



A complex background featuring a blue and white abstract wave pattern. Overlaid on the left side is a grid of binary code (0s and 1s). A grey bar across the middle contains a silver power button icon on the left and the word "Introduction" in white text on the right.



## Introduction

1

## HISTORY, BASIC SYNTAX

2

A complex background featuring a light grey and white abstract wave pattern. In the center is a black and white portrait of Dennis Ritchie, a man with a beard and glasses. To the left of the portrait is a bulleted list of facts about the C programming language.

### What is C?



- ➊ C is a *general-purpose* programming language initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs
- ➋ It is an imperative procedural language intended for system software
  - Strong ties with UNIX operating system
- ➌ It has influenced many other programming languages
  - C++, C#, ObjC, Java, JavaScript, Go, csh, ...

3



### Why C?

- ➊ It's still popular and widely used
  - Available on almost all platforms
  - Lots of libraries for different tasks
- ➋ Provides a relatively low-level (or mid-level) access to the operating system and hardware
  - System-level programming, embedded systems
  - Can lead to better performance

4

## Standards



- First standard by ANSI in 1989, known as “ANSI C” or C89
  - Still the best choice when portability is important!
  - ISO adopted the same standard in 1990 (C90)
- Next revision in 1999, C99
  - New datatypes, complex numbers, variable length arrays,...
  - Not fully backwards compatible with C90
- Current standard is C11
  - Improved Unicode support, atomic operations, multi-threading, bounds-checked functions, ...

5

## Look and feel



```
/* Computing the factorial of an specified (by the user) number */
#include <stdio.h>
int fact(int n);
int main(void){
    int current;
    printf("Enter some POSITIVE integer (non-positive to finish): ");
    scanf("%d", &current);
    while (current > 0) {
        printf("Factorial of %d is %d\n", current, fact(current));
        printf("Enter a POSITIVE integer (non-positive to finish): ");
        scanf("%d", &current);
    }
    return 0;
}

/* This function returns the factorial of the specified integer (n) */
int fact(int n) {
    ...
}
```

6

## Basic syntax



- Code is case sensitive
- Statements terminate with ;
- { } enclose blocks
- There are two ways to comment:

```
/* Single or
   Multiple lines */

// Single line (C99)
```

```
/* example function */
float foo(int bar) {
    int i, c = 0;
    float x;

    x = 0.1;
    for (i = 0; i < bar; i++) {
        x += i*3.14 + c;
        c = i + 2;
    }
    return x;
}
```

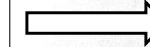
7

- Free format

– Whitespace, newlines, layout etc. do not matter ...

to the computer !

```
/* same example function? */
float
foo(/*float f,*/int
bar) { int i,c=0; float x;
x=0.1;for (/*i=10
;i<2*bar*/ i=
0;i<bar;i++){x+=i*3.14*i*1.2+
*/4+c;c=i+2;}return x; }
```



```
/* example function */
float foo(int bar) {
    int i, c=0;
    float x;

    x = 0.1;
    for (i=0; i<bar; i++) {
        x += i*3.14 + c;
        c = i + 2;
    }
    return x;
}
```

8

### DATATYPES, VARIABLES, ASSIGNMENT AND OPERATORS

9

- Basic datatypes in C are:

<b>char</b>	character
<b>int</b>	integer number
<b>long</b>	long integer number
<b>float</b>	floating point number
<b>double</b>	double precision float

- Integers can be signed (default) or **unsigned**
- C has also *pointer* types
- There is also a special type **void**

10

### Variables

- Variable data types are static
- Declare variables before using them
  - C90 requires that the declarations are before any other statements
- Valid variable names in C:
  - Upper and lower case letters, underline and numbers are allowed

```
// variable declaration
int i;
float f, g;
double total = 1.9;

// valid names
int i3, myINT, I_o;
float o3Gf_ry9;

// invalid names
int 33, 9a, i-o;
float o3.Gf;

// data type matters
char c;
float f;

f = 1.234;
c = f; // ERROR!
        // wrong data type
```

11

### Variable assignment

- Assign a value to a variable:  
**variable = value;**
- Both should have the same data type
  - Implicit conversion between compatible types
  - Explicit conversion (typecast)
   
syntax:  
**(type) var**

```
// examples of assignment
count = 10;
i = j = 0;
k = i*j + 1;

// assign at declaration
int i = 4;

// typecast from int to float
int i;
float f;

i = 5 * 21;
f = (float) i;

// watch out for operator order
(float) i/5 != (float) (i/5)
```

12

## Arithmetics



- Operators:
  - +
  - 
  - \*
  - /
  - %

### Operator precedence

- 1) % \* /
- 2) + -

- Parentheses can be used to group operations () and change evaluation order

13

## Compound assignment



- C has a short-hand notation for combined arithmetic operation and assignment
- Given an operator <op> and values **a** and **b**, the syntax is **a<op>=b** and the result is **a=a<op>b**
- For special cases **a+=1** and **a-=1** there are special operators **++** and **--**

```
// example of compound assignment
int count = 10;
count += 5; // now count is 15
count /= 3; // and now count is 5

// Adding one to count
count++;

// This is also valid!
++count;
```

14

## Arithmetics and assignment examples

```
// addition, subtraction
i = 5 + 2;
i = 5 - 2;
i += 1;      // i = i + 1
f -= 3.4;
i++;        // i = i + 1
i--;

// multiplication, division
i = 5 * 2;
i = 5 / 2;    // integer division
f *= 3.0 / 4.2;

// modulus
m = 25 % 3;
```

```
// grouping with ()
b = a * (1.3 + (25%3));

// watch out for precedence!
f = 1 + q / (1 - q);
f = (1 + q) / (1 - q);

// also function calls use ()
f = r * sin(theta);
f = (r + p)*sin(theta);
```

15

## Other operators



### Logical operators

&&	AND
	OR
>	larger than
<	less than
>=	larger or equal
<=	less or equal
==	equal
!=	unequal
!	negation

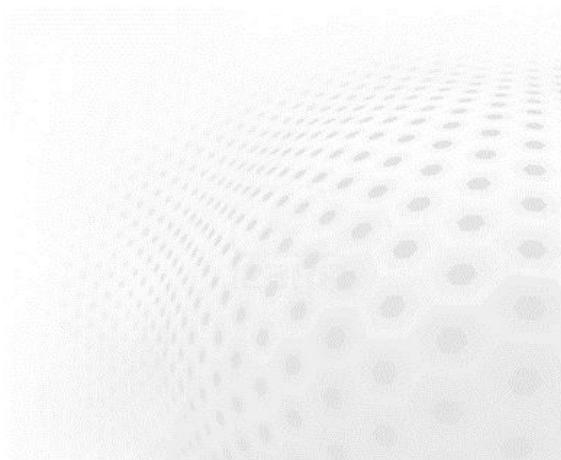
### Bit-wise operators

>>	shift right
<<	shift left
&	AND
	OR
^	XOR
~	complement

16



## BASIC I/O



17

## Basic I/O



- ➊ **printf** prints formatted text to the screen

- ➋ Format of **printf** call is

```
printf(<template> {, <variables>})
```

where

<**template**> arbitrary string with optional  
*placeholders* for data from variables  
<**variables**> (optional) list of variables or values

## Basic I/O



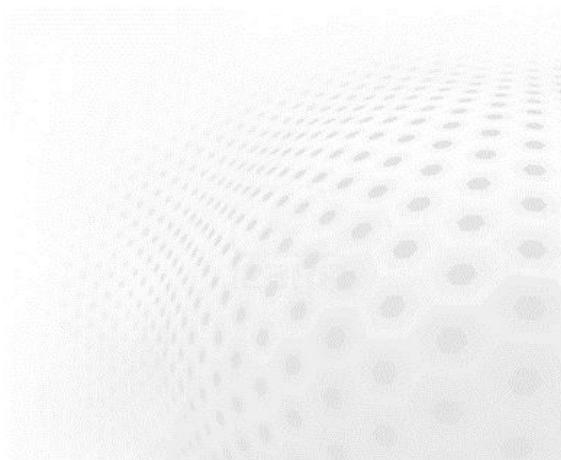
- ➊ Special characters that can be used in the template string:

\n	newline	\t	horizontal tab
\"	double quote	\\	literal backslash

- ➋ Placeholders are marked with % followed by formatting and type information

- For now, we will just use the following formattings:

%d	integer value
%f	float value
%s	string



18

## Basic I/O example



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

```
int main(int argc, char *argv[]) {
    printf("The answer is %d.\n", 42);
    printf("Pi equals to %.2f", 3.14159265);
    printf(" ...at least to the %ndnd decimal.\n", 2);
    return 0;
}

//output:
// The answer is 42.
// Pi equals to 3.14 ...at least to the 2nd decimal.
```

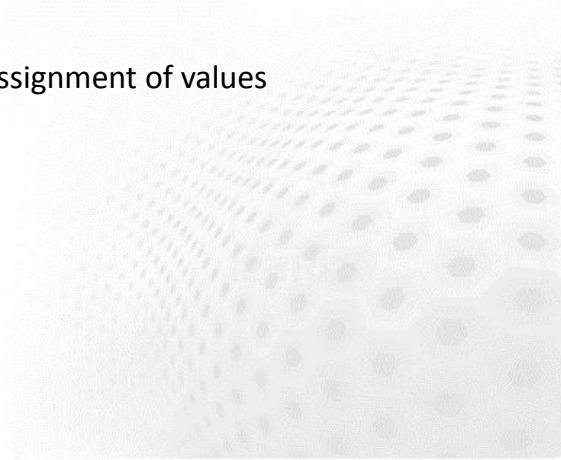
19

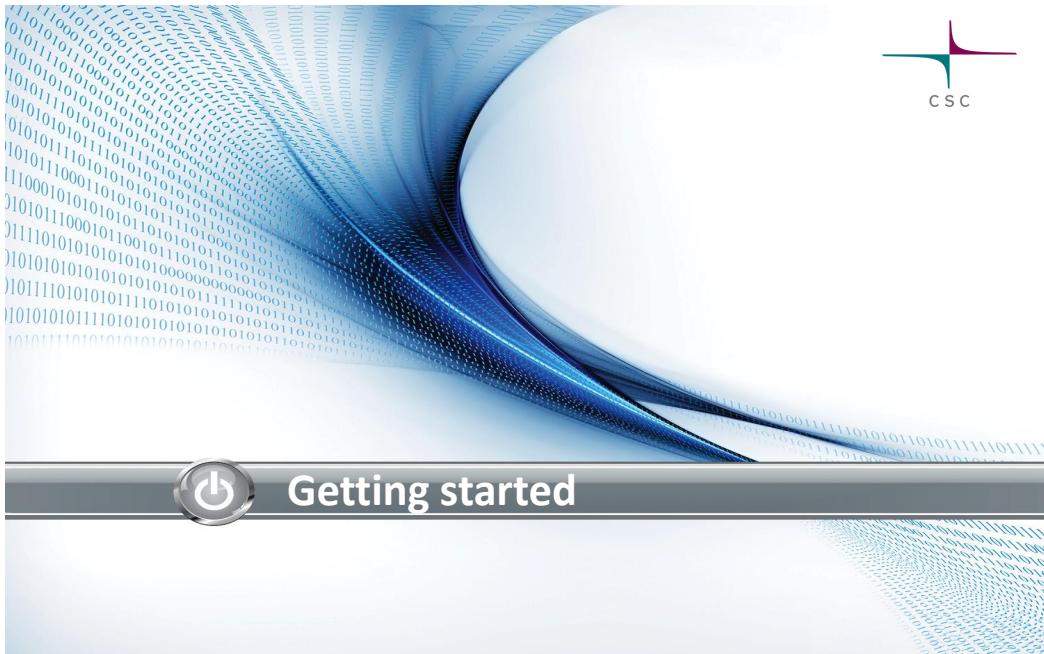
20

## Summary



- ➊ Short history and different standards
- ➋ Basic syntax
- ➌ Variables and their type, assignment of values
- ➍ Arithmetic operations
- ➎ Basic IO (printf)





## Getting started

22

# POINTERS, ARRAYS AND FUNCTIONS

23

## Pointers

- ➊ On the hardware level a variable is actually a reference to a memory location
- ➋ The contents of the memory block determine the value of a variable
  - The size of the memory block depends on the type of the variable
- ➌ Memory addresses can be stored (and manipulated) using *pointers*
- ➍ Pointer variables can be considered as variables that hold the address of a element with given datatype (**int**, **char**, ...)

24

## Pointers

- ➊ Pointer variables are defined by adding \* into the definition, for example a pointer named **ptr** of type **int** is defined as **int \*ptr;**
- ➋ The address of a variable can be obtained using & operator



25

## Arrays



- Static arrays can be introduced using [ ]:

```
char str[20];           Array of 20 characters  
double values[10];     Array of 10 doubles
```

- Elements of array can be accessed using the same [ ] operator. The value inside brackets is interpreted as an offset from the beginning of the array
  - Indices always start from 0
  - Last element of an array A of size n is A[n-1]

26

## Array examples

```
// int array of size 10  
int array[10];  
  
// Set value of third element to 1  
array[2] = 1;  
  
// Print the value  
printf("a[2]=%d\n", array[2]);  
  
// type of array is equivalent to  
// type (int*)  
int *ptr;  
  
array[0] = 3;  
ptr = array;  
printf("*ptr=%d\n", *ptr);
```

27

## FUNCTIONS



28

## Functions

- Functions are the only subroutine type in C
  - But functions do not have to return anything
- Function definitions are of form

```
type func-name(param-list)  
{  
    /* body of function */  
}
```
- Here **type** is the return type of the function
  - void** means that function does not return anything

29



## main function



- Every C program has the main function with definition

```
int main(int argc, char *argv[])
{
    /* body of function */
}
```

- Command line arguments are passed to the program using **argc** and **argv**
- main should always return integer
  - zero means success, non-zero values are errors

30

## Function example



- This function returns the sum of two integer arguments:

```
#include <stdio.h>
int add(int a, int b) {
    return a + b;
}

int main(void) {
    int val;
    val = add(3, 6);
    printf("Sum is %d\n", val);
    return 0;
}
```

31

## Variable scoping



- Variable scoping in C is local unless defined otherwise
  - Variables declared in the function definition are only visible inside the function body
  - Variables of calling scope are not visible inside function body

```
void funcB(float a) {
    int counter;
    ...
}
int funcA(void) {
    float value_1, value2;
    funcB(value_1);
    ...
}
```

Not accessible from funcA

Not accessible from funcB

32

## Arguments



- All arguments are passed as *values*
  - Copy of the value is passed to the function
  - Functions can not change the value of a variable in the *calling scope*
- Pointers can be used to pass a reference to a variable as an argument!

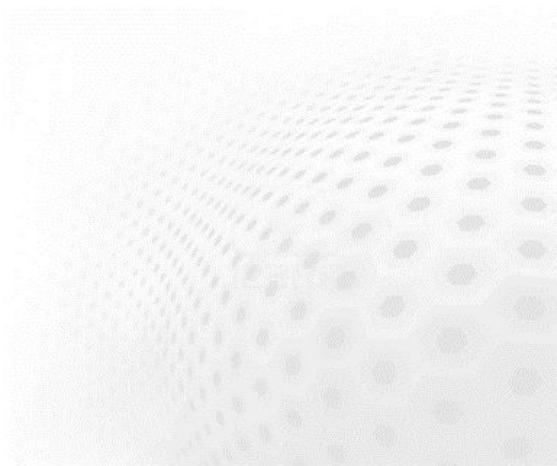
```
void funcB(int a) {
    a += 10;
}

int funcA(void) {
    int var = 0;
    funcB(var);
    // var is still 0!
    ...
}
```

33



## PREPROCESSOR

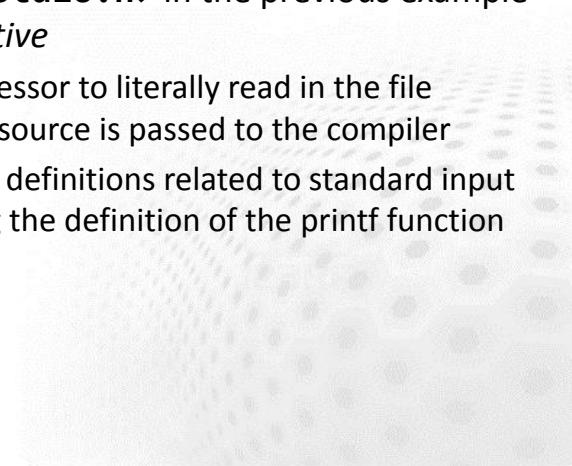


34

## C preprocessor



- The line **#include <stdio.h>** in the previous example is a *preprocessor directive*
  - It directs the preprocessor to literally read in the file `stdio.h` before the source is passed to the compiler
  - `stdio.h` has several definitions related to standard input and output, including the definition of the `printf` function



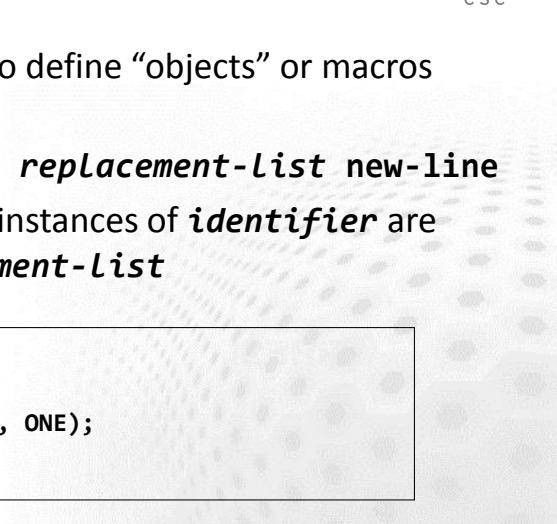
35



## Preprocessor macros and **#define**

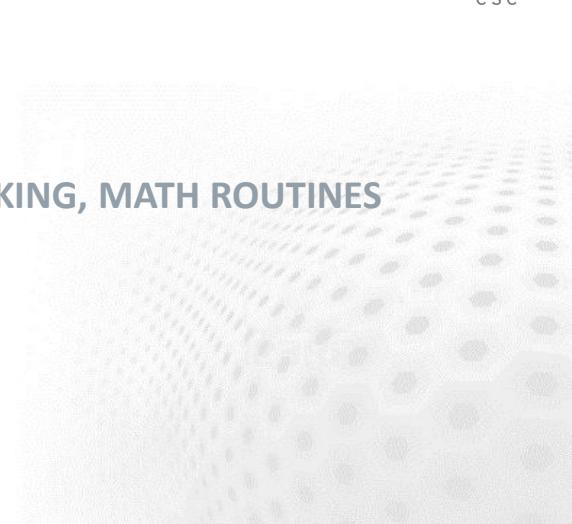
- **#define** can be used to define “objects” or macros
- It has the form of  
**#define identifier replacement-list new-line**
- After the definition, all instances of **identifier** are replaced with **replacement-list**

```
// Example
#define ONE 1
printf("Value is %d\n", ONE);
// Result: Value is 1
```



36

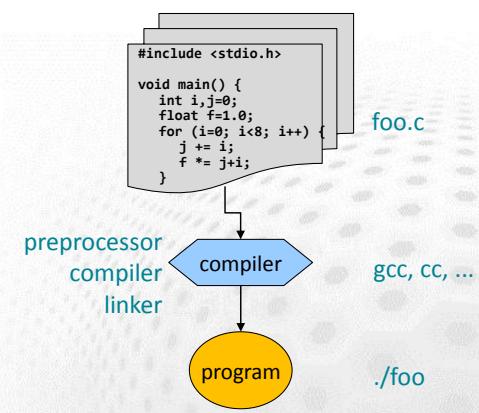
## COMPILING AND LINKING, MATH ROUTINES



37

## Transition from source code to program

- ➊ Compiling:
  - Transforming the C source code to machine language
- ➋ Linker:
  - Combines object files generated by the compiler into a single executable program
- ➌ The result is an executable binary file



38

## Let's try it out!

- ➊ Write the classic first program in C:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

- ➋ Compile your code:

```
$ gcc -o hello hello.c
```

- ➌ Test your program:

```
$ ./hello
Hello, World!
```

39

## Math routines library

- ➊ Math routines are defined in a *library* that is not linked in by default
- ➋ It includes the most common mathematical functions, e.g. **sin()**, **pow()**, **log()**, etc.
- ➌ Header **math.h** also has definitions for constants such as **M\_PI** for  $\pi$
- ➍ For power operation function **pow()** is used (^ is bitwise operator in C)

```
#include <math.h>

float r, theta;
double area, y;

// radius and an angle
r = 1.2;
theta = 0.456;

// calculate something
area = M_PI * pow(r,2);
y = sin(theta)
    + cos(theta/2.0);
y += exp(-3.1 * area);

// echo results
printf("area is %f\n", area);
printf("y=%+.18e\n", y);
```

40

## Compiler flags

- ➊ Compiler flags provide a way to control how the program code is processed
  - ➋ For example following (and many other) options can be used with **gcc**:
- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li><b>-o</b></li> <li><b>-c</b></li> <li><b>-lname</b></li> <li><b>-O</b></li> </ul> | <ul style="list-style-type: none"> <li>name of the output file</li> <li>generate an object file, do not link executable</li> <li>link in library <b>name</b>, for example <b>-lm</b></li> <li>optimize the program code (also <b>-O2</b>, <b>-O3</b>, ...)</li> </ul> |
|--|---|
- ➌ For more comprehensive list of options, see the man page of **gcc** ("man gcc")

41



### 💡 In complex projects:

- Compile each source code file (.c) into an object file (.o)

```
$ gcc -c main.c  
$ gcc -c sub.c
```

- Link object files into a binary and execute the binary

```
$ gcc -o foo main.o sub.o -lm  
$ ./foo
```

Link with math routines library!



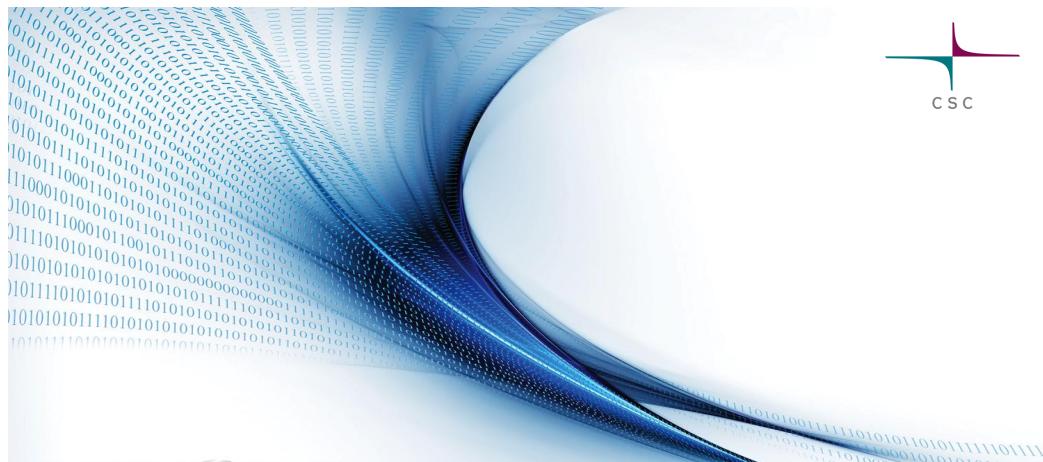
### 💡 Functions

### 💡 Pointers

### 💡 Compiling

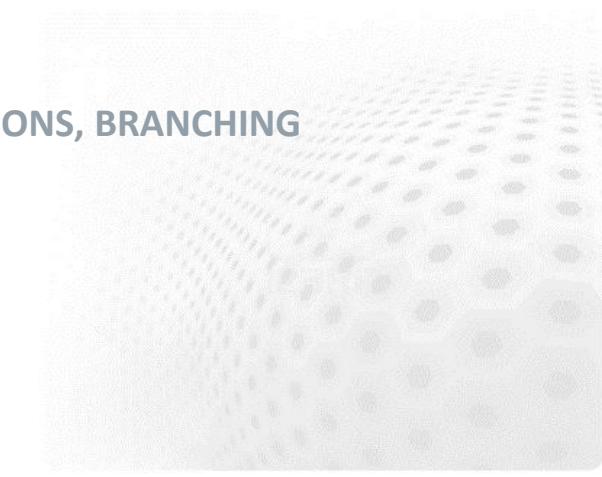
### 💡 Arrays

### 💡 Scoping

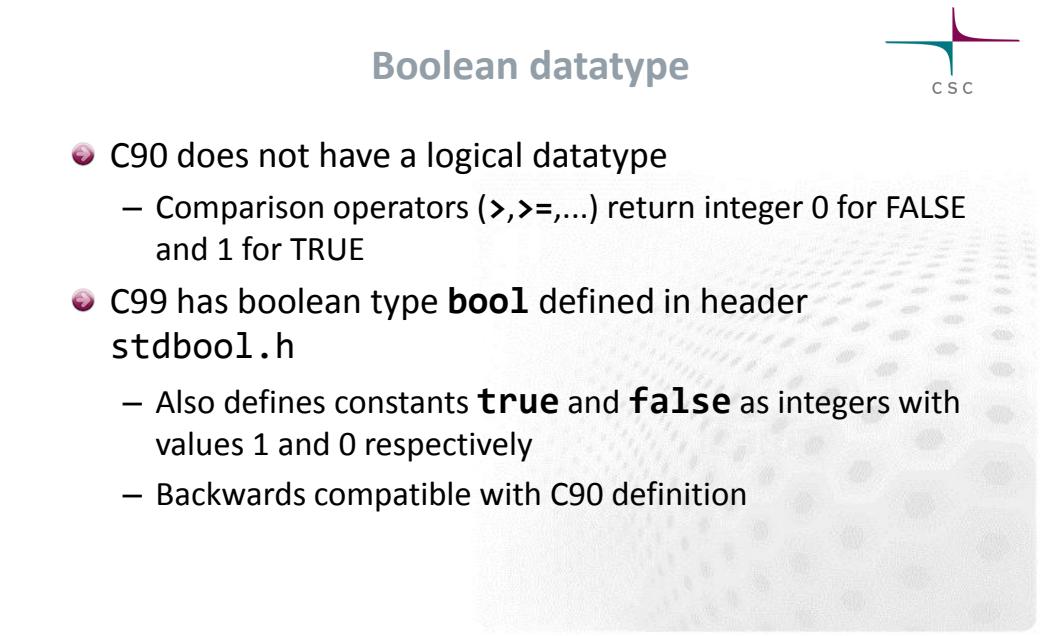


## Control structures

44



45



## Boolean datatype

- ➊ C90 does not have a logical datatype
  - Comparison operators ( $>$ ,  $>=$ , ...) return integer 0 for FALSE and 1 for TRUE
- ➋ C99 has boolean type **bool** defined in header **stdbool.h**
  - Also defines constants **true** and **false** as integers with values 1 and 0 respectively
  - Backwards compatible with C90 definition

46

## Recap: logical operators

➌ Operators	➍ Precedence (high to low)
$>$	<b>larger than</b>
$<$	<b>less than</b>
$>=$	<b>larger or equal</b>
$<=$	<b>less or equal</b>
$==$	<b>equal</b>
$!=$	<b>unequal</b>
$\&\&$	<b>AND</b>
$\  \ $	<b>OR</b>
!	<b>NOT</b>
	$*$ / % + -
	$<$ $<=$ $>$ $>=$
	$==$ $!=$
	$\&\&$
	$\  \ $
	$=$ $+=$ $-=$ $*=$ $/=$ $\%=$

47

## Control structures // if – else



- Code execution can be branched according to a logical test using **if** statement:

```
if (...) {...} else {...}
  conditional TRUE      FALSE
```

- TRUE** block is executed if the **conditional** evaluates to *non zero*, otherwise the (optional) **FALSE** block is executed

48

## Control structures // if – else



```
// simple if-else
if (x > 1.2) {
    y = 4 * x * r; // TRUE
} else {
    y = 0.0;         // FALSE
}

// else is optional
if (x || y) {
    z += x + y;
}
```

```
// complex if-elseif-else
if ((x > 1.2) && (i != 0))
{
    y = x * i; // 1st TRUE
} else if ((x < 1.0) && c)
{
    y = -x;     // 1st FALSE
                // 2nd TRUE
} else {
    y = 0.0; // 1st, 2nd
    FALSE
}
```

49

## Control structures // switch



- switch** (...) {
 **case** ... : ... **test value**
**break**; **end of branch**
**default**: ... } **default branch executed if nothing else matches**
- Condition:
  - single variable or expression
- Branch(=case) with matching value chosen
- break** stops branch

50

## Control structures // switch



```
// Simple switch based on the
// value of integer i
switch (i) {
    case 1:
        printf("one\n");
        break;
    case 2:
        printf("two\n");
        break;
    default:
        printf("many\n");
        break;
}
```

**good style to break even the last branch!**

51

## Control structures // switch



A

```
switch (i) {  
    case 1:  
        printf("one\n");  
        break;  
    case 2:  
        printf("two\n");  
        break;  
    default:  
        printf("many\n");  
}
```

B

```
switch (i) {  
    default:  
        printf("many\n");  
    case 1:  
        printf("one\n");  
        break;  
    case 2:  
        printf("two\n");  
        break;  
}
```

C

```
switch (i) {  
    case 1:  
        printf("one\n");  
    case 2:  
        printf("two\n");  
    default:  
        printf("many\n");  
}
```

In each case, what would  
be printed on the screen if  
i is 1, 2, or 3?

52

## Ternary operator



- ➊ Ternary conditional is an expression that can be used as a short-hand if-else in assignments:

(conditional) ? value-if-true : value-if-false

```
// ternary conditional in use  
total += found ? count : 0  
  
// is equal to this  
if (found)  
    total += count;  
else  
    total += 0;
```

53

## LOOPS



54

## Loops // for



- ➋ Looping over a block of code can be accomplished using for statement:

```
for (<initial>;  
     <condition>;  
     <post-iteration>) {...}
```

<initial>

executed once before entering  
the loop

<condition>

stop loop when condition is FALSE

<post-iteration>

execute after each iteration

55

## Loops // while



• `while (...) {...}`  
    ↑           ↑  
    condition   code block

- Code block executes repeatedly as long as condition is TRUE
- Condition evaluated before iteration

56

## Loops // while

```
// loop using a for-statement  
// i is incremented after each iteration  
for (i = 0; i < bar; i++) {  
    x += i*3.14 + c;  
    c = i + 2;  
}
```

```
// the same loop but with a while-statement  
i = 0;  
while (i < bar) {  
    x += i*3.14 + c;  
    c = i + 2;  
    i++;  
}
```

57

## do while



- The condition for while is evaluated before the block is executed
  - Sometimes it is desirable to start the loop first and then check the condition
- This can be done using `do {...} while ()` loop

```
// Execute the loop at least  
// one time  
counter = 0;  
do {  
    printf("in loop");  
} while (counter != 0)
```

58

## Special for loops



- All statements in the **for** loop construct are optional
- So all following loops are valid C code (with variables correctly defined):

```
for (; a < threshold; a += 1e-3) {...}  
for (; still_valid;) {...}  
for (;;) {...}
```

- Try to avoid using these – use the **while** instead

59

### break

- end a loop (for, while, do) or a switch statement

### continue

- continue with the next iteration of a loop

Note that these apply to the smallest enclosing iteration statement

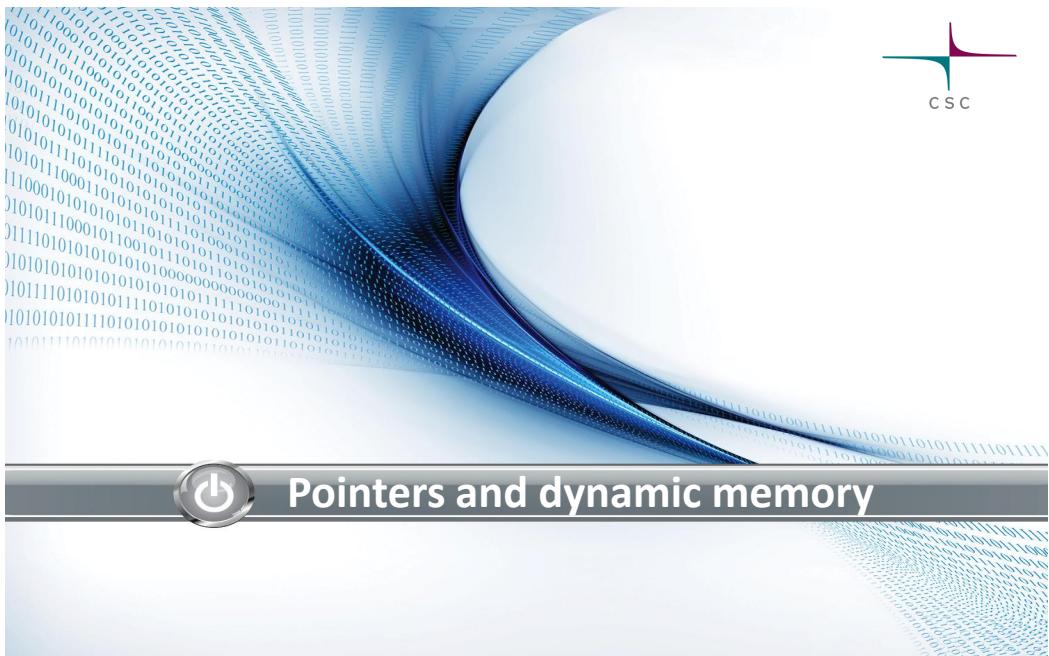
```
// jump out of a loop
// at first iteration
for (i = 0; i < 10; i++) {
    break;
    printf("in loop");
}
printf("i=%d ", i);
// output: i=0

// jump to the next loop
// iteration
for (i = 0; i < 10; i++) {
    continue;
    printf("in loop");
}
printf("i=%d", i);
// output: i=10
```

60

- Boolean values, comparisons
- if-then-else
- switch-case
- Loops

61



## Pointers and dynamic memory

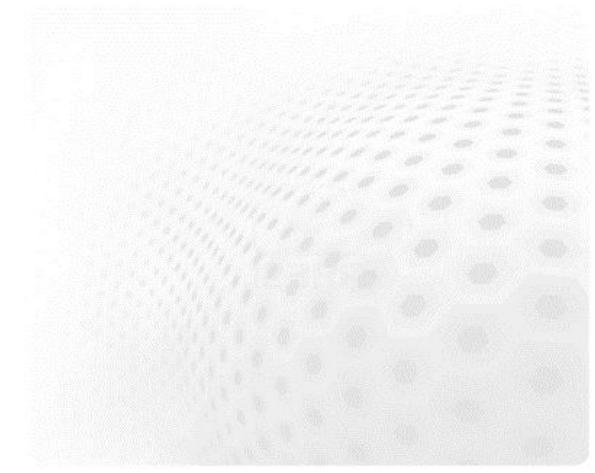
62

### Recap

- Pointer variables hold a reference to a memory location
- Pointers have type (**int**, **float**, ...)
- Pointer variables are defined with \*, for example  
**int \*a;    double \*d;**
- Address of a variable can be obtained using address operator & (same symbol as bit-wise AND)

64

### USING POINTERS



63

### Address operator &

- Operator & returns the reference to the operand
- Type of the pointer matches with the type of the operand
  - That is, reference to an **int** is of type pointer to an integer, (**int \***)

```

int i_value = 0;
double d_value;
int *ptr;

// This is ok
ptr = &i_value;

// Types have to match!
ptr = &d_value;
  
```

65

## Dereferencing pointers with \*



- ➊ Dereferencing means accessing the value from the address that pointer points to
- ➋ Dereferencing is done using \* operator
- ➌ Precedence can be changed with parentheses ()

```
int a = 10;
int *ptr;

// ptr points to a
ptr = &a;

// Let's do some arithmetics
(*ptr)++;

// and print the value of a
printf("a=%d\n", a);

// Result: a=11
```

66

## Functions with pointer arguments



- ➊ All function arguments are passed “by value”
  - When function changes the values of arguments the changes are not visible to the caller (no side effects)
- ➋ When a pointer is passed as a function argument the function can not change the value of the pointer itself, but it *can* modify the referenced value

67

## Pointer arguments example



```
#include<stdio.h>

void demo(int a, int *b) {
    a += 1;
    *b += 1;
    printf("In demo function: a=%d, b=%d\n", a, *b);
}

int main(void) {
    int a = 0, b = 0;
    demo(a, &b);
    printf("In main after call: a=%d, b=%d\n", a, b);
    return 0;
}
// Result:
// In demo function: a=1, b=1
// In main after call: a=0, b=1
```

68

## Pointer arithmetics



- ➊ Pointer variables can also be modified using arithmetic operations
- ➋ Can lead to bugs that are very difficult to find and fix
- ➌ Can be useful in some cases when manipulating arrays
- ➍ Try to avoid when possible!

```
#include<stdio.h>
#include<stddef.h>
int main(void) {
    int a = 0, b = 0;
    int *ptr;
    ptrdiff_t diff;
    diff = &a - &b;
    ptr = &a;
    *(ptr - diff) = 10;
    printf("a=%d,b=%d\n",a,b);
    return 0;
}
// Result: a=0,b=10
```

69

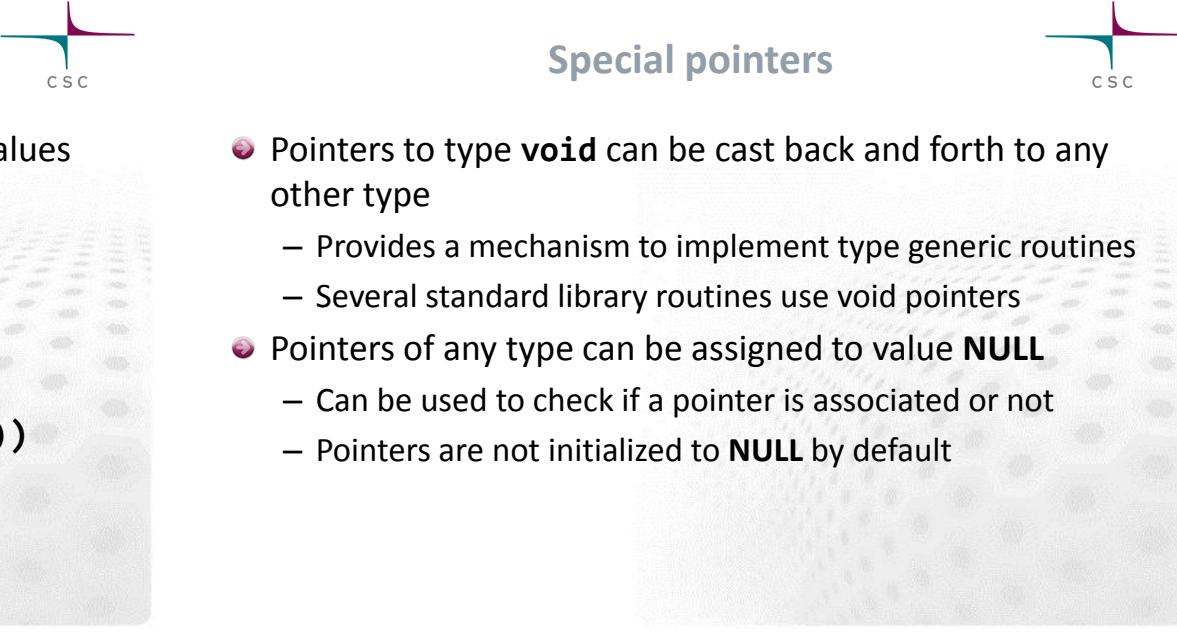
## Dereferencing pointers with []



- For arrays it is more convenient to dereference the values using [] operator
  - This was already used with static sized arrays
- Definition of the [] operator is that **E1[E2]** equals to **(\* (E1 + (E2))**), for example  
**b=3\*arr[n+1]** is same as **b=3\*(\* (arr + (n+1)))**

70

## Special pointers



- Pointers to type **void** can be cast back and forth to any other type
  - Provides a mechanism to implement type generic routines
  - Several standard library routines use void pointers
- Pointers of any type can be assigned to value **NULL**
  - Can be used to check if a pointer is associated or not
  - Pointers are not initialized to **NULL** by default

71

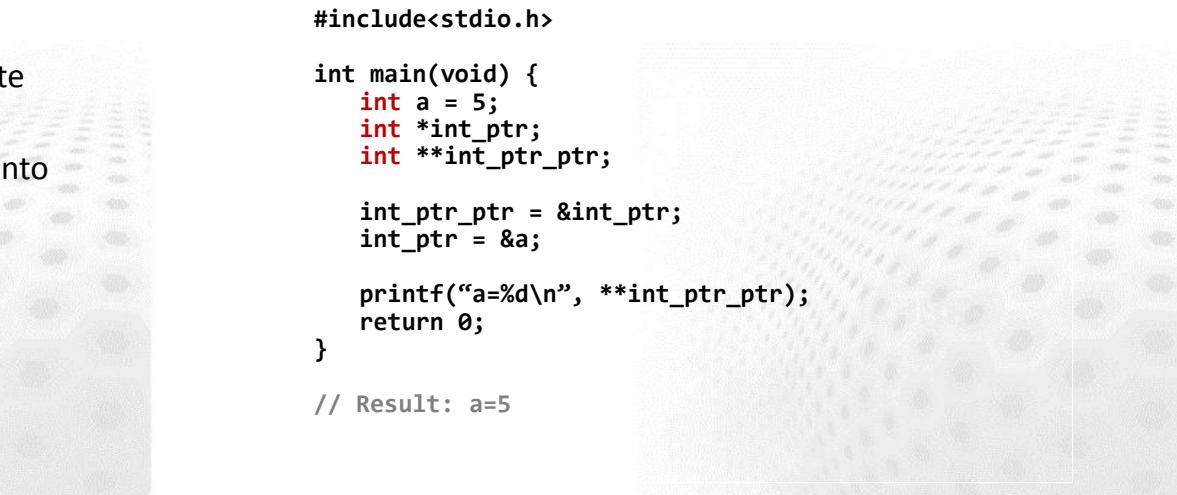
## Pointers to pointers



- It is possible to have pointer references to pointers
  - This is very useful when functions have to manipulate values of arguments of pointer type
  - Multidimensional arrays are also naturally mapped into pointers of pointers

72

## Pointer to a pointer example



```
#include<stdio.h>

int main(void) {
    int a = 5;
    int *int_ptr;
    int **int_ptr_ptr;

    int_ptr_ptr = &int_ptr;
    int_ptr = &a;

    printf("a=%d\n", **int_ptr_ptr);
    return 0;
}

// Result: a=5
```

73

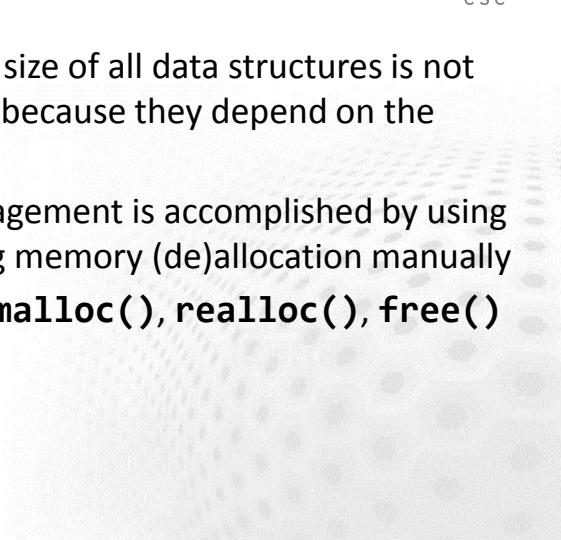
## MALLOC, REALLOC AND FREE



74

### Dynamic memory management

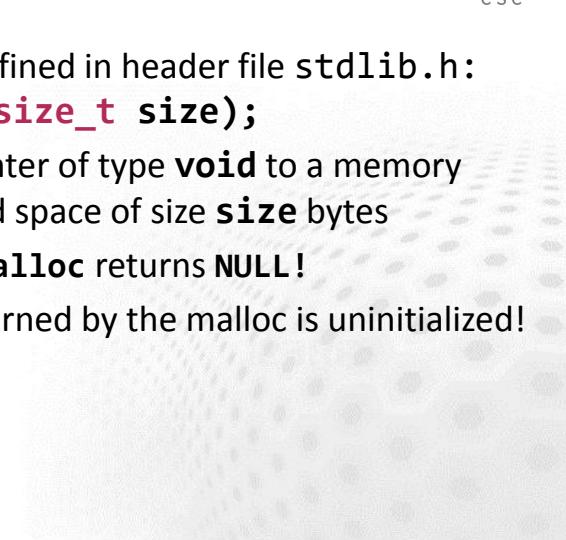
- In most cases the exact size of all data structures is not known at compile time because they depend on the input data
- Dynamic memory management is accomplished by using pointers and controlling memory (de)allocation manually
- Relevant functions are **malloc()**, **realloc()**, **free()**



76

### malloc

- **malloc** function is defined in header file **stdlib.h**:  
**void \*malloc(size\_t size);**
- **malloc** returns a pointer of type **void** to a memory location with allocated space of size **size** bytes
  - If allocation fails, **malloc** returns **NULL**!
- The memory area returned by the **malloc** is uninitialized!



77

## How to determine the size for allocation?



- The only argument for **malloc** function is the size of the object in bytes
- The sizes of different objects can be determined using **sizeof** operator which returns the size of the argument in bytes
  - Return type of the **sizeof** operator is **size\_t**
- Example:

```
int *ptr = (int *) malloc(sizeof(int));
float **ptr = (float **) malloc(sizeof(float *));
```

78

## free



- **free** deallocates previous allocated object
- ```
void free(void *ptr);
```
- After freeing you should not try to access any part of the allocation
- Calling free with a pointer that does not point to a allocated memory can crash the code
  - Calling free twice is a common mistake!

80

## realloc



- **realloc** function changes the size of allocation, its definition is
- ```
void *realloc(void *ptr, size_t size);
```
- The argument **ptr** has to point to a previously allocated object
  - If ptr is **NULL**, **realloc** is equivalent to **malloc**
- Contents of the allocated memory area returned by **realloc** is equal up to the lesser of old and new sizes

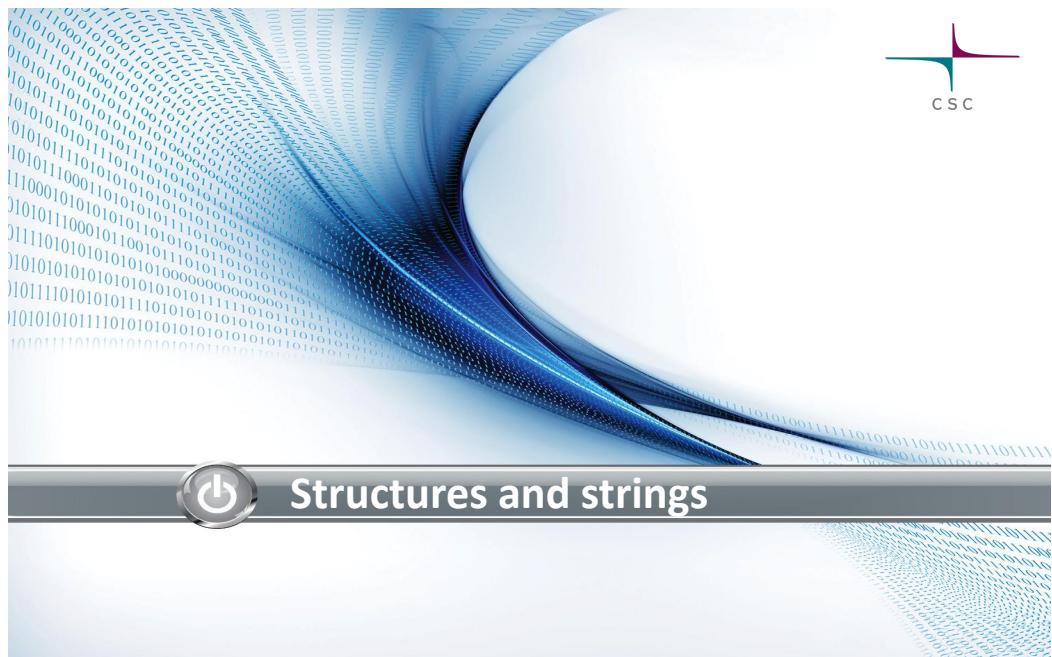
79

## Summary



- Pointers are references to memory locations
- Operators **\***, **&** and **[ ]**
- Functions with pointer arguments can manipulate the values of arguments
- Pointers to pointers
- Dynamic memory management
  - **malloc** and **free**

81



## Structures and strings

82

## STRINGS

83

## Strings

- ➊ Strings in C are arrays of type **char** that end with value 0 (not character 0!)
  - This definition is problematic and has caused many critical bugs especially in operating systems
- ➋ String manipulation routines are defined in header **<string.h>**
  - There are routines with both **str** and **mem** prefixes
    - Different behavior when the array is not NULL terminated

84

## String manipulation

- ➌ Concatenate:
 

```
char *strncat(char *s1, char *s2, size_t n);
```
- ➍ Compare:
 

```
int strncmp(char *s1, char *s2, size_t n);
```
- ➎ Copy:
 

```
char *strncpy(char *s1, char *s2, size_t n);
```
- ➏ Search a character:
 

```
void *memchr(void *s, int c; size_t n);
```
- ➐ String length:
 

```
size_t strlen(char *s);
```

85

## Character manipulation

- Defined in header `<ctype.h>`

<code>int isspace(int c)</code>	test for space
<code>int isalpha(int c)</code>	test for alphabetic letter
<code>int isdigit(int c)</code>	test for decimal digit
<code>int isalnum(int c)</code>	test for letter or digit
<code>int iscntrl(int c)</code>	test for control character
<code>int tolower(int c)</code>	convert to lower case
<code>int toupper(int c)</code>	convert to upper case



## String examples

```
// two string buffers  
char str1[256], str2[256];  
  
// initialize the strings  
strncpy(str1, "abc", 256);  
strncpy(str2, "def", 256);  
  
// add str2 to the end of str1  
strncat(str1, str2, 3);  
  
printf("len(str1)=%d\n", strlen(str1));  
// output: 6  
  
printf("%s, %s\n", str1, str2);  
// output: abcdef, def  
  
printf("tolower(A) = %c\n", tolower('A'));  
// output: tolower(A) = a
```

86

87

## STRUCTURES



## Defining a structure

- Structures are defined using **struct** keyword:

```
struct tag_name {  
    type member1;  
    type member2;  
    ...  
};
```

- Example:

```
struct book {  
    char author[100];  
    char title[100];  
    char publisher[100];  
    int year;  
};
```

88

89



## Declaring a structure variable



- Structure declarations are usually given at the beginning of the file before function definitions
  - Accessible by all functions in the same file
- Variables of a given structure type are declared using the `struct` keyword, for example

```
struct book item;
```

declares a variable `item` of type `struct book`

90

## Accessing structure members



- Structure members can be accessed using `.`, for example setting the year and printing the title for a book defined previously:

```
struct book item;  
...  
item.year = 1984;  
...  
printf("Author: %s\n", item.author);
```

92

## Structures - why?



- Structures can be considered as a collection of variables with (possibly) different datatypes
- Code can be made more readable when all related information is passed between function using just a one argument of relevant structure
  - For example the information of a book in a library
- Also implementing abstract datatypes, such as lists and binary trees, is much more convenient with structures

91

## Pointer to structure



- There is a short-hand notation `->` for accessing elements of a pointer to a structure:

```
struct book item;  
struct book *ptr;  
ptr = &item;  
(*ptr).year = 2015;  
ptr->year = 2015;
```

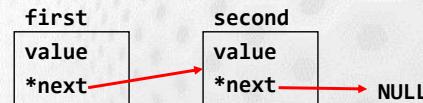
93

## Pointer to structure



- Structures can have other structures as members
  - Also a pointer to the structure itself!
  - Abstract datatypes can be nicely defined using structures with pointers

```
struct node {  
    int value;  
    struct node *next;  
};  
  
struct node first, second;  
first.next = &second;  
second.next = NULL;
```



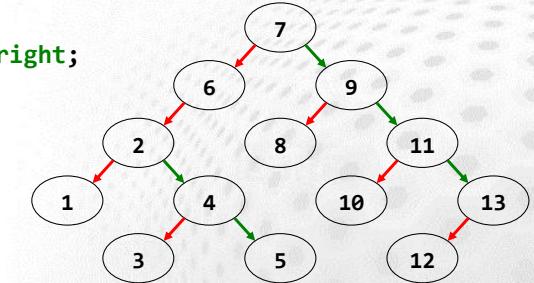
94

## Pointer to structure



- Example of ADT: binary tree

```
struct node {  
    integer value;  
    struct node *left, *right;  
};
```



95

## Summary

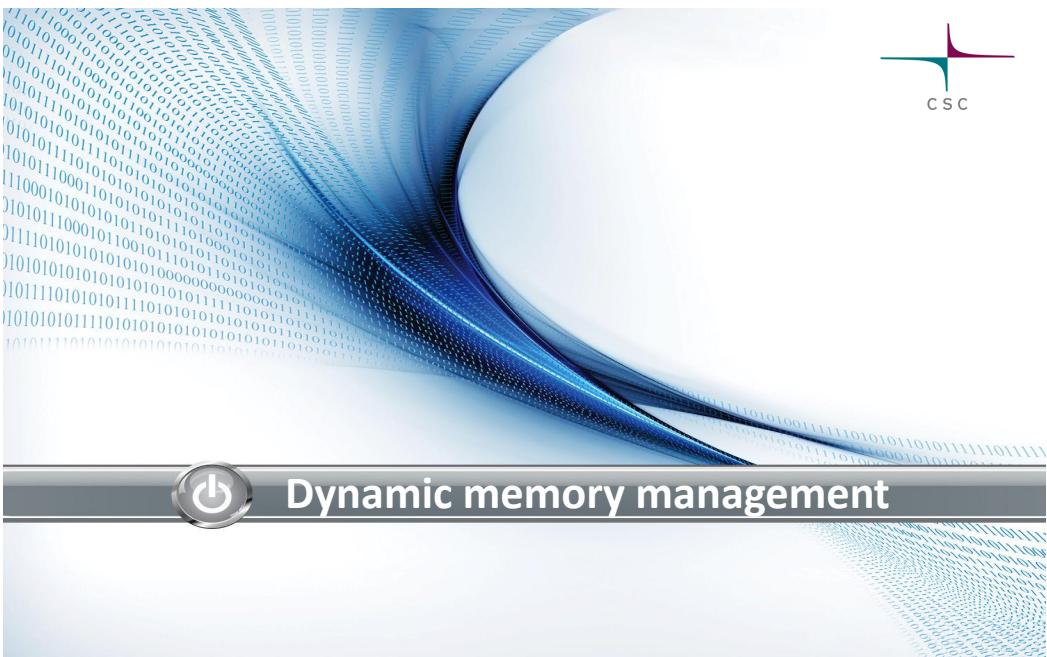


- Strings in C always end with 0
  - Remember the safety implications of string operations
- Structure is datatype that is a collection of members
  - Packing relevant data to a single unit
  - Abstract data types

96

## Dynamic arrays

- ➊ **malloc** can be used to allocate space for arrays too
- ➋ When allocating an array just multiply the size of each element in the array by the number of elements
  - **malloc** returns a pointer to the beginning of the array
  - Elements can be accessed with normal pointer syntax



## Dynamic memory management

97

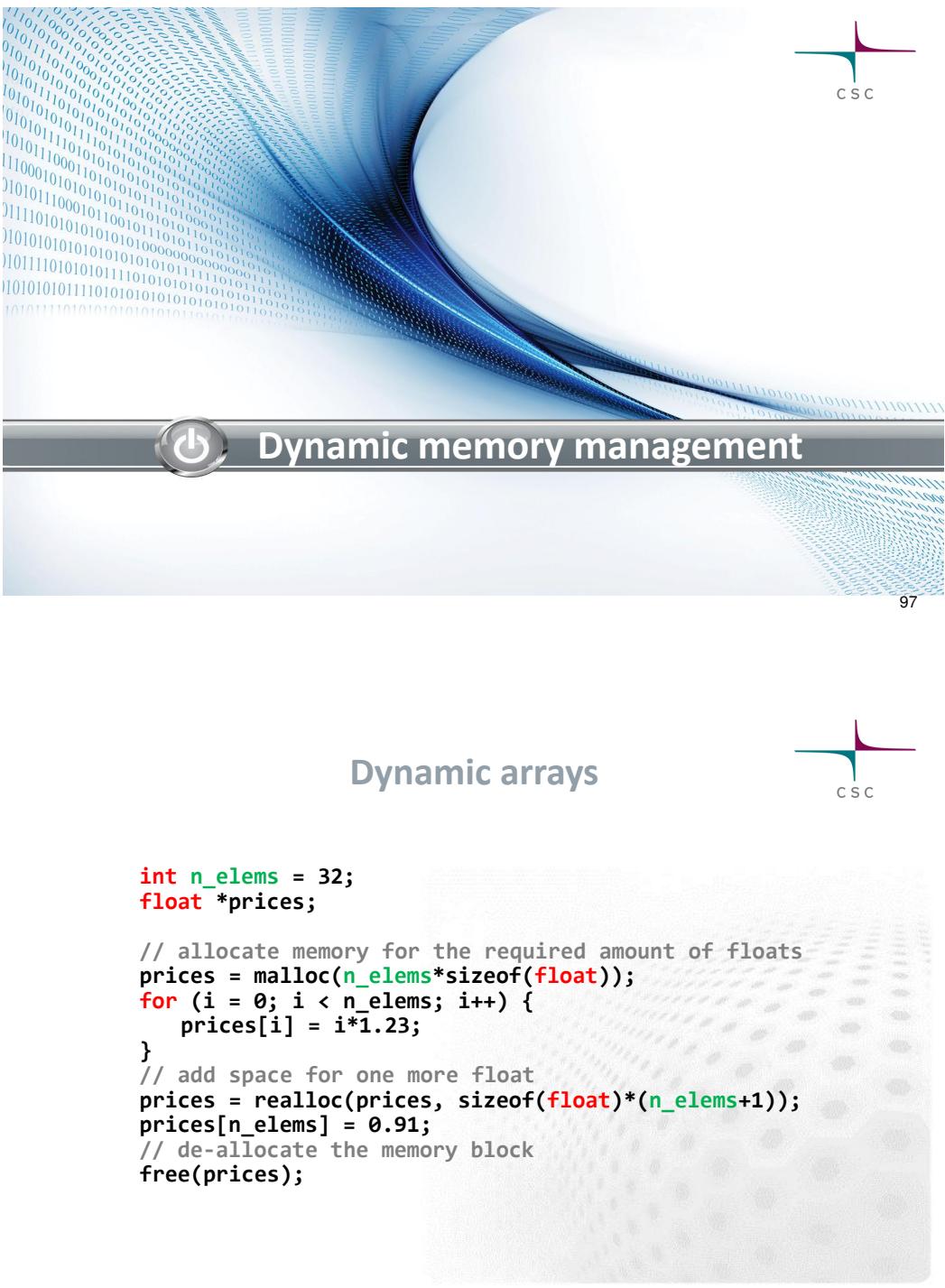
## Dynamic arrays

```

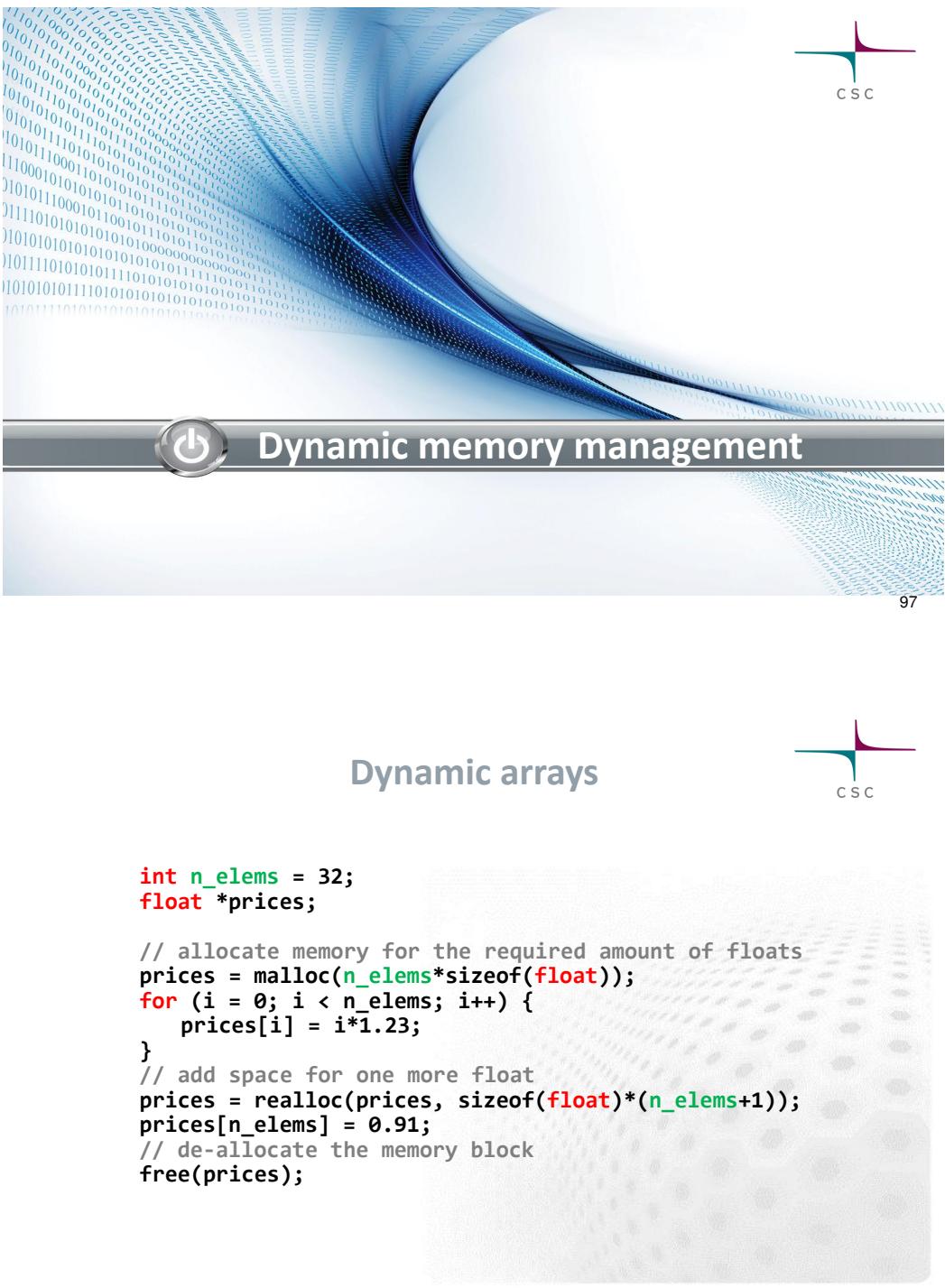
int n_elems = 32;
float *prices;

// allocate memory for the required amount of floats
prices = malloc(n_elems*sizeof(float));
for (i = 0; i < n_elems; i++) {
    prices[i] = i*1.23;
}
// add space for one more float
prices = realloc(prices, sizeof(float)*(n_elems+1));
prices[n_elems] = 0.91;
// de-allocate the memory block
free(prices);

```



- ➊ Doable, but becomes complicated
- ➋ No real multi-dimensional arrays in C, so really just arrays of arrays
  - Two dimensional array maps to a variable that is a pointer to a pointer
- ➌ Memory management by hand
  - There are at least two different ways to do the allocation
  - Easy to make mistakes, beware here lieth dragons!



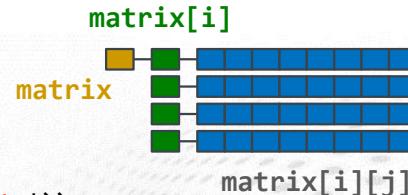
## Allocating row-by-row, not recommended



```
int i;
int rows = 4, cols = 8;
float **matrix;

/* allocate memory */
matrix = malloc(rows * sizeof(float *));
for (i = 0; i < rows; i++)
    matrix[i] = malloc(cols * sizeof(float));

// start using the 2D array
matrix[0][2] = 3.14;
```



101

## Dynamic multi-dimensional arrays



Dynamic 2D array in *contiguous* memory:

- ➊ First, allocate memory for pointers to the first element of each row
- ➋ Second, allocate memory for all elements
- ➌ Third, point each row at the first element of that row

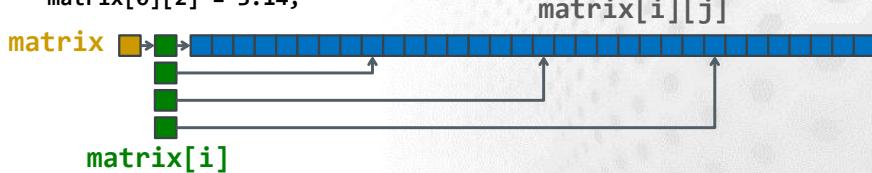
## Dynamic multi-dimensional arrays



```
/* allocate memory */
matrix = malloc(rows * sizeof(float *));
matrix[0] = malloc(rows * cols * sizeof(float));

/* point the beginning of each row at the correct address */
for (i = 1; i < rows; i++)
    matrix[i] = matrix[0] + i * cols;

// start using the 2D array
matrix[0][2] = 3.14;
```



103

## Memory layout



- ➊ Allocating space for the whole array using a single malloc call is the recommended way
  - Number of expensive malloc calls is minimized
  - Array is represented as one contiguous block in memory
  - It can be copied without looping over rows
  - Most IO and numerical libraries assume contiguous storage

104

## Freeing multi-dimensional arrays

```
/* free each row first */  
for (i = 0; i < rows; i++) {  
    free(matrix[i]);  
}  
/* only after that, we can free the  
main matrix */  
free(matrix);
```

OR

```
/* alternatively, when using contiguous  
memory */  
free(matrix[0]);  
free(matrix);
```



- After using a dynamic multi-dimensional array, remember to free each array inside the main array

105

## Summary

- Multidimensional arrays are a bit tricky and there are several different implementations
  - Malloc are usually time consuming
  - Memory layout matters

106



## I/O

107

### I/O – Standard C library stdio.h

To use:

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

1. Do a web search for “stdio.h reference” to see what functions are in the library.
2. Do a web search for “stdio.h printf” to get a detailed description of the printf function.
3. Type “man stdio” and “man printf” on a Linux workstation



109

### I/O – Introduction



#### Common I/O design alternatives

- databases
- data format libraries
- standard C libraries

#### Standard C library stdio.h

- Reading from keyboard and writing to display
- Reading and writing files

108

### I/O – Reading from keyboard and writing to display



**printf()** Print formatted data to stdout.

**scanf()** Read formatted data from stdin.

**putchar()** Print a single character to stdout.

**getchar()** Read a single character from stdin.

**puts()** Print a string to stdout.

**fgets()** Read a string from stdin (or file).

stdin = keyboard

stdout = display

110

## I/O – Reading from keyboard and writing to display



### • int printf(const char \*format, ...)

```
printf("The answer is %d.\n", 42);
printf("Pi equals to %.2f", 3.14159265);
printf("...at least to the %nd decimal.\n", 2);

/* output:
 * The answer is 42.
 * Pi equals to 3.14 ...at least to the 2nd decimal.
 */
```

111

## I/O – printf



\a	alarm (beep) character	\0NN	character code in octal
\p	backspace	\xNN	character code in hex
\f	formfeed	\0	null character
\n	<b>newline</b>		
\r	carriage return		
\t	horizontal tab		
\v	vertical tab		
\\\	<b>backslash</b>		
\?	question mark		
\'	single quote		
\”	<b>double quote</b>		

112

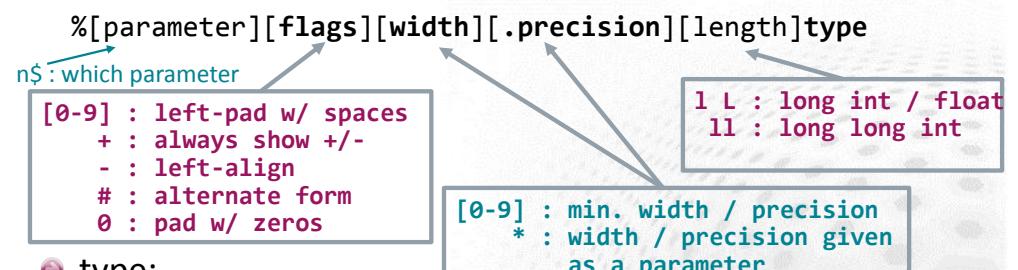
## I/O – printf



%d	decimal integer	%g	use %e or %f, whichever is shorter
%c	character	%u	unsigned decimal integer
%s	string	%o	unsigned octal integer
%e	floating-point number in exponential notation	%x	unsigned hex integer
%f	floating-point number in decimal notation		

113

### • The format specifier:



### • type:

d, i	int	x	hexadecimal
f, e, g	float / double	o	octal
s	string	c	char
%	literal %		

114

## I/O – Reading from keyboard and writing to display



### • `int scanf(const char *format, ...)`

```
printf("Enter a number: ");
scanf("%d", &number);
printf("You entered %d.\n", number);

/* output:
 * Enter a number: 9
 * You entered 9.
 */
```

115

## I/O – `scanf`

- When can you assume that the input is well-formatted?  
Never.
- Good article :“Things to Avoid in C/C++” at [www.gidnetwork.com](http://www.gidnetwork.com) for discussion about `gets()` and `scanf()`.

116

## I/O – Reading from keyboard and writing to display



<code>printf()</code>	Print formatted data to stdout.
<del><code>scanf()</code></del>	<del>Read formatted data from stdin.</del>
<code>putchar()</code>	Print a single character to stdout.
<code>getchar()</code>	Read a single character from stdin.
<code>puts()</code>	Print a string to stdout.
<code>fgets()</code>	Read a string from stdin or a file.

stdin = keyboard  
stdout = display

117

## I/O – Reading from keyboard and writing to display



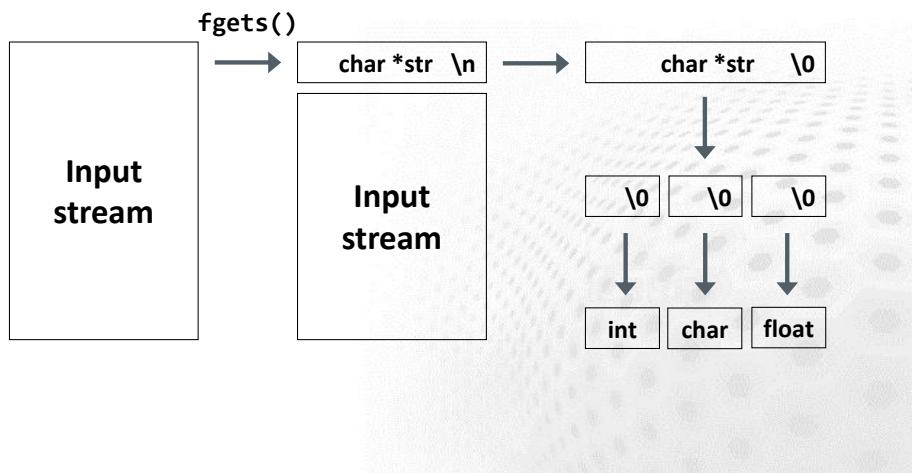
### • `char *fgets(char *str, int num, FILE *stream)`

```
printf("Enter some text\n");
fgets(string, 100, stdin);
printf("You entered:\n");
printf("%s", string);

/* output:
 * Enter some text:
 * Reading and validating input is difficult.
 * You entered:
 * Reading and validating input is difficult.
 */
```

118

## I/O – fgets



119

## I/O – Reading and writing files



- **FILE \*fopen(const char \*filename, const char \*mode)**

r, w, a	read, write, append
r+, w+, a+	read+write
rb, wb, ...	binary mode

- **int fclose(FILE \*stream)**
- **int fflush(FILE \*stream)**

120

## I/O – fopen, fclose



```
char filename[] = "file.txt";
FILE *myfile;
myfile = fopen(filename, "r");
if (myfile == NULL) {
    printf("Can't open %s. Check that it ", filename);
    printf("exists and the permissions.\n");
    return EXIT_FAILURE;
}
fclose(myfile);

/* output (on error):
 * Can't open file.txt. Check that it exists and the
 * permissions.
 */
```

121

## I/O – Reading and writing files



- |                            |   |
|----------------------------|---|
| <b>fprintf()</b>           | Print formatted data to file.             |
| <del><b>fscanf()</b></del> | <del>Read formatted data from file.</del> |
| <b>fputc()</b>             | Print a single character to file.         |
| <b>fgetc()</b>             | Read a single character from file.        |
| <b>fputs()</b>             | Print a string to file.                   |
| <b>fgets()</b>             | Read a string from file.                  |
| <br>                       |   |
| <b>fwrite()</b>            | Write binary data to file                 |
| <b>fread()</b>             | Read binary data from file                |

122

## I/O – Reading and writing files



### • `int fgetc(FILE *stream)`

```
int c;
char filename[] = "file.txt";
FILE *myfile;
myfile = fopen(filename, "r");
c = fgetc(myfile);
while (c != EOF) {
    printf("%c", (char)c);
    c = fgetc(myfile);
}
fclose(myfile);
```

123

## I/O – Reading and writing files



### • `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`

```
char filename[] = "file.dat";
FILE *mybinary;
mybinary = fopen(filename, "wb");
fwrite(&number, sizeof(int), 1, mybinary);
fwrite(array, sizeof(float), 1000, mybinary);
```

124

## I/O – Reading and writing files



### • `size_t fread(const void *ptr, size_t size, size_t count, FILE *stream)`

```
char filename[] = "file.dat";
FILE *mybinary;
mybinary = fopen(filename, "rb");
fread(&number, sizeof(int), 1, mybinary);
fread(array, sizeof(float), 1000, mybinary);
```

125

## I/O – Final words



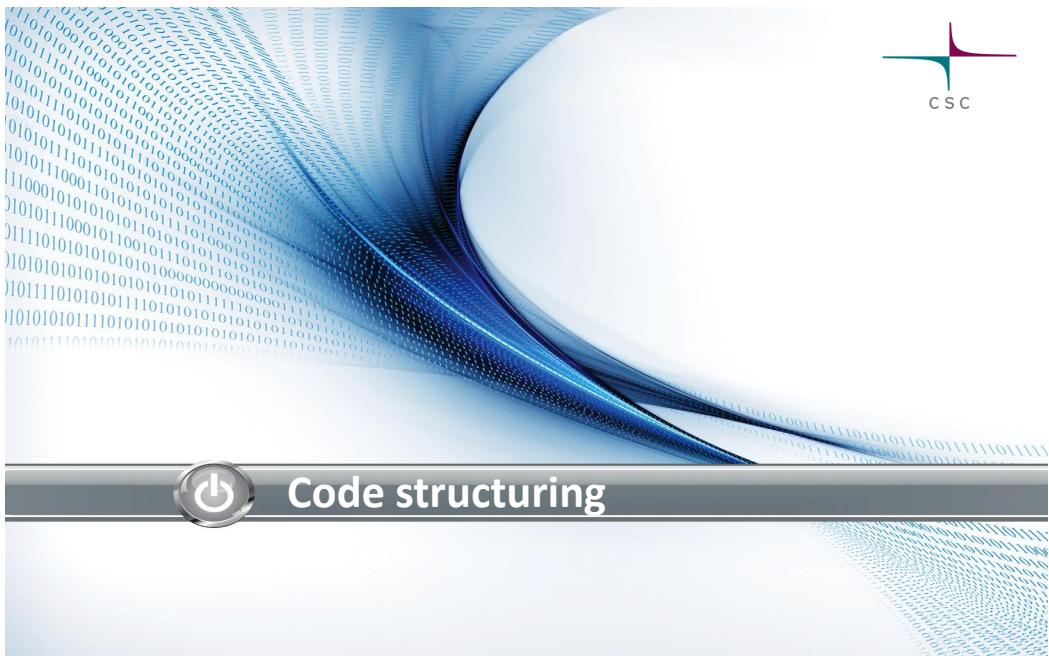
### • Standard C libraries

- text or binary?

### • Alternative solutions

- databases
- data format libraries

126



## Code structuring

127

## C PREPROCESSOR

128

### Preprocessing directives

- ➊ C preprocessor is a part of the compiler that does initial text substitution, manipulation, inclusion and other activities before the actual translation is done
  - We have already used **#include**, which includes a file and **#define** for macros
- ➋ C relies heavily on preprocessor to accomplish
  - Portability of code
  - Source code control
  - Debugging

129

### Directives

- ➌ Preprocessor directives start with **#**, which has to be first token on a line
- ➍ Directives are limited to one line
  - Line can be continued using \
- ➎ Directives are not statements, do not end the line with ;

```
// These are ok
#define one
#define two
// Not the first token - WRONG!
int i; #define one
```

130

## Conditional inclusion with #if



- Conditionals can be used to control if part of the code is included (and compiled) or not
- Conditional part begins with **#if/#ifdef/#ifndef** and ends with **#endif**:

```
#if constant-exp
    text section
#elif constant-exp
    text section
#else
    text section
#endif
```

```
#ifdef identifier
    text section
#else
    text section
#endif
```

```
#ifndef identifier
    text section
#else
    text section
#endif
```

131

## Example

```
#include <stdio.h>
#define PRINT_GREETINGS
#define PRINT_VALUE 3
int main(void) {
#ifndef PRINT_GREETINGS
    printf("Hello, World!\n");
#endif
#if PRINT_VALUE == 3
    printf("Value is %d\n", PRINT_VALUE);
#endif
    return 0;
}
```

132

## Definitions on compiler command line



- It is also possible to set preprocessor definitions on the compiler command line
- Most compilers accept option **-D** for this purpose:

```
gcc -DONE=1 -DUSE_FEATURE
```

is equivalent with

```
#define ONE 1
#define USE_FEATURE
```

133

## COMPILING WITH SEVERAL FILES



134

## Compilation: working with several files



### Advantages:

- Code structure is easier to understand when related parts are in same file
    - *Scoping* of variables can be controlled more strictly
  - Changing a file, only that file needs to be re-compiled to rebuild the program
- UNIX ‘make’ can be very useful tool for this!
- Most IDEs also provide a tools for building this kind of compilation

135

## Function prototypes



- Functions, like variables, have to be defined (declared) before use

### Program execution starts from **main()**

- If function definitions are in the same file and definitions are always before usage then compiler can find everything it needs
- Otherwise we have to introduce the functions using *function prototypes*

136

## Function prototypes



- Function prototypes are like variable declarations: they introduce the name of the function, its arguments and return value
- Note that only types of arguments are needed
- Function prototype declaration is a statement so you have to finish it with ;

```
// Function prototype
int funcA(int, int);
// Function can now be used
int main(void) {
    ...
    val = funcA(arg1, arg2);
    ...
}
// Function definition
int funcA(int a, int b)
{
    return a * b;
}
```

137

## Working with several files



- We can use header files to define functions that we can use later
- Making .h files for your functions allows you to ‘include’ them in your code

```
// we implement the function 'add'
int add(int first, int second)
{
    return first + second;
}
```

File: add.c

```
// we define the function 'add'
#ifndef ADD_H
#define ADD_H
int add(int first, int second);
#endif
```

File: add.h

138



- Another .c file can then use the function **add()** by including the header file:

```
// If we use #include "", the file is searched from
// current directory
#include "add.h"

int example(int x)
{
    return add(x, add(x,x));
}
```

139



- So, this is how headers work:
  - main()** would be in one file, the others will contain functions
  - Headers usually only contain definitions of data types, function prototypes and C preprocessor commands
  - We include the header into the C files
  - We compile the different files and the compiler calls the header file

140

## Global variables, extern



- Scoping of variables depend on the place where the variable is declared
  - Variables declared inside a function are visible only in the scope of that function
- Variables defined in a file have a scope of that file
- If a truly global value is needed (not generally recommended practice), one has to use **extern** keyword
  - Only one definition in file scope
  - Other files refer this value with **extern**

141

## Example



file main.c:

```
...
// File scope variable for
// status
struct status global_status;
...
int main(void) {
    ...
}
```

file utilities.c:

```
...
// We want to use the same
// structure in this file
extern struct status global_status;
...
int set_error_status(int stat) {
    ...
    global_status.error_code = stat;
    ...
}
```

142

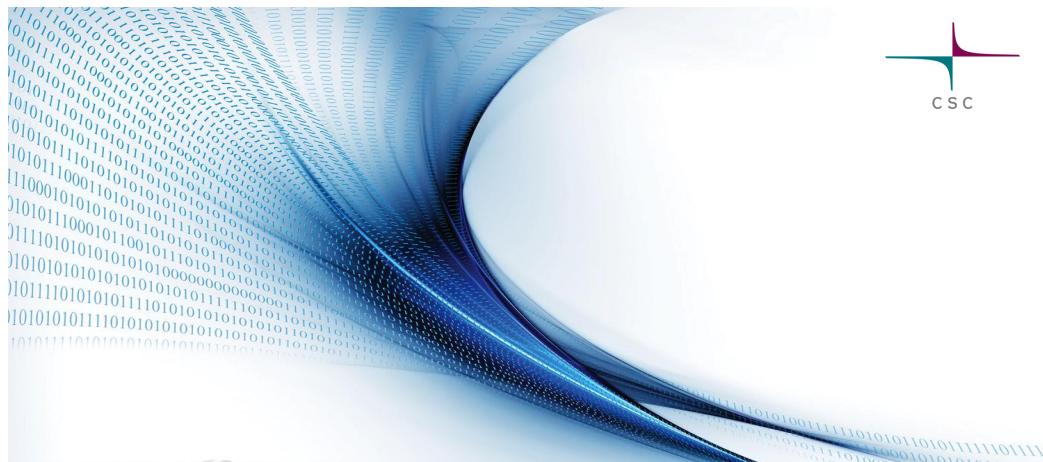
## Summary



- ➊ C preprocessor
  - Macros, conditional compiling
- ➋ Working with several files
  - Function prototypes
  - Compiling and linking



143



## Programming practices

144



## BACKGROUND

145

### What and why?



- ➊ Coding practices are an informal set of rules that community has learned by experience
  - Purpose is to help improve the quality of software
- ➋ Some practices may seem unimportant if you are just starting to write a small ad-hoc application for yourself
  - Software lifetime is hard to predict—small application can grow and become important
  - Wrong choices are much harder to fix when project grows and more developers start to contribute

146

### Software quality



- ➌ Several definitions and metrics, but some common desirable attributes are
  - Maintainability
    - How easy the code is to modify and extend?
  - Reliability
    - Does program work correctly with different inputs?
  - Efficiency
    - Is the program fast enough for the purpose it was made for?
  - Correctness
    - Does the program implement the given specifications correctly?

147

## CODING STANDARDS



### Keep it simple



- ➊ Code should be simple, do not use complicated logic to implement simple things
  - Complicated algorithms are of course difficult, but individual parts of code should not be!
- ➋ Complicated code filled with tricks is difficult to understand, fix and modify
  - After couple of years it will be hard even for the person who wrote it

148

149

### Comments



- ➊ Easily overlooked, but very important for new developers working on old code
- ➋ Comment at least the purpose of different data structures and things that use complicated logic

```
// Do not comment like this  
  
// Increment i by one  
i++;  
  
// This is more useful  
  
// Increment i so that  
// the residual will be  
// computed correctly  
i++;
```

150

### Code formatting



- ➊ As mentioned already in the beginning the C code formatting is very flexible
  - There are many common styles: K&R, GNU, Linux, ...
- ➋ Rule of thumb: *be consistent!*
  - It is much easier to read code that has all blocks formatted similarly
- ➌ Use automatic tools: **indent**

```
// GNU  
for (i = 0; i < n; i++)  
{  
    printf("Hi!\n");  
}  
print("Finished\n");  
// Linux  
for (i = 0; i < n; i++) {  
    printf("Hi!\n");  
}  
print("Finished\n");
```

151

## Naming conventions



- ➊ There are several conventions on naming, but again, choose one and use it
- ➋ In any case the name of variables and functions should be descriptive
- ➌ Preprocessor macros and definitions should always be uppercase!

```
// This  
d = s * t;  
// ... or this  
distance = speed * time_interval;
```

152

## Version control



- ➊ If you do any development you should already be using version control
  - It is helpful even in projects with just one developer
  - ... but it is critical tool for projects with larger teams
- ➋ Without VCS tracking bug fixes and changes becomes almost impossible
- ➌ Common version control systems include git, svn, mercurial, cvs, perforce,...

153

## Build tools



- ➊ When the size of the software project grows compiling becomes more and more complicated
  - Recompiling everything from the beginning becomes too slow and handling the dependencies can be complicated
- ➋ Build tools handle the dependencies
- ➌ Most common tool in UNIX-like systems is Make
- ➍ IDEs also provide build tools

154

## DEBUGGING



155

## Debuggers



- ➊ Debuggers are tools that are used to examine the program execution
  - Check the values of variables during execution without adding any extra I/O calls
  - Execute the program code step-by-step
  - Stop the program execution when a given condition is met
  - Examine the call tree and function call arguments

156

## Debugging demo



- ➊ Demo on tracking down the cause of segmentation fault

157

## COMMON PITFALLS



## Common pitfalls 1



- ➊ Segmentation faults
- ➋ Arrays in C are not protected
  - Trying to access a wrong element can in principle have any effect
  - Most difficult cases to debug are the ones where value of some other variable is modified by accident
  - Segmentation faults are usually easier to find
  - Some compilers have memory debugging tools integrated (e.g. address sanitizer in Clang and gcc)

158

159

## Common pitfalls 2



- Using = in comparison instead of ==

```
int val = 2;  
if (val = 1)  
    printf("val is equal to one");
```

- This will print that the value is one!

- The assignment operator returns the assigned value 1 which is nonzero and interpreted as true

- One way to avoid this is to use the constant on left, compiler will complain about the invalid assignment

160

## Common pitfalls 3



- Manual memory management is error prone

- Make sure that the allocations are always done before trying to dereference pointers

- Memory leak:

- Releasing the pointer to an allocation before free

161

## Common pitfalls 4

162

- Remember the implicit type conversions

- For example integer division:

```
double third = 1/3;
```

- This will set the value of third to zero

- Result of the integer division is 0 and that value is then casted into double precision 0.0

## Summary



- Coding standards help to develop programs

- Based on long time experience

- Debuggers help when you track down errors in your programs

163