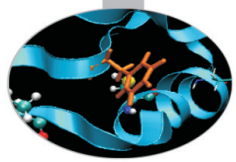
CINECA SCAI
SuperComputing Applications and Innovation

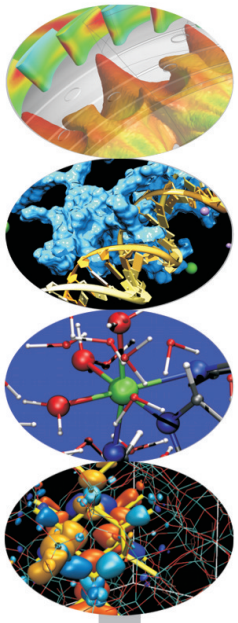


Scientific and Technical Computing in C

Day 1

Luca Ferraro Stefano Tagliaventi

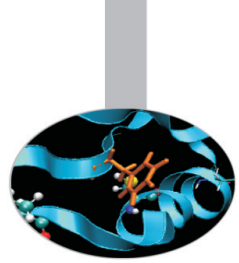
CINECA Roma - SCAI Department



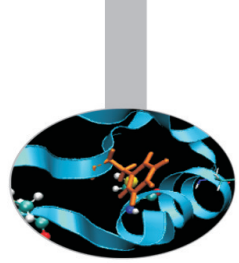
Roma, 29-30 October 2013

CINECA

Outline

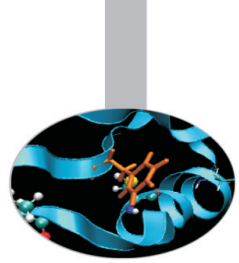


C History



- 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

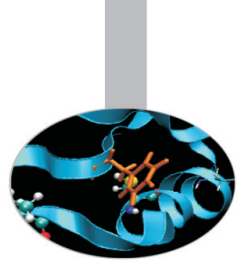


- A simple and efficient language
  - Only 44 reserved keywords
  - Basic data types and operators mapping "naturally" to the CPU
  - Facilities to build data types from the basic ones
  - Flexible flow control structures mapping the most common use cases
  - Translated by a compiler to machine language
- 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

Technical and Scientific Computing

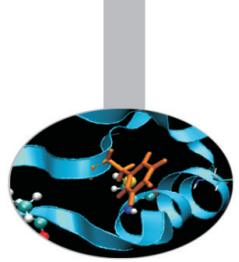
- Why C is bad
  - Number crunching has been traditionally done in Fortran
  - Fortran is older and more “rigid” than C, compilers optimize better
  - Nowadays, performance differences are often a matter of compiler flags and good programming techniques
- Why C is good
  - From the beginning, it had 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

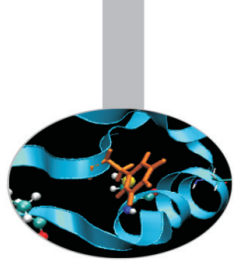
Outline



My First Scientific Program in C

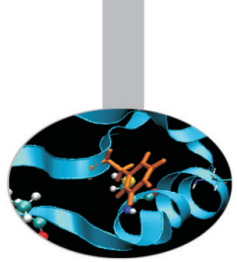

```
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>

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



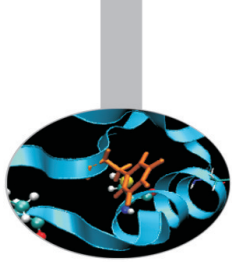
- Text following `/*` is ignored up to the first `*/` encountered, even if it's on a different line
- In C99, text following `//` is ignored up to the end of current line
- Best practice: do comment your code!
  - Variable contents
  - Algorithms
  - Assumptions
  - Tricks
- Best practice: do not over-comment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

Functions, `main()` in Particular



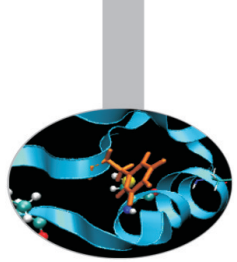
- C code is organized in functions
  - Each function has a name
  - Code goes in between braces
  - Arguments, if any, goes in between parentesis
  - It can return one or zero results using **`return`**
  - More on this later...
- In a program, the function **`main()`** can't be dispensed with
  - It's 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

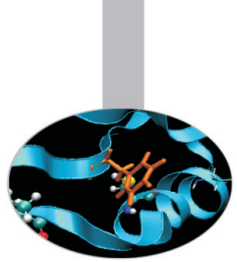


- A lot of functionalities are available in an external library of functions, whose content is defined by the Standard
- The compiler knows nothing about them, so it needs information about:
  - Arguments
  - Type of returned value
- Information about functions is in *header files*
  - Grouped by categories
  - 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

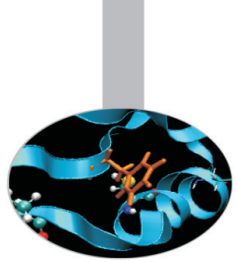


- Most of program work takes place in expressions
- Operators compute values from terms
  - `+`, `-`, `*` (multiplication), and `/` behave like in “human” arithmetic
  - So do unary `-`, `(`, and `)`
- `x1 = x1 + delta` assigns the value of expression `x1 + delta` to variable `x1`
  - An ending `;` makes 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 !



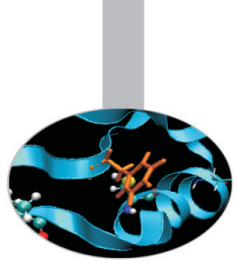
- We will use GNU C Compiler (GCC) during this course
  - Other compilers are available on the market (Intel, PGI, Pathscale, etc)
  - Linux systems comes with the C compiler
  - Windows systems does not have a default one
    - we will use MinGW (a minimal port of GCC for Windows)
- Let's see how to 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:

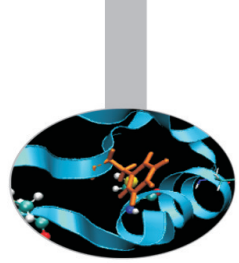
```
user@cineca$> gcc main.c
/tmp/ccWpSr3h.o: In function `main':
main.c:(.text+0xa8): undefined reference to `sqrt'
collect2: ld returned 1 exit status
```

  - **`#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:

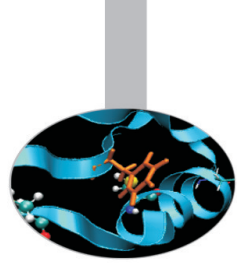
```
delta = sqrt(b*b - 4*a*c);
```

  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:

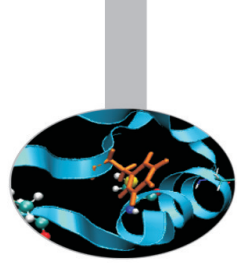
```
if (delta < 0.0)
  return 0;
```

  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:

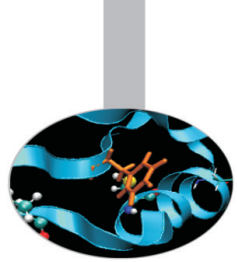
```
if (delta < 0.0) {
  printf("No real roots!\n");
  return 0;
}
```

  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

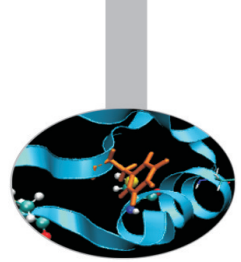



```
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>

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!


```
/* 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



- `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

As Complex as Possible!

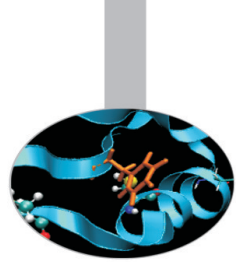

```
/* roots of a 2nd degree equation
   with real coefficients */
#include <math.h>
#include <stdio.h>
#include <complex.h>

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

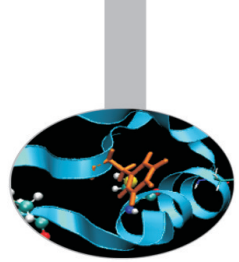


- What if user inputs zeroes for *a*, or *a* and *b*?
- Let's prevent these cases, inserting right after input:

```
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;
}
```

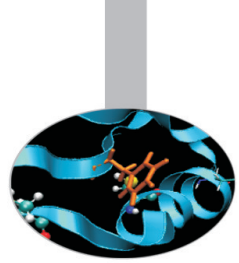
- Can you see the program logic?
- Try it now!
- Did you check that normal cases still work? Good.

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

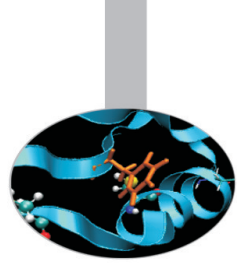


- Comments
  - Compiler disregards them, but humans do not
  - Please, use them
  - Do not abuse them, please
- Functions
  - One, at least: **`main()`**
  - Some of them come from the Standard Library
  - The proper header file must be **`#include`**d to use them
- Variables
  - Named memory locations 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

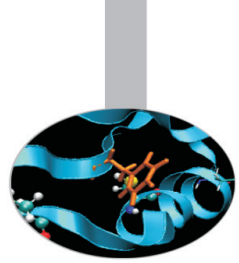
- Expressions
  - Compute values to store in variables
  - Compute values to pass to functions
- Statements
  - Units of work
  - Terminated by a `;`
- Compound statements (also said *blocks*)
  - Group a sequence of statements in a single entity
  - Wrapped in braces `{ }`
  - Do not need a terminating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

Program Flow Control

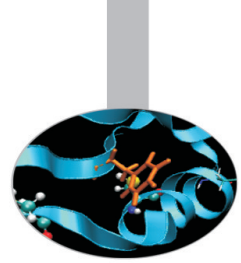


- **`return`** statements
  - Complete execution of the current function
  - Allow to return back a result
- Conditional statements
  - Allow conditional execution of code
  - Allow choice between alternate code paths

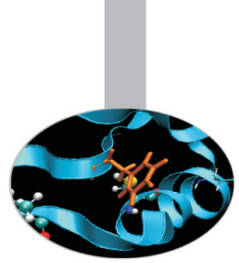
Best Practices



- 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



CINECA SCAI
SuperComputing Applications and Innovation



Scientific and Technical Computing in C

Day 1

Luca Ferraro Stefano Tagliaventi

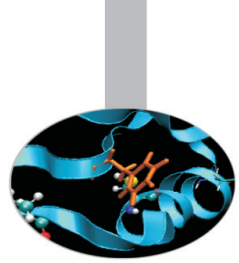
CINECA Roma - SCAI Department

Roma, 29-30 October 2013

CINECA SCAI
SuperComputing Applications and Innovation

CINECA

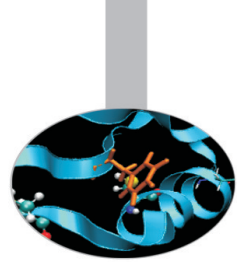
Outline



My First C Functions

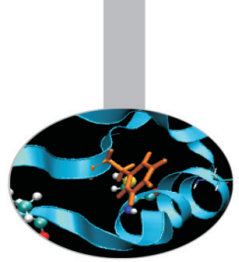

```
#include <math.h>


//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



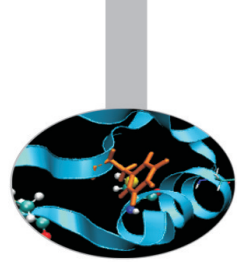
- Like variables, functions have names and types
  - Name must be an identifier
  - Type is the type of the returned result
- They have an associated compound statement, the function "body"
- Functions have formal parameters
  - Declared in a comma separated list, in parentheses
  - Each one is like 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



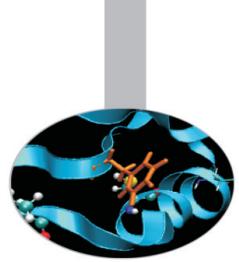
- The **`const`** qualifier
  - A **`const`** qualified variable can only be initialized
  - Compilers will bark if you try to change its value
- Best practice: always give name to constants
  - Particularly if unobvious, like **`1.0/137.0`**
  - It also helps to centralize updates (well, not for π)

- **`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!

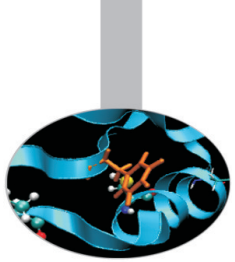

```
#include <math.h>


//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 = 3.141592653589793238;

  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);
}
```

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:

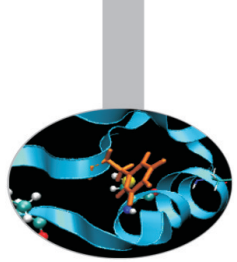
```
user@cineca$> gcc test_dsp.c dsp.c  -o test_dsp  -lm
```

  - **`-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
```

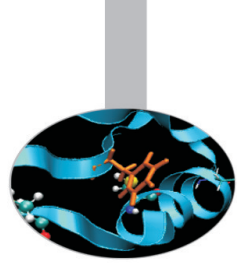
- much better ...

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
```

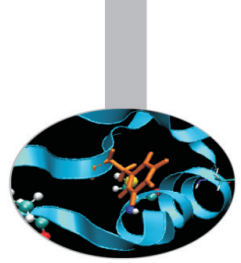
- *Function prototypes* are function declarations: a `;` replaces the function body
  - Parameters names are optional, but can be informative
- If **`DSP_H`** is already defined, preprocessor will remove the code before compiler is invoked
- Best practices:
  - Always play the above trick: complex programs 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




```
#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!


```
#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);
}
```

CINECA

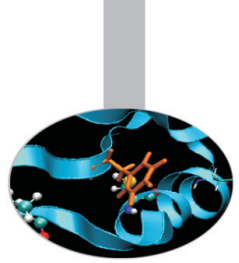
Compiler Errors and Warnings

- 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

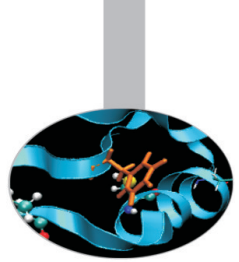
Building a Program



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
```

CINECA

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
```

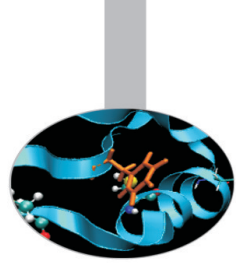
  - the **`sqrt()`** function is contained in the **`libm.a`** library
  - the math library is part of the Standard C Library, thus resides in a directory the compiler already knows about

Managing Wrong Arguments


```
#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

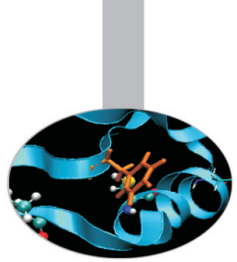
Failing Predictably




```
#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:

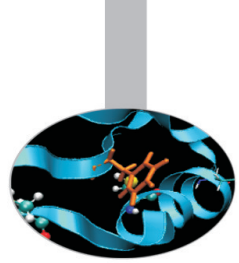
```
errno = 0;
a = rect(b, c);
if (errno)
{
  perror("rect():");
  //recovery action or controlled failure
}
```

- But there is more...

Total Robustness

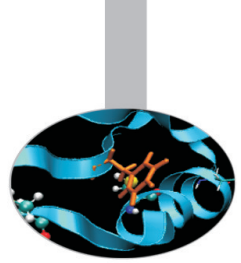
- Your platform could support IEEE floating point standard
  - Most common ones do, at least in a good part
- This means more bad cases:
  - one of the arguments is a NAN
  - both arguments are infinite (they are not ordered!)
- Best strategy: return a NAN and set **`errno`** in these bad cases
  - And 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!

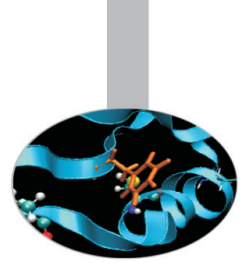


- Functions and their parameters
- Arguments are passed to functions by value
- A program can be subdivided in more source files
- Header files help to do it
- Preprocessor helps to write good header files
- Function prototypes
- **`const`** variables
- To **`if ()`**, zero is false and non zero is true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

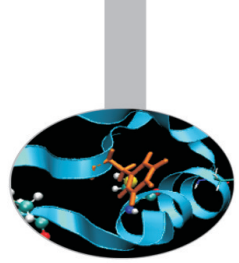
Best Practices



- 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



CINECA SCAI
SuperComputing Applications and Innovation



Scientific and Technical Computing in C

Day 1

Luca Ferraro Stefano Tagliaventi

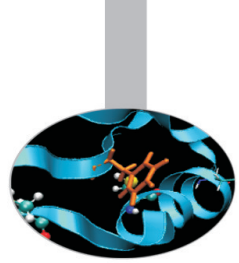
CINECA Roma - SCAI Department

Roma, 29-30 October 2013

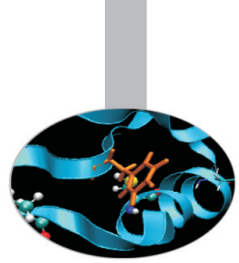
CINECA SCAI
SuperComputing Applications and Innovation

CINECA

Outline

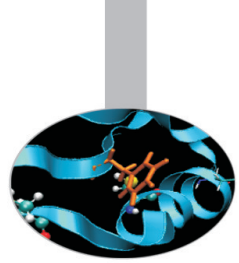


Greatest Common Divisor



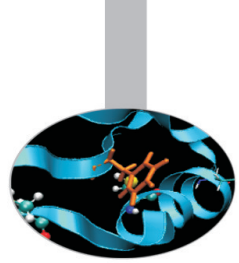
- Euclid's Algorithm
  1. Take two integers a and b
  2. Let $r \leftarrow a \mod b$
  3. Let $a \leftarrow b$
  4. Let $b \leftarrow r$
  5. If b is not zero, go back to step 2
  6. a is the GCD
- Let's implement it and learn some more C

GCD & LCM




```
#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

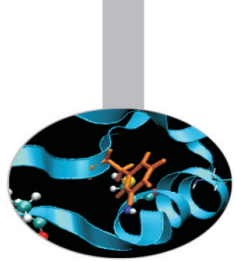


- **`int`** means that a value is an integer
  - Only integer values, positive, negative or zero
  - On most platforms, **`int`** means a 32 bits value, ranging from -2^{31} to $2^{31} - 1$
- Want to know the actual size?
  - **`sizeof(int)`** will return the size in bytes of the internal binary representation 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 ()`

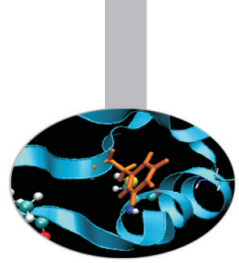


- **`while (`***logical-condition***`)`**
  statement
  1. Evaluates *logical-condition*
  2. If *logical-condition* is false (i.e. zero), goes to 5
  3. Executes *statement*
  4. Goes back to 1
  5. Skips *statement* and proceeds to execute the following code
- **`while ()`** is very similar to **`do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

Time for Testing

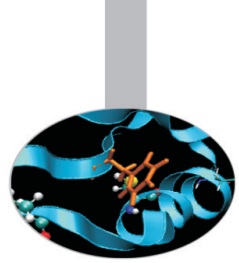
- Put the code in file **`numbertheory.c`**
- Write a suitable **`numbertheory.h`**
- Write a program to test both **`gcd()`** and **`lcm()`** on a pair of integer numbers
- Remember using **`%d`** for I/O
- Test it:
  - with pairs of small positive integers
  - with the following pairs: 15, 18; -15, 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...

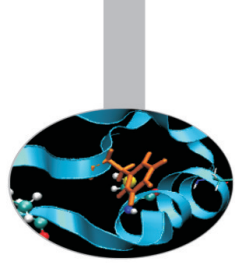


- Best way: generalize algorithm to the whole integer set
- gcd(a, b) is non negative, even if a or b is less than zero
  - Taking the absolute value of **`a`** and **`b`** using **`abs()`** will do
- gcd(a, 0) is $|a|$
  - Conditional statements will do
- gcd(0, 0) is 0
  - Already covered by the previous 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

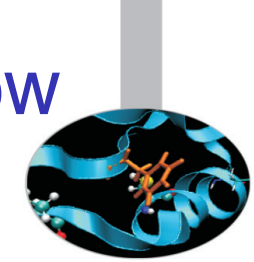

```
#include <stdlib.h>
#include "numbertheory.h"

// 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



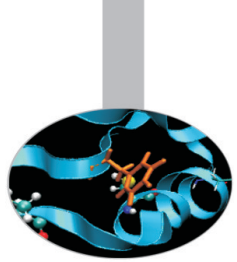
- `a*b/gcd(a,b)` same as `(a*b)/gcd(a,b)`
- What if the result of a calculation cannot be represented in the given type?
  - Technically, you get an arithmetic *overflow*
  - C is quite liberal: the result is implementation defined
  - Best practice: be very careful of intermediate results
- Easy fix: 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

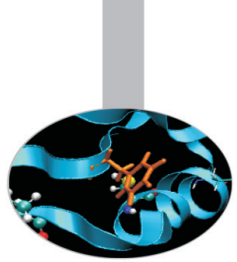


- Sometimes an integer type with a wider range of values is needed
- **`long int`** (commonly shortened to **`long`**)
  - **`LONG_MAX`** and **`LONG_MIN`** from **`limits.h`**
  - **`%ld`** conversion specifier in **`printf()`** and **`scanf()`**
  - But C Standard only says: can't be narrower than 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));
}
```

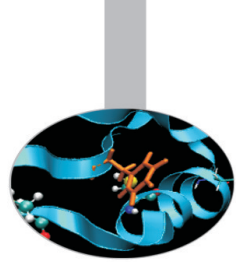
Call the Right Function!



- We had to call different functions for absolute value
  - **`labs()`** for **`long ints`**
  - **`llabs()`** for **`long long ints`**

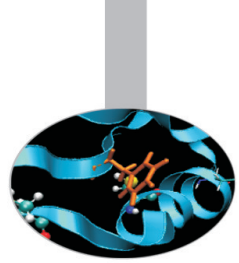
- What if you call, say, **`labs()`** for **`int`** or **`long long`** values?
  - Automatic conversion between different types happens!
  - But a narrower type cannot represent all possible values of a wider one
  - No problem when converting to a wider type
  - At risk of overflow (i.e. implementation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

Unsigned Integer Types



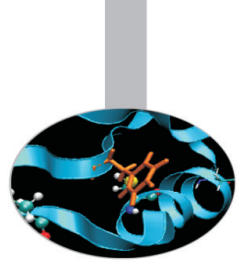
- **`unsigned int`** (often shortened to **`unsigned`**)
  - Same width as an **`int`**
  - No negative values, only positive integers, but nearly twice the ones in an **`int`**
  - **`UINT_MAX`** (from **`limits.h`**) is its greatest value
  - Use conversion specifier **`%u`** in **`printf()`** and **`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



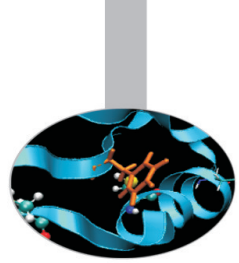
- Best practice: avoid useless work
  - `a*(b/gcd(a,b))` causes error if both `a` and `b` are zero
  - but it's useless anyway if `a` or `b` is zero, let's use `||` (logical OR) to avoid it
- Best practice: be loyal to C approach
  - You have now a `gcd()` function that works on the widest available 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



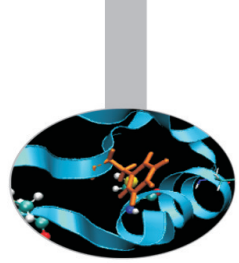

```
#include <stdlib.h>
#include "numbertheory.h"

// 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



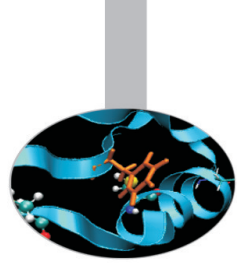
- (*type*) *expression*
  - Is an *explicit cast*
  - Forces conversion from expression type to specified one
  - And tells the compiler you know what you are doing
- The solution is not perfect
  - If you are working with a lot of basic `int`s, you are spending a lot of work in type conversions and 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

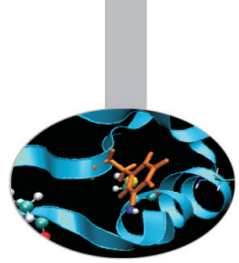


- Preprocessor macros
  - Their content is substituted wherever the macros appear in the code
  - Every occurrence of each parameter is replaced by the text given as argument
- A macro must be a “one-liner”
  - A \ at end of line is needed to continue on the next line
- The **##** operator concatenates 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:

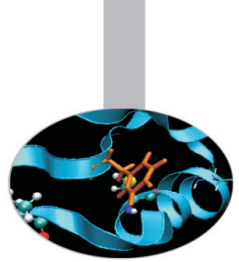
```
#define gcd(A, B) _Generic((A),                  \
                           int: gcd              \
                           long int: lgcd        \
                           long long int: llgcd   \
                           ) (A, B)

#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...

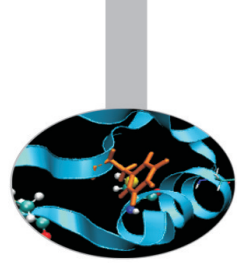
More Types and Flow Control

- 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

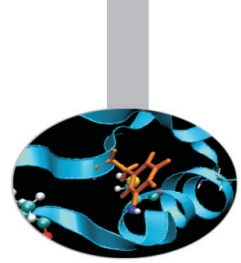
Best Practices



- 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



CINECA SCAI
SuperComputing Applications and Innovation



Scientific and Technical Computing in C

Day 1

Luca Ferraro Stefano Tagliaventi

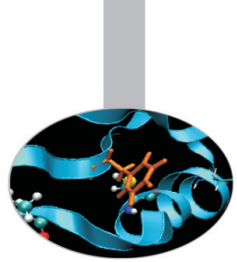
CINECA Roma - SCAI Department

Roma, 29-30 October 2013

CINECA SCAI
SuperComputing Applications and Innovation

CINECA

Outline



CINECA

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 $\leq$ `int` width $\leq$ `long` width $\leq$ `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`	$min\{x \mid 1.0 + x > 1.0\}$ in `float` type	$\leq 10^{-5}$
`DBL_EPSILON`	$min\{x \mid 1.0 + x > 1.0\}$ in `double` type	$\leq 10^{-9}$
`LDBL_EPSILON`	$min\{x \mid 1.0 + x > 1.0\}$ 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)`,	$\lvert\mathtt{x}\rvert$,
`double copysign(double x, double y)`[1]	if $\mathtt{y} \neq 0$ returns $\lvert\mathtt{x}\rvert\mathtt{y}/\lvert\mathtt{y}\rvert$ else returns $\lvert\mathtt{x}\rvert$
`double floor(double x)`, `double ceil(double x)`, `double trunc(double x)`[1], `double round(double x)`[1]	$\lfloor\mathtt{x}\rfloor$, $\lceil\mathtt{x}\rceil$, if $\mathtt{x} > 0$ returns $\lfloor\mathtt{x}\rfloor$ else returns $\lceil\mathtt{x}\rceil$, nearest[2] integer to `x`
`double fmod(double x, double y)`,	`x` mod `y` (same sign as `x`)
`double fdim(double x, double y)`[1]	if $\mathtt{x} > \mathtt{y}$ returns $\mathtt{x} - \mathtt{y}$ else returns 0
`double nextafter(double x, double y)`[1]	next representable value after `x` toward `y`
`double fmin(double x, double y)`[1]	$\min\{\mathtt{x}, \mathtt{y}\}$
`double fmax(double x, double y)`[1]	$\max\{\mathtt{x}, \mathtt{y}\}$

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

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]	$\mathtt{x} + i\mathtt{y}$,
`double complex cabs(double complex z)`, `double complex carg(double complex z)`, `double complex creal(double complex z)`, `double complex cimag(double complex z)`, `double complex conj(double complex z)`	$\lvert\mathtt{z}\rvert$, Argument of `z` (a.k.a. phase angle), Real part of `z`, Imaginary part of `z`, Complex conjugate of `z`
`double complex csqrt(double complex z)`, `double complex cpow(double complex z, double complex w)`	$\sqrt{\mathtt{z}}$, $\mathtt{z}^{\mathtt{w}}$
`double complex cexp(double complex z)`, `double complex clog(double complex z)`	$e^{\mathtt{z}}$, $\log_e \mathtt{z}$

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 $2^{type\ width}$, 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