**CSE 102**

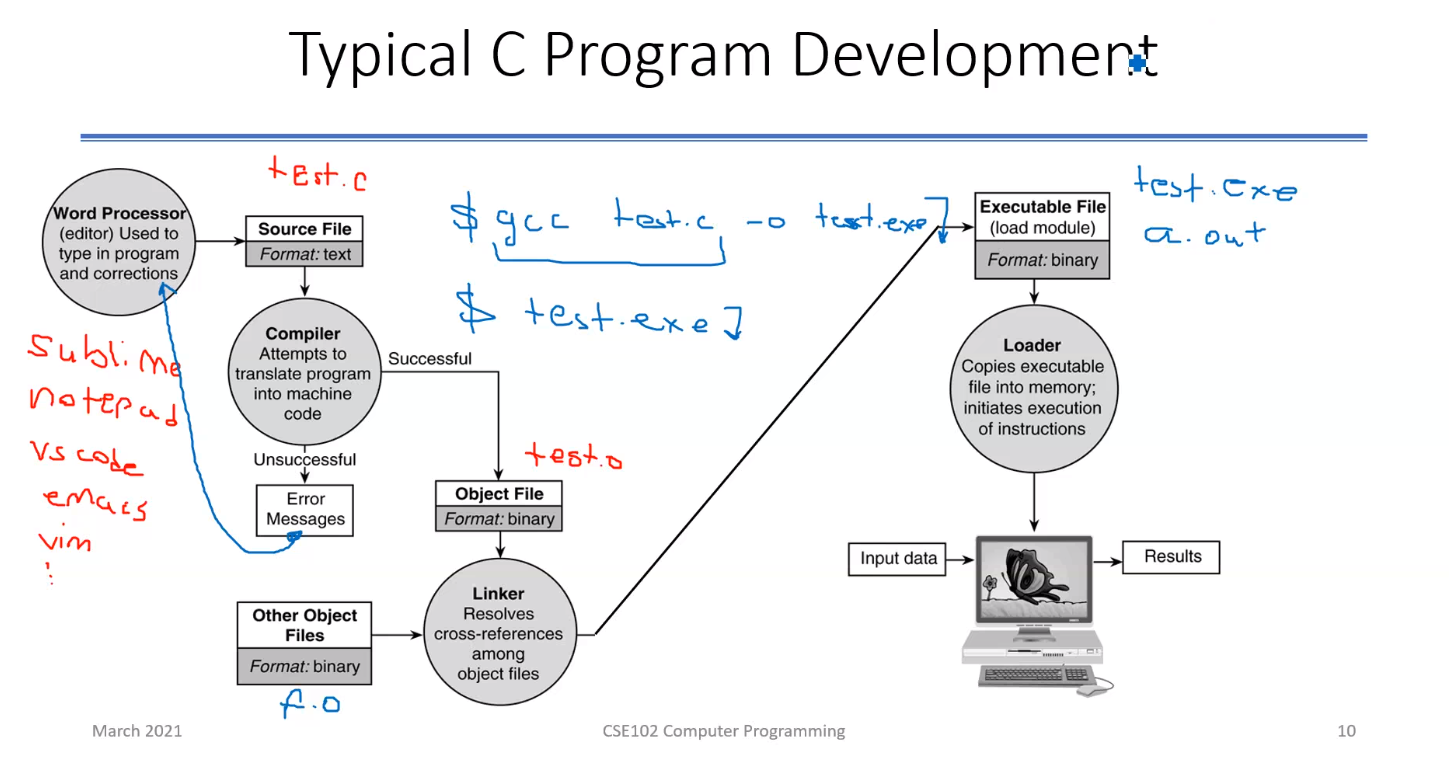
C evolved by Ritchie from 2 previous programming languages: BCPL and B.

C is used to develop UNIX.

C is used to write modern operating systems.

C is hardware independent (portable).

By late 1970s, C had evolved to “Traditional C”.

Standard created in 1989, updated in 1999.

test.c ---> text file : only characters in them

test.o ---> object file : file that your CPU could understand

linker ---> takes your code and other code (library), puts them together. Linker is able to whatever you are referring in test.o is actually extracted, taken and then it will combine them. Linker then will put them in a form that everything is fits together and it’s ready for OS to run as a program. Linker makes object files ready to load to the memory. Linker creates executable files. Linker makes your object file ready to be added with other stuffs that it needs to be loaded to the memory.

It’s OS’s job to put it into the right place in memory, schedule the program, then CPU points to the right place.

gcc test.c | -o test.exe ---> You can break it up if you want to.

gcc test.c -o test.o ---> Does compilation and creates object file (test.o).

gcc test.o -o a.out ---> Does the linking part.

gcc test.c -o test.exe ---> Does compiling and linking in one step. For single one code, this is better.

You can also add -lm command which let linker know that you need that library. So library will be added to the output (executable) file.

Some libraries require you to specify that to the linker, some will be assumed by default.

test.exe ---> I’m telling to OS to actually load my binary code, executable code to the memory and run it.

x = 10 + 3; ---> let’s say this is your code

gcc knows what x assignment 10+3 is. It understand your language. It can converts this to specific machine language. Move, Add… are instructions that CPU can understand.

Below box is machine code that converted from your code by gcc. CPU can understand this.

|  |
| --- |
| Move 10, R1  Move 3, R2  Add R1, R2  Move R2, address(x) |

(Add stores R1+R2 in R2.)

This is not enough. There is an easier representation of this one. This will be represented by very simple binary numbers. This is simple. For example:

F0 0A 11 = Move 10, R1

.o file includes F0 0A 11. This file includes binary representation of what machine can understand.

SHORTLY gcc takes your statement and converts it to assembly language that your target can understand. And it will not be kept in this way because it is not efficient. Efficient representation is just binary numbers.

Then linker takes your .o file and another .o file that you need and takes these two and compiles them to a cohesive one program.

SOFTWARE DEVELOPMENT

Programming = problem solving

Methodology:

* Specify the problem requirements
* Analyze the problem
* Design an algorithm
* Implement the algorithm
  + Writing the algorithm in C by converting each step into statements of C.
* Test and verify the program
  + Run the program for several input cases.
* Maintain and update the program
  + Keep the program up-to-date.

**Problem requirements**

Statements of the problem:

* Understand the problem
* Retrieve the requirements
* Eliminate unimportant aspects

May need to get information from specialists.

**Analysis**

Identify:

* Input data
* Output data
* Additional requirements and constraints

Decide aspects of data:

* Representation
* Relationship

E.g.:

* Input: distance on miles
* Output: distance on kilometers
* Representation: floating point numbers
* Relationship: 1 mile = 1.609 kilometers

**Designing algorithm**

Top-down stepwise refinement:

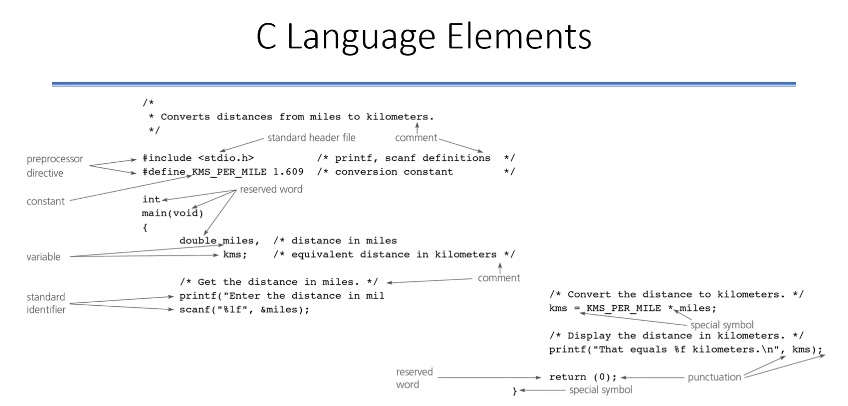
* List major steps (sub-problems)
* Break down each step into a more detailed list

Desk-check your algorithm:

* Perform steps of the algorithm by yourself
* E.g.:

1. Get the distance in miles
2. Convert the distance to kilometers
3. Display the distance in kilometers

* Step 2 may require further refinement:

 2.1. The distance in kilometers is 1.609 times the distance in miles

#include <stdio.h> - - -> Directive for C compiler that we’re using existing library from within C. Compiler knows that you are intending to use certain things, not part of the C itself but part of a library, and so that you can pretend it’s already there for the time being for this part of the coding and then your linker will take this in and put in place.

Within compiler, there are actually 2 steps:

* Preprocessor (Precompilation): Does somethings that has not much to do with the language convertion itself but we can do this convertion within the file like eliminating things like comments. Comments are not needed for compiling, processing.
* Process: Compilation.

So comments just increase the size of .c file, they don’t increase the program size.

#define - - -> Constant definition.

#define [name] [sth] - - - > Now sth is represented by this name.

After the preprocessor is done, what preprocessor part of the compiler is wherever it sees name, it will replace name with the content of the sth till the end and #define line will disappear for compiler.

Standard C only have /\*…\*/ for comments, not //… - - - > So use /\*…\*/

PREPROCESSOR DIRECTIVES

Preprocessor modifies the text of a C program before compilation.

Preprocessor directivers start with a #.

#include <stdio.h> :

* Each library has a header file. Include it to access the library.
* Preprocessor inserts definitions from the header.
* stdio.h includes information about standard input/output.

#define KMS\_PER\_MILE 1.609 :

* Defines a constant macro
  + Value of KMS\_PER\_MILE can not change.
* Preprocessor replaces each occurrence of “KMS\_PER\_MILE” in the text with “1.609”.
* KMS\_PER\_MILE is easier to remember.

FUNCTION MAIN

C programs have exactly one main function which is called by the OS.

* Marks the beginning of the program execution.
* (void) indicates that function receives no data.
* int means that main “returns” an integer value.

Function bodies enclose in braces.

* Function body has 2 parts:
  + Declaration of variables that you’re gonna use.
  + Executable statements that does something on variables.

IDENTIFIERS

Reserved words:

* E.g.: “int” and “void”
* Can not be used for any other purpose

Standard identifiers:

* E.g.: scanf, printf
* Has a special meaning but can be redefined

User-defined identifiers:

* E.g.: name of memory cells (miles and KMS\_PER\_MILE)
* Free to select the name
* Syntax rules:
  + Includes only letters, digits and underscore
  + Can not begin with digit

C is case sensitive. A is different than a.

RESERVED WORDS

Keywords

|  |  |  |  |
| --- | --- | --- | --- |
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |

PROGRAM STYLE

* Pick meaningful identifiers. Long enough to convey the meaning.
* If the identifier consists of 2 words, place an underscore character between words.
* Do not choose similar identifier names.
* Use uppercase letters for names of macros, use lowercase letters otherwise.

ls -la - - - > Shows files and properties. You can see bytes of files.

-rwxrwxr-x - - - > First three (rwx) is you, second three (rwx) is your group, others is (r-x) everybody else in the system. First three what you need to care about. I can read from this file, write to this file and execute this file.

VARIABLES

Variables are abstractions which means you use something, you see something in place of something complex underneath. You don’t care how that complexity happens. Someone handles that. You only see the front end of it. It’s like shifting a gear.

From the programmer’s perspective, abstraction of a variable is something related to memory (memory is compose of transistors) management.

In order for me to understand how 0s and 1s organize within the memory, I need to know a lot of things. You need to have quite a bit of memory to keep track of where you put things and so forth. Instead what programming languages do is, they actually gives you the concept of variable. When you have such a concept, memory becomes a very simple thing. Instead of 0s and 1s, what memory becomes is some logical blocks.

int age; - - - > I will need memory, this memory is gonna be named age so that I can refer to it and the type of the data that I’m gonna keep in there is gonna be integer. Compiler is gonna handle the rest of the things: how many bits you need, place of age in the memory, etc. they all are gonna be taken care of by compiler.

Shortly I say : Please give me place in the memory. This place should be able to keep the integer number and I will refer to that location (memory area) as age. Whenever I say age, you know and I know that age means that particular location that you’re gonna give me. I don’t care how you do it and I don’t know where it is exactly in the memory. From my perspective it’s not important. This is the abstraction.

Abstraction helps you store things in the memory for writing and reading purposes you can put something in memory and retrieve something from memory and you have a name to remember.

Variables: memory cells for storing data

* Values can change

Every variable has:

* Name: identifier
* Type: int, double, char…
* Size
* Value

With abstraction, I don’t know how big a block of memory is requiring but with type of the variable we can know the reserved size.

Data types: abstraction of real types

* Predefined data types (int, float…)
* User-defined data types
* Each constant or variable has a type
* int: whole numbers (-123, 15, 27384) - - - > **32 bits**
  + There is a range because of finite memory cell size
  + C will interpret all numbers without . as integer
* double: real numbers (12.345, 0.5217e-4) - - - > **64 bits**
  + Too large and too small numbers can not be represented
* char: character values (‘a’, ‘5’, ‘^’, ‘,’)

With 32 bits, you can write (2^33) - 1 as the biggest number.

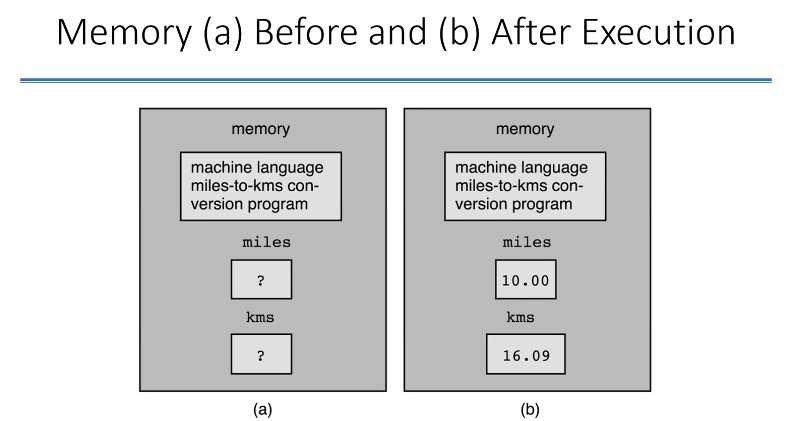
But the leftmost bit must be declare sign so you can write (2^32) - 1 as the biggest number,

and -((2^32) - 1) as the smallest number.

unsigned int - - - > there is no sign bit so biggest number with 32 bits becomes (2^33) - 1

unsigned char (**uses 8 bits**) = char - - - > can hold integer numbers from 0 to 255

signed char (**uses 8 bits**) - - - > can hold integer numbers from -127 to 127



Compiler gives me miles and kms places from the memory. These variables are **uninitialized variables** because we didn’t give values for them.

In the beginning, nothing is stored in miles and kms places according to code in page 4. But maybe these places populated with 1s and 0s but maybe that places filled with nothing (nothing means 0s for every bit or 1s for every bit or maybe some random 0s and 1s) from the beginning of starting the computer.

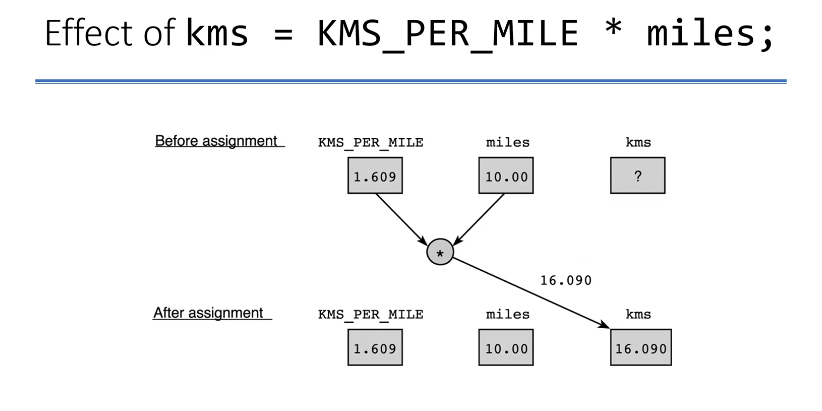
Once we declare, until we actually put something in the memory, we will never know what’s in there.

In assignment, left hand side must be variable. Right hand side should be something that would evaluate to a value which means right hand side is an expression. Expression can be constant value (10, 10.5, etc.), other variable, function calls or arithmetic expression.

\* (multiplication) is a binary operator which means need 2 values to operate. This sign should have same type on both sides. If there is different types (for example int and float) on both sides, languages can behave differently. Language (actually the compiler) could give an error or maybe just warn you but interpret the integer as a float. If they are float and double, then float one will be interpreted as double.

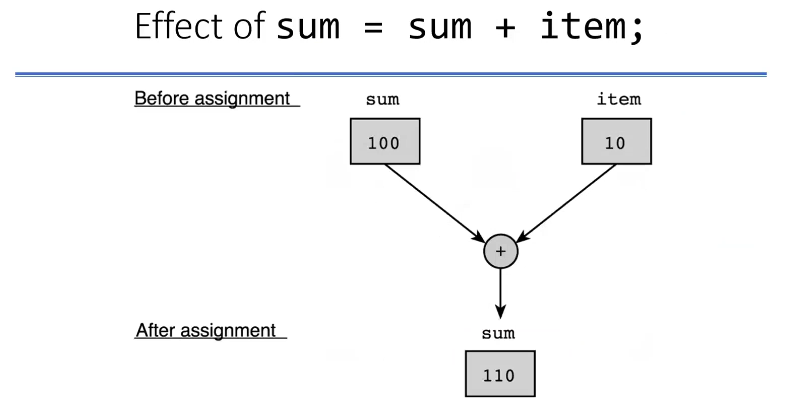
If you want to be sure, if you want to assign the result of the multiplication to a variable, all three should be the same type if you don’t want to get any complains, misunderstandings because of the conversion.

EXECUTABLE STATEMENTS

* Comes after declaration
* Compiler translates to machine language code
* Assignment statements:
  + Used to store value to a variable
  + Ex: kms = KMS\_PER\_MILE \* miles;
  + In general:
    - variable = expression;
  + Assignment operator: “=”
    - Should be read as:
      * becomes
      * gets
      * takes a value of
  + Previous value of variable is lost!

Compiler has already replaced KMS\_PER\_MILE with 1.609 after preprocessor.

miles and kms are in memory but KMS\_PER\_MILE is not in memory, compiler just uses, remembers it.



* Input/Output Operations
  + Input Operation: Reading a value into a variable by scanf
    - A different data can be entered by the user
  + Output Operation: Displaying a value by printf
* Several I/O functions in C
  + All in standard I/O library
    - #include <stdio.h>
* Function call is an executable statement
  + Funtion performs the action for you

**printf**

Displays the output.

printf(“That equals %f kilometers.\n”, kms);

*printf(format string, print list);*

*print list could be any of 3 expressions we discussed in page 9.*

* Function name: printf
* Function arguments: in paranthesis
  + Format string: “That equals %f kilometers.\n”
  + Print list: kms
* Placeholders: %c, %d, %f, %lf…
* Escape sequence:
  + \n means newline: cursor moves the beginning of the next line
  + \t leaves tab space
  + Can be used anywhere in the format string

|  |  |
| --- | --- |
| **Escape Sequence** | **Description** |
| \n | Newline. Position the cursor at the beginning of the next line. |
| \t | Horizontal tab. Move the cursor to the next tab stop. |
| \a | Alert. Sound the system bell. |
| \\ | Backslash. Insert a backslash character in a string. |
| \” | Double quote. Insert a double-quote character in a string. |

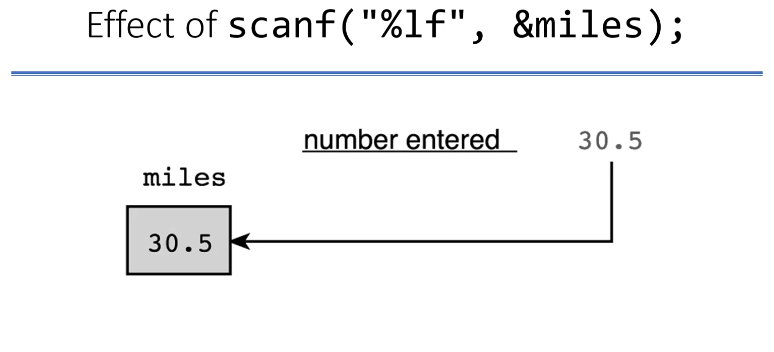
**scanf**

Reads the data into a variable.

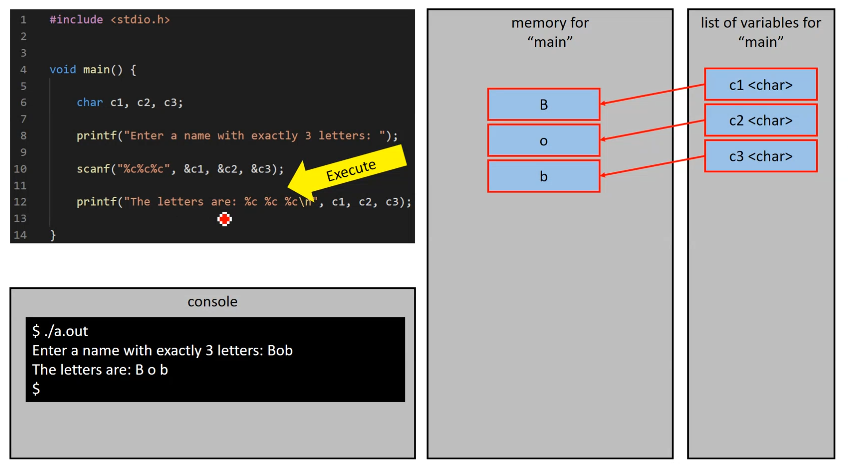
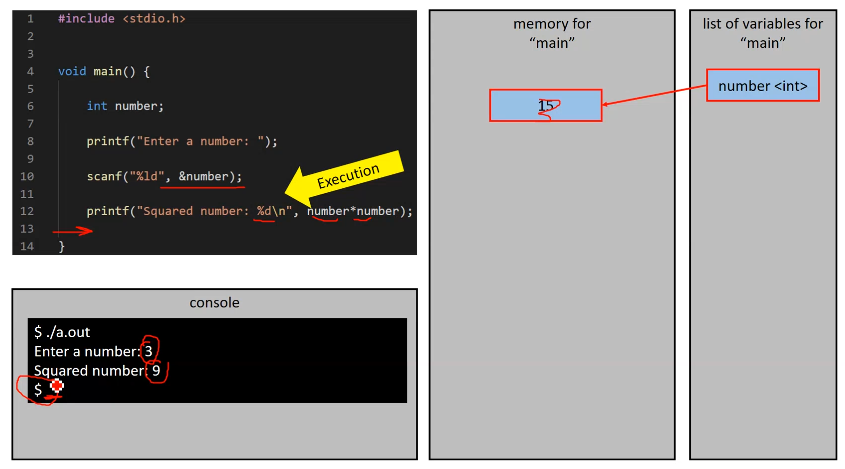
scanf(“%lf”, &miles);

*scanf(format string, input list);*

* Function name: scanf
* Function arguments: in paranthesis
  + Format string: “%lf”
  + Input list: &miles
* Address-of operator: &
  + Used to inform scanf about the location of the variable
  + If not used, scanf knows only the value of the variable



If you use scanf and try to read a decimal number but you print 20b, scanf will just read 20 and stop at b.

Compiler should complain because my number is int but I try to take long integer with scanf.

**The return statement**

* Transfers the control to the OS
* Return value indicates whether the operation is successful or not

**Comments**

* Ignored by the compiler
* Provides information for the programmer

You can only have 1 main function.

**Program Style**

* One statements in each line
* Use extra spaces for readability
  + Compiler ignores them
  + Leave a space before and after operators
  + Indent each block
  + Insert blank lines between sections
* Use comments
  + Write a descriptive comment for
    - the program (header section)
    - each identifier
    - each program section

**Arithmetic Expressions:** ways to generate integer and double data values

* Manipulates type int and double data
* Binary operators: +, -, \*, /, %
  + Two operand: constant, variable or expression
  + Type of the result depend on the types of the operands
    - int if both operands are integer
    - otherwise, it is double
* / operator
  + Integer division: computes integral part of the division
  + Right operand cannot be zero, it is undefined
* % operator
  + Returns integer remainder
  + Right operand cannot be zero, it is undefined
  + You cannot have negative right operand, it is non standard
* Unary operators: +, -
  + One operand
* Assignment:
  + The value of expression is evaluated and result is assigned
  + What if the type of the expression and the type of the variable is different?
    - Assignment of int to double
      * Fractional part is zero
    - Assignment of double to int
      * Fractional part is lost
    - This is automatic type conversion
    - Type casting:
      * (int) 3.7 - - - > returns 3

**Evaluation Rules**

Parenthesis rule:

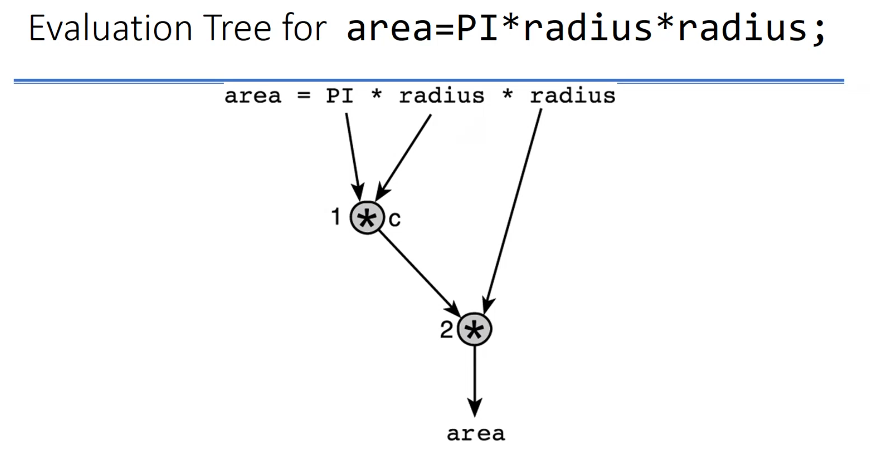
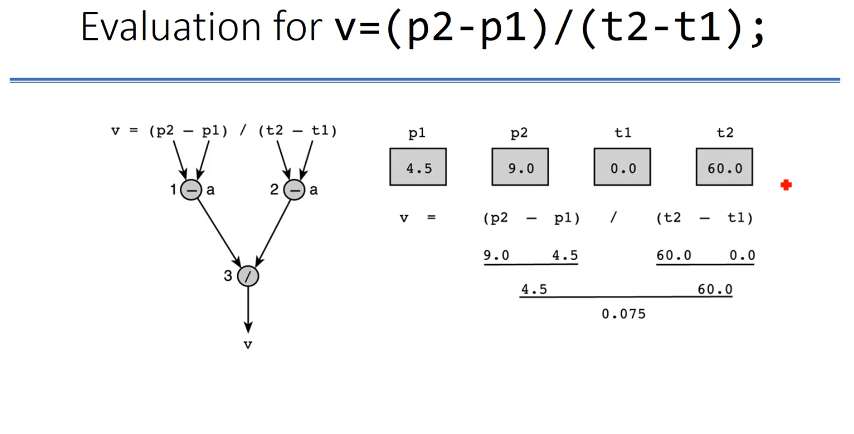
* Firstly, we do paranthesis operations and we begin from the innermost paranthesis
* All expressions in parenthesis are evaluated separately
* Nested parenthesis evaluated inside out

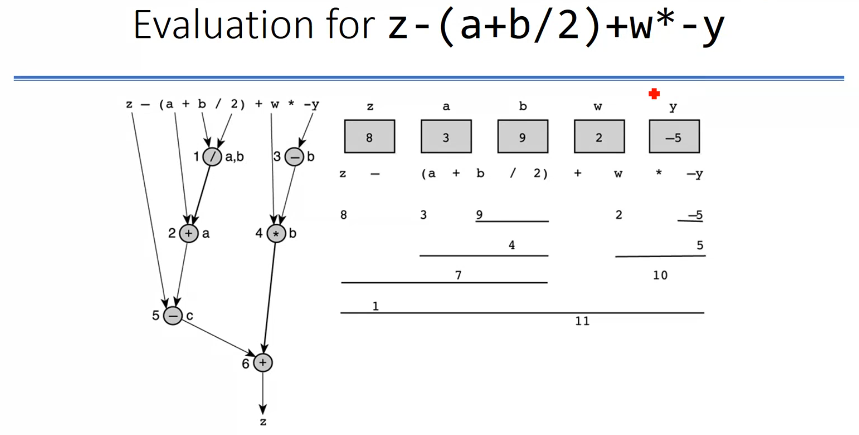
Precedence rule:

* There is an evaluation order in operators
  1. Unary +, -
  2. \*, /, %
  3. Binary +, -

Associativity rule:

* Operators in the same sub- expression and at the same precedence level
  + Unary: right to left
  + Binary: left to right



If there are 2 values one on left and one on right next to + or - operators, then it will be binary operator.

x + - - - 2 : Red is unary so we have to do it right to left which gives us -2. Then we add x. ---> x-2

* Use parentheses as needed to specify the order of evaluation
  + Place numerator and denominator of a division in parentheses

m = (a - b) / (c + d)

* + Use extra parantheses for readability

(a \* b \* c) + (d / e) - (a + b)

* Two operators can be one after the other

a \* - (b + c)

You can use function inside the expression if it returns the right values and right type. Then C evaluates the functions first. C evaluates the function first and then it will store that in a intermediate place.

When you multiply small numbers (10^-5)

Floating point notation says, “I would have the fractional part for number and I am gonna use certain number of bits to represent it.”. If I have limited number of bits to represent the digits after the decimal point, after a while (for example 10^-20) I am not gonna have enough number of bits to represent my number. So some digits may be gone because I don’t have capacity.

So lets say a = b = c = d = 10:

((a \* x + b) \* x + c) \* x + d - - - > is better than - - - > a \* x \* x \* x + b \* x \* x + c \* x + d

So it is better to multiply small numbers with big numbers first.

And it is better to do the addition first.

And it is better to delay multiplications.

**Output Formatting**

* Default formatting
* User-defined format
  + int: %4d (%nd)
    - Field width
    - Right justified
    - - sign included in the count
    - C expands the field width if necessary (For example C expands if you put 5 digit number but you use %4d and it expands to the right)
    - %04d : this put zeros to the blanks left to the number
  + double: %6.2f (%n.mf)
    - Field width
    - Total of 6 digits (one is .), 2 of them after the point
    - Decimal places
    - Decimal point, minus sign included in the field width
      * Values between -99.99 to 999.99 for %6.2f
    - At least one digit before decimal point
    - Values are rounded if there are more decimal places
      * -9.536 becomes -9.54
  + Use %d or %.3f not to have leading blanks

**Input and Output Redirection**

* Interactive Mode
* Batch Mode
  + Input Redirection: standard input is associated with a file instead of keyboard

myprog < inputfile

* + - No need to display prompting message
    - Display the message about input (echo print)
  + Output Redirection: standard output is associated with a file instead of screen

myprog > outputfile

* + - Can print the file to get the hardcopy

./a.out < input.txt > output.txt

Printing things with printf goes to output.txt, taking things from scanf will be taken from input.txt

You can compare your expected result with output.txt (actual result) with diff in Linux.

If you want to take 2 numbers:

You should print inputs in the text file line by line. Your inputs will be taken in order.

If you print your inputs in the text file in the same line with space between them, scanf still takes them.

If you print your numbers like “150v20” in input.txt, scanf will read the first number (150) but scanf will try to read v but it is not a number so scanf will read character as 0.

You can discard the character between your 2 numbers in input.txt with putting a scanf between scanfs that read your numbers that reads a character.

So spaces between 2 numbers in input.txt are okey but other things are not okey.

If scanfs get what they need, rest of the input.txt is ignored.

You divide your program into pieces and you can make function of each piece to make things separate.

Two reasons that we use functions:

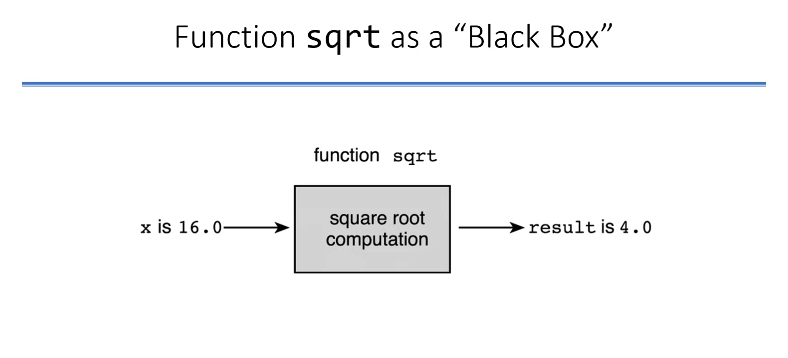
1. Prevent doing things again and again.
2. Avoiding errors.

Software engineering:

* Goal: writing error-free codes
* Use well tested existing codes: code reuse (needful codes)
* Use predefined functions
  + EX: sqrt function in math library
  + Use it as a black box

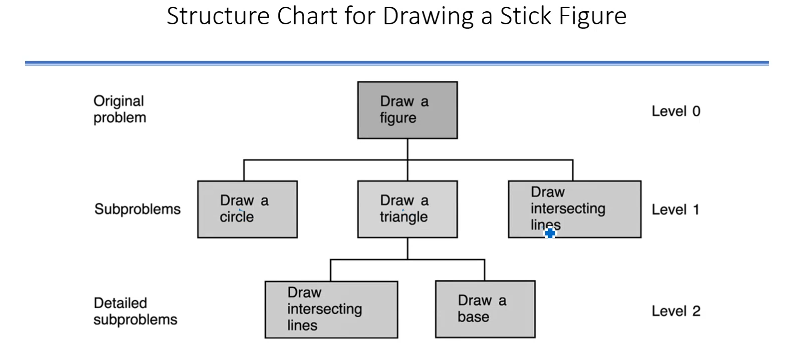
y = sqrt(x);

* + EX: printf and scanf in stdio library



Black Box = Abstraction

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FUNCTION** | **STANDARD HEADER FILE** | **PURPOSE: EXAMPLE** | **ARGUMENT(S)** | **RESULT** |
| abs(x) | <math.h> | Returns the absolute value of its integer argument: if x is -5, abs(x) is 5 | int | int |
| ceil(x) | <math.h> | Returns the smallest integral value that is not less than x: if x is 45.23, ceil(x) is 46.0 | double | double |
| cos(x) | <math.h> | Returns the cosine of angle x: if x is 0.0, cos(x) is 1.0 | double (radians) | double |
| exp(x) | <math.h> | Returns e^x where e = 2.71828…: if x is 1.0, exp(x) is 2.71828 | double | double |
| fabs(x) | <math.h> | Returns the absolute value of its type double argument: if x is -8.432, fabs(x) is 8.432 | double | double |
| floor(x) | <math.h> | Returns the largest integral value that is not greater than x: if x is 45.23, floor(x) is 45.0 | double | double |
| log(x) | <math.h> | Returns the natural logarithm of x for x > 0.0: if x is 2.71828, log(x) is 1.0 | double | double |
| log10(x) | <math.h> | Returns the base-10 logarithm of x for x > 10.0: if x is 100.0, log10(x) is 2.0 | double | double |
| pow(x,y) | <math.h> | Returns x^y. If x is negative, y must be integral; if x is 0.16 and y is 0.5, pow(x,y) is 0.4 | double, double | double |
| sin(x) | <math.h> | Returns the sine of angle x: if x is 1.5708, sin(x) is 1.0 | double (radians) | double |
| sqrt(x) | <math.h> | Returns the non-negative square root of x for x >= 0.0: if x is 2.25, sqrt(x) is 1.5 | double | double |
| tan(x) | <math.h> | Returns the tangent of angle x: if x is 0.0, tan(x) is 0.0 | double (radians) | double |



**USER DEFINED FUNCTIONS**

* Function prototype
  + Functions should be defined before they are used
    - Insert the whole function definition
    - Insert the function prototype
  + Defines
    - Data types of the function
    - Function name
    - Arguments and their types

function\_type function\_name (argument types);

EX: void draw\_circle(void);

* Function definition
  + Defines the operation of a function
  + Similar to main function

function\_type function\_name (argument list)

{

local declerations

executable statements

}

* Function heading: similar to function prototype
* Function body: enclosed in braces

Every function has to have a return type.

void (in function\_type) : You don’t have any output. So you cannot use this function type in any expression or sth like that.

void (in argument list) : You use void when you don’t care about the type of that argument (for pointers mostly). If you don’t have any arguments, you will not use anything.

**Flow of Control** : In what order my executable statements are executed.

Control means what is the CPU actually doing and how that control happens, what is the next one, next one and next one…

* Compiling the program:
  + Function prototypes: compiler knows the functions
    - enables compiler to translate function calls
  + Function definition: translates the code of the function
    - Allocates memory needed
  + Function call: Transfers of the control to the function
  + End of the function: Transfer of the control back to the calling statement
    - Releases the local memory

Control always starts from main function and flows from there on.

Assignments goes like this: compiler looks to expression (right hand side), evaluate it and assigned it to our variable (left hand side).

int f(int x) {

int y; int f(int x){

y = x \* 3; = return (x \* 3);

return y; }

}

When you do the prototype declarations, you don’t have to know the name of the argument itself but only the type of the argument you need. So “int f(int);” is enough for declaration.

Only executable statements change CPU’s flow of control.

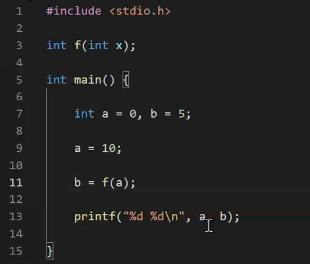
For example you have “int a” in your main function whose value is 10 and you are gonna send it to f function (left one above):

* In main’s memory, a will stay with its value 10.
* In f’s memory, int y and int x variables will be created and x’s place will be populated with 10 because you send the a’s value which is 10.

f(a); - - - > caller

function implementation - - - > callee (which is called)

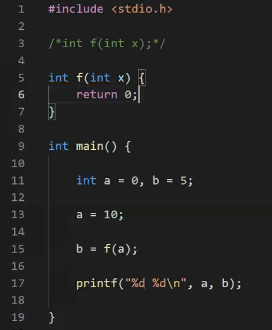
When the function is returned, it gives away all the memory that it used. Everything is gone.

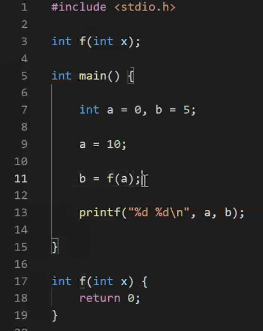
This will be compiled but can’t be linked.

There is a linker problem because we don’t have the code that control flow go to.

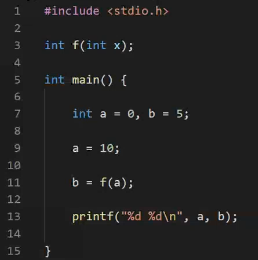
There are 3 places we can put this code.

1. Place

We declared the function and write code in it in 5-6-7 lines.

2. Place

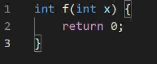
This is better than 1 for readability.

3. Place

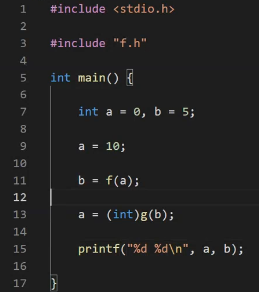
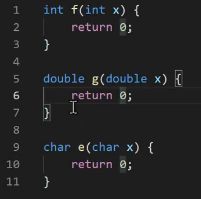
Upper one : main.c

Below one : f.c

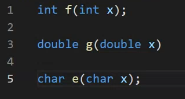
You have to compile this as - - - > “gcc main.c f.c”



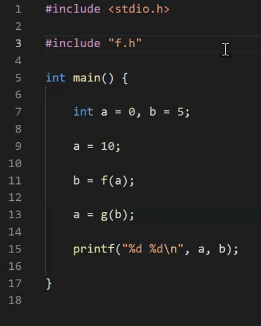
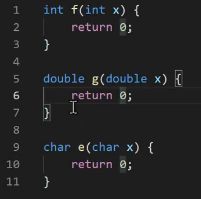
Another way to do this with more than 1 function

Top to bottom : main.c - f.c - f.h

#include “f.h” brings whatever in the f.h file to the same place where #include “f.h” at. It copies all the codes to wherever it is.



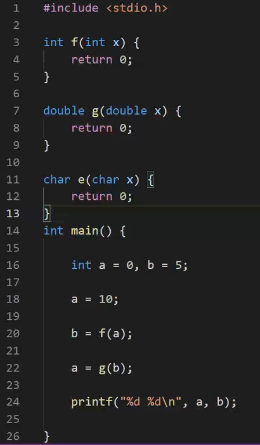
What happens if we delete everything in f.h and put everything at f.c to f.h?

It will work but it is not good.

Left : f.h

Right : main.c

Below : main.c (what happens with #include “f.h”)



So what actually happens is I compile function f, function g, function e, function main and then I will be linking the necessary stuff that these might need from stdio library. Then I will have my output.

If functions f, g, and m are really complex; everytime you need to change main, you have to recompile everything.

But if you do it like the upper page, you can compile your f.c file once, create your f.o file and don’t touch them if you don’t change f.c code. So then you will care only the f.o file which is the compiled one.

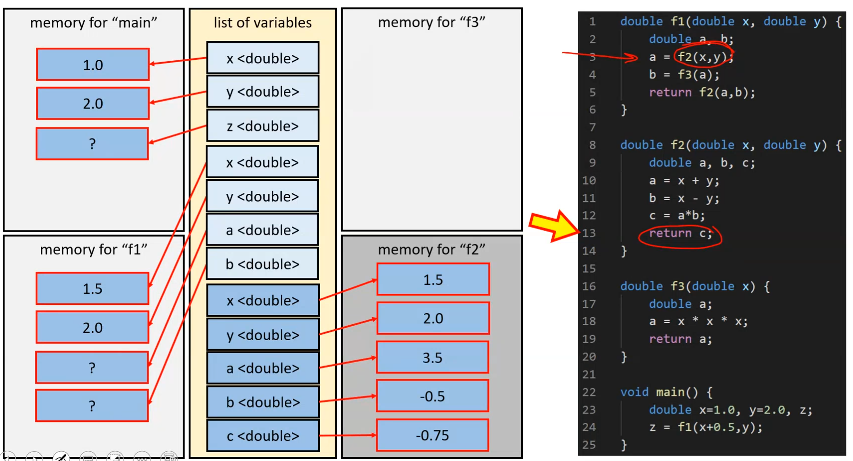
Compilation is expensive process so we want to avoid it as much as we can.

Other reason, maybe writer of the library doesn’t want to show you how does he/she implement the f,g,e functions. He/she wants you to use it but doesn’t want you to see it. So he/she can give you just f.h file that showing the declerations and then he/she can give you the f.o file having only the machine code that is created from the f.c. You can still do the linking without any problems but source code will stay secret. So you can just deliver the compiled code and the .h file.

2 reasons why we should avoid using upper page : Compilation time (you only do the compilation as needed) and protection of code.

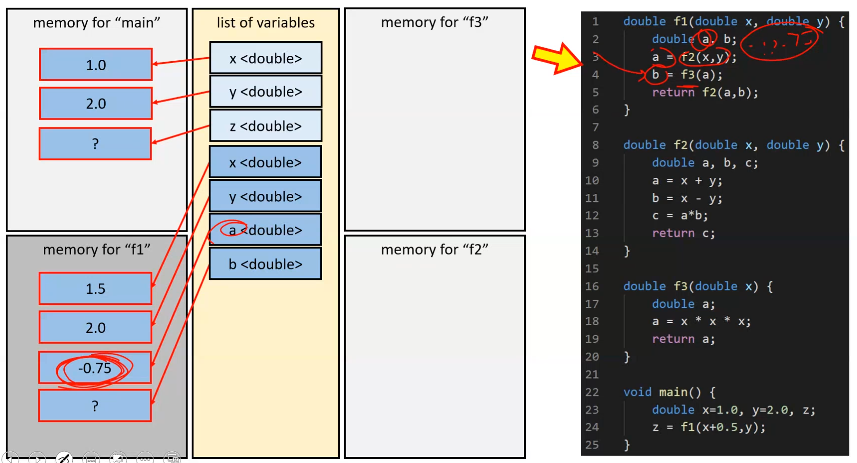
You can do ‘ #include “f.c” ‘. But .c means c source which should be compiled separately and link later to wherever you are using it. “.h” means declarations on the header file. Header files only have declarations. Most of the time .h files have a lot of preprocessor definitions and declarations like variable declarations if you have global variables or mostly name declarations and function prototypes, no source code. Then .c file should use those, include them and build code for your functions and other things so that you can compile them separately and link them later on. So your include should be header file.

Preprocessor directives like include are not part of the compilation, they are before the compilation. Include will take entire thing and copy this at that line after the preprocessor is done so #include … will disappear for the compiler.



Actually these rectangles are for illustrations. All functions use the same memory, there is only 1 part of the memory allocated for this program. They will use and reuse this part of the memory as you get in and get out of the function.

Compiler knows every member of list of variables belongs to which function.

After running f2, all values for f2 will be gone.

…

Also when you return at the very end from the main function, we get nothing left in the computer other than whatever you have returned if main function returns something or whatever you have written to a file etc.. Else we have no effect to the machine.

**CONTROL STRUCTURES**

* Controls the flow of program execution
  + Sequence
  + Selection
  + Repetition
* We used sequence flow
  + Control flows from one statement to next one
  + A compound statement in braces
    - Ex : function body
* We will learn selection control statements
  + if
  + switch
* They select one statement block and executes them

C accepts any nonzero value as a true.

Binary logical operator should result in a binary value (0 or 1).

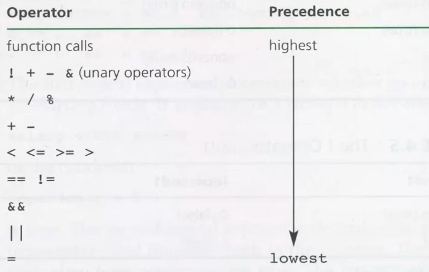
relational-operators : “>” , “<” , “>=” , “<=”

equality-operators : “==” , “!=”

logical-operators : “&&” (and - binary) , “||” (or - binary) , “!” (not - unary(takes whats on the right hand side and logically negate it))

These are binary operators (there is sth on left and sth on right). They return logical value : 0 for false, 1 or more for true.

EXPRESSION (operator) EXPRESSION - - - > expressions must be same type



First precedence is actually parantheses.

Same precedence level goes from left to right.

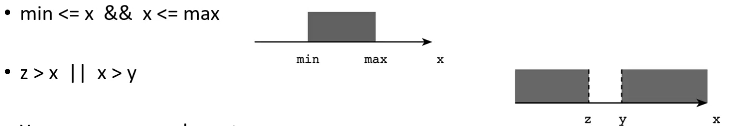
! 10 + x >= 10 \* y; - - - > In here we do “! 10” first and get 0 (logical number).

Short-Circuit Evaluation

C stops evaluation;

* If left operand of && is false
* If left operand of || is true

(1 >= 5) && (x || y || f(12)) - - - > Parantheses will be done first so left one is logically 0. Then right hand side won’t be evaluated so function won’t be called.



You can compare characters :

* ‘z’ > ‘a’ , ‘A’ > ‘a’
* ch == ‘a’ - - - > You can do these type of things for if block.

‘b’ - ‘a’ = 1

! (‘a’ <= ch && ch <= ‘z’) = ‘a’ > ch || ch > ‘z’

Logical values treated as integer in C. They can be stored anywhere where integer is excepted meaning an integer variable can store them.

* int res;

res = x>=0; - - - > res is either 1 or 0

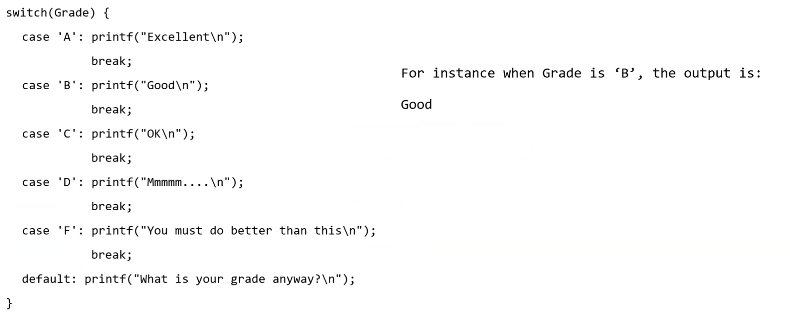
In flowcharts, conditions used in a diamond shape.

Each function should perform single operation and be more reusable.

**The switch statement:**

Select one of the several alternatives.

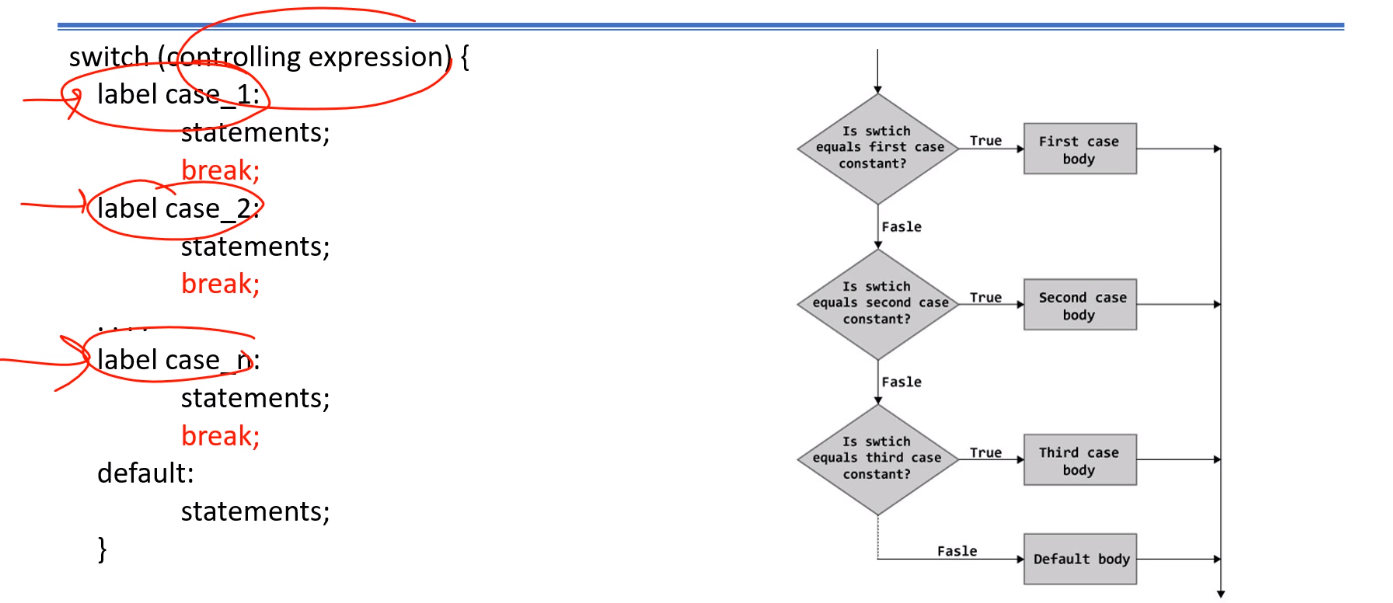
Selection is based on the value of a single variable (of type **int** or **char** not double).



If Grade is ‘B’ and you forgot to use “break;” in case ‘B’. Then next line will be executed so “OK” will be printed to the screen and break.

Grade may be expression.

default is optional. You may not need it.



Controlling expression evaluates a value.

You can’t control more than 1 thing in 1 case but you can do like this:

switch (Grade){

case ‘A’:

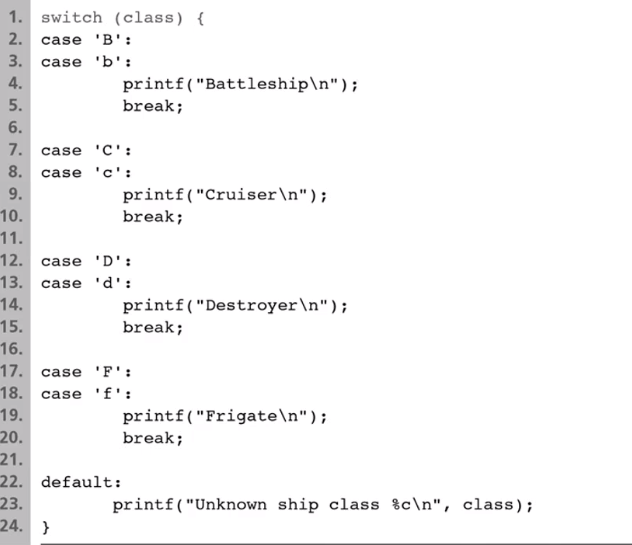
case ‘B’:

case ‘C’: statements;

break;

}

Now you can get same response for different cases.



When you enter from “:” , your codes will execute till you see break statement.

You cannot say || , && etc. in cases.

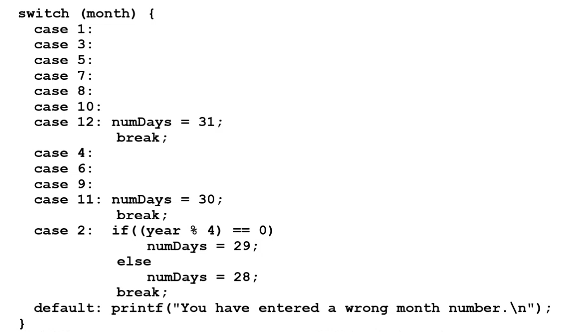
What you write for case (‘B’ or ‘c’ or etc.) must be same type with you write for switch.

case only takes constants. We can’t put expressions.

If you write:

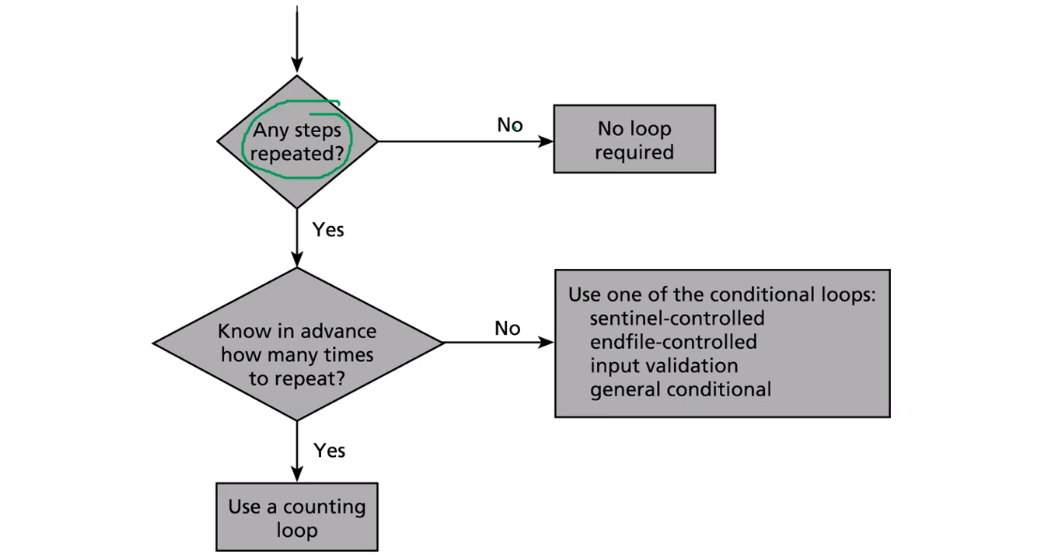
case ‘B’ || ‘C’ :

then switch structure take what is between “case” and “:” so 1 is taken. It is a logical value but compiler thinks it as an integer even though it is a logical value. Character also an integer. Compiler maybe think that these both are same type. So it can try to match them but eventually, let’s say your switch variable is ‘b’, compiler will say no.



**REPETITION**

LOOP CHOICE



|  |  |  |
| --- | --- | --- |
| **KIND** | **WHEN USED** | **C IMPLEMENTATION STRUCTURES** |
| Counting loop | We can determine before loop execution exactly how many loop repetitions will be needed to solve the problem | while  for |
| Sentinel-controlled loop | Input a list of data of any length ended by a special value | while  for |
| Endfile-controlled loop | Input of a single list of data of any length from a data file | while  for |
| Input validation loop | Repeated interactive input of a data value until a value within the valid range is entered | do-while |
| General condition loop | Repeated processing of data until a desired condition is met | while  for |

**COUNTER CONTROLLED LOOP**

We know number of loops.

* Repetition is managed by a loop control variable
  + For example a counter

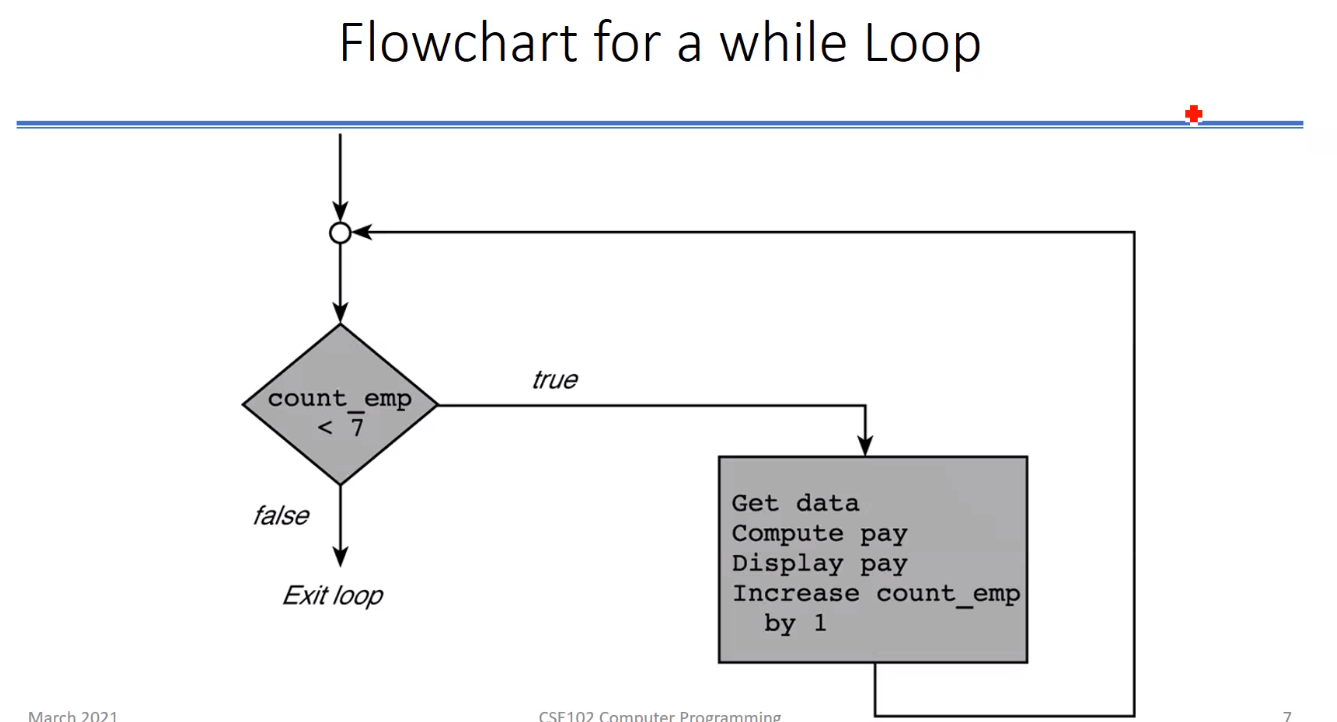
General format:

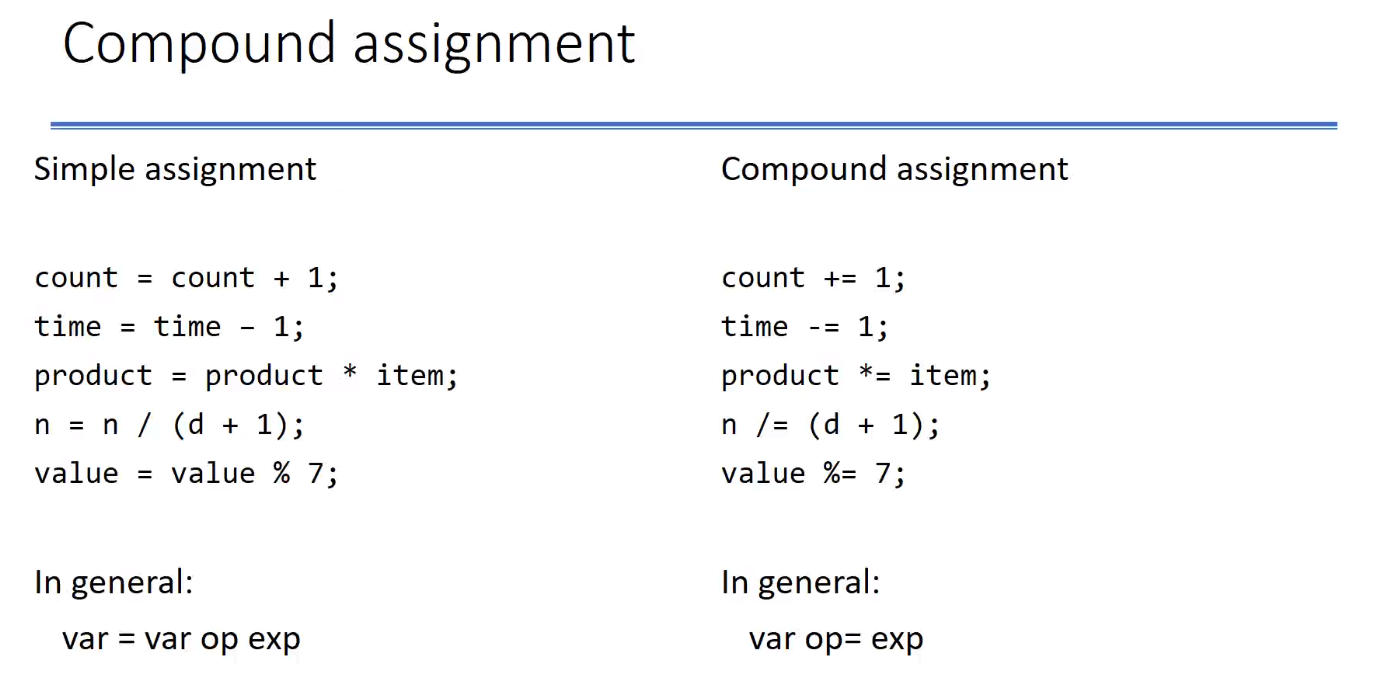
set counter to 0 ***/\* initialization \*/***

while counter < final value ***/\* test \*/***

do one or more things ***/\* loop body \*/***

increase counter by one ***/\* updating \*/***





General syntax of **while** statement:

while (loop repetition control)

statement

General syntax of **for** statement:

for (initialization expression; loop repetition condition; update expression)

statement

for statement supplies a designated space for each of the loop components:

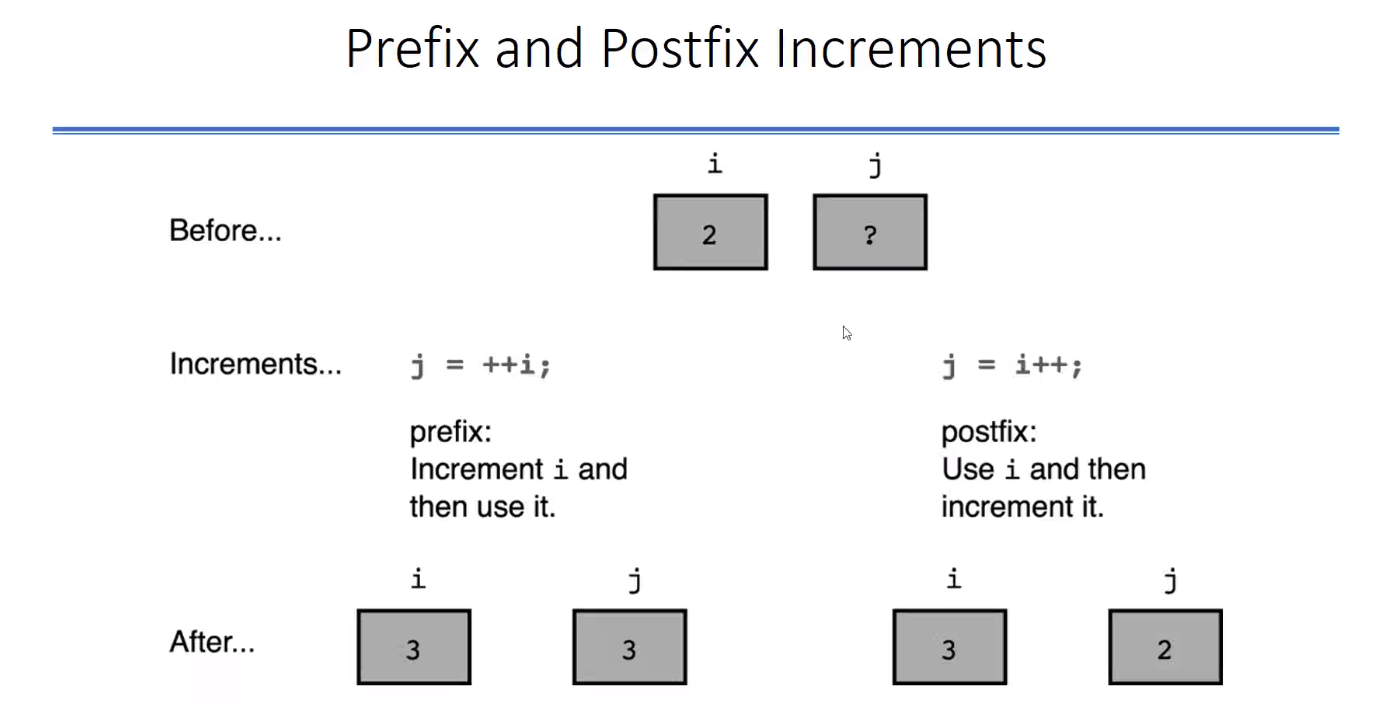
* Initialization of the loop control variable (In while, it is above while)
* Test of the loop repetition control (In while, it is next to while)
* Change of the loop control variable (In while, it is below while)

Increment and Decrement Operators

Unary operators : ++ - - -> Increments the operand / -- - - - > Decrements the operand

The value of the operation depends on the position of the operator

* Pre-increment : operand is after the operator
  + Value is the variable’s value after incrementing
* Post-increment : operand is before the operator
  + Value is the variable’s value before incrementing
* Similar for decrement operator



++i - - - > returns i+1 and increments i

i++ - - - > returns i and increments i

They only differ by the return value.

If you use ++i or i++ in for statement in update expression part. There is no difference because we are not using the return value.

i = 10; i = 10;

x = 2 \* (i++); x = 2 \* (++i);

x - - - > 20 x - - - > 22

i - - - > 11 i - - - > 11

n = 4;

printf(“%3d”, --n); - - - > prints 3

printf(“%3d”, n); - - - > 3

printf(“%3d”, n--); - - - > prints 3

printf(“%3d”, n); - - - > 2

y = n \* 4 + ++n; - - - > ++n will be done first bc it is unary and result (y) is 15 and n is 3

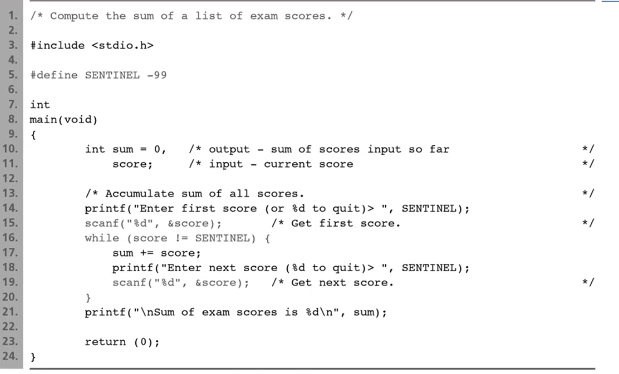
**SENTINEL CONTROLLED LOOPS**

Loop ended based on an input or something that we read from somewhere and when input reaches a certain value, we stop it.

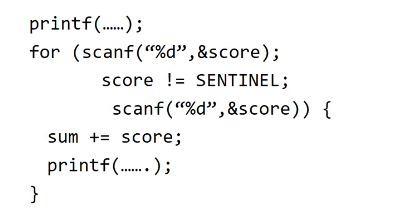
You should be careful about extra repetitions.

Usually number of items is not known in advance.

Sentinel value: unique value to stop repetition. It should be an abnormal value. “Print -1 to exit.” kind of thing.



for (; score != SENTINEL; ) = while (score != SENTINEL)



**END-FILE CONTROLLED LOOPS**

Calculate sum of a list of integers in a file

* A data file is terminated by an endfile character
  + detected by fscanf functions
* special sentinel value is not required
  + uses the status value returned by scanf

Algorithm:

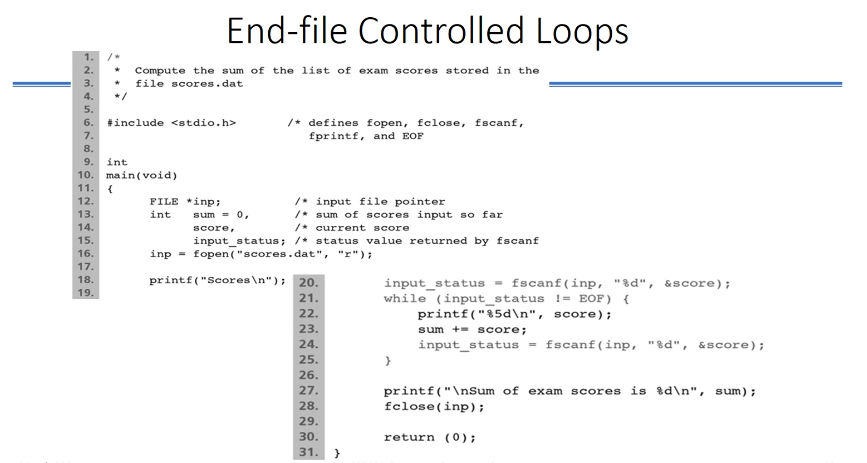
Initialize sum to zero

Read the first value

while end of file is not reached

Add value to sum

Read the next value



**Infinite Loop on Faulty Data**

If the file contains a faulty data 7o, fscanf:

* stops at the letter ‘o’,
* stores the value 7 in score
* leaves the letter ‘o’ unprocessed
* returns a status value of one

On the next loop iteration, fscanf:

* finds the letter ‘o’ awaiting processing
* leaves the variable score unchanged
* leaves the letter ‘o’ unprocessed
* returns a status value of zero

In the previous program:

* the return value of fscanf is not checked for values other than EOF
* unsuccessful attempt to process the letter ‘o’ repeats over and over. - - - > INFINITE LOOP!..

Solution: Change the loop repetition condition to: while (input\_status == 1)

loop exits on

* end of file (input\_status negative) OR
* faulty data (input\_status zero)

Add an if statement after the loop to decide whether to print the results or to warn of bad input.

if (input\_status == EOF)

printf(“Sum of exam scores is %d\n”, sum);

else {

fscanf(inp,”%c”, &bad\_char);

printf(“\*\*\* Error in input: %c \*\*\*\”, bad\_char);

}

sum = 0;

for (studentno = 0 ; studentno < M ; studentno++){

…

Instead of this you can do like:

for (studentno = 0, sum = 0 ; studentno < M ; studentno++){

…

**do-while statement**

for and while statements evaluate loop repetition condition before the first execution of the loop body.

Syntax:

do{

statements

}while (loop repetition condition);

\*\*\*scanf function returns a status information. You can think it is returning the number of entries that has been read. If scanf is successful for example reading an integer number into n (variable), the output is gonna be 1 (1 number successfully read). Otherwise it would be 0.

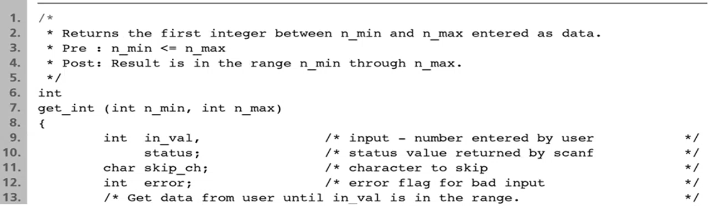
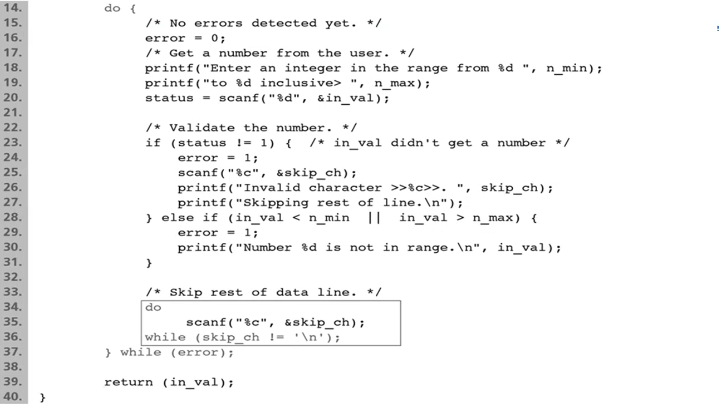
**FLAG CONTROLLED LOOPS**

If loop repetition condition is complex

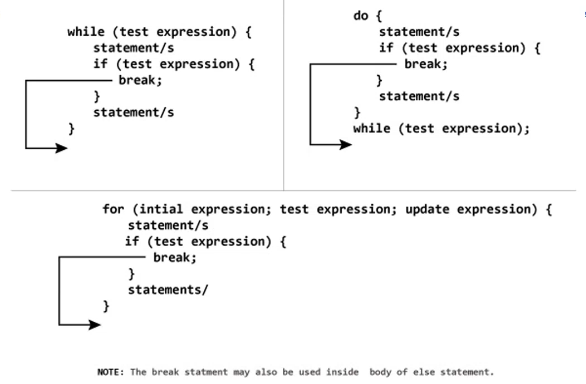
* Use a flag is a type int (values: 1 (true) and 0 (false))
* Flag represents whether a certain event has occurred.

Ex: Input validation

* The do-while is often used in checking for valid input
  + An input is always needed
* Two nested loops
  + Repeat reading input when the input is not valid
    - not in range OR not a number
  + Repeat reading input to skip invalid input line
    - Not to have infinite loop



***Abnormal termination of a loop (Altering normal operation of a loop)***



Using break is not a good thing. If you can terminate the loop with the test condition properly, you should do so.