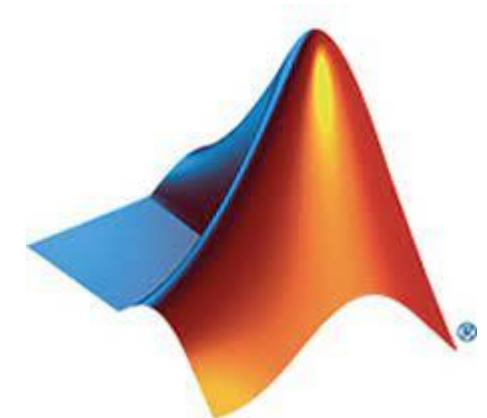
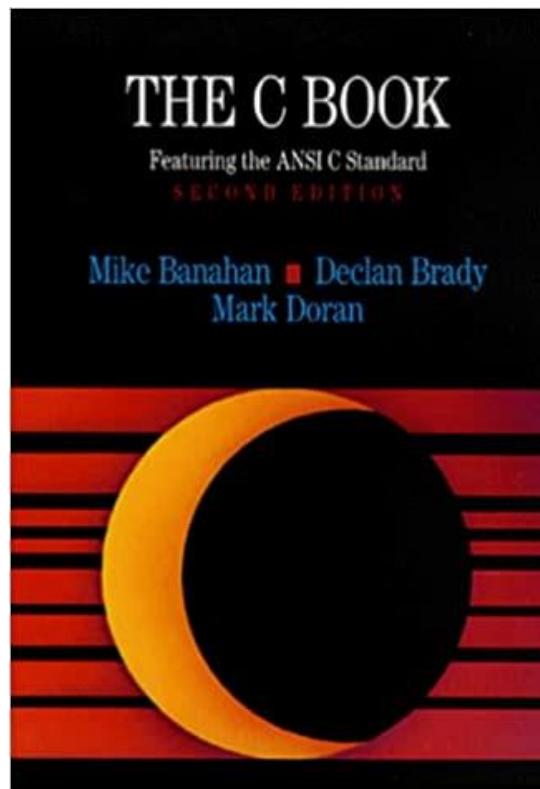
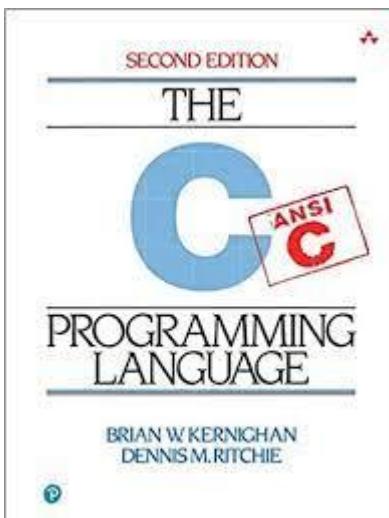


Using C with MATLAB #01



https://publications.gbdirect.co.uk/c_book/copyright.html

Requirements

- Prerequisites: Basics of MATLAB programming, no C knowledge required
- Learning activities and teaching methods: teaching, independent work & **homework** after each lecture and **final project** work including sufficiently long reports.
 - You need to pass all homework assignments separately. Also, the final project must be passed. So, all homework need to be done and passed, and final project must be passed.
 - Rough idea is that you spend around 5-7 hours per week on self study (e.g., reading the course book and other materials) and for homework.
 - Final grading is pass/fail
- After completing the course, the student
 - Knows fundamentals of C programming language
 - Knows how to call C function from MATLAB as a part of larger programs
 - Knows complex data types and multithreaded programming (OpenMP)
 - Knows most important functions of C standard library
 - **Knows about use of AI in coding [new]**

Why we want to use C with MATLAB?

- MATLAB is designed to be a high-level language with a focus on ease of use and readability, and therefore may not always be optimized for performance-critical tasks. MATLAB is typically used for prototyping and simulation and may not be suitable for real-time applications.
- C is very useful skill for industry and programming in general
 - Also, microcontrollers etc. do not run MATLAB. Instead, C code can be compiled to run natively on embedded systems, such as microcontrollers.
 - Much more compact than alternatives such as C++ (those can be learned later if really needed)
- Using C with MATLAB makes input and output and visualization easy (not a strength of C...)
- Speed difference in code with lots of for loops, if else statements, **integer operations**, ...
 - Well written C code can be faster (but MATLAB's JIT is often surprisingly fast)
- We can even make C code faster by using SIMD instructions via intrinsics or GCC vector types (not covered in this course) and parallel processing (covered in this course)
- Speed difference in codes with lots of matrix operations / FFTs / sorting is not a good reason to use C since MATLAB uses heavily optimized matrix routines (faster than we can write unless SIMD and cache optimization is used) and optimized FFT routine
- The input to C code in MATLAB should be as large as possible to avoid overheads of function calls (avoid calling C code inside for loop)
- Overall, using C with MATLAB provides a powerful combination of high-level functionality and performance, allowing you to create custom solutions that are tailored to your specific needs

C was developed at Bell Labs by Dennis Ritchie 1972-1973



Course Book

- Recommend student book for this course is:
 - K. N. King, C Programming: A Modern Approach Second Edition (**might be best book ever written about C**, especially for beginners!) !
- Lectures are partly based on “The C Book” by Mike Banahan, Declan Brady and Mark Doran
- Available freely at:
<https://github.com/wardvanwanrooij/thecbook/releases/latest>

This first lecture covers some aspects of Chapter 1

C language difficulty

- ANSI C (C89/C90) has these keywords [note that we use C99 in this course, but these are the major 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

- You only need to remember 32 keywords! (plus, some library functions, pointers, etc.) [also for example “auto” and “register” are rarely used, “volatile” is more for embedded systems]
- Today we will mostly use “double”, “int”, “return”, “sizeof”, “void”, “while”, “extern”, “if” (that is already 8 out of 32!)

Installing C/C++ compiler

- You need MATLAB version 2018 or later
- Install C/C++ compiler from this link
- <https://se.mathworks.com/matlabcentral/fileexchange/52848-matlab-support-for-mingw-w64-c-c-compiler>
- Installing it does not require admin rights!
- If for some reason you cannot install compiler, you can use MATLAB Online at <https://matlab.mathworks.com/>
 - It has gcc installed (**Currently under bug fixing, GCC compiling does NOT work**)
- If you have Linux, MATLAB should automatically detect GCC
- If you have Mac, you can check links in Moodle

Background in MATLAB

- Student is assumed to know basics of MATLAB
- If not go through the free course at
<https://matlabacademy.mathworks.com/> [MATLAB Onramp]
- How to install MATLAB: Go
to <https://se.mathworks.com/academia/tah-portal/university-of-oulu-873976.html>

Background before first C program in MATLAB

- In the next slide the outputs are from C++ interpreter (by CERN)
- Normally, C is a compiled language (you must compile the code before running it). For interactive experiments (a bit like in MATLAB), ROOT is recommended
- Notice that is C++ interpreter (not C) but for most parts of this course, the difference is not important.

```
root [0] double result;
root [1] 10
(int) 10
root [2] 10.0
(double) 10.000000
root [3] result = 10
(double) 10.000000
root [4] (double) 10
(double) 10.000000
root [5] 9/5
(int) 1
root [6] 9%5
(int) 4
root [7] 9.0/5.0
(double) 1.8000000
```

Background before first C program in MATLAB

- In C we need to declare variables before using
- We also need to specify the data type of the variable
- “double” in C corresponds to the standard data type in MATLAB
 - If you do not have good reason otherwise, use “double”
- Let us define variable with name “result” with type “double” and assign 10.0 to it

```
root [0] double result;  
root [1] result = 10.0;
```

- The 10.0 above is double but just 10 would be of type “int” (integer). However, if we write result = 10; => C will automatically cast the integer to 10.0 (in this case)

```
root [6] 10.0  
(double) 10.000000  
root [7] 10  
(int) 10
```

Background before first C program in MATLAB

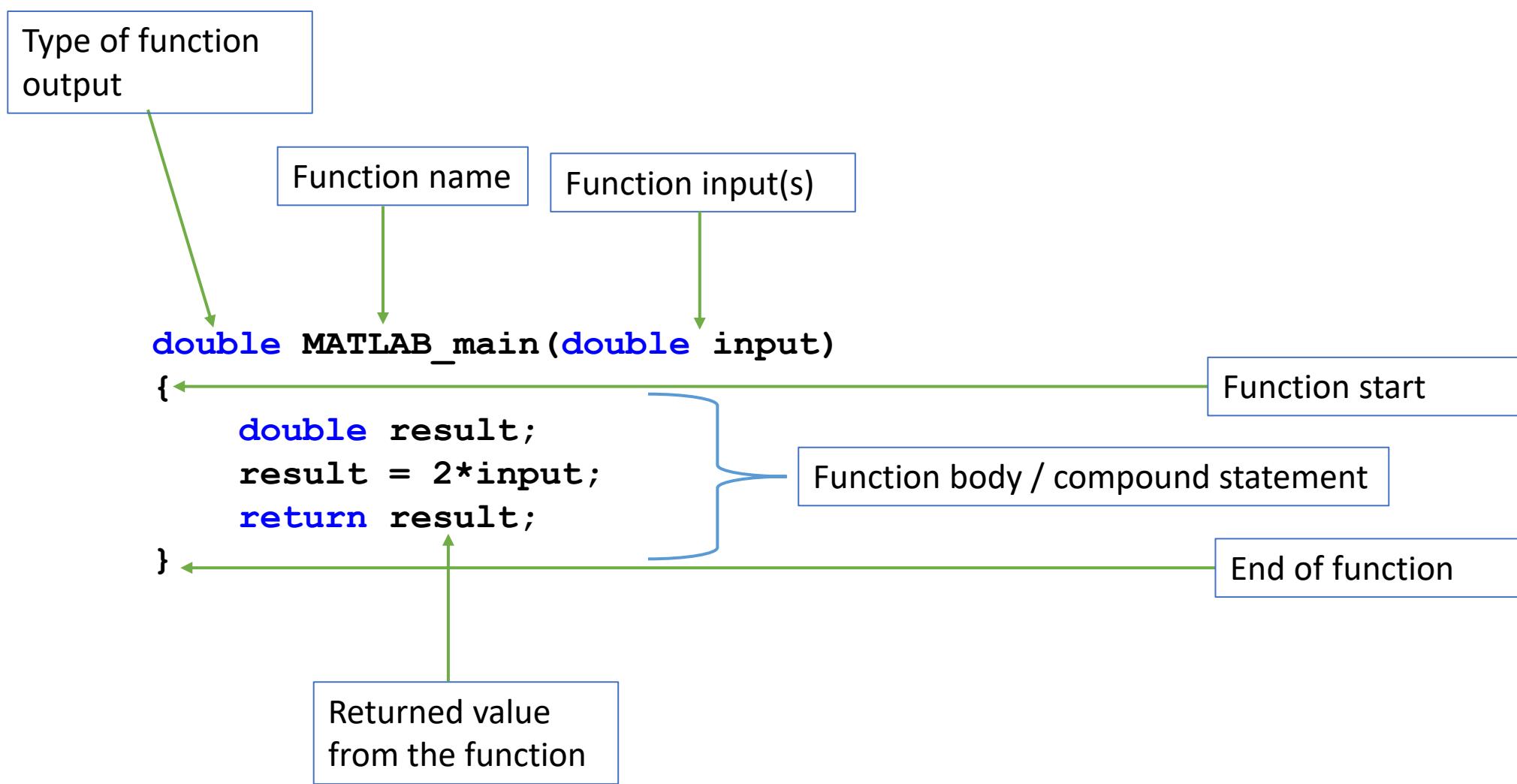
- The provided header file will call MATLAB_main function with one input: variable with name “input”
- Double before function name means that function will return a double (by using the “return” statement with a value)
- A function cannot normally modify the input variables
 - Variable defined inside a function are local to that function
- We communicate with outside “world” by returning a value
 - Another option is to use pointers that will be discussed later

Background before first C program in MATLAB

operator			
+	addition	$a+b$	Sum of a and b
-	subtraction	$a-b$	a minus b
*	product	$a*b$	product (multiplication) of a and b
/	division	a/b	a divided by b
%	remainder	$a\%b$	Remainder of a divided by b

For integers only!

Background before first C program in MATLAB



Background before first C program in MATLAB

- `#include` directive copies the contents of the pointed header file to the current file
- `#include "testi.h"` looks first for testi.h in the current directory
- `#include <stdio.h>` looks outside current directory
- Every C program must have `main()` function [and only one `main()` function] (in MATLAB C/MEX we have chosen to use to `MATLAB_main()`)
- Main function will always run first, no matter where it is in the code
- `\n` is a newline character which changes the line in the output terminal

Every C program must exactly one `main()` function

Our header file requires that every MATLAB C program to have exactly one `MATLAB_main()` function instead of `main()` [MATLAB does not work with `main`]

Our first C program in MATLAB

- You are supplied with the header file
"MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"
- It makes calling C from MATLAB easier by already doing the error checking in inputs etc.
- Make file **example_one_input_one_output.c** and include in it:

```
#define NINPUT_ARGUMENTS 1 // Must be first in the file (value 1 to 7) !
#include "MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"
double MATLAB_main(double input)
{
    double result;
    // Calculate result
    result = 2*input;
    // Return the result
    return result;
}
```

```
function output = example_one_input_one_output_MATLAB(input)
    result = 2*input;
    output = result;
```

Equivalent MATLAB code

In normal C, we need to use main() as main function instead of MATLAB_main and also input arguments are different

Our first C program in MATLAB

- // Compile using (on MATLAB Command Window):
• // mex -R2018a example_one_input_one_output.c
- And output is (from MATLAB command prompt)
- >> out = example_one_input_one_output(10)
- out = 20

Correct!!!

```
#define NINPUT_ARGUMENTS 1
#include "MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"
double MATLAB_main(double input)
{
    double result;
    // Calculate result
    result = 2*input;
    // Return the result
    return result;
}
```

```
function output = example_one_input_one_output_MATLAB(input)
result = 2*input;
output = result;
```

We call the function by the
filename (.C file) that contained
MATLAB_main

Equivalent MATLAB code

Our first “real” C programs

```
1 #include <stdio.h>
2 // above is a preprocessor directive needed to use printf
3
4 int main(void) //void means no arguments are passed to the function
5 {
6     printf("Hello World!\n"); // \n is a newline character
7     return 0; // return 0 to indicate successful completion
8 }
```

Code using main() will not compile using mex compiler! Use GCC or online C compiler.

```
>> fprintf("Hello World\n")
Hello World
```

Equivalent MATLAB code

Our first “real” C programs

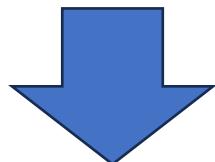
```
1 #include <stdio.h>
2 // above is a preprocessor directive needed to use printf
3
4 // void means no arguments are passed to the function
5 int main(void)
6 {
7     double value = 1.23; // declare a variable of type double and assign it a value
8     printf("The value is %lf\n", value);
9     // %lf (%f can also be used) is a format specifier for a double
10    return 0; // return 0 to indicate successful completion
11 }
```

```
>> value = 1.23;
>> fprintf("The number is %f\n", value);
The number is 1.230000
```

Equivalent MATLAB code (MATLAB
needs %f (%lf doesn't work)

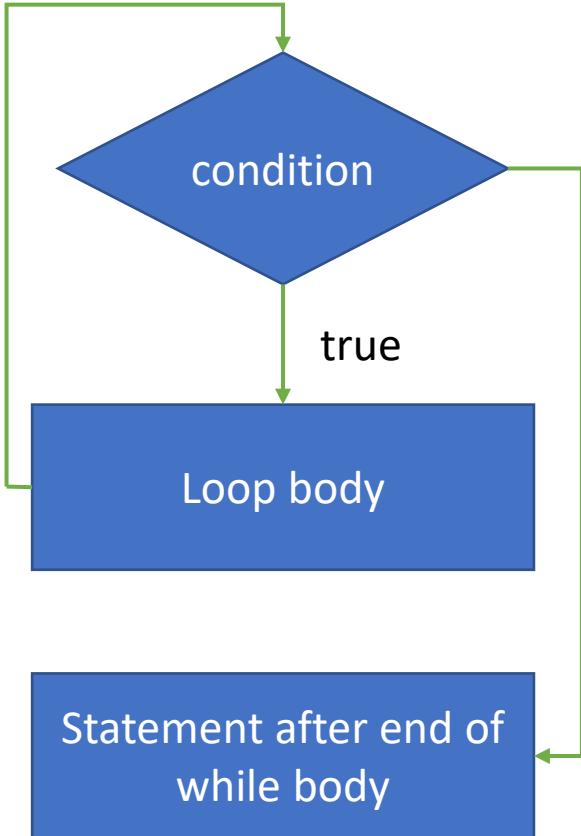
Our first “real” C programs

```
1 #include <stdio.h>
2
3 void showMessage(void)
4 {
5     printf("Hello World! [from showMessage]\n");
6 }
7
8 int main(void)
9 {
10    printf("Hello World! [from main]\n");
11    showMessage(); // call the function showMessage
12    return 0;
13 }
```



```
Hello World! [from main]
Hello World! [from showMessage]
```

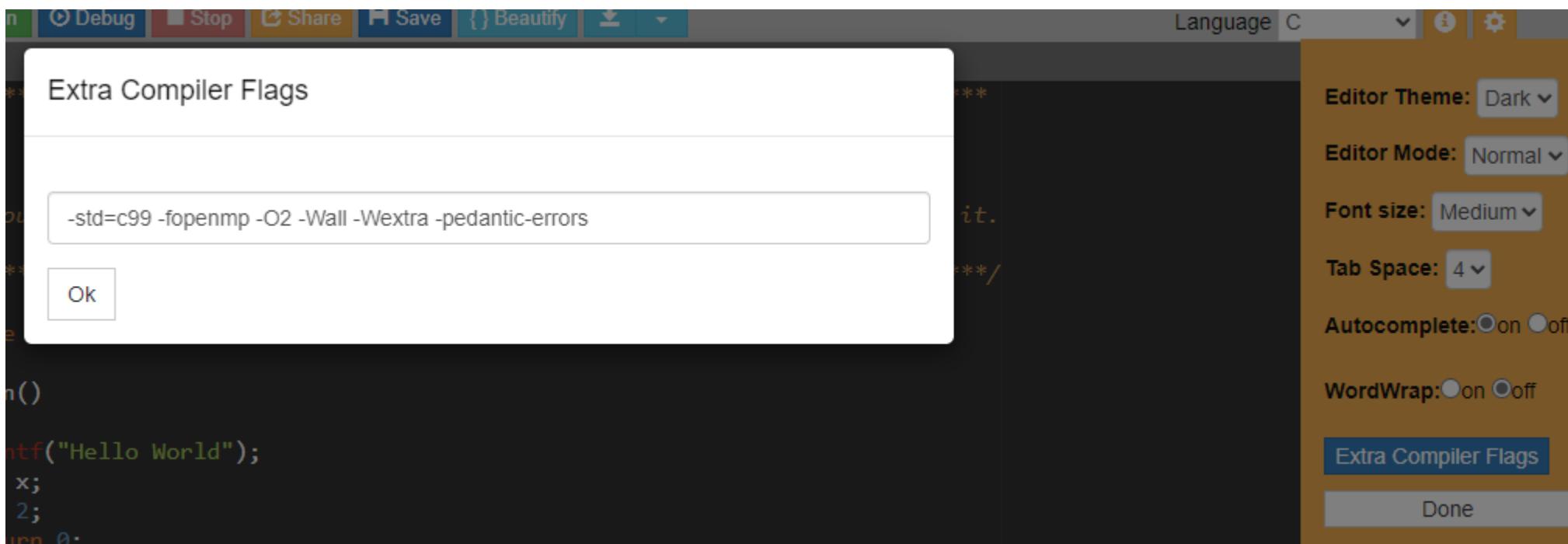
While statement



```
while (condition)
{
    [ loop body ]
}
[Statements after while body]
```

```
1 #include <stdio.h>
2 void showMessage(void);
3 // above is a function prototype for showMessage
4 // showMessage returns void (nothing) and takes no arguments (input)
5 // this is a promise to the compiler that showMessage
6 // will be actually defined later in this file or in some external file
7 int main(void)
8 {
9     int counter = 0; // integer variable counter is declared and initialized to 0
10
11    while (counter < 5) // run as long as counter is less than 5
12    {
13        showMessage(); // call the function showMessage
14        counter = counter + 1; // can also be written as counter++;
15    }
16
17    return 0; // return 0 to indicate successful completion
18 }
19
20 // since we actually define the function showMessage after main [here]
21 // we need to provide a function prototype (see line 2)
22 void showMessage(void)
23 {
24     printf("Hello World! [from showMessage]\n");
25 }
```

- You can run the standard C programs with Online C compiler:
https://www.onlinegdb.com/online_c_compiler
- Please make sure to use these compiler flags (they will help to spot errors and code that does not follow the C99 standard):
-std=c99 -fopenmp -O2 -Wall -Wextra -pedantic-errors



C slide23_main.c > ...

```
1 #include <stdio.h>
2 #include "slide23_showm.h"
3 // above is a preprocessor directive to include the contents of slide23_showm.h
4
5 int main(void)
6 {
7     printf("Hello World! [from main]\n");
8     showMessage(); // call the function showMessage
9     return 0; // return 0 to indicate successful completion
10 }
```

Example of multiple files!

The “primary” code file (slide23_main.c) that includes main-function (only one file can have main)

C slide23_showm.c > ...

```
1 #include <stdio.h>
2 #include "slide23_showm.h"
3 // it is important to include the header file that contains the function prototype
4 // even in the file that actually defines the function (for error checking)
5 void showMessage(void)
6 {
7     printf("Hello World! [from showMessage]\n");
8 }
```

The file “slide23_showm.c” defines the function showMessage

C slide23_showm.h > ...

```
1 #pragma once // Only include this header file once per compilation
2 // it is non-standard but widely supported
3 // alternative is to use #ifndef, #define, #endif (include guards)
4 // for simple codes, include guards (or #pragma once) are usually not necessary
5 void showMessage(void);
```

This the header file “slide23_showm.h” supporting the slide_23_showm.c
Compile (if manually) with:
gcc slide23_main.c slide23_showm.c

Multiple files (using build system)

- If you have many source code files, it may be time to consider a build system. In this course we recommend CMake.
 - CMake (cross-platform). It uses a **CMakeLists.txt** file to describe the build process. This file contains a set of commands and variables that describe the project structure and dependencies.
 - Below is the **CMakeLists.txt** used for codes so far in these slides (executable p5 is based on two source (.c) files). No need for manual compiling! Notice that header files (.h) [such as slide23_showm.h] are NOT listed as compile inputs even if they are actually used by the compiler (this is usual).

M CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.10)
2 project(test_project VERSION 0.1.0 LANGUAGES C)
3 set(CMAKE_C_FLAGS "-std=c99 -fopenmp -O2 -Wall -Wextra -pedantic-errors")
4 add_executable(p1 slide16.c)
5 add_executable(p2 slide17.c)
6 add_executable(p3 slide18.c)
7 add_executable(p4 slide21.c)
8 add_executable(p5 slide23_main.c slide23_showm.c)
```

Division

- In C, the division operator ("/") returns an integer result when dividing two integers. This means that any decimal part of the result is truncated. To avoid this, use datatype double when dividing decimals.
- The modulo operator ("%") is another arithmetic operator in C that returns the remainder of integer division. For example, 7 % 3 would return 1, because 3 goes into 7 twice with a remainder of 1. The modulo operator can be used to check if a number is even or odd, by checking if the result of the modulo operation with 2 is equal to 0 or 1, respectively.

By default, C does integer division! $3/2$ is NOT 1.5 but 1!
Use instead $3.0/2.0$, or $3.0/2$ or $3/2.0$ (for example)

Simple Arrays

- In C, an array is a collection of variables of the same data type, which are accessed using a common name and an index that represents their position in the array.

- Define array of 5 integers (array size is 5) using

```
int arr[5];
```

- We can initialize it using: (this syntax only works for the initialization)

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

It's important to note that the first element of the array is "arr[0]", not "arr[1]". This is because array indices in C start from 0, so the first element is always at index 0, and the last element is at index "size-1", where "size" is the size of the array.

There is no element arr[5]! Accessing it would be serious error

Simple Arrays

C slide27.c > ...

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int arr[5] = {7, 2, 3, 4, 10}; // in C arrays are 0-indexed and array size is fixed after declaration
5     unsigned int counter = 0; // unsigned int is a non-negative integer
6     // sizeof(arr) returns the size of the array in bytes NOT the number of elements
7     // sizeof(int) returns the size of an integer in bytes
8     // Therefore, sizeof(arr) / sizeof(int) returns the number of elements in the array
9     // This avoids hardcoding the number of elements in the array (error-prone)
10    // also #define can be used to define constants (such as array size) at the beginning of the file
11    // notice that C does not protect against buffer overflows (writing beyond the array bounds)!!!
12    while (counter < sizeof(arr) / sizeof(int)) // actually even better to use sizeof(arr) / sizeof(arr[0])
13    {
14        printf("arr[%u] = %d\n", counter, arr[counter]);
15        // %d is a format specifier for an integer
16        // for unsigned int use %u
17        counter++;
18    }
19    return 0; // return 0 to indicate successful completion
20 }
```

[Optional] Simple Arrays (advanced error finding)

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int arr[5] = {1,2,3,4,77};
5     int offset = 5;
6     arr[offset] = 123; // very serious error!!!
7     // last valid array element is arr[4]
8     // you can attempt to detect overflows with WSL using
9     // clang -fsanitize=address ajo.c
10    return 0;
11 }
```

```
=====
==2321==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f381aa00034
WRITE of size 4 at 0x7f381aa00034 thread T0
#0 0x55fdd2c51865 in main (/home/sto2010/wsl_projects/a.out+0x104865)
#1 0x7f381c6de1c9  (/lib/x86_64-linux-gnu/libc.so.6+0x2a1c9) (BuildId:
#2 0x7f381c6de28a in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.
#3 0x55fdd2b78344 in _start (/home/sto2010/wsl_projects/a.out+0x2b344)

Address 0x7f381aa00034 is located in stack of thread T0 at offset 52 in frame
#0 0x55fdd2c5173f in main (/home/sto2010/wsl_projects/a.out+0x10473f)

This frame has 1 object(s):
 [32, 52) 'arr' <== Memory access at offset 52 overflows this variable
```

Comments

- Comment is introduced to a C program by the pair of characters /*, which must not have a space between them. From then on, everything found up to and including the pair of characters */ is gobbled up and the whole lot is replaced by a single space.
- Preferred way to make single line comments is // (C99)

The image shows a code editor window with a dark theme. The file is named "main.c". The code contains two types of comments:

```
1  /* This is a comment.  
2  Comment continues. */  
3  
4  // This is another Comment
```

A callout box highlights the multi-line comment starting at line 1. The text in the callout box is:

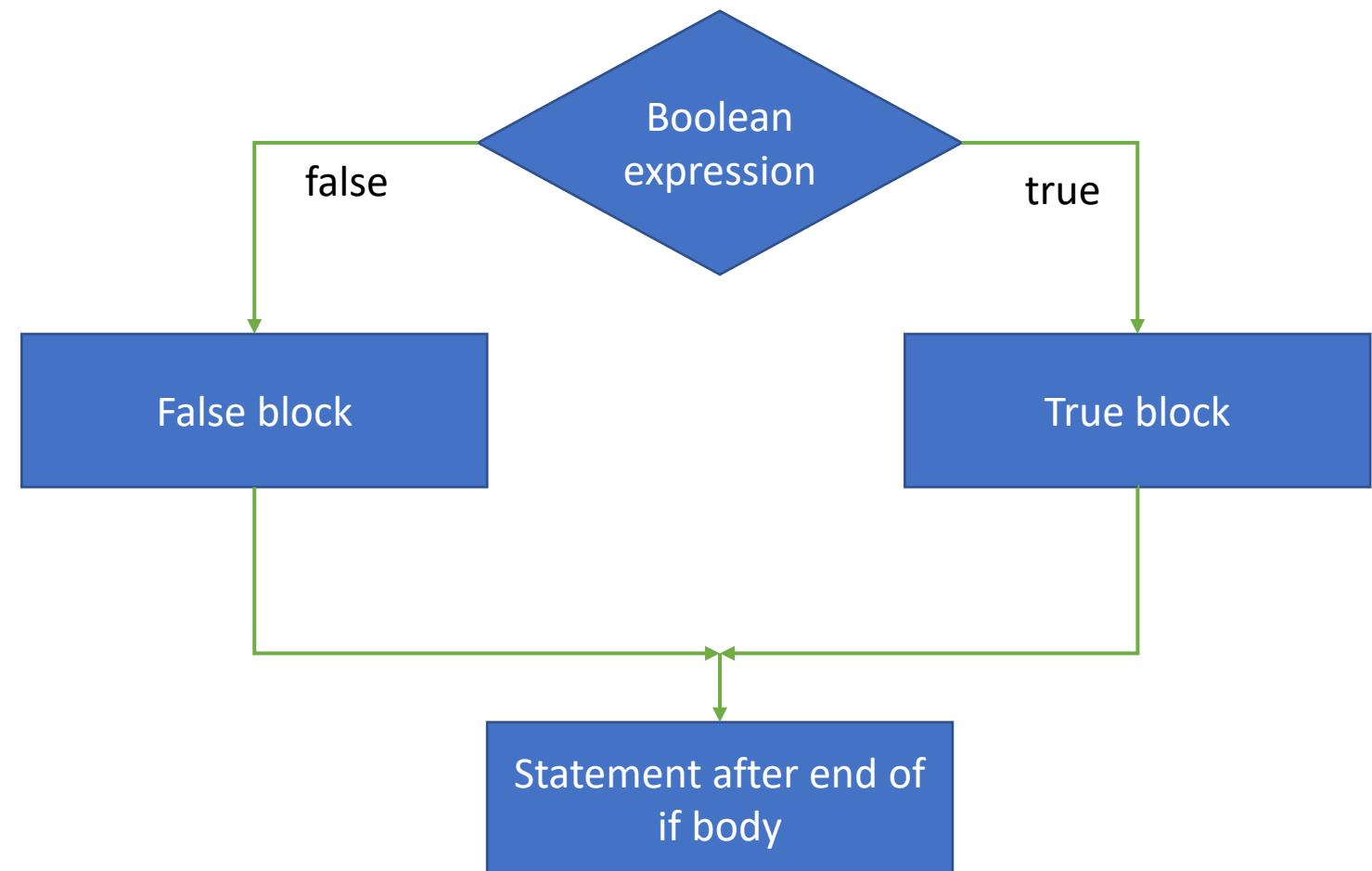
```
/*  
 * Tell the compiler that we intend to use a function called show_message.  
 * It has no arguments and returns no value This is the "declaration".  
 */
```

#define macro example

C slide30.c > ...

```
1 #include <stdio.h>
2 #define ARRAY_SIZE 5
3 // please note ARRAY_SIZE is not a variable (so it does not have a data type)
4 // instead preprocessor replaces ARRAY_SIZE with 5 before compilation in the entire file
5 // compiler will never see ARRAY_SIZE (it will see 5)
6 int main(void)
7 {
8     int arr[ARRAY_SIZE] = {7, 2, 3, 4, 10};
9     unsigned int counter = 0; // unsigned int is a non-negative integer
10    while (counter < ARRAY_SIZE)
11    {
12        printf("arr[%u] = %d\n", counter, arr[counter]);
13        counter++;
14    }
15    return 0;
16 }
```

If-else statement



```
if (expression)
{
    [true block]
}
else
{
    [false block]
}
[Statements after if body]
```

If statement

The if statement

The if statement has two forms:

```
if (expression) statement
```

```
if (expression) statement1  
else statement2
```

Equivalent MATLAB code

```
if expression  
    statement  
end
```

```
if expression  
    statement1  
else  
    statement2  
end
```

In the first form, if (and only if) the expression is non-zero, the statement is executed. If the expression is zero, the statement is ignored. Remember that the statement can be compound; that is the way to put several statements under the control of a single if.

The second form is like the first except that if the statement shown as statement1 is selected then statement2 will not be, and vice versa.

```
C slide33.c > 3 main(void)
1     #include <stdio.h>
2     int main(void)
3     {
4         int number = 10;
5         // Example of if statement
6         if (number > 0)
7             printf("The number is positive.\n");
8         // for single line statements, curly brackets are optional
9         // Example of if-else statement
10        if (number % 2 == 1)
11            printf("The number is odd.\n");
12        else
13            printf("The number is even.\n");
14
15        // Example of if-else with curly brackets for clarity
16        if (number > 5)
17        {
18            printf("The number is %d and ", number); // no newline character
19            printf("the number is greater than 5.\n");
20        }
21        else
22        {
23            printf("The number is 5 or less.\n");
24        }
25        return 0;
26    }
```

```
1 #include <stdio.h>
2 int simple_function(int); // function prototype
3 int main(void)
4 {
5     int a = 7;
6     int b = 8;
7     int a_plus_10 = simple_function(a);
8     printf("a_plus_10 = %d\n", a_plus_10);
9     // we can also pass the result of a function directly to printf
10    printf("b_plus_10 = %d\n", simple_function(b));
11    return 0;
12 }
13
14 int simple_function(int a)
15 {
16     // simple function that adds 10 to the input
17     // return type is int
18     // input type is also int
19     int result = a + 10; // add 10 to the input
20     // result (and also a) are local variables
21     // and will be destroyed when the function returns
22     // but copy of result will be returned to the caller
23     // changing a would not affect the argument (such as "a" or "b") in the caller
24     return result; // return the result of the addition to the caller
25 }
```

Example
of using
“return”
to return
value from
a function

Multiple inputs and one output

- Header file `#include "MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"` is flexible as it allows up to 7 inputs!

First specify `NINPUT_ARGUMENTS` and then `#include` the header

Then in line 5, we notice that we have put 3 double inputs [`input1`, `input2`, `input3`] (this must match the `NINPUT_ARGUMENTS`)

The screenshot shows a MATLAB Command Window with the title bar "example_var_inputs_one_output.c". The window contains the following text:

```
1 // Compile using (on MATLAB Command Window):  
2 // mex -v -R2018a example_var_inputs_one_output  
3 #define NINPUT_ARGUMENTS 3 // Must be first in the file!  
4 #include "MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"  
5 double MATLAB_main(double input1, double input2, double input3)  
6 {  
7     double result;  
8     // Calculate result  
9     result = input1 + input2 + input3;  
10    // Return the result  
11    return result;  
12 }
```

Red arrows point from the `NINPUT_ARGUMENTS` define to the `input1`, `input2`, and `input3` parameters in line 5. A red arrow also points from the `out` variable in the command window to the `out =` part of the command. A yellow box highlights the command `>> out = example_var_inputs_one_output(10,20,100)`. A green arrow points to a text box containing the following text:

We call our own function by the filename (C file)

[Optional Practice] codewars.com



- <https://www.codewars.com/>
- Lot of practice problems!
- If you want to practice, begin with problems of difficulty level 8 kyu
(and even from those select the easiest ones!)

Further issues

The two operators that you haven't seen before are the remainder operator %, and the equality operator, which is a double equal sign ==. That last one is without doubt the cause of more bugs in C programs than any other single factor.

The problem with the equality test is that wherever it can appear it is also legal to put the single = sign. The first, ==, compares two things to see if they are equal, and is generally what you need in fragments like these:

```
if(a == b)  
while (c == d)
```

The assignment operator = is, perhaps surprisingly, also legal in places like those, but of course it assigns the value of the right-hand expression to whatever is on the left. The problem is particularly bad if you are used to the languages where comparison for equality is done with what C uses for assignment. There's nothing that you can do to help, so start getting used to it now. (Modern compilers do tend to produce warnings when they think they have detected 'questionable' uses of assignment operators, but that is a mixed blessing when your choice was deliberate.)

**Do not confuse equality test “==”
and assignment operator “=”**

Mandatory Homework

- **Every week, deadline for the homework return is Wednesday 12:00 (15 minutes before the lecture starts)** It is recommended not to leave returning to last minute but to do this well before the deadline. Upload your solutions to Moodle as ZIPPED file. Your programs must be ready to compile. Include all necessary files to compile the programs in the ZIP file. In addition to the well-commented program, write a sufficiently long report explaining how your program works.

The homework are likely to be easy (not prepared yet, so not 100% sure) but if you feel it is helpful, cooperation between students is allowed but 100% identical codes are not allowed.

Written reports must be 100% independently written by each student (**all group members must anyways be listed in the individual report returned to Moodle by each group member**).

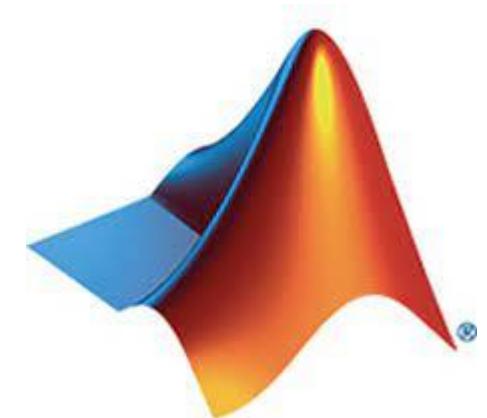
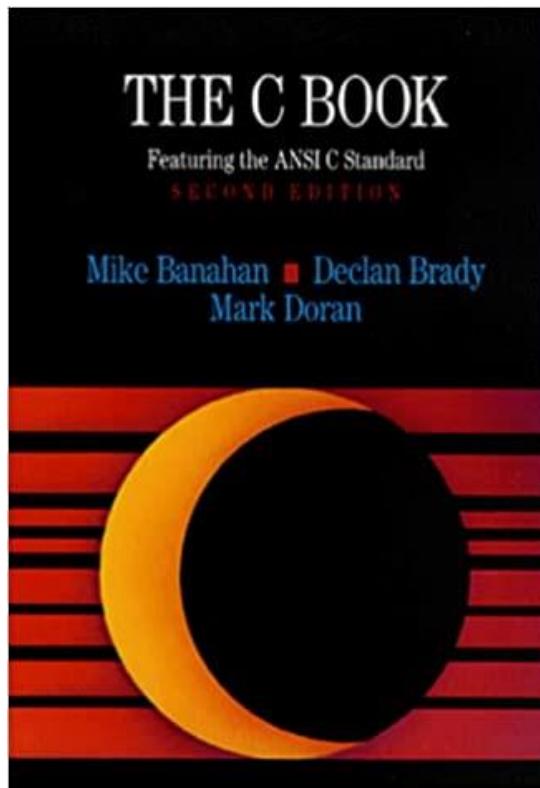
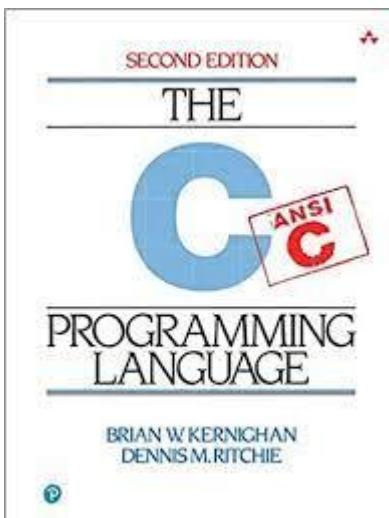
You are not allowed to automatically generate answers to all homework questions with AI. You are allowed to first try your best and then check with AI. Also, you can ask AI coding questions (remember that it can answer incorrectly. For basic questions is might be anyways 99% correct). AI is very useful after you first yourself know the fundamentals of C.

The following is direct feedback from local industry:

"**:understanding how to code because even though the use of AI in coding is increasing, you have to understand the logic of coding**"

- AI can quite often create code with headers, but for "MATLAB C/MEX" tasks, you must use the headers given in the course (in the file CMAT2023 all headers). **Use of different headers files will lead to rejection.** If one of you knows how to make your own headers, then the course is of no use (it would be too simple), so you have to use the provided headers. And in addition, this will hopefully slightly reduce direct use of AI without understanding anything.

Using C with MATLAB #02



https://publications.gbdirect.co.uk/c_book/copyright.html

Keywords

Today we will focus on datatypes: "char", "float", "long",
"short", "signed", "unsigned"

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

Identifiers

- Identifier is the fancy term used to mean ‘name’. In C, identifiers are used to refer to upper- and lower-case things: we’ve already seen them used to name variables and functions.
- The rules for the construction of identifiers are simple: you may use the 52 upper- and lower-case alphabetic characters, the 10 digits and finally the underscore ‘_’, which is an alphabetic character for this purpose.
- The only restriction is the usual one; identifiers must start with an alphabetic character. **However, starting external linkage identifier with “_” is reserved and is not allowed!**

Length of identifiers

- C99 has at least 63 significant characters for internal identifiers and 31 significant characters for external linkage identifiers (such as function names)! Older versions of C (such as ANSI C) may be much more restrictive especially for external linkage identifiers
- C is case-sensitive: for upper- and lower-case "job" and "JoB" are different identifiers
- For full list of **reserved** identifiers (that are not allowed be used by you) please see the link below:
- Identifier - cppreference.com [This website is known to be highly accurate for reference information!]

Variables

- A declaration tells the compiler (and linker) the name and type of a variable or function but does not necessarily allocate storage.
- A definition also allocates storage (for variables) or provides the function body (for functions).
- Function prototype (e.g., `int func(int);`) is a declaration and not a definition.
- `extern int a;` is typically a *declaration*—it tells the compiler that “`a`” exists somewhere else. It does not allocate storage in the current translation unit.
- Automatic (local) **variables** (such as `int x;` inside a function) **are not automatically initialized to 0**; they have indeterminate (garbage) values if not explicitly initialized.
- Static and global (external linkage) variables are zero-initialized by default if no other initializer is provided.

Real types

The varieties of real numbers are these:

```
float  
double  
long double
```

```
root [0] sizeof(float)  
(unsigned long) 4  
root [1] sizeof(double)  
(unsigned long) 8  
root [2] sizeof(long double)  
(unsigned long) 16
```

Each of the types gives access to a particular way of representing real numbers in the target computer. If it only has one way of doing things, they might all turn out to be the same; if it has more than three, then C has no way of specifying the extra ones. The type `float` is intended to be the small, fast representation corresponding to what FORTRAN would call `REAL`. You would use `double` for extra precision, and `long double` for even more.

Example sizes
(in bytes) of
real types

Use of “double” is highly recommended especially when working with MEX
Long double can be in reality be “just” 80 bits padded with zeros! But it has high speed due to hardware support for 80-bits (but may prevent auto vectorization because SSE/AVX typically operate on 64-bit double or 32-bit float data rather than 80-bit extended precision)

Table 2.4. Format codes for real numbers

Real types

```
root [4] 10.0L  
(long double) 10.000000L  
root [5] 10.0  
(double) 10.000000  
root [6] 10.0F  
(float) 10.0000f
```

Type	Format	
float	%f	root [7] 15.75 (double) 15.750000
double	%lf (or %f for printf)	root [8] 1.575E1 (double) 15.750000
long double	%Lf (note: error in PDF version of the book)	root [9] 1575E-2 (double) 15.750000
	<i>Format codes for real numbers</i>	root [10] -2.5E-3 (double) -0.0025000000
		root [11] 1E3 (double) 1000.0000

Real types

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

```
>> 2^(133-127)*(1+7759462*(2^(-23)))  
ans =  
123.199996948242
```

```
>> single(123.2) [in MATLAB Command prompt]  
ans = single      123.2  
>> ans-123  
ans =  
    single   0.1999969
```

```
>> fprintf("%3.16f\n",single(123.2))  
123.1999969482421875
```

But fprintf can show the true value

MATLAB (and C with default printf) is hiding (even with format long g) the full information of the floating point numbers

NAN and INFINITY

```
1 #define NINPUT_ARGUMENTS 1 // Must be first in the file (value
2 #include "MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"
3 double MATLAB_main(double input)
4 {
5     if (isnan(input))
6         mexPrintf("NaN detected\n");
7     if (isinf(input))
8         mexPrintf("Positive or negative infinity detected\n");
9
10    double result;
11    // Calculate result
12    result = 2*input;
13    // Return the result
14    return result;
15 }
```

Nan (not a Number) and Inf (Infinity) are quite important. Basically assuming C99, you can use macro NAN for Nan and INFINITY for Inf.

Remember to include math.h (or better tgmath.h) [already included in our headers]

```
>> a = example(Inf) => Positive or negative infinity detected! => a =  Inf
>> a = example(NaN) => NaN detected! => a =  NaN
>> a = example(10) => a = 20.
Also, INFINITY*0 => NAN [correct] & INFINITY-INFINITY = NAN [correct]
```

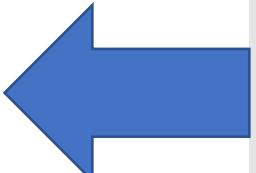
NAN and INFINITY

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

double calc(double input)
{
    if (isnan(input))
        printf("==> NaN detected!\n");
    if (isinf(input))
        printf("==> +-Inf detected!\n");
    return 2*input;
}

int main()
{
    double arr[4] = {NAN, INFINITY, -INFINITY, 10};
    int counter = 0;
    while (counter < sizeof(arr)/sizeof(double))
    {
        printf("2*%lf = %lf\n", arr[counter], calc(arr[counter]));
        counter = counter + 1;
    }
    return 0;
}
```

==> NaN detected!
2*nan = nan
==> +-Inf detected!
2*inf = inf
==> +-Inf detected!
2*-inf = -inf
2*10.000000 = 20.000000



IEEE Single Precision floats

What we want	What we get	In binary representation (32 bit float)
0	0	0 0000000 000000000000000000000000
-0	-0	1 0000000 000000000000000000000000
0.1	0.10000001490116	0 0111011 10011001100110011001101
12345678	12345678	0 10010110 01111000110000101001110
1E-12	0.99999996004197e-12	0 01010111 00011001011110011001100
1E12	99999995904	0 10100110 11010001101010010100101
INF	INF	0 1111111 000000000000000000000000
-INF	-INF	1 1111111 000000000000000000000000
NAN	NAN	0 1111111 100000000000000000000000

- A 32-bit floating-point value (float in most systems) can represent numbers between **1.17549e-38** and **3.40282e+38** (FLT_MIN and FLT_MAX [include <float.h> to use]).
 - Operations that would result in a larger value will produce INF or -INF instead.
 - Operations that would create smaller numbers than FLT_MIN can lead to 0 or denormalized numbers (that can represent even smaller numbers than FLT_MIN but with reduced precision)
- For a 64-bit double, the limits are **1.7976931e+308** to **2.22507389e-308** (DBL_MAX and DBL_MIN on systems where double is 64 bits)
- Long double is typically (but not always) 80 bits with range from **3.3621031E-4932** to **1.1897315E+4932** (LDBL_MIN and LDBL_MAX)

Since C11 these are also defined to support denormalized numbers (working in MEX but not in online C compiler)

```
root [207] FLT_TRUE_MIN
(float) 1.40130e-45f
root [208] DBL_TRUE_MIN
(double) 4.9406565e-324
root [209] LDBL_TRUE_MIN
(long double) 3.6451995e-4951L
```

Overflow and underflow (real types)

- Overflow occurs when a number would be too large to represent, ie, the maximum exponent would be exceeded as the result of an operation. **The result is +/-infinity**.
- Underflow occurs when a number is too small to represent, and the **result is simply replaced by zero**.
- Copyleft 2005 [Fred Swartz MIT License](#)

You can even “catch” for example floating overflow by exceptions!!

https://en.cppreference.com/w/c/numeric/fenv/FE_exceptions

```
root [34] float a = 1E38
(float) 1.00000e+38f
root [35] a = a*10
(float) inff
```

Real types

- Unlike most of the floating point values, integers can be stored exactly in a double up to a large limit (exactly 2^{53} , this and all the smaller integers can be stored exactly).
 - This leads to more range than 32-bit integers!
 - This is one reason why MATLAB is working so well even though its native data type is double
- Of course ≥ 64 bit integer data type (long long) does even better

Real types

- Because of the errors in floating point representation and round-off error you should not compare two floating pointer numbers for equality using “`==`” operator (there are exceptions, such as when you know that the numbers are integers that are stored exactly)
- For example $0.1 + 0.1 + 0.1$ is typically NOT $0.3!!!$

```
root [214] (0.1 + 0.1 + 0.1 == 0.3)
(bool) false
root [215] (1.0 + 1.0 + 1.0 == 3.0)
(bool) true
```

- But $1.0 + 1.0 + 1.0$ is indeed $3.0!$ (since these integers can be stored exactly)
- Better to compare absolute difference between two numbers (use `fabs()` function for absolute floating point value) to some small value (for example)

Real types

The main points of interest are that in the increasing ‘lengths’ of `float`, `double` and `long double`, each type must give at least the same range and precision as the previous type. For example, taking the value in a `double` and putting it into a `long double` must result in the same value.

There is no requirement for the three types of ‘real’ variables to differ in their properties, so if a machine only has one type of real arithmetic, all of C’s three types could be implemented in the same way. None the less, the three types would be considered to be different from the point of view of type checking; it would be ‘as if’ they really were different. That helps when you move the program to a system where the three types really are different—there won’t suddenly be a set of warnings coming out of your compiler about type mismatches that you didn’t get on the first system.

```
int f(void){  
    float f_var;  
    double d_var;  
    long double l_d_var;  
  
    f_var = 1; d_var = 1; l_d_var = 1;  
    d_var = d_var + f_var;  
    l_d_var = d_var + f_var;  
    return(l_d_var);  
}
```

- There are a lot of forced conversions in that example. Let's look at the assignments of the constant value 1 to each of the variables. As the section on constants will point out, that 1 has type int, i.e. it is an integer, not a real constant. The assignment converts the integer value to the appropriate real type, which is easy to cope with.
- The interesting conversions come next. The first of them is on the line
- $d_var = d_var + f_var;$
- What is the type of the expression involving the + operator? The answer is easy when you know the rules. Whenever two different real types are involved in an expression, the lower precision type is first implicitly converted to the higher precision type and then the arithmetic is performed at that precision. The example involves both a double and a float, so the value of f_var is converted to type double and is then added to the value of the double d_var . The result of the expression is naturally of type double too, so it is clearly of the correct type to assign to d_var .

Real types

- The second of the additions is a little bit more complicated, but still perfectly O.K. Again, the value of `f_var` is converted and the arithmetic performed with the precision of `double`, forming the sum of the two variables. Now there's a problem. The result (the sum) is `double`, but the assignment is to a `long double`. Once again the obvious procedure is to convert the lower precision value to the higher one, which is done, and then make the assignment.
- So we've taken the easy ones. The difficult thing to see is what to do when forced to assign a higher precision result to a lower precision destination. In those cases it may be necessary to lose precision, in a way specified by the implementation. Basically, the implementation must specify whether and in what way it rounds or truncates. Even worse, the destination may be unable to hold the value at all.
- If the destination is unable to hold the necessary value—say by attempting to add the largest representable number to itself—then the behaviour is undefined, your program is faulty and you can make no predictions whatsoever about any subsequent behaviour.

Real types

- A lot of expressions involve the use of subexpressions of mixed types together with operators such as +, * and so on. If the operands in an expression have different types, then there will have to be a conversion applied so that a common resulting type can be established; these are the conversions:
- If either operand is a long double, then the other one is converted to long double and that is the type of the result.
- Otherwise, if either operand is a double, then the other one is converted to double, and that is the type of the result.
- Otherwise, if either operand is a float, then the other one is converted to float, and that is the type of the result.

Binary numbers (example using 4-bit numbers)

2^3	2^2	2^1	2^0
8	4	2	1
0	1	0	0
$0*8$	$+ 1*4$	$+ 0*2$	$+ 0*1 = 4$

```
int numb0 = 0b0000;  
int numb1 = 0b0001;  
int numb2 = 0b0010;  
int numb3 = 0b0011;  
int numb4 = 0b0100;
```

In MATLAB C/, you can specify binary values (integers) using prefix 0b (not in standard C!)

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Hexadecimal numbers

- Typical use of hexadecimal numbers is to set bits to 1
- One hexadecimal number is 4 bits
- “F” means that all 4 bits are 1
- 0xFFFF => means that lowest 16 bits are 1
- 0xFFFFFFFF => means lowest 32 bits 1
(this is all bits is the data type is 32-bit integer)

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Warning about octal numbers

- Never start integer constant with 0
 - Those will be octal numbers!!!
 - Compiler will not warn you about using octal numbers!

```
root [5] int test = 0100  
(int) 64  
root [6] test = 0010  
(int) 8
```

Not 100 as you expected but 64!

The alphabet of C

- The Standard requires that an alphabet of 96 symbols is available for C as follows:

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

! " # % & ' () * + , - . /

: ; < = > ? [\] ^ _ { | } ~

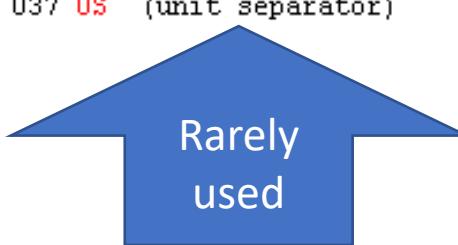
space, horizontal and vertical tab

form feed, newline

US-ASCII CODE (7 bits), contains all C characters (and more)!

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	Ø	96	60	140	`	~
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com



Character constants

- Each valid character has integer value (0-127) based on ASCII code
- For example ‘A’ has integer value 65

```
root [173] int ch = 'A'  
(int) 65  
root [174] printf("Character is ch is %c and its ASCII code is %d\n",ch,ch);  
Character is ch is A and its ASCII code is 65
```

- “A” is not a character but a string! It is composed of two characters, ‘A’ and ‘\0’ (NUL which equal to integer 0) which ends the string
- We can do mathematics with characters since they are integers
- For example ‘A’+10 = 75
- Datatype “**char**” (which is exactly 1 byte) is enough to store the *basic C character set*. It is integer type. It can be **signed** or **unsigned** depending on the implementation. We can for unsigned by “**unsigned char**” or signed with “**signed char**”.
- Notice that for other integer datatypes, “**signed**” is automatically default.

International characters

- UTF-8 coding can in theory support international characters with 8-bit characters, for example “Ö” is sequence of two bytes 0xc3, 0x96
- あ is 0xe3 0x81 0x82
 - However actual success depends on terminal support
- Results in Windows of mexPrintf("あなたはCが好きですか？\n");
- =>
- Not even Ä/Ö is working
- In Linux/MATLAB Online, we get
- => I do not recommend using international character set in Windows MEX
[Even in Linux MEX I recommend just sticking to mexPrintf not doing more complicated string operations]

Char type

```
main.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c = 'a';
6     while (c <= 'z')
7     {
8         printf("Value %d Char %c\n", c, c);
9         c = c + 1;
10    }
11    return 0;
12 }
```

Integral types

Full list at https://en.wikipedia.org/wiki/C_data_types

Format specifier (e.g.
for scanf)

		Minimum bits:	
char	Can contain basic character set. Can be signed or unsigned. It is integer type.	8	%c
short (also short int signed short signed short int)	<i>Short</i> signed integer type. Range contained at least [-32767, +32767]	16	%hi or %hd
int signed signed int	Fundamental signed integer type. Range contained at least [-32767, +32767] range (usually much more!!!!)	16 (usually at least 32!)	%d
unsigned unsigned int	Fundamental unsigned integer type. Range contained at least [0, 65535] (usually much more!!!!)	16 (usually at least 32!)	%u

Integral types

long long int signed long signed long int	<i>Long</i> signed integer type. Range contained at least [-2147483647, +2147483647].	32	%ld
unsigned long unsigned long int	<i>Long</i> unsigned integer type. Can contain at least the range[0, 4294967295].	32	%lu
long long long long int signed long long signed long long int	<i>Long long</i> signed integer type. Can contain numbers at least in the range [-9223372036854775807, +9223372036854775807] . Specified since C99.	64	%lld

You need “long long” (≥ 64 bits) for guaranteed suitability for huge integer sums, etc.

Integral types, printf format codes

Format	Use with
%c	char (in character form)
%d	decimal signed int, short, char
%u	decimal unsigned int, unsigned short, unsigned char
%x	hexadecimal int, short, char
%o	octal int, short, char
%ld	decimal signed long
%lu %lx %lo	as above, but for longs
%lld	decimal signed long long

Table 2.5: More format codes

int64_t

- “long long” does not guarantee that it is exactly 64 bits (it can in theory be more)
- int64_t is alias for some of the already existing integral types (int, long, long long) that is exactly 64 bits (if available)
- How do we print int64_t then when we do not know its actual datatype (could be int, long or long long)? Solution: Use macros such as PRIId64 from inttypes.h:

```
1 #include <stdio.h>
2 #include <inttypes.h>
3
4 int main(void)
5 {
6     int64_t n = 123;
7     printf("Value of n is %" PRIId64 "\n", n);
8     return 0;
9 }
```

NAN and INFINITY (integer types)

Notice that there is no INFINITY or NAN for integer types!

```
root [148] int test = INFINITY  
(int) 2147483647  
root [149] int test = NAN  
(int) 0
```

Note that huge number + something is also not Infinity but instead undefined number (for signed integers)

```
root [164] int test = INFINITY;  
root [165] test+1  
(int) -2147483648
```

Adding 1 to a signed variable that already contains the maximum possible positive number for its type will result in overflow, and the **program's behaviour becomes undefined**.

NAN and INFINITY (integer types)

For unsigned integer types, the standard defines wrapping around

```
root [26] unsigned char test = 255; // Maximum value that can be stored unsigned char
root [27] test = test + 1; // This will deterministically wrap back to zero
root [28] test
(unsigned char) '0x00'
root [29] test = 200;
root [30] test = test + 200;
root [31] printf("Now test is %d\n",test);
Now test is 144
root [32] // 200 + 200 - 256 = 144!!
```

Unsigned numbers work “modulo one greater than the maximum number that they can hold”

```
root [33] (200+200)%(255+1)
(int) 144
```

Expressions and arithmetic

- A lot of expressions involve the use of subexpressions of mixed types together with operators such as +, * and so on. If the operands in an expression have different types, then there will have to be a conversion applied so that a common resulting type can be established; these are the conversions:
- If the integral promotions are applied to both operands and the following conversions are applied:
 - If either operand is an unsigned long int, then the other one is converted to unsigned long int, and that is the type of the result.
 - Otherwise, if either operand is a long int, then the other one is converted to long int, and that is the type of the result.
 - Otherwise, if either operand is an unsigned int, then the other one is converted to unsigned int, and that is the type of the result.
 - Otherwise, both operands must be of type int, so that is the type of the result.

```
root [48] 1U-2
(unsigned int) 4294967295
root [49] if (1U>-1) printf("True\n"); else printf("False\n");
False
```

Do not mix signed and unsigned integer values!

Casts

2.8.1.6 Casts

From time to time you will find that an expression turns out not to have the type that you wanted it to have and you would like to force it to have a different type. That is what **casts** are for. By putting a type name in parentheses, for example

(int)

you create a unary operator known as a **cast**. A cast turns the value of the expression on its right into the indicated type. If, for example, you were dividing two integers a/b then the expression would use integer division and discard any remainder. To force the fractional part to be retained, you could either use some intermediate float variables, or a cast.

Bitwise vs Logical operations

Most common bitwise/logical operations and AND, OR, and NOT.

Here are their truthtables: (XOR is also included)

<https://www.dcode.fr/boolean-truth-table>

AND Truth Table			OR Truth Table			XOR Truth Table			NOT Truth Table		
A	B	A AND B	A	B	A OR B	A	B	A XOR B	A	NOT A	
0	0	0	0	0	0	0	0	0	0	1	
0	1	0	0	1	1	0	1	1	1	0	
1	0	0	1	0	1	1	0	1	0	1	
1	1	1	1	1	1	1	1	0	0	0	

Here 0 means FALSE
and 1 means TRUE

Bitwise vs Logical operations

In C (and in MATLAB), any non-zero value is True. Zero is False.

So for example,

If (7)

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

will print “Hello”.

Same in MATLAB:

```
>> if 7  
    fprintf("Hello\n");  
end  
Hello
```

Bitwise vs Logical operations

Here is for example bitwise AND: (11 & 7)

Bit index	3	2	1	0
11	1	0	1	1
7	0	1	1	1
& operation	0	0	1	1

So result of 11 & 7 is 3.

Logical AND for these values 11 && 7 => 1 (True) [Both values are non-zero meaning True, so we find True AND True => True = 1]

Bitwise vs Logical operations

Here is for example bitwise OR: $(11 \mid 7)$

Bit index	3	2	1	0
11	1	0	1	1
7	0	1	1	1
operation	1	1	1	1

So result of $11 \mid 7$ is 15.

Logical OR for these values $11 \mid\mid 7 \Rightarrow 1$ (True) [Both values are non-zero meaning True, so we find True OR True \Rightarrow True = 1]

Short circuit operation

- The `&&` and `||` operators evaluate arguments left-to-right and do not care about the second argument if the first argument already determines the answer; so

```
0 && collapseSystem();
```

```
1 || collapseSystem();
```

is safe to run.

^ (bitwise XOR) ← “`^`” is not power like in MATLAB but bitwise XOR!!!

`<<`

left shift

`>>`

right shift

Bitwise vs Logical operations

In MATLAB language:

```
>> 11 && 7
```

```
ans =
```

```
logical
```

```
1
```

```
>> 11 & 7
```

```
ans =
```

```
logical
```

```
1
```

Note that in MATLAB `&&` and `&` BOTH do logical AND
Easy to do confuse C and MATLAB! Same for OR (`||`, `|`)
I recommend using `&&` and `||` in MATLAB too!

Bitwise vs Logical operations

Logical not in C is “!”

For example `!100 = false = 0` (because non-zero numbers are true so 100 is considered true and not TRUE = FALSE)

Bitwise not in C is “`~`”

```
root [14] (unsigned char)~0x00
(unsigned char) '0xff'
root [15] (unsigned char)~0xFF
(unsigned char) '0x00'
root [16] (unsigned char)~0x0F
(unsigned char) '0xf0'
root [17] (unsigned char)~0xF0
(unsigned char) '0x0f'
```

In MATLAB logical NOT is “`~`”!!!
Easy to confuse MATLAB and C!

Bitwise vs Logical operations

- For example, to set the low order bits of an int to 0x0f0 and all the other bits to 1, this is the way to do it:

```
unsigned int some_variable= ~0xf0f;
```

- That gives exactly the required result and is completely independent of word length; it is a very common sight in C code.

```
root [38] unsigned int some_variable = ~0xF0F
(unsigned int) 4294963440
root [39] printf("Result = %x\n",some_variable);
Result = fffff0f0
```

Increment and decrement operations

- $x = x + 1;$
- Can be replaced with
`x++;`
or
`++x;`
- $x = x - 1;$
- Can be replaced with
`x--;`
or
`--x;`

Difference between ++x and x++ (and --x and x--)

```
root [67] int y;
root [68] int x = 0;
root [69] y = x++ // Do assignment with current value of x and AFTER that increment x
(int) 0
root [70] x
(int) 1
root [71] x = 0;
root [72] y = ++x //Do increment operation before assignment
(int) 1
root [73] x
(int) 1
```

- So to summarize if “++” is before the variable, BEFORE anything we increment the variable and use the updated value in the expression such as assignment
- If the “++” is after the variable name, we use the current value of x for the expression such as assignment, and AFTER that increment x

Compound assignment operators

- We can replace for example “ $x = x + 5$ ” with “ $x += 5$ ”.
- We can replace for example “ $x = x - 5$ ” with “ $x -= 5$ ”

$*=$	$/=$	$\%=$
$+=$	$-=$	
$\&=$	$ =$	$\hat{=}^=$
$>>=$	$<<=$	

Table 2.8: Compound assignment operators

Operator	Direction	Notes
<code>() [] -> .</code>	left to right	1
<code>! ~ ++ -- - + (cast) * & sizeof</code>	right to left	all unary
<code>* / %</code>	left to right	binary
<code>+ -</code>	left to right	binary
<code><< >></code>	left to right	binary
<code>< <= > >=</code>	left to right	binary
<code>== !=</code>	left to right	binary
<code>&</code>	left to right	binary
<code>^</code>	left to right	binary
<code> </code>	left to right	binary
<code>&&</code>	left to right	binary
<code> </code>	left to right	binary
<code>? :</code>	right to left	2
<code>= += and all combined assignment</code>	right to left	binary
<code>,</code>	left to right	binary

Operator precedence and associativity

```

root [8] 2+3*4/2*12 // 2+12/2*12 = 2+6*12 = 2+72 = 74
(int) 74
root [9] 3*4%5/2 // 12%5/2 = 2/2 = 1
(int) 1
root [10] 3*(4%5)/2 // 3*4/2 = 12/2 = 6
(int) 6

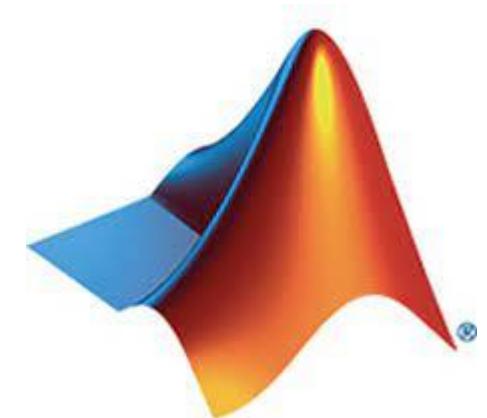
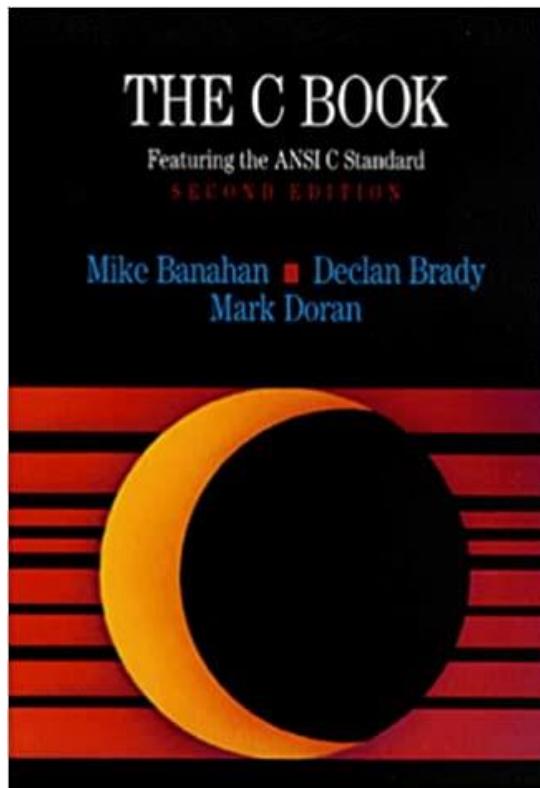
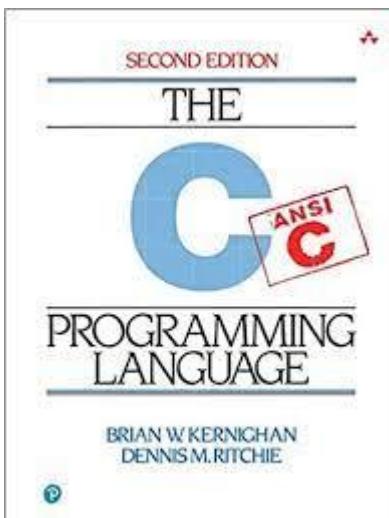
```

Side effects

2.8.5 Side Effects

To repeat and expand the warning given for the increment operators: it is unsafe to use the same variable more than once in an expression if evaluating the expression changes the variable and the new value could affect the result of the expression. This is because the change(s) may be ‘saved up’ and only applied at the end of the statement. So `f = f+1;` is safe even though `f` appears twice in a value-changing expression, `f++;` is also safe, but `f = f++;` is unsafe.

Using C with MATLAB #02



https://publications.gbdirect.co.uk/c_book/copyright.html

Keywords

Today we will focus on datatypes: “char”, “float”, “long”,
“short”, “signed”, “unsigned”

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

Identifiers

- Identifier is the fancy term used to mean ‘name’. We have already seen them used to name variables and functions.
- The rules for the construction of identifiers are simple: you may use the 52 upper- and lower-case alphabetic characters, the 10 digits and finally the underscore ‘_’, which is an alphabetic character for this purpose.
- The only restriction is the usual one; identifiers must start with an alphabetic character. However, starting external linkage identifier with “_” is **reserved** and is not allowed (reserved for the implementation)!

Length of identifiers

- C99 has at least 63 significant characters for internal identifiers and 31 significant characters for external linkage identifiers (such as function names)! Older versions of C (such as ANSI C) may be much more restrictive especially for external linkage identifiers.
- C is case-sensitive: for upper- and lower-case "job" and "JoB" are different identifiers
- For full list of **reserved** identifiers (that are not allowed be used by you) please see the link below:
- Identifier - cppreference.com [This website is known to be highly accurate for reference information!]

Variables

- A declaration tells the compiler (and linker) the name and type of a variable or function but does not necessarily allocate storage.
- A definition also allocates storage (for variables) or provides the function body (for functions).
- Function prototype (e.g., `int func(int);`) is a declaration and not a definition.
- `extern int a;` is typically a *declaration*—it tells the compiler that “`a`” exists somewhere else. It does not allocate storage in the current translation unit.
- Automatic (local) **variables** (such as `int x;` inside a function) **are not automatically initialized to 0**; they have indeterminate (garbage) values if not explicitly initialized.
- Static and global (external linkage) variables are zero-initialized by default if no other initializer is provided.

Real types

The varieties of real numbers are these:

```
float  
double  
long double
```

```
root [0] sizeof(float)  
(unsigned long) 4  
root [1] sizeof(double)  
(unsigned long) 8  
root [2] sizeof(long double)  
(unsigned long) 16
```

Each of the types gives access to a particular way of representing real numbers in the target computer. If it only has one way of doing things, they might all turn out to be the same; if it has more than three, then C has no way of specifying the extra ones. The type `float` is intended to be the small, fast representation corresponding to what FORTRAN would call `REAL`. You would use `double` for extra precision, and `long double` for even more.

Example sizes
(in bytes) of
real types

- **Use of “double” is highly recommended when working with MATLAB+C**
- Long double can be in reality be “just” 80 bits padded with zeros! But it has high speed due to hardware support for 80-bits (but may prevent auto vectorization because SSE/AVX typically operate on 64-bit double or 32-bit float data rather than 80-bit extended precision)

Table 2.4. Format codes for real numbers

Real types

```
root [4] 10.0L  
(long double) 10.000000L  
root [5] 10.0  
(double) 10.000000  
root [6] 10.0F  
(float) 10.0000f
```

Type	Format	
float	%f	root [7] 15.75 (double) 15.750000
double	%lf (or %f for printf)	root [8] 1.575E1 (double) 15.750000
long double	%Lf (note: error in PDF version of the book)	root [9] 1575E-2 (double) 15.750000
<i>Format codes for real numbers</i>		
		root [10] -2.5E-3 (double) -0.0025000000
		root [11] 1E3 (double) 1000.0000

⚠ Avoid long doubles in Windows ⚠

- On Windows, the default C runtime basically treats long double as double and can produce wrong output with %Lf
- The following (correct) code can wrongly print zero!
- Avoid platform-specific functions (even with preprocessor selection)!

```
C slide8.c > ...
1 #include <stdio.h>
2 int main(void)
3 {
4     long double c = 42;
5     printf("c = %Lf\n", c); // does not work in Windows + MinGW
6     __mingw_printf("c = %Lf\n", c); // works in Windows + MinGW
7     return 0;
8 }
```

Background: Binary numbers (4-bit example)

- Binary numbers use **base 2** (digits 0 and 1) instead of decimal (base 10).
- Each digit (bit) represents a power of 2. For example, $0b1011 = 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 8 + 2 + 1 = 11$

```
int x = 0b1011; // binary literal  
(since C23)
```

- Why Binary?
 - This is how computers actually work and store information (series of ones and zeros).
 - Useful for bitwise operations, setting flags, or hardware register manipulation.
 - Easier to see which individual bits are 1 or 0 compared to decimal or hex.

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

IEEE 32-bit floats

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

```
>> single(123.2) [in  
MATLAB Command prompt]  
ans = single  
123.2
```

```
>> fprintf("%3.16f\n",single(123.2))  
123.1999969482421875
```

What you “want” is not what you get!

But fprintf can
show the true
value

MATLAB is hiding (even with “format long g”) the full information of the floating point numbers

IEEE 754 Converter (JavaScript), V0.22

Sign	Exponent							Mantissa														
Value:	+1	2^6							1.9249999523162842													
Encoded as:	0	133							7759462													
Binary:	<input type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/>							<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>													
You entered															123.2							
Value actually stored in float:															123.1999969482421875							
Error due to conversion:															-0.0000030517578125							
Binary Representation															0100010111011001100110011001100110							
Hexadecimal Representation															0x42f66666							
																		+1				
																		-1				

```
>> temp = 2^(133-127) * (1+7759462*(2^(-23)));
>> fprintf("%.16f\n", temp)
123.1999969482421875
```

In the IEEE 754 single-precision format, the exponent is stored with a bias of 127, so first we subtract 127 from the stored exponent (133) to get 6. Next, the fraction bits represent the part after the decimal, and we add 1.0 in front because normalized floats always have an implicit leading 1. This results in a number which is very close to 123.2 but not exact due to how binary floating-point stores numbers.

⚠️ Warnings about real types ⚠️

- Floating-Point Is an Approximation
 - Not all decimal values (even “simple” ones like 123.2) can be represented exactly in binary floating-point.
- Avoid Direct Equality Checks
 - Do not compare floats or doubles with `==` unless you are certain they represent integers, or you have a known pattern. For example this is false `0.1+0.1+0.1==0.3`
 - Instead, compare within a small tolerance (an “epsilon”).
- Avoid long doubles in Windows
- Accumulated Error
 - Repeated arithmetic on floating-point values can accumulate rounding errors, especially in loops or large sums.
 - Order of operations can affect the final result.

Infinity (INFINITY) and Not-a-Number (NAN) in C99

- Why they matter
 - Provide explicit representations for invalid or unbounded results in floating-point arithmetic. Results will not be “random”.
 - Help detect and handle special cases.
- Key Points
 - Stored as pre-agreed special binary patterns.
 - NAN arises for an undefined result (e.g., $0.0 / 0.0$).
 - INFINITY represents positive infinity (e.g., $1.0 / 0.0$ on many systems). Can occur due to “overflow”. Negative version -INFINITY.
 - These macros are defined in C99’s `<math.h>` and in `<tgmath.h>`
 - Check for `isnan()` and `isinf()` if needed for your algorithm.
 - INFINITY and NAN apply only to floating-point (not integers)
- If really needed, you can even “detect” overflow (and also underflow)
 - https://en.cppreference.com/w/c/numeric/fenv/FE_exceptions

NAN and INFINITY

```
slide12.c  X +  
1 #define NINPUT_ARGUMENTS 1  
2 #include "MULTIPLE_INPUTS_ONE_OUTPUT_REAL.h"  
3  
4 double MATLAB_main(double input)  
5 {  
6     if (isnan(input))  
7         printf("The input to C is NaN (not a number)\n");  
8     if (isinf(input))  
9         printf("The input to C is infinity\n");  
10    return 2*input;  
11 }
```

out = slide12(Inf)

The input to C is infinity

out = Inf

>> out = slide12(NaN)

The input to C is NaN (not a number)

out = NaN

>> out = slide12(10)

out = 20

In C99, you can use macro NAN for NaN and INFINITY for Inf.

Remember to include math.h (or tgmath.h)

[already included in our MEX headers]

```
1 #include <stdio.h>
2 #include <math.h> // alternative: <tgmath.h> (supports complex numbers)
3 double calc(double x);
4 int main(void)
5 {
6     double arr[4] = {NAN, INFINITY, -INFINITY, 10.0};
7     for (size_t i = 0; i < 4; i++) // size_t because it is the type returned by sizeof
8     {
9         printf("Processing element %zu\n", i);
10        if (isnan(arr[i]))
11        {
12            printf("NaN detected\n");
13        }
14        else if (isinf(arr[i]))
15        {
16            if (signbit(arr[i])) // non-zero if negative
17                printf("Negative Infinity detected\n");
18            else
19                printf("Infinity detected\n");
20        }
21        printf("calc(%lf) = %lf\n\n", arr[i], calc(arr[i]));
22    }
23    return 0;
24 }
25 double calc(double x)
26 {
27     return 2 * x; // doubles can handle infinities and NaNs
28 }
```

⚠️ Using compiler flags like `-ffast-math` (GCC) can break strict IEEE 754 rules and assume no NaN or ∞ values occur, potentially causing incorrect behavior or unexpected optimizations when dealing with special floating-point values.

However, such compiler flags can also (potentially) greatly increase speed

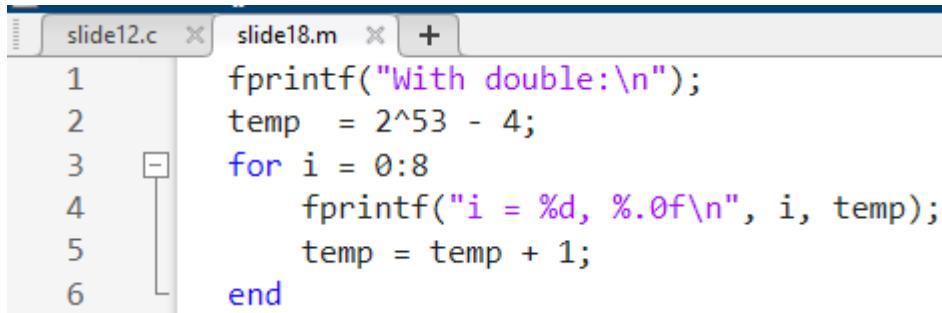
IEEE Single Precision floats

What we want	What we get	In binary representation (32 bit float)
0	0	0 0000000 000000000000000000000000
-0	-0	1 0000000 000000000000000000000000
0.1	0.10000001490116	0 0111011 10011001100110011001101
12345678	12345678	0 10010110 01111000110000101001110
1E-12	0.99999996004197e-12	0 01010111 00011001011110011001100
1E12	99999995904	0 10100110 11010001101010010100101
INF	INF	0 1111111 000000000000000000000000
-INF	-INF	1 1111111 000000000000000000000000
NAN	NAN	0 1111111 100000000000000000000000

Real types for storing integer-values

- Unlike most of the floating-point values, a 64-bit **double** in IEEE 754 can exactly represent every integer value up to 2^{53}
- This exceeds the maximum value of a 32-bit integer and explains why MATLAB's double-based loops (remember in MATLAB double is the default) can handle reasonable integer ranges without losing precision.
- A **64-bit integer** (e.g., “long long” on many platforms) can represent integers up to $2^{63}-1$, which is larger than 2^{53} . So, for **extremely** large integers, a 64-bit integer type will still “do better” (but will not have INFINITY etc.)

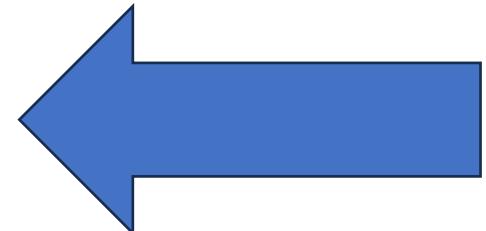
Real types for storing integer-values



```
slide18.m × +
1     fprintf("With double:\n");
2     temp = 2^53 - 4;
3     for i = 0:8
4         fprintf("i = %d, %.0f\n", i, temp);
5         temp = temp + 1;
6     end
```

With double:

i = 0, 9007199254740988
i = 1, 9007199254740989
i = 2, 9007199254740990
i = 3, 9007199254740991
i = 4, 9007199254740992
i = 5, 9007199254740992
i = 6, 9007199254740992
i = 7, 9007199254740992
i = 8, 9007199254740992



⚠ Loops gets stuck at
9007199254740992!!!!
This is not “overflow”
but issue with precision

Conversions of Real types

- A lot of expressions involve the use of subexpressions of mixed types together with operators such as +, * and so on. If the operands in an expression have different types, then there will have to be a conversion applied so that a common resulting type can be established; these are the conversions:
 - If either operand is a long double, then the other one is converted to long double and that is the type of the result.
 - Otherwise, if either operand is a double, then the other one is converted to double, and that is the type of the result.
 - Otherwise, if either operand is a float, then the other one is converted to float, and that is the type of the result.
 - In C you can force casting (conversion) by for example writing:

```
(double) x          // this will cast x to be double
```

Hexadecimal and octal numbers !

- Typical use of hexadecimal numbers is to set bits to 1
- One hexadecimal number is 4 bits
- “F” means that all 4 bits are 1
- 0xFFFF => means that lowest 16 bits are 1
- Never start integer constant with 0 !
 - Those will be octal number!!!
 - Compiler will not warn you about using octal numbers

```
root [5] int test = 0100
(int) 64
root [6] test = 0010
(int) 8
```

Not 100 as you expected but 64!

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Character constants

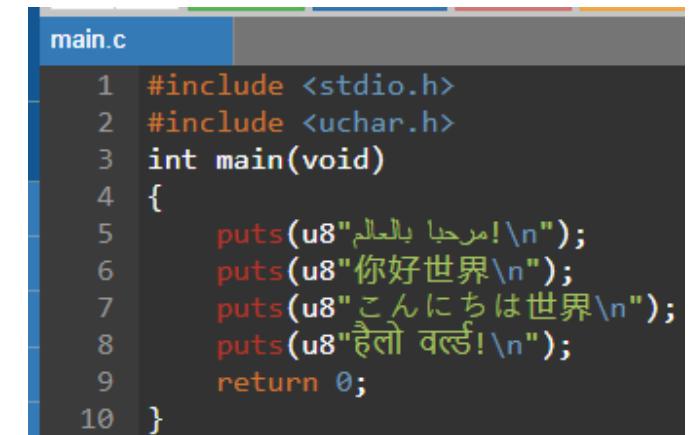
- Each valid character has integer value typically based on ASCII code
- For example, ‘A’ has integer value 65 (ASCII)

```
root [173] int ch = 'A'  
(int) 65  
root [174] printf("Character is ch is %c and its ASCII code is %d\n",ch,ch);  
Character is ch is A and its ASCII code is 65
```

- We can do mathematics with characters since they are integers
- For example, ‘A’+10 = 75
- Datatype “**char**” (which is almost always exactly 1 byte = 8 bits) is enough to store the *basic C character set*. It is integer type. It can be **signed** or **unsigned** depending on the implementation. We can for unsigned by “**unsigned char**” or signed with “**signed char**”. Notice that for other integer datatypes, “**signed**” is automatically default.
- “A” is totally different than ‘A’ ! ! !
 - “A” is array of characters (string) composed of two characters, ‘A’ and ‘\0’ (NUL which equal to integer 0) which ends the string. (Unlike C++, C does not have a “proper” string type)
 - [More exact description]: Actually, “A” is string literal and gives pointer to the array of two characters (not modifiable as it may be stored in read-only location)

UTF-8 in C23

- Normal “C” does not support international characters reasonable way
- UTF-8 stores international characters as sequence of 8-bit characters (each international character can take variable number of 8-bit characters, chars)
- C23 allows use of u8 prefix and for example printf may work (depending on for example terminal support)
- However, string functions such as strlen (returns length of a string) does not work as expected (strlen etc. will be discussed in later lectures)
- For realistic use, external libraries such iconv are required.



```
main.c
1 #include <stdio.h>
2 #include <uchar.h>
3 int main(void)
4 {
5     puts(u8"مرحبا بالعالم\n");
6     puts(u8"你好世界\n");
7     puts(u8"こんにちは世界\n");
8     puts(u8"הOLA וולְדִי!\n");
9
10 }
```

Char type

```
C slide23.c > ⚭ main(void)
1   #include <stdio.h>
2
3   int main(void)
4   {
5       char c = 'a'; // ASCII value of 'a' is 97
6       while (c <= 'z')
7       {
8           printf("c: %c has integer value: %d\n", c, c);
9           c++; // increment c by 1
10      }
11      return 0;
12  }
```

Integral types

Full list at https://en.wikipedia.org/wiki/C_data_types

Minimum bits:

char	Can contain basic character set. Can be signed or unsigned. It is integer type.	8		
short (also short int signed short signed short int)	<i>Short</i> signed integer type. Range contained at least [-32767, +32767]	16		
int signed signed int	Fundamental signed integer type. Range contained at least [-32767, +32767] range (usually much more!!!!)	16 (usually at least 32!)		
unsigned unsigned int	Fundamental unsigned integer type. Range contained at least [0, 65535] (usually much more!!!!)	16 (usually at least 32!)		

Integral types

long long int signed long signed long int	<i>Long</i> signed integer type. Range contained at least [-2147483647, +2147483647].	32	
unsigned long unsigned long int	<i>Long</i> unsigned integer type. Can contain at least the range[0, 4294967295].	32	
long long long long int signed long long signed long long int	<i>Long long</i> signed integer type. Can contain numbers at least in the range [-9223372036854775807, +9223372036854775807] . Specified since C99.	64	

`size_t` (output type of `sizeof`)

int64 t

exactly 64 bits (signed)

uint64 t

exactly 64 bits (unsigned)

(⚠️ Implementations are (in theory) not required to support these, also int8_t int16_t int32_t etc.)

Common Integer printf Format Specifiers (C99+)

Type	Specifier	Example
int	%d (signed)	printf("%d", myInt);
	%u (unsigned)	printf("%u", myUInt);
long	%ld (signed)	printf("%ld", myLong);
	%lu (unsigned)	printf("%lu", myULong);
long long	%lld (signed)	printf("%lld", myLL);
	%llu (unsigned)	printf("%llu", myULL);
size_t	%zu (unsigned) (for signed ssize_t in POSIX, %zd)	printf("%zu", mySize);
Fixed-Width Types	Format Macros from <inttypes.h>	
int64_t	PRId64 (signed)	printf "%" PRId64, my64);
uint64_t	PRIu64 (unsigned)	printf "%" PRIu64, myU64);



Integer overflow

Adding 1 to a signed variable that already contains the maximum possible positive number for its type will result in overflow, and the **program's behavior becomes undefined**.

Notice that there is no INFINITY or NAN for integer types

```
root [148] int test = INFINITY
(int) 2147483647
root [149] int test = NAN
(int) 0
```

Integer overflow

For unsigned integer types, the standard defines wrapping around

```
root [26] unsigned char test = 255; // Maximum value that can be stored unsigned char
root [27] test = test + 1; // This will deterministically wrap back to zero
root [28] test
(unsigned char) '0x00'
root [29] test = 200;
root [30] test = test + 200;
root [31] printf("Now test is %d\n",test);
Now test is 144
root [32] // 200 + 200 - 256 = 144!!
```

Unsigned numbers work “modulo one greater than the maximum number that they can hold”

```
root [33] (200+200)%(255+1)
(int) 144
```



Expressions and arithmetic

- A lot of expressions involve the use of subexpressions of mixed types together with operators such as +, * and so on. If the operands in an expression have different types, then there will have to be a conversion applied so that a common resulting type can be established; these are the conversions:
- If the integral promotions are applied to both operands and the following conversions are applied:
 - If either operand is an unsigned long int, then the other one is converted to unsigned long int, and that is the type of the result.
 - Otherwise, if either operand is a long int, then the other one is converted to long int, and that is the type of the result.
 - Otherwise, if either operand is an unsigned int, then the other one is converted to unsigned int, and that is the type of the result.
 - Otherwise, both operands must be of type int, so that is the type of the result.

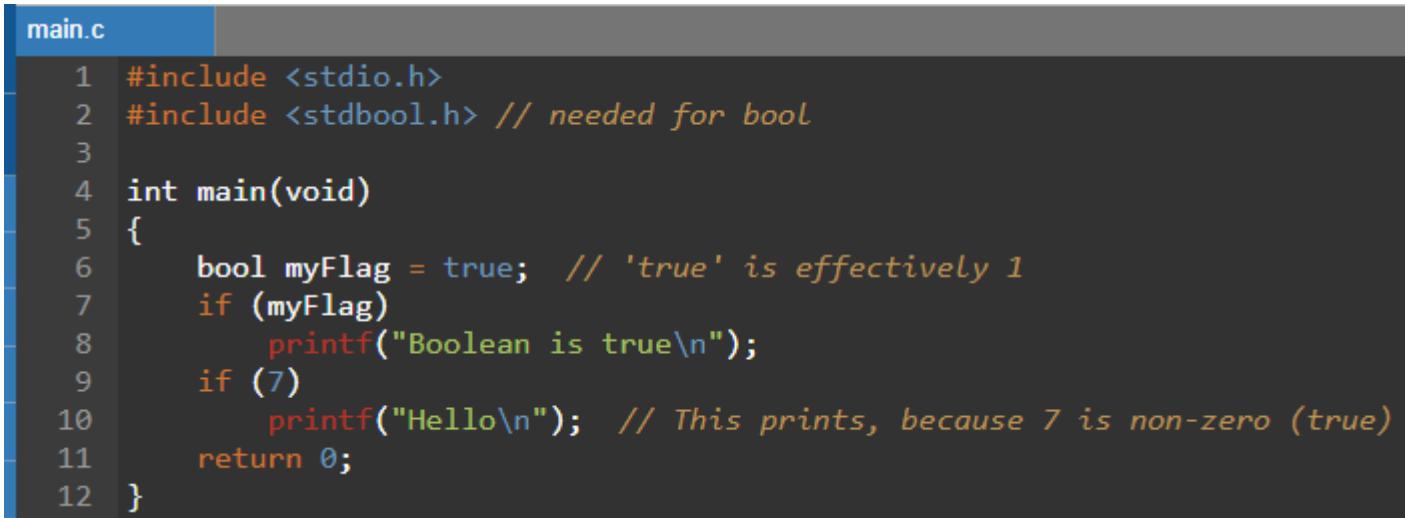
```
root [48] 1U-2
(unsigned int) 4294967295
root [49] if (1U>-1) printf("True\n"); else printf("False\n");
False
```



Do not mix signed and unsigned integer values!

True and false

- In C (and in MATLAB), any non-zero value is True.
- Booleans can be either true or false.
- Using the C99 bool type can make code more readable and self-documenting than just using int for Booleans.

A screenshot of a code editor showing a file named "main.c". The code uses the C99 stdbool.h header to define a boolean variable "myFlag". It demonstrates that both "true" (which is effectively 1) and the non-zero integer 7 evaluate to true in a boolean context. The code is as follows:

```
main.c
1 #include <stdio.h>
2 #include <stdbool.h> // needed for bool
3
4 int main(void)
5 {
6     bool myFlag = true;    // 'true' is effectively 1
7     if (myFlag)
8         printf("Boolean is true\n");
9     if (7)
10        printf("Hello\n"); // This prints, because 7 is non-zero (true)
11     return 0;
12 }
```

Bitwise vs Logical operations

Here is for example bitwise AND: (11 & 7)

Bit index	3	2	1	0
11	1	0	1	1
7	0	1	1	1
& operation	0	0	1	1

So, result of 11 & 7 is 3. AND outputs 1 only if both inputs are 1.

Logical AND for these values 11 && 7 => 1 (True) [Both values are non-zero meaning True, so we find True AND True => True = 1]

Bitwise vs Logical operations

Here is for example bitwise OR: $(11 \mid 7)$

Bit index	3	2	1	0
11	1	0	1	1
7	0	1	1	1
operation	1	1	1	1

So, result of $11 \mid 7$ is 15. OR outputs 1 if one (or both) inputs are 1.

Logical OR for these values $11 \mid\mid 7 \Rightarrow 1$ (True) [Both values are non-zero meaning True, so we find True OR True \Rightarrow True = 1]

Short circuit operation

- The `&&` and `||` operators evaluate arguments left-to-right and do not care about the second argument if the first argument already determines the answer; so

```
0 && collapseSystem();
```

```
1 || collapseSystem();
```

is safe to run.



`^` (bitwise XOR) ← “`^`” is not power like in MATLAB but
bitwise XOR!!!

For example, 2^8 is 10 not 256

Bitwise vs Logical operations

Logical not in C is “!”

For example `!100 = false = 0` (because non-zero numbers are true so 100 is considered true and not TRUE = FALSE)

Bitwise not in C is “`~`”

```
root [14] (unsigned char)~0x00
(unsigned char) '0xff'
root [15] (unsigned char)~0xFF
(unsigned char) '0x00'
root [16] (unsigned char)~0x0F
(unsigned char) '0xf0'
root [17] (unsigned char)~0xF0
(unsigned char) '0x0f'
```



In MATLAB logical NOT is “`~`”!!!
Easy to confuse MATLAB and C!

Increment and decrement operations

- $x = x + 1;$
- Can be replaced with
`x++;`
or
`++x;`
- $x = x - 1;$
- Can be replaced with
`x--;`
or
`--x;`

Difference between ++x and x++ (and --x and x--)

```
root [67] int y;
root [68] int x = 0;
root [69] y = x++ // Do assignment with current value of x and AFTER that increment x
(int) 0
root [70] x
(int) 1
root [71] x = 0;
root [72] y = ++x //Do increment operation before assignment
(int) 1
root [73] x
(int) 1
```

- So to summarize if “++” is before the variable, BEFORE anything we increment the variable and use the updated value in the expression such as assignment
- If the “++” is after the variable name, we use the current value of x for the expression such as assignment, and AFTER that increment x

Compound assignment operators

- We can replace for example “ $x = x + 5$ ” with “ $x += 5$ ”.
- We can replace for example “ $x = x - 5$ ” with “ $x -= 5$ ”

$*=$	$/=$	$\%=$
$+=$	$-=$	
$\&=$	$ =$	$\hat{=}^=$
$>>=$	$<<=$	

Table 2.8: Compound assignment operators

Operator	Direction	Notes
<code>() [] -> .</code>	left to right	1
<code>! ~ ++ -- - + (cast) * & sizeof</code>	right to left	all unary
<code>* / %</code>	left to right	binary
<code>+ -</code>	left to right	binary
<code><< >></code>	left to right	binary
<code>< <= > >=</code>	left to right	binary
<code>== !=</code>	left to right	binary
<code>&</code>	left to right	binary
<code>^</code>	left to right	binary
<code> </code>	left to right	binary
<code>&&</code>	left to right	binary
<code> </code>	left to right	binary
<code>? :</code>	right to left	2
<code>= += and all combined assignment</code>	right to left	binary
<code>,</code>	left to right	binary

Operator precedence and associativity

```

root [8] 2+3*4/2*12 // 2+12/2*12 = 2+6*12 = 2+72 = 74
(int) 74
root [9] 3*4%5/2 // 12%5/2 = 2/2 = 1
(int) 1
root [10] 3*(4%5)/2 // 3*4/2 = 12/2 = 6
(int) 6

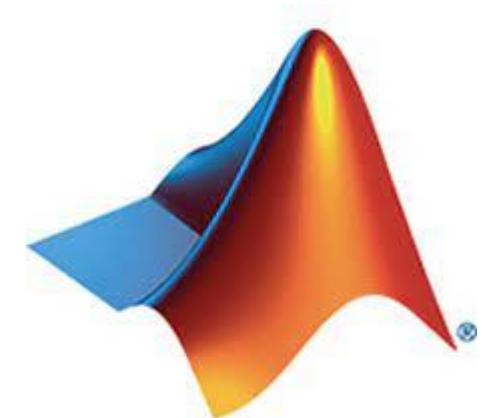
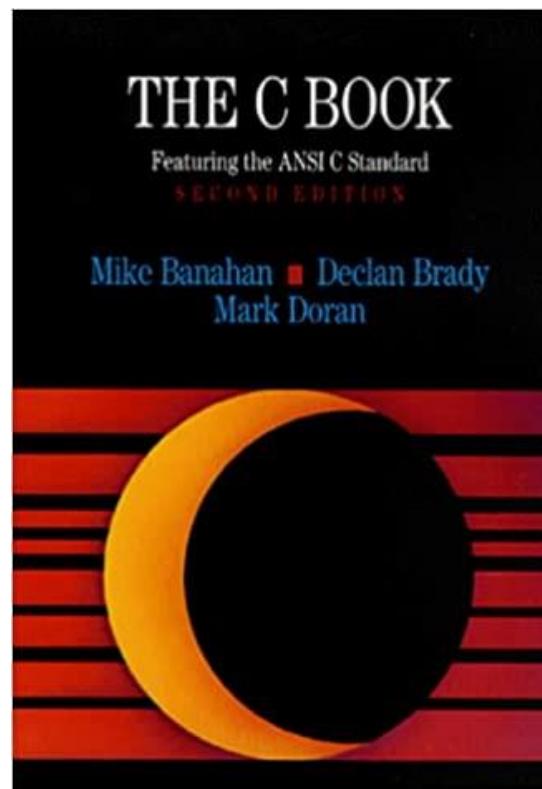
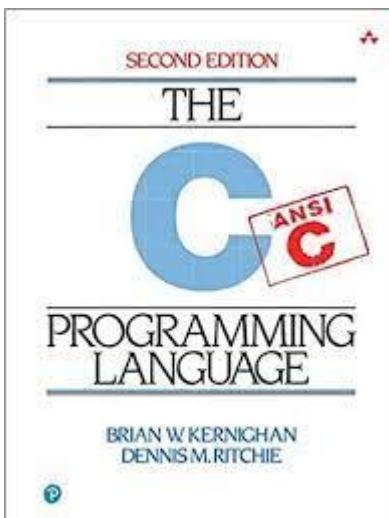
```

⚠ Side effects

2.8.5 Side Effects

To repeat and expand the warning given for the increment operators: it is unsafe to use the same variable more than once in an expression if evaluating the expression changes the variable and the new value could affect the result of the expression. This is because the change(s) may be ‘saved up’ and only applied at the end of the statement. So `f = f+1;` is safe even though `f` appears twice in a value-changing expression, `f++;` is also safe, but `f = f++;` is unsafe.

Using C with MATLAB #03



https://publications.gbdirect.co.uk/c_book/copyright.html

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

Logical expressions and Relational Operators

- We have already seen the use of simple logical expressions such as this:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int age = 20;
6
7     if (age != 10)
8         printf("Age is not equal to 10!\n");
9     return 0;
10 }
```

Logical expressions and Relational Operators

- Here is list of the relational expressions:

Operator	Operation
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Table 3.1: Relational operators

Logical expressions and Relational Operators

C's concept of 'true' and 'false' boils down to simply 'non-zero' and 'zero', respectively. Where we have seen expressions involving relational operators used to control do and if statements, we have just been using expressions with numeric results. If the expression evaluates to non-zero, then the result is effectively true. If the reverse is the case, then of course the result is false. Anywhere that the relational operators appear, so may any other expression. The relational operators work by comparing their operands and giving zero for false and (remember this) one for true. The result is of type int.

Extra material: C++ and C99 bool

- In C++, the output type of relational operators is “bool”
 - However, in real life the difference to C is quite minimal as true basically corresponds to “1” and false to “0”
- C99 can use “bool” datatype but you need to #include <stdbool.h>
- In C++ and C99, **bool flag = 123;** will lead to true (or 1) being stored (not 123)
- One benefit is that std::print (C++23) can directly print Booleans as true or false [not 1 or 0] (std::print is recommended over classical std::cout)
- To get std::print working, #include <print> and in MinGW:

```
cmake_minimum_required(VERSION 3.14)
project(cpp_learn VERSION 0.1.0 LANGUAGES CXX)
set(CMAKE_CXX_FLAGS "-std=gnu++2b -O2 -Wall -Wextra -pedantic-errors")
add_executable(test test.cpp)
target_link_libraries(test stdc++exp)
```

Extra material: C++ and C99 bool

```
1 #include <cstdio> // for printf
2 #include <print> // C++23 for std::print, highly recommended!
3 #include <iostream> // Old C++ way for printing
4 int main()
5 {
6     bool done = (10 > 5);
7
8     printf("printf: The value of done is %d\n", done); // Normal C has no format specifier for bool
9     // std::printf is recommended instead of printf in C++ code
10
11    // std::print does not need format specifiers, its automatic!!!
12    std::println("println: The value of done is {} and as int {}", done, static_cast<int>(done));
13
14    std::cout << "cout: The value of done is " << done << std::endl; // Old C++ way
15    std::cout << "cout: The value of done is " << std::boolalpha << done << std::endl; // Old C++ way
16
17    return 0;
18 }
```

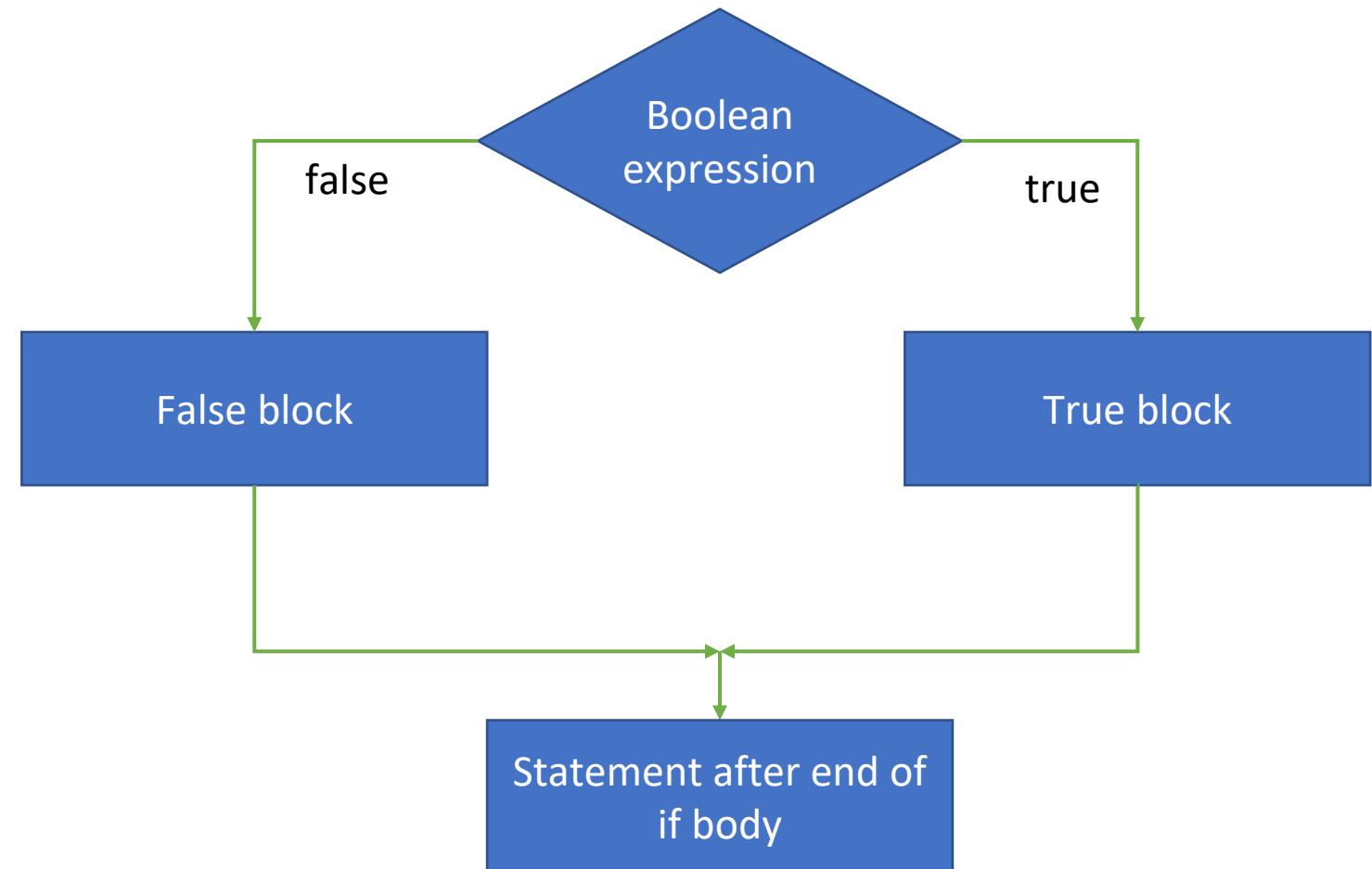
Relational operators

Be extra careful of the test for equality, `==`. As we have already pointed out, it is often valid to use assignment `=` where you might have meant `==` and C can't tell you about your mistake. The results are normally different and it takes a long time for beginners to get used to using `==` and `=`.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a = 0;
6     if (a = 3) ← Coding error!
7         printf("True\n");
8     else
9         printf("False\n");
10    return 0;
11 }
```

In “`if (a = 3)`” `a` will first get the value of 3 and then that will be “output” to `if`. Remember that any non-zero value will be counted as True. Avoid using “`=`” in `if` unless you know what you are doing. Our usual flags “`-std=c99 -O2 -Wall -Wextra -pedantic-errors`” will give message about this (but not error)

If-else statement



```
if (expression)
{
    [true block]
}
else
{
    [false block]
}
[Statements after if body]
```



Dangled if

User's
indentation
does not
change what
“if” does
“else” belong
to
=> Use {} or
at least
automatic
code
formatter

```
e_03 > c.h suggest explicit braces to avoid ambiguous 'else' [-Wdangling-else] GCC
(int)1
View Problem (Alt+F8) Quick Fix... (Ctrl+.)
Fix using Copilot (Ctrl+I)

int main()
{
    if (1)
        if (0)
            printf("Inner TRUE\n");
        else
            printf("Inner FALSE\n"); // <== THIS WILL PRINT

    if (1 > 0) // more clear with {}
    {
        if (0)
            printf("Inner TRUE\n");
        else
            printf("Inner FALSE\n"); // <== THIS WILL PRINT
    }

    if (0)
    {
        if (0)
            printf("Inner TRUE\n");
    }
    else
        printf("Outer FALSE\n"); // <== THIS WILL PRINT
```

```
lecture_03 > C hw_q1.c > main(void)
1 #include <stdio.h>
2 int main(void)
3 {
4     if (2 > 1)
5     {
6         if (0 > 1)
7             printf("True in inner if\n");
8             printf("We are not in inner if anymore\n");
9     }
10    else
11        printf("False outer if\n");
12    return 0;
13 }
```

⚠ hw_q1.c 1 of 1 problem

this 'if' clause does not guard its block [-Wmisleading-indentation] GCC

hw_q1.c(8, 13): ...this statement, but the latter is misleadingly indented as if it were guarded by the 'if'

Fix misleading indentation

If else structure can get non-intuitive

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int points = 75;
5     if (points > 90)
6         printf("Great job\n");
7     else
8         if (points > 80)
9             printf("Good job\n");
10    else
11        if (points > 70)
12            printf("Pretty good job!\n");
13    else
14        printf("...\n");
15
16 }
```

“else if” to the rescue!
(actually it is just indentation change... In C there is no elseif or elif [preprocessor has #elif]) [MATLAB has **elseif**]

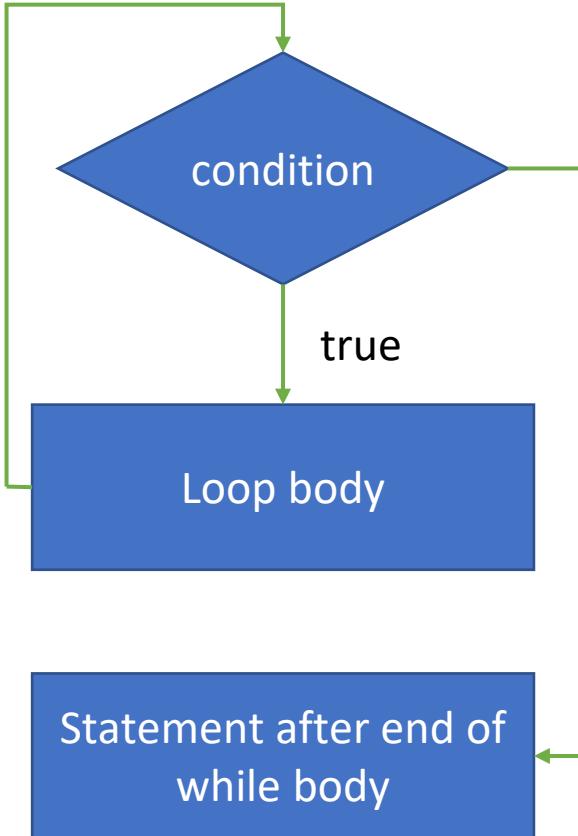
```
1 #include <stdio.h>
2 int main(void)
3 {
4     int points = 75;
5     if (points > 90)
6         printf("Great job\n");
7     else if (points > 80)
8         printf("Good job\n");
9     else if (points > 70)
10        printf("Pretty good job!\n");
11    else
12        printf("...\n");
13
14 }
```

Extra material: C++ if (C++17 and later)

- C++ if allows initializer within if statement
- The main purpose is scope limitation (the variable is limited to if else blocks)
[if we define “int ch;” earlier, in C we can use **if ((ch = getchar()) != EOF)**]

```
1 #include <print> // C++23 for std::print
2 #include <cstdio> // for getchar() and EOF
3
4 int main()
5 {
6     if (int ch = std::getchar(); ch != EOF)
7         std::print("You entered {} (its ASCII code is {})\n", static_cast<char>(ch), ch);
8         // without static_cast, the integer value of ch will be printed
9         // avoid C-style cast "(char) ch" in C++, use static_cast instead
10    else
11        std::print("End of file reached\n");
12    return 0;
13 }
```

While statement



```
while (condition)
{
    [ loop body ]
}
```

[Statements after **while** body]

The while Loop in C

Syntax: `while (condition) { ... }`

- Executes the loop body *as long as* the condition remains true.
- You have to handle initializing and updating separately.

Example:

```
int i = 0;
while (i <= 10) {
    printf("%d\n", i);
    i++;
}
```

Key Point: Use `while` when the number of iterations is *not* known in advance, or when you want to keep looping until some event occurs.

The while statement

```
1 #include <stdio.h>
2 int main()
3 {
4     int counter = 0;
5     while (counter < 10)
6     {
7         printf("Value of counter is %d\n", counter);
8         counter++;
9     }
10    printf("Now value of counter is %d\n", counter);
11 }
```

```
Value of counter is 0
Value of counter is 1
Value of counter is 2
Value of counter is 3
Value of counter is 4
Value of counter is 5
Value of counter is 6
Value of counter is 7
Value of counter is 8
Value of counter is 9
Now value of counter is 10
```

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int ch;                      Another example:
5         // read characters from the standard input
6         // first output of getchar is stored in ch
7         // the value of expression is the value of ch
8         // if the value is not 'a', then the loop will continue
9         // parenthesis around ch = getchar() is necessary
10    while ((ch = getchar()) != 'a' && ch != 'b')
11    {
12        putchar(ch);
13        printf("\nFirst while loop is done\n");
14        // any character after 'a' will still be in the input buffer!
15        // below code will read all characters in the input buffer until '\n' is read
16        printf("Start reading the rest of the characters in the input buffer\n");
17        while ((ch = getchar()) != '\n')
18        {
19            putchar(ch);
20        }
21    }
22 }
```

The do-while statement

It is occasionally desirable to guarantee at least one execution of the statement following the while, so an alternative form exists known as the do statement. It looks like this:

```
do  
    statement  
while (expression);
```

There is no do-while construct in MATLAB

and you should pay close attention to that semicolon—it is not optional! The effect is that the statement part is executed before the controlling expression is evaluated, so this guarantees at least one trip around the loop. It was an unfortunate decision to use the keyword `while` for both purposes, but it doesn't seem to cause too many problems in practice.

The for Loop in C

Syntax: `for (initialization; condition; update) {...}`

- **Initialization:** set a loop counter (e.g. `i = 0`).
- **Condition:** checked before each iteration (e.g. `i <= 10`).
- **Update:** runs after each loop body (e.g. `i++`).

Example:

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

Key Point: When you know exactly how many iterations you need, a `for` loop is usually clearer and more maintainable.

Comparing for vs. while

For Loop

- All loop-related elements (init, condition, update) in one line
- Great for counting-based loops
- Easy to see the iteration pattern at a glance

While Loop

- Condition only; init/update happen elsewhere
- Better when loop count is unknown or data-driven
- Can become infinite if condition never fails

Key Insight: *Both* loops can accomplish the same task; the difference is code clarity and the mental model you want to convey.

Example: Same Task, Two Approaches

While Loop:

```
int i = 0;  
while (i <= 10) {  
    printf("%d\n", i);  
    i++;  
}
```

For Loop:

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

- Both print the numbers from 0 to 10
- `for` keeps the loop logic in one place
- `while` can be simpler when you don't know end conditions upfront



Warning about for loops < vs <=

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int my_array[5] = {1, 2, 123, 4, 5};
5     for (int j = 0; j < 5; j++) // using j <= 5 would cause a buffer overflow
6         printf("my_array[%d] = %d\n", j, my_array[j]);
7     return 0;
8 }
```

Extra material: for comparison here is modern C++ based similar for loop that avoids < or <=

```
1 #include <vector>
2 #include <print> // C++23 feature (recommended over iostream / "cout")
3 int main()
4 {
5     std::vector<int> my_vector = {100,5,32}; // Dynamic-size array
6     for (auto &element : my_vector)
7         std::println("vector element = {}", element);
8     return 0;
9 }
```

Using fmt (not part of standard) to avoid any for loop:

```
#include <vector>
#include <fmt/ranges.h>
int main()
{
    std::vector<int> my_vec = {1, 100, 1000};
    fmt::println("my_vec = {}", my_vec);
    return 0;
}
// Output is
// my_vec = [1, 100, 1000]
```



Extra material, bounds checking with C++

```
1 #include <print> // C++23 for std::print, highly recommended!
2 #include <vector> // for std::vector
3 int main(void)
4 {
5     std::vector<int> my_vector = {1, 2, 123, 4, 5};
6     for (int j = 0; j < 5; j++) // my_vector.size() can be used instead of 5
7         std::println("my_vector[{}] = {}", j, my_vector[j]);
8
9     for (int j = 0; j < 115; j++) // out of bounds!
10        std::println("Again: my_vector[{}] = {}", j, my_vector.at(j));
11    // at() does bounds checking
12    // you can catch the exception thrown by at() if you want to
13    // and print a more user-friendly message
14    return 0;
15 }
```

Parallelizing Loops for Performance

- **Modern CPUs** are multi-core:
 - Single-threaded code may use only one core.
 - Wastes potential computing power on large problems.
- **Parallelize heavy loops:**
 - OpenMP (for C/C++) can distribute loop iterations across multiple cores.
 - C++ offers algorithms like `std::reduce` for parallel operations on collections.
 - Intrinsics (e.g., SIMD) can exploit vectorized instructions.
- **Why now?**
 - CPU clock speeds have plateaued; the focus is on more cores.
 - Parallel programming harnesses full CPU power.
- **Coming lectures:**
 - We will learn how to use OpenMP, vector intrinsics (briefly), to speed up loops.

for (initialize; check; update) statement

- We can also have loop counting in negative direction

```
1 #include <stdio.h>
2 int main()
3 {
4     for (int counter = 9; counter >= 0; counter--)
5         printf("Counter is %d\n", counter);
6 }
7
```

```
Counter is 9
Counter is 8
Counter is 7
Counter is 6
Counter is 5
Counter is 4
Counter is 3
Counter is 2
Counter is 1
Counter is 0
```

This will likely give error since counter was local variable to for loop!

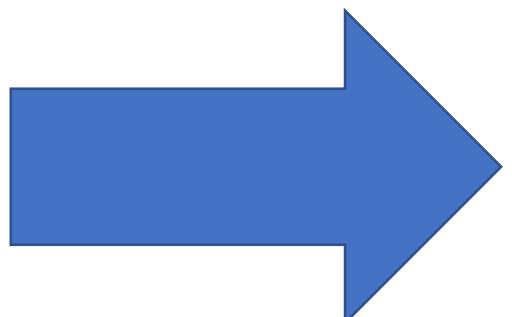
```
for (int counter = 9; counter >= 0; counter--)
    printf("Counter is %d\n", counter);
printf("Now counter is %d\n", counter);
```

For loop within for loop

```
#include <stdio.h>

int main()
{
    int i, j;
    for (i = 0; i < 2; i++)
    {
        printf("Outer for loop %d is starting\n", i);
        for (j = 0; j < 3; j++)
            printf("\t Inner loop %d is here\n", j);
    }

    return 0;
}
```



```
Outer for loop 0 is starting
    Inner loop 0 is here
    Inner loop 1 is here
    Inner loop 2 is here
Outer for loop 1 is starting
    Inner loop 0 is here
    Inner loop 1 is here
    Inner loop 2 is here
```

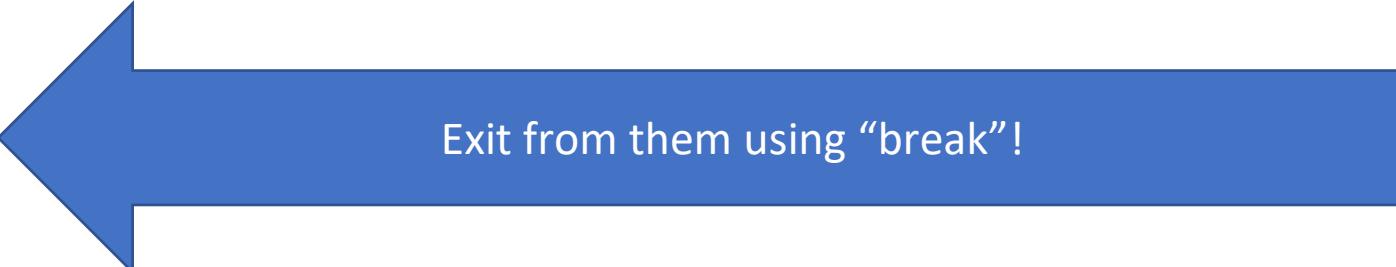
Infinite loops

Any of the initialize, check and update expressions in the for statement can be omitted, although the semicolons must stay. This can happen if the counter is already initialized, or gets updated in the body of the loop. If the check expression is omitted, it is assumed to result in a ‘true’ value and the loop never terminates. A common way of writing never-ending loops is either

```
for (;;) {
```

or

```
while (1) {
```



Exit from them using “break”!

and both can be seen in existing programs.

Warning about if and while

- Do not include “;” after if or while
- [unless you really want]

```
for (i=0; i<8; i++)  
    sum += i;
```

```
for (i=0; i<8; i++);  
    sum += i;
```

- Are doing totally different things
- In the second version, the for loop is run without running sum+= i
[it is being run only once after the for loop]

```
// Correct indentation is  
for (i=0; i<8; i++)  
    sum += i;
```

```
for (i=0; i<8; i++);  
sum += i;
```

Q: What is the value of sum?
(if it originally was 0)

Switch statement

This is not an essential part of C. You could do without it, but the language would have become significantly less expressive and pleasant to use.

It is used to select one of a number of alternative actions depending on the value of an expression, and nearly always makes use of another of the lesser statements: the `break`. It looks like this.

```
switch (expression) {  
    case const1:    statements  
    case const2:    statements  
    default:        statements  
}
```

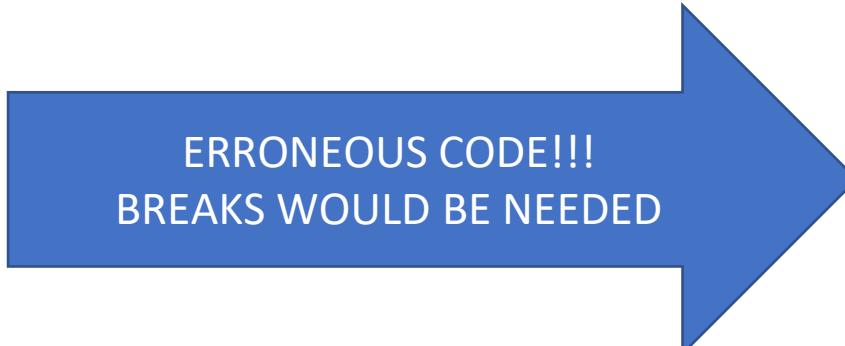
The expression is evaluated and its value is compared with all of the `const1` etc. expressions, which must all evaluate to different constant values (strictly they are **integral constant expressions**, see Chapter 6 and below). If any of them has the same value as the expression then the statement following the `case` label is selected for execution. If the `default` is present, it will be selected when there is no matching value found. If there is no `default` and no matching value, the entire `switch` statement will do nothing and execution will continue at the following statement.

Switch statement

- One curious feature is that the cases are not exclusive, as this example shows.

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

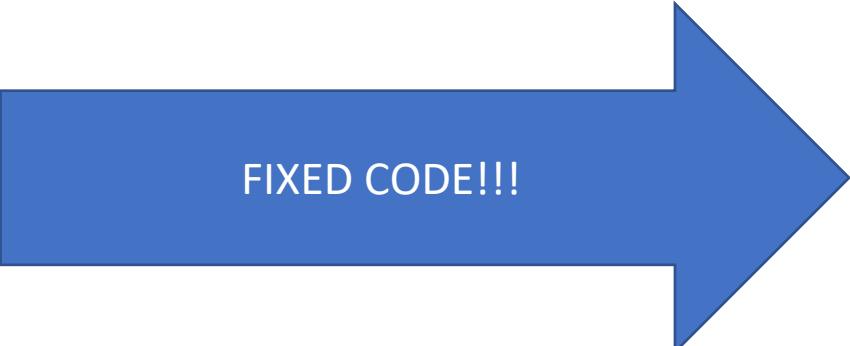
main(){
    int i;
    for(i = 0; i <= 10; i++){
        switch(i){
            case 1:
            case 2:
                printf("1 or 2\n");
            case 7:
                printf("7\n");
            default:
                printf("default\n");
        }
    }
    exit(EXIT_SUCCESS);
}
```



ERRONEOUS CODE!!!
BREAKS WOULD BE NEEDED

Example 3.5

Switch statement



FIXED CODE!!!

```
main() {  
    int i;  
    for (i = 0; i <= 10; i++) {  
        switch (i) {  
            case 1:  
            case 2:  
                printf("1 or 2\n");  
                break;  
            case 7:  
                printf("7\n");  
                break;  
            default:  
                printf("default\n");  
        }  
    }  
    exit(EXIT_SUCCESS);  
}
```

Switch statement

```
main.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i, ch;
6
7     while( (ch = getchar()) != '\n')
8     {
9         switch (ch)
10        {
11             case 'a':
12                 printf("You entered a\n");
13                 break;
14             case 'b':
15                 printf("You entered b\n");
16                 break;
17             default:
18                 printf("What you entered was not a or b\n");
19         }
20     }
21 }
```

This is also constant integer value!

Switch statement

```
switch (option)
{
    case 'y':
    case 'Y':
        printf("You chose YES.\n");
        break;
    case 'n':
    case 'N':
        printf("You chose NO.\n");
        break;
    default:
        printf("Invalid option.\n");
        break;
}
```

The break statement

This is a simple statement. Exits the **innermost** loop or switch. It only makes sense if it occurs in the body of a switch, do, while or for statement. When it is executed the control of flow jumps to the statement immediately following the body of the statement containing the break. Its use is widespread in switch statements, where it is more or less essential to get the control that most people want.

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

    for(i = 0; i < 10000; i++){
        if(getchar() == 's')
            break;
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

Example 3.7

It reads a single character from the program's input before printing the next in a sequence of numbers. If an 's' is typed, the break causes an exit from the loop.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int target = 3;
5     printf("Demonstrating break in nested loops:\n");
6     for (int i = 0; i < 5; i++)
7     {
8         printf("Outer loop iteration i = %d\n", i);
9         for (int j = 0; j < 5; j++)
10        {
11            printf("  Inner loop iteration %d\n", j);
12            if (j == target)
13            {
14                printf("  -> Breaks out of the inner loop when j == %d\n", target);
15                break; // Exits the innermost loop (the inner loop)
16            }
17        }
18        // At this point, inner loop has been exited if break was executed
19        printf("Back in the outer loop after break or normal completion of inner loop\n");
20    }
21    return 0;
22 }
```

continue statement (note that $1.0/0 = \text{NAN}$ usually but integer division with zero should be avoided (can even crash the program)!)

```
1 #include <stdio.h>
2 #include <math.h>
3 int main(void)
4 {
5     puts("Demonstrating continue in a loop:");
6     for (int i = 0; i < 5; i++)
7     {
8         if (i == 0)
9             continue; // Skip the rest of the loop body and go to the next iteration
10        printf("1/%d = %lf\n", i, 1.0 / i);
11    }
12    // Using ternary operator
13    // not exactly equivalent to continue above as this will print NaN for i = 0
14    puts("Demonstrating ternary operator:");
15    for (int i = 0; i < 5; i++)
16        printf("1/%d = %lf\n", i, i ? 1.0 / i : NAN);
17        // ? will select the first value if i is non-zero, otherwise the second value
18    return 0;
19 }
```

goto statement

```
main.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     goto skip_label;
6
7     printf("This will not be run!\n");
8
9 skip_label: printf("We skipped directly here\n");
10
11 }
```

goto cannot transfer control to any function except the one that is currently active
(longjmp is best to be avoided...)

Generally discouraged in structured programming

☺ E. Dijkstra, "Go To Statement Considered Harmful", 1968 ☺

More about logical expressions

- `&&` is the logical AND (not to be confused with bitwise and `&`)
- `||` is the logical OR (not to be confused with bitwise OR `|`)

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int age = 22;
6
7     if (age >= 18 && age <= 30)
8         printf("Age between 18 and 30\n");
9
10    if (age < 18 || age > 100)
11        printf("Age less than 18 or more than 100\n");
12    return 0;
13 }
```

Beware of & vs. && in C

- **Logical AND (&&)** checks whether both operands are nonzero (true).
 - It **short-circuits**: if the first operand is zero (false), it *won't* evaluate the second operand.
 - Example: `if (1 && 2)` evaluates to true because both 1 and 2 are nonzero.
- **Bitwise AND (&)** does a bit-by-bit operation.
 - It does *not* short-circuit; both operands are always evaluated.
 - Example: $1 \& 2$ in binary is $01 \& 10 = 00$, which is 0 (false).
 - So `if (1 & 2)` is false.
- **Pitfall:**
 - & instead of && can lead to wrong conditions or even infinite loops.
 - Always double-check boolean expressions for && vs. &.

More about logical expressions

Operator	Direction	Notes
() [] -> .	left to right	1
! ~ ++ -- - + (cast) * & sizeof	right to left	all unary
* / %	left to right	binary
+ -	left to right	binary
<< >>	left to right	binary
< <= > >=	left to right	binary
== !=	left to right	binary
&	left to right	binary
^	left to right	binary
	left to right	binary
&&	left to right	binary
	left to right	binary
? :	right to left	2
= += and all combined assignment	right to left	binary
,	left to right	binary

? operator

expression1?expression2:expression3

- If expression1 is true, then the result of the whole expression is expression2, otherwise it is expression3; depending on the value of expression1, only one of them will be evaluated when the result is calculated.
- The easiest case is when both expressions have arithmetic type (i.e. integral or real). The usual arithmetic conversions are applied to find a common type for both expressions and then that is the type of the result. For example
- `a>b?1:3.5`

contains a constant (1) of type int and another (3.5) of type double. Applying the arithmetic conversions gives a result of type double.

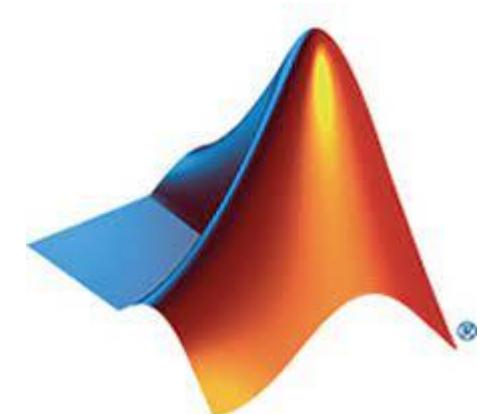
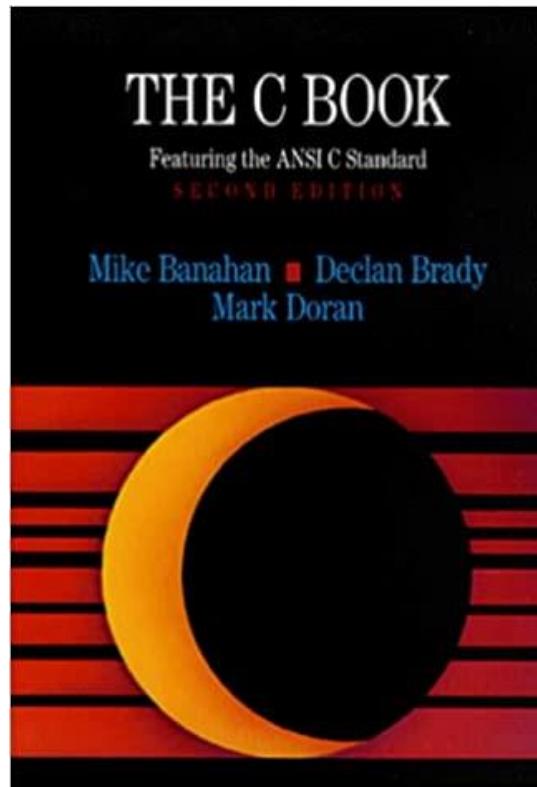
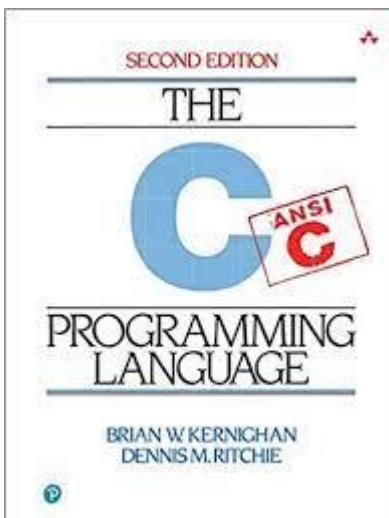
Can be nested (example of sign check):

```
int sign = (x > 0) ? 1 : ((x < 0) ? -1 : 0);
```

Can be written equivalently (but less clear):

```
int sign = x > 0 ? 1 : x < 0 ? -1 : 0;
```

Using C with MATLAB #04



https://publications.gbdirect.co.uk/c_book/copyright.html

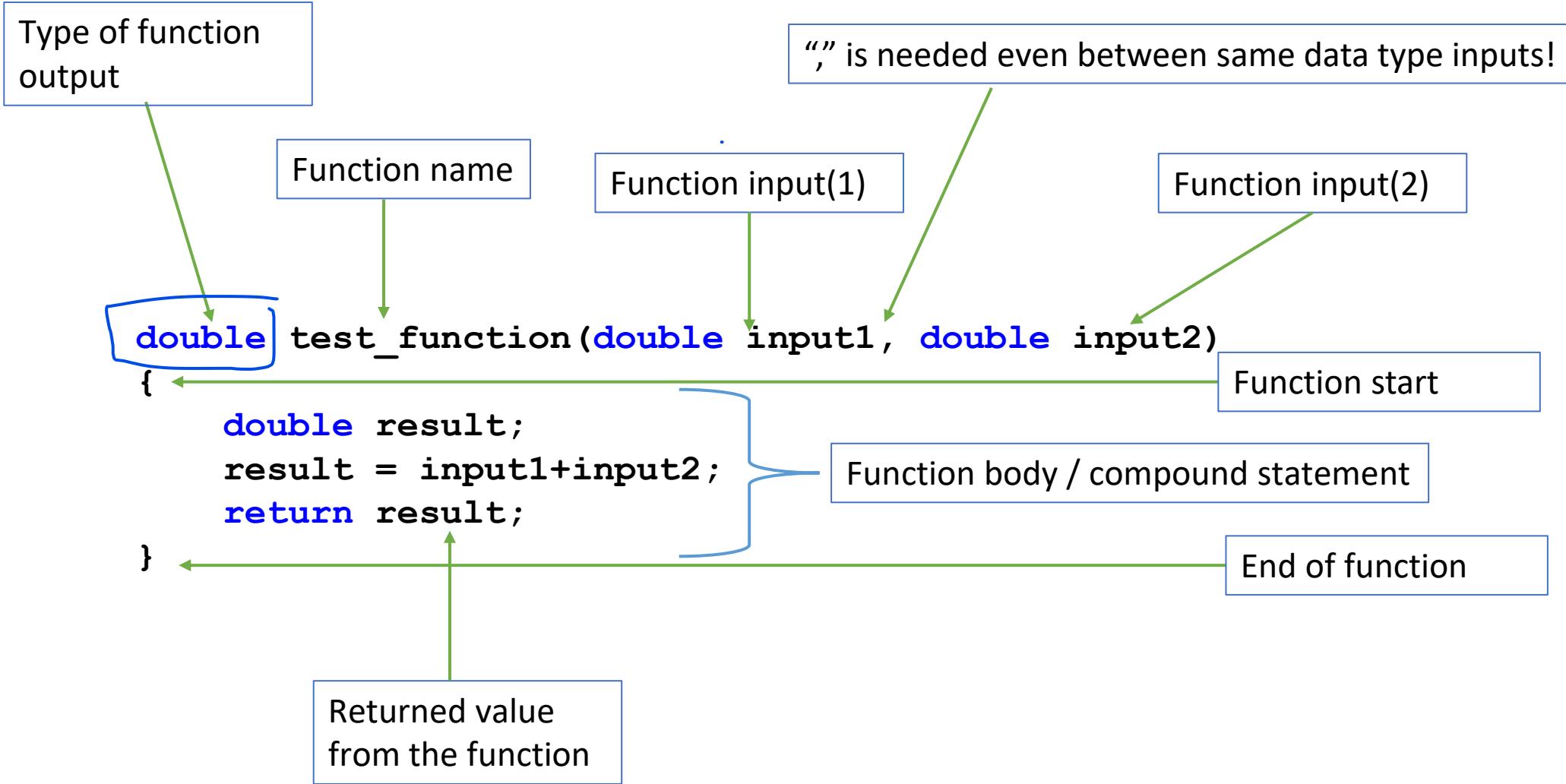
Keywords

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

Types of function

- C is Procedural programming language, meaning that it emphasizes splitting programs into functions (procedures) that contain steps of instructions to be carried out [C++ supports also Object-Oriented programming (OOP) with objects that encapsulate data and behavior]
- Using functions makes programs easier to maintain and understand
- Functions can be reused in other programs
- Program can be divided into smaller pieces that can be assigned to different programmers.
- *Standard library functions* such as printf are *built-in functions* that are declared in header files. For example, printf is declared in stdio.h header.
 - To use printf #include <stdio.h> [In C++ there is much broader C++ Standard Library but functions similar to C Standard Library (but often overloaded) are available for example with #include <cmath> etc.]
- *User-defined functions*

Function definition



Function Declaration/Prototype

- A function needs to be declared before using it
- If your function if before main() you can at the same time declare and define the function
- If your function defined after main, you need function prototype before main()
- Example of function declaration for function with name “aax1”:

```
double aax1(void);
```

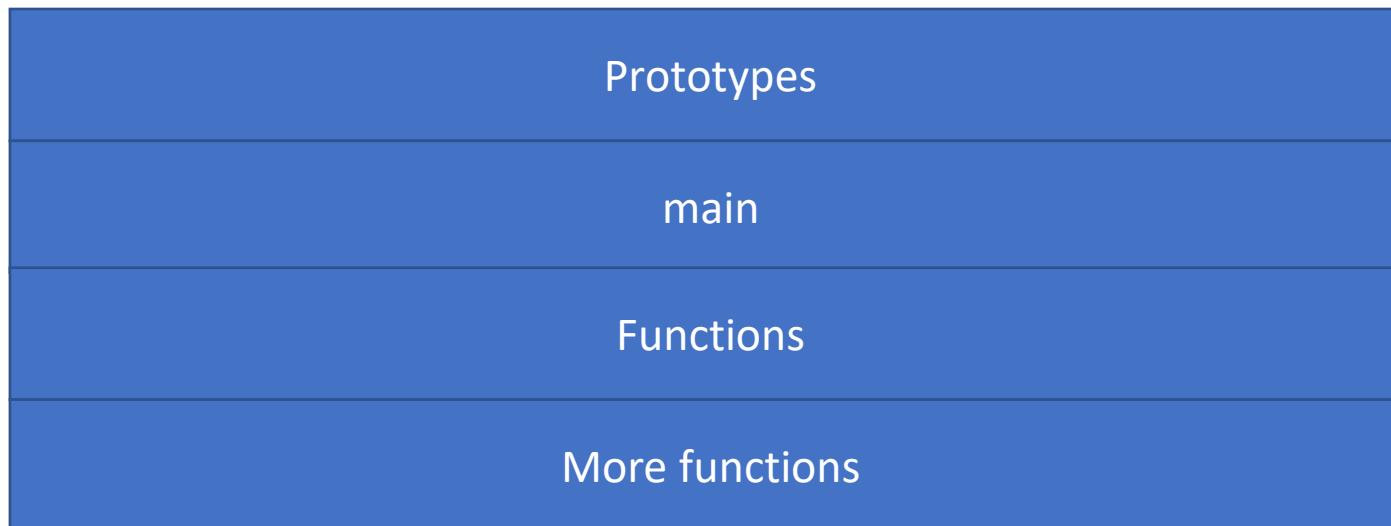
- Above void means that function takes no inputs [In **C++** avoid void for inputs but instead use **double aax1();**  **In C never use empty parenthesis!** - Double means that function returns a double

Function prototype declares the function name, parameters, and return type. The actual function is not contained.

Use prototype for all functions!

Function definitions before or after main

- For coding style, it is preferred that all functions are after main and include prototypes before main [for larger programs put functions in different .c file(s) and include prototypes in header “.h” files [in C++ use “.h” or “.hpp” depending on your preferences] [In C++20 **modules** can be used instead of header file-based programming]
 - Functions can be defined in any order
- If you include all function before main prototypes may not be needed
 - But you will get into problems if functions call each other!!!



- In any case, do not put main in the middle of functions!!!

Function Declaration/Prototype

- Why prototypes are so important?
- Once the compiler knows the types of a function's arguments, having seen them in a prototype, it's able to check that the use of the function conforms to the declaration.
- If we have function with prototype

```
double fsquare(double);
```

- When we call it with

```
double result = fsquare(2)*10;
```

- The compiler knows that fsquare takes double as input so integer “2” will be converted to double (2.0). Without conversion result would be totally wrong
- Note that “10” is integer but as return type of fsquare is double, multiplication is done in “double” domain in here “fsquare(2)*10”

Function Call

- Assume we have function with prototype

```
double fsquare(double);
```

- We can store the output of the function call into a variable
- ```
double result;
```

```
result = fsquare(3);
```

- We can also use the returned value without storing it

```
printf("%f\n", fsquare(3));
```

- We can also simply throw away the return value

```
fsquare(3);
```

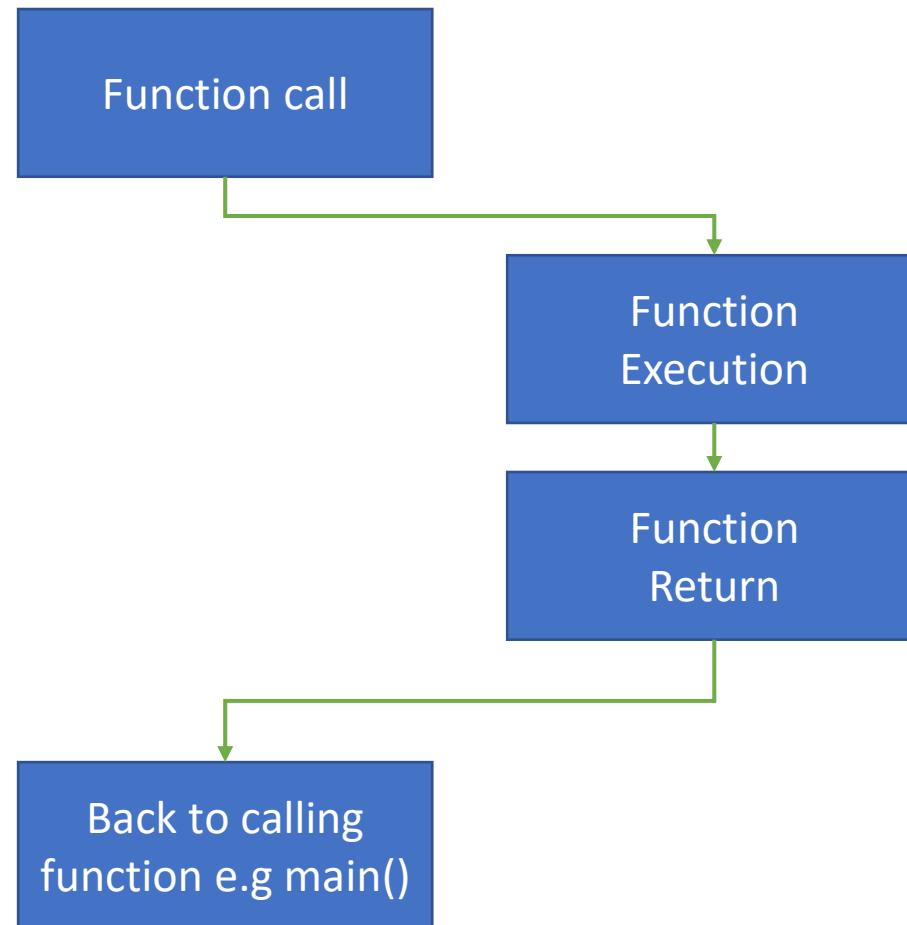
# Function prototypes

- If a function is called with arguments of the wrong type, the presence of a prototype means that the actual argument is converted to the type of the formal argument ‘as if by assignment’.
- For example, if prototype specified that input is double but function called with integer value, then the integer value is promoted to double
- Without prototype compiler would not know to do this!

# Function prototypes

```
1 #include <stdio.h>
2 double fsquare(double);
3
4 int main()
5 {
6 int i = 10;
7 printf("%f\n", fsquare(i));
8 return 0;
9 }
10
11 double fsquare(double in)
12 {
13 double result;
14 result = in*in;
15 return result;
16 }
```

Prototype specified input as double but “i” is integer!  
⇒ Integer will be converted to double  
⇒ Calculations are correct



# Function return types

- Functions can return any type supported by C (except for arrays and functions), including the pointers, structures and unions which are described later.
- For the types that can't be returned from functions, the restrictions can often be sidestepped by using pointers instead.
- All functions can be called recursively.
- Void means that function does not return anything
- Functions returning void cannot be used to anywhere where a value would be needed. Also, Cannot be used to assign a value to a variable (no value is being returned!)

A C function cannot return array [C++ function can return std::vector for example!]

# Function types

- void checkPrime(void); [In C++ use “void checkPrime();”]
  - Takes no input and has no output
  - Call with checkPrime();
- int test(void); [In C++ use “int test();”]
  - Takes no input but returns integer as output
  - Call with out = test(); //where out is typically integer (if not arithmetic conversions will be done such as int to double)
- void test2(int);
  - Takes input but has no return value
  - Call with test2(in); //where in is typically integer
- int add\_two\_numbers(int,int);
  - Takes two inputs and returns integer as output
  - Call with out = add\_two\_numbers(in1,int2); // where out, in1,in2 are typically all integers

# C++ Function Overloading and Automatic Templates

C++ not required in mandatory homework (may be present in optional questions)

It is good to know about C++ features so you can choose between C and C++, depending on your application requirements.

**Goal:** Demonstrate how we can write multiple functions with the *same name* but different parameter types (called *function overloading*), and then introduce *auto* in C++ to create a more flexible function.

- We'll see two overloaded functions: one for double inputs, another for int inputs.
- We'll then see a function that uses **auto** for *type deduction*, allowing automatic detection of parameter types.
- Finally, we'll see how they are called in `main()` to produce different outputs.

## Key Concepts:

- Function Overloading
- Automatic Type Deduction (**auto**)
- Return Type Inference

# The Program's Code (1/2)

## Code Explanation:

```
// 1) Overloaded function for double
double add_two_inputs(double in1, double in2) {
 std::cout << "Double version called!" << '\n';
 return in1 + in2;}
```

```
// 2) Overloaded function for int
int add_two_inputs(int in1, int in2) {
 std::cout << "Int version called!" << '\n';
 return in1 + in2;}
```

## Points to Note:

- `add_two_inputs` appears twice, but first accepts `double`, the other accepts `int`.
- Depending on the argument types passed in, C++ chooses which function to call.
- This is called *function overloading*.

## The Program's Code (2/2)

### Continuing:

```
// 3) Function that uses 'auto'
auto add_two_input_auto(auto in1, auto in2) {
 std::cout << "Automatic version being called!" << '\n';
 return in1 + in2; }

.

int main() {
 std::cout << add_two_inputs(10.5, 20.3) << '\n';
 std::cout << add_two_inputs(10, 20) << '\n';
 std::cout << add_two_input_auto(10.5, 20.3) << '\n';
 std::cout << add_two_input_auto(10, 20) << '\n';
}
```

### Key Details:

- `auto add_two_input_auto(auto in1, auto in2)` uses a C++20 feature called **abbreviated function templates**.
- It automatically infers the parameter types (and return type!).

# Why Overloading and auto?

## Function Overloading

- Lets us use the same function name for different parameter types.
- Eases code readability: `add_two_inputs` is always the name, but the compiler knows which version to call.
- Example: double version vs. int version.

## auto in Functions

- Introduced in C++20: abbreviated function templates.
- The compiler deduces the types of `in1` and `in2` at compile time.
- Helps write generic code without explicitly creating multiple overloads.

**Takeaway:** Overloads are great when you know the types you want to support. `auto` is useful for more generic or *type-agnostic* code.

# Running the Program

## What's Happening?

- First call: add\_two\_inputs(10.5, 20.3) matches the *double version*, prints "Double version called!" and sums as double.
- Second call: add\_two\_inputs(10, 20) matches the *int version*, prints "Int version called!".
- Third call: add\_two\_input\_auto(10.5, 20.3) deduces both parameters as double.
- Fourth call: add\_two\_input\_auto(10, 20) deduces both as int.

# A Similar C Example (No Overloading)

```
#include <stdio.h>

double add_two_inputs(double in1, double in2) {
 puts("add_two_inputs called!");
 return in1 + in2;

int main(void) {
 printf("%f\n", add_two_inputs(10.5, 20.3));
 printf("%f\n", add_two_inputs(10, 20));
 return 0;
}
```

## Key Points:

(we focus on C99, C11 does have Generic selection which is similar as overloading)

- C doesn't support function overloading. Here, `add_two_inputs` always expects double parameters.
- Passing 10 or 20 implicitly converts them to double for the call.
- Return type is always double, so `printf` with `%f` is valid.

# Call by value

- The **call by value** is the default in C programming. It means that passing parameters to a function copies the actual value of each argument into formal parameters of the function. Any possible changes made to the parameter inside the function have no effect on the argument.
- This means that function cannot modify parameters of the function that called it!
- The way to communication to outside world is to use **return** statement to return a value
  - It is possible to write functions that take *pointers* as their arguments, giving a form of **call by reference**. This is described in [Chapter 5 of the course book](#) and **does** allow functions to change values in their callers. [In C++ "references" are preferred to pointers]

# Call by value – Failed attempt at swap function

```
1 #include <stdio.h>
2 void swap(int, int);
3 int main(void)
4 {
5 int a = 10, b = 20;
6 swap(a, b);
7 printf("Value of a = %d, value of b = %d\n", a, b);
8 }
9 void swap(int a, int b)
10 {
11 int temp = b;
12 b = a;
13 a = temp;
14 }
```

These a and b are NOT same as  
a and b in main()  
These are local variable to function  
**Their names do not matter**  
Modifying them does NOT change  
the a and b in main()!



Value of a = 10, value of b = 20

# C++: Implementing a Simple and Working swap Function

## Function Declaration:

```
void swap_ints(int &a, int &b)
{
 int temp = a;
 a = b;
 b = temp;
}
```

*(Handwritten annotations: circled 'a' and 'b' in the parameter list; circled 'a' and 'b' in the assignment statements; circled 'temp' in the assignment statement; circled 'a' and 'b' in the final assignment statement; a large bracket underlines the entire function body with the label 'a = b' written below it.)*

## Usage Example:

```
int main()
{
 int x = 10, y = 20;
 swap_ints(x, y);
 // Now x == 20
 // And y == 10
 return 0;
}
```

## Key Points:

- References (the & symbol) allow direct access to original variables.
- No extra pointers or returning needed; changes affect the original arguments.

**References make swapping simple by avoiding pointer syntax**

# Return statement

- The return statement is very important. Every function except those returning void should have at least one, each return showing what value is supposed to be returned at that point.
- If function returns for example integer (int) we can use, in the function body, the command return 10; //10 is just an example, to return from that line to the calling function
- There can be multiple returns in a function (obviously maximum one of them will be executed if any)

# Return statement

- Reaching end of the function (without any return hitting before it), is equivalent to return statement without an expression (“return”).
  - This is an error if the function is supposed to return something (its return type is not “void”) [this is only valid in C99 and later for **main** returning int, also valid in C++]
- Function return type will be casted to the specified type (if possible)
  - For example, if you return int (and function is supposed to return double), int will be automatically casted to double

# Example function

```
8
9 #include <stdio.h>
10
11 int add_two_numbers(int, int); ← Function prototype: takes two integers and returns an integer
12
13 int main()
14 {
15 int number1 = 10;
16 int number2 = 20;
17 int sum; ←
18 sum = add_two_numbers(number1, number2); ←
19 printf("The sum of %d and %d is %d\n", number1, number2, sum);
20 return 0; |
21 }
22
23 int add_two_numbers(int a, int b) ←
24 {
25 return a+b; ←
26 }
```

# Example function

main.c

```
1 #include <stdio.h>
2 #define MAX_NUMBER 100
3
4 int is_prime(int);
5
6 int main(void)
7 {
8 int this_number;
9 this_number = 3;
10
11 printf("2 is a prime number\n");
12
13 while (this_number <= MAX_NUMBER)
14 {
15 if (is_prime(this_number))
16 {
17 printf("%d is a prime number\n", this_number);
18 }
19 this_number = this_number + 1;
20 }
21 return 0;
22 }
23
24 int is_prime(int input_number)
25 {
26 int i;
27 i = 2;
28 while (i < input_number)
29 {
30 if (input_number % i == 0)
31 return 0;
32 i = i + 1;
33 }
34 return 1;
35 }
```

The image shows a screenshot of a code editor with a dark theme. A file named 'main.c' is open. The code is a C program that prints prime numbers up to 100. It includes a helper function 'is\_prime' that checks if a number is divisible by any integer from 2 to itself minus one. Hand-drawn blue annotations highlight specific parts of the code: a circle around the 'if' statement in line 15, a large bracket grouping the entire 'is\_prime' function, another bracket grouping the main loop's body, and a line connecting the 'return 0;' in line 31 to the 'return 1;' in line 34.

# Integer/float promotion in printf

- For Variadic functions [avoid writing your own variadic functions] (that can take varying number of input arguments), char and short are promoted to int and float is promoted to double
  - This why format specific %f can be used for both float and double in printf (since float will anyways be promoted to double)

⚠ For scanf we need to use proper format specifiers (even for char, short, float, etc.)

# Visibility (Scope) – local variables

- Local variables are visible only inside the block they are defined in. They have default automatic (auto) storage type [they will have allocated memory only during the block they are defined in]. **In C++ keyword “auto” has a new function, it means that datatype will be automatically inferred.**

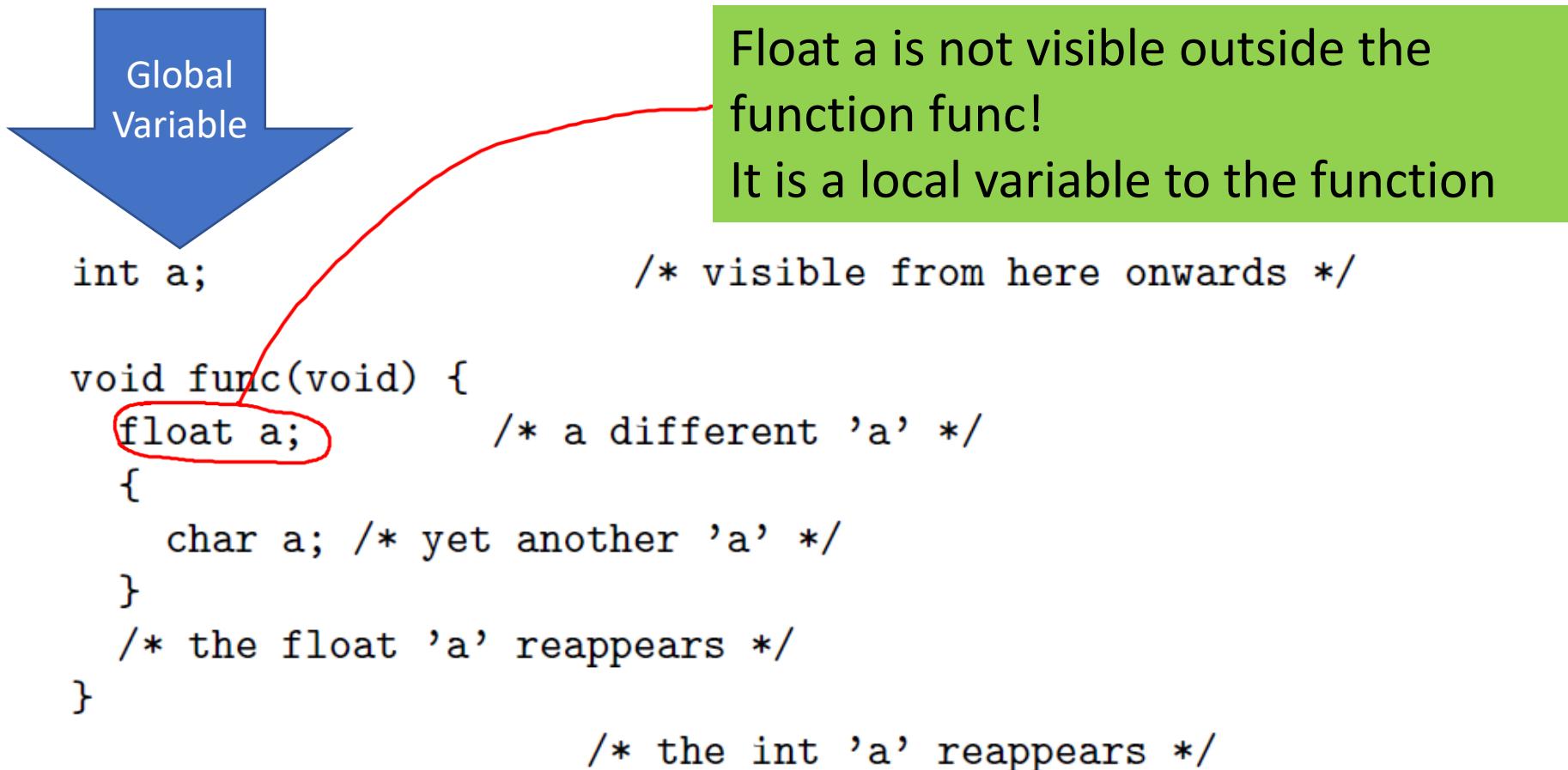
```
9 #include <stdio.h>
10
11 int main()
12 {
13 for (int i = 0; i < 10; i++)
14 {
15 printf("i is %d\n", i);
16 }
17
18 printf("i is %d\n", i); //INVALID! i is not visible anymore
19
20 return 0;
21 }
```

```
root [0] auto x = 123
(int) 123
root [1] auto y = 123.7
(double) 123.70000
```

# Visibility

- Variable declared with a block (code enclosed between curly brackets { and }) is only visible at that block (local scope)
  - This also applies to functions
- Blocks can even be nested
  - In this case, inner block definitions will override outer block definitions until the end of the inner block
  - After that the outer block definition will be again valid
- Global variables (defined outside functions) will be available when there are no other definitions using same variable name within the current function

# Visibility – local variables



# Visibility

```
1 #include <stdio.h>
2 int a = 777;
3 void print_a(void);
4 int main(void)
5 {
6 int a = 10;
7 printf("Value of a = %d\n", a);
8 {
9 int a = 20;
10 printf("Value of a in inner block is %d\n", a);
11 }
12 printf("Now value of a is again %d\n", a);
13
14 print_a();
15 }
16
17 void print_a(void)
18 {
19 printf("In function print_a value of a %d\n", a);
20 }
```

Start of the outer block

Start of a new block

End of the block

End of the outer block

Terminal output:

Value of a = 10

Value of a in inner block is 20

Now value of a is again 10

In function print\_a value of a 777

# Local Variables & Function Returns in C and C++

**Key Idea:** Local (automatic) variables only exist while their function is active.

- Once the function ends, these variables are destroyed.
- Recursion also benefits from this: each new call creates a fresh set of local variables.

**Implication:**

- Returning a pointer to a local array or variable is dangerous because that memory may be overwritten after the function returns.

# Returning Local Variables & Arrays in C

## Why Not Return an Array Directly?

- In C: returning a single int or double is fine (it's a copy).
- C does not allow returning a built-in array (it will not "copy"). The local array's memory ceases to exist when the function ends.

## Returning By Value Is Safe:

```
int getNumber() {
 int num = 42;
 return num;
}

// The original num is gone after this function
// But this is safe, since we return a copy of num
```

## But Not:

```
int* getArray() {
 int arr[5] = {1,2,3,4,5};
 return arr; // DANGER: arr is destroyed on function exit
}
```

## C++: Returning std::vector

**C++ Advantage:** Containers such as `std::vector` can be returned by value efficiently especially with Move Semantics and Copy Elision.

```
#include <vector>
#include <iostream>

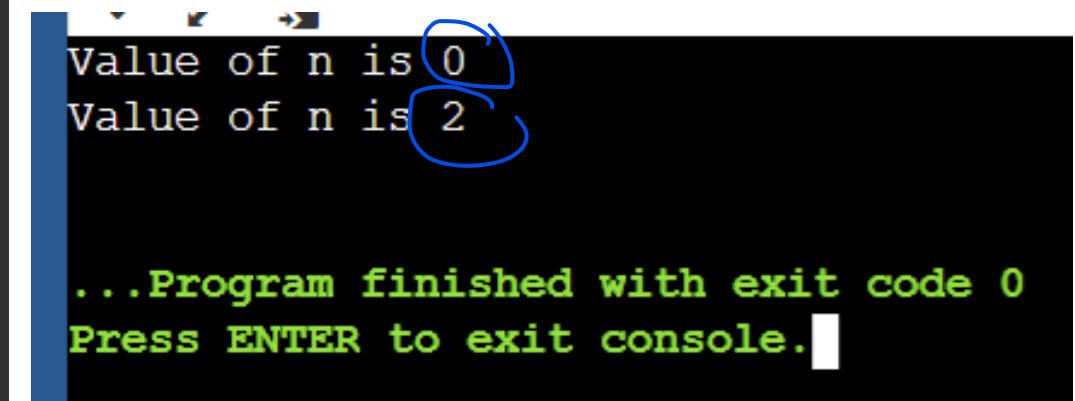
std::vector<int> createVector() {
 std::vector<int> vec = {1, 2, 3, 4, 5};
 return vec; // Safe
}

int main() {
 std::vector<int> myVec = createVector();
 for (auto num : myVec) {
 std::cout << num << " ";
 }
}
```

# Visibility – global variables

- Global variables are accessible from any function in the program
  - Avoid global variables as much as possible (constant values like PI are OK)

```
1 #include <stdio.h>
2 void increment_n(void);
3 int n; // Global variable (initialized to zero)
4
5 int main()
6 {
7 printf("Value of n is %d\n",n);
8 n = n + 1;
9 increment_n();
10 printf("Value of n is %d\n", n);
11 return 0;
12 }
13
14 void increment_n(void)
15 {
16 n = n + 1;
17 }
```



```
Value of n is 0
Value of n is 2
...Program finished with exit code 0
Press ENTER to exit console.
```

# Recursion

Every function in C may be called from any other or itself. Each invocation of a function causes a new allocation of the variables declared inside it. In fact, the declarations that we have been using until now have had something missing: the keyword `auto`, meaning ‘automatically allocated’.

```
/* Example of auto */
main() {
 auto int var_name;
 .
 .
 .
}
```

The storage for `auto` variables is automatically allocated and freed on function entry and return. If two functions both declare large automatic arrays, the program will only have to find room for both arrays if both functions are active at the same time. Although `auto` is a keyword, it is never used in practice because it’s the default for internal declarations and is invalid for external ones. If an explicit initial value (see ‘initialization’) isn’t given for an automatic variable, then its value will be unknown when it is declared. In that state, any use of its value will cause undefined behaviour.

# Recursion in C (Concepts)

## What is Recursion?

- A function that calls itself.
- Each call solves a smaller portion of the problem.
- Continues until a base case stops the recursion.

## Why Use It?

- Natural way to express solutions to certain problems (e.g. factorial, Fibonacci).
- Can be simpler to read than iterative approaches in some cases.

## Key Idea:

- **Base Case:** Simple condition where we no longer recurse.
- **Recursive Case:** The function calls itself with smaller input.

# Example: Factorial Function in C

## Factorial Definition:

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases}$$

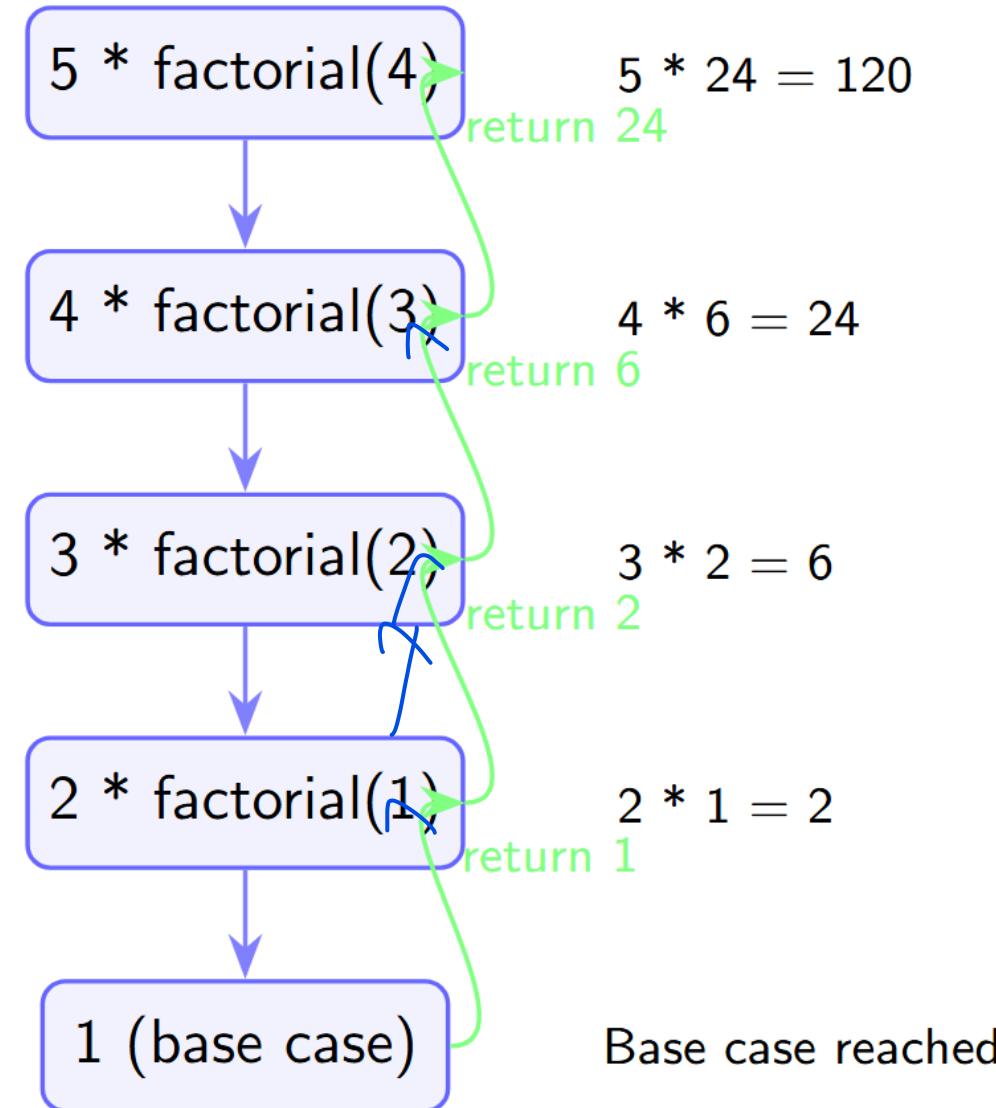
```
int factorial(int n) {
 if (n <= 1) {
 return 1; // Base case
 } else {
 return n * factorial(n - 1); // Recursive call
 }
}
```

## What's Happening?

- `factorial(5)` calls `factorial(4)`, which calls `factorial(3)`, etc.
- When `n` is 1 or less, we stop (base case = 1).
- All pending multiplications then complete as the function “unwinds.”

# Recursive Factorial Visualization

factorial(5)



# Static variables

You are also allowed to declare internal objects as `static`. Internal variables with this attribute have some interesting properties: they are initialized to zero when the program starts, they retain their value between entry to and exit from the statement containing their declaration and there is only one copy of each one, which is shared between all recursive calls of the function containing it.

Internal statics can be used for a number of things. One is to count the number of times that a function has been called; unlike ordinary internal variables whose value is lost after leaving their function, statics are convenient for this. Here's a function that always returns a number between 0 and 15, but remembers how often it was called.

```
int
small_val (void) {
 static unsigned count;
 count++;
 return (count % 16);
}
```

# Static variables

```
1 #include <stdio.h>
2 void testi(void);
3 int main(void)
4 {
5 testi();
6 testi();
7 testi();
8 return 0;
9 }
10
11 void testi(void)
12 {
13 static int count = 10; ← Only initialized once!
14 count++;
15 printf("Count is %d\n", count);
16 }
17 }
```

Terminal output:

Count is 11  
Count is 12  
Count is 13

# Program and file scope

Static means different thing at block scope  
and variables/functions outside blocks

```
1 #include <stdio.h>
2 void testi(void);
3 static int file_scope_variable = 123;
4 // above variable only visible in this file!
5 int global_scope_variable = 234;
6 // above variable visible also in other files!
7 int main(void)
8 {
9 testi();
10 }
```

Would be error to  
try to access  
file\_scope\_variable  
here!

Different file:

```
1 #include <stdio.h>
2 extern int global_scope_variable;
3 void testi(void)
4 {
5 printf("The value of global_scope_variable is %d\n",global_scope_variable);
6 }
```

We need this, otherwise variable is not found

# Introducing Lambda Functions in C++

**More Advanced & Totally Optional (will cover in lecture if enough time)**  
**Lambdas has many further details. This is just “taste” of lambdas**

## What is a Lambda?

- A *lambda function* (or *lambda expression*) is a small, anonymous function you can define *inline* in your code.
- Great for short operations passed to algorithms.
- Syntax roughly: [capture] (params) { body }.

## Why Use Them?

- Keeps code *local* and *concise*.
- Avoids making lots of separate named functions for small tasks.
- Perfect for “one-off” operations.

## Example: Simple Lambda Adding a Constant

### Code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
 double c = 0.25;
 std::vector<double> my_vec{1, 2, 10.5, 1000.25};

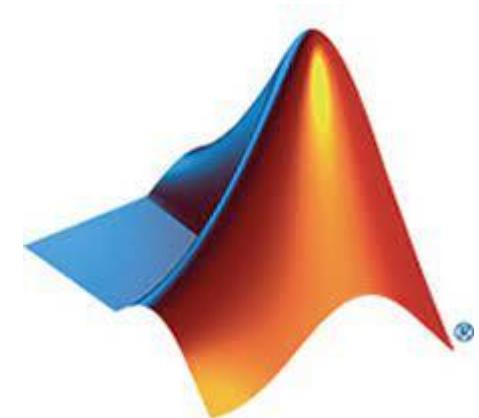
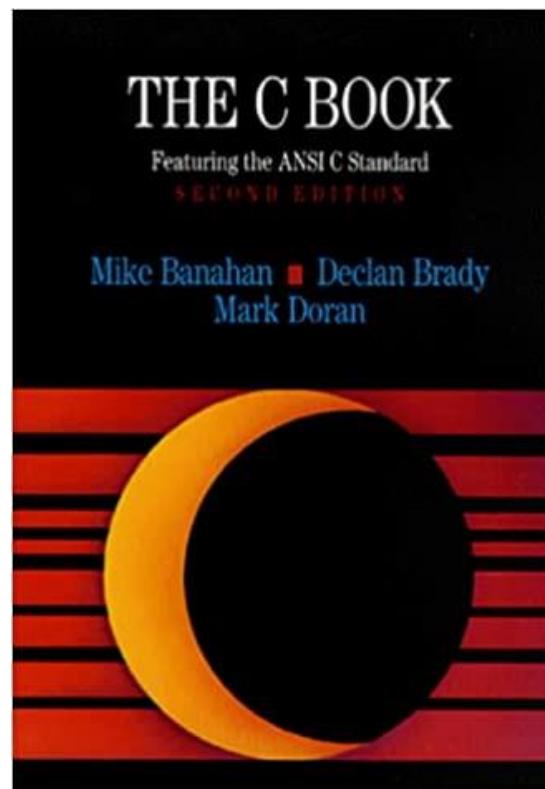
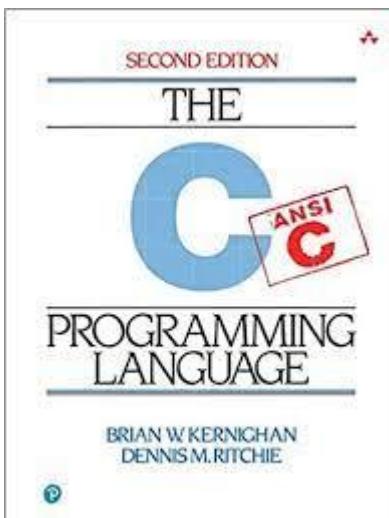
 std::ranges::for_each(my_vec,
 [c] (auto &n) {
 n += c; // Add c to each element
 }
);
}
```

# Example: Simple Lambda Adding a Constant

## What Happens?

- [c]: *Capture* the variable c by value. Capturing lets the lambda *use* variables from the surrounding scope.
- (auto &n): The lambda parameter is a reference to each element.
- n += c;: Adds c to that element.
- Result: Each element in my\_vec is increased by 0.25.
- This modifies my\_vec *directly*.

# Using C with MATLAB #05



[https://publications.gbdirect.co.uk/c\\_book/copyright.html](https://publications.gbdirect.co.uk/c_book/copyright.html)

# Arrays

- You define arrays this way:

```
datatype array_name[array_size]; // ! Array size
CANNOT be changed afterwards! (C++ std::vector size
can be changed)
```

- For example

```
double ar[100];
```

- defines array with name “ar” with 100 elements each of which is a double
- First element of the “ar” is ar[0] and last element is ar[99] (totally 100 elements)



Diagram showing an array consisting of elements labelled 'ar[0]', 'ar[1]', etc., up to 'ar[99]'.



Array indexing in C starts from zero (not one like in MATLAB)

# Arrays

- We change array values by indexing them with square brackets
- For example,

```
double ar[100]; // So far values of ar elements are
“random” (not guaranteed to be zero)
```

```
ar[0] = 1.23; //changes the value of the 1st element
```

```
ar[1] = 3.45; // changes 2nd element value
```

- We can initialize all elements of ar to be zero by declaring it with

```
double ar[100] = {0}; // just one zero is enough
```

# Arrays

```
root [6] int test_arr[] = {1,2,3,4,100};
root [7] sizeof(test_arr)/sizeof(test_arr[0])
(unsigned long) 5
```



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

- will initialize all elements in the array

You can let compiler calculate the size of the array

```
int arr_int[5] = {1,2};
```

- will initialize the first two elements (rest will be set to be zero)

It is enough to initialize just one element (rest will be set to zero)

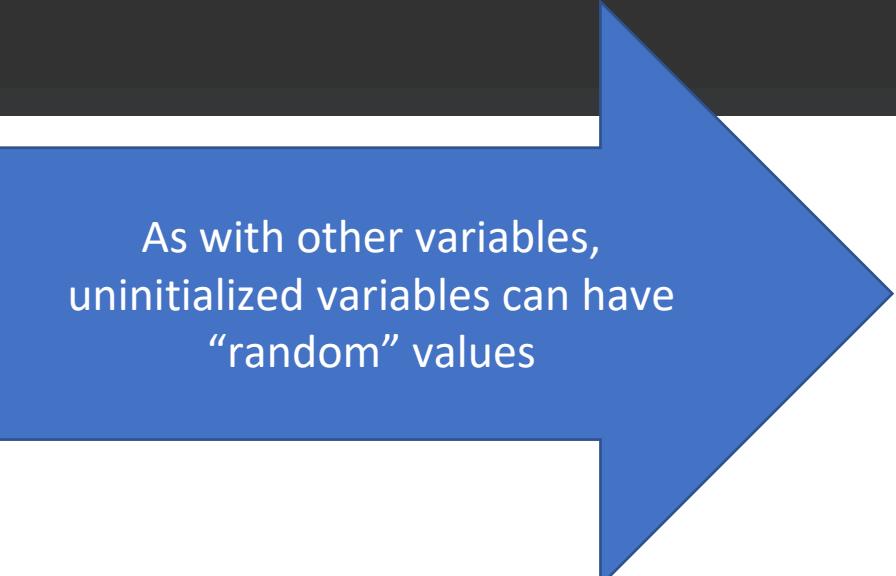
- Notice that after definition of variable you cannot use curly brackets

```
root [5] int arr_int[5];
root [6] arr_int = {1,2,3,4,5};
ROOT_prompt_6:1:9: error: array type 'int [5]' is not assignable
```

You cannot assign {2,3,4} (for example) to array [after it has been initialized]

# Arrays

```
1 #include <stdio.h>
2
3 int main()
4 {
5 int int_arr[10];
6 for (int i = 0; i < 10; i++)
7 printf("Value of arr[%d] = %d\n", i, int_arr[i]);
8 return 0;
9 }
10
11 |
```



As with other variables,  
uninitialized variables can have  
“random” values

Uninitialized values can be random!

Value of arr[0] = -1984856120  
Value of arr[1] = 32736  
Value of arr[2] = -472514080  
Value of arr[3] = 22030  
Value of arr[4] = 0  
Value of arr[5] = 0  
Value of arr[6] = -472514432  
Value of arr[7] = 22030  
Value of arr[8] = 991944400  
Value of arr[9] = 32764

# ⚠️ Array size ⚠️

- The number of elements in array can be found with `sizeof(array_name)/sizeof(array_name[0])`
- ⚠️ This only works for C-style true arrays (not C-“array” passed to functions as those are actually pointers) ⚠️ **Program will compile (maybe with a warning) but will give WRONG results**

```
1 #include <stdio.h>
2 int main()
3 {
4 double test[5] = {1,2,3,4,5.5};
5 for (int i = 0; i < sizeof(test)/sizeof(test[0]); i++)
6 printf("test[%d]=%f\n", i, test[i]);
7 return 0;
8 }
```



This code is OK

# Array size C++

- ! In C++, `sizeof` should be avoided. Instead use for example:
  - `std::size` (will work for true arrays but also with `std::vector`. Very nice that it will **FAIL** for pointers!) for `std::vector` you can also use directly `.size` method (for example `my_vec.size()`) `std::size` will give size directly in elements (not bytes)

```
#include <iostream>
void passing_C_array(double *a)
{
 // Will print 1 (no matter how many elements a has)
 // because a is actually a pointer (not true array)
 std::cout << "Numel attempt by sizeof in function: " << sizeof(a)/sizeof(a[0]) << std::endl;
 // std::cout << std::size(a) << std::endl; // Will not compile (means safer than sizeof)
}
int main()
{
 double a[7] = {1, 2, 3, 4, 5, 6, 7123};
 std::cout << "Numel by old method: " << sizeof(a)/sizeof(a[0]) << std::endl;
 std::cout << "Numel by std::size: " << std::size(a) << std::endl;
 passing_C_array(a);
 return 0;
}
```

Numel by old method: 7  
Numel by std::size: 7  
Numel attempt by sizeof in function: 1



⚠ Arrays cannot be compared (e.g. with ==, >, <)

```
root [27] int a[3]={1,2,3};
root [28] int b[3]={1,2,3};
root [29] (a==b)
ROOT_prompt_29:1:3: warning: array comparison always evaluates to false
```



⚠ You cannot assign to array

```
root [36] int a[3]={1,2,3};
root [37] int b[3]={1,2,3};
root [38] a=b
ROOT_prompt_38:1:2: error: array type 'int [3]' is not assignable
```



⚠ C-style arrays cannot be returned ⚠



## std::vectors can be compared C++

```
root [31] vector<int> eka{1,100,100};
root [32] vector<int> toka{1,100,100};
root [33] vector<int> kolmas{1,100,777};
root [34] auto test1 = (eka==toka)
(bool) true
root [35] auto test2 = (eka==kolmas)
(bool) false
```

## You can assign to std::vector C++

```
root [44] vector<int> eka{1,2,100};
root [45] vector<int> toka;
root [46] vector<int> kolmas{1,2,777};
root [47] toka = eka;
root [48] kolmas = eka;
```

std::vector can be passed to function  
(preferred as a reference) and also can be  
returned C++

```
1 #include <iostream>
2 #include <vector>
3 void modify_vector(std::vector<double>& v)
4 {
5 for (auto &e: v)
6 e += 2; // add 2 to each element
7 v.push_back(42); // add 42 to the end
8 }
9
10 int main()
11 {
12 std::vector<double> zz{1, 3, 5};
13 std::cout << "Before modification: ";
14 for (auto &e: zz)
15 std::cout << e << " ";
16 modify_vector(zz);
17 std::cout << "\nAfter modification: ";
18 for (auto &e: zz)
19 std::cout << e << " ";
20 return 0;
21 }
```

We can even add elements!  
For vectors with more complex  
datatypes, use v.emplace\_back(..)  
to call constructor directly

std::vector can be  
passed to a function  
(preferred as a  
reference to avoid  
overhead and to allow  
modification, use  
**“const”** reference if no  
need to modify it) C++

Before modification: 1 3 5  
After modification: 3 5 7 42

# Pointers

- Each variable is stored in memory in some address
- The address of operator (&) will give you the address of a variable [This is not C++ reference which also uses “&” in a different way]
  - Typically not the “real” physical address but virtual (logical) address (we do not care about this difference)
  - Cannot be used for register variables
- We can print address with format specifier %p

```
root [20] int test = 314;
root [21] printf("Address of variable test is %p\n",&test);
Address of variable test is 0x7f0d7ea34090
```

- To store results of address of operator, we need a variable called pointer.
- A pointer is variable that stores an address. A pointer has an address of its own (since it's a variable).
- ! Avoid pointers of any kind in C++, if really needed use std::unique\_ptr, std::shared\_ptr, std::weak\_ptr [not covered in this lecture]

# Pointers

- General format for defining a pointer is

```
data_type *name_of_pointer;
```

- For example

```
int *test_pointer1, *test_pointer2;
```

- Now “test\_pointer1” and “test\_pointer\_2” are pointers to an integer. The variable store address and that address is assumed to contain an integer.

- Now if we have int variable “test”, we can store its address into the pointer test\_pointer1

```
test_pointer1 = &test;
```

```
int* ptr1, * ptr2, not_a_pointer;
int *ptr1, *ptr2, not_a_pointer;
```

ptr1 and ptr2 are pointers  
not\_a\_pointer is not!  
Lower example has better coding style!

# Pointers

- We dereference pointers with “\*” (do not confuse this with multiplication operator)
- By dereferencing a pointer, we go to the value stored at the address that the pointer contains
- We can read the value of the original variable and every modify that!
- There can be multiple pointers pointing to same memory address
  - However, each pointer itself is stored in different memory address

```
root [24] int test_int = 123;
root [25] int *test_ptr = &test_int;
root [26] *test_ptr
^H(int) 123
root [27] *test_ptr = 777;
root [28] test_int // value of test_int has been changed!
(int) 777
```

# Call by reference by pointers

- C++ version using references

- Swapping two values by feeding pointers to the swap function instead of actual values [Avoid this in C++]

```
9 #include <stdio.h>
10 void swap(int*,int*);
11
12 int main()
13 {
14 int a = 10, b = 20;
15 printf("a = %d, b = %d\n", a,b);
16 swap(&a,&b);
17 printf("a = %d, b = %d\n", a,b);
18 return 0;
19 }
20
21 void swap(int * v1,int * v2)
22 {
23 int temp;
24 temp = *v1;
25 *v1 = *v2;
26 *v2 = temp;
27 }
```

```
#include <iostream>
void swap(auto &a, auto &b)
{
 auto temp = a;
 a = b;
 b = temp;
}

int main()
{
 int a = 5, b = 10;
 std::cout << "Before swap: " << a << " " <<
 swap(a, b);
 std::cout << "After swap: " << a << " " <<
 return 0;
}
```

# Pointers and arrays

- Name of C-style array corresponds/decays to the address of its first element for example, or function calls (the array is NOT copied!)
  - (mostly, but not for sizeof operator)

```
root [41] int test_arr[3]={1,2,3};
root [42] int *int_ptr;
root [43] int_ptr = &test_arr[0]
(int *) 0x7f0d7ea340e8
H root [44] int_ptr = test_arr //
(int *) 0x7f0d7ea340e8
```

Equivalent!  
Array name is converted to the pointer to its  
first element

# Pointers and arrays

- Adding integer  $n$  to a pointer gives a pointer which points  $n$  **elements** further along an array than the original pointer did.
  - Not  $n$  bytes further but  $n$  elements ( further!!! )

```
root [0] int test_arr[3] = {1,2,3};
root [1] int *int_ptr = test_arr;
root [2] int_ptr[1] = 256;
root [3] *(int_ptr+2) = 512;
root [4] test_arr
(int [3]) { 1, 256, 512 }
```

int\_ptr[1] = 256; is equivalent to  
\*(int\_ptr+1) = 256;

# Passing name of C-style array (=>pointer) to a function



Mandatory to also include the size of the array (it cannot be found from the pointer alone!)

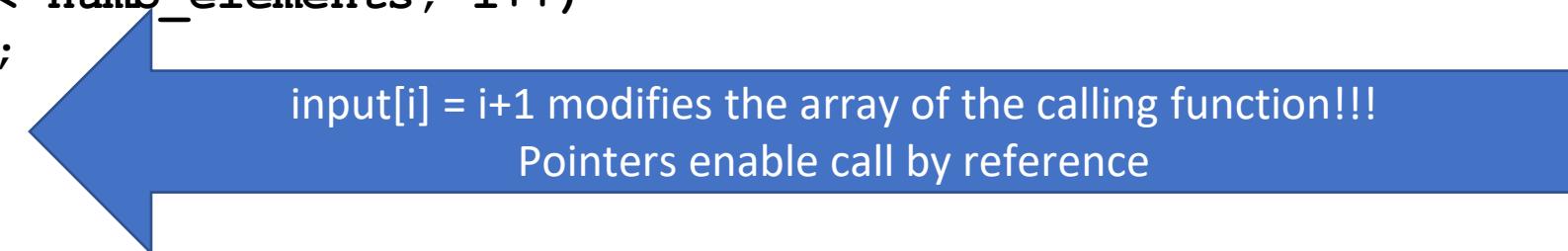
```
#include <stdio.h>
double sum_array(double *input, size_t numb_elements)
{
 double result = 0;
 for (int i = 0; i < numb_elements; i++)
 result += input[i];
 return result;
}

int main()
{
 double arr[] = {1, 2, 10.5};
 printf("Sum = %f\n", sum_array(arr, sizeof(arr) / sizeof(arr[0])));
 return 0;
}
```

# Call by reference (in place editing)

```
#include <stdio.h>
void change_array(double *input, size_t num_elements)
{
 for (int i = 0; i < num_elements; i++)
 input[i] = i+1;
}

int main()
{
 double arr[] = {1123,4.25,100.5};
 for (int i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
 printf("arr[%d]=%f\t", i, arr[i]);
 printf("\n");
 change_array(arr, sizeof(arr)/sizeof(arr[0]));
 for (int i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
 printf("arr[%d]=%f\t", i, arr[i]);
 return 0;
}
```



Before function call:

arr[0]=1123.000000 arr[1]=4.250000 arr[2]=100.500000

After function call:

arr[0]=1.000000 arr[1]=2.000000 arr[2]=3.000000

# In place editing in MATLAB/MEX

- Unfortunately, MATLAB does not allow C/MEX to do in-place editing of the input vector/matrix.
- We need a separate output array that can be modified
  - Provided by our header files



**Do not modify input array in MEX**

# Vector input one output in MATLAB C (MEX), using parameters

```
#define NEXTRA_PARAMETERS 1 // CAN BE ZERO!
#include "VECTOR_INPUT_ONE_OUTPUT_REAL.h"
double MATLAB_main(double *input, size_t numb_elements, double param1)
{
 double temp = 0;
 for(int c = 0; c < numb_elements; c++)
 temp += pow(input[c],param1);
 return temp;
}
```

Input can be row or column vector (but not matrix!)  
Output is scalar.

```
>> A = [1 2 3 4 5 6 7 8 9 10];
>> out = example_vector_input_one_output(A,3)
```

out =

3025

```
>> out = sum(A.^3)
```

out =

3025

# Vector input vector output in MATLAB C (MEX), using parameters

```
#define NEXTRA_PARAMETERS 1 // CAN BE ZERO!
#include "VECTOR_INPUT_VECTOR_OUTPUT_REAL.h"

void MATLAB_main(double* output, double* input, size_t numb_elements, double param1)
{
 for(int c = 0; c < numb_elements; c++)
 output[c] = input[c] + param1;
}
```

No return statement!

Input can be row or column vector  
(but not matrix!)

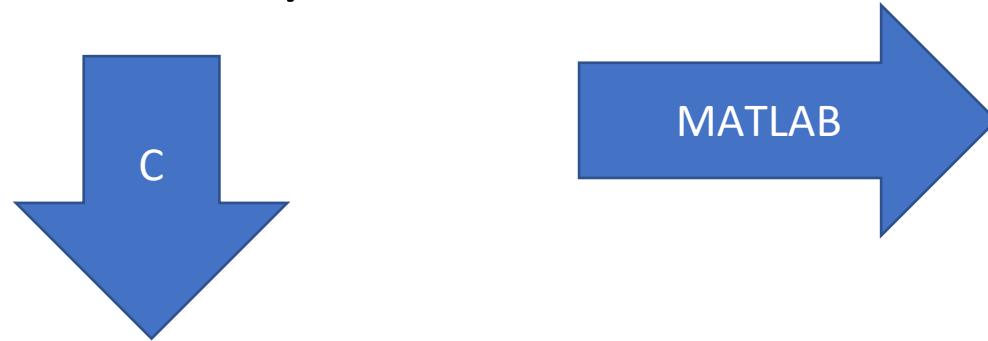
Output is vector of the same size as  
the largest dimension of the input

```
>> A = [1 2 3 4 5 6 7 8 9 10];
>> out = example_vector_input_vector_output(A,3)
```

out =

4 5 6 7 8 9 10 11 12 13

# Multidimensional arrays



```
root [30] int a[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
root [31] a[0][0]
(int) 1
root [32] a[0][1]
(int) 2
root [33] a[0][2]
(int) 3
root [34] a[0][3]
(int) 4
root [35] a[1][0]
(int) 5
root [36] a[2][3]
(int) 12
```

```
>> a = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

a =

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

Matrices (multidimensional arrays) are not that useful in C and not even in C++

**Recommended to NOT use them.**

Instead of use for example external C++ library called "eigen"

# Multidimensional arrays

matrix:  $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$

| Address | Row-major order (used by C) | Column-major order (MATLAB) |
|---------|-----------------------------|-----------------------------|
| 0       | $a_{11}$                    | $a_{11}$                    |
| 1       | $a_{12}$                    | $a_{21}$                    |
| 2       | $a_{13}$                    | $a_{12}$                    |
| 3       | $a_{21}$                    | $a_{22}$                    |
| 4       | $a_{22}$                    | $a_{13}$                    |
| 5       | $a_{23}$                    | $a_{23}$                    |

- For example MATLAB stores  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$  in memory with data in order [1,4,2,5,3,6]. But C will store in order [1,2,3,4,5,6].
- ⚠ Different memory order for matrices in C and MATLAB (does not matter for vectors) must be taken into account

```
1 #include <iostream>
2 #include <Eigen/Eigen>
3 int main()
4 {
5 Eigen::MatrixXd m{{1, 2, 3}, {4, 5, 6}};
6 std::cout << "Here is the matrix n:\n" << m << std::endl;
7 m.array() += -1; // Add -1 to each element of the matrix
8 std::cout << "[Again] Here is the matrix n:\n" << m << std::endl;
9 m.transposeInPlace(); // Transpose the matrix in place
10 std::cout << "[Again2] Here is the matrix n:\n" << m << std::endl;
11 m += Eigen::MatrixXd::Ones(3,2); // Add two matrices
12 std::cout << "[Again3] Here is the matrix n:\n" << m << std::endl;
13 std::cout << "m(1,0) = " << m(1,0) << std::endl; // Access an element
14 }
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    COMMENTS

Here is the matrix n:

1 2 3

4 5 6

[Again] Here is the matrix n:

0 1 2

3 4 5

[Again2] Here is the matrix n:

0 3

1 4

2 5

[Again3] Here is the matrix n:

1 4

2 5

3 6

m(1,0) = 2

- Avoid C++ multidimensional arrays
- Instead use for example “eigen”
  - It can enable “MATLAB-like” array processing in C++
  - Installation is quite easy
  - Eigenvalues, norm, etc.
- For quick reference, please see
- [eigen.tuxfamily.org/dox/AsciiQuickReference.txt](http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt)

# Matrix in MATLAB C (MEX)

```
// mex -R2018a CFLAGS="$CFLAGS -fopenmp -std=c11" LDFLAGS="$LDFLAGS -fopenmp"
COPTIMFLAGS="-O3" example_matrix_input_one_output.c
#define NEXTRA_PARAMETERS 0
#include "MATRIX_INPUT_ONE_OUTPUT_REAL.h"
double MATLAB_main(double* input, size_t numb_rows, size_t numb_columns)
{
 double temp = 0;
 for(int c = 0; c < numb_columns; c++)
 {
 for(int r = 0; r < numb_rows; r++)
 {
 // Now we are at using MATLAB array indexing at input(r+1,c+1)
 // mexPrintf("input[%d] [%d] = %0.2f \t", r, c, input[r + numb_rows*c]);
 temp += input[numb_rows*c + r];
 }
 // mexPrintf("\n");
 }
 return temp;
}
```

We can interpret MATLAB matrix as (one dimensional) vector,  
can be useful.

If we would loop over rows first, speed would be 6x times slower!!!

The multiplication "numb\_rows\*c" needs to be done only once per inner for loop!!! [compiler is smart]

# Const qualifier

- The const qualifier states that variable is constant
- `const int two = 2 // this cannot be modified`

```
root [39] const int two = 2;
root [40] two = 123;
ROOT_prompt_40:1:5: error: cannot assign to variable 'two' with const-qualified type 'const int'
two = 123;
~~~~~^
ROOT_prompt_39:1:11: variable 'two' declared const here
const int two = 2;
~~~~~^~~~~~
```

- Reference can also be const (useful for example for function calls where copying overhead needs to be avoided but data is not modified)

```
root [0] int x = 123;
root [1] const int &y = x;
root [2] y
(const int) 123
root [3] y = 543;
ROOT_prompt_3:1:3: error: cannot assign to variable 'y' with
y = 543;
~ ^
ROOT_prompt_1:1:12: note: variable 'y' declared const here
const int &y = x;
~~~~~^~~~~~
```

# Pointer to const

- A pointer to memory that should not be modified can be declared as “pointer to const”

```
root [48] int arr[3]={1,2,3};  
root [49] const int *p = arr;  
root [50] p[0]  
(const int) 1  
root [51] *(p+0)  
(const int) 1  
root [52] *(p+1)  
(const int) 2  
root [53] *(p+2)  
(const int) 3  
root [54] *p=10;  
ROOT_prompt_54:1:3: error: read-only variable is not assignable  
*p=10;  
~~^  
root [55] p[0]=10;  
ROOT_prompt_55:1:5: error: read-only variable is not assignable  
p[0]=10;  
~~~~~^
```

p is pointer to const int  
(p itself can be modified, it is not constant)

# Pointers to void

*Pointers of different types are not the same. There are no implicit conversions from one to the other (unlike the arithmetic types).*

There are a few occasions when you *do* want to be able to sidestep some of those restrictions, so what can you do?

The solution is to use the special type, introduced for this purpose, of ‘pointer to `void`’. This is one of the Standard’s invented features: before, it was tacitly assumed that ‘pointer to `char`’ was adequate for the task. This has been a reasonably successful assumption, but was a rather untidy thing to do; the new solution is both safer and less misleading. There isn’t any other use for a pointer of that type—`void *` can’t actually point to anything—so it improves readability. A pointer of type `void *` can have the value of any other pointer assigned to and can, conversely, be assigned to any other pointer. This must be used with great care, because you can end up in some heinous situations. We’ll see it being used safely later with the `malloc` library function.

# Null pointers

Assigning 0  
to pointer is  
equal to  
assigning  
NULL to a  
pointer  
(both lead to  
null pointer)

You may also on occasion want a pointer that is guaranteed not to point to any object—the so-called **null pointer**. It's common practice in C to write routines that return pointers. If, for some reason, they can't return a valid pointer (perhaps in case of an error), then they will indicate failure by returning a null pointer instead.

An example could be a table lookup routine, which returns a pointer to the object searched for if it is in the table, or a null pointer if it is not.

How do you write a null pointer? There are two ways of doing it and both of them are equivalent: either an integral constant with the value of 0 or that value converted to type `void *` by using a cast. Both versions are called the **null pointer constant**. If you assign a null pointer constant to any other pointer, or compare it for equality with any other pointer, then it is first converted the type of that other pointer (neatly solving any problems about type compatibility) and will not appear to have a value that is equal to a pointer to any object in the program.

```
root [84] char *q = 0;
root [85] q
(char *) nullptr
root [86] if (q) printf("Valid pointer\n"); else printf("Invalid pointer!\n");
Invalid pointer!
root [87] char *p = NULL;
root [88] p
(char *) nullptr
```

# Strings in C [⚠️ recommended to be avoided, if using C++ use C++ std::string]

- String is an C is array of characters ending with 0 (to indicate end of string)
- Based on above we can define

```
char a[3]={'H', 'i', 0};
```

- Now we can print it using printf("%s\n",a) resulting into  
“Hi”

- We can more conveniently write

```
char a[3] = “Hi” // (we need space for “H”, “i” and 0)
```

or

```
char a[] = “Hi” // Automatically correct size
```

We can modify the string, e.g., using a[0] = ‘h’; => String will become “hi”.  
Here we used character constant ‘h’ since “h” would be two characters (“h” and 0)

# Strings in C

- String can be shorter than the character array it is stored in (if the end of the string (marked with 0) is earlier than end of the character array. *The maximum length is the array length minus one (since last character needs to be 0)*)
- For example,

```
root [7] char ch_array[256] = "Hi";
root [8] ch_array[0]
(char) 'H'
root [9] ch_array[1]
(char) 'i'
root [10] ch_array[2]
(char) '0x00'
root [11] puts(ch_array);
Hi
```

0x00 marks the end of string

# String in C

- The character constants such as ‘A’ take only one byte.
- With double quotes “A” is actually string composed of two characters, ‘A’ and 0.

```
H root [13] char ch = 'A';
root [14] char test_strng[] = "A";
root [15] sizeof(ch)
(unsigned long) 1
RT root [16] sizeof(test_strng)
H(unsigned long) 2
```

# Strings constants/literals

- We can also write

```
const char *s = "Hello World!";
```

- Now s is pointer to constant array (string constant cannot be modified!)

```
root [23] const char *s = "Hi";
root [24] s
(const char *) "Hi"
root [25] s[0]
(const char) 'H'
root [26] s[1]
(const char) 'i'
root [27] s[2]
(const char) '0x00'
root [28] s[0] = 'A'
ROOT_prompt_28:1:6: error: read-only variable is not assignable
s[0] = 'A'
~~~~~ ^
```

# String constants

- Unlike normal arrays, string constants can be returned from a function!
- String constants have static duration and remain alive for the program lifetime.
- Remember that typically local variables of a function will be destroyed once the function exits
- Function that takes an integer as an argument and returns "Even" or even numbers or "Odd" for odd numbers. As below, use “const char \*” instead of “char \*” as string literals cannot be modified!

```
const char * even_or_odd(int number)
{
    if (number % 2 == 0)
        return "Even";
    return "Odd";
}
```

# String length

- First, we need to include #include <string.h>
- Then we can use the strlen function

```
root [22] char test_array[256] = "Hello";
root [23] printf("Len is %zu\n",strlen(test_array));
Len is 5
root [24] sizeof(test_array) // Not string length!
(unsigned long) 256
```



%zu is the  
format  
specifier for  
size\_t (strlen  
output type)

- strlen will give length of the string Excluding the zero at the end. If the string is not zero terminated, the whole program can collapse!
  - ! ! ! Strlen as most of the C string functions is not good for secure programming...

# Reverse a string in-place

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

void strrev(char *string) // return nothing since reversing is in-place!
{
    int string_length = strlen(string);
    for (int ind = 0; ind < string_length/2; ind++)
    {
        char temp = string[ind];
        string[ind] = string[ string_length - 1 - ind];
        string[ string_length - 1 - ind] = temp;
    }
}

int main()
{
    char string_test[] = "World";
    strrev(string_test);
    puts(string_test);
    return 0;
}
```



# Stack vs heap memory

- Local variables are allocated to stack
- Stack is very limited, typically only 1-8 megabytes
- Heap memory is only limited by total memory of the system (can be gigabytes)
- Do not put large variables in stack
  - Instead allocate memory for them from heap using malloc
- Variables in heap are not automatically destroyed on function exit
  - You need to free the memory when you have finished using it using “free” command
-  C++ has “new” and “delete” as “better” alternatives, but they are not recommended anymore.
  - Instead use for example std::vector (preferred), or smart pointers, etc.

# Heap memory / malloc & free

```
#include <stdio.h>
#include <stdlib.h>
#define NUMB_INT 5

void init_array(int *p, int number_of_elements)
{
    for (int i = 0; i < number_of_elements; i++)
        p[i] = i*i;
}

int main()
{
    int *int_ptr = malloc(sizeof(int)*NUMB_INT) ;
    init_array(int_ptr,NUMB_INT);
    for (int i = 0; i < NUMB_INT; i++)
        printf("array element %d is %d\n", i, int_ptr[i]);
    free(int_ptr); // free the allocated memory
    return 0;
}
```



If we allocate a double array, it would say  
malloc(sizeof(double)\*NUMB\_DOUBLE)

# Codewars.com

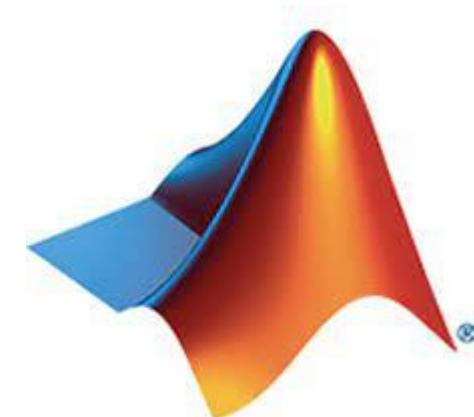
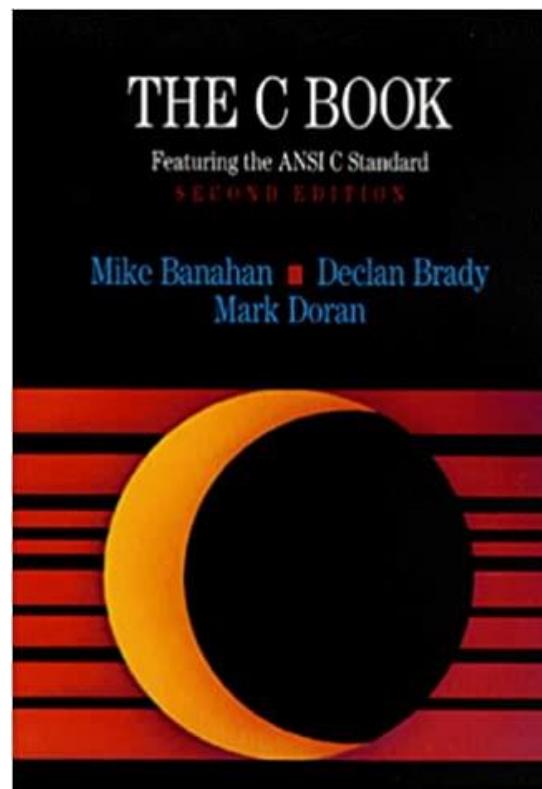
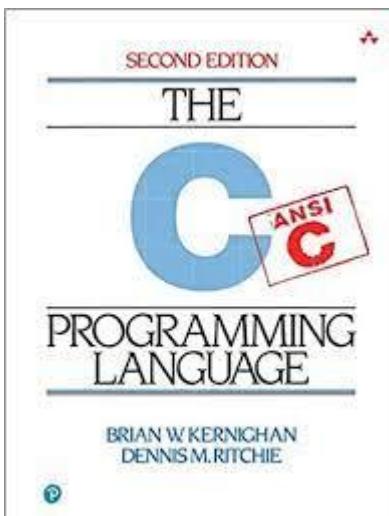
- Build a function that returns an array of integers from n to 1 where n>0. Example : n=5 --> [5,4,3,2,1]. Note: allocate memory yourself.

```
/* return NULL in case n == 0 */
#include <stdlib.h>
unsigned short *reverse_seq(unsigned short n)
{
    if (n == 0)
        return NULL;
    unsigned short *p = malloc(sizeof(unsigned short )*n);
    for (int i = 0; i < n; i++)
        p[i] = n-i;
    return p;
}
```



Notice no free!  
Caller has to  
call free!!!

# Using C with MATLAB #06



[https://publications.gbdirect.co.uk/c\\_book/copyright.html](https://publications.gbdirect.co.uk/c_book/copyright.html)

# ANSI C 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    |

---

# Volatile keyword

- Volatile tells the compiler that the variable may be modified outside the program and our actions in it (such as in an interrupt etc.)
- Compiler cannot do optimisations such as buffering the value in cache
- For example, if you write 3 to a volatile int, when you in next line add something to that int, compiler cannot assume that its value is 3

```
volatile int interface_flag = 0;  
...  
while(interface_flag == 0){  
    // do something  
    // that does not modify the flag  
}
```



Not necessarily infinite while loop!  
Other means than our code can change the value of the variable

# Typedef

- Typedef allows us to create alias for another type
- For example,
- `typedef unsigned char BYTE;`
- Now BYTE is alias for unsigned char and we can write for example
- `BYTE byte_variable = 100; // define and initialize variable of type byte  
(alias for unsigned char)`

```
root [47] typedef unsigned short shrt_uint;
root [48] shrt_uint var1 = 1000;
root [49] var1
(unsigned short) 1000
```

# Register keyword

- Suggestion for compiler to place the variable in a CPU register
- Compiler is fully free to ignore this suggestion
- Not very useful these days and also prevents you from using address of operation on it (& is not allowed for register variables)

# Structures

- Arrays can be used group together same-type items
- Structures can be used to group together items of different kinds
- Below declaration declares structure with tag “car” with members “year” of the type int, “weight” of the type double, and “horsepower” of the type int

```
struct car {  
    int year;  
    double weight;  
    int horsepower;  
};
```

“car” is optional structure tag

Notice the semicolon “;”. It is not optional!

- Above does not yet define any variable! “car” is not a variable, just a structure type!

# Structures

- After declaring structure type with tag car, we can define a variable(s)

```
struct car first_car, second_car;
```

- Above defines two structure variables of type car: first\_car and second\_car.
- Both first\_car and second\_car contain all the 3 elements of “struct car” (previous slide).
- We can also declare and define structure at the same time:

```
struct car {  
    int year;  
    double weight;  
    int horsepower;  
} car1, car2;
```

- Above declares structure type “car” and also defines two structure variables car1 and car2. We can still define car3 same way as before:

```
struct car car3;
```

# Structures

- By using `typedef` we can use structure without typing the `struct` keyword.

```
1  typedef struct car {
2      int year;
3      double weight;
4      int horsepower;
5  } car;           ← The new name can be same as struct tag
6
7  int main()
8  {
9      car car1, car2, car3;
10 }
```

car1, car2, car3 are all structures of type “car” (struct car)

# Structures

- We can access members of the structure variable by dot operator “.”

```
root [9] car1.year = 2000; // assigns 2000 to member year of car1
root [10] int temp = car1.year;
root [11] printf("Car year is %d\n",car1.year);
Car year is 2000
```

- In MATLAB we also use dot operator:

```
>> car1 = struct('year',2000,'weight',1950,'horsepower',200)

car1 =

struct with fields:

    year: 2000
    weight: 1950
    horsepower: 200

>> car1.year

ans =

    2000
```

# Structures

- When define a new variable, we can initialize it at the same time by using curly brackets { }

```
root [13] car car4 = {2000,1950,200};  
root [14] car4.year  
(int) 2000  
root [15] car4.weight  
(double) 1950.0000  
root [16] car4.horsepower  
(int) 200
```

- If you put fewer initializers than there are elements in the structure, rest will be automatically initialized to zero

# Structures,C99 designated initialization

```
1 #include <stdio.h>
2 struct xy_w_string
3 {
4     int x;
5     int y;
6     char *p;
7 };
8 int main()
9 {
10    struct xy_w_string a_xy_point = {.y=256, .x=10, .p = "Red"};
11    printf("x = %d, y = %d\n", a_xy_point.x, a_xy_point.y);
12    puts(a_xy_point.p);
13    return 0;
14 }
```

Order of initialization does not  
matter in this case

- Uninitialized members are automatically set to zero

x = 10, y = 256

Red

# Structures

- One structure can be assigned to another one with “=” operator.
- However, ==, >, >=, <, <= etc. operations are not permitted on structures.
- Only operation permitted on structures is assignment
  - They can be assigned to each other, passed as arguments to functions and returned by functions. However, it is not a very efficient operation to copy structures and most programs avoid structure copying by manipulating pointers to structures instead. It is generally quicker to copy pointers around than structures.

```
root [1] car car5, car6 = {1990,1500,200};  
root [2] car5 = car6;  
root [3] car5.year  
(int) 1990  
root [4] car5.year = 2000;  
root [5] car6.year //not affected by above modification of car5!  
(int) 1990
```

# Structure copy

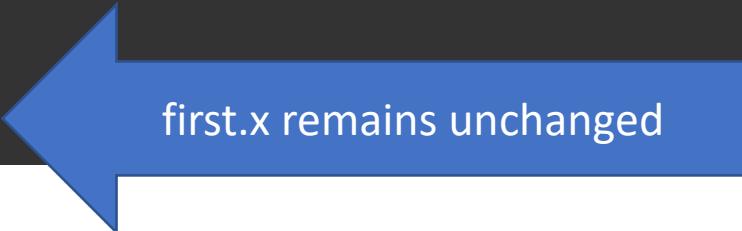
- Structure copy is not deep copy
  - The data that pointers point to is not copied to a new memory location
  - This would be impossible since length of where pointers point to is not known to runtime
- Instead, pointer is copied as a new pointer pointing to the same address

```
root [0] struct xy_w_string {int x; int y; char *p;};
root [1] char stringi[]="Hello";
root [2] struct xy_w_string a_xy_point = {10,20,stringi};
root [3] struct xy_w_string second_point = a_xy_point;
root [4] second_point.x = 256;
root [5] a_xy_point.x
(int) 10
root [6] second_point.p[0] = 'Y';
root [7] puts(a_xy_point.p)
Yello
```

# Structure copy

- Genuine arrays are properly copied!

```
root [11] struct stru_w_array { int x[4];};  
root [12] struct stru_w_array first = { .x={1,2,3,4} }, second;  
root [13] second = first;  
root [14] second.x  
(int [4]) { 1, 2, 3, 4 }  
root [15] second.x[0] = 10;  
root [16] second.x  
(int [4]) { 10, 2, 3, 4 }  
root [17] first.x  
(int [4]) { 1, 2, 3, 4 }
```



first.x remains unchanged

# Structures and functions

```
#include <stdio.h>
typedef struct car {
    int year;
    double weight;
    int horsepower;
} car;
```

```
void print_car(car);
```

```
int main()
{
    car car1 = {2000, 1950.0, 150};
    print_car(car1);
    return 0;
}
```

```
void print_car(car temp_car)
{
    printf("Car year = %d, weight = %f, horsepower = %d\n",
           temp_car.year,
           temp_car.weight, temp_car.horsepower);
}
```



Car year = 2000, weight = 1950.000000, horsepower = 150



# Structures

- C functions can be called with structure input and they can also return structures (unlike arrays).
- Name of structure is NOT a pointer
- Call-by-value!
- Original car1 is not modified!

```
1 #include <stdio.h>
2
3 typedef struct car {
4     int year;
5     double weight;
6     int horsepower;
7 } car;
8
9 car modify_car(car);
10
11 int main()
12 {
13     car car1 = {2000,1950,150}, car2;
14     car2 = modify_car(car1);
15     printf("Car1 year = %d, Car 2 year = %d\n",car1.year,car2.year);
16 }
17
18 car modify_car(car input_car)
19 {
20     input_car.year++;
21     input_car.weight = 1500;
22     return input_car;
23 }
```

**We return struct variable just like any other variable!**

input

Car1 year = 2000, Car 2 year = 2001

```
#include <stdio.h>
typedef struct car {
    int year;
    double weight;
    int horsepower;
} car;

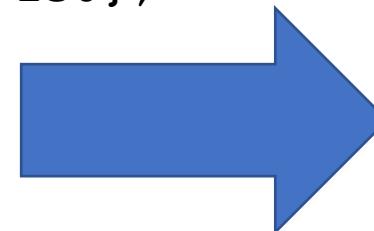
void print_car(car);

car edit_car(car temp_car)
{
    temp_car.year = temp_car.year + 1;
    temp_car.horsepower += 50;
    return temp_car;
}

int main()
{
    car car1 = {2000, 1950.0, 150};
    print_car(car1);
    car1 = edit_car(car1);
    print_car(car1);
    return 0;
}
```

# Modify a structure (call-by-value)

```
void print_car(car c)
{
    printf("Car year = %d, weight = %f, horsepower = %d\n",
c.year, c.weight, c.horsepower);
}
```



Car year = 2000, weight = 1950.000000, horsepower = 150  
Car year = 2001, weight = 1950.000000, horsepower = 200

# Structures

```
#include <stdio.h>
typedef struct complex
{
    double real;
    double imag;
} complex_double;

complex_double multiply(complex_double n1, complex_double n2);

int main() {
    complex_double n1 = {10,20}, n2 = {30,40}, res;
    res = multiply(n1, n2);
    printf("Product = %f %fi\n", res.real, res.imag);
    return 0;
}

complex_double multiply(complex_double n1, complex_double n2) {
    complex_double temp;
    temp.real = n1.real*n2.real - n1.imag*n2.imag;           >> (10+20i)*(30+40i) %MATLAB
    temp.imag = n1.imag*n2.real + n2.imag*n1.real;
    return temp;
}

=> Product = -500.000000 1000.000000i
                                         -500 +
                                         1000i
```

# Structures

- Pointers to structure variable can be passed to functions
- For large structures, this is more efficient than passing copy of the whole structure to a function
- If we have pointer to structure, its members can be accessed with arrow operator “->”

```
#include <stdio.h>
typedef struct car {
    int year;
    double weight;
    int horsepower;
} car;

void print_car(car *);

int main()
{
    car car1 = {2000, 1950.0, 150};
    print_car(&car1);
    return 0;
}

void print_car(car *p)
{
    printf("Car year = %d, weight = %f, horsepower
= %d\n", p->year, p->weight, p->horsepower);
}
```

# Structures

- Pointers to structure variable can be passed to functions
- For large structures, this is more efficient than passing copy of the whole structure to a function
- If we have pointer to structure, its members can be accessed with arrow operator “->”
- Call-by-reference!
- Original car1 is modified!



```
1 #include <stdio.h>
2
3 -> typedef struct car {
4     int year;
5     double weight;
6     int horsepower;
7 } car;
8
9 void modify_car(car *);
10
11 int main()
12 {
13     car car1 = {2000,1950,150};
14     printf("Car1 year is = %d\n",car1.year);
15     modify_car(&car1);
16     printf("Car1 year is now = %d\n",car1.year);
17 }
18
19 void modify_car(car *input_car_pointer)
20 {
21     input_car_pointer->year = 1950;
22 }
```

input

Car1 year is = 2000  
Car1 year is now = 1950

# Structures

- You can define array of structures as usual with for example “int” data-type.
- Remember that array (of any type) cannot be returned from a function

```
1 typedef struct car {
2     int year;
3     double weight;
4     int horsepower;
5 } car;
6
7 int main()
8 {
9     car array_of_car[2]; // Array of structures!
10    array_of_car[0].year = 2000;
11    array_of_car[1].year = 1950;
12 }
```

# Array of structures

```
#include <stdio.h>
#include <stdlib.h>
typedef struct xy {
    int x,y;
} xy;

void print(xy *, size_t);

int main()
{
    xy arr[2] = {{10,20},{30,40}};
    print(arr, sizeof(arr)/sizeof(arr[0]));
    return 0;
}

void print(xy * p, size_t numb_elements)
{
    for (int i = 0; i < numb_elements; i++)
        printf("x = %d, y = %d\n", p[i].x, p[i].y);
}
```

Name or array “arr” is a pointer!  
Call-by-reference!

x = 10, y = 20  
x = 30, y = 40

# Structures

- If you need large number of structures in an array, it is better to use malloc to get memory from them
  - This is because stack memory is limited compared to heap (also we can dynamically decide the number of elements)

```
3  typedef struct car {  
4      int year;  
5      double weight;  
6      int horsepower;  
7  } car;  
8  
9  int main()  
10 {  
11     car * car_ptr = malloc(sizeof(car)*2);  
12     car_ptr[0].year = 2000;  
13     car_ptr[1].horsepower = 128;  
14     // Accessing car_ptr[2] would be serious mistake!  
15     free(car_ptr);  
16 }
```

# Structures Erroneous Code

```
1 #include <stdio.h>
2
3 - typedef struct xy{
4     int x;
5     int y;
6 } xy;
7
8 xy * make_xy(void);
9
10 int main()
11 {
12     xy * point_ptr;
13     point_ptr = make_xy();
14 }
15
16 xy * make_xy(void)
17 {
18     xy temp;
19     temp.x = 100;
20     temp.y = 128;
21     return &temp;
22 }
23
```

As usual, local variables are destroyed once the function exits! We cannot return pointer to a local variable!



input

```
main.c:21:12: warning: function returns address of local variable [-Wreturn-local-addr]
21 |     return &temp;
|           ^~~~~~
```

# Return a pointer to structure

```
#include <stdio.h>
#include <stdlib.h>
typedef struct xy {
    int x,y;
} xy;

xy* make_xy(int in1, int in2)
{
    xy * p = malloc(sizeof(xy));
    p->x = in1;
    p->y = in2;
    return p;
}

int main()
{
    xy *point_ptr;
    point_ptr = make_xy(10,20);
    printf("Point x = %d, y = %d\n", point_ptr->x, point_ptr->y);
    free(point_ptr);
    return 0;
}
```

# Structures

- Structures can be nested: there can a structure inside a structure

```
#include <stdio.h>
struct color{
    int r,g,b;
};

typedef struct xy{
    int x, y;
    struct color rgb;
} xy;

int main()
{
    xy tpoint= {100,128,{10,11,12}};
// OR: xy tpoint= {.x=100,.y=128,.rgb = {10,11,12}};

    printf("x=%d,y=%d,r=%d,g=%d,b=%d\n",
          tpoint.x, tpoint.y, tpoint.rgb.r, tpoint.rgb.g, tpoint.rgb.b);
}
```

This needs to be declared before xy  
since xy is using this (and not just a pointer to it)

Element  
b within  
element  
rgb of  
tpoint

# Structures

- One way to pass (and return) arrays with pass-by-value is to embed them into a structure!
- Beware of the overhead is copying the full array to temp variable and then copying full array to var2!

Returning temp is OK!

We can return structure same was as we can return e.g. "int"

```
1 #include <stdio.h>
2
3 typedef struct embed_arr{
4     int arr[3];
5 } embed_arr;
6
7 embed_arr arr_proc_func(embed_arr);
8
9 int main()
10 {
11     embed_arr var1 = { .arr = {10,20,30} };
12     embed_arr var2 = arr_proc_func(var1);
13     for (int i = 0; i < 3; i++)
14         printf("Var1.arr[%d] = %d, Var2.arr[%d] = %d\n",
15                i, var1.arr[i], i, var2.arr[i]);
16 }
17
18 embed_arr arr_proc_func(embed_arr temp)
19 {
20     for (int i = 0; i < 3; i++)
21         temp.arr[i] += i;
22     return temp;
23 }
24
25
```

Call-by-value!  
Var1 is not changed!

input  
Var1.arr[0] = 10, Var2.arr[0] = 10  
Var1.arr[1] = 20, Var2.arr[1] = 21  
Var1.arr[2] = 30, Var2.arr[2] = 32

# Structures

- Structures can be heavily padded, due to alignment requirements of its different members
- Below, the sizeof structure is 12 bytes when the sum of the sizeof of the individual members is only 6 bytes!

```
root [15] struct testi2 {char a; int b; char c;};
root [16] sizeof(struct testi2)
(unsigned long) 12
root [17] sizeof(char)
(unsigned long) 1
root [18] sizeof(int)
(unsigned long) 4
```

# Structures

- The ordering of data elements inside a struct can affect its size!

```
root [29] struct testi2 {char a; int b; char c;};
root [30] sizeof(struct testi2)
(unsigned long) 12
root [31] struct testi3 {char a; char c; int b;};
root [32] sizeof(struct testi3)
(unsigned long) 8
```

# Structures

- We can use compiler specific instructions to pack structures more tightly.
- With GCC we can use `__attribute__((__packed__))`

```
root [10] struct __attribute__((__packed__)) testi {char a; int b; char c;};
root [11] sizeof(struct testi)
(unsigned long) 6
```

- Now only 6 bytes are consumed! (But accessing the elements can be quite much slower due to non-alignment)
  - We can also write
- ```
#pragma pack(1)
```
- which will affect all subsequent structures

# Unions

- Union is like structure but all its members share the same memory.
- Can in theory be used for example to convert between different types
- Let us extract hexadecimal representation of a float

```
root [57] union float_and_int { float x; int y;};  
root [58] union float_and_int testi;  
root [59] testi.x = 123.5;  
root [60] printf("Float %f in hexadecimal is %x\n",testi.x, testi.y);  
Float 123.500000 in hexadecimal is 42f70000
```

- Notice that we cannot do `printf("Float %f in hexadecimal is %x\n", testi.x, (int) testi.x)`! This would just print 123 in hexa.

|                                 |                                  |
|---------------------------------|----------------------------------|
| You entered                     | 123.5                            |
| Value actually stored in float: | 123.5                            |
| Error due to conversion:        | 0.0                              |
| Binary Representation           | 01000010111101110000000000000000 |
| Hexadecimal Representation      | 0x42f70000                       |

Same

# Unions

- Remember that union members share the same memory.
  - Changing one member will change the other members
  - Only one member is valid at a time!

```
root [67] union int_and_int { int x; int y;};  
root [68] union int_and_int testi3;  
root [69] testi3.x = 10;  
root [70] testi3.x  
(int) 10  
root [71] testi3.y  
(int) 10  
root [72] testi3.y = 777;  
root [73] testi3.y  
(int) 777  
root [74] testi3.x  
(int) 777
```

# Bitfields

- Like structures but we use keywords “signed” and “unsigned” and specify the number of bits each element consumes

```
root [0] struct test_bfield {unsigned x:4; unsigned y:4;};
root [1] struct test_bfield test;
root [2] test.x = 15;
root [3] test.y = 16; //valid values from 0 15 (4 bits)
ROOT_prompt_3:1:8: warning: implicit truncation from 'int' to bit-field changes value
from 16 to 0 [-Wbitfield-constant-conversion]
test.y = 16; //valid values from 0 15 (4 bits)
^ ~
root [4] sizeof(struct test_bfield)
(unsigned long) 4
```

- But above `test_bfield` takes huge number of bytes (4)!
- We need to use `__attribute__((__packed__))`
  - Now `test_bfield` consumes only 1 byte!!!

# Preprocessor

- Preprocessor looks at the source code before compilation and for example replaces `#define`'d values with the definitions.
- Preprocessor lines start with `#`
- We are already familiar with `#include` and simple `#define`
- Preprocessor statements do not end semicolon “;” instead they end with a newline (“enter”)

Do not put semicolon at the end of preprocessor statements

```
#define
```

```
#include <stdio.h>
#define NUMB_EL 3
#define END_MESSAGE "Returning now!"

int main(void)
{
    int arr[NUMB_EL];
    for (int i = 0; i < NUMB_EL; i++)
        arr[i] = i;
    puts(END_MESSAGE);
    return 0;
}
```

# #undef

The name of any `#defined` identifier can be forcibly forgotten by saying

```
#undef NAME
```

It isn't an error to `#undef` a name which isn't currently defined.

This occasionally comes in handy. Chapter 9 points out that some library functions may actually be macros, not functions, but by undefining their names you are guaranteed access to a real function.

# Macro arguments

- Let us define preprocessor macro SQUARE which takes one input
- Notice the multiple parenthesis (they are required to avoid problems)
- Notice how data type of output depends on the data type on input

```
root [158] #define SQUARE(a) ((a)*(a))
root [159] SQUARE(100)
(int) 10000
root [160] SQUARE(50000)
ROOT_prompt_160:1:1: warning: overflow in expression; re
er-overflow
SQUARE(50000)
^

ROOT_prompt_158:1:23: expanded from macro 'SQUARE'
#define SQUARE(a) ((a)*(a))
^

(int) -1794967296
root [161] SQUARE(50000L)
(long) 2500000000
root [162] SQUARE(10.0)
(double) 100.00000
```

# Macro arguments [ERRONEOUS MACRO]

- Let us define preprocessor macro SQUARE which takes one input
- What can go wrong without parenthesis???

```
root [0] #define SQUARE(a) (a*a)
root [1] SQUARE(10)
(int) 100
root [2] SQUARE(1+9) // will not give 100!
(int) 19
root [3] // above is 1+9*1+9 = 1 + 9 + 9 = 19
```

# Macro arguments

- We can also have multiple input arguments to a preprocessor “function”

```
root [163] #define PRODUCT(a,b) ((a)*(b))
root [164] PRODUCT(10,20)
(int) 200
root [165] PRODUCT(10.0,20.0)
(double) 200.00000
```

```
root [23] #define MAX(a,b) ((a)>(b)?(a):(b))
root [24] MAX(10,20)
(int) 20
root [25] MAX(20,10.0)
(double) 20.00000
```

- We can check if the definition for given name or not
- Remember this happens before compilation
- Compiler only sees the text on right hand side below:

- Would be enough to write
- “#define VAL” instead of “#define VAL 1”
- Even “#define VAL 0” would hit #ifdef as TRUE!!!!

The diagram illustrates the preprocessing flow. On the left, the original C code is shown with a red oval highlighting the preprocessor directive `#define VAL 1`. A large blue arrow points from this code to the right, where a simplified version of the code is shown. In the simplified version, the `#define` line has been removed, and the condition `VAL` is now evaluated directly by the compiler. The resulting output is displayed in the terminal window on the right.

```
#include <stdio.h>
#define VAL 1

int main(void)
{
#ifdef VAL
    printf("Val is defined!\n");
#else
    printf("Val is not defined!\n");
#endif
}
```

```
#include <stdio.h>
int main(void)
{
    printf("Val is defined!\n");
}
```

Even value defined as zero (or no value at all) will hit #ifdef as TRUE

# #ifdef vs #if

```
#include <stdio.h>
#define VAL 0
int main(void)
{
#ifdef VAL
    printf("Val is defined\n");
#else
    printf("Val is non-defined\n");
#endif
}
```

=> Val is defined

```
#include <stdio.h>
#define VAL 0
int main(void)
{
#if VAL
    printf("Val is non-zero\n");
#else
    printf("Val is zero/does not exist\n");
#endif
}
```

=> Val is zero/does not exist

#if check is the value is non-zero  
#ifdef checks is the macro defined at all  
(even with value 0!)

# #ifndef and #if !defined

```
#include <stdio.h>
int main(void)
{
#ifndef VAL
    printf("Val is not defined\n");
#else
    printf("Val is defined\n");
#endif
}
```

Val is not defined

```
#include <stdio.h>
int main(void)
{
#if !defined(VAL)
    printf("Val is not defined\n");
#else
    printf("Val is defined\n");
#endif
}
```

Val is not defined

# #if & #elif

```
#include <stdio.h>
#define VAL 3
int main(void)
{
    #if VAL == 2
        printf("Value is two!\n");
    #elif VAL == 3
        printf("Value is three!\n");
    #else
        printf("Value is not 2 or 3!\n");
    #endif
}
```

No “;” at the end

No brackets ( ) around expression

#elif is support by preprocessor!

If nothing above hits...

We need to end #if with #endif

Value is three!

VAL is not a variable!

#if #elif #else is happening before compilation by the C preprocessor (CPP)

# Structures vs. Classes

- **C Structures:**
  - Group related data together.
  - All members are `public` by default.
  - No built-in support for functions.
- **C++ Classes:**
  - Can contain both data and functions (methods).
  - Default access specifier is `private`.
  - Enable encapsulation (data hiding) and abstraction.

# Encapsulation and Data Hiding

- **Encapsulation** means bundling the data (members) with the functions (methods) that operate on that data.
- **Data Hiding** restricts direct access to some of an object's components.
- **Benefits:**
  - Protects internal state from unintended modification.
  - Simplifies maintenance: Users call methods without needing to know the internal representation.
  - Promotes modularity and abstraction.

# Passing Objects to Functions

- Objects can be passed **by value**, **by reference**, or **by pointer**.
- **By Value:** Copies the entire object (can be expensive for large objects).
- **By Reference:** Avoids copying and allows the function to use the original object. Use `const` if modification is not needed.
- **Example:**

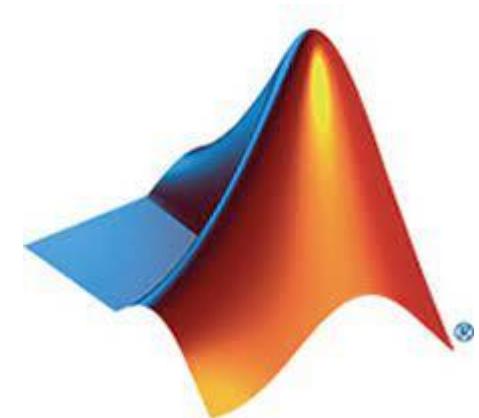
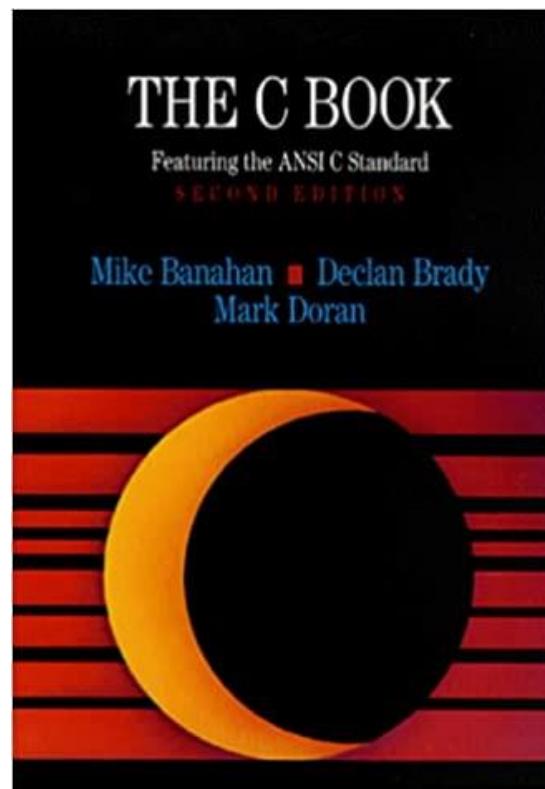
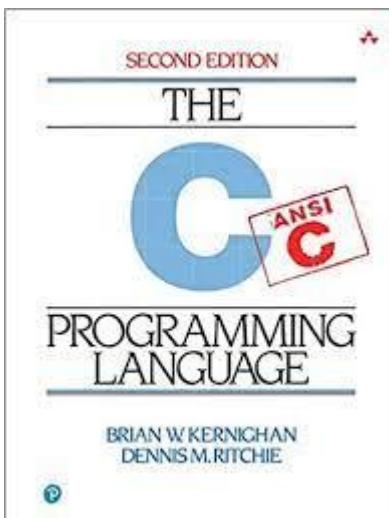
```
void printPoint(const Point& p);
```

```
1 #include <iostream>
2 #include <cmath>
3 class Point
4 {
5 private:
6     double x, y; // Private members
7 public:
8     // Constructor using an initializer list
9     Point(double x, double y) : x(x), y(y) {}
10    // Method
11    double distanceFromOrigin() const
12    {
13        return std::sqrt(x * x + y * y);
14    }
15 };
16 void printPoint(const Point &p)
17 {
18     std::cout << "Distance from origin: " << p.distanceFromOrigin() << std::endl;
19 }
20 int main()
21 {
22     Point p1(3, 10);
23     printPoint(p1); // Passing by reference
24     std::cout << "Direct method call: " << p1.distanceFromOrigin() << std::endl;
25     return 0;
26 }
```

# Summary

- **C++ Classes** extend C structures by combining data and methods.
- **Encapsulation** helps protect internal data and provides a clear interface.
- Objects can be efficiently passed to functions by reference.
- The Point example demonstrates how to define a class, use a constructor, and create public methods.

# Using C with MATLAB #07



[https://publications.gbdirect.co.uk/c\\_book/copyright.html](https://publications.gbdirect.co.uk/c_book/copyright.html)

# Standard libraries

- The C89/90 standard includes these libraries

|            |            |            |
|------------|------------|------------|
| <assert.h> | <locale.h> | <stddef.h> |
| <ctype.h>  | <math.h>   | <stdio.h>  |
| <errno.h>  | <setjmp.h> | <stdlib.h> |
| <float.h>  | <signal.h> | <string.h> |
| <limits.h> | <stdarg.h> | <time.h>   |

- C99/C11 added several new libraries

- We will cover <tgmath.h> in addition to “classical” <math.h> [<>tgmath.h> is more similar to <cmath> in C++ than <math.h>]

- For full list see

[https://en.wikipedia.org/wiki/C\\_standard\\_library#Application\\_programming\\_interface](https://en.wikipedia.org/wiki/C_standard_library#Application_programming_interface)

# Malloc

- For malloc, we need to #include <stdlib.h>
- malloc allocates storage for n bytes (where n is the input argument to malloc) and return a pointer to the start the memory block (if possible). If malloc fails to allocate the memory, it will return a null pointer.
- It is good coding practice to check the return value of malloc.
- Because return type of malloc is void \*, its return value can be assigned to any pointer variable.
- ! Remember to free allocated memory! !



Memory returned by malloc is uninitialized (garbage)



# Malloc and free

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define NINT 8
4
5 int * allocate_array(void);
6
7 int main()
8 {
9     int * q = allocate_array();
10
11    for (int i = 0; i < NINT; i++)
12        printf("Array element %d is %d\n", i, q[i]);
13
14    free(q); // Free the memory from malloc!
15    return 0;
16 }
17
18 int * allocate_array(void)
19 {
20     int * temp = malloc(sizeof(int)*NINT);
21     for (int i = 0; i < NINT; i++)
22         temp[i] = i;
23     return temp;
24 }
```



Array obtained with malloc can be returned! (it is responsibility of calling function to free the memory)

# Memcpy [quite fast 🏃 way to copy memory]

- #include <string.h> [memcpy is not limited to “string”!!!]

```
memcpy(destination,source,sizeof(double)*numb_elements);
```

- Above copies numb\_elements **doubles** from memory addresses starting from **source** (a pointer) to memory addresses starting at **destination (a pointer)**
- Same approach can be used for copying **integers** etc.

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     int source[3] = {1,2,100};
6     int destination[3]; // Not initialized
7     memcpy(destination,source,sizeof(int)*3);
8     for (int i = 0; i < 3; i++)
9         printf("Element %d is %d\n", i, destination[i]);
10
11     return 0;
12 }
```

-  **Make sure “destination” has enough memory!**
- **Overlap between source and destination is NOT allowed**

# <assert.h>

- While you are debugging programs, it is often useful to check that the value of an expression is the one that you expected. The assert function provides such a diagnostic aid.
- In order to use assert you must first include the header file <assert.h>. The function is defined as

```
#include <assert.h>
void assert(int expression)
```

- If the expression evaluates to zero (i.e. false) then assert will write a message about the failing expression, including the name of the source file, the line at which the assertion was made and the expression itself. After this, the abort function is called, which will halt the program.

```
assert(1 == 2);
```

- /\* Might result in \*/

```
Assertion failed: 1 == 2, file silly.c, line 15
```

# <assert.h>

- Assert is actually defined as a macro, not as a real function. In order to disable assertions when a program is found to work satisfactorily, defining the name **NDEBUG** **before** including <assert.h> will disable assertions totally. You should beware of side effects that the expression may have: when assertions are turned off with NDEBUG, the expression is not evaluated. Thus, the following example will behave unexpectedly when debugging is turned off with the #define NDEBUG.

```
#define NDEBUG
#include <assert.h>

void
func(void)
{
    int c;
    assert((c = getchar()) != EOF);
    putchar(c);
}
```

```
p = malloc(sizeof(int));
assert(p)
```

Is not a very good idea since asserts are typically disabled in release version of a program!

*Example 9.2*

- Note that assert returns no value.

# <ctype.h>

isalnum(int c)

True if c is alphabetic or a digit;  
specifically (isalpha(c) || isdigit(c)).

isalpha(int c)

True if (isupper(c) || islower(c)).

isdigit(int c)

True if c is a decimal digit.

islower(int c)

True if c is a lower case alphabetic letter.

isprint(int c)

True if c is a printing character (including space).

isspace(int c)

True if c is a white space character

isupper(int c)

True if c is an upper case alphabetic character.

isxdigit(int c)

True if c is a valid hexadecimal digit.

# Limits.h

- You can use this for example to check if “int” is sufficient for your purposes or you need for example “long long”

| Name       | Allowable value        | Comment                                      |
|------------|------------------------|----------------------------------------------|
| CHAR_BIT   | ( $\geq 8$ )           | bits in a <code>char</code>                  |
| CHAR_MAX   | see note               | max value of a <code>char</code>             |
| CHAR_MIN   | see note               | min value of a <code>char</code>             |
| INT_MAX    | ( $\geq +32767$ )      | max value of an <code>int</code>             |
| INT_MIN    | ( $\leq -32767$ )      | min value of an <code>int</code>             |
| LONG_MAX   | ( $\geq +2147483647$ ) | max value of a <code>long</code>             |
| LONG_MIN   | ( $\leq -2147483647$ ) | min value of a <code>long</code>             |
| MB_LEN_MAX | ( $\geq 1$ )           | max number of bytes in a multibyte character |
| SCHAR_MAX  | ( $\geq +127$ )        | max value of a <code>signed char</code>      |
| SCHAR_MIN  | ( $\leq -127$ )        | min value of a <code>signed char</code>      |
| SHRT_MAX   | ( $\geq +32767$ )      | max value of a <code>short</code>            |
| SHRT_MIN   | ( $\leq -32767$ )      | min value of a <code>short</code>            |
| UCHAR_MAX  | ( $\geq 255U$ )        | max value of an <code>unsigned char</code>   |
| UINT_MAX   | ( $\geq 65535U$ )      | max value of an <code>unsigned int</code>    |
| ULONG_MAX  | ( $\geq 4294967295U$ ) | max value of an <code>unsigned long</code>   |
| USHRT_MAX  | ( $\geq 65535U$ )      | max value of an <code>unsigned short</code>  |

|            |                                         |                                    |
|------------|-----------------------------------------|------------------------------------|
| LLONG_MIN  | Min value for a long long int           | $\leq -9,223,372,036,854,775,807$  |
| LLONG_MAX  | Max value for a long long int           | $\geq +9,223,372,036,854,775,807$  |
| ULLONG_MAX | Max value for an unsigned long long int | $\geq +18,446,744,073,709,551,615$ |

# Float.h

- Most interesting values here are

DBL\_EPSILON ( $\leq 1E-9$ ) minimum positive number such that  $1.0 + x \neq 1.0$

- When double is 64 bits this is actually
- ```
#define DBL_EPSILON  
2.2204460492503131e-16
```
- which is (naturally) same as MATLAB “eps”
- It is smallest number such that  $1+x$  is not 1
- For long double there is

```
root [6] LDBL_EPSILON  
(long double) 1.0842022e-19L
```

```
root [0] DBL_EPSILON  
(double) 2.2204460e-16  
root [1] 1+DBL_EPSILON  
(double) 1.0000000  
root [2] DBL_EPSILON  
(double) 2.2204460e-16  
root [3] (1+DBL_EPSILON)-1  
(double) 2.2204460e-16  
root [4] (1+DBL_EPSILON/2)-1  
(double) 0.0000000
```

# Random number generation

Might be very poor quality random numbers, better to use for example POSIX (often available) drand48-function

- Function `rand()` in `<stdlib.h>` return pseudo-random number from 0 to `RAND_MAX`.
- To get random integer between 0 and 99 we can use `rand()%100`
- **⚠️ `rand()` will always give the exactly same sequence (with default seed)**  
`srand` allows a given starting point in the sequence to be chosen according to the value of seed. If `srand` is not called before `rand`, the value of the seed is taken to be 1. The same sequence of values will always be returned from `rand` for a given value of seed.
- To get different random numbers each time we run the program, we can set use current time as seed

```
root [101] #include <time.h>
root [102] srand(time(NULL));
root [103] rand() % 1000
```

- **⚠️ Make sure to call `srand` only once, at the beginning of the program!**

# QSORT

- For sorting data in C, we need comparison function such for doubles

```
int cmpfunc (const void * a, const void * b){  
    return (*(double*)a > *(double*)b) ? 1 : (*(double*)a < *(double*)b) ? -1:0 ; }
```

Above returns 1 if  $*a > *b$ , 0 if  $*a$  is equal to  $*b$ , and -1 if  $*a < *b$

We can basically compare any types if we provide reasonable comparator function. Comparison function needs to return negative value if the first argument is less than the second, positive value if the first argument is greater than the second and zero if the arguments are the same.

After we have comparison assembly-based we call qsort this way

- `qsort(data_to_be_sorted, numb_elements, sizeof(double[for example]), cmpfunc);`
- Where **data to be sorted (it's a pointer)** points to the data be sorted in-place, **numb\_elements** is the number of elements, `sizeof(double)` is the size of each element (here `double`), and `cmpfunc` is the comparator function
- **⚠** qsort is sadly not so “quick sort” 😱 due to for example using a generic function pointer for comparisons (even if we just use `int` or `double`), which prevents inlining and incurs overhead on each comparison
- In C++, use for example `std::sort` instead (typically faster).
- In C, you can for example look for assembly-based non-standard implementations

# QSORT

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int cmpfunc (const void * a, const void * b)
5 {
6     return (*(double*)a > *(double*)b) ? 1 : (*(double*)a < *(double*)b) ? -1:0 ;
7 }
8
9
10 int main()
11 {
12     double data_to_be_sorted[4] = {7,1,100,5};
13     qsort(data_to_be_sorted,4, sizeof(double),cmpfunc);
14     for (int i = 0; i < 4; i++)
15         printf("Element %d is %lf\n", i, data_to_be_sorted[i]);
16
17     return 0;
18 }
```

Element 0 is 1.000000

Element 1 is 5.000000

Element 2 is 7.000000

Element 3 is 100.000000

Codewars.com problems

Create a function that returns the sum of the two lowest numbers given an array. Do not modify original array.

```
#include <stddef.h>
#include <stdlib.h> // MALLOC, FREE, QSORT
#include <string.h> // MEMCPY
int cmpfunc(const void * a, const void * b) {
    int arg1 = *(const int*)a;
    int arg2 = *(const int*)b;
    if (arg1 < arg2) return -1;
    if (arg1 > arg2) return 1;
    return 0;
}
long sum_two_smallest_numbers(size_t n, const int *numbers)
{
    long result = 0;
    int *temp_ptr = malloc(sizeof(int)*n);
    memcpy(temp_ptr, numbers, sizeof(int)*n);
    qsort(temp_ptr, n, sizeof(int), cmpfunc);
    result = (long)temp_ptr[0] + (long)temp_ptr[1];
    free(temp_ptr);
    return result;
}
```



⚠ Many cmpfunc from “Internet” have errors ⚠  
Use only **reliable** places such as [cppreference.com](http://cppreference.com)  
[qsort, qsort.s - cppreference.com](http://qsort.qsort.s-cppreference.com)  
For example, don’t use:  
`return ( *(int*)a - *(int*)b );`  
It fails for INT\_MIN

*Do not modify the original array.*  
So we are allocating memory for that with malloc  
Copying original to copy with memcpy  
Then qsort:ing inplace (it modifies the copy)  
[notice this is not the fastest approach but just an example]

# Working with files

- FILE is a structure defined in <stdio.h>. Pointer to FILE is called a file pointer.
- Following defines a file pointer ftpr:

```
FILE *ftpr;
```

**FILE** The type of an object used to contain stream control information. Users of stdio never need to know the contents of these objects, but simply manipulate pointers to them. It is not safe to copy these objects within the program; sometimes their addresses may be ‘magic’.

# File opening and closing

- Named files are opened by a call to the fopen function, whose declaration is this:

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

```
FILE *fopen(const char *pathname, const char *mode);
```

- The pathname argument is the name of the file to open
- Files can be opened in a variety of modes, such as read mode for reading data, write mode for writing data, and so on. [modes can be “binary” or “text”. We focus on “text”](#)
- If error occurs during opening file, fopen() function will return a NULL pointer.
-  Remember to close file with fclose-function!

# Working with files

- Three streams are available without any special action; they are normally all connected to the physical device associated with the executing program: usually your terminal.
- They are referred to by the names
  - **stdin**, the standard input
  - **stdout**, the standard output
  - **stderr**, the standard error streams.
- Normal keyboard input is from stdin, normal terminal output is to stdout, and error messages are directed to stderr.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void)
4 {
5     FILE *f = NULL;
6     f = fopen("output.txt", "w");
7     if (f == NULL)
8     {
9         printf("Error opening file!\n");
10        return 1; // 1 is returned to indicate error
11    }
12    for (int i = 0; i < 10; i++)
13    {
14        fprintf(f, "Now for loop index is %d\n", i);
15        // fprintf with stdout is same as printf
16        fprintf(stdout, "File written successfully\n");
17        fclose(f); // remember to close the file!
18        return 0; // 0 is returned to indicate success
19 }
```

"w" means for writing

```
File Edit View
output.txt
Now for loop index is 0
Now for loop index is 1
Now for loop index is 2
Now for loop index is 3
Now for loop index is 4
Now for loop index is 5
Now for loop index is 6
Now for loop index is 7
Now for loop index is 8
Now for loop index is 9
```

fprintf to stdout will "print" to screen (not a file in hard disk)

# fscanf

- For easily reading data from text file, we can use fscanf.
- Its format specifiers are listed, for example, in
  - <https://www.cplusplus.com/reference/cstdio/fscanf/>
- Several differences to printf format specifiers!!!!
- For example, double must be %lf (%f is only for float)
-  Always check format specifiers for (f)scanf! Do not just directly use printf format specifiers!
  - (the reason is that printf has automatic promotions but scanf not)

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void)
4 {
5     double n1, n2;
6     FILE *f = fopen("input.txt", "r");
7     if (f == NULL)
8     {
9         perror("Error opening file"); // Print error message
10    return 1; // Exit with error
11 }
12 // fscanf returns the number of items successfully read!
13 while (fscanf(f, "%lf %lf", &n1, &n2) == 2)
14 {
15     printf("Read: %lf %lf\n", n1, n2);
16 }
17 fclose(f); // Close the file
18 return 0; // 0 means success
19 }
```

"r" means for reading

fscanf cannot read "HEI!!!!" as double => it will return 0 (not 2) as number of items read

Read: 1.200000 3.000000  
Read: 5.000000 777.000000  
Read: 10.000000 1023.123123  
Read: 100000.000000 1000.000000

# Complex data type (C99+) [Do not use in C++ but use std::complex instead]

- Need to include `#include <tgmath.h>` (`tgmath` includes `complex.h`)
  - Automatically included in our header files!
- `complex double cval; // defines complex variable cval`
- `complex float cval; // defines single-precision complex variable cval`
- `complex long double cval; // defines extended precision complex variable cval`

# Complex data type

- Depending on the input type the type generic math functions automatically call the correct function

Type-generic  
“function”

carg

conj

creal

cimag

carg: Computes the argument (also called phase angle) of z. In MATLAB this is “angle” command. Returns a real number.

conj: Calculates complex conjugate of z. In MATLAB this is “conj” command.

creal: Return the real part of a complex number. In MATLAB this is “real” command.

cimag: Return the imaginary part of a complex number. In MATLAB this is “imag” command.

# Complex data type

CMPLX(real\_part,imag\_part)  
[type is complex double]

"I" is the imaginary unit (C99+)

```
1 #include <stdio.h>
2 #include <tgmath.h>
3
4 int main(void)
5 {
6     complex double c1 = CMPLX(5.0, -4.0), c2;
7     printf("c1 = %f%+fi\n", creal(c1), cimag(c1));
8     c2 = c1 + (3+2*I); // complex numbers support basic operations!!!
9     printf("c2 = %f%+fi\n", creal(c2), cimag(c2));
10    return 0;
11 }
```

c1 = 5.000000-4.000000i

c2 = 8.000000-2.000000i

# <tgmath.h> [C99+] Type generic math

- Also for real numbers, there are many type generic math functions
- For functions taking real or imaginary input, appropriate function is called

<https://en.cppreference.com/w/c/numeric/tgmath>

For different input types (for example float/double), type generic math functions will call the proper function and return the proper type

Use type generic math functions instead of manually finding function for float/double/long double. Also type generic functions handle equally well complex and real input arguments!

Convenience of type-generic macros  
[cppreference.com]

There are also many real-only type-generic macros:

|           |            |      |
|-----------|------------|------|
| atan2     | cbrt       | ceil |
| copysign  | erf        |      |
| erfc      | exp2       |      |
| expm1     | fdim       |      |
| floor     | fma        |      |
| fmax      | fmin       |      |
| fmod      | frexp      |      |
| hypot     | ilogb      |      |
| ldecp     | lgamma     |      |
| llrint    | llround    |      |
| log10     | log1p      |      |
| log2      | logb       |      |
| lrint     | lround     |      |
| nearbyint |            |      |
| nextafter | nexttoward |      |
| remainder | remquo     |      |
| rint      | round      |      |
| scalbln   | scalbn     |      |
| tgamma    | trunc      |      |

| Type-generic macro | Real function variants |        |             | Complex function variants |        |             |
|--------------------|------------------------|--------|-------------|---------------------------|--------|-------------|
|                    | float                  | double | long double | float                     | double | long double |
| <b>fabs</b>        | fabsf                  | fabs   | fabsl       | cabsf                     | cabs   | cabsl       |
| <b>exp</b>         | expf                   | exp    | expl        | cexpf                     | cexp   | cexpl       |
| <b>log</b>         | logf                   | log    | logl        | clogf                     | clog   | clogl       |
| <b>pow</b>         | powf                   | pow    | powl        | cpowf                     | cpow   | cpowl       |
| <b>sqrt</b>        | sqrtf                  | sqrt   | sqrtl       | csqrtf                    | csqrt  | csqrnl      |
| <b>sin</b>         | sinf                   | sin    | sinl        | csinf                     | csin   | csinl       |
| <b>cos</b>         | cosf                   | cos    | cosl        | ccosf                     | ccos   | ccosl       |
| <b>tan</b>         | tanf                   | tan    | tanl        | ctanf                     | ctan   | ctanl       |
| <b>asin</b>        | asinf                  | asin   | asinl       | casinf                    | casin  | casinl      |
| <b>acos</b>        | acosf                  | acos   | acosl       | cacosf                    | cacos  | cacosl      |
| <b>atan</b>        | atanf                  | atan   | atanl       | catanf                    | catan  | catanl      |
| <b>sinh</b>        | sinhf                  | sinh   | sinhl       | csinhf                    | csinh  | csinhl      |
| <b>cosh</b>        | coshf                  | cosh   | coshl       | ccoshf                    | ccosh  | ccoshl      |
| <b>tanh</b>        | tanhf                  | tanh   | tanh        | ctanhf                    | ctanh  | ctanh       |
| <b>asinh</b>       | asinhf                 | asinh  | asinh       | casinhf                   | casinh | casinh      |
| <b>acosh</b>       | acoshf                 | acosh  | acosh       | cacoshf                   | cacosh | cacosh      |
| <b>atanh</b>       | atanhf                 | atanh  | atanh       | catanhf                   | catanh | catanh      |

6 choices!

## Example: C99 tgmath.h Usage (Code)

```
#include <stdio.h>
#include <tgmath.h> // Provides type-generic macros
#include <complex.h> // For C99 complex types

int main(void) {
    double x = -3.14;
    float complex fc = -2.0f + 1.0f*I;

    // tgmath will figure out the correct function:
    printf("fabs(x) = %f\n", fabs(x)); // Expands to
        fabs(double)
    printf("fabs(fc) = %f\n", fabs(fc)); // Expands to
        cabsf(float complex)
    return 0;
}
```

# Example

- fabs: Computes the absolute value of the z. For complex z this is also known as norm, modulus, or magnitude. In MATLAB this is “abs” command. Returns a real number.

```
#include <stdio.h>
#include <tgmath.h>
int main()
{
    complex double c1 = 5 - 4*I, c2;
    printf("c1 = %f%+fi\n",creal(c1),cimag(c1));
    printf("abs(c1) = %f\n", fabs(c1));
    return 0;
}
```

c1 = 5.000000-4.000000i

abs(c1) = 6.403124

In MATLAB:

```
>> c1 = 5-4*1i;
>> abs(c1)
```

ans =

6.40312423743285

⚠ Do not use abs function in C for floating point numbers! It is in an integer operation but compiler will not warn you! Use fabs with <tgmath.h>!

[In C++, you actually should use std::abs especially for complex numbers]

# Example: [ ! WRONG CODE]

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
int main()
{
    complex long double c1 = 5 - 4*I, c2;
    printf("abs(c1) = %Lf\n", fabs(c1));
    return 0;
}
```

Fabs will not call the type-generic math function since math.h was included instead of tgmath.h

Type generic math requires tgmath.h not math.h!!!

If we include math.h we need to manually specify the proper function with correct datatype

<math.h>: correct function would be (instead of fabs) cabsl

<tgmath.h>: fabs is always correct

# Example

**pow**

**powf**

**pow**

**powl**

**cpowf**

**cpow**

**cpowl**

- **pow** (the type-generic version): Computes the (possibly complex) power function  $x^y$ 
  - In MATLAB this is simply  $x^y$ .
  - Be very careful since  $x^y$  in C means XOR operation! You cannot use “ $\wedge$ ” in C for power operation.

```
root [24] 10^2
(int) 8
root [25] pow(10,2)
(double) 100.00000
```

⚠ You must use `pow(x,y)` for power function [when using `tgmath.h`]!  $x^y$  in C will return totally different results!!!

# MATLAB mxComplexDouble

- MATLAB complex datatype is mxComplexDouble which is struct composed of real part and imaginary part
- `typedef struct { mxDouble real, imag; } mxComplexDouble;`
- We can access real and imaginary parts as usual with the dot operator
- Storage in memory is using new interleaved format (mx -R2018a)



- Exactly same as storage style as C99 complex double!  
=> We can cast between them

# Complex number representation

- If we have array of `mxComplexDouble` input we can access real and imaginary part this way:
  - `input[i].real`
  - `input[i].imag`
- We can cast `input` to be of type `(complex double *)` for directly converting between `mxComplexDouble` and C99 complex doubles.
- Or we if we process one number at a time, we can make `complex double` `cval` and assign to it

`input[i].real + I*input[i].imag`

# Example vector input one output

```
example_complex_vector_input_one_output.c +  
1 // mex -v -R2018a CFLAGS="$CFLAGS -fopenmp -std=c11" LDFLAGS="$LDFLAGS -fopenmp" COPTIME  
2 #define NEXTRA_PARAMETERS 1  
3 #include "VECTOR_INPUT_ONE_OUTPUT_COMPLEX.h"  
4 complex double MATLAB_main(mxComplexDouble* input, size_t numb_elements, double param1)  
5 {  
6     complex double cval, output = 0;  
7     for(int c = 0; c < numb_elements; c++)  
8     {  
9         cval = input[c].real+ I*input[c].imag;  
10        output += pow(cval, param1);  
11    }  
12    return output;  
13 }
```

>> r = randn(1,1E7)+1i\*randn(1,1E7);  
>> output\_mex = example\_complex\_vector\_input\_one\_output(r,1.25)  
output\_mex =  
-2489681.67657273 + 4728.69423938935i

We convert from mxComplexDouble to “complex double”

Doing complex power operation on cval

This is slower than MATLAB sum( $r.^{1.25}$ ) but with OpenMP (discussed later) we are a bit faster than MATLAB vector function!

# Example vector input one output (using casting)

```
// mex -v -R2018a CFLAGS="$CFLAGS -fopenmp -std=c11" LDFLAGS="$LDFLAGS -fopenmp"  
COPTIMFLAGS="-O3" example_complex_vector_input_one_output.c  
#define NEXTRA_PARAMETERS 1  
#include "VECTOR_INPUT_ONE_OUTPUT_COMPLEX.h"  
complex double MATLAB_main(mxComplexDouble* input, size_t numb_elements, double param1)  
{  
    complex double output = 0;  
    complex double *input_ptr = (complex double *) input;  
    for(int c = 0; c < numb_elements; c++)  
        output += pow(input_ptr[c], param1);  
    return output;  
}
```

Output is  
the same as  
with the  
program in  
the previous  
slide

We cast pointer to  
**mxComplexDouble**  
to a pointer to  
**complex double**!  
Due to same  
memory layout  
this works!!!

# Example vector input vector output (using casting)

```
#define NEXTRA_PARAMETERS 1
#include "VECTOR_INPUT_VECTOR_OUTPUT_COMPLEX.h"
void MATLAB_main(mxComplexDouble* output, mxComplexDouble* input,
size_t numb_elements, double param1)
{
    complex double *output_ptr = (complex double *) output;
    complex double *input_ptr = (complex double *) input;
    for(int c = 0; c < numb_elements; c++)
        output_ptr[c] = pow(input_ptr[c], param1);
}
```

```
>> r = randn(1,1E7)+1i*randn(1,1E7);
>> output = example_complex_vector_input_vector_output(r,1.25);
>> output_MATLAB = r.^1.25; norm(output-output_MATLAB)
1.13088226530426e-12
```

We cast pointers to  
mxComplexDouble  
to pointers to  
complex double!  
Due to same  
memory layout this  
works!!!

Order of floating-point operations matters a bit so there will be some  
difference  
This difference is very small and is fine

# What is the STL?

- The **Standard Template Library (STL)** is a core part of C++.
- Provides **containers**, **iterators**, and **algorithms**:
  - **Containers**: `std::vector`, `std::list`, `std::map`, `std::set`, etc.
  - **Iterators**: Pointer-like objects used to traverse containers.
  - **Algorithms**: `std::sort`, `std::find`, `std::accumulate`, etc.
- Allows for concise, reliable, and *high-performance* code with less manual memory management.

# Key Container: std::vector

- Similar to a dynamic array (like a C array with `malloc`), but manages memory automatically.
- **Advantages:**
  - Automatically resizes as needed.
  - Provides random access via `operator[]` and iterators.
  - Commonly used for most “growable array” cases.
- Basic usage:
  - `std::vector<int> v; // create empty vector`
  - `v.push_back(10); // add element at end`
  - `v.size(); // get current size`
  - `v[i] = 20; // index-based access`

## Example: std::sort

```
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort

int main() {
    std::vector<int> v{10, 3, 7, 1, 12};

    // Sort in ascending order
    std::sort(v.begin(), v.end());
    // v is now {1, 3, 7, 10, 12}

    // Print
    for (auto val : v)
        std::cout << val << " ";
    std::cout << "\n";
    return 0;
}
```

# Summary

- The **STL** is a powerful part of C++: **containers, iterators, algorithms**.
- **std::vector** is a go-to container for dynamic arrays.
- Functions like `std::sort`, `std::find`, and `std::accumulate` greatly simplify common tasks.
- Explore other containers (`std::list`, `std::map`, `std::set`, etc.) and algorithms to get the most out of the STL.

# **Using C with MATLAB #08**

**Adapted from course Book for this topic:  
Parallel Programming for Science  
Engineering, Open source CC-BY-4.0**

**by Victor Eijkhout**

**Chapters 17-20 (and selected aspects of  
later ones) covered in this lecture**

# ⚠️ OpenMP

- OpenMP is designed for shared-memory architectures, which are typical in modern computers.
  - This means all processors share a single memory space, unlike distributed memory systems where each processor has its own private memory (and usually use MPI).
  - OpenMP offers an easy way to take advantage of the increasing number of processor cores in today's systems, making it essential for high-performance computing.
  - ⚠️ However, unlike standard C code, you might write parallel code that compiles without warnings but contains race conditions, leading to inconsistent or incorrect results on each run. ⚠️

# MATLAB and C++ options

- MATLAB Coder (automatic MATLAB to C conversion with OpenMP support) can be a good option for beginners since it automatically analyzes code for parallelization and includes checks to reduce the risk of race conditions.
  - **This is often easiest and safest option to make parallel code for a beginner**
  -  While it often produces safe OpenMP-based code, thorough testing is still essential to catch hidden data dependencies.
-  Meanwhile, modern C++ offers parallel algorithms via tool such as **`std::execution::parallel_policy`**, which can accelerate operations on standard containers. However, just like with OpenMP, it doesn't guarantee race-free execution if shared data isn't carefully managed, so careful design and testing remain crucial for correctness.
  - `std::execution::par` can still be more "easier"/"safer" than OpenMP for simple vector operations

# SIMD Optimization in Modern Development

- Modern compilers are very good at autovectorization, which means they automatically generate SIMD instructions for simple loops.
- This reduces the need for manual SIMD optimization, especially for beginners.
- With limited time, it is often more beneficial to focus on parallelization (e.g., using OpenMP or C++ parallel algorithms) to leverage multiple cores.
- In summary, while SIMD can improve performance, autovectorizing compilers lessen the need for manual tuning, so prioritizing parallelization is usually best.

# Introduction: More Than Just Pragmas

- Many beginners assume that simply adding a pragma (e.g., `#pragma omp parallel for`) will automatically speed up their code.
- In reality, performance optimization is complex and iterative.
- Blindly adding pragmas can lead to race conditions or even slower performance.

# Compiler Feedback is Invaluable

```
slide14.c:21:23: missed: couldn't vectorize loop  
slide14.c:10:5: missed: not vectorized: unsupported data-type double  
slide14.c:10:5: note: vectorized 1 loops in function.
```

opt.log

```
[build] C:\Users\jjlehtom\Documents\projects\lecture_08\slide14.c:69:13: optimized: loop  
vectorized using 32 byte vectors  
[build] C:\Users\jjlehtom\Documents\projects\lecture_08\slide14.c:56:39: optimized: loop  
vectorized using 32 byte vectors
```

Auto-vectorized

Terminal output

- Modern compilers offer detailed feedback about optimizations.
- Look at compiler logs (such as opt.log) to see:
  - Why functions might not be inlined.
  - Which loops were not vectorized.
  - Potential data dependency issues.
- This feedback helps guide your optimization efforts.

# Common Optimization Pitfalls

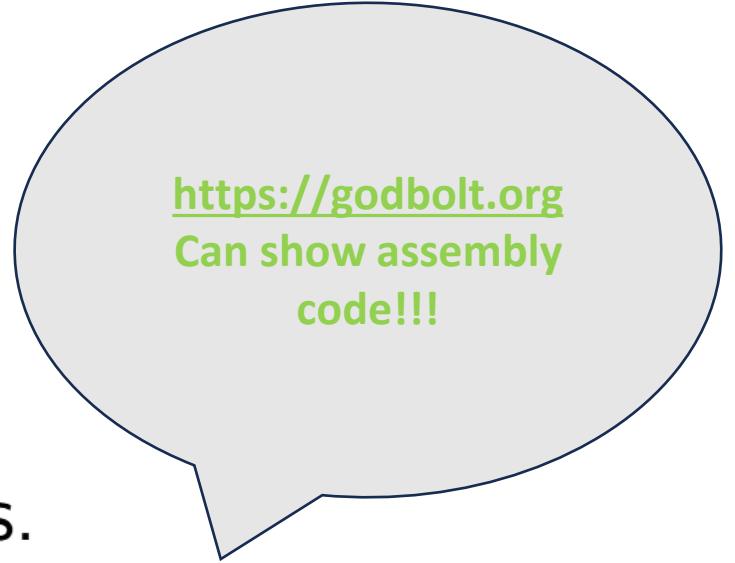
- **Function Call Overhead:** Frequent function calls can slow down your code.
- **Data Dependencies:** Loops with dependencies between iterations are hard to parallelize.
- **Memory Access Patterns:** Non-sequential accesses may hurt cache performance.
- **Race Conditions:** Unprotected concurrent writes can lead to unpredictable results.

# Basic Optimization Techniques

- **Inlining:** Use inline functions to reduce the overhead of function calls.
- **Vectorization:** Modern compilers often autovectorize simple loops.
- **Loop Unrolling:** Manually unroll loops or let the compiler optimize them.
- **Parallelization:** Use OpenMP to exploit multiple cores (but be cautious of race conditions).
- **Memory Alignment:** Align data structures for better cache utilization.

# A Real-World Optimization Journey

- Start with a simple, working code example.
- Measure performance and identify bottlenecks.
- Incrementally add optimizations (e.g., pragmas, inlining).
- Use compiler feedback and tools (like assembly inspection) to verify improvements.
- Iterate and test thoroughly to ensure correctness.



<https://godbolt.org>  
Can show assembly  
code!!!

# Conclusion

- Performance optimization is a gradual, iterative process.
- Rely on compiler tools and logs to guide your changes.
- Focus on high-level parallelization while letting modern compilers handle low-level vectorization.
- Always test your code for correctness, especially when introducing parallelism.

# Important Optimization Flags

(use appropriate flags for “Debug” and “Release” configurations)

## *Some of below ones GCC specific*

- -O3: Enables high-level optimizations.
- -march=native: Generates code optimized for your local CPU.
- -ffast-math and -fno-math-errno: Aggressive math optimizations ignoring strict IEEE rules. May ignore special values like INF and NAN.
- **-fopenmp**: Enables OpenMP support.
- -ftree-vectorize: Allows automatic loop vectorization.
- -fopt-info-vec-all=opt.log: Writes detailed vectorization info to opt.log. Check it to see which loops were vectorized and why others were not.

Required for OpenMP

# OpenMP

- OpenMP is based on two concepts: the use of threads and the fork/join model of parallelism.
- The fork/join model says that a thread can split itself ('fork') into a number of threads that are identical copies.
- At some point these copies go away and the original thread is left ('join'), but while the team of threads created by the fork exists, you have parallelism available to you. The part of the execution between fork and join is known as a parallel region.

# OpenMP

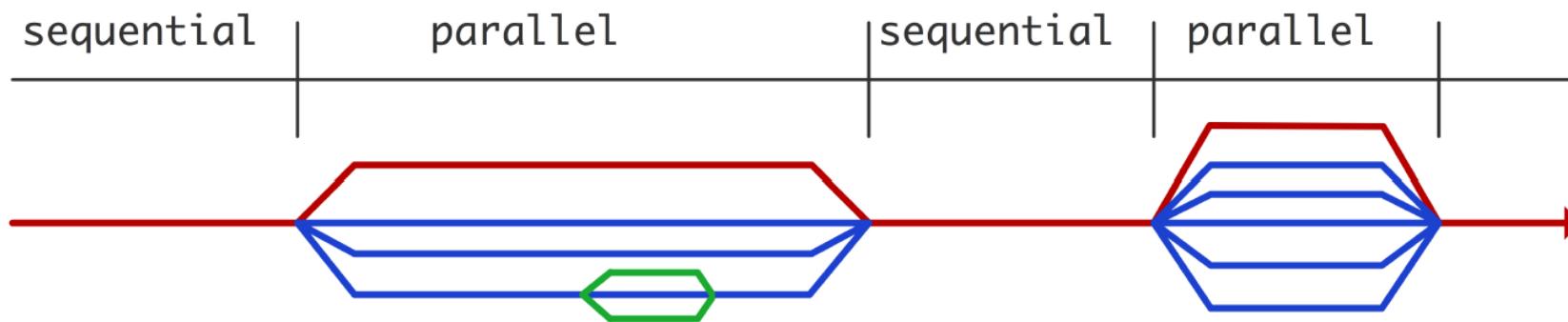


Figure 17.3: Thread creation and deletion during parallel execution

# OpenMP

- The threads that are forked are all copies of the master thread: they have access to all that was computed so far; this is their shared data.
- Of course, if the threads were completely identical the parallelism would be pointless, so they also have private data, and they can identify themselves: they know their thread number.
- This allows you to do meaningful parallel computations with threads.

# OpenMP

- OpenMP programming is typically done to take advantage of multicore processors.
- Thus, to get a good speedup you would typically let your number of threads be equal to the number of cores.
- However, there is nothing to prevent you from creating more threads: the operating system will use time slicing to let them all be executed. You just don't get a speedup beyond the number of actually available cores.
- On some modern processors there are hardware threads, meaning that a core can actually let more than one thread be executed, with some speedup over the single thread. To use such a processor efficiently you would let the number of OpenMP threads be (typically) 2 times the number of cores.

# OpenMP

- In C, you can redeclare a variable inside a nested scope:

```
{  
    int x;  
    if (something) {  
        double x; // same name, different entity  
    }  
    x = ... // this refers to the integer again  
}
```

- In OpenMP the situation is a bit more tricky because of the threads. When a team of threads is created they can all see the data of the master thread.
- However, they can also create data of their own. We will go into the details later

OpenMP has a similar mechanism:

```
{  
    int x;  
#pragma omp parallel  
    {  
        double x;  
    }  
}
```

There is an important difference: each thread in the team gets its own instance of the enclosed variable.

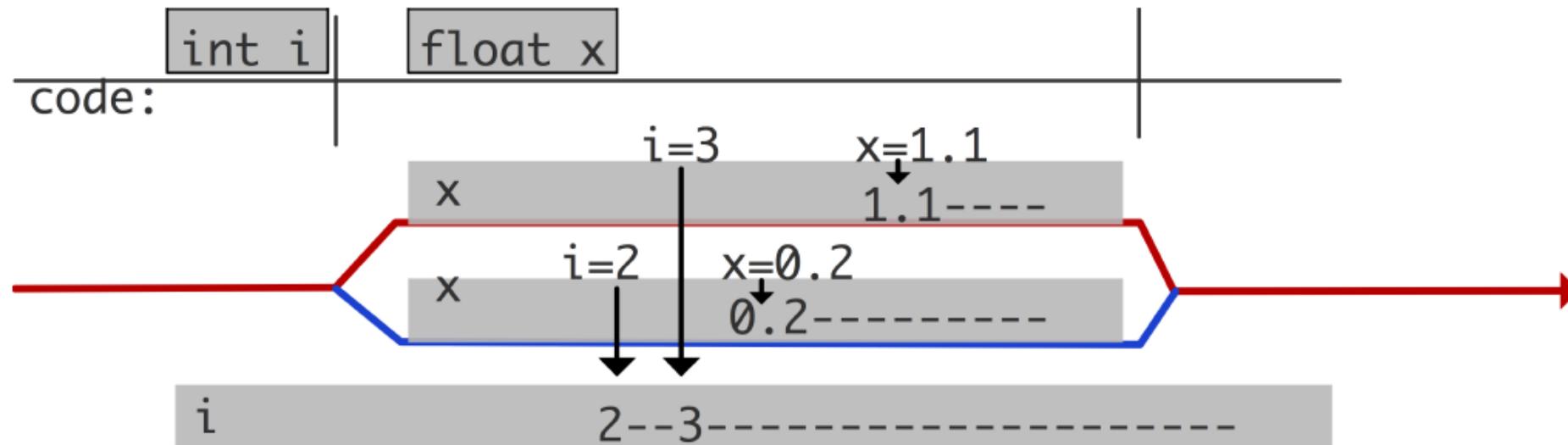


Figure 17.4: Locality of variables in threads

# OpenMP

- Your file needs to include `#include "omp.h"`
- On gcc compiler you compile program using OpenMP with `gcc foo.c -fopenmp`
- Without using “`-fopenmp`” your OpenMP program will run only single thread (meaning no gain!) or give run time error
- In MATLAB MEX, you can use these compilation options:

```
mex -R2018a CFLAGS="$CFLAGS -fopenmp" LDFLAGS="$LDFLAGS -fopenmp" foo.c
```

# OpenMP

- In MATLAB / MEX you can verify that file has been compiled properly with OpenMP by using

```
#ifndef _OPENMP  
    mexErrMsgTxt("Incorrect compile options! Compile with OpenMP  
support!");  
#endif
```

at the beginning of your MATLAB\_main()

- The value of this `_OPENMP` is a decimal value yyyymm denoting the OpenMP standard release that this compiler supports.

# OpenMP

- The simplest way to create parallelism in OpenMP is to use the parallel pragma. A block preceded by the `omp parallel` pragma is called a parallel region; it is executed by a newly created team of threads. This is an instance of the Single Program Multiple Data (SPMD) model: all threads execute (redundantly) the same segment of code.

```
#pragma omp parallel
{
    // this is executed by a team of threads
}
```

# OpenMP

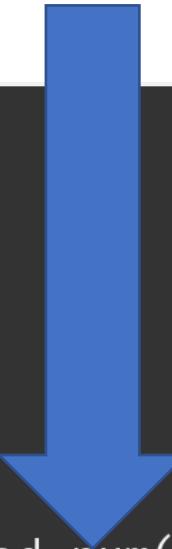
- Immediately preceding the parallel block, one thread will be executing the code. In the main program this is the *initial thread*.
- At the start of the block, a new *team of threads* is created, and the thread that was active before the block becomes the *master thread* of that team.
- After the block only the master thread is active.
- Inside the block there is team of threads: each thread in the team executes the body of the block, and it will have access to all variables of the surrounding environment. How many threads there are can be determined in a number of ways; we will get to that later.

# OpenMP

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

int main(void)
{
    printf("This is only run one (outside omp parallel)\n");
    #pragma omp parallel
    {
        printf("Hello World from thread %d\n",omp_get_thread_num());
    }
    printf("Parallel region has finished!\n");
}
```

**omp\_get\_thread\_num** reports the number of the thread that makes the call.



OMP\_NUM\_THREADS=4

```
This is only run one (outside omp parallel)
Hello World from thread 1
Hello World from thread 2
Hello World from thread 0
Hello World from thread 3
Parallel region has finished!
```

# OpenMP

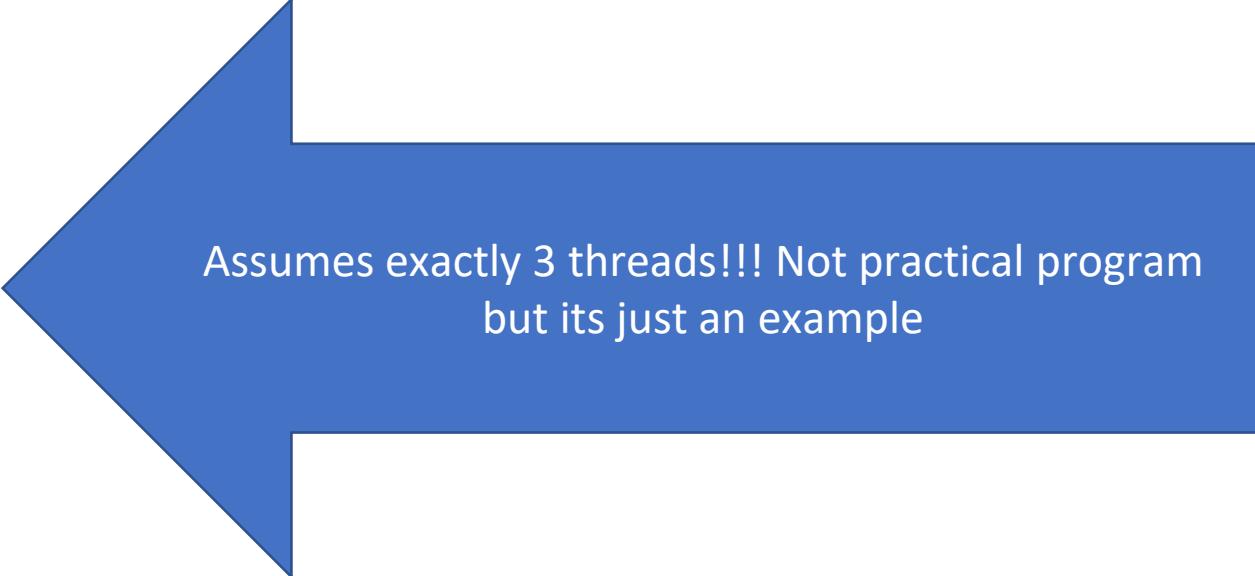
It would be pointless to have the block be executed identically by all threads. One way to get a meaningful parallel code is to use the function `omp_get_thread_num`, to find out which thread you are, and execute work that is individual to that thread. There is also a function `omp_get_num_threads` to find out the total number of threads. Both these functions give a number relative to the current team; recall from figure 17.3 that new teams can be created recursively.

For instance, if you program computes

```
|| result = f(x)+g(x)+h(x)
```

you could parallelize this as

```
double result,fresult,gresult,hresult;
#pragma omp parallel
{ int num = omp_get_thread_num();
  if (num==0)      fresult = f(x);
  else if (num==1) gresult = g(x);
  else if (num==2) hresult = h(x);
}
result = fresult + gresult + hresult;
```



Assumes exactly 3 threads!!! Not practical program  
but its just an example

# OpenMP – Erroneous Code (race condition)!

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

int main(void)
{
    int tsum = 0, temp;
    #pragma omp parallel
    {
        temp = omp_get_thread_num();
        tsum += temp;
    }
    printf("Sum is %d\n", tsum);
}
```

Sum is 496

janne@pop-os:~/c\_course\$ ./a.out

Sum is 483

janne@pop-os:~/c\_course\$ ./a.out

Sum is 496



Result  
varies!!

Multiple thread try to update at the same time shared variable!  
Also, temp not guaranteed to be from this thread!

Many of the difficulties of parallel programming with OpenMP stem from the use of shared variables. For instance, if two threads update a shared variable, you not guarantee an order on the updates.

# OpenMP – Erroneous Code (race condition)!

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

int main(void)
{
    int tsum = 0, temp;
    #pragma omp parallel
    {
        temp = omp_get_thread_num();
        tsum += temp;
    }
    printf("Sum is %d\n", tsum);
}
```

temp not guaranteed to be from  
this thread!

Thread 1 can modify temp

But before it can update tsum,

Thread 2 can modify temp

Then thread1 will add wrong temp  
to tsum!!!

# OpenMP – Erroneous Code (race condition)!

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

int main(void)
{
    int tsum = 0, temp;
    #pragma omp parallel
    {
        temp = omp_get_thread_num();
        tsum += temp;
    }
    printf("Sum is %d\n", tsum);
}
```

Multiple thread try to update at the same time shared variable!

$tsum = tsum + temp$

What if read current value of  $tsum$  but before we update  $tsum$  another thread already updated  $tsum$ !

Then we will wrongly update  $tsum$  based on its old value!

# OpenMP – Erroneous Code (race condition)!

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0, temp;
    #pragma omp parallel private(temp)
    {
        temp = omp_get_thread_num();
        printf("Temp is before %d\n", temp);
        tsum += temp;
        printf("Temp is after %d\n", temp);
    }
    printf("Sum is %d\n", tsum);
}
```

Temp is now private variable; each thread has own copy of it!!!! [without private clause temp would be shared!]

Now temp guaranteed to be the same in this thread!  
However, tsum update can still happen same time with wrong results!

# OpenMP – Erroneous Code (race condition)!

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0;
    #pragma omp parallel
    {
        int temp;
        temp = omp_get_thread_num();
        tsum += temp;
    }
    printf("Sum is %d\n", tsum);
}
```

Temp is now private variable; each thread has own copy of it!!!! [local variables are automatically private]

Now temp guaranteed to be the same in this thread!  
However, tsum update can still happen same time with wrong results!

# OpenMP – Correct Code (no race condition)!

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0;
    #pragma omp parallel
    {
        int temp;
        temp = omp_get_thread_num();
        #pragma omp critical
        tsum += temp;
    }
    printf("Sum is %d\n", tsum);
}
```

Temp is private variable; each thread has own copy of it!!!! [local variables are automatically private]

- critical: following section of code can only be executed by one thread at a time;
- Do not use #critical inside for loops (high performance penalty)

- **Critical vs. Atomic:**

- For a single integer increment, `#pragma omp atomic` is typically faster than `#pragma omp critical` because it reduces synchronization overhead.
- Reduction (for “for loops”) is much better way [to be discussed next]

# OpenMP

- Loop parallelism is a very common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP ‘worksharing’ constructs (see section 21 for the full list): constructs that take an amount of work and distribute it over the available threads in a parallel region, created with the parallel pragma.
- The parallel execution of a loop can be handled a number of different ways. For instance, you can create a parallel region around the loop, and adjust the loop bounds manually.

# OpenMP

- A natural option is to use the for pragma

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
    // do something with i
}
```

- This has several advantages. For one, you don't have to calculate the loop bounds for the threads yourself.
- Without #pragma omp for, each thread would run full for loop!!!
- With #pragma omp for, each thread automatically gets its own portion of the loop indices!!!

**Remark 28** *The loop index needs to be an integer value for the loop to be parallelizable.*

# OpenMP

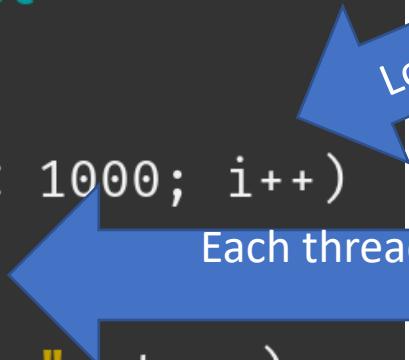
- Note that the for pragma does not create a team of threads: it takes the team of threads that is active, and divide the loop iterations over them.
- This means that the omp for directive needs to be inside a parallel region.
- It is also possible to have a combined omp parallel for directive

```
#pragma omp parallel for  
for (i=0; .....
```

# OpenMP – erroneous code (race condition!)

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0;
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 1000; i++)
            tsum += i;
    }
    printf("tsum is %d\n", tsum);
}
```

Loop variable "i" is automatically private!



Each thread can try to update tsum at the same time!!!  
(tsum is shared variable)

# OpenMP – correct but slow code

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0;
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 1000; i++)
            #pragma omp critical
            tsum += i;
    }
    printf("tsum = %d\n", tsum);
}
```

Only one thread will update tsum at a time => correct results

# OpenMP – correct code (easy and fast way)

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0;
    #pragma omp parallel
    {
        #pragma omp for reduction(+:tsum)
        for (int i = 0; i < 1000; i++)
            tsum += i;
    }
    printf("tsum is %d\n", tsum);
}
```

By using reduction we can tell OpenMP that  
tsum is being modified inside loop using “+”  
operation!!!

# OpenMP – correct code (another way)

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    int tsum = 0;
    #pragma omp parallel
    {
        int temp = 0;
        #pragma omp for
        for (int i = 0; i < 1000; i++)
            temp += i;

        #pragma omp critical
        tsum += temp;
    }
    printf("tsum is %d\n", tsum);
}
```

Temp is private variable, unique for each thread

#critical is outside for loop! No performance penalty!  
We sum local temp sums to the final tsum  
Each local sum is partial sum!

# OpenMP

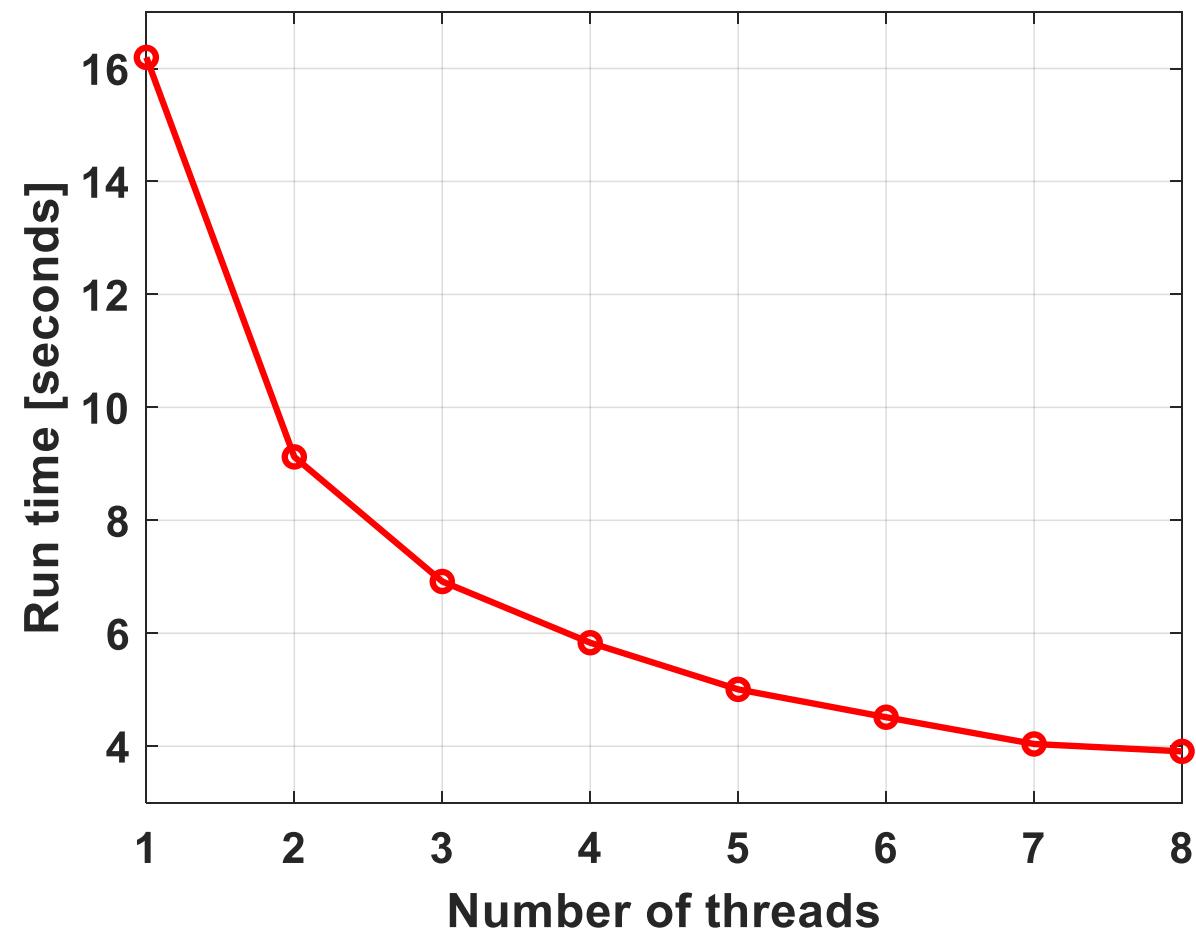
There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

- The loop can not contains break, return, exit statements, or goto to a label outside the loop.
- The continue (C) or cycle (F) statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not changes to it inside the loop are allowed.

```

// mex -R2018a CFLAGS="$CFLAGS -fopenmp" LDFLAGS="$LDFLAGS -fopenmp"
example_vector_input_one_output.c
#define NEXTRA_PARAMETERS 1
#include "VECTOR_INPUT_ONE_OUTPUT_REAL.h"
// After mex compilation, call from MATLAB for example with:
// out = example_vector_input_one_output([1 2 3 4 5],2)
double MATLAB_main(double *input, size_t numb_elements, double param1)
{
    double temp = 0;
#pragma omp parallel
    {
#pragma omp for reduction (+:temp)
        for(int c = 0; c < numb_elements; c++)
            temp += pow(input[c],param1);
    }
    return temp;
}

```



## 20.1.2 Reduction clause

The easiest way to effect a reduction is of course to use the reduction clause. Adding this to an `omp parallel` region has the following effect:

- OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
- Each thread will then reduce into its local variable;
- At the end of the parallel region, the local results are combined, again using the reduction operator, into the global variable.

```
// reductpar.c
m = INT_MIN;
#pragma omp parallel reduction(max:m) num_threads(ndata)
{
    int t = omp_get_thread_num();
    int d = data[t];
    m = d>m ? d : m;
}
```

## 20.2 Built-in reduction operators

Arithmetic reductions: `+`, `*`, `-`, `max`, `min`.

Logical operator reductions in C: `&` `&&` `|` `||` `^`

If you want to reduce multiple variables with the same operator, use

```
|| reduction(+:x,y,z)
```

For multiple reduction with different operators, use more than one clause.

# Single

The `single` and `master` pragma limit the execution of a block to a single thread. This can for instance be used to print tracing information or doing *I/O* operations.

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

int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Single time run!\n");
        printf("Run multiple times!\n");

    }
}
```

# OpenMP

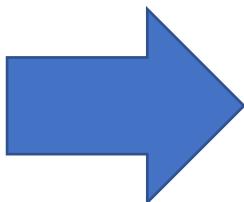
Output varies due to race condition

## 22.1 Shared data

In a parallel region, any data declared outside it will be shared: any thread using a variable `x` will access the same memory location associated with that variable.

Example:

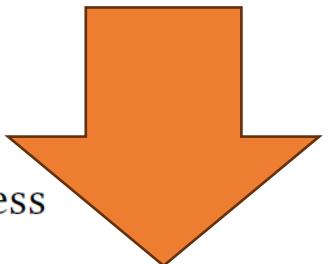
```
int x = 5;
#pragma omp parallel
{
    x = x+1;
    printf("shared: x is %d\n",x);
}
```



```
janne@pop-os:~/lect8_progs$ ./a.out
Final x = 1028
janne@pop-os:~/lect8_progs$ ./a.out
Final x = 1026
janne@pop-os:~/lect8_progs$ ./a.out
Final x = 1028
janne@pop-os:~/lect8_progs$ ./a.out
Final x = 1027
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved. See HPSC-2.6.1.5 for an explanation of the issues involved; see 23.2.2 for a solution in OpenMP.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.



# Private data

The `private` directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

```
int x = 5;
#pragma omp parallel private(x)
{
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
}
printf("after: x is %d\n",x); // also dangerous
```

All variables defined inside parallel region are by default private

# Private data

Private variables are uninitialized! Its just “luck”  
they are zero (then  $x = x + 1$  leads to 1)  
After parallel section  $x$  reverts to original value!

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

int main(void)
{
    int x = 5;
    #pragma omp parallel private(x)
    {
        x = x + 1;
        printf("Private: x is %d\n", x);
    }
    printf("After: x = %d\n", x);
}
```

# OpenMP

## 23.1 Barrier

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of y can not proceed until another thread has computed its value of x.

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a barrier pragma:

```
#pragma omp parallel
{
    int mytid = omp_get_thread_num();
    x[mytid] = some_calculation();
    #pragma omp barrier
    y[mytid] = x[mytid]+x[mytid+1];
}
```

# OpenMP

Apart from the barrier directive, which inserts an explicit barrier, OpenMP has *implicit barriers* after a load sharing construct. Thus the following code is well defined:

```
#pragma omp parallel
{
#pragma omp for
    for (int mytid=0; mytid<number_of_threads; mytid++)
        x[mytid] = some_calculation();
#pragma omp for
    for (int mytid=0; mytid<number_of_threads-1; mytid++)
        y[mytid] = x[mytid]+x[mytid+1];
}
```

# OpenMP

## 23.1.1 Implicit barriers

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*.

There is some *barrier behavior* associated with `omp for` loops and other *worksharing constructs* (see section 21.3). For instance, there is an *implicit barrier* at the end of the loop. This barrier behavior can be cancelled with the `nowait` clause.

You will often see the idiom

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i=0; i<N; i++)
            a[i] = // some expression
    #pragma omp for
        for (i=0; i<N; i++)
            b[i] = ..... a[i] .....
```

Here the `nowait` clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses `a[i]` can indeed rely on it that that value has been computed.

# OpenMP

- Here is implicit barrier between for loops. Second for loop will only run once the first has finished.

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 4; i++)
            printf("Hello from first for loop!\n");
        #pragma omp for
        for (int i = 0; i < 4; i++)
            printf("Hello from second for loop!\n");
    }
}
```

# OpenMP

- Here is implicit barrier between for loops has been removed with “nowait”. The two printf's will print in random order.

```
#include <stdio.h>
#include "omp.h"
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (int i = 0; i < 4; i++)
            printf("Hello from first for loop!\n");
        #pragma omp for
        for (int i = 0; i < 4; i++)
            printf("Hello from second for loop!\n");
    }
}
```

# OpenMP

- Here reduction clause is not need since each thread will touch different part of output vector.

```
// mex -R2018a CFLAGS="$CFLAGS -fopenmp -std=c11" LDFLAGS="$LDFLAGS -fopenmp" COPTIMFLAGS=O3" example_vector_input_vector_output.c
#define NEXTRA_PARAMETERS 1 // CAN BE ZERO!
#include "VECTOR_INPUT_VECTOR_OUTPUT_REAL.h"
// After mex compilation, call from MATLAB for example with:
// out = example_vector_input_vector_output([1 2 3 4 5],3)
void MATLAB_main(double* output, double* input, size_t numb_elements, double param1)
{
    #pragma omp parallel for
    for(int c = 0; c < numb_elements; c++)
        output[c] = input[c] + param1;
}
```



Here calculation is very simple  
(just "+") so OpenMP speed up is  
less than 2x

TBD: Cache optimization effect?  
TBD: Compiler optimization flag  
effect?

# AI-Assisted Programming

## Modern Topics in Telecommunications and Radio Engineering 6

### Using C with MATLAB 521196S<sup>1</sup>

Janne Lehtomäki

University of Oulu

March 5, 2025

---

<sup>1</sup>Various AI tools such as from OpenAI, Google, Anthropic, and xAI were used to help in preparing the slides. There is no single best model!

# Presentation Roadmap

- 1 What can AI do?
- 2 Understanding AI Models
- 3 AI for Coding vs. General Chat
- 4 The Importance of Fundamentals
- 5 Near-Future Developments
- 6 Conclusion

# AI in Action: Example Scenarios

## Practical Examples of AI in Programming

- **Auto-Completion:** Suggesting entire lines or blocks of code as you type.
- **Comment-Based Generation:** Creating a function just by describing it in a comment.
- **Chat-Based Prompting:** Posing questions such as “Optimize this function” to an AI and getting solutions.
- **Cross-File Assistance:** Analyzing multiple files to ensure consistency across a whole project. AI can suggest edits to multiple files at the same time. Available for example in GitHub Copilot.

# Why Use AI for Programming?

## Benefits of AI in Programming

- **Faster Development:** AI handles boilerplate, so you can focus on high-level logic. Offload repetitive tasks to AI, freeing up mental space for innovation.
- **Higher Code Quality:** This is your responsibility but a set of second eyes always helps!
- **Enhanced Learning:** Receive real-time feedback and examples while you code. But beginners should first solve the problem themselves, before using AI.
- **Learn New Programming Language:** Great for assisting in learning new programming languages such as C++ (recommended to also read the best book and/or best web resources on the topic for true learning)
- When something (such as one time script in Python) needs to be quickly done in some unfamiliar language.

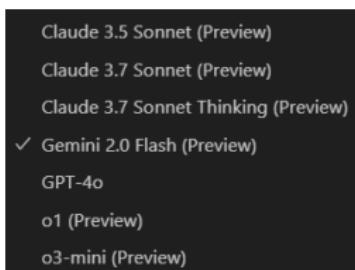
# Key AI Models in Programming

## Popular AI Models for Coding

- **Claude (Anthropic)**: Known for substantial context capacity and strong code synthesis.
- **Gemini (Google)**: Prioritizes responsiveness; excels at multi-language integration.
- **OpenAI Models** (e.g., GPT-4o, o1, o3-mini):
  - Varying strengths in logic, speed, and detail.
- **Grok 3 (xAI)**: Launched in 2025, designed to be competitive with top models, featuring advanced reasoning, a 1 million token context window, and strong coding performance.
- **DeepSeek-R1** (can be used with Azure AI Foundry, the main website is currently blocked in University of Oulu). Can be very good for creative tasks, such making a drawing using LaTeX (even better than o1). Good for coding. Achieved breakthrough in cost effectiveness.

# GitHub Copilot: Your AI Assistant

- Can propose entire blocks of code or entire functions.
- Both auto-completion and Chatting.
- Integrates directly into e.g. Visual Studio Code (install extensions: GitHub Copilot Chat and the basic GitHub Copilot).
- **Free for University students** (requires proof of status, for example screenshot from Tuudo student card might suffice).



**Figure:** Currently available models in GitHub Copilot

- **Copilot can Use various AI Models:** Some models are tuned for speed, others for deeper reasoning or large context handling. There is no always-best model. Experiment!

# Claude (Anthropic)

## Claude 3.7 Sonnet / Sonnet Thinking

- Released on February 24, 2025.
- Enhances logic and multi-step problem solving
- The “Thinking” variant is more thorough but slower
- User can choose fast responses or step-by-step reasoning (hybrid AI model).

# Gemini 2.0 Flash (Google)

## Gemini 2.0 Flash (Google)

- Released in January 30 2025
- Popular model for developers, with enhanced performance at fast response times
- Extreme context window (up to 1 million tokens).

# OpenAI Models in Copilot

## OpenAI Models in Copilot

- Built on top of GPT architectures, known for strong reasoning and language understanding.

## GPT-4o

- Robust general knowledge and detailed explanations

## o1 (OpenAI reasoning model)

- Excellent chain-of-thought reasoning
- Handy for debugging or multi-step logic

## o3-mini / o3-mini-high reasoning models

- Released January 31, 2025
- Great for coding.
- **o3-mini-high** has demonstrated better performance than full o1 (performance varies).

# Understanding Tokens and Context Window

## What Are Tokens?

- Small textual chunks (words, parts of words, symbols) used by AI models.
- “I love programming” **could** tokenize into ["I", "love", "program", "ming"].

## What Is the Context Window?

- The amount of text the model can “see” at once during generation.
- Larger windows let the model reference more of your code, increasing relevancy.

# Why Does Context Window Size Matter?

## Advantages of a Large Context Window

- **Holistic Understanding:** Tracks the entire codebase or lengthy functions.
- **Better Relevance:** AI suggestions remain consistent across files or modules.
- **Fewer Oversights:** Reduces trivial mistakes that arise from missing context.
- **Consistent Coding Style:** Maintains uniform naming, formatting, and logic patterns.
- But also, more costly (if you have to get API access).

## OpenAI Canvas Model

- Single interactive window shared by the programmer and the AI.
- Both parties collaborate in real time on the same code snippet.
- Promotes a more direct iterative approach: fewer context switches.
- Slightly similar to experience you would get from IDE with AI.

## Why It Matters

- Minimizes versioning friction.
- Maintains a single, always-updated code source.
- Streamlines quick prototyping and debugging.

## GPT-5 (Subject to Change)

- (Background): GPT-4.5 is the latest model available to Pro-users [not available in Plus] (as a research preview). Normal users still use GPT-4o or one of the reasoning models.
- Rumored to be a **unified model**, merging reasoning and generation seamlessly.
- May incorporate advanced features from the o3 lineage (full o3 might never be released as a separate product).
- Release timeline unknown; details remain speculative.
- **Key Point:** In the future, specialized “reasoning” or “speed” variants might be replaced by a single comprehensive model.

# Coding-Focused vs. General Chat AI

## Coding-Focused AI in GitHub Copilot

- IDE integration gives great benefits.
- Gives short, targeted suggestions (developer prompts given to it might vary, sometimes responses are long, especially for models under Preview)
- Basic models are the same or similar to general-purpose models (sometimes extra training data has been used).

## General-Purpose external AI (e.g., ChatGPT)

- Excellent for conceptual explanations or brainstorming.
- Provides more verbose, conversational answers.
- Requires manual copy-paste of code context.
- Great for generic programming questions/chat.

# You Still Need to Know C

- **Memory & Pointers:** AI might overlook pointer arithmetic quirks or produce unsafe code.
- **Debugging Skills:** AI suggestions can help, but diagnosing real errors remains your job.
- **Optimization:** AI typically offers generic code; fine-tuning performance can be done by iterative process (such as asking for generation of intrinsics). Without deep knowledge, tedious trial-and-error iterative can be the result even with AI.
- **Professional Growth:** Solid language fundamentals are essential for a robust career.

# Limitations & Pitfalls: Real-World Examples

## Even the Best AI Can Make Mistakes!

- **Hallucinations (Inventing Things):** AI might suggest using functions or libraries that don't actually exist.
- **Security Risks (Unsafe Code):** AI might generate code with buffer overflows, format string vulnerabilities, or other security flaws.
- **Misleading Confidence (Logically Flawed Code):** The AI-generated code might compile and run, but produce incorrect results due to logical errors. *Example:* An AI might generate a sorting algorithm that doesn't correctly handle duplicate values.
- **C Style Errors (Empty Parenthesis):** Even top-tier AI models can make basic C syntax errors, such as declaring a function prototype as `void print_message()` instead of `void print_message(void)`, which has different meanings in C.
- **Verification Is Key (Always Test!):** Never blindly trust AI-generated code. Always test it thoroughly with different inputs and edge cases to ensure it works correctly. Iteration is usually needed.

# What to Expect in AI-Driven Development

- **Ultra-Large Contexts:** Models capable of parsing entire repositories (Gemini 2.0 and Grok 3 already have huge context windows).
- **Persistent Memory:** Remembering a project's evolution across multiple sessions.
- **Automated Debugging:** AI able to run test suites and propose targeted fixes.
- **"Agent" Mode:** Already available for testing in GitHub Copilot (February, 2025, preview)<sup>2</sup> "capable of iterating on its own code, recognizing errors, and fixing them automatically. It can suggest terminal commands and ask you to execute them"
- **Tighter Tool Integration:** Working closely with compilers, linters, and documentation systems.
- **Unified Models (GPT-5?)**: A single advanced model that balances speed and depth.

---

<sup>2</sup><https://github.blog/news-insights/product-news/github-copilot-the-agent-awakens/>

## Key Takeaways

- AI can significantly boost productivity for C and other languages.
- Different **models** suit different use cases: some excel at reasoning, others at speed.
- Always **verify** outputs: your testing and review are essential.
- Future AI developments promise **bigger context windows** and deeper debugging support.
- **Shift in Focus:** Developers move from rote coding to validation, design, and creative problem-solving.
- Solid **language fundamentals** remain indispensable. Industry expects actual knowledge, not just prompt engineering without actual programming knowledge.
- Embrace AI as a **collaborative partner**, not a silver bullet.