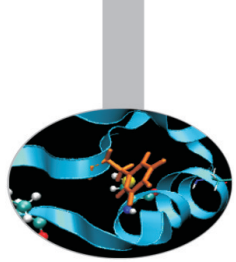
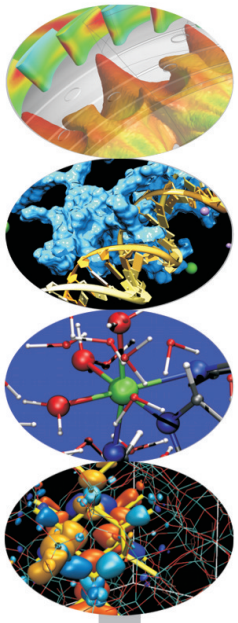
CINECA SCAI
SuperComputing Applications and Innovation



Scientific and Technical Computing in C

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department



Roma, 11-13 November 2015

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;
}
```

CINECA

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 parentheses
  - 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

CINECA

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 and Test It!


```c
/* 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;
}
```

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

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Roma, 11-13 November 2015

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 ...

CINECA

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

```
user@cineca$> gcc -E  dsp.c  >  dsp_cpp.c
user@cineca$> gcc -E  test_dsp.c  >  test_dsp_cpp.c
```

    - **`-E`** option, tells **`gcc`** to stop after pre-process
    - simply call **`cpp`**
    - by default output is sent to standard output (use > to redirect to a file)
  - compiling sources

```
user@cineca$> gcc -c  dsp_cpp.c
user@cineca$> gcc -c  test_dsp_cpp.c
```

    - **`-c`** option, tells **`gcc`** to compile the source
    - an object file **`.o`** is produced from each source file
  - linking object files together with external libraries

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`**is 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

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Roma, 11-13 November 2015

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 $2^{type\ width\ in\ bits}$
- 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)
```

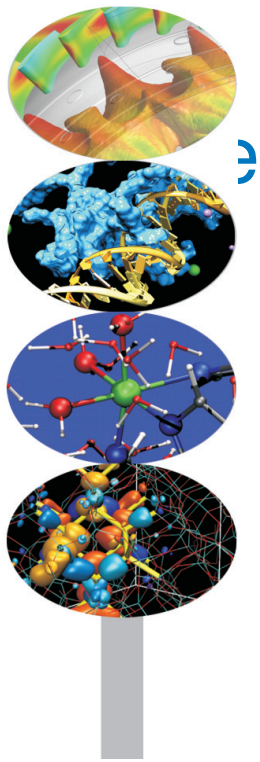
- Now you can use **`gcd()`** and **`lcm()`** for all argument types
- Coming to a compiler near you...

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



entific and Technical Computing in C

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Roma, 11-13 November 2015

Outline

Fibonacci Numbers

- Fibonacci numbers are of interest in many fields
  - Computer science: sorting, searching, algorithm analysis
  - Finance: trading algorithms and strategies
  - Simulation: pseudo random number generators
  - Data management: Fibonacci heaps, Fibonacci compression
  - Mathematics: number theory, graph theory
- Defined by a recurrence relation
  1. $F_0 = 0$
  2. $F_1 = 1$
  3. $F_n = F_{n-1} + F_{n-2}$
- Let's implement this definition in C

Recursive Fibonacci


```
#include "fibonacci.h"

unsigned int fib(unsigned int n) {
  unsigned int fn;

  if (n < 2)
    return n;

  fn = fib(n-1) + fib (n-2);

  return fn;
}
```

Recursion

- C supports recursive function calls
  - Direct: `fna()` calls `fna()`
  - Indirect: `fna()` calls `fnb()`, that calls `fnc()`, that in turn calls `fna()`...
- How can it work?

- Variables declared in a function are of *automatic* storage class
  - Automatically allocated when the function is called
  - Automatically deallocated when the function exits
  - Each call gets separate instances of the variables
  - Ditto for function parameters

- An elegant way to express some algorithms
  - As long as recursion terminates at some point...

Tests and Performance

- Put the code in file **`fibonacci.c`**
- Write a suitable **`fibonacci.h`**
- Write a program to test **`fib()`**
- Remember using **`%u`** for I/O
- Test it:
  - with integers from 0 to 10
  - with 45
- OK for small arguments, but gets really slow as the argument grows

Recursive Fibonacci: Try It Now!


```
#include "fibonacci.h"

unsigned int fib(unsigned int n) {
  unsigned int fn;

  if (n < 2)
    return n;

  fn = fib(n-1) + fib (n-2);

  return fn;
}
```

Recursion Can Be Really Bad

- Recursive Fibonacci helps explain recursion and why it can be bad
  - To compute `fib(n)`, `fib(n-1)` and `fib(n-2)` are called
  - But `fib(n-2)` is also called to compute `fib(n-1)`!
  - And `fib(n-3)` is called three times...
  - And `fib(n-k)` is called *k* times
  - Bottom line, the number of calls grows exponentially
- Memory can be an issue too!
  - The deepest chain of recursive calls has *n* levels
  - What if the function uses many variables?
  - Particularly when a limited memory area is used as a stack for automatic variables, as most implementations do
- Best practices:
  - avoid recursion if not strictly necessary
  - and pay attention to memory usage
  - Let's do better

Iterative Fibonacci


```
#include "fibonacci.h"

unsigned int fib(unsigned int n) {
  unsigned int fn = 1;
  unsigned int fnm1 = 0;

  if (n < 2)
    return n;

  while (--n) {
    unsigned int fnp1 = fn + fnm1;
    fnm1 = fn;
    fn = fnp1;
  }

  return fn;
}
```

Iterative Fibonacci: Try It Now!


```
#include "fibonacci.h"

unsigned int fib(unsigned int n) {
  unsigned int fn = 1;
  unsigned int fnm1 = 0;

  if (n < 2)
    return n;

  while (--n) {
    unsigned int fnp1 = fn + fnm1;
    fnm1 = fn;
    fn = fnp1;
  }

  return fn;
}
```

More Tests, and Robustness

- Test the new version:
  - with integers from 0 to 10
  - with 45, then 46, 47, and 48
- Much faster! but...
  - **`fib(48)`** is less than **`fib(47)`**!
  - The range of **`unsigned int`** has been exceeded
  - Let's take care in a standard way
- When a result exceeds the range of the function type
  - Set **`errno`** to **`ERANGE`**
  - And return a value that makes sense
  - The maximum representable value makes sense

Iterative Fibonacci: More Robust


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

#define UINT_MAX_FIB_N 47

unsigned int fib(unsigned int n) {
  unsigned int fn = 1;
  unsigned int fnm1 = 0;

  if (n > UINT_MAX_FIB_N) {
    errno = ERANGE;
    return UINT_MAX;
  }

  if (n < 2)
    return n;

  while (--n) {
    unsigned int fnp1 = fn + fnm1;
    fnm1 = fn;
    fn = fnp1;
  }

  return fn;
}
```

More About Performance

- The new version is much faster
  - In a single call, each term in the sequence is evaluated only once
- But still less than optimal
  - A term in the sequence can be evaluated again and again in different calls

- Possible solution:
  - let's build an indexed table of Fibonacci numbers, and let `fib()` return entries from it
  - we need to index the table with `fib()` argument
  - we need the table to persist across function calls

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

- ***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


```
#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];
}
```

More Speed ...

- More efficiency is possible
  - The first `fib()` call is slower
  - In subsequent calls, the call itself costs more than actual operations in `fib()`
  - No known plans to change Fibonacci numbers
  - Let's initialize the array with a list of precomputed values
  - And make it `const` to forbid assignments
- Pay attention to those capital `U`!
  - Tell the compiler the type is `unsigned int`
  - Use `L` for `long`, `LL` for `long long`, `UL` for `unsigned long`, `ULL` for `unsigned long long`
  - Lowercase letters also do, but are less readable
- Best practice:
  - rules assigning types to literal constants are complicated
  - and differ from C89 to C99
  - always make literal constants types explicit

fastestfibonacci.c


```
#include "fastestfibonacci.h"

const unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1] =
  { 0U, 1U, 1U, 2U, 3U, 5U, 8U, 13U, 21U, 34U, 55U,
    89U, 144U, 233U, 377U, 610U, 987U, 1597U, 2584U,
    4181U, 6765U, 10946U, 17711U, 28657U, 46368U,
    75025U, 121393U, 196418U, 317811U, 514229U,
    832040U, 1346269U, 2178309U, 3524578U, 5702887U,
    9227465U, 14930352U, 24157817U, 39088169U, 63245986U,
    102334155U, 165580141U, 267914296U, 433494437U, 701408733U,
    1134903170U, 1836311903U, 2971215073U
  };
```

fastestfibonacci.h


```
#ifndef FASTESTFIBONACCI_H

#define FASTESTFIBONACCI_H

#define UINT_MAX_FIB_N 47

extern const unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1];

#define fib(n) (FibonacciNumbers[(n)])

#endif
```

...Using Some “Magic”...

- **`extern`** tells the compiler that **`FibonacciNumbers`** array is a global symbol

- **`fib()`** became a preprocessor macro:
  - its body is textually substituted wherever **`fib()`** appears
  - **`n`** occurrences are replaced by the text given as argument
- In macros, do not spare parentheses
  - **`fib()`** could appear inside an expression
  - Macro parameter **`n`** could be an expression as well
  - Parentheses ensure correct order of subexpression evaluations

- As fast as possible, but we lost something
  - Robustness: where is error checking?
  - Preprocessor macros are text replacements
  - **`fib()`** is not a real function, no statements can be inserted

`fastestfibonacci.h`, Release 1.5


```
#ifndef FASTESTFIBONACCI_H

#define FASTESTFIBONACCI_H

#include <limits.h>
#include <errno.h>

#define UINT_MAX_FIB_N 47

extern const unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1];

#define fib(n) ((n)>UINT_MAX_FIB_N ? (errno = ERANGE , UINT_MAX) \
                                   : FibonacciNumbers[(n)])

#endif
```

...and More C “Magic”...

- In C is possible to do in an expression what is usually done with statements
- More operators:
  - *cond* `?` *expr1* `:` *expr2* evaluates *expr1* or *expr2* if *cond* is true or false respectively
  - *expr1* `,` *expr2* evaluates both operands but returns the value of the second one
- Again, a macro must be a “one-liner”
  - A `\` at end of line is needed to continue on the next line
- But a macro has no parameter types
  - Thus no argument type checking
  - Thus `n` could be negative...
  - Let’s play the same trick with `?` `:` and `,`
  - In a different way, to make it more readable

`fastestfibonacci.h`, Release 2.0


```
#ifndef FASTESTFIBONACCI_H

#define FASTESTFIBONACCI_H

#include <limits.h>
#include <errno.h>

#define UINT_MAX_FIB_N 47

extern const unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1];

#define _ONLYPOSN_fib(n) ((n)>UINT_MAX_FIB_N ? (errno = ERANGE , UINT_MAX) \
                                             : FibonacciNumbers[(n)])

#define fib(n) ((n)<0 ? (errno = EDOM, 0) : _ONLYPOSN_fib(n))

#endif
```

... at a Price

- Complexity!
- Macros can ...
  - be nested
  - generate code for you
  - play elegant tricks
- ...but...
  - You loose type checking of arguments
  - An expression passed for `n` may be evaluated more than once
  - And this is particularly bad if expression contains operators with side effects like = or ++
- Best practices:
  - use macros only if you really need them
  - force correct calculations with parentheses
  - avoid , operator, it's confusing, and at least enclose its expression in parentheses

Really Big FIbonacci Numbers

- **`unsigned`** type range usually allows for 47 exact terms to be computed
- **`unsigned long long`** will guarantee you at least 93 exact terms
- Want more headroom and all digits?
  - Use a multiple precision library like GMP
- Just need a wider range and the most significant digits?
  - Use floating point types
  - A typical IEEE **`double`** will give you 53 bits for the mantissa (i.e. ~17 decimal digits) and ranges up to 10^{308}
  - A **`long double`**, on an x86 CPU, will give you 64 mantissa bits (i.e. ~20 decimal digits) and ranges up to 10^{4932}

A Possible Range Extension


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

long double fibl(unsigned int n) {
  unsigned int fn;

  errno = 0;
  fn = fib(n);
  if (errno == ERANGE) {
    const long double s5 = sqrtl(5.0);
    long double nl = (long double)n;

    errno = 0;
    return (powl((1.0+s5)*0.5, nl) + powl((1.0-s5)*0.5, nl))/s5;
  }

  return (long double)fn;
}
```

- Calls exact one for small values
- Should exact one fail, computes (in finite precision!) a closed form: $F_n = \frac{\phi^n+(1-\phi)^n}{\sqrt{5}}, \phi = \frac{1+\sqrt{5}}{2}$
- Robustness for free:
  - Argument is guaranteed to be a positive integer
  - Relies on normal floating point behavior

More and More C

- More types:
  - unsigned integers and **`long double`**
  - **`void`**
  - types of integer literal constants
  - careful mixing of different types
  - and arrays!
- More flow control
  - recursion (if really needed)
  - **`while ()`** and **`for (;;)`**
- More operators:
  - **`++`**, **`--`**, **`? :`**
  - and **`,`** (if really needed)
- How to make a local variable outlast a function call
- How to control variable and function “visibility”
- Something about preprocessor macros

Best Practices

- Avoid recursion if possible
  - May impact performance
  - May consume a lot of memory
- Hide implementation details
  - They are often irrelevant
  - Exposing them paves the way to troubles
- Always give explicit types to literal constants
  - Unless the value can be represented in all types
- Use macros only when really needed
  - And parenthesize pedantically
- For robustness, rely on the language whenever possible

Rights & Credits

Slides and examples were authored by:

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