SuperComputing Applications and Innovation

Scientific and Technical Computing in C

Stefano Tagliaventi   Isabella Baccarelli

CINECA Roma - SCAI Department

Rome, 3rd-5th May 2017

Outline

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**	≤ -128
SCHAR_MAX	maximum value of **signed char**	≥ 127
UCHAR_MAX	maximum value of **unsigned char** type	≥ 255
SHRT_MIN	minimum value of **short**	≤ -32768
SHRT_MAX	maximum value of **short**	≥ 32767
USHRT_MAX	maximum value of **unsigned short**	≥ 65535
INT_MIN	minimum value of **int**	≤ -32768
INT_MAX	maximum value of **int**	≥ 32767
UINT_MAX	maximum value of **unsigned**	≥ 65535
LONG_MIN	minimum value of **long**	≤ -2147483648
LONG_MAX	maximum value of **long**	≥ 2147483647
ULONG_MAX	maximum value of **unsigned long**	≥ 4294967295
LLONG_MIN	minimum value of **long long**	$\leq -9223372036854775808$
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

More Bitwise Arithmetic

- Left and right shifts
  - `a«n` same as `a*`2^n modulo $2^{type\ width\ in\ bits}$
  - `a»n` same as `a/`2^n
  - Precedence lower than `~` but higher than `|`, `&`, and `^`
  - Beware: if $n >$ *type width in bits*, or $n < 0$, result is undefined
- Applications
  - `isodd = (a&1);` same as `isodd = a%2;`
  - `b&255` same as `b%256`
  - `a | 15` same as `(a/16)*16 + 15`
- You have to think in base 2 to get why and if it works
  - Think of the examples above ... did you get the pattern?
  - 256 is 2^8 and 255 is $2^8 - 1$
  - 16 is 2^4 and 15 is $2^4 - 1$
  - `a | 19` is NOT the same as `(a/20)*20 + 19`

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>

Arithmetic
Integers
Floating
Expressions
Mixing Types

Aggregate
Structures
Defining Types
Arrays
Storage & C.
More Arrays

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

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

Floating Literal Constants

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

`double` Math

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

1. C99
2. If $\mathbf{x}$ 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]	$\mathbf{x} + i\mathbf{y}$,
`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)`	$\lvert\mathbf{z}\rvert$, 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)`	$\sqrt{\mathbf{z}}$, $\mathbf{z}^{\mathbf{w}}$
`double complex cexp(double complex z)`, `double complex clog(double complex z)`	$e^{\mathbf{z}}$, $\log_e \mathbf{z}$
1. C11	

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

`float` and `long double` Math

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

Expressions

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

Arithmetic Expressions

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

Hitting Limits

- All types are limited in range
- What about:
  - `INT_MAX + 1`? (too big)
  - `INT_MIN*3`? (too negative)
- Technically speaking, this is an arithmetic *overflow*
- And division by zero is a problem too
- For signed integer types, the Standard says:
  - behavior and results are unpredictable
  - i.e. up to the implementation
- For other types, the Standard says:
  - arithmetic on unsigned integers must be exact modulo $2^{type\ width}$, 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

Scientific and Technical Computing in C

Stefano Tagliaventi   Isabella Baccarelli

CINECA Roma - SCAI Department

Rome, 3rd-5th May 2017

Outline

struct


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

struct vect3D va, vb;

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

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

  return c;
}

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

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

structs Are Flexible


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

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

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

`structs`: a Concrete Example

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

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

typedef


```
typedef struct vect3D position, velocity;

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

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

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

ion a;
```

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

`typedef` in C Standard Library

- `typedef` is widely used in C Standard Library
- Mostly to abstract details that may differ among implementations
- E.g. `size_t` from `stddef.h`
  - Type of value returned by `sizeof()`
  - Different platforms allow for different memory sizes
  - `size_t` must be "`typedef`ed" to an integer type able to represent the maximum possible variable size allowed by the implementation
- E.g. `clock_t` from `time.h`
  - Type of value returned by `clock()`
  - Cast it to `double`, divide by `CLOCK_PER_SEC`, ...
  - and you'll know the CPU time in seconds used by your program from its beginning

Arrays

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

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

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

Array Memory Layout


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

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

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

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

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

A Very Important Digression

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

Storage Duration

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

Scopes

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

Variable Initializers

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

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

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

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

is legal, while the following:

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

obviously is not

More on Variable Initializers

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

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

- Same for arrays, as in:

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

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

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

Arrays and Storage Classes

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

Arrays as Function Arguments

- Arrays can be huge
  - And usually are, in S&T computing
  - Passing them by value would be too costly
- Moreover, arrays cannot be used in assignments
  - Thus a function cannot return an array
- The solution
  - The address of the array is passed to a function
  - And elements can be accessed by it
  - (Later on, you'll understand how)
- This allows elements to be assigned to
  - Thus a function has a way to "return" an array result
  - A mixed blessing: allows changes to happen by mistake
- Best practice: declare an array parameter `const` if your only intent is reading its elements

Averaging, the C99 Way

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

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

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

  return sum/n;
}
```

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

Averaging, the Old Way

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

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

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

  return sum/n;
}
```

Calling **`avg()`**

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

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

// read or compute N doubles into mydata[]

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

Averaging Arrays of 5 Elements

- Let's write a function to average arrays of 5 **`double`**s
- And make it generic, as usual
- Again, VLA parameters come to the rescue

```
void avg5(int n, const double a[n][5], double b[5]) {
  int i, j;

  for (j=0; j<5; ++j)
    b[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<5; ++j)
      b[j] += a[i][j];

  for (j=0; j<5; ++j)
    b[j] /= n;
}
```

Notice: this order of loops nesting gives faster execution

Averaging Arrays of 5 Elements, the Old Way

- Let's write a function to average arrays of 5 **doubles**
- And make it generic, as usual
- Again, do not specify first bound
- Again, it's equivalent

```
void avg5(int n, const double a[][5], double b[5]) {
  int i, j;

  for (j=0; j<5; ++j)
    b[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<5; ++j)
      b[j] += a[i][j];

  for (j=0; j<5; ++j)
    b[j] /= n;
}
```

Calling **avg5()**

- New or old style, simply pass array dimension and name
- If **avg5()** is written using VLAs, pedantic compilers may give a warning on function call, even if it's correct: they are wrong, check with Standard document or good book

```
double mydata[N][5];
double mydata_avg[5];

// read or compute N 5-uples of doubles into mydata[]

avg5(N, mydata, mydata_avg);
```

Averaging Arrays of Arbitrary Length

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

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

  int i, j;

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

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

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

Notice: this order of loops nesting gives faster execution

Calling Generic **avg()**

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

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

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

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

More on casts

(type-name) cast-expression

- Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.
- i.e

```
double mydata4[N];
foo((double [N])mydata4); // INVALID. The type name does not specify a scalar type.

.....

struct bar
{
   double x,y,z;
};
struct bar var;

foo((struct bar)var); // INVALID. Neither the type name of the cast nor the operand hav
```

Matrix Algebra, the C99 Way

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

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

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

  return sum;
}
```

Beware: compiler will not check the array dimensions match!

Matrix Algebra, the Old Way

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

Rights & Credits

Slides and examples were authored by:

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