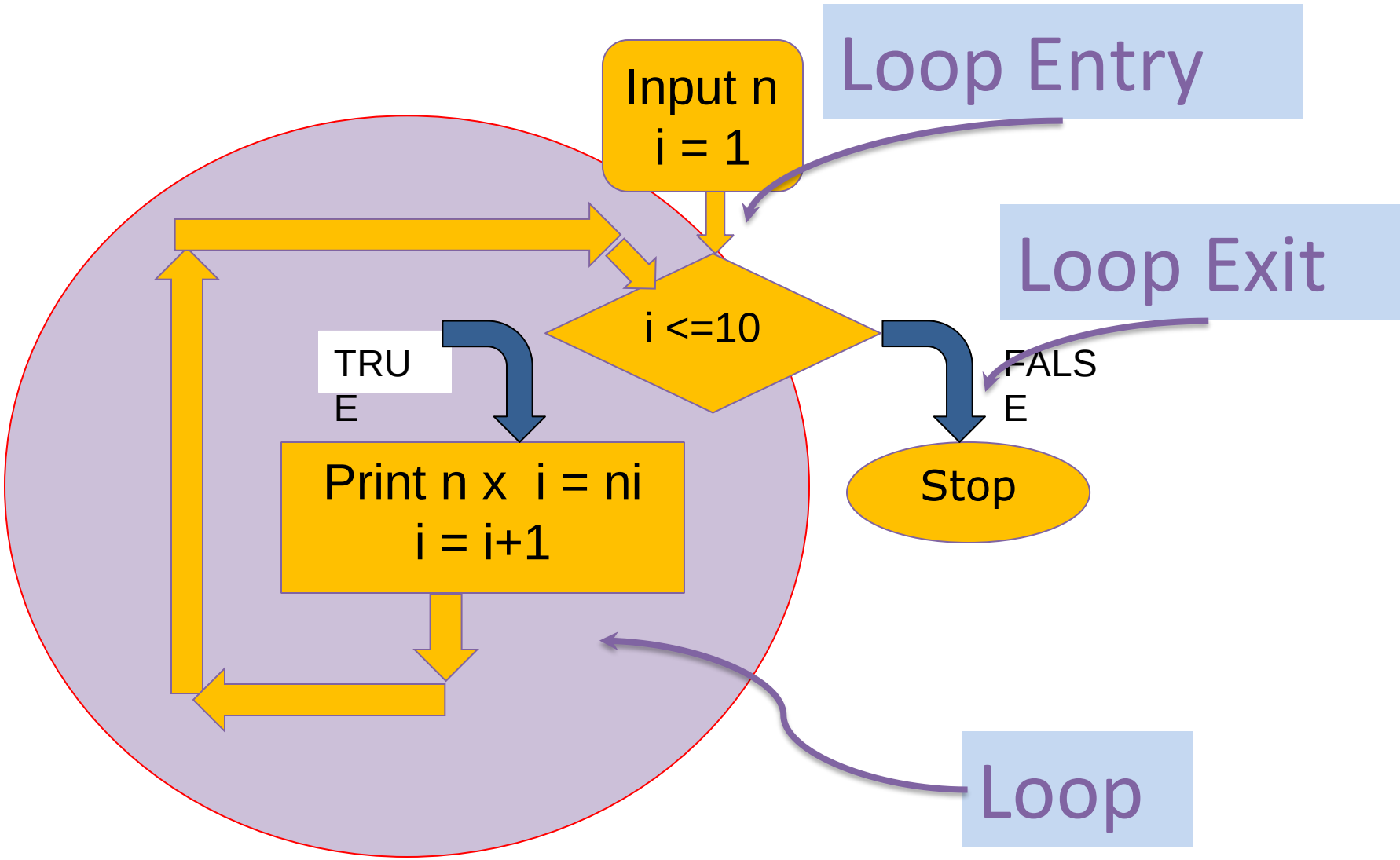
5	X	1	=	5
5	X	2	=	10
5	X	3	=	15
5	X	4	=	20
5	X	5	=	25
5	X	6	=	30
5	X	7	=	35
5	X	8	=	40
5	X	9	=	45
5	X	10	=	50

Program…

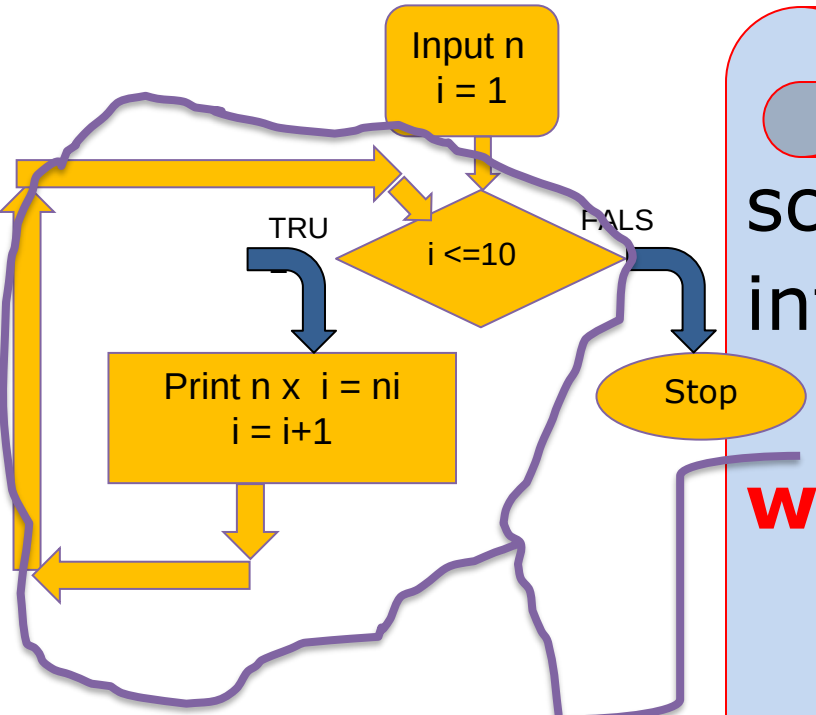
```
int n;
scanf("%
printf(                                  , n*1);
print                                    , n*2);
prin                                     , 3, n*3);
prin                                     n, 4, n*4);
....
```



Printing Multiplication Table



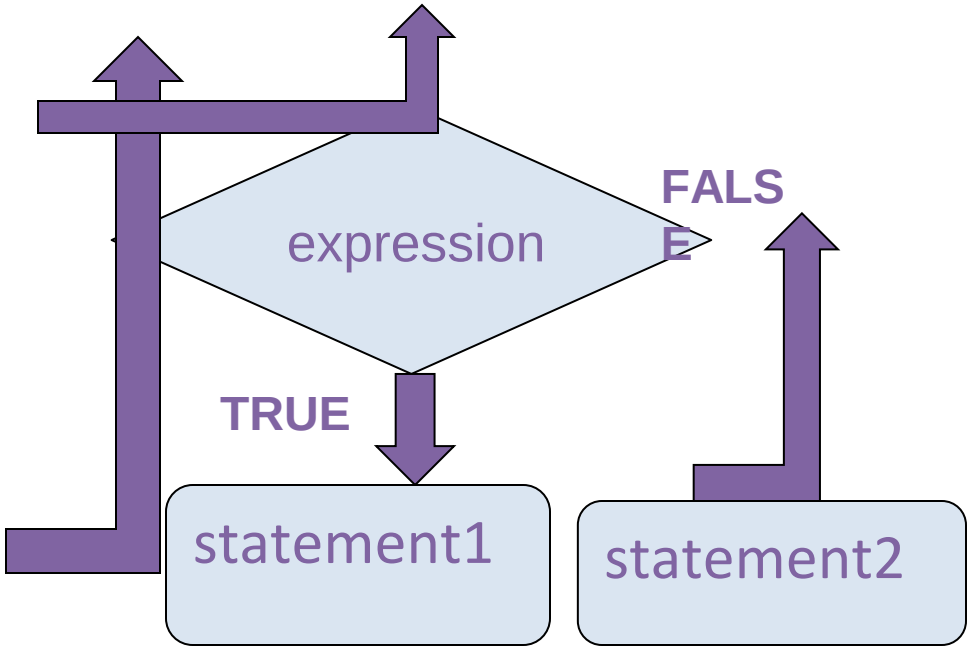
Printing Multiplication Table




```
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;
```



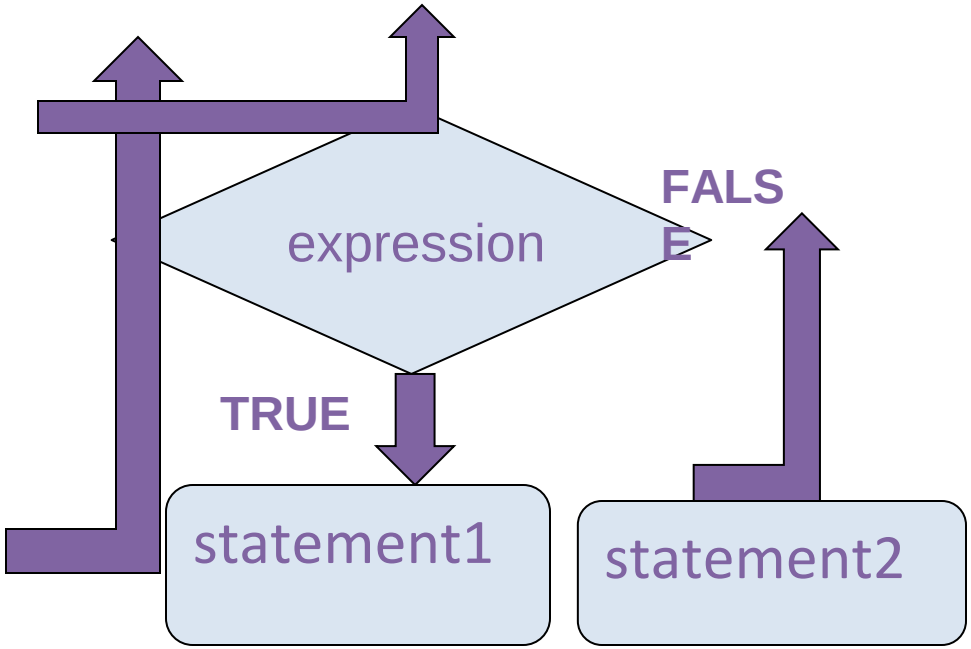
Read in English as:

As long as expression is TRUE execute statement1.

when expression becomes FALSE execute statement 2.

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.

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

-1

Trace of memory location a

- **One scanf is executed every time body of the loop is executed.**
- **Every scanf execution reads one integer.**

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; // not seen any a yet
scanf("%d", &a);     // read into a
while (a != -1) {
    s = s + a;  // last a is not -1
    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.
  - 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)

Program

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

  x = 1;              // [while] initialization
  while ( x <= n) {   // [while] condition
    if ((x%3 == 0) || (x%5 == 0)) { // [if] condition
      printf("%d\n", x);
    }
    x = x+1;          // [while] update
  }
  return 0;
}
```

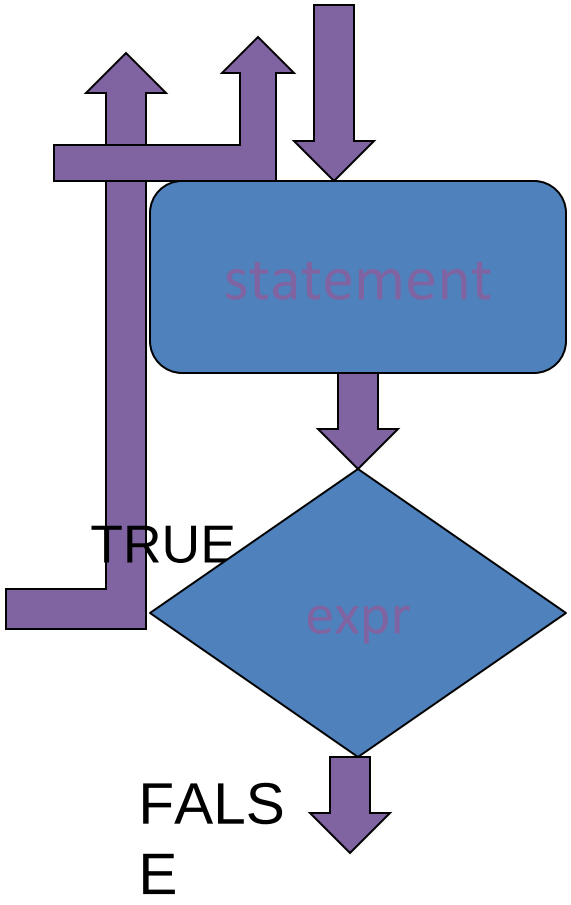
do-while Loops

- 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);
```

For loop in C

- General form

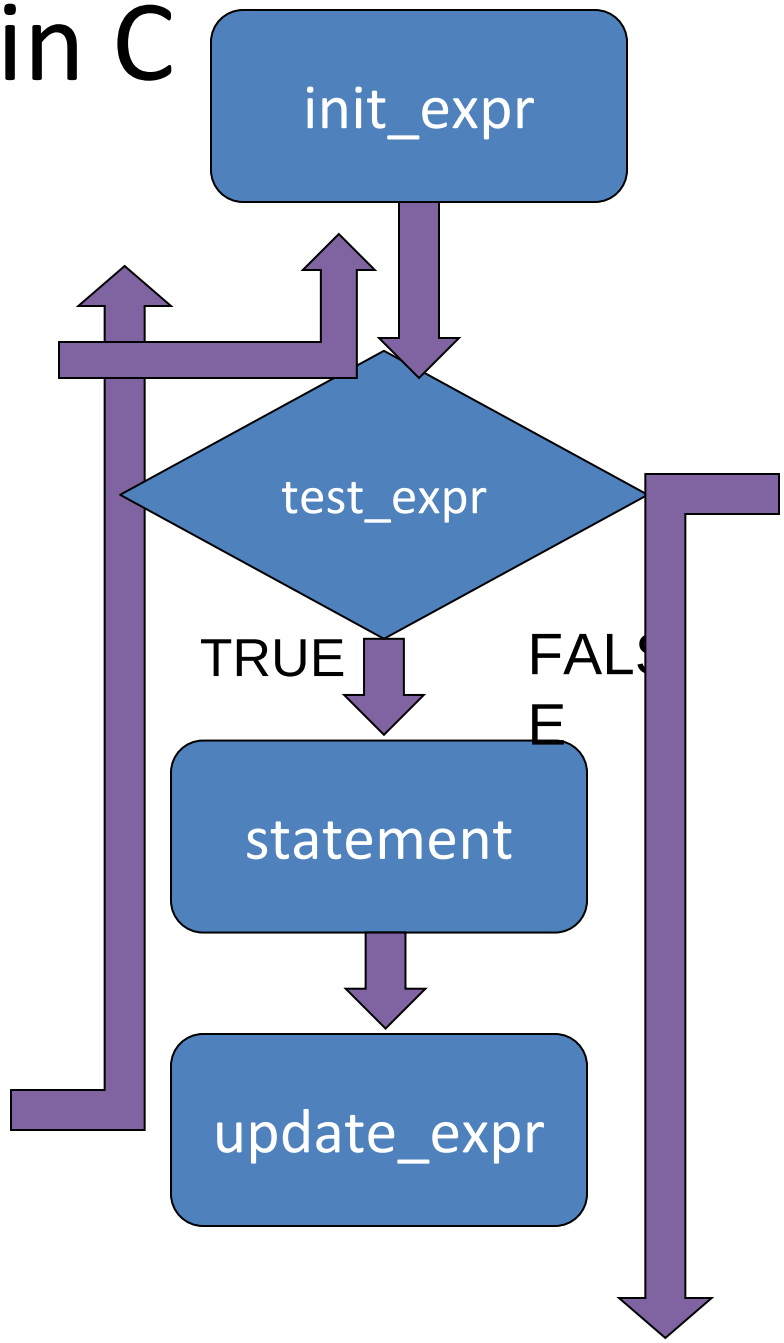
```
for (init_expr; test_expr; update_expr)
    statement;
```

- 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).
- statement is the work to repeat (can be multiple statements in {…})

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



For Loop

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

```
int i; // counter from 1..100
float rsum = 0.0;// the sum

// the for loop
for ( i=1; i<=100; i=i+1 ) {
    rsum = rsum +  (1.0/i);
}
printf("sum is %f ", rsum);
```

```
int i;
float rsum = 0.0;

for (i=1; i<=4; i=i+1) {
rsum = rsum + (1.0/i);
}

printf("sum is %f", rsum);
```

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

sum is 2.083333

sum of reciprocals of 1 to 4 is 2.083333 $

for Loop in Terms of while Loop

```
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 (almost) 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.

Program: Geometric Progression

```
#include <stdio.h>

int main(void) {
   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++)
   {
      printf(%f\n, term);   // Displaying the i-th term
      term = term * r;      // Computing the (i+1)-th term
   }
   return 0;
}
```

Program: Geometric Progression

```
#include <stdio.h>

int main(void) {
   int n, i;
   float r, a, term;

   // Reading inputs from the u
   scanf("%f", &r);
   scanf("%f", &a);
   scanf("%d", &n);

   term = a;
   for (i = 1; i <= n; i++)
   {
      term = term * r;      // Computing the (i+1)-th term
      printf(%f\n, term);   // Displaying the (i+1)-th term
   }
   return 0;
```

Careful: Changing the order of the statements will change the meaning of the program.

a ar ar^2 ar^3 …. Vs

ar ar^2 ar^3 ar^4 ….

For loop in C

- General form

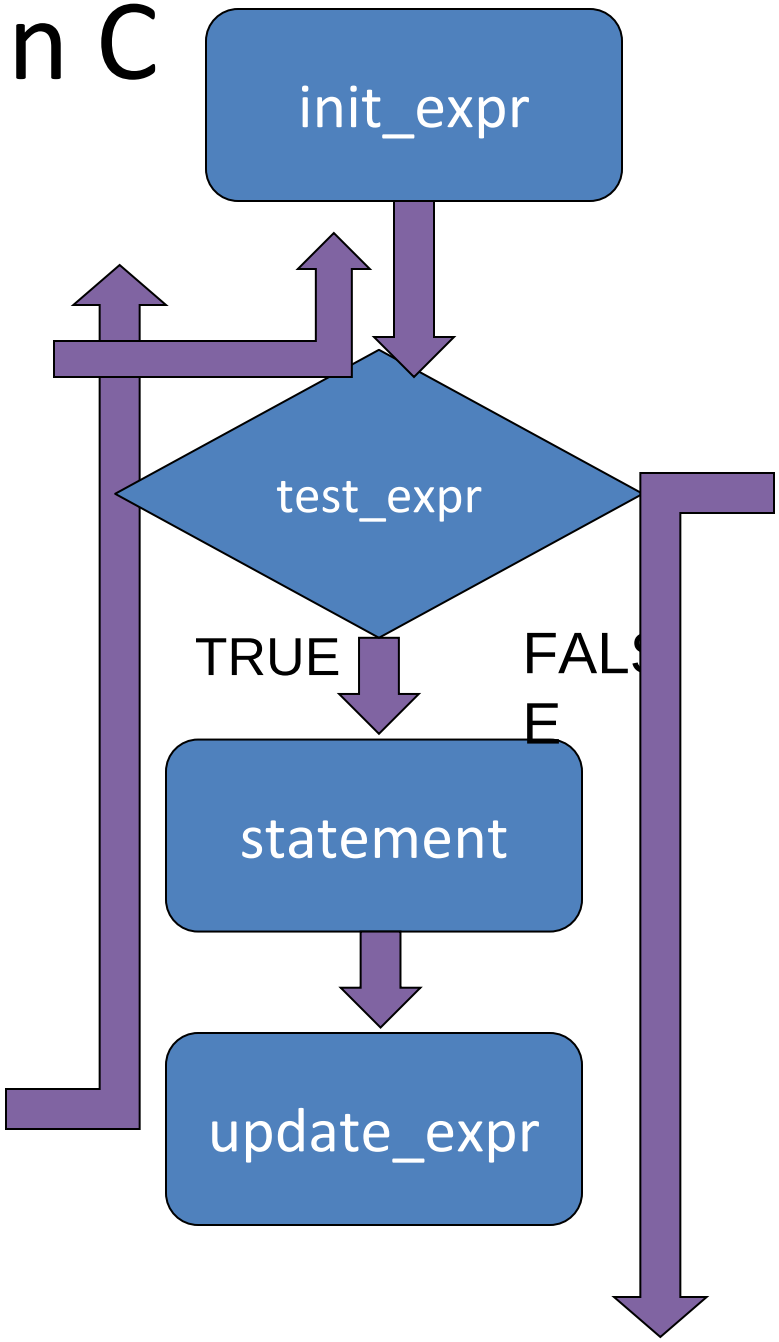
```
for (init_expr; test_expr; update_expr)
    statement;
```

- 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).
- statement is the work to repeat (can be multiple statements in {…})

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



For loop in Action

```
#include <stdio.h>
int main() {
   int i;
   for (i=1;i<=3; i++) {
      printf("%d\t",i)
   }
   printf("\n%d\t",i);

   return 0;
}
```

```
1 2 3
4
```

```
#include <stdio.h>
int main() {
   int i;
   for (i=1;i<=3; ++i) {
      printf("%d\t",i);
   }
   printf("\n%d\t",i);

   return 0;
}
```

```
1 2 3
4
```

Important Consideration: Scope

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

 for (int i=1;i<=2;i++)
          printf("%d\n",i);
 return 0;
}
```

- Output?

1

2

Important consideration: Scope

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

 for (int i=1;i<=2;i++)
          printf("%d\n",i);
 printf("%d\n",i);
 return 0;
}
```

- Output?

Compiler error: 'i' undeclared

Block Scope of a Variable

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

 { //start block
 int i;
 for (i=1;i<=2;i++)
          printf("%d\n",i);

 } //end block

 return 0;
}
```

- Output?

  1

  2

Block Scope of a Variable

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

 {
 int i;
 for (i=1;i<=2;i++)
          printf("%d\n",i);
 }
 printf("outside %d\n",i);

 return 0;
}
```

- Output?

Compiler error: 'i' undeclared

Block scope of a Variable

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

          int j=0;

     printf("j=%d\n",j+1);

 }

 return 0;
}
```

- Output?

1

j=1

2

j=1

Flexibility in Expression Checking

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

```
1 2 3
4 3
```

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

```
1 gbg
```

Undesirable Flexibility

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

```
1  4200288
```

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

```
1    2    3
4    3
```

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 5 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

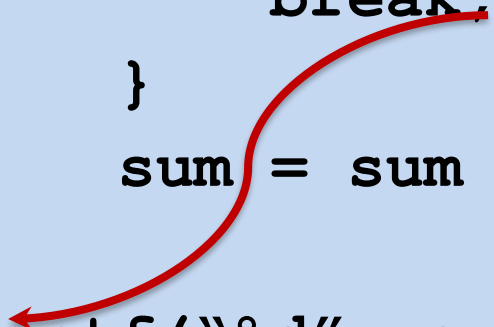
5 6 7 8 9

Back to 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);
```



When to Break?

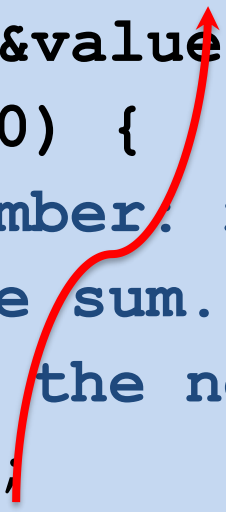
- Use of break sometimes can simplify exit condition from loop.
- However, it can make the code a bit harder to read and understand.
- Tip: if the loop terminates in at least two ways which are sufficiently different and requires substantially different processing then consider the use of termination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

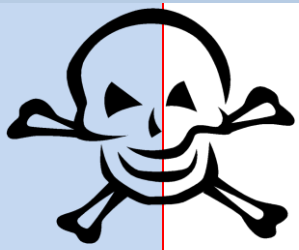
```
for (i = 0; i < 100; i++) {
   for (j = 0; j < 100; j++) {
     if (…) break;
   }
   …
}
```

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

Continue and Update Expr

- Correct Code:

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

Big difference between *for* and *while*

```
#include <stdio.h>
int main() {
  int i = 0;
  for(i = 0; i < 10; i++) {
      if(i == 5) continue;
      printf("%d\n", i);
  }
  return 0;
}
```

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

```
#include <stdio.h>
int main() {
  int i = 0;
  while(i < 10) {
    if(i == 5) continue;
    printf("%d\t", i);
    i++;
  }
  return 0;
}
```

```
0  1  2  3  4  .....................
```

For loop in Terms of while Loop

```
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
- Both are equivalent in power.
- Which loop structure to use?

Loop Best Practices

- I want to do something a times
  - for(i = 0; i < N; i++)

- I want to do something an indeterminate number of times until a condition is true
  - while (condition)

How Many Times is the Loop Executed?

```
int a = 10 - 6;
while (a < 10) {
        if (a = 5)  {
                printf("%d\n", a);
        }
        a=a+1;
}
```

Output
5
5
5
...

A common bug

Probable intention:

```
int a =10 - 6;
while (a < 10) {
        if (a == 5)  {
                printf("%d", a);
        }
        a=a+1;
}
```

Output
5

ACKNOWLEDGEMENTS

Slides were inspired from Prof Rajat Moona and Prof. Amey Karkare for the course ESC101 in IIT Kanpur.