Scientific and Technical Computing in C

Day 1

Luca Ferraro   Stefano Tagliaventi

CINECA - SCAI Department

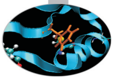
SuperComputing Applications and Innovation

Outline

C History

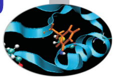
- Born in the 70s as an operating system programming language (*traditional C*)
- Widely adopted for application development because of its efficiency and availability on most systems
- First ANSI standard in 1989 (C89), adopted by ISO in 1990
- Second ISO standard in 1995 (C95), just a few extensions and fixes
- Third ISO standard in 1999 (C99), adding many new features (usability, more numeric types and math, more characters, inlining and restrict)
- Current standard is C11 (more usability, threads, Unicode characters, more robustness)

C General Philosophy



- A simple and efficient language
  - Only 44 reserved keywords
  - Basic data types and operators mapping "naturally" to the CPU
  - Facilities to build data types from the basic ones
  - Flexible flow control structures mapping the most common use cases
  - Translated by a compiler to machine language
- A rich Standard Library
  - Math functions, memory management, string manipulation, I/O, ... are not part of the language
  - Implemented separately in a library of subprograms
  - Linked into the executable after compilation
- A “preprocessor” to manage the code
  - Conditional compilation and automated code changes
  - Manipulates the code before compilation

- Why C is bad
  - Number crunching has been traditionally done in Fortran
  - Fortran is older and more “rigid” than C, compilers optimize better
  - Nowadays, performance differences are often a matter of compiler flags and good programming techniques
- Why C is good
  - From the beginning, it had more powerful data types
  - Non-numeric computing in Fortran is a real pain
  - There are more C than Fortran programmers
  - GUI and DB accesses are best programmed in C
  - Mixing C and Fortran uses (used...) to be troublesome
  - C99 seriously addressed numerical computing needs
  - ... and solved aliasing rules for memory pointers
- Bottom line:
  - Significant scientific libraries written in C
  - Significant scientific applications written in C
  - C compilers got much better at optimizing



Our Aims

- Teach you the fundamentals of the C language
- For both reading and writing programs
- Showing common idioms
- Illustrating best practices
- Blaming bad ones
- Making you aware of the typical traps
- Focusing on scientific and technical use cases
- You'll happen to encounter something we didn't cover, but it will be easy for you to learn more... or to attend a more advanced course!
- A course is not a substitute for a reference manual or a good book!
- Neither a substitute for personal practice

Outline

My First Scientific Program in C


```c
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>

int main() {
  double delta;
  double x1, x2;
  double a, b, c;

  printf("Solving ax^2+bx+c=0, enter a, b, c: ");
  scanf("%lf ,%lf ,%lf", &a, &b, &c);

  delta = sqrt(b*b - 4.0*a*c); // square root of discriminant
  x1 = x2 = -b;
  x1 = x1 + delta;
  x2 -= delta;
  x1 = x1/(2.0*a);
  x2 /= 2.0*a;

  printf("Real roots: %lf, %lf\n", x1, x2);

  return 0;
}
```

Comments to Code

- Text following `/*` is ignored up to the first `*/` encountered, even if it's on a different line
- In C99, text following `//` is ignored up to the end of current line
- Best practice: do comment your code!
  - Variable contents
  - Algorithms
  - Assumptions
  - Tricks
- Best practice: do not over-comment your code!
  - Obvious comments obfuscate code and annoy readers
  - **`// square root of discriminant`** is a bad example

Functions, **`main()`** in Particular

- C code is organized in functions
  - Each function has a name
  - Code goes in between braces
  - Arguments, if any, goes in between parentesis
  - It can return one or zero results using **`return`**
  - More on this later...
- In a program, the function **`main()`** can't be dispensed with
  - It's called automatically to execute the program
- **`main()`** returns an integer type value
  - A UNIX heritage
  - Passed to parent process (e.g. the *command shell*)
  - Rule: 0 if everything completed successfully

Variables

- **`double x1, x2;`** declares two variables
  - Named memory locations where values can be stored
  - Declared by specifying a data type followed by a comma-separated list of names, ended by a semicolon
  - On x86 CPUs, **`double`** means that **`x1`** and **`x2`** host IEEE double-precision (i.e. 64 bits) floating point values
- A legal *identifier* must be used for a variable name:
  - Permitted characters: **`a-z`**, **`A-Z`**, **`0-9`**, _
  - The first one cannot be a digit (e.g. **`x1`** is a valid identifier, **`1x`** is not)
  - 31 characters are guaranteed to be considered
  - A good advice: do not exceed 31 characters in an identifier
- Case counts: **`anIdent`** is not the same as **`anident`**!
- Common convention: avoid variable names entirely made of capital letters

Using the Standard Library

- A lot of functionalities are available in an external library of functions, whose content is defined by the Standard
- The compiler knows nothing about them, so it needs information about:
  - Arguments
  - Type of returned value
- Information about functions is in *header files*
  - Grouped by categories
  - Must be inserted in the source code before functions are used
  - **`#include`** causes the preprocessor to do it automatically
  - Specifying the header file name between angle brackets forces the preprocessor to look in the directories where the Standard header files are located
- Want to compute a square root?
  - **`#include <math.h>`**
  - Use **`sqrt()`**

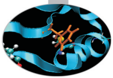
A Few First Words on I/O

- Related functions are grouped in `stdio.h`
- The bare minimum: textual input output from/to the user terminal
  - `scanf()` reads
  - `printf()` writes
- `printf("Solving ...");` is obvious
  - Writes the text between double quotes
- `printf("Real roots:  %lf, %lf\n", x1, x2);` is more interesting
  - Conversion specifiers `%lf` are substituted by the textual representation of values in `x1` and `x2`
  - And a new line is forced by `\n`
- `scanf("%lf ,%lf ,%lf", &a, &b, &c);`
  - Reads three double precision numbers from the terminal, converts them in internal binary format, stores them
  - Enough for now, disregard details

Expressions and Operators

- Most of program work takes place in expressions
- Operators compute values from terms
  - **`+`**, **`-`**, **`*`** (multiplication), and **`/`** behave like in “human” arithmetic
  - So do unary **`-`**, **`(`**, and **`)`**
- **`x1 = x1 + delta`** assigns the value of expression **`x1 + delta`** to variable **`x1`**
  - An ending **`;`** makes it into an executable *statement*
  - But it’s still an expression, with the same value assigned to **`x1`**
  - Thus we can write **`x1 = x2 = -b;`**, which is same as **`x1 = (x2 = -b);`**
- Practical shorthands to read/modify/write a variable:
  - **`x2 -= delta`** is same as **`x2 = x2 - delta`**
  - **`x2 /= 2.0*a`** is same as **`x2 = x2/(2.0*a)`**

Try It Now!




```
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>

int main() {
  double delta;
  double x1, x2;
  double a, b, c;

  printf("Solving ax^2+bx+c=0, enter a, b, c: ");
  scanf("%lf ,%lf ,%lf", &a, &b, &c);

  delta = sqrt(b*b - 4.0*a*c); // square root of discriminant
  x1 = x2 = -b;
  x1 = x1 + delta;
  x2 -= delta;
  x1 = x1/(2.0*a);
  x2 /= 2.0*a;

  printf("Real roots: %lf, %lf\n", x1, x2);

  return 0;
}
```

Compile your first C program !

- We will use GNU C Compiler (GCC) during this course
  - Other compilers are available on the market (Intel, PGI, Pathscale, etc)
  - Linux systems comes with the C compiler
  - Windows systems does not have a default one
    - we will use MinGW (a minimal port of GCC for Windows)
- Let's see how to compile and run your first C program:
  - put your first C code into **`main.c`** file
  - Compile your source code using the command:

```
user@cineca$> gcc main.c
```

    An executable file named **`a.out`** will be generated

  - Run the program with:

```
user@cineca$> ./a.out
```

Compile your first C program ! (II)

- ... probably you got something like this:

```
user@cineca$> gcc main.c
/tmp/ccWpSr3h.o: In function `main':
main.c:(.text+0xa8): undefined reference to `sqrt'
collect2: ld returned 1 exit status
```

  - **`#include<math.h>`** declares some math functions and constants (**`sqrt()`** among them)
  - the **`sqrt()`** function code is in the math library
  - **`gcc`** does not automatically link the math library
- you have to link the library explicitly into the executable:

```
user@cineca$> gcc main.c -lm
```

- now run the program!

Fixing a Problem

- User wants to solve $x^2 + 1 = 0$
  - Enters: **`1, 0, 1`**
  - Gets: **`Real roots: nan, nan`**
- Discriminant is negative, its square root is Not A Number, nan
- Let's avoid this, by changing from:

```
delta = sqrt(b*b - 4*a*c);
```

to:

```
delta = b*b - 4*a*c;
if (delta < 0.0)
  return 0;
delta = sqrt(delta);
```

- Try it now!
- Did you check that normal cases still work? Good.

Conditional Statement

- **`if`** **(***logical-condition***)** *statement*
  - Executes *statement* only if *logical-condition* is true
  - Comparison operators: **`==`** (equal), **`!=`** (not equal), **`>`**, **`<`**, **`>=`**, **`<=`**
- But our fix is not user friendly, let's be more polite by changing from:

```
if (delta < 0.0)
  return 0;
```

to:

```
if (delta < 0.0)
{
  printf("No real roots!\n");
  return 0;
}
```

- Try it now!
- Did you check that normal cases still work? Good.

Compound Statements

- Wherever a statement is legal in C, you can use a sequence of statements enclosed in braces
- Some folks prefer this:

```
if (delta < 0.0) {
  printf("No real roots!\n");
  return 0;
}
```

  and it's OK
- Some folks write:

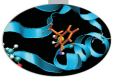
```
if (delta < 0.0) {printf("No real roots!\n"); return 0;}
```

  but this is not that good...
- In general, C disregards white space and line breaks, but indentation makes program control flow explicit

Let's Refactor Our Program


```
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>

int main() {
  double delta;
  double rp;
  double a, b, c;

  printf("Solving ax^2+bx+c=0, enter a, b, c: ");
  scanf("%lf ,%lf ,%lf", &a, &b, &c);

  delta = b*b - 4.0*a*c;
  if (delta < 0.0)
  {
    printf("No real roots!\n");
    return 0;
  }
  delta = sqrt(delta)/(2.0*a);

  rp = -b/(2.0*a);

  printf("Real roots: %lf, %lf\n", rp+delta, rp-delta);

  return 0;
}
```

And Now Make It More Complex!




```c
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>
#include <stdbool.h>

int main() {
  double delta;
  double rp;
  double a, b, c;
  bool rroots = true;

  printf("Solving ax^2+bx+c=0, enter a, b, c: ");
  scanf("%lf ,%lf ,%lf", &a, &b, &c);

  delta = b*b - 4.0*a*c;
  if (delta < 0.0)
  {
    delta = -delta;
    rroots = false;
  }
  delta = sqrt(delta)/(2.0*a);

  rp = -b/(2.0*a);

  if (rroots)
    printf("Real roots: %lf, %lf\n", rp+delta, rp-delta);
  else
    printf("Complex roots: %lf+%lfI, %lf-%lfI\n", rp, delta, rp, delta);

  return 0;
}
```

More Types and Choices

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

- **`bool`** represents logical values
  - C99 only
  - Actually an integer type in disguise
  - And most types would work, if it's non zero then it's true
- **`else`** has to match with an **`if ()`**, and the immediately following statement is executed when **`if ()`** logical condition is false
  - Allows for choosing between alternative paths
  - Again, a compound statement could be used
  - Again, use proper indentation
- By the way, variables can be initialized at declaration, as with **`rroots`**
- By the way, expressions can be passed as function arguments, as to **`printf()`**:
  their value will be computed and passed to the function

SCAI SuperComputing Applications and Innovation

As Complex as Possible!


```c
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>
#include <complex.h>

int main() {
  double complex delta;
  double complex z1, z2;
  double a, b, c;

  printf("Solving ax^2+bx+c=0, enter a, b, c: ");
  scanf("%lf ,%lf ,%lf", &a, &b, &c);

  delta = csqrt(b*b - 4.0*a*c);

  z1 = (-b+delta)/(2.0*a);
  z2 = (-b-delta)/(2.0*a);

  printf("Complex roots: %lf%+lfI, %lf%+lfI\n",
         creal(z1), cimag(z1), creal(z2), cimag(z2));

  return 0;
}
```

CINECA

Complex Numbers and Other Stuff

- C99 introduced the **`complex`** type
  - Include **`complex.h`**
  - All math and manipulation functions are defined
  - Use an expression to specify a constant, like **`1.0-2.0*I`**
  - In an older program that already defines its own **`complex`** type, use **`_Complex`** instead
- **`printf()`** doesn't know about complex numbers, yet
  - Output real and imaginary parts separately
- By the way, the **`+`** in conversion specifiers forces output of the sign, even if positive

Making It More Robust

- What if user inputs zeroes for *a*, or *a* and *b*?
- Let's prevent these cases, inserting right after input:

```
if (a == 0.0)
{
  if (b == 0.0)
    if (c == 0.0)
      fprintf(stderr, "A trivial identity!\n");
    else
      fprintf(stderr, "Plainly absurd!\n");
  else
    fprintf(stderr, "Too simple problem!\n");

  return -1;
}
```

- Can you see the program logic?
- Try it now!
- Did you check that normal cases still work? Good.

Miscellaneous Remarks

- Nested `if`s can be a problem
  - `else` always marries innermost `if`
  - Proper indentation is almost mandatory to sort it out
  - In doubt, put it in a compound statement: helps legibility too
- What's this `fprintf(stderr,...)` stuff?
  - `fprintf()` allows to specify an output file
  - `stderr` is a special file, mandatory for error messages to the user terminal
  - By the way, `printf(...)` is nothing more than `fprintf(stdout,...)`
  - And `scanf(...)` is nothing less than `fscanf(stdin,...)`
- Best practice: have your program always fail in a controlled way
- Convention: return negative values on failure
  - Use different values for different failures, so that a Unix shell script can test `$?` or `$status` and take action

A C Program is Made of: I

- Comments
  - Compiler disregards them, but humans do not
  - Please, use them
  - Do not abuse them, please
- Functions
  - One, at least: **`main()`**
  - Some of them come from the Standard Library
  - The proper header file must be **`#include`**d to use them
- Variables
  - Named memory locations you can store values into
  - Must be declared
- Variables declarations
  - Give name to memory location you can store values into
  - An initial value can be specified

A C Program is Made of: II

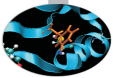
- Expressions
  - Compute values to store in variables
  - Compute values to pass to functions
- Statements
  - Units of work
  - Terminated by a `;`
- Compound statements (also said *blocks*)
  - Group a sequence of statements in a single entity
  - Wrapped in braces `{ }`
  - Do not need a terminating `;`

Program Flow Control

- `return` statements
  - Complete execution of the current function
  - Allow to return back a result
- Conditional statements
  - Allow conditional execution of code
  - Allow choice between alternate code paths

Best Practices

- Use proper indentation
  - Compilers don't care about
  - Readers visualize flow control
- Do non-regression testing
  - Whenever functionalities are added
  - Whenever you rewrite a code in a different way
- Fail in a controlled way
  - Giving feedback to humans
  - Giving feedback to the parent process



Scientific and Technical Computing in C

Day 1

Luca Ferraro   Stefano Tagliaventi

CINECA - SCAI Department

SuperComputing Applications and Innovation

Outline

My First C Functions


```
#include <math.h>


//Heaviside function, useful in DSP
double theta(double x) {

  if (x < 0.0)
    return 0.0;
  return 1.0;
}

//sinc function, as used in DSP
double sinc(double x) {
  const double pi = 3.141592653589793238;

  x = x*pi;
  if (x == 0.0)
    return 1.0;
  return sin(x)/x;
}

//generalized rectangular function, useful in DSP
double rect(double t, double tau) {

  t = fabs(t);
  tau = 0.5*tau;
  if (t = tau)
    return 0.5;
  return theta(tau - t);
}
```

Functions and Their Definition

- Like variables, functions have names and types
  - Name must be an identifier
  - Type is the type of the returned result
- They have an associated compound statement, the function "body"
- Functions have formal parameters
  - Declared in a comma separated list, in parentheses
  - Each one is like a variable declaration
  - In fact, they can be used like variables inside the function
- Parameters vs. *arguments*
  - "Arguments" are the actual values passed to a function when it is called
  - Formal parameters are the names used in the function to access these values

Function Parameters

- What if two functions have parameters with identical names?
  - No conflicts of sort, they are completely independent
- What if a parameter has the same name of a variable elsewhere in the program?
  - No conflicts of sort, they are completely independent
- Wait!
- What happens on assignment to a parameter?
  - Does something change in the calling function?
  - No!
- Arguments are passed *by value* in C
  - Parameters are like local variables, storing arguments values
  - Feel free to change their content as needed!

Miscellaneous Remarks

- The **`const`** qualifier
  - A **`const`** qualified variable can only be initialized
  - Compilers will bark if you try to change its value
- Best practice: always give name to constants
  - Particularly if unobvious, like `1.0/137.0`
  - It also helps to centralize updates (well, not for π)

- **`fabs()`** returns absolute value of a floating point number
  - Remember to **`#include <math.h>`**

- **`return`** ends function execution returning a result

- **`else`** isn't always needed
  - In this case, because **`return`** will end function execution anyway

On to Testing

- Let's put the code in a file named **`dsp.c`**
- Best practice: always put different groups of related functions in different files
  - Helps to tame complexity
  - You can always pass all source files to the compiler
  - And you'll learn to do better ...
- And let's write a program to test all functions
- Best practice: always write a special purpose program to test each subset of functions
  - Best to include in the program automated testing of all relevant cases
  - Let's do it by hand with I/O for now, to make it short

Try It Now!


```
#include <math.h>


//Heaviside function, useful in DSP
double theta(double x) {

  if (x < 0.0)
    return 0.0;
  return 1.0;
}

//sinc function, as used in DSP
double sinc(double x) {
  const double pi = 3.14159265358979323 8;

  x = x*pi;
  if (x == 0.0)
    return 1.0;
  return sin(x)/x;
}

//generalized rectangular function, useful in DSP
double rect(double t, double tau) {

  t = fabs(t);
  tau = 0.5*tau;
  if (t = tau)
    return 0.5;
  return theta(tau - t);
}
```

Testing DSP Functions

- we collect DSP functions in **`dsp.c`** source file
- we want to test these functions
- let's write a **`test_dsp.c`** program:

```
#include <stdio.h>

int main() {

    double t, tau;
    printf("Test DSP functions, enter t, tau: ");
    scanf("%lf, %lf", &t, &tau);

    printf("theta(%lf) = %lf\n", t, theta(t));
    printf("sinc(%lf) = %lf\n", t, sinc(t));
    printf("rect(%lf,%lf) = %lf\n", t, tau, rect(t,tau));

    return 0;
}
```

Testing DSP Functions (II)

- let's build our test program putting all together:

```
user@cineca$> gcc test_dsp.c dsp.c  -o test_dsp  -lm
```

  - **-lm** links the math library
  - **-o** gives the name **test_dsp** to the executable
- Now run the program:

```
user@cineca$> ./test_dsp
Test DSP functions, enter t, tau: 1., 1.

theta(1.000000) = 0.000000
sinc(1.000000) = 654810880.000000
rect(1.000000,1.000000) = 0.000000
```

-

Testing DSP Functions (III)

- results were incorrect since **`main`** function didn't know anything about our custom functions
- compiler assumed they all take and return integer types
- create and include a **`dsp.h`** header file in the main source file

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

int main() {
...
```

- now your compiler knows the right types for DSP functions arguments and return values:

```
user@cineca$> ./test_dsp
Test DSP functions, enter t, tau: 1., 1.
theta(1.000000) = 1.000000
sinc(1.000000) =  0.000000
rect(1.000000,1.000000) = 0.500000
```

- much better ...

Header File: **dsp.h**


```
#ifndef DSP_H
#define DSP_H
double theta(double x);
double sinc(double x);
double rect(double t, double tau);
#endif
```

- *Function prototypes* are function declarations: a **;** replaces the function body
  - Parameters names are optional, but can be informative
- If **DSP_H** is already defined, preprocessor will remove the code before compiler is invoked
- Best practices:
  - Always play the above trick: complex programs cause multiple inclusions of header files
  - Use all capitals identifiers for preprocessor symbols
  - Include **dsp.h** in **dsp.c** too: compiler will complain if you make them inconsistent

My First C Functions


```c
#include <math.h>
#include "dsp.h"

//Heaviside function, useful in DSP
double theta(double x) {

  if (x < 0.0)
    return 0.0;
  return 1.0;
}

//sinc function, as used in DSP
double sinc(double x) {
  const double pi = 3.141592653589793238;

  x = x*pi;
  if (x == 0.0)
    return 1.0;
  return sin(x)/x;
}

//generalized rectangular function, useful in DSP
double rect(double t, double tau) {

  t = fabs(t);
  tau = 0.5*tau;
  if (t = tau)
    return 0.5;
  return theta(tau - t);
}
```

Debugging `rect()`

- Everything fine with **`theta()`** and **`sinc()`**, but **`rect()`** behaves unexpectedly
    - If **`tau`** is zero, it always returns 1.0
    - If **`tau`** is non zero, it always returns 0.5
- Let's reread it carefully
- We wrote **=** where we actually meant **==**
    - Assignments are expressions, so **`tau`** value is returned
    - A zero means false to **`if ()`**
    - Anything different from zero means true to **`if ()`**
- Let's fix it and test again!
- Best practice:
    - Always enable compiler warnings and pay attention to them
    -

My First C Functions Fixed!


```c
#include <math.h>
#include "dsp.h"

//Heaviside function, useful in DSP
double theta(double x) {

  if (x < 0.0)
    return 0.0;
  return 1.0;
}

//sinc function, as used in DSP
double sinc(double x) {
  const double pi = 3.14159265358979323 8;

  x = x*pi;
  if (x == 0.0)
    return 1.0;
  return sin(x)/x;
}

//generalized rectangular function, useful in DSP
double rect(double t, double tau) {

  t = fabs(t);
  tau = 0.5*tau;
  if (t == tau)
    return 0.5;
  return theta(tau - t);
}
```

Compiler Errors and Warnings

- compiler stops on grammar and syntax violations
- goes on if you write code semantically absurd, but syntactically correct!
- compiler can perform extra checks and report warnings
  - very useful in early development phases
  - pinpoint “suspect” code... sometimes pedantically
  - read them carefully anyway
- **`-Wall`** option turns on all-warnings on **`gcc`**
- if only we used it earlier ...

```
user@cineca$> gcc -Wall -o test_dsp test_dsp.c dsp.c -lm
test_dsp.c: In function 'main':
test_dsp.c:9: warning: implicit declaration of 'theta'
test_dsp.c:10: warning: implicit declaration of 'sinc'
test_dsp.c:11: warning: implicit declaration of 'rect'
dsp.c: In function 'rect':
dsp.c:20: warning: suggest parentheses around assignment
  used as truth value
```

- something is an error for a selected C standard
  - use **`-std=c99`** to force C99 standard

Building a Program

Creating an executable from source files is a three step process:

- pre-processing:
  - each source file is read by the pre-processor
    - substitute (**`#define`**) MACROs
    - insert code per **`#include`** statements
    - insert or delete code according **`#ifdef`**, **`#if`** ...
- compiling:
  - each source file is translated into an object code file
  - an object code file contains global variables and functions defined in the code, as well as references to external ones
- linking:
  - object files are combined into a single executable file
  - every symbol should be resolved
    - symbols can be defined in your object files
    - or in other object code (Standard or external libraries)

Compiling and Linking with GCC

- when you give the command:

```
user@cineca$> gcc test_dsp.c dsp.c  -lm
```

- it's like going through three steps:
  - pre-processing: with **-E** option compiler stops after this stage
  - compiling: with **-c** compiler produces an object file **.o** without linking
  - linking object files together with external libraries

```
user@cineca$> gcc dsp.o test_dsp.o  -lm
```

Compiling and Linking with GCC

- In order to resolve symbols defined in external libraries, you have to specify:
  - which libraries to use (`-l` option)
  - in which directories they are (`-L` option)
- an example: let's use the library `/home/user/mylibs/libfoo.a`

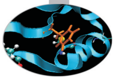
```
user@cineca$> gcc file1.o file2.o -L/home/user/mylibs -lfoo
```

  - we just use the name of the library for `-l` switch
- the DSP example:

```
user@cineca$> gcc dsp.o test_dsp.o -lm
```

  - the `sqrt()` function is contained in the `libm.a` library
  - the math library is part of the Standard C Library, thus resides in a directory the compiler already knows about

Managing Wrong Arguments



Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

```
#include <math.h>

double rect(double t, double tau) {

  t = fabs(t);
  tau = 0.5*fabs(tau); // fix for tau<0
  if (t == tau)
    return 0.5;

  return theta(tau - t);
}
```

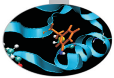
- What if **`rect()`** is passed a negative argument for **`tau`**?
  - Wrong results
- Taking the absolute value of **`tau`** it's a possibility
- But not a good one, because:
  - a negative rectangle width is nonsensical
  - probably flags a mistake in the calling code
  - and a zero rectangle width is also a problem

Failing Predictably


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

double rect(double t, double tau) {

  if (tau <= 0.0) {
    fprintf(stderr, "rect() invalid argument, tau: %lf\n", tau);
    exit(EXIT_FAILURE);
  }

  t = fabs(t);
  tau = 0.5*tau;
  if (t == tau)
    return 0.5;

  return theta(tau - t);
}
```

- A known approach...
- with a new twist!
  - **`return`** doesn't terminate programs unless in **`main()`**
  - **`exit()`** from **`stdlib.h`** works everywhere
  - **`-1`** may be used instead of **`EXIT_FAILURE`**, but is less portable

A More “Standard” Approach




```
#include <math.h>
#include <errno.h>

double rect(double t, double tau) {

  if (tau <= 0.0) {
    errno = EDOM;
    return 0.0;
  }

  t = fabs(t);
  tau = 0.5*tau;
  if (t == tau)
    return 0.5;

  return theta(tau - t);
}
```

- And a prudent user would check it, and use **`perror()`** from **`stdio.h`**, as in:

```
errno = 0;
a = rect(b, c);
if (errno)
{
  perror("rect():");
  //recovery action or controlled failure
}
```

- But there is more...

Total Robustness

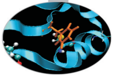
- Your platform could support IEEE floating point standard
  - Most common ones do, at least in a good part
- This means more bad cases:
  - one of the arguments is a NAN
  - both arguments are infinite (they are not ordered!)
- Best strategy: return a NAN and set `errno` in these bad cases
  - And do it also for non positive values of `tau`
  - But then the floating point environment configuration should be checked, proper floating point exceptions set...
- Being absolutely robust is difficult
  - Too advanced stuff to cover in this course
  - But not an excuse, some robustness is better than none
  - It's a process to do in steps
  - Always comment in your code bad cases you don't address yet!

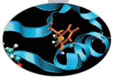
We Did Progress!

- Functions and their parameters
- Arguments are passed to functions by value
- A program can be subdivided in more source files
- Header files help to do it
- Preprocessor helps to write good header files
- Function prototypes
- **`const`** variables
- To **`if ()`**, zero is false and non zero is true
- Mistyping = for **==** is very dangerous
- **`exit()`** terminates a program
- **`errno`** is a standard way to report issues
- And **`perror()`** translates each issue for humans

Best Practices

- Name constants, do not use magic numbers in the code
- Group different sets of functionalities in different files
  - Helps to separate concerns and simplifies work
- Plan for header files to be included more than once
  - It happens, sooner or later and it's easy to take care of
- Use all capitals names to easily spot preprocessor symbols
- Test every function you write
  - Writing specialized programs to do it
- Use compilers and other tools to catch mistakes
- Anticipate causes of problems
  - Find a rational way to react
  - Fail predictably and in a standard way
  - The road to robustness is a long walk to do in steps
  - Comment issues still to be addressed in your code





Scientific and Technical Computing in C

Day 1

Luca Ferraro Stefano Tagliaventi

CINECA - SCAI Department

Outline

Greatest Common Divisor

- Euclid's Algorithm
  1. Take two integers a and b
  2. Let $r \leftarrow a \mod b$
  3. Let $a \leftarrow b$
  4. Let $b \leftarrow r$
  5. If b is not zero, go back to step 2
  6. a is the GCD
- Let's implement it and learn some more C

GCD & LCM


```c
#include "numbertheory.h"

// Greatest Common Divisor
int gcd(int a, int b) {
  do {
    int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
int lcm(int a, int b) {

  return a*b/gcd(a,b);
}
```

The Integer Type

- **`int`** means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, **`int`** means a 32 bits value, ranging from -2^{31} to $2^{31} - 1$
- Want to know the actual size?
  - **`sizeof(int)`** will return the size in bytes of the internal binary representation of type **`int`**
- Want to know more? **`#include <limits.h>`**
  - **`INT_MAX`** is the greatest positive value an **`int`** can assume
  - **`INT_MIN`** is the most negative value an **`int`** can assume
  - These are preprocessor macros expanding to literal constants (more on this later...)
- Want to convert to/from textual decimal representation?
  - Use conversion specifier **`%d`** in **`printf()`** format string
  - Use conversion specifier **`%d`** in **`scanf()`** format string

Iterating with `do ... while ()`

- **`do`**
  statement
  `while (`*logical-condition***`)`**
  1. Executes *statement*
  2. Evaluates *logical-condition*
  3. If *logical-condition* is true (i.e. not zero), goes back to 1
  4. If *logical-condition* is false, proceeds to execute the following code
- **`while (b)`** will also do, but **`while (b != 0)`** is more readable and costs no more CPU work
- What's this variable declaration here?
  - **`t`** can only be used inside the block it is declared into
  - I.e. its *scope* is limited to the block it is declared into
  - It's not special to **`do...while ()`**,it works in any compound statement

Iterating with `while ()`

- **`while (`***logical-condition***`)`**
  statement
  1. Evaluates *logical-condition*
  2. If *logical-condition* is false (i.e. zero), goes to 5
  3. Executes *statement*
  4. Goes back to 1
  5. Skips *statement* and proceeds to execute the following code
- **`while ()`** is very similar to **`do ... while ()`**, but the latter always performs at least one iteration

Time for Testing

- Put the code in file **`numbertheory.c`**
- Write a suitable **`numbertheory.h`**
- Write a program to test both **`gcd()`** and **`lcm()`** on a pair of integer numbers
- Remember using **`%d`** for I/O
- Test it:
  - with pairs of small positive integers
  - with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
- In some cases, we get wrong results or runtime errors
  - Euclid's algorithm is only defined for positive integers

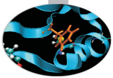
GCD & LCM: Try It Now!


```
#include "numbertheory.h"

// Greatest Common Divisor
int gcd(int a, int b) {
  do {
    int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
int lcm(int a, int b) {

  return a*b/gcd(a,b);
}
```

Let’s Fix It...

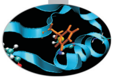
- Best way: generalize algorithm to the whole integer set
- gcd(a, b) is non negative, even if a or b is less than zero
  - Taking the absolute value of `a` and `b` using `abs()` will do
- gcd($a, 0$) is $|a|$
  - Conditional statements will do
- gcd($0, 0$) is 0
  - Already covered by the previous item, but let’s pay attention to `lcm()`
- By the way, `&&` is the logical AND of two logical conditions
- Try and test it:
  - with pairs of small positive integers
  - with the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
  - and with the pair: 1000000, 1000000

GCD & LCM: Dealing with 0 and Negatives




```c
#include <stdlib.h>
#include "numbertheory.h"

// Greatest Common Divisor
int gcd(int a, int b) {

  a = abs(a);
  b = abs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
int lcm(int a, int b) {

  if (a == 0 && b == 0)
    return 0;
  return a*b/gcd(a,b);
}
```

Beware of Type Ranges



- **`a*b/gcd(a,b)`** same as **`(a*b)/gcd(a,b)`**
- What if the result of a calculation cannot be represented in the given type?
  - Technically, you get an arithmetic *overflow*
  - C is quite liberal: the result is implementation defined
  - Best practice: be very careful of intermediate results
- Easy fix: $\gcd(a, b)$ is an exact divisor of b
- Try and test it:
  - with pairs of small positive integers
  - on the following pairs: 15, 18; -15, 18; 15, -18; -15, -18; 0, 15; 15, 0; 0, 0
  - with the pair: 1000000, 1000000
  - and let's test also with: 1000000, 1000001

GCD & LCM: Avoiding an Overflow


```
#include <stdlib.h>
#include "numbertheory.h"

// Greatest Common Divisor
int gcd(int a, int b) {

  a = abs(a);
  b = abs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
int lcm(int a, int b) {

  if (a == 0 && b == 0)
    return 0;
  return a*(b/gcd(a,b));
}
```

Wider Integer Types

- Sometimes an integer type with a wider range of values is needed
- `long int` (commonly shortened to `long`)
  - `LONG_MAX` and `LONG_MIN` from `limits.h`
  - `%ld` conversion specifier in `printf()` and `scanf()`
  - But C Standard only says: can't be narrower than an `int`
  - In practice, it can be 32 or 64 bits wide, depending on platform and compiler
  - As usual, use `sizeof(long int)` to check
- C99 `long long int` (shortened to `long long`)
  - `LLONG_MAX` and `LLONG_MIN` from `limits.h`
  - `%lld` conversion specifier in `printf()` and `scanf()`
  - C99 Standard requires: must be at least 64 bits wide!
  - As usual, use `sizeof(long long)` to check if you got more than that

GCD & LCM: Wider Integers


```
#include <stdlib.h>
#include "numbertheory.h"

// Greatest Common Divisor
long long int gcd(long long int a, long long int b) {

  a = llabs(a);
  b = llabs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    long long int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
long long int lcm(long long int a, long long int b) {

  if (a == 0 || b == 0)
    return 0;
  return a*(b/gcd(a,b));
}
```

Call the Right Function!

- We had to call different functions for absolute value
  - **`labs()`** for **`long int`**s
  - **`llabs()`** for **`long long int`**s
- What if you call, say, **`labs()`** for **`int`** or **`long long`** values?
  - Automatic conversion between different types happens!
  - But a narrower type cannot represent all possible values of a wider one
  - No problem when converting to a wider type
  - At risk of overflow (i.e. implementation defined surprise) when converting to a narrower one
  - Best practice: enable compiler warnings or use tools like **`lint`** to catch mistakes

Unsigned Integer Types

- **`unsigned int`** (often shortened to **`unsigned`**)
  - Same width as an **`int`**
  - No negative values, only positive integers, but nearly twice the ones in an **`int`**
  - **`UINT_MAX`** (from **`limits.h`**) is its greatest value
  - Use conversion specifier **`%u`** in **`printf()`** and **`scanf()`**
- And there are more unsigned types...
  - Like **`unsigned long`** and **`unsigned long long`**
  - **`ULONG_MAX`** and **`ULLONG_MAX`** from **`limits.h`**
  - **`%lu`** and **`%llu`** in **`printf()`** and **`scanf()`**
- No arithmetic overflows!
  - C Standard requires arithmetic in any unsigned type to be exact modulo $2^{\textit{type width in bits}}$
- Beware of signed to/from unsigned conversions!
  - Negative values cannot be represented in an unsigned
  - And vice versa for the biggest half of unsigned values
  - You are in for implementation defined surprises!

A Couple of Issues

- Best practice: avoid useless work
  - **`a*(b/gcd(a,b))`** causes error if both **`a`** and **`b`** are zero
  - but it's useless anyway if **`a`** or **`b`** is zero, let's use **`||`** (logical OR) to avoid it
- Best practice: be loyal to C approach
  - You have now a **`gcd()`** function that works on the widest available integer type
  - And you could use it safely for narrower types
  - But at the cost of getting compiler warnings, even if you do it correctly
  - And this is not the C way (think of **`abs()`**, **`labs()`**, **`llabs()`**)
- Let's try an easy solution

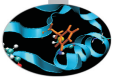
GCD Flavors


```
#include <stdlib.h>
#include "numbertheory.h"

// Greatest Common Divisor
long long int llgcd(long long int a, long long int b) {

  a = llabs(a);
  b = llabs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    long long int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

long int lgcd(long int a, long int b) {
  return (long int)llgcd((long long int)a, (long long int)b);
}

int gcd(int a, int b) {
  return (int)llgcd((long long int)a, (long long int)b);
}
```

Getting in Control of Type Conversions

- `(`*type*`)` *expression*
  - Is an *explicit cast*
  - Forces conversion from expression type to specified one
  - And tells the compiler you know what you are doing
- The solution is not perfect
  - If you are working with a lot of basic **`int`**s, you are spending a lot of work in type conversions and wider than necessary arithmetic
  - And there are more integer types we didn't mention yet...
- Writing specialized copies is not an option
  - If you want to change something, you have to make the same change in different places
  - Best practice: avoid replicating similar code
- The preprocessor can generate specialized function copies for you

GCD: 3 for the Price of 1




```
#include <stdlib.h>
#include "numbertheory.h"

#define GGCD(TYPE,PREFIX) \
TYPE PREFIX ## gcd(TYPE a, TYPE b) { \
  a = PREFIX ## abs(a); \
  b = PREFIX ## abs(b); \
  if (a == 0) \
    return b; \
  if (b == 0) \
    return a; \
  do {\
    TYPE t = a % b; \
    a = b; \
    b = t; \
  } while (b); \
  return a; \
}

#define GLCM(TYPE,PREFIX) \
TYPE PREFIX ## lcm(TYPE a, TYPE b) { \
  if (a == 0 || b == 0) \
    return 0; \
  return a*(b/PREFIX ## gcd(a,b)); \
}

GGCD(int,)
GGCD(long int, l)
GGCD(long long int, ll)

GLCM(int,)
GLCM(long int, l)
GLCM(long long int, ll)
```

Generating Code With Macros

- Preprocessor macros
  - Their content is substituted wherever the macros appear in the code
  - Every occurrence of each parameter is replaced by the text given as argument
- A macro must be a "one-liner"
  - A \ at end of line is needed to continue on the next line
- The **##** operator concatenates two neighbouring tokens
  - As if they had been typed with no space in between
- Six functions are defined by *macro expansion*

```
int gcd(int a, int b)
long int lgcd(long int a, long int b)
long long int llgcd(long long int a, long long int b)
int lcm(int a, int b)
long int llcm(long int a, long int b)
long long int lllcm(long long int a, long long int b)
```

- Beware: debugging macros can be difficult

C11 Type-Generic Macros

- Still, unlike in higher level languages, you have to remember the right function name to invoke according to argument types
- C11 has a better way:

```
#define gcd(A, B) _Generic((A),                \
                         int: gcd              \
                         long int: lgcd        \
                         long long int: llgcd   \
                         ) (A, B)

#define lcm(A, B) _Generic((A),                \
                         int: lcm              \
                         long int: llcm        \
                         long long int: lllcm   \
                         ) (A, B)
```

- Now you can use **`gcd()`** and **`lcm()`** for all argument types
- Coming to a compiler near you...

More Types and Flow Control

- There are many integer types
  - With implementation dependent ranges
  - Range limits are defined in **`limits.h`**
  - **`sizeof(`***type***`)`** can be used to know their size in bytes
- Automatic type conversions take place
  - And can be controlled with explicit casts
- Different library functions for different types
  - Ditto for **`printf()`** and **`scanf()`** conversion specifiers
- Behavior on integer overflow is implementation defined
  - Some control is possible using parentheses
- Variables can be declared inside a block
  - Limiting access to the block scope
- Sequence of statements can be iterated according to a logical condition
- Logical conditions can be combined using **`||`** (OR) and **`&&`** (AND) operators

Best Practices

- Do not rely on type sizes, they are implementation dependent
- Think of intermediate results in expressions: they can overflow or underflow
- Unintended implicit conversions can take you by surprise
  - Put compiler warnings and specialized tools to good use
- Avoid unnecessary computations
- Avoid code replication
- Be consistent with C approach
  - Even if it costs more work
  - Even if it costs learning more C
  - Once again, you can do it in steps
  - You'll appreciate it in the future

Scientific and Technical Computing in C

Day 1

Luca Ferraro Stefano Tagliaventi

CINECA - SCAI Department

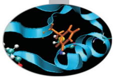
Outline

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

Data

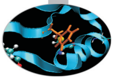
- Computing == manipulating data and calculating results
  - Data are manipulated using internal, binary formats
  - Data are kept in memory locations and CPU registers
- C is quite liberal on internal data formats
  - Most CPU are similar but all have peculiarities
  - C only mandates what is *de facto* standard
  - Some details depend on the specific executing (a.k.a. target) hardware architecture and software implementation
  - C Standard Library provides facilities to translate between internal formats and human readable ones
- C allows programmers to:
  - think in terms of data types and named containers
  - disregard details on actual memory locations and data movements

C is a Strongly Typed Language



- Each literal constant has a type
  - Dictates internal format of the data value
- Each variable has a type
  - Dictates content internal format and amount of memory
  - Type must be specified in a declaration before use
- Each expression has a type
  - And subexpressions have too
  - Depends on operators and their arguments
- Each function has a type
  - That is the type of the returned value
  - Specified in function declaration or definition
  - If the compiler doesn't know the type, it assumes **`int`**
- Function parameters have types
  - I.e. type of arguments to be passed in function calls
  - Specified in function declaration or definition
  - If the compiler doesn't know the types, it will accept any argument, applying some type conversion rules

Integer Types (as on Most CPUs)



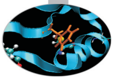
Type	Sign	Conversion	Width (bits)		Size (bytes)	
			Minimum	Usual	Minimum	Usual
`signed char`	+/-	`%hhd`[1]	8	8	1	1
`unsigned char`	+	`%hhu`[1]				
`short` `short int`	+/-	`%hd`	16	16	2	2
`unsigned short` `unsigned short int`	+	`%hu`				
`int`	+/-	`%d`	16	32	2	4
`unsigned` `unsigned int`	+	`%u`				
`long` `long int`	+/-	`%ld`	32	32 or 64	4	4 or 8
`unsigned long` `unsigned long int`	+	`%lu`				
`long long`[2] `long long int`[2]	+/-	`%lld`	64	64	8	8
`unsigned long long`[2] `unsigned long long int`[2]	+	`%llu`				

Constraint: **`short`** width ≤ **`int`** width ≤ **`long`** width ≤ **`long long`** width

1. C99, in C89 use conversion to/from `int` types
2. C99

- New platform/compiler? Always check with **`sizeof(`*`type`*`)`**
- Values of **`char`** and **`short`** types just use less memory, they are promoted to **`int`** types in calculations

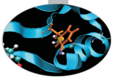
#include <limits.h>



Name	Meaning	Value
CHAR_BIT	width of any **char** type	≥ 8
SCHAR_MIN	minimum value of **signed char**	≤ -127
SCHAR_MAX	maximum value of **signed char**	≥ 127
UCHAR_MAX	maximum value of **unsigned char** type	≥ 255
SHRT_MIN	minimum value of **short**	≤ -32767
SHRT_MAX	maximum value of **short**	≥ 32767
USHRT_MAX	maximum value of **unsigned short**	≥ 65535
INT_MIN	minimum value of **int**	≤ -32767
INT_MAX	maximum value of **int**	≥ 32767
UINT_MAX	maximum value of **unsigned**	≥ 65535
LONG_MIN	minimum value of **long**	≤ -2147483647
LONG_MAX	maximum value of **long**	≥ 2147483647
ULONG_MAX	maximum value of **unsigned long**	≥ 4294967295
LLONG_MIN	minimum value of **long long**	$\leq -9223372036854775807$
LLONG_MAX	maximum value of **long long**	≥ 9223372036854775807
ULLONG_MAX	maximum value of **unsigned long long**	$\geq 18446744073709551615$

- Use them to make code more portable across platforms
- New platform/compiler? Always check values

Integer Literal Constants



- Constants have types too
- Compilers must follow precise rules to assign types to integer constants
  - But they are complex
  - And differ among standards
- Rule of thumb:
  - write the number as is, if it is in **`int`** range
  - otherwise, use suffixes **`U`**, **`L`**, **`UL`**, **`LL`**, **`ULL`**
  - lowercase will do as well, but **`l`** is easy to misread as **`1`**
- Remember: do not write **`spokes = bycicles*2*36;`**
  - **`#define SPOKES_PER_WHEEL 36`**
  - or declare:
    `const int SpokesPerWheel = 36;`
  - and use them, code will be more readable, and you'll be ready for easy changes

Integer Types Math

- **`#include <stdlib.h>`** to use:

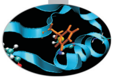
Function	Returns
`abs()`	absolute value of an **`int`**
`labs()`	absolute value of a **`long`**
`llabs()`	absolute value of a **`long long`**

- Use like: **`a = abs(b+i) + c;`**
- For values of type **`short`** or **`char`**, use **`abs()`**

Bitwise Arithmetic

- Integer types are encoded in binary format
  - Each one is a sequence of bits, each having state 0 or 1
  - Bitwise arithmetic manipulates state of each bit
- Each bit of the result of unary operator `~` is in the opposite state of the corresponding bit of the operand
- Each bit of the result of binary operators `|`, `&`, and `^` is the OR, AND, and XOR respectively of the corresponding bits in the operands
- Precedence
  - `a&b | c^d&e` same as `(a&b) | (c^(d&e))`
  - `~a&b` same as `(~a)&b`
- Associativity is from left to right
  - `a | b | c` same as `(a | b) | c`
- As usual, precedence and associativity can be overridden using explicit `(` and `)`, and `|=`, `&=`, and `^=` are available

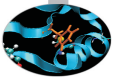
Enumerated Types




```
enum boundary {
  free_slip,
  no_slip,
  inflow,
  outflow
  };

enum boundary leftside, rightside;

enum liquid {water, mercury} fluid; //may confuse readers

leftside = free_slip;
```

- A set of integer values represented by identifiers
  - Under the hood, it's an **`int`**
  - **`free_slip`** is an *enumeration* ***constant*** with value 0
  - **`no_slip`** is an enumeration constant with value 1
  - **`inflow`** is an enumeration constant with value 2
  - ...

Choosing Values for Enumeration Constants




```
enum spokes {SpokesPerWheel = 36};

enum element {
  hydrogen = 1,
  helium,
  carbon = 6,
  oxygen = 8,
  fluorine
  };
```

- Enumeration constants can be given a specified value
- When the enumeration constant value is not specified:
  - if it's the first in the declaration, gets the value 0
  - if it's not, gets (*value of the previous one*+1)
  - thus **`helium`** above gets 2, and **`fluorine`** gets 9
  - negative values can be used too
- A convenient way to give names to related integer constants

Floating Types (as on Most CPUs)

Type	Conversion	Width (bits)	Size (bytes)
		Usual	Usual
`float`	**`%f`**, **`%E`**, **`%G`**[2]	32	4
`double`	**`%lf`**, **`%lE`**, **`%lG`**[2]	64	8
`long double`	**`%Lf`**, **`%LE`**, **`%LG`**[2]	80 or 128	10 or 16
`float _Complex`[1]	*none*	NA	8
`double _Complex`[1]	*none*	NA	16
`long double _Complex`[1]	*none*	NA	20 or 32

Constraints:
all **`float`** values must be representable in **`double`**
all **`double`** values must be representable in **`long double`**

1. C99
2. **`%f`** forces decimal notation, **`%E`** forces exponential decimal notation, **`%G`** chooses the one most suitable to the value

- New platform/compiler? Always check with **`sizeof(`*`type`*`)`**
- In practice, always in IEEE Standard binary format, but not a C Standard requirement
- **`#include <complex.h>`** and use **`float complex`**, **`double complex`**, and **`long double complex`**, if your program does not already uses the **`complex`** identifier

#include <float.h>

Name	Meaning	Value
FLT_EPSILON	$min\{x \| 1.0 + x > 1.0\}$ in **float** type	$\leq 10^{-5}$
DBL_EPSILON	$min\{x \| 1.0 + x > 1.0\}$ in **double** type	$\leq 10^{-9}$
LDBL_EPSILON	$min\{x \| 1.0 + x > 1.0\}$ in **long double** type	$\leq 10^{-9}$
FLT_DIG	decimal digits of precision in **float** type	≥ 6
DBL_DIG	decimal digits of precision in **double** type	≥ 10
LDBL_DIG	decimal digits of precision in **long double** type	≥ 10
FLT_MIN	minimum normalized positive number in **float** range	$\leq 10^{-37}$
DBL_MIN	minimum normalized positive number in **long** range	$\leq 10^{-37}$
LDBL_MIN	minimum normalized positive number in **long double** range	$\leq 10^{-37}$
FLT_MAX	maximum finite number in **float** range	$\geq 10^{37}$
DBL_MAX	maximum finite number in **long** range	$\geq 10^{37}$
LDBL_MAX	maximum finite number in **long double** range	$\geq 10^{37}$
FLT_MIN_10_EXP	minimum x such that 10^x is in **float** range and normalized	≤ -37
DBL_MIN_10_EXP	minimum x such that 10^x is in **double** range and normalized	≤ -37
LDBL_MIN_10_EXP	minimum x such that 10^x is in **long double** range and normalized	≤ -37
FLT_MAX_10_EXP	maximum x such that 10^x is in **float** range and finite	≥ 37
DBL_MAX_10_EXP	maximum x such that 10^x is in **double** range and finite	≥ 37
LDBL_MAX_10_EXP	maximum x such that 10^x is in **long double** range and finite	≥ 37

- Use them to make code more portable across platforms
- New platform/compiler? Always check values
- “Normalized”? Yes, IEEE Standard allows for even smaller values, with loss of precision, and calls them “denormalized”
- “Finite”? Yes, IEEE Standard allows for infinite values

Floating Literal Constants

- Need something to distinguish them from integers
  - Decimal notation: `1.0`, `-17.`, `.125`, `0.22`
  - Exponential decimal notation: `2E19` (2×10^{19}), `-123.4E9` (-1.234×10^{11}), `.72E-6` (7.2×10^{-7})
- They have type `double` by default
  - Use suffixes `F` to make them `float` or `L` to make them `long double`
  - Lowercase will do as well, but `l` is easy to misread as `1`
- Never write `charge = protons*1.602176487E-19;`
  - `#define UNIT_CHARGE 1.602176487E-19`
  - or declare:
    `const double UnitCharge = 1.602176487E-19;`
  - and use them in the code to make it readable
  - it will come handier when more precise measurements will be available

double Math

Function/Macro	Returns
HUGE_VAL[1]	largest positive finite value
INFINITY[1]	positive infinite value
NAN[1]	IEEE quiet NaN (if supported)
double fabs(double x),	$\lvert\mathbf{x}\rvert$,
double copysign(double x, double y)[1]	if $\mathbf{y} \neq 0$ returns $\lvert\mathbf{x}\rvert \mathbf{y}/\lvert\mathbf{y}\rvert$ else returns $\lvert\mathbf{x}\rvert$
double floor(double x), double ceil(double x), **double trunc(double x)**[1], **double round(double x)**[1]	$\lfloor\mathbf{x}\rfloor$, $\lceil\mathbf{x}\rceil$, if $\mathbf{x} > 0$ returns $\lfloor\mathbf{x}\rfloor$ else returns $\lceil\mathbf{x}\rceil$, nearest[2] integer to $\mathbf{x}$
double fmod(double x, double y),	$\mathbf{x}$ mod $\mathbf{y}$ (same sign as $\mathbf{x}$)
double fdim(double x, double y)[1]	if $\mathbf{x} > \mathbf{y}$ returns $\mathbf{x} - \mathbf{y}$ else returns 0
double nextafter(double x, double y)[1]	next representable value after $\mathbf{x}$ toward $\mathbf{y}$
double fmin(double x, double y)[1] **double fmax(double x, double y)**[1]	$\min\{\mathbf{x}, \mathbf{y}\}$ $\max\{\mathbf{x}, \mathbf{y}\}$

1. C99
2. If **x** is halfway, returns the farthest from 0

- **#include <math.h>**
- Before C99, there were no **fmin()** or **fmax()**
  - Preprocessor macros have been widely used to this aim
  - Use the new functions, instead
- More functions are available to manipulate values
  - Mostly in the spirit of IEEE Floating Point Standard
  - We encourage you to learn more about

`double` Higher Math

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

Functions	Return
`double sqrt(double x)`, `double cbrt(double x)`[1], `double pow(double x, double y)`, `double hypot(double x, double y)`[1]	$\sqrt{\mathbf{x}}$, $\sqrt[3]{\mathbf{x}}$, $\mathbf{x}^{\mathbf{y}}$, $\sqrt{\mathbf{x}^2 + \mathbf{y}^2}$
`double sin(double x)`, `double cos(double x)`, `double tan(double x)`, `double asin(double x)`, `double acos(double x)`, `double atan(double x)`	Trigonometric functions
`double atan2(double x, double y)`	Arc tangent in $(-\pi, \pi]$
`double exp(double x)`, `double log(double x)`, `double log10(double x)`, `double expm1(double x)`[1], `double log1p(double x)`[1]	$e^{\mathbf{x}}$, $\log_e \mathbf{x}$, $\log_{10} \mathbf{x}$, $e^{\mathbf{x}} - 1$, $\log(\mathbf{x}+1)$
`double sinh(double x)`, `double cosh(double x)`, `double tanh(double x)`, `double asinh(double x)`[1], `double acosh(double x)`[1], `double atanh(double x)`[1]	Hyperbolic functions
`double erf(double x)`[1]	error function: $\frac{2}{\sqrt{\pi}} \int_0^{\mathbf{x}} e^{-t^2} dt$
`double erfc(double x)`[1]	$1 - \frac{2}{\sqrt{\pi}} \int_0^{\mathbf{x}} e^{-t^2} dt$
`double tgamma(double x)`[1], `double lgamma(double x)`[1]	$\Gamma(\mathbf{x})$, $\log(\lvert\Gamma(\mathbf{x})\rvert)$
1. C99	

- Again, `#include <math.h>`

double complex Math C99 & C11

Function/Macro	Returns
`double complex CMPLX(double x, double y)` [1]	$\mathbf{x} + i\mathbf{y}$,
`double complex cabs(double complex z)`,	$\|\mathbf{z}\|$,
`double complex carg(double complex z)`,	Argument of **z** (a.k.a. phase angle),
`double complex creal(double complex z)`,	Real part of **z**,
`double complex cimag(double complex z)`,	Imaginary part of **z**,
`double complex conj(double complex z)`	Complex conjugate of **z**
`double complex csqrt(double complex z)`,	$\sqrt{\mathbf{z}}$,
`double complex cpow(double complex z, double complex w)`	$\mathbf{z}^{\mathbf{w}}$
`double complex cexp(double complex z)`,	$e^{\mathbf{z}}$,
`double complex clog(double complex z)`	$\log_e \mathbf{z}$

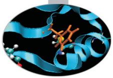
1. C11

- To use them, `#include <complex.h>`
  - You'll also get:
    `csin()`, `ccos()`, `ctan()`,
    `casin()`, `cacos()`, `catan()`,
    `csinh()`, `ccosh()`, `ctanh()`,
    `casinh()`, `cacosh()`, `catanh()`
  - And `I` for the imaginary unit

float and **long double** Math

- Before C99, all functions were only for **doubles**
  - And automatic conversion of other types was applied
- But from 1999 C is really serious about floating point math
  - All functions exist also for **float** and **long double**
  - Same names, suffixed by **f** or **l**
  - Like **acosf()** for arccosine of a **float**
  - Or **cacosl()** for **long double complex**
  - Ditto for macros, like **HUGE_VALF** or **CMPLXL()**
- If you find this annoying (it is!):
  - **#include <tgmath.h>**
  - and use everywhere, for all real and complex types, function names for **double** type
  - These are clever type generic processor macros, expanding to the function appropriate to the argument

Expressions



- A fundamental concept in C
  - A very rich set of operators
  - Almost everything is an expression
  - Even assignment to a variable
- C expressions are complicated
  - Expressions can have side effects
  - Not all subexpressions are necessarily computed
  - Except for associativity and precedence rules, order of evaluation of subexpressions is up to the compiler
  - Values of different type can be combined, and a result produced according to a rich set of rules
  - Sometimes with surprising consequences
- We'll give a simplified introduction
  - Subtle rules are easily forgotten
  - Relying on them makes the code difficult to read
  - When you'll find a puzzling piece of code, you can always look for a good manual or book

Arithmetic Expressions

- Binary operators **+**, **−**, ***** (multiplication) and **/** have the usual meaning and behavior
- Unary operator **−** evaluates to the opposite of its operand
- Unary operator **+** evaluates to its operand
- Precedence
  - `-a*b + c/d` same as `((-a)*b) + (c/d)`
  - `-a + b` same as `(-a) + b`
- Associativity of binary ones is from left to right
  - `a + b + c` same as `(a + b) + c`
  - `a*b/c*d` same as `((a*b)/c)*d`
- Explicit `(` and `)` override precedence and associativity
- Only for integer types, `%` is the modulo operator (`27%4` evaluates to 3), same precedence as `/`

Hitting Limits

- All types are limited in range
- What about:
  - **`INT_MAX + 1`**? (too big)
  - **`INT_MIN*3`**? (too negative)
- Technically speaking, this is an arithmetic *overflow*
- And division by zero is a problem too
- For signed integer types, the Standard says:
  - behavior and results are unpredictable
  - i.e. up to the implementation
- For other types, the Standard says:
  - arithmetic on unsigned integers must be exact modulo $2^{\textit{type width}}$, no overflow
  - with floating types, is up to the implementation (you can get **`DBL_MAX`**, or a NaN, or an infinity)
- Best practice: NEVER rely on behaviors observed with a specific architecture and/or compiler

Assignment Operator

- Binary operator =
  - assigns the value of the right operand to the left operand
  - and returns the value of the right operand
  - thus `a = b*2` is an expression with value `b*2` and the side effect of changing variable `a`
  - `a = b*2;` is an assignment statement
- The left operand must be something that can store a value
  - In C jargon, an *lvalue*
  - `a = 20` is OK, if `a` is a variable
  - `20 = a` is not
- Precedence is lowest (except for `,` operator) and associativity is from right to left
  - `a = b*2 + c` same as `a = (b*2 + c)`
  - `z = a = b*2 + c` same as `z = (a = (b*2 + c))`
- You'll read the latter form, particularly in `while ()` statements, but avoid writing it

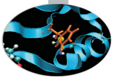
More Assignment Operators

- Most binary operators offer useful shortcut forms:

Expression	Same as
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a*b`
`a /= b`	`a = a/b`
`a %= b`	`a = a%b`

- In heroic times, used to map some CPUs optimized instructions
- With nowadays optimizing compilers, only good to spare keystrokes
- You'll find them often, particularly in `for(;;)` statements

More Side Effects



- Pre-increment/decrement unary operators: `++` and `--`
  - `++i` same as `(i = i + 1)`
  - `--i` same as `(i = i - 1)`
- Post-increment/decrement unary operators: `++` and `--`
  - `i++` increments `i` content, but returns the original value
  - `i--` decrements `i` content, but returns the original value
- Operand must be an *lvalue*
- Precedence is highest
- Quite handy in `while ()` and `for (;;)` statements
- Easily becomes a nightmare inside expressions
  - Particularly when you change the code

Order of Subexpressions Evaluation

- `i` is an `int` type variable whose value is `5`

```
j = 4*i++ - 3*++i;
foo(++i, ++i);
```

- Which value is assigned to `j`?
  - Could be
  - Or could as well be
- Which values are passed to `foo()`?
  - Could be `foo( ,  )`
  - Or could as well be `foo( ,  )`
- Order of evaluation of subexpressions is implementation defined!
- Ditto for order of evaluation of function arguments!
- NEVER! NEVER pre/post-in/de-crement the same variable twice in a single expression, or function call!

Logical Expressions

- Comparison operators
  - `==` (equal), `!=` (not equal), `>`, `<`, `>=`, `<=`
  - Compare operand values
  - Return `int` type 0 if evaluation is false, 1 if true
  - Precedence lower than arithmetic operators, higher than bitwise and logical operators
  - In doubt, add parentheses, but be sober

- Logical operators
  - `!` is unary NOT, `&&` is binary AND, `||` is binary OR
  - Zero operand are considered false, non zero ones true
  - Return `int` type 0 if comparison is false, 1 if true
  - Precedence of `!` just lower than `++` and `--`
  - `&&`, `||`: higher than = and friends
  - `!a&&b || a&&!b` means `((!a)&&b) || (a&&(!b))`
  - Again: in doubt, add parentheses, but be sober

More Logic from `math.h`

- Some macros to tame floating point complexity
- **`isfinite()`**
  - True if argument value is finite
- **`isinf()`**
  - True if argument value is an infinity
- **`isnan()`**
  - True if argument value is a NaN
- And more, if you are really serious about floating point calculations
  - Mostly in the spirit of IEEE Floating Point Standard
  - Learn more about it, before using them

Being Completely Logical

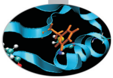
- C99 defines integer type `_Bool`
  - Only guaranteed to store 0 or 1
  - Perfect for logical (a.k.a. boolean) expressions
  - Use it for “flag” variables, and to avoid surprises
  - Better yet, `#include <stdbool.h>`, and use type `bool`, and values `true` and `false`
- Watch your step!
  - Simply mistype `&` for `&&` or vice versa
  - Simply mistype `||` for `|`
  - You’ll discover, possibly after hours of debugging, that (bitwise arithmetic) `!=` (logical arithmetic)
- C99 offers a fix to this unfortunate choice
  - `#include <iso646.h>`
  - And use `not`, `or`, and `and` in place of `!`, `||` and `&&`

Even More Side Effects

- Right operand of `||` and `&&` is evaluated after left one
- And is not evaluated at all if:
  - left one is found true for an `||`
  - left one is found false for an `&&`
- Beware of “short circuit” evaluation...
  - ... if the right operand is an expression with side effects!
  - A life saver in preprocessor macros and a few more cases
  - But makes your code less readable
  - Use nested `if ()` whenever you can

- *logical-expr* `?` *expr1* `:` *expr2*
  - *expr1* is only evaluated if *logical-expr* is true
  - *expr2* is only evaluated if *logical-expr* is false
  - Again, is a life saver in preprocessor macros
  - But in normal use an `if ()` is more readable

Mixing Types in Expressions

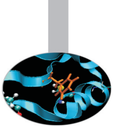


- C allows for expressions mixing any arithmetic types
  - A result will always be produced
  - Whether this is the result you expect, it's another story
- Broadly speaking, the base concept is clear
- For each binary operator in the expression, in order of precedence and associativity:
  - if both operands have the same type, fine
  - otherwise, operand with narrower range is converted to type of other operand
- OK when mixing floating types
  - The wider range includes the narrower one
- OK when mixing signed integer types
  - The wider range includes the narrower one
- OK even when mixing unsigned integer types
  - The wider range includes the narrower one

Type Conversion Traps

- For the assignment operator:
  - if both operands have the same type, fine
  - otherwise, right operand is converted to left operand type
  - if the value cannot be represented in the destination type, it's an overflow, and you are on your own
- We said: in order of precedence and associativity
  - if **`a`** is a type **`long long int`** variable, and **`b`** is a 32 bits wide **`int`** type variable and contains value **`INT_MAX`**, in:
    `a = b*2`
    multiplication will overflow
  - and in:
    `a = b*2 + 1LL`
    multiplication will overflow too
  - while:
    `a = b*2LL + 1`
    is OK

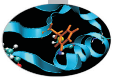
More Type Conversion Traps



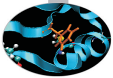
- Think of mixing floating and integer types
  - Floating types have wider range
  - But not necessarily more precision
  - A 32 bits **`float`** has fewer digits of precision than a 32 bits **`int`**
  - And a 64 bits **`double`** has fewer digits of precision than a 64 bits **`int`**
  - The result could be smaller than expected

- Think of mixing signed and unsigned integer types!
  - Negative values cannot be represented in unsigned types
  - Half of the values representable in an unsigned type, cannot be represented in a signed type of the same width
  - So, you are in for implementation defined surprises!
  - And Standard rules are quite complicated
  - We spare you the gory details, simply don't do it!

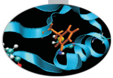
Cast Your Subexpressions



- **`(type)`**
  - Unsurprisingly, it's an operator
  - Precedence just higher than multiplication, right-to-left associative
  - Use it like **`(unsigned long)(sig + ned)`**
- Casting let you override standard conversion rules
  - In previous example, you could use it like this:
    `a = (long long int)b*2 + 1`
- Type casting is not magic
  - Just instructs compiler to apply the conversion you need
  - Only converts values, not type of variables you assign to
- Do not abuse it
  - Makes codes unreadable
  - Could be evidence of design mistakes
  - Or that your C needs a refresh

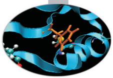


Rights & Credits



Slides and examples were authored by:

- Michela Botti
- Federico Massaioli
- Luca Ferraro
- Stefano Tagliaventi



Scientific and Technical Computing in C

Day 2

Luca Ferraro   Stefano Tagliaventi

CINECA - SCAI Department

Outline

struct


```
struct vect3D {
  double x, y, z;
};

struct vect3D va, vb;

// REMINDER: I have to make vcross() more efficient!
struct vect3d vcross(struct vect3D u, struct vect3D v) {
  struct vect3D c;

  c.x = u.y*v.z - u.z*v.y;
  c.y = u.z*v.x - u.x*v.z;
  c.z = u.x*v.y - u.y*v.x;

  return c;
}

//...
  vc = vcross(va, vb);
```

- Aggregates a single type from named, typed components (a.k.a. members)
- The **`vect3D`** *tag* must be unique among structure tags
- **`struct`** components can be independently accessed using the **`.`** binary operator

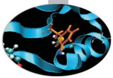
`struct`s Are Flexible

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

```
struct ion {
  struct vect3D r; // position
  struct vect3D v; // velocity
  enum element an; // atomic number
  int q;           // in units of elementary charges
};

struct ion a;
//...
  a.r.x += dt*a.v.x; // very low order in time...
```

- `struct` components can be inhomogeneous
- And they can also be `struct`s, of course
  - To access nested `struct` components, chain `.` expressions
- Best practice: order components by decreasing size
  - You'll get better performances
  - To know, you can use `sizeof()` operator on any type

structs: a Concrete Example



- **structs** are widely used in C Standard Library
- Like in **struct tm**, below, defined in **time.h**
  - Used to convert from/to internal time representation **time_t**

```
struct tm {
  int tm_sec; // seconds after the minute [0, 60]
  int tm_min; // minutes after the hour [0, 59]
  int tm_hour; // hours since midnight [0, 23]
  int tm_mday; // day of the month [1, 31]
  int tm_mon; // months since January [0, 11]
  int tm_year; // years since 1900
  int tm_wday; // days since Sunday [0, 6]
  int tm_yday; // days since January 1 [0, 365]
  int tm_isdst; // Daylight Saving Time flag
};
```

typedef

```c
typedef struct vect3D position, velocity;

typedef enum element element; // let's spare keystrokes

typedef int charge;           // I'll maybe switch to short or signed char

typedef struct ion {
  position r;
  velocity v;
  element an;
  charge q;
} ion;

ion a;
```

- **`typedef`** turns a normal declaration into a declaration of a new type (as usual, a legal identifier)
- The new type can be used as the native ones
  - Great to save keystrokes
  - Even better to write self-documenting code
  - Shines in hiding and factoring out implementation details
- **`struct`** tags and type identifiers belong to separate sets

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

`typedef` in C Standard Library

- **`typedef`** is widely used in C Standard Library
- Mostly to abstract details that may differ among implementations

- E.g. **`size_t`** from **`stddef.h`**
  - Type of value returned by **`sizeof()`**
  - Different platforms allow for different memory sizes
  - **`size_t`** must be “**`typedef`**ed” to an integer type able to represent the maximum possible variable size allowed by the implementation

- E.g. **`clock_t`** from **`time.h`**
  - Type of value returned by **`clock()`**
  - Cast it to **`double`**, divide by **`CLOCK_PER_SEC`**, ...
  - and you’ll know the CPU time in seconds used by your program from its beginning

Arrays

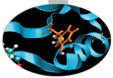
Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

- ***some_type*** **a[*n*];**
  - declares a collection of *n* variables of type ***some_type***
  - the variables (a.k.a. elements) are laid out contiguously in memory
  - each element can be read or written using the syntax **a[***integer indexing expression***]**
  - first element is **a[0]**, second one is **a[1]**, last one is **a[***n***-1]**
- You can't work on an array as a whole
  - Use array elements (if allowed...) in expressions and assignments
- There is no bound checking!
  - Use a negative index, or an index too big, and you are accessing something else, if any
  - Compiler options to (very slowly) check every access
- A common mistake:
  - to access from **double a[1]** to **double a[*n*]**
  - Fortran programmers beware!

Arrays of(Arrays of(Arrays of(...)))

- C has no concept of multidimensional arrays
- But array is a regular C type (you can even **`sizeof(double[150])`**)
- Thus, arrays of arrays can be declared
  - A simple, practical abstraction
  - Very annoying to Fortran or Matlab programmers
- **`int a[12][31];`**
  - declares an array of 12 elements
  - and each element is itself an array of 31 **`ints`**
- **`double b[130][260][260];`**
  - declares an array of 130 elements
  - and **`b[37]`** is itself an array of 260 elements
  - and **`b[37][201]`** is again an array of 260 **`doubles`**
- By the way, you can also use **`sizeof(b)`**, it works

Array Memory Layout



```
int a[10];
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

```
int b[5][2];
```

b[0]	b[1]	b[2]	b[3]	b[4]

b[0][0]	b[0][1]	b[1][0]	b[1][1]	b[2][0]	b[2][1]	b[3][0]	b[3][1]	b[4][0]	b[4][1]

A Very Important Digression

- Storage duration
  - To make it simple, the life time of a variable
  - Also influences the part of memory where it's allocated
- Scope
  - The region where a variable or function is accessible, a.k.a. "visible"
- Qualifiers
  - The value in a **`const`** variable cannot be changed
  - There are more, but we'll not discuss them
- Initializers
  - Values assigned to a variable at declaration

Storage Duration

- A variable can be
  - Automatic: it can be created when needed, and destroyed when not needed anymore
  - Static: it persists for the whole duration of the program
- Variables declared outside of any functions (i.e. at file scope) are static
- By default, are automatic:
  - all variables declared inside a compound statement
  - function parameters
- The default can be overridden using **`static`**
- Functions are static too, because to call them you need their code to persist in memory

Scopes

- By default, variables declared at file scope and functions are **`extern`**
  - i.e. visible to the linker, and to the whole program
  - Unless you declare them to be **`static`** only
- Variables declared at file scope and functions are visible to all blocks in the same source file
- Variables declared in a block are only visible in the block and in all scopes it encloses
  - Unless you declare them **`extern`**
  - But in most cases that's a symptom of bad design
- A variable declared in a block hides anything declared with the same name in enclosing scopes

Variable Initializers

- The content of an automatic variable is *uninitialized* until the variable is assigned a value
- *Uninitialized* is a polite form for ”unpredictable rubbish”
- **`double f = 2.5;`** is a practical shorthand for:

```
double f;
f = 2.5;
```

- Expressions can be used as initializers, as long as they can be computed at that point:

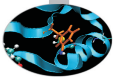
```
double pi = acos(-1.0);
double pihalf = pi/2.0;
```

is legal, while the following:

```
double pihalf = pi/2.0;
double pi = acos(-1.0);
```

obviously is not

More on Variable Initializers



- **`structs`** can be initialized too, as in:

```
struct vect3D V = {0.0, 1.0, 0.0};
```

- Same for arrays, as in:

```
float rot[2][2] = {{0.0, -1.0}, {1.0, 0.0}};
```

- **`{0.0, 1.0, 0.0}`** and **`{{0.0, -1.0}, {1.0, 0.0}}`** are said *compound literals*

- By default, static variables are initialized to 0
- But they can be initialized to different values
- Expressions can also be used, with some restrictions
  - For a static variable, initialization expression must be computed at compile time
  - I.e. it must be a *constant expression*, containing only constants
  - No variables, no function calls are permitted

Fast Fibonacci

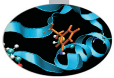

```
#include <limits.h>
#include <errno.h>
#include "fibonacci.h"

#define UINT_MAX_FIB_N 47

unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1];

void fibinit(void) {
  int i;
  FibonacciNumbers[0] = 0;
  FibonacciNumbers[1] = 1;

  for (i = 2; i <= UINT_MAX_FIB_N; ++i)
    FibonacciNumbers[i] = FibonacciNumbers[i-1] + FibonacciNumbers[i-2];

}

unsigned int fib(unsigned int n) {
    if (n > UINT_MAX_FIB_N) {
    errno = ERANGE;
    return UINT_MAX;
  }
  return FibonacciNumbers[n];
}
```

Arrays

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

- ***`some_type name`*** `[`*n*`]`
  - declares a collection of *n* variables of type ***`some_type`***
  - the variables are laid out contiguously in memory
  - each variable can be read or written using the syntax ***`name`*** `[`*index*`]`
  - where *index* is an integer expression ranging from 0 to *n*-1
- Variables declared at *file scope*
  - Variables declared outside of any function
  - Persist for the whole program life
  - By default, they can be accessed by any function...
  - ...except where the same name is used for a parameter or local variable
- *n* can also be an expression, as long as it can be evaluated at compile time

`for (;;)`, and Some `void` Too



- **`for`** **`(`***init-expr* **`;`** *logical-condition* **`;`** *incr-expr* **`)`**
  statement

  same as

  init-expr **`;`**
  `while` **`(`***logical-condition***`)`**
  `{`
  statement
  incr-expr **`;`**
  `}`
- But it's more compact and makes iteration bounds explicit in a single line
- What type is **`void`**?
  - As a return type, it tells a function returns nothing
  - As a parameter, it tells no arguments are accepted
- Why there is no **`return`** statement in **`fibinit()`**?
  - It returns nothing and completes at the closing brace

Hiding Implementation Details

- Array **`FibonacciNumbers`** is by default visible to the whole program
  - It could be accidentally modified or clash with another variable of the same name
  - Declaring it **`static`** will make it invisible to other modules
- **`fibinit()`** must be called in advance for **`fib()`** to return correct results
  - What if the call is omitted? Let's automate the process
  - Declaring it **`static`**, we make a function invisible to other modules
  - A variable declared in a function "disappears" when function returns, **`static`** will make it persist from call to call
- Best practices:
  - always hide irrelevant implementation details
  - if possible, automate initialization mechanisms

Fast Fibonacci: More Robust

Intro
Basics
1st Program
Choices
More T&C
Wrap Up 1
More C
1st Function
Testing
Compile and Link
Robustness
Wrap Up 2
Integers
Iteration
Test&Fixes
Overflow
Wider Ints
Polishing
Wrap Up 3
Arithmetic
Integers
Floating
Expressions
Mixing Types
Aggregate
Structures
Defining Types
Arrays
Storage & C.

```
#include <limits.h>
#include <stdbool.h>
#include <errno.h>
#include "fibonacci.h"

#define UINT_MAX_FIB_N 47

static unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1];

static void fibinit(void) {
  int i;
  FibonacciNumbers[0] = 0;
  FibonacciNumbers[1] = 1;

  for (i = 2; i <= UINT_MAX_FIB_N; ++i)
    FibonacciNumbers[i] = FibonacciNumbers[i-1] + FibonacciNumbers[i-2];

}

unsigned int fib(unsigned int n) {
  static bool doinit = true;

  if (doinit) {
    fibinit();
    doinit = false;
  }
  if (n > UINT_MAX_FIB_N) {
    errno = ERANGE;
    return UINT_MAX;
  }
  return FibonacciNumbers[n];
}
```

Arrays and Storage Classes

- Static arrays must be dimensioned with constant expressions
- Before C99, this was true for automatic arrays too
  - So to use an array in a function, you had to dimension it for the largest possible amount of work
  - A waste of memory and error prone
- C99 has a much better way
- Variable length arrays
  - Arrays whose size is unknown until run time
  - Automatic arrays can have their dimension specified by a nonconstant expression
  - Every time execution enters the block, the expression is evaluated
  - And the array size is determined, up to exit from the block

Arrays as Function Arguments

- Arrays can be huge
  - And usually are, in S&T computing
  - Passing them by value would be too costly
- Moreover, arrays cannot be used in assignments
  - Thus a function cannot return an array

- The solution
  - The address of the array is passed to a function
  - And elements can be accessed by it
  - (Later on, you'll understand how)

- This allows elements to be assigned to
  - Thus a function has a way to "return" an array result
  - A mixed blessing: allows changes to happen by mistake
- Best practice: declare an array parameter `const` if your only intent is reading its elements

Averaging, the C99 Way

- Let's write a function to average an array of **doubles**
- And make it generic in the array length
- Variable length array parameters come to the rescue

```
double avg(int n, const double a[n]) {
  int i;
  double sum = 0.0;

  for (i=0; i<n; ++i)
    sum += a[i];

  return sum/n;
}
```

Beware: **double avg(double a[n], int n)** does not work!

Averaging, the Old Way

- Before C99, there were no VLAs
- The solution was simple
  - Compiler just uses type size to find the right element
  - No bounds checking, no bound needed
- Many still write that way: it's equivalent, but less readable

```
double avg(int n, const double a[]) {
  int i;
  double sum = 0.0;

  for (i=0; i<n; ++i)
    sum += a[i];

  return sum/n;
}
```

Calling **avg()**

- New or old style, simply pass array dimension and name
- If **avg()** is written using VLAs, pedantic compilers may give a warning on function call, even if it's correct: they are wrong, check with Standard document or good book

```
double mydata[N];
double mydata_avg;

// read or compute N doubles into mydata[]

mydata_avg = avg(N, mydata);
```

Averaging Arrays of Arbitrary Length

- Let's generalize the average to set of *m* numbers
- And make it generic, as usual
- Again, VLA parameters come to the rescue

```
void avg(int n, int m, const double a[n][m], double b[m]) {

  int i, j;

  for (j=0; j<m; ++j)
    b[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<m; ++j)
      b[j] += a[i][j];

  for (j=0; j<m; ++j)
    b[j] /= n;
}
```

Notice: this order of loops nesting gives faster execution

Calling Generic **avg()**

- Again, simply pass array dimension and name
- Using casts for arrays of doubles
- If **avg()** is written using VLAs, pedantic compilers may give a warning on function call, even if it's correct: they are wrong, check with Standard document or good book

```
double mydata1[N][12];
double mydata1_avg[12];
double mydata2[N][7];
double mydata2_avg[7];
double mydata3[N][1];
double mydata3_avg[1];
double mydata4[N];
double mydata4_avg[1];

// read or compute N 12-uples of doubles into mydata1[]
// read or compute N 7-uples of doubles into mydata2[]
// read or compute N 1-uples of doubles into mydata3[]
// read or compute N  doubles into mydata4[]

avg(N, 12, mydata1, mydata1_avg);
avg(N, 7, mydata2, mydata2_avg);
avg(N, 1, mydata3, mydata3_avg);
avg(N, 1, (double [N][1])mydata4, mydata4_avg);
```

Matrix Algebra, the C99 Way

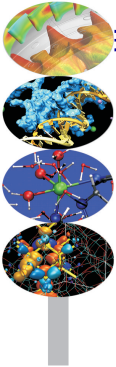
- Let's write a function to compute the trace of a matrix of **`double`**s
- And make it generic in the matrix size
- Again, variable length array parameters come to the rescue
- Again, you may get warnings on calls, and they could prove wrong

```
double tr(int n, const double a[n][n]) {
  int i;
  double sum = 0.0;

  for (i=0; i<n; ++i)
    sum += a[i][i];

  return sum;
}
```

Beware: compiler will not check the array dimensions match!

Matrix Algebra, the Old Way

- Before C99, there were no VLAs
- The solution was not that simple...
  - Only the ‘first dimension’ of an array parameter could be left unspecified at compile time
- To understand the solution, you have to learn more



entific and Technical Computing in C

Day 2

Luca Ferraro   Stefano Tagliaventi

CINECA - SCAI Department

Outline

You May Need More

- You may find yourself in need to return more than one result from a function
- And you may find yourself in need to pass a big `struct` to a function, without paying the price of copying its value
- And, believe it or not, in some part of your program you may find yourself in need to access a variable whose name is not known
- And to represent things as multiblock, unstructured grids, or building structures, or complex molecules, you may find yourself in need to access variables that don’t even have a name

- In all these cases, you have to use memory addresses

Memory? Addresses?

- You can think of memory as a huge array of units of storage (usually 8 bits bytes)
  - The index in this array is termed *address*
- But how many bytes are needed to store a value?
  - It depends on value type and platform
- And it's even worse...
  - Not all locations are good for any value (at least performancewise)
  - Not all locations can be read/written
  - What are the starting and ending address?
  - The amount of memory seen by your program could vary during execution
  - You could have 'holes' in this ideal array
  - Or this ideal array could be made of separate, independent segments

Enter C Pointers

- Dealing directly with memory addresses is cumbersome
  - Easily makes the program non portable
  - Makes the program difficult to manage and confusing
  - Exhibits low level details you don't really want to care about
- How to avoid it?
- Named variables leave the whole issue to the compiler
  - You use the name and don't care about address
- C pointers let you manipulate addresses in a transparent and consistent way
  - They contain memory addresses
  - Allow you to manipulate addresses disregarding their actual values
  - Associate a C type to the memory location they point to
  - And give you a way to read or write this memory location, much like a named variable

Pointers Basics

- **`int i, *p;`**
  - declares an **`int`** variable **`i`**
  - and a ‘pointer to **`int`**’ variable **`p`**
  - in the latter, you can store the address of a memory location suitable to store an **`int`** type value
- **`p = &i;`**
  - **`&i`** evaluates to the address of variable **`i`**
  - **`p`** gets a valid address in
  - Got something familiar? Do you remember **`scanf()`**?
- **`*p = 10;`**
  - Expression **`*p`** is an *lvalue* of type **`int`**
  - You can performe assignment to it
  - You can use it in expressions to access the stored value
  - **`*`** has same precedence and associativity of unary **–**

Pointer vs. Pointee

Code	p:	a:
`int *p = NULL;` `int a = 5;`	0	5
`p = &a;`	address of **a**	5
`*p += 10;`	address of **a**	15
`a += 1;`	address of **a**	16

Avoiding Costly Copies


```
struct vect3D {
  double x, y, z;
};

// REMINDER: I have to make vcross() more efficient! DONE!!
struct vect3d vcross(const struct vect3D *u, const struct vect3D *v) {
  struct vect3D c;

  c.x = u->y*v->z - u->z*v->y;
  c.y = u->z*v->x - u->x*v->z;
  c.z = u->x*v->y - u->y*v->x;

  return c;
}
```

- Copying 6 **`doubles`** for very little work
- Let's put pointers to good use
- **`u->y`** is a convenient shorthand for **`(*u).y`**
- But now we have the address of the arguments and could make a mistake and change their contents
- Let's make the pointees **`const`**

Did we say “valid”?

- A valid pointer value is an address that:
  - is in the process memory space
  - points to something which exists
  - and whose type matches
- Invalid pointers
  - uninitialized pointers (point to the wrong place, at best)
  - the address of a variable that does not exist anymore
  - the address of one type put in pointer to another type (unless you REALLY know what you are doing)
  - a null pointer, i.e. a 0 address
- Dereferencing (with `*`) a null pointer forces runtime error
- Good practice:
  - Always initialize pointers
  - If you don’t know yet the right address, use **`NULL`** from **`stddef.h`**
  - **`0`** may also be used, but less readable

A Naive Mistake


```
struct vect3D {
  double x, y, z;
};

// REMINDER: I have to make vcross() more efficient! DONE!! Trying to do better...
struct vect3d *vcross(const struct vect3D *u, const struct vect3D *v) {
  struct vect3D c;

  c.x = u->y*v->z - u->z*v->y;
  c.y = u->z*v->x - u->x*v->z;
  c.z = u->x*v->y - u->y*v->x;

  return &c; // MADNESS!!
}
```

- Sparing another copy it's tempting...
- But it's very naive!
- `c` is an automatic variable, and it's gone when the pointer is used
- And probably the memory locations have been already reused and overwritten!

Pointers and Arrays

- `double *p[10]`
  - it's an array of 10 pointers to `double`
- and `double *p[10][3]`
  - it's an array of 10 arrays, each of 3 pointers to `double`
- while `double (*p)[10]`
  - it's a pointer to array of 10 `doubles`
- and `double (*p)[10][3]`
  - it's a pointer to an array of 10 arrays, each of 3 `doubles`
- Confusing? It's logical: operator `[]` has higher precedence than `*`
- But easily becomes nasty!
  - What's `double (*p[10])[3]`?
  - And `double (*(*p[10])[3][5])[8][2]`?
- Best practice: use `cdecl` tool to familiarize and decrypt

Pointers Arithmetic

- Useful to poke around in arrays
- `p + 7`
  - will give you an address
  - that is `7*sizeof(*p)` after the one in `p`
- You can also use `-`, `+=`, `-=`, `++`, and `--`
- `p1 - p2`
  - if of the same pointer type, will give you an integer value
  - more precisely, of `ptrdiff_t` type (from `stddef.h`)
  - the displacement from `p2` to `p1` in units of `sizeof(*p1)`
- Pointer comparison
  - `==` (equal), `!=`, `>`, `<`, `>=`, `<=` can be used on pointers of the same type
- Pointer casting
  - Pointer values can be cast to pointers of different type
  - Do it VERY carefully, it's easy to do the wrong thing
  - Pointers may also be cast to some integer type, but it's highly non portable, don't do it

Pointers and Array Equivalence

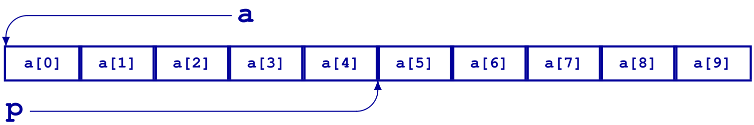
- `*(p+7)` can be shortened to `p[7]`
- Aha!
- Can a pointer be used as an array?
  - `true`
- I see... so is the array name a pointer?
  - `true`, but it's constant, you can't change it
- But if I have `int a[N]`, and `int *p`, may I assign `p=a`?
  - `true`, you can
- Then, what's the difference between an array variable and a pointer variable declarations?
  - An array declaration allocates memory for data
  - A pointer declaration allocates memory for a data address only
- And between array and pointer function parameters?
  - Irrelevant, an array argument passes a pointer
  - You are now ready to understand good old C tricks

Skeptical? Try to Believe

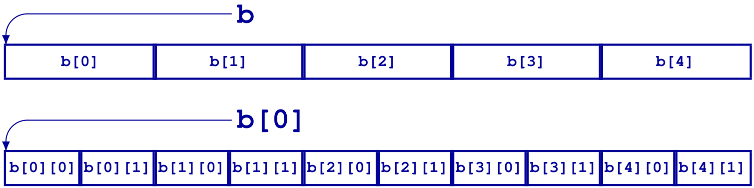

```
#include <stdio.h>

double a[] = {1.0, 2.0, 3.0, 4.0, 5.0};

int main() {

  double *p;

  p = a; // variable p now stores the address of array a

  printf("%lf\n", a[2]); // will print 3.0
  printf("%lf\n", *(p+2)); // will print 3.0

  p[2] = 7.0; // reassigns a[2]

  printf("%lf\n", p[2]); // will print 7.0
  printf("%lf\n", a[2]); // ditto, it's the same location

  return 0;
}
```

Array Names and Pointers


```
int a[10];
int *p = a + 5;
```




```
int b[5][2];
```



Averaging, with Pointers

- This one should be quite obvious
- Perfectly equivalent to using **`const double a[]`**
- You'll often encounter something like this, particularly in libraries

```
double avg(int n, const double *a) { /* which one is const? */
  int i;
  double sum = 0.0;

  for (i=0; i<n; ++i)
    sum += a[i];

  return sum/n;
}
```

`const int *p` is a pointer to **`const`**, **`int *(const p)`** is a **`const`** pointer

Averaging Arrays, with Pointers

- Let's generalize to sets of *m* numbers
- And make it generic, as usual
- Now you are ready for the traditional solution
- And for an application of pointer casting

```
void avg(int n, int m, const double (*a)[], double *b) {
  int i, j;
  const double *p = (const double *)a;

  for (j=0; j<m; ++j)
    b[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<m; ++j)
      b[j] += p[i*m + j];     /* mapping two indexes */
                              /* to one `by hand' */
  for (j=0; j<m; ++j)
    b[j] /= n;
}
```

Calling Generic **avg()**

- New or old style, arrays or pointers, simply pass array dimension and name
- Using casts for arrays of doubles
- If **avg()** is written using VLAs, pedantic compilers may give a warning on function call, even if it's correct: they are wrong, check with Standard document or good book

```
double mydata1[N][12];
double mydata1_avg[12];
double mydata2[N][7];
double mydata2_avg[7];
double mydata3[N][1];
double mydata3_avg[1];
double mydata4[N];
double mydata4_avg;

// read or compute N 12-uples of doubles into mydata1[]
// read or compute N 7-uples of doubles into mydata2[]
// read or compute N 1-uples of doubles into mydata3[]
// read or compute N  doubles into mydata4[]

avg(N, 12, mydata1, mydata1_avg);
avg(N, 7, mydata2, mydata2_avg);
avg(N, 1, mydata3, mydata3_avg);
avg(N, 1, (double [N][1])mydata4, &mydata4_avg);
```

Averaging Arrays, Another Classic Flavor

- Again averages sets of *m* numbers
- For arbitrary *m*
- This idiom arose when compilers were not good at optimization

```
void avg(int n, int m, const double (*a)[], double *b) {
  int i, j;
  const double *p = (const double *)a;

  for (j=0; j<m; ++j)
    b[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<m; ++j) {
      b[j] += *p;    /* array elements ‘walked by’ */
      ++p;           /* in the same sequence */
    }

  for (j=0; j<m; ++j)
    b[j] /= n;
}
```

Matrix Algebra, the Old Way

- Let's write a function to compute the trace of a matrix of **`double`**s
- And make it generic in the matrix size
- And use a traditional way
- Again, you'll often encounter something like this, particularly in libraries

```
double tr(int n, const double (*a)[]) {
  int i;
  double sum = 0.0;
  const double *p = *a;  /* works like casting here, why? */

  for (i=0; i<n; ++i)
    sum += p[i*n + i];

  return sum;
}
```

Matrix Algebra, Another Old Way

- Let's write a function to compute the trace of a matrix of **`double`**s
- And make it generic in the matrix size
- And use another traditional way, from times when compilers didn't optimize well

```
double tr(int n, const double (*a)[]) {
  int i;
  double sum = 0.0;
  const double *p = *a;

  for (i=0; i<n; ++i) {
    sum += *p;
    p += n + 1;   /* next element on diagonal */
  }

  return sum;
}
```

Matrix Algebra, yet Another Classic Flavor

- Bottom line, we are working on **`double`**s
- Call it like **`tr(8, (double *)mp)`**
- Or call it like **`tr(8, mp[0])`**
- Widely used in numerical libraries, but write new code using VLAs

```
double tr(int n, const double *a) {
  int i;
  double sum = 0.0;

  for (i=0; i<n; ++i) {
    sum += *a;
    a += n + 1; /* next element on diagonal */
  }

  return sum;
}
```

Matrix Algebra, a Bad Way

- A way of getting rid of all complexity
- It's the "third" use of type **`void`**
- Sometimes you'll find sloppy code like this
- But not a good idea in this case, it's dangerous

```
double tr(int n, const void *a) {
  int i;
  double sum = 0.0;
  double *p = a;

  for (i=0; i<n; ++i) {
    sum += *p;
    p += n + 1; /* next element on diagonal */
  }

  return sum;
}
```

`void` and Pointers

- **`void *p;`** declares a *generic pointer*
- I.e. a pointer pointing to unknown type
- If type is unknown, size is unknown
- So no arithmetic is possible, only assignment and comparisons
- The value of any pointer can be converted to a generic one
- A generic pointer can be converted to any pointer type

- So, what's the danger with **`tr()`**?
  - **`tr()`** assumes something pointing to **`doubles`**
  - With **`void *`**, pointers at any type will do
  - A pedantic compiler would warn you at any use of **`tr()`**
  - And you'd get annoyed and switch off warnings
- But generic pointers are essential to other purposes

qsort()

- Declaration (from **`stdlib.h`**):

```
void qsort(
  void *base,
  size_t count,
  size_t size,
  int (*compare)(const void *el1, const void *el2) );
```

- Sorts an array of **`count`** elements of unknown type, starting at **`base`**
- Each element has size **`size`**
- What's **`compare`**?
  - **`qsort()`** doesn't know elements type
  - And has no clue at how to compare them
  - **`compare`** is a pointer to a function that knows more
- Yes, a function has an address and function name evaluates to it

Sorting with `qsort()`

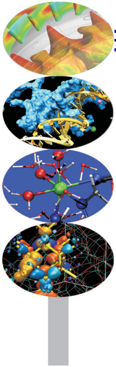
- Define a comparison function like:

```
int comparedoubles(const double *a, const double *b) {
  if (*a == *b)
    return 0;

  if (*a > *b)
    return 1;

  return -1;
}
```

- Can you see how it matches the **compare** parameter?
- Then, if **g** is an array of 10000 **doubles**, you can sort it in ascending order like this:

```
qsort(g, 10000, sizeof(double), comparedoubles);
```

- Want it sorted in descending order?
  - Substitute < to >
- Have an array sorted in ascending order?
  - You can use **bsearch()** to find an element



entific and Technical Computing in C

Day 2

Luca Ferraro   Stefano Tagliaventi

CINECA - SCAI Department

Outline

Characters

- In C, characters have type `char`
- I.e. an integer type holding the numeric character code
- But it's implementation defined if `char` is signed or not
- Encoding may depend on implementation and OS
- In most implementations, characters numbered 0 to 127 match the standard ASCII character set
- Literal character constants are specified like this: `'C'`
  - `'\n'` is new line
  - `'\t'` is tab
  - `'\r'` is carriage return
  - `'\\'` is backslash `\`
  - `'\''` is `'`
  - `'\"'` is `"`
  - and `'\0'` is ASCII NUL, with code 0, quite important despite of its value

#include <ctype.h>

Function	Returns
`int isalpha(int c)`	true if alphabetic character
`int isdigit(int c)`	true if a digit character
`int isalnum(int c)`	`isalpha(c) || isdigit(c)`
`int isprint(int c)`	true if printable character (including `' '`)
`int iscntrl(int c)`	`!isprint(c)`
`int islower(int c)`	true if lowercase alphabetic character
`int isupper(int c)`	true if uppercase alphabetic character
`int isspace(int c)`	true if `' '`, `'\t'`, `'\n'`, ...
`int tolower(int c)`	converts uppercase ones to lowercase others unchanged
`int toupper(int c)`	converts lowercase ones to uppercase others unchanged

- Do you remember? `char` types are converted to `int` in all arithmetic expressions
- Do not play with character codes, use these functions, they make the code portable

Strings

- Strings are not first-class citizens in C
- Simply arrays of **`chars`**
- The string must be terminated by a `'\0'` character
- Commonly referred to as *null terminated* strings
- This has annoying consequences
  - String lengths must be computed by scanning
  - No way for bounds checking
  - And a source of program weaknesses

- String constants are specified like this:
  `"A null terminated string"`
- A terminating `'\0'` is automatically appended
- You already met them using **`printf()`**
- Use a `\` at end of lines to write multiline string constants

The Biggest Mistake


```
char decdigits[10];
//...
strcpy(decdigits, "0123456789");
```

- The string is 10 characters long
- But it has a terminating `'\0'`
- So its internal representation is **11** characters long

Fixing the Biggest Mistake


```
char decdigits[] = "0123456789";
```

- An 11 characters array will be automatically allocated
- (Yes, you could do this for any array)
- But this only fixes the problem on initialization
- Not when you build string dynamically or do simple minded I/O
- Ever heard of '*buffer overflows*'?

#include <string.h>

Function	Does
`size_t strlen(const char *s)`	returns actual string length

`char *strncpy(char *d,`
`const char *s,`
`size_t n)`
	copies **n** characters from **s** to **d**, returns **d**

`char *strncat(char *d,`
`const char *s,`
`size_t n)`
	appends **n** characters from **s** to **d**, returns **d**

`int strcmp(const char *s1,` `const char *s2)`	lexicographic comparison of **s1** and **s2**

`int strncmp(const char *s1,`
`const char *s2,`
`size_t n)`
	lexicographic comparison of **s1** and **s2**, up to **n** characters

`char *strchr(const char *s,` `int c)`	returns pointer to first occurrence in **s** of character **c**, **NULL** if not found
`char *strrchr(const char *s,)` `int c)`	returns pointer to last occurrence in **s** of character **c**, **NULL** if not found
`char *strcspn(const char *s,` `const char *set)`	returns pointer to first occurrence in **s** of any character in **set**, **NULL** if not found
`char *strspn(const char *s,` `const char *set)`	returns pointer to first occurrence in **s** of any character not in **set**, **NULL** if not found
`char *strstr(const char *s,` `const char *sub)`	returns pointer to first occurrence in **s** of string **sub**, **NULL** if not found
`char *strtok(const char *s,` `const char *set)`	allow to separate string **s** into tokens, read documentation

- Do you remember? **`char`** types are converted to **`int`** in many cases
- You'll also find in use **`strcpy()`** and **`strcat()`**: dangerous! avoid them
- Way too common mistake: forgetting about and writing code doing the same
- Don't reinvent the wheel, use library functions!

More Friends from `stdlib.h`

Function	Returns conversion of initial portion of `s` to
`strtof(const char *s, char **p)`[3]	`float`[1]
`strtod(const char *s, char **p)`	`double`[1]
`atof(const char *s)`	`double`
`strtold(const char *s, char **p)`[3]	`long double`[1]
`atoi(const char *s)`	`int`
`strtol(const char *s, char **p, int base`[2]`)`	`long`[1]
`atol(const char *s)`	`long`
`strtoul(const char *s, char **p, int base`[2]`)`	`unsigned long`[1]
`strtoll(const char *s, char **p, int base`[2]`)`[3]	`long long`[1]
`atoll(const char *s)`[3]	`long long`
`strtoull(const char *s, char **p, int base`[2]`)`[3]	`unsigned long long`[1]

1. If `p` is not null, sets it to point to first character after converted portion of `s`
2. The `base` used in string representation ranges from 2 to 36 (!).
3. C99

- More practical than `scanf()` family in many cases
- `strto...()` form preferred
- Use `sprintf()` to convert the other way around
- Where `char **p` appears, pass the address of a `char *` pointer variable...

Yes, Pointers can be Pointees!


```
int **p = NULL;
int *q = NULL;
int a = 5;
```

p:	0
q:	0
a:	5

```
p = &q;
```

p:	address of q
q:	0
a:	5

```
*p = &a;
```

p:	address of q
q:	address of a
a:	5

```
**p += 10;
```

p:	address of q
q:	address of a
a:	15

`argc` and `argv`

- Up to now, we disregarded **`main()`** parameters
  - Which is legal
  - And writing **`int main(void)`** is legal too
- In its full glory, **`main(int argc, char *argv[])`** receives two arguments
  - An integer count, **`argc`**
  - And an array of **`argc`** pointers to string, **`argv`**
  - Names are not mandatory, just a solid tradition
- On most systems
  - **`argv[0]`** contains the name of program executable
  - **`argv[1]`** through **`argv[argc-1]`** contain the command line parameters specified at program invocation
- Form **`int main(int argc, char **argv)`** is fully equivalent

Use of `argc` and `argv`


```
void print_help_and_exit(){
        printf("Usage: ./shapp [-l|-t|-h]\n");
        exit(EXIT_FAILURE);
}
int main(int argc,char *argv[]){

        if(argc < 2 || argv[1][0]!='-')
                print_help_and_exit();
        switch(argv[1][1])
        {
                case 't':
                        timestamp_ordering();
                        break;
                case 'r':
                        reverse_order();
                        break;
                case 'h':
                        print_help_and_exit();
                default:
                        print_help_and_exit();
        }

}
```

More Alternatives with **`switch ()`**

- **`switch (`**_integer-expression_**`) {`**
  `case` _constant-expression_**`:`**
  statements
  [**`case`** _constant-expression_**`:`**
  statements]
  [**`default:`**
  statements]
  `}`

  1. Evaluates _integer-expression_
  2. If value equals one _constant-expression_, execution jumps to the statement following it
  3. Otherwise, if **`default:`** exists, execution jumps to statement following it
  4. Otherwise execution leaves **`switch()`** and proceeds to the following code

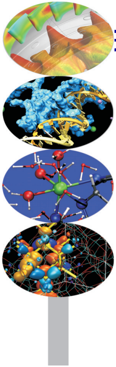
A `switch ()` ’Feature’

- Beware: once 2 or 3 above happened, encounter of another **`case`** or of **`default`** does not imply exit from **`switch`**!
- A **`break;`** statement is needed to this purpose

- This is way too easily forgotten
- Best practices:
  - Always add a **`break;`** statement at end of each ’**`case`**’
  - Even if it’s unreachable, you’ll appreciate on code changes
  - Unless you really intend to execute two or more ’**`cases`**’ at once

More **`break`**, and **`continue`**

- A **`break;`** statement forces execution to bail out from innermost enclosing statement among:
  - **`switch ()`**
  - **`while ()`**
  - **`do...while ()`**
  - **`for (;;)`**
- A **`continue;`** statement terminates execution of current iteration of innermost enclosing statement among:
  - **`while ()`**
  - **`do...while ()`**
  - **`for (;;)`**
- Execution continues with next iteration



entific and Technical Computing in C

Day 2

Luca Ferraro   Stefano Tagliaventi

CINECA - SCAI Department

Outline

Files

- C thinks of files as *streams* of data you can read/write from/to
- C has no notion of file content or structure: user knows about
  - You read what you know is there
  - You write what you want to put there
- Files are managed by internal data structures of **`FILE`** type
  - Whose details may be implementation defined
- All functions are declared in **`stdio.h`**
- Most functions return or accept pointers to **`FILE`** structures
- You simply declare variables of **`FILE *`** type and use these functions
  - And usually may disregard details

Three Files for Free

- When **`main()`** is called, three files have already been opened for you
- Accessible by three expressions of **`FILE *`** type
  - **`stdin`** for standard input
  - **`stdout`** for standard output
  - **`stderr`** for error messages output
- Usually map to user's terminal, unless they were redirected at command launch

Using More Files is not Free

- If **`myfile`** is a **`FILE *`** variable, open a file using:
  `myfile = fopen("mydata.dat", "r");`
- Second string is a mode:
  - **`"r"`** to read existing text file
  - **`"w"`** to create a new text file or truncate existing one to zero length
  - **`"a"`** to create a new text file or append to existing one
  - Use **`"rb"`**, **`"wb"`**, or **`"ab"`** for binary files
  - **`"r+"`** and **`"r+b"`** to both read and write to existing file
- Biggest mistake: assuming **`fopen()`** succeeded
  - **`fopen()`** returns NULL on failure
  - Always check and use **`errno`** to know more
- **`fclose(FILE *f)`** orderly closes an open file, do it when you are done with it
- A string **`FILENAME_MAX`** long is big enough for any file name

Simple String I/O

- **`char *fgets(char *s, int n, FILE *stream)`**
  - Reads in at most one less than **`n`** characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline.
  - Returns **`s`** on success, **`NULL`** on failure
  - A robust I/O function. Use it in your code.
- Use **`int feof(FILE *stream)`** to check if **`NULL`** was returned because end of file was reached
- **`char *fputs(const char *s, FILE *stream)`**
  - Writes **`s`** string to file
  - Returns **`EOF`** on error
- **`char *puts(const char *s)`**
  - Like **`fputs()`** on **`stdout`**, but adds a **`'\n'`**
- You'll encounter **`gets()`** in codes: offers no control on maximum input size, don't use it

Talking to Humans

- **`fprintf()`** converts internal formats of basic data types to human readable formats
- **`fprintf(`*`file, "control string", arguments`*`)`**
  - Characters in ***`control string`*** are emitted verbatim
  - But conversion specifications beginning with `%` cause the conversions and output of arguments
  - Arguments (i.e. expressions) must match conversion specifications in number, types, and positions
  - Conversion specification `%%` emits a `%` character and consumes no arguments
- **`printf()`** outputs to **`stdout`**
- **`snprintf()`** and **`sprintf()`**
  - Write to string instead of file
  - **`snprintf()`** is preferable as maximum string length can be specified

Common Mistakes

- Beware: if you want to remove item **`c`** from output in

  ```
  printf("Parameters: %lf, %lf, %lf\n", a, b, c);
  ```

  the following is not enough:

  ```
  printf("Parameters: %lf, %lf, %lf\n", a, b);
  ```

  you need to update the format string too:

  ```
  printf("Parameters: %lf, %lf\n", a, b);
  ```

- And on adding an item you have to add a proper conversion specifier
- Ditto for type mismatches: no argument checking is required
- In some cases, dire consequences could follow
- A clever compiler may be able to warn you, if you ask

`printf()`: Integer Types

- In **`%d`** and **`%u`**, **`d`** and **`u`** are conversions
  - Internal to base 10 text representation
- **`l`**, **`ll`**, **`h`**, and **`hh`**, are size modifiers
  - Look back at integer types table if you need a refresh
- Variations on a theme
  - **`%10d`**: at least 10 characters, right justified, space padded
  - **`%.4d`**: at least 4 digits, right justified
  - **`%010d`**: at least 10 characters, right justified, leading `0`s
  - **`%-10d`**: at least 10 characters, left justified, space padded
  - **`%+d`**: sign is always printed (not relevant for **`u`**)
  - **`% d`**: same, but a space if positive (not relevant for **`u`**)
- **`printf("%-5d%+6.4d", 12, 12);`**
  Prints?

`printf()`: Floating Types

- Conversions
  - `%f`: `float` to base 10 decimal text
  - `%E`: `float` to base 10 exponential text
  - `%G`: most suitable of the above ones
- `l` and `L` are size modifiers
  - Look back at floating types table if you need a refresh
- Variations on a theme
  - `%10f`: at least 10 characters, right justified, space padded
  - `%.4f`: 4 digits after decimal point (`f` and `E` only)
  - `%.7G`: 7 significant digits
  - `%010f`: at least 10 characters, right justified, leading `0`s
  - `%-10f`: at least 10 characters, left justified, space padded
  - `%+f`: sign is always printed
  - `% f`: same, but a space if positive
- `printf("%+8.2lf %.4lE", 12.0, 12.0);`
  Prints?

`printf()`: Characters and Strings

- **`%c`**: emits character with specified code
- No variations

- **`%s`**: emits a string
- Variations on a theme
  - **`%10s`**: at least 10 characters, right justified, space padded
  - **`%.7s`**: exactly(!) 7 characters from string
  - **`%-10s`**: at least 10 characters, left justified, space padded
- **`printf("%-7s%4.3s", "Vigna", "Vigna");`** Prints?

- And more conversions are defined, but we'll not cover them

Listening to Humans

- **`fscanf()`** converts human writable formats of basic data types to internal ones
- **`fscanf(`*`file`*`, "`*`control string`*`", `*`arguments`*`)`**
  - Arguments must be pointers!
  - Arguments must match conversion specifications in number, types, and positions
  - White-space in ***`control string`*** matches an arbitrary sequence of zero or more spaces
  - All other characters must match verbatim with characters in input
- **`scanf()`** reads from **`stdin`**
- **`sscanf()`** reads from string instead of file

`scanf()` Conversions

- Conversions discussed for **`printf()`** work, the other way around
- They skip white-space characters before reading and converting, except for **`%c`**
- Number too big for the type? Result is implementation defined
- Fewer variations on the theme (for most conversions)
  - **`%10d`**: no more than 10 characters considered (not for **`%c`**)
  - **`%*d`**: looks for text matching an **`int`**, but ignores it
- **`scanf("%4d%*6d%3d", &i1, &i2);`**
  Input: **`12   34567890`** (notice: 3 space characters)
  Reads?

Common Mistakes

- Any mismatch in input to a **`scanf()`** will stop input and conversions
- **`scanf()`** always returns the number of conversions performed, do not discard it:

  ```
  itemsread = scanf("%lf ,%lf", &a, &b);
  ```

  check the result, and take correcting actions (or fail gracefully)
- Giving fewer arguments than conversion specifiers, as in:

  ```
  itemsread = scanf("%lf ,%lf ,%lf", &a, &b);
  ```

  is a very good recipe for disaster, and one difficult to debug
- So is giving the wrong pointer or a pointer to the wrong type

User Input


```
//...
    printf("Enter t max: ");
    scanf("%lf", &tmax);
```

- User mistypes `U.0` for `7.0`
- Program behaves in unintended ways
- Could check `scanf()` return value and fail gracefully, but let's give user a chance

Wrong Solution


```
  int itemsread;
//...
  do {

    printf("Enter t max: ");

    itemsread = scanf("%lf", &tmax);

  } while (itemsread == 0);
```

- Again, user mistypes `U.0` for `7.0`
- Program stops responding, burning CPU cycles
- `scanf()` is very finicky about input
  - As soon as a character doesn't match the format string, puts it back in input buffer
  - To find it again at each iteration

Better Solution


```
  int itemsread;
//...
  do {
    char s[257];

    printf("Enter t max: ");
    if (fgets(s, sizeof(s), stdin) == NULL)
      exit(EXIT_FAILURE);

    itemsread = sscanf(s, "%lf", &tmax);

  } while (itemsread == 0);
```

- This form causes wrong input to be consumed and removed
- Use **`fscanf()`** for rigidly formatted files
- With imprecise formats (as user input is), use **`fgets()`**, then **`sscanf()`**

Dealing with Many Data

- Text I/O is human readable
- Text I/O is platform independent
- But text I/O is huge
  - Because of issues in base 2 vs. base 10 representation
- To recover exact binary form of a floating type, you need:
  - at least 9 decimal digits in text I/O for a **`float`**
  - at least 19 decimal digits in text I/O for a **`double`**
- And text I/O is slow
  - Because of size
  - And because conversions take time
- Best practice:
  - Use text I/O to talk to humans or as a last resort for some programs
  - Use binary I/O otherwise

Binary Reads and Writes


```
size_t fread(void *data, size_t elsz,
                  size_t count, FILE *f);
size_t fwrite(const void *data, size_t elsz,
                  size_t count, FILE *f);
```

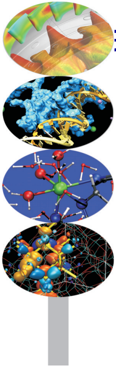
- Read/write ***count*** elements of size ***elsz*** from/to file ***f*** to/from address ***data***
- Both return the number of elements actually read/written
  - Can be less than requested if error occurred, or (**fread()** only) end of file was encountered
  - Use **feof()** or **ferror()** to determine cause
- Best practice:
  - do binary I/O in chunks as large as possible
  - performance will sky-rocket

Walking Around in a File

- Each I/O operation takes place from the position in the file where the last one ended
- But position can be changed
- Not special to binary files, but mostly used with them
- **`fseek(f, 4096L, wherefrom)`** moves forward by 4096 bytes relative to:
  - file beginning, if ***`wherefrom`*** is **`SEEK_SET`**
  - current position, if ***`wherefrom`*** is **`SEEK_CUR`**
  - file end, if ***`wherefrom`*** is **`SEEK_END`**
  - and returns zero if successful, non zero otherwise
- **`ftell(f)`** returns the current position (**`long`**)
  - on failure, returns -1L and sets **`errno`**
- This is a 64 bits world: files can be huge!
  - In case, use **`fsetpos()`** and **`fgetpos()`**
  - They use an **`fpos_t`** type large enough

Dealing with Fortran Binary Files

- You may need to read Fortran binary files
- And Fortran adds two extra 32 or 64 bits integers, one at beginning and one at end of each record (i.e. of each **`WRITE`** for unformatted files)
- Option 1: skip them with **`fseek()`**
- Option 2: read them and forget the values
- Option 3: write the file from Fortran opening it in **`STREAM`** mode
    - Designed to match the C file concept
    - Introduced in Fortran 2003
    - But already available in most implementations



entific and Technical Computing in C

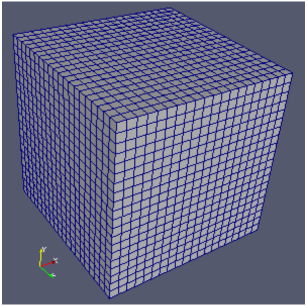
Day 2

Luca Ferraro Stefano Tagliaventi

CINECA - SCAI Department

Outline

A PDE Problem



- Let's imagine we have to solve a PDE
- On a dense, Cartesian, uniform grid
  - Mesh axes are parallel to coordinate ones
  - Steps along each direction have the same size
  - And we have some discretization schemes in time and space to solve for variables at each point

A Rigid Solution


```
#define NX 200
#define NY 450
#define NZ 320

double deltax; // Grid steps
double deltay;
double deltaz;
//...
double u[NX][NY][NZ]; // x velocity component
double v[NX][NY][NZ]; // y velocity component
double w[NX][NY][NZ]; // z velocity component
double p[NX][NY][NZ]; // pressure
```

- We could write something like that at file scope
- But it has annoying consequences
  - Recompile each time grid resolution changes
  - A slow process, for big programs
  - And error prone, as we may forget about
- Couldn't we size data structures according to user input?

Looking for Flexibility


```
int main(int argc, char *argv[]) {
  double deltax, deltay, deltaz; // Grid steps
  int nx, ny, nz
//...
  double u[nx][ny][nz];
  double v[nx][ny][nz];
  double w[nx][ny][nz];
  double p[nx][ny][nz];
```

- We could think of declaring variable length arrays inside **`main()`** or other functions
- This is unwise
  - Automatic arrays are usually allocated on the process stack
  - Which is a precious resource
  - And limited in most system configurations

A Better Approach


```
#define MAX_NX 400
#define MAX_NY 400
#define MAX_NZ 400

double u[MAX_NX*MAX_NY*MAX_NZ];
double v[MAX_NX*MAX_NY*MAX_NZ];
double w[MAX_NX*MAX_NY*MAX_NZ];
double p[MAX_NX*MAX_NY*MAX_NZ];

void my_pde_solver(int nx, int ny, int nz,
                   double u[nx][ny][nz],
                   double v[nx][ny][nz],
                   double w[nx][ny][nz],
                   double p[nx][ny][nz]);
```

- We could use VLA parameters
- But we should cast on calls, to avoid compiler warnings
  - How would you cast **`u[MAX_NX*MAX_NY*MAX_NZ]`** into **`double u[nx][ny][nz]`**?
- Maximum problem size is program limited: **`nx*ny*nz`** must be less than **`MAX_NX*MAX_NY*MAX_NZ + 1`**

Slightly More Comfortable, the Old Way


```
void my_pde_solver(int nx, int ny, int nz,
                   double u[],
                   double v[],
                   double w[],
                   double p[]) {
  // variable declarations and solver code...

  u[(i*ny + j)*nz + k] = ...;
  v[(i*ny + j)*nz + k] = ...;
  w[(i*ny + j)*nz + k] = ...;
  p[(i*ny + j)*nz + k] = ...;

  // more solver code...
```

- We could write code as the above, no need for casting on `my_pde_solver()` calls
- And you'll encounter code like this, that was a C89 way
- But so old fashioned!! Don't do that for new codes
- And remember, maximum problem size is limited

More Comfortable, Thanks to C99


```
void my_pde_solver(int nx, int ny, int nz,
                  double um[],
                  double vm[],
                  double wm[],
                  double pm[]) {

  double (*u)[ny][nz] = (double (*)[ny][nz])um;
  double (*v)[ny][nz] = (double (*)[ny][nz])vm;
  double (*w)[ny][nz] = (double (*)[ny][nz])wm;
  double (*p)[ny][nz] = (double (*)[ny][nz])pm;

  // solver code using u, v, w, and p as humans do
```

- Let's rewrite `my_pde_solver()` like this (and update function declaration as well!)
- Definitely easier to use
  - No casting on `my_pde_solver()` calls
  - And writing `my_pde_solver()` is easier too
- Maximum problem size still program limited, however

Removing Limitations

- Being program limited is annoying
- It's much better to accommodate to any user specified problem size
  - Right, as long as there is enough memory
  - But if memory is not enough, not our fault
  - It's computer or user's fault
- And there are many complex kinds of computations
  - Those in which memory need cannot be foreseen in advance
  - Those in which arrays do not fit
  - Those in which very complex data structures are needed

Enter Dynamic Allocation (from `stdlib.h`)


```
void *malloc(size_t size)
void *calloc(size_t el_count, size_t el_size)
```

- `malloc()` allocates a memory area suitable to host a variable whose size is `size`
  - Allocated memory is uninitialized.
  - Use it like this:

    ```
    a_ion_ptr = (ion *)malloc(sizeof(ion));
    ```

- `calloc()` allocates a memory area suitable to host an array of `count` elements, each of size `size`
  - Allocated memory is initialized to zero: can be slow, but useful
  - Use it like this:

    ```
    a_flt_ptr = (float *)calloc(nx*ny*nz, sizeof(float));
    ```

- Best practice: always cast return values, gives less compiler warnings and helps readability

The Biggest Mistake

- Assuming **`malloc()`** or **`calloc()`** succeeded!
- Where all these 'dynamic allocated memory' comes from?
  - From an internal area, often termed "*memory heap*"
  - When that is exhausted, OS is asked to give the process more memory
  - And if OS is short of memory, or some configuration limit is exhausted...
- On failure, **`malloc()`** and **`calloc()`** return null pointers
  - Dereferencing it forces program termination (usually a "segmentation fault")
  - We could say you deserve it
  - But all time spent in previous computations would be lost
- Best practice: ALWAYS, ALWAYS, always check

```
if ((p = malloc(some_size)) == NULL) {
   // save your precious data, if any
   // and fail gracefully
 }
```

Resizing

`void *realloc(void *ptr, size_t new_size)`

- **`realloc()`** takes a previously allocated memory area, and gives you a new area whose size is **`size`**
  - Original area contents are copied in the new area, up to min(*oldsize*, **`size`**)
  - Use it like this:

```
new_ptr = (float *)realloc(a_flt_ptr,
                          nx*ny*2*nz*sizeof(float));
```

- Particularly handy to shrink or lengthen arrays
- On failure, returns null pointer and leaves old area unchanged
- Biggest mistakes
  - Assuming **`realloc()`** succeeded: always check
  - Assuming only size changes and address remains the same: it can happen, but only in particular cases

Getting Rid of Memory Areas


```
void free(void *ptr)
```

- An allocated memory area persists until it is “freed”
- Of course, heap allocated memory is claimed back at process termination
- But better give back a memory area to the dynamic memory “pool” for reuse, as soon as you are over with it
  - Just imagine you are processing one item at a time...
  - Allocating new memory areas at each item without freeing previously allocated ones...
  - Your process size will grow until...
  - In jargon, this is a *memory leak*
- Remember: programmers causing memory leaks have particularly bad reputation

The First Big Mistake with `free()`


```
  char s[BIG_STRING + 1];
  char *p;
//....
  if ((p = malloc(BIG_STRING + 1)) == NULL) {
    // save your precious data, if any
    // and fail gracefully
  }
  strncpy(p, s, BIG_STRING);

  while (++p) {
    // process characters
  }
  free(p);  // p has been incremented!
  free(s);  // MADNESS: s not `malloced'!
```

- `free()` MUST be passed a pointer returned by `malloc()` and friends
- Otherwise behavior is implementation defined
- In most practical cases, program execution is aborted

The Second Big Mistake with `free()`


```
int *p, i;
long long *q;

if ((p = malloc(sizeof(int)*n)) == NULL) { /*take action*/ }
// process some data
free(p);

if (!(q = malloc(sizeof(long long)*m))) { /*take action*/ }
for(i=0; i<m; ++i)
  p[i] = i - m;  // a typo!
//...
```

- Memory still there, but could have been reused!
- Or could have not been reused as well...
- Could appear to work, very difficult to catch
- Good advice: always zero a pointer after freeing it
  - Can be done “automagically” if you

```
#define free(ptr_var) (free(ptr_var), ptr_var = NULL)
```

The Third Big Mistake with `free()`


```
  typedef struct mydata {
    int n;
    double *somedata;
    int *moredata;
  } mydata;

  mydata *p = calloc(1, sizeof(mydata));
  if (!p) { /* take action */ }

  p->n = datasize;
  p->somedata = calloc(datasize, sizeof(double));
  p->moredata = calloc(datasize, sizeof(int));
  if (!p->somedata || !p->moredata) { /* take action */ }

//input and process data

  free(p);  // forgot something?
```

- Freeing `p`, `p->somedata` and `p->moredata` are gone, so we can't free their pointees, memory leak!
- Free `p->somedata` and `p->moredata` first, then `p`

Memory Friends from `string.h`

Function	Does
`void *memmove(void *d, const void *s, size_t len)`	copies a `len` bytes sized memory area from `s` to `d`, returns `d`
`void *memset(void *p, int val, size_t len)`	writes `len` copies of `(unsigned char)val` starting from address `p`, returns `p`

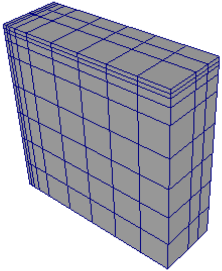
- You'll happen to encounter `memcpy()` too
  - Copies almost as `memmove()` does
  - If memory areas happen to overlap, `memmove()` is safe and does the right thing
  - While `memcpy()` could be faster, but is unsafe
  - Be prudent, and prefer `memmove()`
  - Surprisingly, `memmove()` is also faster in quite a few implementations!
- Way too common mistake: forgetting about and writing code doing the same
- Don't reinvent the wheel, use library functions!

Comfortable, and User Friendly


```
void my_pde_solver(int nx, int ny, int nz,
                  // physical parameters
                   ) {
//...
  double (*u)[ny][nz] = (double (*)[ny][nz])calloc(nx*ny*nz, sizeof(double));
  double (*v)[ny][nz] = (double (*)[ny][nz])calloc(nx*ny*nz, sizeof(double));
  double (*w)[ny][nz] = (double (*)[ny][nz])calloc(nx*ny*nz, sizeof(double));
  double (*p)[ny][nz] = (double (*)[ny][nz])calloc(nx*ny*nz, sizeof(double));

  if (u == NULL || v == NULL || w == NULL || p == NULL) {
    fprintf(stderr, "Not enough memory!\n");
    exit(exit_failure);
  }

  // solver code using u, v, w, and p in as humans do
```

- Now available memory is the limit
- And still easy to use

Nonuniform Grids



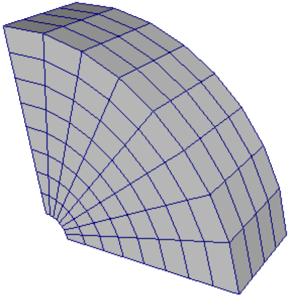
- Let’s imagine we have to solve a PDE
- On a dense, Cartesian, non uniform grid
  - Mesh axes are parallel to coordinate ones
  - Steps along each direction differ in size from point to point

Keeping Information Together

```
typedef struct nonuniform_grid {
  int nx, ny, nz;

  double *deltax; // Grid steps
  double *deltay;
  double *deltaz;
} nonuniform_grid;
//...
nonuniform_grid my_grid;

//...

mygrid.deltax = calloc(nx - 1, sizeof(double));
mygrid.deltay = calloc(ny - 1, sizeof(double));
mygrid.deltaz = calloc(nz - 1, sizeof(double));
// Check immediately for NULL pointers!
```

- Related information is best kept together
- Grid size and grid steps are related information

Structured Grids in General Form



- Let's imagine we have to solve a PDE
- On a dense structured mesh
  - Could be continuously morphed to a Cartesian grid
  - Need to know coordinates of each mesh point

Sketching a Mesh Description


```
typedef vect3D meshpoint;
typedef vect3D normal;

typedef struct mesh {
  int nx, ny, nz;

  meshpoint *coords;

  normal *xnormals;
  normal *ynormals;
  normal *znormals;

  double *volumes;
} mesh;
//...
nonuniform_grid my_grid;

mygrid.coords = calloc(nx*ny*nz, sizeof(meshpoint));
mygrid.xnormals = calloc(nx*ny*nz, sizeof(normal));
mygrid.ynormals = calloc(nx*ny*nz, sizeof(normal));
mygrid.znormals = calloc(nx*ny*nz, sizeof(normal));
mygrid.volumes = calloc((nx-1)*(ny-1)*(nz-1), sizeof(double));
// Check immediately for NULL pointers!
```

- No VLAs allowed in structures
- Cast to VLA array pointer in functions using it

Multiblock Meshes and More

- A multiblock mesh is an assembly of connected structured meshes
  - You could dynamically allocate a **`mesh`** array
  - Or build a **`block`** type including a **`mesh`** and connectivity information
- Adaptive Mesh Refinement
  - You want your blocks resolution to adapt to dynamical behavior of PDE solution
  - Which means splitting blocks to substitute part of them with more resolved meshes
- Eventually, you'll need more advanced data structures
  - Like lists (and recursion comes handy)
  - Like binary trees, oct-trees, n-ary trees (and recursion becomes essential)

If You Read Code Like This...


```
struct block_item;

typedef struct block_item {
  block *this_block;

  struct block_item *next;
} block_item;

//...
   while (p) {
     advance_block_in_time(p->this_block);
     p = p->next;
   }
```

- It is processing a singly-linked list of mesh blocks
- You need to learn more on abstract data structures
- Don’t be afraid, it’s not that difficult

And If You Read Code Like This...


```
struct block_tree_node;

typedef struct block_tree_node {
  block *this_block;

  int children_no;
  struct block_tree_node **childrens;
} block_tree_node;

//...
void tree_advance_in_time(block_tree_node *p) {
   int i;

   for(i=0; i<p->children_no; ++i)
     tree_advance_in_time(p->childrens[i]);

   advance_block_in_time(p->this_block);
}
```

- It is processing a tree of mesh blocks (AMR, probably)
- You need to learn more on abstract data structures
- Don’t be afraid, it’s not that difficult

Outline

What We Left Out (1 of 2)

- More preprocessor magic, like:
  - lots of predefined macros to automatically adapt your code to platforms and compilers
  - macros to write function with variable number of arguments
- More types, like:
  - extended integer types
  - wide and Unicode characters and related facilities
  - unions and bit fields, mostly used for OS programming
- More facilities to:
  - control the floating point environment
  - interact with the process environment
  - localize your program
- More facilities for robustness:
  - static and dynamic assertions
  - bounds checking functions for I/O and string management (C11 Annex K)
  - precise control of process termination

What We Left Out (2 of 2)

- More facilities for performance:
  - **`inline`** functions
  - control of data alignment in memory
- C11 threads support
- More functions
- More C practice
  - That's your job
- More about programming
  - Code development management tools
  - Debugging tools
  - Look among Cineca HPC courses

Looking for More

ANSI WG14
C Standard and Technical Corrigenda
`http://www.open-std.org/jtc1/sc22/wg14/www/standards`
`http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`

S. Summit
comp.lang.c Frequently Asked Questions
`http://www.c-faq.com/`

D. Dyer
The Top 10 Ways to get screwed by the "C" programming language
`http://www.andromeda.com/people/ddyer/topten.html`

S. Harbison, G. Steele
C A Reference Manual
Prentice Hall, 5th ed., 2002

A. Kelley, I. Pohl
C by Dissection: The Essentials of C Programming
Addison Wesley, 4th ed., 2000

A. Koenig
C Traps and Pitfalls
Addison Wesley, 1989