

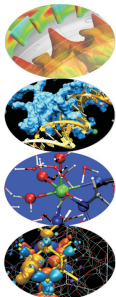


Scientific and Technical Computing in C

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Rome, 3-5 February 2016



Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- 1 Pointer Types
 - Pointers Basics
 - Pointers and Arrays
 - Generic Pointers
- 2 Characters and Strings
- 3 Input and Output
- 4 Managing Memory
- 5 Conclusions

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Memory? Addresses?

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointer vs. Pointee

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

```
p = &a;
```

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

```
a += 1;
```

p:
a:

p:
a:

p:
a:

p:
a:

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

    return c;
}
```

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

Did we say “valid”?

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
 And Arrays
 void

Strings

Chars
 Strings
 Manipulations
 Command Line

I/O

Files
 Text
 Binary

Memory

Allocation
 Data Structures

Finale

```

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

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

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

    return &c; // MADNESS!!
}

```

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

Returning More Than One Result

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
#include <math.h>

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

struct vect3d versor_norm(const struct vect3D *u, double *norm) {
    struct vect3D c = {0.0, 0.0, 0.0};
    double n, invn;

    n = u->x*u->x + u->y*u->y + u->z*u->z;
    if (n == 0.0) {
        *norm = 0.0;
        return c;
    }

    n = sqrt(n);
    *norm = n;
    invn = 1.0/n;
    c.x = u->x*invn;
    c.y = u->y*invn;
    c.z = u->z*invn;
    return c;
}
```

- We have to return two results
- Of very different types and meanings
- Assembling them in a bigger **struct** makes little sense

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- `double *p[10]`
 - it's an array of 10 pointers to `double`
- and `double *p[10][3]`
 - it's an array of 10 arrays, each of 3 pointers to `double`
- while `double (*p)[10]`
 - it's a pointer to array of 10 `doubles`
- and `double (*p)[10][3]`
 - it's a pointer to an array of 10 arrays, each of 3 `doubles`
- Confusing? It's logical: operator `[]` has higher precedence than `*`
- But easily becomes nasty!
 - What's `double (*p[10])[3]`?
 - And `double (*(p[10])[3][5])[8][2]`?
- Best practice: use `cdec1` tool to familiarize and decrypt

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
#include <stdio.h>

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

int main() {

    double *p;

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

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

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

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

    return 0;
}
```

Array Names and Pointers

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

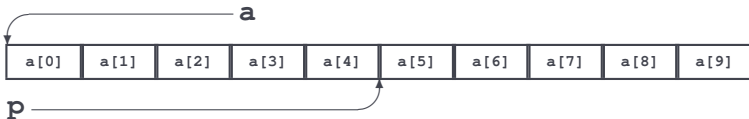
Files
Text
Binary

Memory

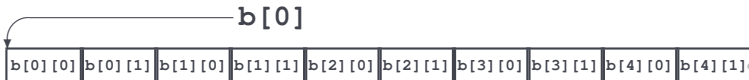
Allocation
Data Structures

Finale

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



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



Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

    return sum/n;
}
```

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- New or old style, array or pointer, simply pass array dimension and name
- And part of arrays could be managed too, independently of how it is written

```
double mydata[N];  
double mydata_avg;  
double firsthalf_avg, secondhalf_avg;  
  
// read or compute N doubles into mydata[]  
  
mydata_avg = avg(N, mydata);  
firsthalf_avg = avg(N/2, mydata);  
secondhalf_avg = avg(N - N/2, mydata + N/2);
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

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

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

Calling Generic `avg()`

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

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

Averaging Arrays, Another Classic Flavor

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

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

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

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

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

Matrix Algebra, the Old Way



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

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

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

    return sum;
}
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale



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

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

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

    return sum;
}
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Matrix Algebra, yet Another Classic Flavor

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

    return sum;
}
```


Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

    return sum;
}
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

- But generic pointers are essential to other purposes

qsort ()

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Declaration (from `stdlib.h`):

```
void qsort (  
    void *base,  
    size_t count,  
    size_t size,  
    int (*compare)(const void *e1, const void *e2) );
```

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

Sorting with `qsort ()`

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Define a comparison function like:

```
int comparedoubles(const double *a, const double *b) {  
    if (*a == *b)  
        return 0;  
  
    if (*a > *b)  
        return 1;  
  
    return -1;  
}
```

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

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

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

Scientific and Technical Computing in C

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Rome, 3-5 February 2016

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

1 Pointer Types

2 Characters and Strings

Characters

Strings

String Manipulation Functions

Parsing the Command Line

3 Input and Output

4 Managing Memory

5 Conclusions

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

#include <ctype.h>

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Strings

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
char decdigits[10];
```

```
//...
```

```
strcpy(decdigits, "0123456789");
```

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
// Frequencies of alphabetic characters in a text

#include <stdio.h>
#include <ctype.h>

#define LETTERS 26
#define CHUNK 256
unsigned counts[LETTERS];
char s[CHUNK+1];

int main() {
    int i;

    while ( fgets(s, sizeof(s), stdin) != NULL ) {
        char *p = s;

        while (*p) {
            if (isalpha(*p))
                ++counts[toupper(*p) - 'A'];
            ++p;
        }
    }

    for(i=0; i<LETTERS; ++i)
        printf("%c\t%9u\n", i + 'A', counts[i]);

    return 0;
}
```

Putting Characters and Strings to Work...

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- We work on up to 256 characters at the time
 - But must accommodate for terminating `'\0'`
- `fgets ()` is a robust I/O function
 - Reads from a file until end of line
 - Stores characters, including `'\n'`, into `s`
 - But no more than `sizeof(s) - 1`
 - Null terminates the string
 - And returns NULL on end of input, or failure
- Loop terminates when `p` points to terminating `'\0'`

...Putting Characters and Strings to Work

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- **char** is an integer type, we can do arithmetic on it
 - And alphabetic characters are coded in alphabetic order
- Remember: static variables are initialized to zero
- And some more I/O:
 - **%c**: emits a character from its code
 - **%9u**: prints a right-justified number in a field of width 9
 - **%s** is used for strings, as you'll see shortly
- Let's try it right now!
- Giving input from keyboard first, then from file...

charfreq.c

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
// Frequencies of alphabetic characters in a text

#include <stdio.h>
#include <ctype.h>

#define LETTERS 26
#define CHUNK 256
unsigned counts[LETTERS];
char s[CHUNK+1];

int main() {
    int i;

    while ( fgets(s, sizeof(s), stdin) != NULL ) {
        char *p = s;

        while (*p) {
            if (isalpha(*p))
                ++counts[toupper(*p) - 'A'];
            ++p;
        }
    }

    for(i=0; i<LETTERS; ++i)
        printf("%c\t%9u\n", i + 'A', counts[i]);

    return 0;
}
```

#include <string.h>

Function	Does
<code>size_t strlen(const char *s)</code>	returns actual string length
<code>char *strcpy(char *d, const char *s, size_t n)</code>	copies <code>n</code> characters from <code>s</code> to <code>d</code> , returns <code>d</code>
<code>char *strncat(char *d, const char *s, size_t n)</code>	appends <code>n</code> characters from <code>s</code> to <code>d</code> , returns <code>d</code>
<code>int strcmp(const char *s1, const char *s2)</code>	lexicographic comparison of <code>s1</code> and <code>s2</code>
<code>int strncmp(const char *s1, const char *s2, size_t n)</code>	lexicographic comparison of <code>s1</code> and <code>s2</code> , up to <code>n</code> characters
<code>char *strchr(const char *s, int c)</code>	returns pointer to first occurrence in <code>s</code> of character <code>c</code> , <code>NULL</code> if not found
<code>char *strrchr(const char *s, int c)</code>	returns pointer to last occurrence in <code>s</code> of character <code>c</code> , <code>NULL</code> if not found
<code>char *strcspn(const char *s, const char *set)</code>	returns pointer to first occurrence in <code>s</code> of any character in <code>set</code> , <code>NULL</code> if not found
<code>char *strspn(const char *s, const char *set)</code>	returns pointer to first occurrence in <code>s</code> of any character not in <code>set</code> , <code>NULL</code> if not found
<code>char *strstr(const char *s, const char *sub)</code>	returns pointer to first occurrence in <code>s</code> of string <code>sub</code> , <code>NULL</code> if not found
<code>char *strtok(const char *s, const char *set)</code>	allow to separate string <code>s</code> into tokens, read documentation

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

More Friends from `stdlib.h`

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Function	Returns conversion of initial portion of <code>s</code> to
<code>strtof(const char *s, char **p)³</code>	<code>float</code> ¹
<code>strtod(const char *s, char **p)</code>	<code>double</code> ¹
<code>atof(const char *s)</code>	<code>double</code>
<code>strtold(const char *s, char **p)³</code>	<code>long double</code> ¹
<code>atoi(const char *s)</code>	<code>int</code>
<code>strtol(const char *s, char **p, int base²)</code>	<code>long</code> ¹
<code>atol(const char *s)</code>	<code>long</code>
<code>strtoul(const char *s, char **p, int base²)</code>	<code>unsigned long</code> ¹
<code>strtoll(const char *s, char **p, int base²)³</code>	<code>long long</code> ¹
<code>atoll(const char *s)³</code>	<code>long long</code>
<code>strtoull(const char *s, char **p, int base²)³</code>	<code>unsigned long long</code> ¹
1. If <code>p</code> is not null, sets it to first character after converted portion of <code>s</code> 2. The <code>base</code> used in string representation ranges from 2 to 36 (!). 3. C99	

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

Yes, Pointers can be Pointees!

Pointers

Basics
 And Arrays
 void

Strings

Chars
 Strings
 Manipulations
 Command Line

I/O

Files
 Text
 Binary

Memory

Allocation
 Data Structures

Finale

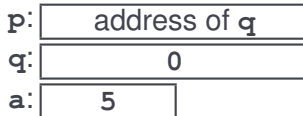
```

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

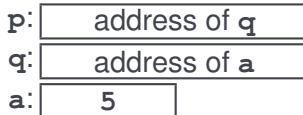
```



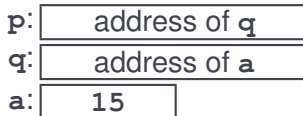
```
p = &q;
```



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



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



Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Our program to compute characters frequencies in texts was appreciated and we got request for improvements
 - It's the price of success with software
- Some folks dislike uppercase output and want it lowercase
- Some folks disregard frequencies lower than some threshold
- Some more folks do not want zero frequencies to be output at all
 - Actually a restricted form of the previous request
- And some folks want the text to be read by a user specified file
- To accommodate their requests, let's refactor first

Computational Linguistics

Refactored

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
// Frequencies of alphabetic characters in a text
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define LETTERS 26
#define CHUNK 256
unsigned counts[LETTERS];
char s[CHUNK+1];
char *filename = NULL; // will point to filename command line argument, if any
char outcase = 'A'; // change to 'a' for lowercase output
int minoutcount = 0; // minimum frequency suitable for output

void parsecmdln(int n, char *args[]) { /* add command line processing here */ }

int main(int argc, char *argv[]) {
    int i;

    parsecmdln(argc, argv);

    while ( fgets(s, sizeof(s), stdin) != NULL ) {
        char c, *p = s;

        while ((c = *p++))
            if (isalpha(c))
                ++counts[toupper(c) - 'A'];
    }

    for(i=0; i<LETTERS; ++i)
        if (counts[i] >= minoutcount)
            printf("%c\t%9u\n", i + outcase, counts[i]);

    return 0;
}
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
// Frequencies of alphabetic characters in a text
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define LETTERS 26
#define CHUNK 256
unsigned counts[LETTERS];
char s[CHUNK+1];
char *filename = NULL; // will point to filename command line argument, if any
char outcase = 'A'; // change to 'a' for lowercase output
int minoutcount = 0; // minimum frequency suitable for output

void parsecmdln(int n, char *args[]) { /* add command line processing here */ }

int main(int argc, char *argv[]) {
    int i;

    parsecmdln(argc, argv);

    while ( fgets(s, sizeof(s), stdin) != NULL ) {
        char c, *p = s;

        while ((c = *p++))
            if (isalpha(c))
                ++counts[toupper(c) - 'A'];
    }

    for(i=0; i<LETTERS; ++i)
        if (counts[i] >= minoutcount)
            printf("%c\t%9u\n", i + outcase, counts[i]);

    return 0;
}
```

Our Options

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- `-l` will force lowercase output
- `-t n` will set a minimum threshold for output
- An optional *filename* will specify a file to read from
- Let's add before `parsecmdln()` a function to call in response to `-h`
- And an helper function to manage user mistakes

```
void printUsage(void) {
    printf("charfreq [options] [filename]\n");
    printf("filename      input text (default: stdin)\n");
    printf("Options:\n");
    printf("-t n          frequency threshold\n");
    printf("-l          lowercase output\n");
    printf("-h          this help\n");
}
```

```
void illegalopt(const char *o) {
    fprintf(stderr, "illegal option: %s\n", o);
    printUsage();
    exit(EXIT_FAILURE);
}
```

Command Line Parsing

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
void parsecmdln(int n, char *args[]) {
    int i = 0;

    while (++i < n) {
        char *p = NULL;
        long th;

        if (args[i][0] != '-') {
            filename = args[i];        // must be filename
            break;                    // ignore anything following
        }

        switch (args[i][1]) {
            case 'l':
                outcase = 'a';
                break;
            case 't':
                th = strtol(args[++i], &p, 10); // numeric argument follows
                if (p == args[i] || th < 0) {
                    fprintf(stderr, "invalid or negative threshold\n");
                    exit(EXIT_FAILURE);
                }
                minoutcount = th;
                break;
            case 'h':
                printUsage();
                exit(0);
                break;
            default:
                illegalopt(args[i]);
        }
    }
}
```


Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
void print_help_and_exit () {  
    printf("Usage: ./shapp [-l|-t|-h]\n");  
    exit(EXIT_FAILURE);  
}  
  
int main(int argc, char *argv[]) {  
  
    if(argc < 2 || argv[1][0]!='-')  
        print_help_and_exit();  
    switch(argv[1][1])  
    {  
        case 't':  
            timestamp_ordering();  
            break;  
        case 'r':  
            reverse_order();  
            break;  
        case 'h':  
            print_help_and_exit();  
        default:  
            print_help_and_exit();  
    }  
}
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
• switch (integer-expression) {  
    case constant-expression:  
        statements  
    [case constant-expression:  
        statements]  
    [default:  
        statements]  
}
```

- 1 Evaluates *integer-expression*
- 2 If value equals one *constant-expression*, execution jumps to the statement following it
- 3 Otherwise, if **default**: exists, execution jumps to statement following it
- 4 Otherwise execution leaves **switch()** and proceeds to the following code

A switch () 'Feature'

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Beware: once 2 or 3 above happened, encounter of another **case** or of **default** does not imply exit from **switch**!
- A **break;** statement is needed to this purpose
- This is way too easily forgotten
- Best practices:
 - Always add a **break;** statement at end of each '**case**'
 - Even if it's unreachable, you'll appreciate on code changes
 - Unless you really intend to execute two or more '**cases**' at once

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Let's try it right now!
- Does it work? Good!
- This approach is portable
 - But on UNIXes you'd have a better life using `getopt ()`
- Now we have to implement input from *filename* file

Let's add it to charfreq.c

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```

void parsecmdln(int n, char *args[]) {
    int i = 0;

    while (++i < n) {
        char *p = NULL;
        long th;

        if (args[i][0] != '-') {
            filename = args[i];          // must be filename
            break;                       // ignore anything following
        }

        switch (args[i][1]) {
            case 'l':
                outcase = 'a';
                break;
            case 't':
                th = strtol(args[++i], &p, 10); // numeric argument follows
                if (p == args[i] || th < 0) {
                    fprintf(stderr, "invalid or negative threshold\n");
                    exit(EXIT_FAILURE);
                }
                minoutcount = th;
                break;
            case 'h':
                printUsage();
                exit(0);
                break;
            default:
                illegalopt(args[i]);
        }
    }
}

```

Scientific and Technical Computing in C

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Rome, 3-5 February 2016

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- 1 Pointer Types
- 2 Characters and Strings
- 3 Input and Output
 - Files
 - Text I/O
 - Binary I/O
- 4 Managing Memory
- 5 Conclusions

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- If **myfile** is a **FILE *** variable, open a file using:

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

Simple String I/O

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Beware: if you want to remove item `c` from output in `printf("Parameters: %lf, %lf, %lf\n", a, b, c);`
the following is not enough:
`printf("Parameters: %lf, %lf\n", a, b);`
you need to update the format string too:
`printf("Parameters: %lf, %lf\n", a, b);`
- And on adding an item you have to add a proper conversion specifier
- Ditto for type mismatches: no argument checking is required
- In some cases, dire consequences could follow
- A clever compiler may be able to warn you, if you ask

printf(): Integer Types

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

printf(): Floating Types

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- Conversions
 - **%f**: **float** to base 10 decimal text
 - **%E**: **float** to base 10 exponential text
 - **%G**: most suitable of the above ones
- **l** and **L** are size modifiers
 - Look back at floating types table if you need a refresh
- Variations on a theme
 - **%10f**: at least 10 characters, right justified, space padded
 - **%.4f**: 4 digits after decimal point (**f** and **E** only)
 - **%.7G**: 7 significant digits
 - **%010f**: at least 10 characters, right justified, leading 0s
 - **%-10f**: at least 10 characters, left justified, space padded
 - **%+f**: sign is always printed
 - **% f**: same, but a space if positive
- `printf("%+8.21f %.41E", 12.0, 12.0);`
Prints?

printf(): Characters and Strings

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- **%c**: emits character with specified code
- No variations
- **%s**: emits a string
- Variations on a theme
 - **%10s**: at least 10 characters, right justified, space padded
 - **%.7s**: exactly(!) 7 characters from string
 - **%-10s**: at least 10 characters, left justified, space padded
- **printf("%-7s%4.3s", "Vigna", "Vigna");**
Prints?
- And more conversions are defined, but we'll not cover them

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

scanf () Conversions

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
//...
```

```
printf("Enter t max: ");
```

```
scanf("%lf", &tmax);
```

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

```
int itemsread;  
//...  
do {  
  
    printf("Enter t max: ");  
  
    itemsread = scanf("%lf", &tmax);  
  
} while (itemsread == 0);
```

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
// includes, defines, variable declarations, and function definitions unchanged

int main(int argc, char *argv[]) {
    int i;
    FILE *textfile = stdin;

    parsecmdln(argc, argv);

    if (filename != NULL) {
        textfile = fopen(filename, "r");
        if (!textfile) {
            perror(filename);
            exit(EXIT_FAILURE);
        }
    }

    while ( fgets(s, sizeof(s), textfile) != NULL ) {
        char c, *p = s;

        while ((c=*p++))
            if (isalpha(c))
                ++counts[toupper(c) - 'A'];
    }

    if (filename != NULL)
        fclose(textfile);

    for(i=0; i<LETTERS; ++i)
        if (counts[i] >= minoutcount)
            printf("%c\t%9u\n", i + outcase, counts[i]);

    return 0;
}
```


Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- `fgets ()` is passed `textfile`, initialized to `stdin FILE` pointer
- If no filename was provided on command line, `filename` will still be `NULL`
 - Business as usual
- Otherwise, `filename` will point to filename command line argument string
 - Let's open it
 - Let's fail orderly, if `fopen ()` failed
 - Let's close file as soon as we are done with it
- Let's try it right now!

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
// includes, defines, variable declarations, and function definitions unchanged

int main(int argc, char *argv[]) {
    int i;
    FILE *textfile = stdin;

    parsecmdln(argc, argv);

    if (filename != NULL) {
        textfile = fopen(filename, "r");
        if (!textfile) {
            perror(filename);
            exit(EXIT_FAILURE);
        }
    }

    while ( fgets(s, sizeof(s), textfile) != NULL ) {
        char c, *p = s;

        while ((c=*p++))
            if (isalpha(c))
                ++counts[toupper(c) - 'A'];
    }

    if (filename != NULL)
        fclose(textfile);

    for(i=0; i<LETTERS; ++i)
        if (counts[i] >= minoutcount)
            printf("%c\t%9u\n", i + outcase, counts[i]);

    return 0;
}
```

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Scientific and Technical Computing in C

Stefano Tagliaventi Luca Ferraro

CINECA Roma - SCAI Department

Rome, 3-5 February 2016

Outline

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

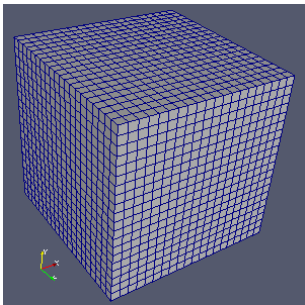
Memory

Allocation
Data Structures

Finale

- 1 Pointer Types
- 2 Characters and Strings
- 3 Input and Output
- 4 Managing Memory
 - Dynamic Memory Allocation
 - Sketchy Ideas on Data Structures
- 5 Conclusions

A PDE Problem



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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

A Rigid Solution

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

Looking for Flexibility

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

A Better Approach

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

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

Slightly More Comfortable, the Old Way

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

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

    // more solver code...
```

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

More Comfortable, Thanks to C99

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
void my_pde_solver(int nx, int ny, int nz,
                  double um[],
                  double vm[],
                  double wm[],
                  double pm[]) {
    double (*u)[ny][nz] = (double (*)[ny][nz])um;
    double (*v)[ny][nz] = (double (*)[ny][nz])vm;
    double (*w)[ny][nz] = (double (*)[ny][nz])wm;
    double (*p)[ny][nz] = (double (*)[ny][nz])pm;

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

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

Removing Limitations

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Enter Dynamic Allocation (from `stdlib.h`)

```
void *malloc(size_t size)
```

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

- `malloc()` allocates a memory area suitable to host a variable whose size is **size**

- Allocated memory is uninitialized.
- Use it like this:

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

- `calloc()` allocates a memory area suitable to host an array of **count** elements, each of size **size**

- Allocated memory is initialized to zero: can be slow, but useful
- Use it like this:

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

The Biggest Mistake

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

Resizing

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

- `realloc()` takes a previously allocated memory area, and gives you a new area whose size is `size`

- Original area contents are copied in the new area, up to `min(oldsiz, size)`
- Use it like this:

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Getting Rid of Memory Areas

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

The First Big Mistake with `free()`

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

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

The Second Big Mistake with `free()`

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

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

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

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

The Third Big Mistake with `free()`

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

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

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

//input and process data

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

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

Memory Friends from `string.h`

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Comfortable, and User Friendly

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

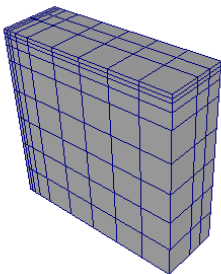
```
void my_pde_solver(int nx, int ny, int nz,
                  // physical parameters
                  ) {
//...
    double (*u)[ny][nz] = (double (*)[ny][nz])calloc(nx*ny*nz, sizeof(double));
    double (*v)[ny][nz] = (double (*)[ny][nz])calloc(nx*ny*nz, sizeof(double));
    double (*w)[ny][nz] =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

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

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

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

Nonuniform Grids



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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Keeping Information Together

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

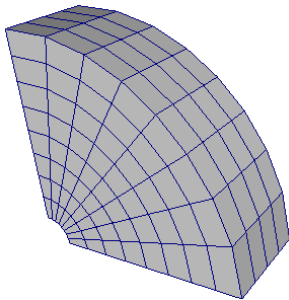
    double *deltax; // Grid steps
    double *deltay;
    double *deltaz;
} nonuniform_grid;
//...
nonuniform_grid my_grid;

//...

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

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

Structured Grids in General Form



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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Sketching a Mesh Description

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
typedef vect3D meshpoint;
typedef vect3D normal;

typedef struct mesh {
    int nx, ny, nz;

    meshpoint *coords;

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

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

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

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

Multiblock Meshes and More

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

If You Read Code Like This...

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

```
struct block_item;

typedef struct block_item {
    block *this_block;

    struct block_item *next;
} block_item;

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

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

And If You Read Code Like This...

```
struct block_tree_node;

typedef struct block_tree_node {
    block *this_block;

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

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

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

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

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

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

Outline

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

- 1 Pointer Types
- 2 Characters and Strings
- 3 Input and Output
- 4 Managing Memory
- 5 Conclusions**

What We Left Out (1 of 2)

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

What We Left Out (2 of 2)

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

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

Looking for More

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale



ANSI WG14

C Standard and Technical Corrigenda

<http://www.open-std.org/jtc1/sc22/wg14/www/standards>

<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>



S. Summit

comp.lang.c Frequently Asked Questions

<http://www.c-faq.com/>



D. Dyer

The Top 10 Ways to get screwed by the "C" programming language

<http://www.andromeda.com/people/ddyer/topten.html>



S. Harbison, G. Steele

C A Reference Manual

Prentice Hall, 5th ed., 2002



A. Kelley, I. Pohl

C by Dissection: The Essentials of C Programming

Addison Wesley, 4th ed., 2000



A. Koenig

C Traps and Pitfalls

Addison Wesley, 1989

Rights & Credits

Pointers

Basics
And Arrays
void

Strings

Chars
Strings
Manipulations
Command Line

I/O

Files
Text
Binary

Memory

Allocation
Data Structures

Finale

These slides are ©CINECA 2016 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Slides and examples were authored by:

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