The C-programming language

Covering

- Compiled versus Interpreted language
- Python, Matlab and C
- Hello World
- Data types, casting
- loops
- conditionals
- character input and output, printf, file i/o
- stdin, stderr, stdout

- arrays
- functions
- strings
- pointers and memory allocation
- execution performance
- Linking C with Python: cython, types, weave
- contigous multidimensional array allocation
- command line arguments

Introduction

- Purpose of this short course: provide *an introduction* to the C programming language
- Will roughly cover chapter 1 of K&R, *"The C Programming Language"*, 2nd edition (1988), and a bit more
- Assume knowledge of either Python or Matlab
- Show how compiled code can be linked with high level code
- Aim to provide sufficient fundamental understanding to enable self learning

Instruct the computer

We use different levels of programming languages. We distinguish three fundamentally different ones:

- machine code (assembler: very low level machine code)
- higher level languages (C, Fortran, typically compiled to machine code)
- very high level language (Python, Matlab, commonly interpreted)

Machine code

- instructions that the Central Processing Unit (CPU) of a computer can understand and execute

- Machine code is *low-level code*. Typical statements are
  - ▷ load a datum (typically a number) from a memory address
  - ▷ add another number to it, increment it by one, . . .
  - ▷ save the datum to (another) memory address
  - ▷ read next instruction from some memory address

- machine code depends on the CPU and the Operating System (OS)
  - ▷ PowerPC machine code cannot be executed on x86 or ARM hardware

- machine code is stored in not-human readable format (binary)
  - ▷ can use *assembler* to read/write machine code
  - ▷ very tedious, only for extreme problems reasonable (if speed must be maximised at all costs)

Interpreted code: Example Matlab

- Source file (the m-file) is *parsed* line by line by the *interpreter* while it is being executed
- The interpreter converts the code into *machine code* which can be executed by the CPU
- For m-files, MATLAB is the interpreter
- The interpretation process is
  - ▷ hard work for the computer,
  - ▷ reason why interpreted languages are slower than compiled languages:
  - ▷ statements in for-loops are (generally) being interpreted again and again
- Interpreted code can be executed on any OS and CPU if the interpreter is available

Compiled code: Example C

- Source file (the c-file) is *compiled* once
- The resulting machine code is stored in a file
- To execute the program, the file containing the machine code is executed
  - ▷ the source file is not required for execution
  - ▷ file with machine code is often called "executable"
  - ▷ Under Windows: executable files end with `.exe`
  - ▷ Under UNIX/Linux: any filename can be executable

Comparison compiled and interpreted languages

- Compiled languages (with static data types) such as C, Fortran, C++:
  - + execute very fast — tool of choice if speed is primary concern
  - ○ don't need source code to execute (there may be commercial interest in not revealing the source code)
  - – more complicated to write, debug and read code
- Interpreted languages (with dynamic data types) such as Python, Matlab
  - + writing code is easier (saves a lot of time)
  - – execution of code is slower (can differ by factor 10-1000 from compiled languages)
  - ○ cannot execute without source code

On performance

- Compiled programs (with static data types) tend to execute much faster than interpreted programs (with dynamic data types)
  - ▷ because machine code is generated in advance (when we compile the source) but not at run-time
  - ▷ and because we know in advance what type of object to expect

C, Matlab and Python

- Assume reader knows at least one programming language
- Assume has either learned Matlab or Python in the past
  - ▷ Opportunity to learn Python: FEEG1001
  - ▷ Opportunity to learn Matlab: SESG1009 (past)
- We compare Matlab/Python code with C throughout this course.
- We briefly look at combining Python and C

Why C?

- Prototyping and a lot of simulation can be done in high-level (interpreted) language (such as Python and Matlab)
- Sometimes, we need very fast code ($\rightarrow$ C, Fortran)
- Parallel execution ($\rightarrow$ C, Fortran): MPI, OpenMP
- Unix, Linux and Mac OS X are written in C
- high portability (ANSI C 90)
- Many languages and tools provide a C-interface
- Fundamental concepts important: should have seen C at least once.

Hello World

It is tradition to write a “Hello World” program when learning a new language.

- Matlab:

matlab/hello.m
```
fprintf('Hello World\n')
```

- Python:

python/hello2.py
```
print("Hello World")
```

Hello World (in C)

Our first C program:

c/hello.c

```
1 #include<stdio.h>
2
3
4 int main( void ) {
5
6   printf("Hello World!\n");
7
8   return 0;
9 }
```

- Line 1 includes the STandDard Input Output (stdio) *header* file
  - ▷ provides commands such as `printf`
  - ▷ we will usually need (at least) this header file

- In line 4, the “main program” starts
  - ▷ the main program is a function and returns an integer (therefore `int`)
  - ▷ the main function has to be called “main” (therefore `main`)
  - ▷ the main function does not take arguments (therefore `void`)
  - ▷ the main function body starts after the opening curly brace ({)

- Line 6 is a print statement (very similar to MATLAB’s `fprintf`) but use double quotes to enclose format string

- Line 8 returns 0 to the operating system to indicate that the program executed and finished correctly (a convention).

- Line 9 ends the main function with the closing curly brace (})

- *Every* statement (such as `printf` and `return`) has to be followed by a semicolon (`;`)

- For now, use lines 1–4 and 8–9 as a template (*i.e.* just copy them) for your C programs.

Compiling `hello.c`

- Compilation depends on the operating system and environment

- If command line compiler available (for example on UNIX/Linux/Mac OS X/MinGW):
  - ▷ Compile `hello.c` into executable `hello` using compiler `gcc`
    `gcc -o hello hello.c`
  - ▷ `cc` is the C-compiler
  - ▷ `-o hello` states that the output file should be called `hello`
  - ▷ `hello.c` is the name of the input file (typically the last argument)
  - ▷ many possible compilers available, including `gcc`, Portland Group Compiler, . . . but all have similar syntax (as shown above)
  - ▷ For gcc, we recommend switches `-ansi -pedantic -Wall`, i.e.:
    `gcc -ansi -pedantic -Wall -o hello hello.c`

- A good C-compiler is GNU's `gcc` (see www.gnu.org for details). Can be used on UNIX/Linux, Windows and many other platforms.

- A simple and free graphical user interface and editor for GCC (on Windows) is Quincy $\rightarrow$ laboratories

- On Linux/Mac, use editor that supports C (Emacs, ...) and command prompt to call `gcc`.

Execution of `hello`

- On UNIX/Linux
  - ▷ `./hello`
- On Windows
  - ▷ Either find `hello.exe` and type `hello.exe` in MS-DOS prompt
  - ▷ or click on appropriate button to execute ($\rightarrow$ labs)
- Output is

```
Hello World!
```

Hello World (in B)

The first ever "Hello World" program is believed to have been written for the programming language B:

b/hello.b

```
main( ) {
  extrn a, b, c;
  putchar(a); putchar(b); putchar(c); putchar('!*n');
}
a 'hell';
b 'o, w';
c 'orld';
```

Source: http://en.wikipedia.org/wiki/Hello_world_program (accessed 1/10/2010)

ooo

→ **lab 1**

Absence of restriction

Central design ideas of the C programming language were *economy of expression* and *absence of restriction*.

This requires some discipline from the programmer. How useful are these hello world programs?

c/hello3.c

```
#include<stdio.h>
main() {printf("Hello World!\n");}
```

c/hello4.c

```
#include<stdio.h>
int main(
int argc, char* argv[]
) {printf("Hello World!\n")
;return 0;}
```

Summary Hello World

- Write C-program in so-called source file (`hello.c`)
  - ▷ looks similar to m-file, py-file, etc
- Compile source to create executable file
  - ▷ Can choose name of executable, usually same root as source file but no extension or `.exe`, (*i.e.* `hello` or `hello.exe`)
  - ▷ executable is binary file with machine code
- Run program by executing executable
  - ▷ Don't need source code any more for execution of executable
  - ▷ Don't need C compiler for execution of executable

Execution comparison with Matlab

- Comparison with MATLAB
  - ▷ Could have achieved the same Hello World greeting in MATLAB with

matlab/hello.m

```
fprintf('Hello World\n')
```

```
[fangohr@lyceum ~]$ matlab -nojvm -nodisplay -r hello2

                              < M A T L A B (R) >
                    Copyright 1984-2008 The MathWorks, Inc.
                           Version 7.7.0.471 (R2008b)
                               September 17, 2008


  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

Hello World
```

  - ▷ Matlab executes the m-file given after the `-r` switch.

Execution comparison with Python

- Python Hello World program

python/hello2.py

```
print("Hello World")
```

- needs the Python interpreter to be executed:

```
[fangohr@lyceum ~]$ python hello.py
Hello World
```

- Python executes the argument (`hello.py`) given to the intepreter program (`python`).

Data types

C knows 4 basic data types:

- `char` (character): letters, *i.e.* `'a'`, `'b'`, . . .
- `int` (integer): integers, *i.e.* 1, 10, -344, 10000, . . .
- `float` (floating point number): *i.e.* 1.1, -10.34, 1.41,. . .
- `double` (more accurate floating point number)

Chars and ints can be signed (default) or unsigned. ints can be short or long. The actual number of bits used for short int, int, long int, float, double, and long double is implementation dependent (see standard headers `<limits.h>` and `<float.h>`.)

Whenever we want to use a variable in C, we have to say ("declare") in advance of which type it is!

For example:

```
int a;        /* declares an integer variable with name a */
double f;     /* ... a double float with name f */
int i=0;      /* declares the variable i to be of type int and
                 initialises it with the value 0 */
```

Useful to combine *declaration* of variable (for example `int i;` with the *initialisation* of the variable (for example `int i=0;`. This avoids accidental use of `i` with an undefined (i.e. random) value.

Using variables in C

- C-Code

c/var1.c

```c
#include<stdio.h>
int main( void ) {
  int a, b, c;     /* define variables first:
                      a, b and c are of type int */
  a=10;            /* assign values */
  b=20;
  c = a + b;       /* do computation */
  printf("The sum of %d and %d is %d.\n",a,b,c);
  return 0;
}
```

- Output

```
The sum of 10 and 20 is 30.
```

- Comments
  - ▷ text enclosed by /* and */ is ignored by the compiler
  - ▷ The latest ANSI C (and C++) allows to use // to ignore the rest of the line (like MATLAB 's % and Python's #) but this is not part of the ANSI C90 standard (and the switch `-ansi` will prevent usage of //).

- Equivalent programs in Python and Matlab

python/var1.py

```
a = 10
b = 20
c = a + b
print("The sum of %d and %d is %d." % (a, b, c))
```

matlab/var1.m

```
a=int32(10);  %int32() to get an integer: by default
b=int32(20);  %numbers are doubles in Matlab
c=a+b;
fprintf('The sum of %d and %d is %d.\n',a,b,c)
```

When we forget to declare variables . . .

- c/var1wrong1.c

```c
#include<stdio.h>
int main( void ) {
  a=10;                /* assign values */
  b=20;
  c = a + b;           /* do computation */
  printf("The sum of %d and %d is %d.\n",a,b,c);
  return 0;
}
```

- Output from gcc `-ansi -pedantic -Wall -o var1wrong1 var1wrong1.c`

```
var1wrong1.c:3:3: error: use of undeclared identifier 'a'
  a=10;           /* assign values */
  ^
var1wrong1.c:4:3: error: use of undeclared identifier 'b'
  b=20;
  ^
```

```
var1wrong1.c:5:3: error: use of undeclared identifier 'c'
  c = a + b;     /* do computation */
  ^
var1wrong1.c:5:7: error: use of undeclared identifier 'a'
  c = a + b;     /* do computation */
      ^
var1wrong1.c:5:11: error: use of undeclared identifier 'b'
  c = a + b;     /* do computation */
          ^
var1wrong1.c:6:42: error: use of undeclared identifier 'a'
  printf("The sum of %d and %d is %d.\n",a,b,c);
                                         ^
var1wrong1.c:6:44: error: use of undeclared identifier 'b'
  printf("The sum of %d and %d is %d.\n",a,b,c);
                                           ^
var1wrong1.c:6:46: error: use of undeclared identifier 'c'
  printf("The sum of %d and %d is %d.\n",a,b,c);
                                             ^
8 errors generated.
```

When we forget a semicolon . . .

- c/var1wrong2.c

```c
#include<stdio.h>
int main( void ) {
  int a, b, c;    /* define variables first:
                     a, b and c are of type int */
  a=10            /* assign values */
  b=20;
  c = a + b;      /* do computation */
  printf("The sum of %d and %d is %d.\n",a,b,c);
  return 0;
}
```

- Output from gcc `-ansi -pedantic -Wall -o var1wrong2 var1wrong2.c`

```
var1wrong2.c:5:7: error: expected ';' after expression
  a=10            /* assign values */
      ^
      ;
1 error generated.
```

Summary Compilation

- Writing compiled code is not easy
  - ▷ have to get everything right, *i.e.*
  - ▷ declare variables before using them
  - ▷ stick to strict syntax (don't forget semicolon, parenthesis, braces, . . .)
  - ▷ otherwise the compiler will issue an error message and stop
- Compiler checks syntax of code at compile time

Documentation

- It is similarly important to document C-code as it is for any other code. For example

c/hellodoc.c

```c
/* hello.c - a program printing "Hello World!" to the screen
Hans Fangohr, March 2003, University of Southampton, fangohr@soton.ac.uk

History: - inspired by standard C-greeting program
Purpose: - to provide students a small example of working C-code
Modifications:
                Sept 2010: minor changes for SESG6025
*/
#include<stdio.h>
int main( void ) {          // The latest ANSI C (>1999) allows one-line
                            // comments starting with a double slash
                            // but best not to rely on this.
  printf("Hello World!\n");
  return 0;
}
```

- Syntax for comments:
  - ▷ Enclose (long) comments between `/*` and `*/`
  - ▷ New feature: Use `//` to mark end of line as a comment
    - ∗ This was introduced in C++, and
    - ∗ later (around 1998) made its way back into ANSI C, but not part of ANSI C90 and gcc's switch `-ansi` prevents usage of `//`.
    - ∗ Best avoided to maximise portability of our code.

- Suggestions for comments:
  - ▷ Do not explain what is visible anyway:

    ```
    s = sta + stu;   /* computing the sum of sta and stu */
    ```

    instead provide the context or reason for the operation, for example

    ```
    s = sta + stu;   /* total number s of STAff and STUdents */
    ```
  - ▷ if a piece of code is self-explaining, there is no need to add comments.

- Indentation:
  - ▷ Commonly, deeper levels of functions/blocks of statemens/etc are indented according to their depth.

- However, C does not require this (as commands are grouped through the use of curly braces), so it is the programmers responsibility to indent code correctly (most editors will help).
- A lot of time has been wasted through not correctly indented code.

printf

- the `printf` function is used to PRINT Formatted values to the standard output (`stdout`).
- Same principle as in Matlab/Python: a string with format specifiers is followed by a list of values, such as

```
printf("%d bottles of milk at %4.2f each cost %5.2f in total",5,1.42,5*1.42);
```

  with output

```
5 bottles of milk at 1.42 each cost  7.10 in total.
```

- Format specifiers are
  - ▷ `d` and `i` for *signed integers, short, chars* (int, char, short)
  - ▷ `ld` and `li` for *signed long integers* (long)

- ▷ `f` for *floating point representation of double* (double)
- ▷ `f` for *floating point representation of float* (float)
- ▷ `e` for *exponential notation of float and double* (double,float)
- ▷ `g` for *exponential notation or floating point representation: the shorter one is chosen* (double,float). Good general purpose choice for floating point
- ▷ `s` for *arrays of characters* (char *, char[])
- ▷ `u` for *unsigned integers* (unsigned int, short, char)

- The `printf` function is part of the standard library and we must include `<stdio.h>` in the beginning of the file.

- Related commands `fprintf` and `sprintf` to print into file stream and string, respectively.

- Choosing the wrong format specifier results in undefined behaviour.

Example 1

c/printf1.c

```c
#include<stdio.h>
int main(void){
  int var1=-42;
  unsigned int var2 =142;
  float var3=3.14;
  double var4 = 3.1428;
  char var5[] = "Hello World";
  printf("Printing int var1=%d\n",var1);
  printf("Printing unsigned int var2=%u\n",var2);
  printf("Printing float var3=%f\n",var3);
  printf("Printing double var4=%f\n",var4);
  printf("Printing double in scientific notation var4=%e\n",var4);
  printf("Printing char[] var5=%s\n",var5);
  return 0;
}
```

Output:

```
Printing int var1=-42
Printing unsigned int var2=142
Printing float var3=3.140000
Printing double var4=3.142800
Printing double in scientific notation var4=3.142800e+00
Printing char[] var5=Hello World
```

Example 2

c/printf2.c

```c
#include<stdio.h>
int main(void){
  double pi = 3.141592653589793l;
  printf("as a standard float representation: pi=%f\n",pi);
  printf("in exponential notation: pi=%e\n",pi);
  printf("whichever of the above two is shorter: pi=%g\n",pi);
  printf("Request 10 digits overall: pi=%10f\n",pi);
  printf("With 3 postdecimal digits: pi=%10.3f\n",pi);
  printf("12 postdecimal digits: pi=%.12f\n",pi);
  return 0;
}
```

Output:

```
as a standard float representation: pi=3.141593
in exponential notation: pi=3.141593e+00
whichever of the above two is shorter: pi=3.14159
Request 10 digits overall: pi=  3.141593
```

```
With 3 postdecimal digits: pi=      3.142
12 postdecimal digits: pi=3.141592653590
```

Another printf example

c/printf0.c

```c
#include<stdio.h>
int main(void){
  double price=1.42;
  int n=5;
  double total=0.;
  total = n*price;
  printf("%d bottles of milk at %4.2f "\
         "each cost %5.2f in total.",
         n,price,total);
  return 0;
}
```

c/printf0.out

```
5 bottles of milk at 1.42 each cost  7.10 in total.
```

python/printf0.py

```python
price = 1.42
n = 5
total = n * price
print("%d bottles of milk at %4.2f "\
      "each cost %5.2f in total." \
      % (n, price, total))
```

matlab/printf0.m

```matlab
price=1.42;
n=5;
total=n*price;
fprintf('%d bottles of milk at %4.2f ',n,price);
fprintf('each cost %5.2f in total.\n',total);
```

scanf()

- This is a useful function for enabling user input to a program. You may not fully understand the use of it until future lectures but for now it can be used as a template to enable more interesting programs.
- The function scanf() is very similar to printf() and uses the same formatting rules but in the other direction.

- The basic form is as seen here:

c/scanf0.c

```c
#include<stdio.h>

int main(void)
{
        int answer;
        scanf("%d",&answer);
        printf("You entered %d.\n",answer);
        return 0;
}
```

- the integer variable answer is declared prior to the use of scanf. The same %d format code is used within the quotation marks as for printf when using an integer. The variable that input is being assigned to is the second argument and needs the & before it.

- When using scanf to accept user input in the form of char values, it may be necessary to use the form:

c/scanf1.c

```c
#include<stdio.h>

int main(void)
{
        char answer;
        scanf(" %c",&answer);
        printf("You entered %c.\n",answer);
        return 0;
}
```

the space before the % allows any white space still in the input buffer to be ignored before the most recent user input is retrieved.

The for-loop in C

- Example: printing numbers i=0 to i=9. Desired output

```
0  1  2  3  4  5  6  7  8  9
```

- In Matlab

matlab/forloop1.m

```
for i=0:9
  fprintf('%d ',i);
end
```

- In Python (using print in C-style)

python/forloop1.py

```
for i in range(10):
    print("%d" % i),
```

- In C:

c/forloop1.c

```c
#include<stdio.h>
int main( void ) {
  int i;
  for (i=0; i<10; i = i + 1) {
    printf("%d ",i);
  }
  return 0;
}
```

- `for`-command has 4 parts:

```
for ( A ; B ; C ) {
  D}
```

  ▷ A: initial assignment (executed once)
  ▷ B: logical expression, for-loop continues while this expression is true
  ▷ C: assignment to be executed *after* every loop
  ▷ D: block of statements to be executed in every loop (limited by enclosing curly braces)

- Improving on our programming style: make loop more general

  ▷ count from 0 to $N-1$

c/forloop1a.c

```c
#include<stdio.h>
int main( void ) {
  int i;
  int N;
  N=10;
  for (i=0; i<N; i=i+1) {
    printf("%d\n",i);
  }
  return 0;
}
```

▷ can *initialise* variable when *declaring* it: combine lines 4 and 5 (nothing do to with for-loop):

c/forloop1c.c

```
1 #include<stdio.h>
2 int main( void ) {
3   int i;
4   int N = 10;
5   for (i=0; i<N; i=i+1) {
6     printf("%d ",i);
7   }
8   return 0;
9 }
```

- Use C-typical notation:
  - ▷ `i = i + 1;` can be written as `++i;` (or `i++;`) using the "increment operator"

c/forloop1b.c

```c
#include<stdio.h>
int main( void ) {
  int i;
  int N = 10;
  for (i=0; i<N; ++i) {
    printf("%d ",i);
  }
  return 0;
}
```

ooo

→ lab 2

The increment operator (i=i+1)

- `i = i + 1` can be written as `i++` or `++i`

c/increment0.c

```c
#include<stdio.h>
int main(void) {
  int k=5,n=0;
  n = k;
  k = k + 1;   /* could also write 'k++;' or '++k' */
  printf("case 1: -> k=%d, n=%d\n", k, n);
  k = 5; n=0; /* reset to original values */
  k = k + 1;   /* could also write 'k++;' or '++k' */
  n = k;
  printf("case 2: -> k=%d, n=%d\n", k, n);
  return 0;}
```

Output:

```
case 1: -> k=6, n=5
case 2: -> k=6, n=6
```

- The `++` operator can be used *within* other statements (such as – in this example – assignments), and then it matters whether prefix notation (`++i`) or postfix notation (`i++`) is being used:

c/increment.c

```c
#include<stdio.h>
int main(void) {
  int k=5,n=0;
  n = k++;      /* instead of: n=k; k=k+1 */
  printf("postfix: n=k++ -> k=%d, n=%d\n", k, n);
  k = 5; n=0;
  n = ++k;     /* instead of: k=k+1; n=k */
  printf("prefix : n=++k -> k=%d, n=%d\n", k, n);
  return 0;
}
```

Output:

```
postfix: n=k++ -> k=6, n=5
prefix : n=++k -> k=6, n=6
```

Other concise expressions

- ▷ `i--` and `--i` are postfix and prefix decrements
- ▷ `i += c` $\leftrightarrow$ `i = i + c`
- ▷ `i -= c` $\leftrightarrow$ `i = i - c`
- ▷ `i *= c` $\leftrightarrow$ `i = i * c`
- ▷ `i /= c` $\leftrightarrow$ `i = i / c`

Legal but not soo nice (example for-loop)

- c/forloop1d.c

```c
#include<stdio.h>
int main( void ) {
  int i;
  int N = 10;
  for (i=0; i<N; printf("%d ",i++)) {
    ;
  }
  return 0;
}
```

c/forloop1e.c

```c
#include<stdio.h>
int main(void){int i;int N=10; for(i=0;i
<N;printf("%d ",i++)){;}return 0;}
```

Tabulating $\sqrt{x}$

A more complex example: computing a table of square roots as shown below

```
sqrt(1.000000) = 1.000000
sqrt(1.500000) = 1.224745
sqrt(2.000000) = 1.414214
sqrt(2.500000) = 1.581139
sqrt(3.000000) = 1.732051
sqrt(3.500000) = 1.870829
sqrt(4.000000) = 2.000000
sqrt(4.500000) = 2.121320
sqrt(5.000000) = 2.236068
sqrt(5.500000) = 2.345208
sqrt(6.000000) = 2.449490
sqrt(6.500000) = 2.549510
sqrt(7.000000) = 2.645751
sqrt(7.500000) = 2.738613
sqrt(8.000000) = 2.828427
sqrt(8.500000) = 2.915476
sqrt(9.000000) = 3.000000
sqrt(9.500000) = 3.082207
sqrt(10.000000) = 3.162278
```

can be generated using

c/forloop2.c

```c
#include<stdio.h>
#include<math.h>        /* include math-header for sqrt-function */

int main( void ) {
  /* print table of N square roots sqrt(x) for x in [a,b] */

  int N = 19;           /* total number of lines in table */
  double a=1;           /* starting with x=a */
  double b=10;          /* ending with x = b */
  double x;             /* being used in for-loop */
  double y;
  int i;                /* iteration counter for for-loop */

  for (i=0; i<N; i++) {
    x = a + i*(b-a)/(N-1);   /* compute x */
    y = sqrt(x);
    /* compute sqrt(x) and print result */
    printf(" sqrt(%f) = %f\n", x, y);
  }

  return 0;
}
```

In Matlab/Python (can be done even shorter in both)

matlab/forloop2.m

```
%print table of N square roots sqrt(x) for x in [a,b]
N=19;
a=1;
b=10;
for i=0:N-1
    x = a + i*(b-a)/(N-1.);
    fprintf('sqrt(%f) = %f\n',x,sqrt(x));
end
```

python/forloop2.py

```
"""print table of N square roots sqrt(x) for x in [a,b]"""
from math import sqrt
N = 19
a = 1
b = 10
for i in range(N):
    x = a + i * (b - a) / (N - 1.)
    print("sqrt(%f) = %f" % (x, sqrt(x)))
```

Symbolic constants

(K&R, Sec 1.4, page 14)

If we use constants in a program, it is good practice to define a *symbolic name* or *symbolic constant*:

`#define` *name replacement-text*

The pre-processor (runs silently before the actual compilation) will replace all occurrences of *name* in the file with *replacement-text*.

We can use `#define` to define global constants:

c/forloop3.c

```c
#include<stdio.h>
#include<math.h>        /* include math-header for sqrt-function */

#define N 19            /* total number of lines in table */
#define a 1.0           /* starting with x=a */
#define b 10.0          /* ending with x = b */

int main( void ) {
  /* print table of N square roots sqrt(x) for x in [a,b] */

  double x;             /* being used in for-loop */
  int i;                /* iteration counter for for-loop */

  for (i=0; i<N; i++) {
    x = a + i*(b-a)/(N-1);   /* compute x */

    /* compute sqrt(x) and print result */
    printf(" sqrt(%f) = %f\n", x, sqrt(x) );
  }

  return 0;
}
```

Output: [as on slide 54]

More #define examples on slide 93.

The while loop in C

c/while.c

```c
#include<stdio.h>
int main( void ) {
  double x = 1;
  while (x < 50) {
    printf("In this iteration, x=%f\n",x);
    x = x * 1.5;
  }
  return 0;
}
```

The `while`-loop has two parts:

```
while ( A ) {
  B
}
```

- A: logical expression, while-loop continues while this expression is true
- B: block of statements to be executed in every loop (limited by enclosing curly braces)

The for-loop and the while loop are closely related in C. In fact, this for-loop

```
for ( A; B; C ) {  D;   }
```

is equivalent to this while-loop:

```
A;
while( B ) { D; C; }
```

Output:

```
In this iteration, x=1.000000
In this iteration, x=1.500000
In this iteration, x=2.250000
In this iteration, x=3.375000
In this iteration, x=5.062500
In this iteration, x=7.593750
In this iteration, x=11.390625
In this iteration, x=17.085938
In this iteration, x=25.628906
In this iteration, x=38.443359
```

Python:

python/while.py

```python
x = 1
while x < 50:
    print("In this iteration, x=%f" % x)
    x = x * 1.5
```

Matlab:

matlab/while.m

```matlab
x=1;
while x < 50
    fprintf("In this iteration, x=%f\n",x);
    x = x * 1.5;
end
```

If-then statements

- MATLAB

matlab/ifthen1.m

```
a=10;
b=20;
if a==b
    fprintf('a is the same as b\n')
else
    fprintf('a is not equal to b\n')
end
```

- Python

python/ifthen1.py

```
a = 10
b = 20
if a == b:
    print('a is the same as b')
else:
    print('a is not equal to b')
```

- C

c/ifthen1.c

```
 1 #include<stdio.h>
 2 int main( void ) {
 3   int a=10;
 4   int b=20;
 5   if (a==b) {
 6     printf("a is the same as b\n");
 7   }
 8   else {
 9     printf("a is not equal to b\n");
10   }
11   return 0;
12 }
```

In C,

- the logical expression (`a==b`) `has` to placed in parenthesis in `if`-statement (5)

- block of statements to be executed (6) if `a==b` is true has[1] to be enclosed by

[1]This is not strictly true, but we strongly advise you to do this. If there are no curly braces, then only the first

curly braces (5, 7)

- block of statements for the `else`-part (9) has to be enclosed in curly braces (8, 10)

statement is considered to be a part of the `if`-statement. Using curly braces is the safe way.

Grouping

The grouping of commands to form a block is done

- in Matlab: through *some keyword* (such as `for`, `if`, `else`, `while`) followed by the keyword `end`.
- in C: through enclosing curly braces { and }.
- in Python: through *some keyword* (such as for, if, else, while, def) followed by a colon (`:`) and indenting all commands in the block to the same level.

ooo

→ lab 3

Integer division truncates

Integer division truncates:

c/intdiv.c

```
#include<stdio.h>

int main( void ) {
  int a=10; int b=3; double c;

  c = b/a;

  printf("a=%d, b=%d, c=%f\n", a, b, c);

  return 0;
}
```

The output is:

```
a=10, b=3, c=0.000000
```

Solution: “cast” a or b into a double (line 6):

c/intdiv1a.c

```
 1 #include<stdio.h>
 2 
 3 int main( void ) {
 4   int a=10; int b=3; double c;
 5 
 6   c = (double) b/a;      /* OR  c = b/(double) a;  */
 7 
 8   printf("a=%d, b=%d, c=%f\n", a, b, c);
 9 
10   return 0;
11 }
```

The output is

```
a=10, b=3, c=0.300000
```

Integer division truncates: avoid with “Casting”

- Watch out for intermediate integer calculations

c/intdiv2.c

```c
#include<stdio.h>

int main( void ) {  /* convert from fahrenheit to celsius */

  double celsius;
  double fahrenheit;
  fahrenheit = 82;

  celsius = (fahrenheit-32)*(5/9);

  printf("%f degree Fahrenheit are %f degree Celsius.\n",
         fahrenheit, celsius);

  return 0;
}
```

Output:

```
82.000000 degree Fahrenheit are 0.000000 degree Celsius.
```

- Reason: `(5/9)` is integer operation, thus it is truncated, thus `(5/9)` evaluates to zero! (line 9)

- Solution: modify 5 or 9 (or both) to be floating point numbers
  - ▷ `celsius = (fahrenheit-32)*(5.0/9);`
  - ▷ `celsius = (fahrenheit-32)*(5/9.0);`
  - ▷ `celsius = (fahrenheit-32)*((double) 5/9);`
  - ▷ `celsius = (fahrenheit-32)*(5/(double) 9);`
  - ▷ `celsius = 5*(fahrenheit-32)/9);`

Numbers (in mathematics)

- Natural numbers:

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$$

- Integer numbers:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

- Rational numbers : (all fractional numbers, fraction = ratio)

$$\mathbb{Q} = \left\{\frac{m}{n} : m \in \mathbb{Z}, n \in \mathbb{Z}^*\right\}$$

- Real numbers (irrational numbers):

$$\mathbb{R} = \mathbb{Q} \cup \{\text{“all other scalars”}\}$$

Examples for $x \in \mathbb{R}$: $x = \sqrt{2}$, $x = \pi$, $x = \exp(1) = e$

- (for completeness): Complex numbers:

$$\mathbb{C} = \{a + ib : a \in \mathbb{R}, b \in \mathbb{R}\} \quad \text{and} \quad i = \sqrt{-1}$$

- Formally:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$$

Memory: bits and bytes

- Computers store information in "bits"
- a bit is either 1 or 0
  - ▷ RAM: low voltage or high voltage
  - ▷ Hard disk: magnetisation pointing up or down
- eight bits are one byte
- 1024 $\approx 10^3$ byte are one kilo byte (kb)
- $1024^2 \approx 10^6$ byte are one mega byte (Mb)
- $1024^3 \approx 10^9$ byte are one giga byte (Gb)

Natural and integer numbers

How can we represent integer numbers in a computer using bits?

- Suppose we have one byte (i.e. eight bit):

natural number	bit-representation
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
⋮	⋮
254	11111110
255	11111111

- with 8 bit we can represent 256 natural numbers (for example from 0 to 255 as in the `unsigned char` type) – $2^8 = 256$.

- with 8 bit we can also represent 256 integer numbers (for example from -128 to $+127$ as in the (signed) `char` type)

Example: determine largest `char` integer

c/maxint1.c

```c
#include <stdio.h>
int main(void) {
        char a = 120;  /* educated guess just below
                          the max char */
        while (a > 0) {
                printf("%d ", a);
                a = a + 1;
        }

        printf("%d ", a);
        return 0;
}
```

Output:

```
120 121 122 123 124 125 126 127 -128
```

(Numeric) data types

The numeric data types of C are (K&R, page 9)

- `char`: character - a single byte
- `short`: short integer
- `int`: integer (16 bits [between -32768 and +32767] or 32 bits [between -2147483648 and 2147483647])
- `long`: long integer
- `float`: single precision floating point (typically 32 bit with at least 6 significant digits and magnitude between 10^{-38} and 10^{38}
- `double` double-precision floating point (typically 2x32 bit with at least 15 significant digits and magnitude between 10^{-308} and 10^{308})

Guidelines:

1. Use `double` whenever you need floating point numbers

2. Do not use `float` (unless you know exactly what you are doing)

   - `float` is less accurate than `double`
   - with today's PCs (great CPU-power and memory) there is to need to use `float`s anymore: computation with floats is generally not faster that computation with doubles.
   - the main reason for using `float`s instead of `double`s at all is (usually) to reduce memory requirements

   Note: when people speak about "floats" they probably mean doubles – terminology varies from language to language. Python, for example, has only one data type "float" which is in C-terms a double.

Finding actual limits of numbers for given compiler/OS

c/limits.c

```

#include <limits.h>  /* limits for integers */
#include <float.h>   /* limits for floats */
#include <stdio.h>

int main(void) {
          printf("    CHAR_MIN = %12d\n", CHAR_MIN);
          printf("    CHAR_MAX = %12d\n", CHAR_MAX);
          printf("    SHRT_MIN = %12d\n", SHRT_MIN);
          printf("    SHRT_MAX = %12d\n", SHRT_MAX);
          printf("     INT_MIN = %12d\n", INT_MIN);
          printf("     INT_MAX = %12d\n", INT_MAX);
          printf("    LONG_MIN = %12ld\n", LONG_MIN);
          printf("    LONG_MAX = %12ld\n", LONG_MAX);
          printf("     FLT_MIN = %12e\n", FLT_MIN);
```

```
           printf("      FLT_MAX = %12e\n", FLT_MAX);
           printf("      DBL_MIN = %12e\n", DBL_MIN);
           printf("      DBL_MAX = %12e\n", DBL_MAX);
return 0;
        }
```

Output:

```
    CHAR_MIN =          -128
    CHAR_MAX =           127
    SHRT_MIN =        -32768
    SHRT_MAX =         32767
     INT_MIN =   -2147483648
     INT_MAX =    2147483647
    LONG_MIN = -9223372036854775808
    LONG_MAX = 9223372036854775807
     FLT_MIN = 1.175494e-38
     FLT_MAX = 3.402823e+38
     DBL_MIN = 2.225074e-308
     DBL_MAX = 1.797693e+308
```

Floating point numbers ("floats" and "double")

- Any real number x (thus also integers) can be written as

$$x = a \cdot 10^b$$

where a is the mantissa and b the exponent.

- Examples:

x		a	b
123.456	$= 1.23456 \cdot 10^2$	1.23456	2
100000	$= 1.0 \cdot 10^6$	1.00000	6
0.0000024	$= 2.4 \cdot 10^{-6}$	2.40000	-6
Can write a and b as integers:			
		123456	2
		100000	6
		240000	-6

- $\rightarrow$ we can use 2 integers to code one floating point number

$$x = a \cdot 10^b$$

- For example: x is an 8 byte float: these 64 bits are split as
  - ▷ 10 bit for the exponent b and
  - ▷ 54 bit for the mantissa a.

  This results in

  - ▷ largest possible float $\approx 10^{308}$
  - ▷ smallest possible float $\approx 10^{-308}$ (quality measure for b)
  - ▷ distance between 1.0 and next larger number $\approx 10^{-16}$ (quality measure for a)

- This is *in principle* how floating point numbers are represented.

Numbers in computer

- Mostly, we use *approximations* to the real number, i.e. $\pi \approx 3.14159265358979$.
- Often, this doesn't matter
- but sometimes it does:

```
>>sin(pi)
ans =
     1.2246e-16
```

- Even “simple floating point numbers“ cannot be represented exactly (using Python to demonstrate but works the same in C and Matlab):

```
>>> "%.25f" % 0.1
'0.1000000000000000055511151'
```

```
>>> "%.25f" % 0.2
'0.2000000000000000111022302'
>>> "%.25f" % 0.3
'0.2999999999999999888977698'
>>> "%.25f" % 0.4
'0.4000000000000000222044605'
>>> "%.25f" % 0.5
'0.5000000000000000000000000'
```

- Be aware that
  - ▷ there is an upper limit to possible floats
  - ▷ there is a lower limit to possible floating point numbers
  - ▷ the next largest number to 1 is `1+eps`
  - ▷ see `float.h`
- This is an import issue in computation
  - ▷ we'd like to know *how* inaccurate our results are,
  - ▷ we look for *stable* numerical methods which don't amplify small errors.

Example program to determine `eps`

eps is the 'distance' between 1.0 and the next larger number (i.e. 1.0+eps) that can be represented:

c/eps.c

```c
#include<stdio.h>
int main(void) {
  double eps = 1.0;
  while (1.0+eps>1.0) {
    eps /= 2.0;
  }
  printf("eps is approximately %g.\n", eps);
  return 0;
}
```

```
eps is approximately 1.11022e-16.
```

Functions

c/hello2.c

```
1 #include<stdio.h>
2 int main( void ) {
3   printf("Hello World!\n");
4   return 0;
5 }
```

- C programs consist only of functions
- there has to be exactly one "main"-function with name `main` (this is executed first, and may call other functions)
- have to define what argument(s) each function takes (line 2)
- have to define what type these arguments are (different from Matlab and Python): in example above (line 2):
  - ▷ main takes no arguments (`void`) and
  - ▷ main returns an `int` (line 4)

Functions, example 1

c/func1.c

```
 1 #include<stdio.h>
 2 
 3 int square(int a) {
 4   return a*a;
 5 }
 6 
 7 int main(void) {
 8   int mynumber;                /* declare variable    */
 9   mynumber = 4;                /* initialise variable */
10 
11   printf("mynumber is %d, and mynumber squared is %d.\n",
12          mynumber, square(mynumber) );
13 
14   return 0;                    /* return 0 to operating system */
15 }
```

Two functions: `main` (lines 7-15) and `square` (lines 3-5)

Output:

```
mynumber is 4, and mynumber squared is 16.
```

In summary, a function definition has this form (K&R, p25):

```
return-type function-name(parameter declaration, if any)
{
        declarations
        statements
}
```

Functions, prototypes K&R, 1.7, p24

c/powers.c

```c
#include<stdio.h>

int power(int m, int n);                /* function prototype */

/* main(): demo power function */
int main(void) {
  int i;
  for (i=0; i<6; ++i) {
    printf("%2d %6d %6d\n", i, power(2,i), power(-3,i));
  }
  return 0;
}

/* power: raise base to the n-th power; n >= 0 */
int power(int base, int n) {
  int i, p;
  p = 1;
  for (i=1; i<=n; ++i) {
    p = p*base;
  }
  return p;
}
```

Output:

```
0       1       1
1       2      -3
2       4       9
3       8     -27
4      16      81
5      32    -243
```

- Functions definitions need to be given before the function can be called.
- Instead of the full function definition, we can provide a *function prototype* which provides
  - ▷ the name of the function (`powers`)
  - ▷ the type of the function parameters (`int, int`)
  - ▷ the type of the return value (`int`).

  The actual function definition can follow then at any point in the file.

It is convention to have the `main` function as the first (or last) function definition in the file.

Some terminology: function's parameters and arguments

c/snippet–only/powers_func.c

```c
/* power: raise base to the n-th power; n >= 0 */
int power(int base, int n) {
  int i,p;
  p = 1;
  for (i=1; i<=n; ++i) {
    p = p*base;
  }
  return p;
}
```

- K&R refer to the the function's *parameters* when they talk about the *type* of the variables "base" and "n".
- K&R like to refer to "base" and "n" as the function's *arguments* when talking about the actual values (that are passed to "base" and "n" in this case when the function is called).

Other names used: formal arguments (for parameters), actual argument(for arguments), or – less specific – input arguments, arguments, input.

Arguments – call by value

Arguments in C are passed *by value*: the called function is given the values of its arguments in *temporary variables* rather than the originals.

→ function can safely modify argument values without causing side effects

BUT, this is not true for (i) arrays and (ii) pointers.

Example:

c/passbyvalue.c

```
#include<stdio.h>

void f(int k){ /* f: print k dots in line*/
  while (k>0) {
    printf(".");
    --k;
  }
```

```c
  printf("\n");
} /* k is zero when function is left */

int main(void) {
  int k = 10;
  printf("k=%d\n",k);
  f(k);
  printf("k=%d\n",k);
  return 0;
}
```

c/passbyvalue.out

```
k=10
..........
k=10
```

ooo → **lab 4**

Symbolic constants

(K&R, Sec 1.4, page 14)

If we use constants in a program, it is good practice to define a *symbolic name* or *symbolic constant*:

`#define` *name replacement-text*

The pre-processor (runs silently before the actual compilation) will replace all occurrences of *name* in the file with *replacement-text*.

For example: allow to define type of integer that has the right size for use now but can be changed easily when needed. For example:

```
/* Temperature is never more than 127 */
#define TEMPERATURE char

/* Number of vertices is never more than 255 */
#define VERTEX_INDEX unsigned char
```

Another example: With a (32-bit) `int` integer variable:

c/define.c

```c
#include <stdio.h>
#define MYINT int

MYINT square(MYINT x) {
  return x*x;
}

int main(void) {
    MYINT a=3;
    printf("Myint a=%d uses %ld bytes. %d^2=%d\n",
           a, sizeof(a), a, square(a));
    return 0;
}
```

Output:

```
Myint a=3 uses 4 bytes. 3^2=9
```

With `char` integer variable (here 8 bit). Note: only change is in line 2.

c/define2.c

```c
 1 #include <stdio.h>
 2 #define MYINT char
 3 
 4 MYINT square(MYINT x) {
 5   return x*x;
 6 }
 7 
 8 int main(void) {
 9     MYINT a=3;
10     printf("Myint a=%d uses %ld bytes. %d^2=%d\n",
11            a, sizeof(a), a, square(a));
12     return 0;
13 }
```

Output:

```
Myint a=3 uses 1 bytes. 3^2=9
```

We can also use `#define` to define global constants:

c/symbolicconstant.c

```c
#include <stdio.h>

#define LOWER 0
#define UPPER 100
#define STEP 20

int main () /*print Fahrenheit-Celsius table */
{
  int fahr;
  for (fahr=LOWER; fahr <= UPPER; fahr += STEP) {
    printf("%3d %6.1f\n", fahr, 5.0/9.0*(fahr-32));
  }
  return 0;
}
```

Output:

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
```

stdin, stdout and sterr

- Unix(/Linux/Mac OS X) and (to some degrees) Windows have three default file descriptors:
  - ▷ 0: *standard input* (`stdin`)
  - ▷ 1: *standard output* (`stdout`)
  - ▷ 2: *standard error* (`stderr`)
- stdin is (by default) connected to the keyboard
- stdout and stderr are connected to the display
- Can use redirection (“>” and “<”) to change this

stdout and sterr redirection

- A command such as `ls` (Unix) or `dir` (Windows) will list the files in the current working directory (by sending the output to stdout).
- We can direct the stdout output (which has file descriptor 1) to a file:
```
dir 1> filelist.txt
```
- Error messages will be send to stderr and do not go into this file. To capture those, we need to redirect stderr (with file descriptor 2):
```
dir *.ccc  1>filelist.txt 2>filelist.error
```
- Instead of `1>` can simply use >
- Common notation: `dir *.ccc  >dir.stdout 2>dir.stderr`

Character Input and Output

(K&R, Sec 1.5, page 15)

- Command `c=getchar()` reads the *next input character* from the standard input text stream and returns it

- Command `putchar(c)` writes the c to the standard output text stream.

- `EOF` is a special macro representing End Of File (Linux: use CTRL+d on the keyboard to create this, Windows command: use CTRL+z (may have to be at beginning of new line, followed by RETURN)):

  Often EOF = -1, but implementation dependent. Must be a value that is not a valid value for any possible character.

  For this reason, `c` is of type `int` (not `char` as one may have expected).

  EOF is defined in `<stdio.h>`.

File copying

(K&R, Sec 1.5.1, page 16)

c/cat_long.c

```c
#include <stdio.h>

/* main: copy stdin to stdout */
int main(void) {
  int c;
  c=getchar();
  while (c != EOF) {    /* != means 'not equal' */
      putchar(c);
      c=getchar();
    }
  return 0;
}
```

Example usage (making copy of `cat.c` to `copy_of_cat.c`):

```
fangohr$ ./cat_long <cat_long.c >copy_of_cat_long.c
```

Experienced C programmers will shorten `cat_long.c` to:

c/cat.c

```c
#include <stdio.h>

/* main: copy stdin to stdout */
int main(void) {
  int c;
  while ((c=getchar())!=EOF){
    putchar(c);
  }
  return 0;
}
```

Counting Characters in file

(K&R, Sec 1.5.2, page 17)

c/charcount.c

```c
#include <stdio.h>

/* main: count number of characters in stdin */
int main(void) {
  long nc=0;             /* Number of Characters */
  while (getchar()!=EOF){
    nc++;
  }
  printf("%ld\n", nc);
  return 0;
}
```

```
$ ./charcount < charcount.c
215
$ wc -m charcount.c
     215 charcount.c
```

Line counting

(K&R, Sec 1.5.3, page 19)

c/linecount.c

```c
#include <stdio.h>

/* main: count number of lines in stdin */
int main(void) {
  long nl=0;          /* Number of Lines */
  int c;
  while ((c=getchar())!=EOF) {
    if (c=='\n')
      nl++;
  }
  printf("%ld\n", nl);
  return 0;
}
```

```
$ ./linecount < linecount.c
14
$ wc -l linecount.c
      14 linecount.c
```

Word counting

(K&R, Sec 1.5.4, page 20)

c/wordcount.c

```c
#include <stdio.h>

#define IN  1 /*inside a word*/
#define OUT 0 /*outside a word*/

/* main: count lines, words and characters in input */
int main(void) {
  int c, nl, nw, nc, state;
  state = OUT;
  nl = nw = nc = 0;
  while ((c=getchar())!=EOF) {
    ++nc;
    if (c=='\n')
      ++nl;
    if (c==' ' || c=='\n' || c =='\t')
      state = OUT;
    else if (state == OUT) {
      state = IN;
      ++nw;
    }
  }
  printf("%d %d %d\n", nl, nw, nc);
  return 0;

}
```

```
$ ./wordcount < wordcount.c
26 84 474
$ wc < wordcount.c
      26      84     474
```

ASCII table

The American Standard Code for Information Interchange (ASCII, pronunciation: ass-kee) is a character-encoding scheme based on the ordering of the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that use text. (from Wikipedia, http://en.wikipedia.org/wiki/ASCII, 6 Nov 2011

C-program that prints ASCII numbers from 32 up to 126.

c/ascii–table.c

```c
#include <stdio.h>
int main(void) {
  int c;
  for (c=32; c<127; c++) {
    printf("%3d : %c\n",c,c);
  }
  return 0;
}
```

ASCII table output

```
32 :      52 : 4    72 : H     92 : \    112 : p
33 : !    53 : 5    73 : I     93 : ]    113 : q
34 : "    54 : 6    74 : J     94 : ^    114 : r
35 : #    55 : 7    75 : K     95 : _    115 : s
36 : $    56 : 8    76 : L     96 : `    116 : t
37 : %    57 : 9    77 : M     97 : a    117 : u
38 : &    58 : :    78 : N     98 : b    118 : v
39 : '    59 : ;    79 : O     99 : c    119 : w
40 : (    60 : <    80 : P    100 : d    120 : x
41 : )    61 : =    81 : Q    101 : e    121 : y
42 : *    62 : >    82 : R    102 : f    122 : z
43 : +    63 : ?    83 : S    103 : g    123 : {
44 : ,    64 : @    84 : T    104 : h    124 : |
45 : -    65 : A    85 : U    105 : i    125 : }
46 : .    66 : B    86 : V    106 : j    126 : ~
47 : /    67 : C    87 : W    107 : k
48 : 0    68 : D    88 : X    108 : l
49 : 1    69 : E    89 : Y    109 : m
50 : 2    70 : F    90 : Z    110 : n
51 : 3    71 : G    91 : [    111 : o
```

Bools

- False is represented by the value 0
- True is represented by the value 1
- Any non-zero value evaluates to True
- the `||` operator is a logical OR
- the `&&` operator is a logical AND
- AND has just slightly higher precedence than OR (K&R, p 21)
- Expressions connected by `||` and `&&` are evaluated left to right, evaluation stops as soon as truth or falsehood is known

Letter or number? (char, int)

c/char–experiment.c

```c
#include<stdio.h>

int main(void) {
  int c;
  c = 'A';
  printf(" c='A' => c = %c = %d\n",c,c);
  c = 'd';
  printf(" c='d' => c = %c = %d\n",c,c);
  c = 'd'-'A';
  printf(" c='d'-'A' => c = %c = %d\n",c,c);
  return 0;
}
```

```
c='A' => c = A = 65
c='d' => c = d = 100
c='d'-'A' => c = # = 35
```

Arrays

(K&R, Sec 1.6, page 22)

c/digitcount.c

```c
#include <stdio.h>

/* main: count digits, white space, others */
int main(void) {
  int c, i, nwhite, nother;
  int ndigit[10];
  nwhite = nother = 0;
  for (i = 0; i < 10; ++i) {
    ndigit[i] = 0;
  }

  while ( (c=getchar())!=EOF) {
    if (c >= '0' && c <= '9') {
      ndigit[c-'0'] += 1;
    }
    else {
      if (c==' ' || c=='\n' || c=='\t')
        ++nwhite;
      else
        ++nother;
    }
  }

  printf("digits =");
  for (i=0; i<10; ++i) {
    printf(" %d", ndigit[i]);
  }
  printf(", white space = %d, other = %d\n",
         nwhite, nother);

  return 0;
}
```

```
$ ./digitcount < digitcount.c
digits = 10 4 0 0 0 0 0 0 0 1, white space = 170, other = 374
```

Arrays

- Can declare array variable (here `ndigit`) providing an array with space for n elements of a particular (and fixed) type: `int ndigit[10]` (here $n = 10$).
- Values can be accessed by indexing `ndigit[i]` using square brackets
- First element has index 0 and last element has index $n - 1$.
- Must remember the size of the array (`ndigit` cannot tell us)
- Must avoid read/write to elements outside the array (i.e. < 0 or $>= n - 1$)
- Must initialise elements (may carry random values)

Strings (=character arrays)

- Strings ("chunks of text") are represented through arrays of char
- Can declare char array s of length 10 as:

  ```
  char s[10];
  ```

- `s` is actually a pointer to the beginning of the array.
- Convention in C: put special value \0 in array to indicate end of string (called the *null character*, K&R, p30). For example "`hello\n`" is stored as an array of char with 7 entries:

  ```
  -----------------------------
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
  -----------------------------
  | h | e | l | l | o | \n| \0|
  -----------------------------
  ```

c/helloarray0.c

```c
#include<stdio.h>

/* demonstrate string termination I */
int main( void ) {
  char a[10];
  a[0]='h'; a[1]='e'; a[2]='l'; a[3]='l';
  a[4]='o'; a[5]='\n'; a[6]='\0';
  printf("%s",a);
  return 0;
}
```

```
hello
```

Allocate array of appropriate size automatically from initial value:

c/helloarray.c

```c
#include<stdio.h>

/* demonstrate string termination I */
```

```c
int main( void ) {
  char s[]="hello\n";
  int i;
  printf("%s",s);
  for (i=0; i<7; i++)
    printf("s[%d]='%c'=%3d\n",i,s[i],s[i]);
  return 0;
}
```

```
hello
s[0]='h'=104
s[1]='e'=101
s[2]='l'=108
s[3]='l'=108
s[4]='o'=111
s[5]='
'= 10
s[6]=''=  0
```

Manual setting of \0:

c/stringtest.c

```c
#include <stdio.h>

int main(void) {
  char s[27];
  int i;
  for (i=0; i<26; i++) {/* put abc...xyz in s */
    s[i] = 'A'+i;
  }
  s[26]='\0';              /* terminate string */
  printf("s=%s\n",s);
  s[10]='\0';              /* terminate string earlier*/
  printf("s=%s\n",s);
  return 0;
}
```

```
s=ABCDEFGHIJKLMNOPQRSTUVWXYZ
s=ABCDEFGHIJ
```

Character Arrays: print longest line

K&R, 1.9, page 28

Write a program that reads a set of text lines and prints the longest. Strategy:

```
while (there is another line)
    if (it is longer than the previous longest)
        save it
        save its length
print longest line
```

- clear strategy divides problem in subproblems

- write code to reflect strategy: use functions
  - ▷ `getline()` to fetch the next line of input and
  - ▷ `copy(to,from)` to save line

plus main program putting it all together.

c/longestline.c

```c
#include <stdio.h>
#define MAXLINE 1000

/* function prototypes */
int get_line(char line[], int maxline);
void copy(char to[], char from[]);

/* main: print longest input line */
int main(void) {
  int len;                  /* current line length       */
  int max;                  /* maximum length seen so far */
  char line[MAXLINE];       /* current input line        */
  char longest[MAXLINE];    /* longest line saved here   */

  max = 0;
  while ((len = get_line(line, MAXLINE))>0) {
    if (len > max) {
      max = len;
      copy(longest, line);
    }
  }
  if (max > 0) {      /* there was a line */
    printf("%s",longest);
  }
  return 0;
}

/* get_line: read a line into s, return length */
int get_line(char s[], int lim) {
  int c, i;

  for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i) {
    s[i]=c;
  }
  if (c == '\n') {
    s[i] = c;
    ++i;
  }
  s[i] = '\0';
  return i;
}

/* copy: copy 'from' to 'to': assume to is big enough */
void copy(char to[], char from[]) {
  int i=0;
  while ((to[i]=from[i])!='\0') {
    i++;
  }
}
```

- `getline()` communicates through
  - ▷ (i) return value (length of line), 0 means EOF (because an empty line contains one character: `'\n'`.
  - ▷ (ii) parameter `char s[]`: s is pointer to location in memory that keeps array

- `copy()` communicates through parameters

- `char c[1000]` reserves space for 1000 characters.

ooo $\rightarrow$ **lab 5,6**

Common coding Gotchas in C

“Gotcha” from “Got you”

1. missing semicolon
2. missing closing curly brace
3. using uninitialised variables
4. unexpected integer division truncation
5. format specifier in printf-statement of wrong data type
6. missing parenthesis around logical expression in if-statement

   (. . . there are more . . .)

Finding errors(I): compiling fails

1. read the error message carefully:
   - it could point to the error (the ideal case)
   - it could point close to the point of error

2. check code for missing semicolons and non-matching curly-braces

3. "reduce" your program by commenting out parts of the code (using `/*` and `*/`) until the remaining program compiles.

4. Then
   - remove the comments (line by line) and
   - re-compile until compilation fails

   That's where the bug is hiding.

Finding errors(II): execution fails

Typical failure modes:

- program never stops (infinite loop)
- program computes the wrong answer
- execution aborts with error message (or computer crash)

General advice:

- Introduce printf-statements in the code that by reading these (while the program executes), you can deduce what happens.

(There are more sophisticated tools to de-bug programs, so-called “debuggers”.)

sizeof

- The “sizeof(t)“ operator returns the number of bytes[2] used to store a variable of type “t“.
- The return value of `sizeof` is of type `size_t` (which must be an unsigned integer type).
- Need “sizeof“ to allocate memory dynamically.
- Can use to study the platform we are working on using a program like

c/sizeof.c

```
#include <stdio.h>

```

[2] strictly the number of `chars` which is the basic unit in C, but effectively all important architectures use char=byte today.

```
int main(void) {
  printf("sizeof(char)    = %d\n",(int) sizeof(char));
  printf("sizeof(short)   = %d\n",(int) sizeof(short int));
  printf("sizeof(int)     = %d\n",(int) sizeof(int));
  printf("sizeof(long)    = %d\n",(int) sizeof(long int));
  printf("sizeof(float)   = %d\n",(int) sizeof(float));
  printf("sizeof(double)  = %d\n",(int) sizeof(double));
  printf("sizeof(double*)= %d\n",(int) sizeof(double*));
  printf("sizeof(char*)   = %d\n",(int) sizeof(char*));
  printf("sizeof(FILE*)   = %d\n",(int) sizeof(FILE*));

  return 0;
}
```

Output on a 64-bit Mac OS X 10.6 on Intel Core 2 Duo (MacBook) and

Output on a 64-bit Ubuntu Linux 10.6 on Intel Intel i7 CPU Nehalum (PC)

c/sizeof.out64bit

```
sizeof(char)    = 1
sizeof(short)   = 2
sizeof(int)     = 4
sizeof(long)    = 8
sizeof(float)   = 4
sizeof(double)  = 8
sizeof(double*)= 8
sizeof(char*)   = 8
sizeof(FILE*)   = 8
```

Output on a 32-bit Debian 5.0 on AMD64 CPU (PC)

c/sizeof.out32bit

```
sizeof(char)    = 1
sizeof(short)   = 2
sizeof(int)     = 4
sizeof(long)    = 4
sizeof(float)   = 4
sizeof(double)  = 8
sizeof(double*)= 4
sizeof(char*)   = 4
sizeof(FILE*)   = 4
```

Pointers

- Pointer is variable that contains address of a variable (K&R, chap 5, p 93).
- & is the *address operator*: for “char c;“ and “p“ being a pointer to char:

  ```
  p = &c;
  ```

  assigns the address of “c” to the pointer “p”. We say that “p” *points to* “c”.
- To *declare* a pointer of type “char”, we need to use

  ```
  char *p;
  ```

- Outside declarations and outside function parameter lists, “*” is the *indirection* or *redirection operator*: when applied to a pointer, it accesses the object the pointer points to.

c/pointer1.c

```c
#include <stdio.h>
int main(void) {
  int x=1, y=2;
  int *pi;                    /* pi is pointer to int */
  pi = &x;                    /* pi now points to x */
  y = *pi;                    /* y is now 1 */
  *pi = 0;                    /* x is now 0 */
  pi = &y;                    /* pi points to y */
  printf("x=%d, y=%d \n",x,y);
  printf("address x=%p, address y=%p \n",
         (void *) &x,  (void *) &y     );
  printf("pi=%p, *pi=%d \n",(void *) pi,*pi);
return 0;}
```

```
x=0, y=1
address x=0x7fff59ac02b8, address y=0x7fff59ac02b4
pi=0x7fff59ac02b4, *pi=1
```

More pointer rules

- The declaration of a pointer “ip“ to int “i“:

  ```
  int *ip;
  ```

  is intended as a mnemonic; it says that the expression “*ip“ is an int.

  The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear (see function “swap“).

- If `ip` points to the integer `i`, then `*ip` can occur in any context where `i` could, for example `*ip = *ip + 10`, or `y=*ip+1` or `*ip+=1` which is equivalent to `++*ip` and also `(*ip)++`.

- unary operators `*` and `&` bind more strongly than arithmetic operators

- unary operators associate from right to the left, i.e. `*ip++` $\Leftrightarrow$ `*(ip++)`.

Example: swap, K&R, 5.2, p95

- C passes arguments to functions by value, i.e. by providing a local copy of the argument value.
- Thus, functions can not alter a variable in the calling function. Example:

c/pointer_swap1.c

```c
#include <stdio.h>
void swap( int x, int y) { int tmp=x; x=y; y=tmp;}

int main(void) {
  int a=1, b=2;
  printf("Before swap a=%d, b=%d\n",a,b);
  swap(a,b);
  printf("After swap a=%d, b=%d\n",a,b);
return 0;}
```

```
Before swap a=1, b=2
After swap a=1, b=2
```

Solution: pass pointers of values to be swapped to function:

c/pointer_swap2.c

```
#include <stdio.h>
void swap(int *px, int *py)
{int tmp=*px; *px=*py; *py=tmp;}

int main(void) {
  int a=1, b=2;
  printf("Before swap a=%d, b=%d\n",a,b);
  swap(&a,&b);
  printf("After swap a=%d, b=%d\n",a,b);
return 0;}
```

```
Before swap a=1, b=2
After swap a=2, b=1
```

⇒ pointer arguments enable a function to access and change objects in the function that called it.

scanf() again

- We now know enough to understand exactly what is happening when using the scanf() function seen earlier. The argument that goes in to scanf after the format specifier is meant to be a pointer so that any input is assigned to the memory address of the appropriate variable.

Pointers and arrays K&R, 5.3, p97

- In C, there is a strong relationship between pointers and arrays
- Any operation that can be achieved using subscripting can also be done with pointers.
- K&R: "The pointer version will generally be faster but, at least to the uninitiated, somewhat harder to understand". Not true anymore: use index notation to help compiler.
- The declaration

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

  defines an array a of size 10, that is, a block of 10 consecutive objects named `a[0], a[1], ..., a[9]`. Each object requires `sizeof(int)` bytes storage.

- With the declaration

  ```
  int *pa;          /* pointer to int */
  ```

  the assignment `pa = &a[0]` sets `pa` to point to `a[0]`.

- but since the array name `a` is a synonym for the location of the initial element, we can use `pa=a` instead of `pa = &a[0]`.

- Using so-called *pointer arithmetic*, one can write

  ▷ a+0=a instead of `&a[0]`
  ▷ a+1 instead of `&a[1]`
  ▷ a+2 instead of `&a[2]`, i.e. generically
  ▷ `a+i` instead of `&a[i]` to access *the address of element* `i`

- Using indirection operator on expressions above, we can access *the element* `i`:

  ▷ `*(a+0)=*a` instead of `a[0]`
  ▷ `*(a+1)` instead of `a[1]`

- ▷ `*(a+2)` instead of `a[2]`, i.e. generically
- ▷ `*(a+i)` instead of `a[i]` to access the value of element `i`

Array and pointer example

c/pointer_array.c

```c
#include <stdio.h>
#include <string.h>
int main(void) {
  char c[]="Hello World---";
  int i=0;
  for (i=0; i<strlen(c); i++) {/* use array      */
    printf("%c", c[i]);         /* subscripts     */
  }
  for (i=0; i<strlen(c); i++) {/* use    pointer */
    printf("%c", *(c+i));       /* arithmetic     */
  }
return 0;}
```

```
Hello World---Hello World---
```

strcpy example

c/snippet–only/strcpy1.c

```c
/* strcpy: copy t to s; array subscript version */
void strcpy( char *s, char *t) {
  int i=0;
  while ((t[i]=s[i]) != '\0')
    i++;
}

/* strcpy: copy t to s; pointer version */
void strcpy( char *s, char *t) {
  while ((*t=*s) != '\0') {
    s++;
    t++;
  }
}


```

```c
/* strcpy: copy t to s; pointer version 2 */
void strcpy( char *s, char *t) {
  while ((*t++=*s++) != '\0')
    ;
}
```

Use of pointers

Use of pointers is required when

- allocating memory *dynamically*
- sharing data between/with functions (can pass pointer to data [such as an array] to functions)
- passing functions to functions (K&R, 5.11, p 118)
- Data structures (trees, lists, . . .)

show memory allocation example, trivial

c/mallocint.c

```c
#include <stdio.h>
#include <stdlib.h>  /* provides malloc */
int main(void) {
  int *pi;
  pi = (int *)malloc(sizeof(int));
  if (pi == NULL) {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *pi = 5;
  printf("%d\n", *pi);
  free(pi);
  return 0;
}
```

```
5
```

The `malloc` function (called in line 5):

- checks that enough memory is available (on heap)
- if not enough memory available, return address zero (=`NULL`) as an error code.
- if memory available, `malloc()` *allocates* or *reserves* the required amount of memory and
- returns a pointer to that reserved block (i.e. the pointer variable then contains the address of that reserved block).
- The return type of `malloc` is pointer to `void` (i.e. `void *`) so we need to *cast* the pointer to the right type (i.e. `int *`).

Lines 10 and 11 represent the main activity, i.e. use of the pointer `pi` to store an integer.

The `free` function in line 12 marks the reserved memory as released ("freed").

Questions:

- *Is it really important to check that the pointer is zero after each allocation?*

  Yes, should always be checked.

- *What happens if I forget to release a block of memory before the program terminates?*

  The operating system will release the memory when the program terminates. Not releasing memory that is unused can lead to *memory leaks*, and is not considered good style either.

- What is the difference from using `malloc` to just declaring an int using `int i;`?

  Memory allocation through `malloc` takes place *at run-time*, where as declarations such as `int i` are static and have to be known at compile time.

Memory allocation example for array

c/mallocint2.c

```c
#include <stdio.h>
#include <stdlib.h>  /* provides malloc */
#define N 10         /* define array length */
int main() {
  int i;
  int *a; /* array of int */
  a = (int *)malloc(sizeof(int)*N);
  if (a == NULL) {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  else {
    printf("Have allocated %d bytes for a.\n",
           (int) sizeof(int)*N);
  }
  for (i=0; i<N; a[i++]=0) ; /* initialise */
  a[4]=42;                   /* some activity */
  for (i=0; i<N; printf("%d ",a[i++])) ;
  free(a);
  return 0;
}
```

```
Have allocated 40 bytes for a.
0 0 0 0 42 0 0 0 0 0
```

Slides starting on 210 show how to allocate memory for 2d and 3d arrays.

Typical use of pointers

- allocate memory using malloc and use pointer to point to that memory
- point the pointer to an address at which memory has been reserved already (and is typically in use by another variable), either by assigning an existing pointer to a pointer, or by using the address operator `&` to obtain the address of a variable.
- Initialise the pointer with `NULL` (==0). This is a signal (to the programmer) that the pointer does not point to a meaningful position.

Pointers are usually used if the “length” of data is not known at compile time. For example when

- reading a file and we don’t know the length of the file (can read line by line, use dynamic memory allocation memory line by line)

- running a simulation and we don't know the size of the simulation (for example the number of particles is read from a file, and memory needs to be allocated for each particle),

- having to use a vector/matrix/tensor where size will vary from run to run.

ooo $\rightarrow$ **lab 7,8**

Structures

- It often makes sense to be able to group data together in to a single place so that it can be more easily understood and used.
- For instance, you may be dealing with shapes - all have a colour, a size, a position.
- You may be keeping records of people and each will have a name, a height, weight etc.
- It makes more sense and is often conceptually clearer to encapsulate this data.
- In C, we use the *struct* to do this.

struct declaration example

c/struct1.c

```c
#include<stdio.h>

struct person
{
	int age;
	double height;
	double weight;
};

int main(void) {
	struct person BillyBob={34,1.93,85.0};
	return 0;
}
```

struct usage example: name.member

c/struct2.c

```c
#include<stdio.h>

struct person
{
  int age;
  double height;
  double weight;
};

int main(void) {
  struct person BillyBob={34,1.93,85.0};
  printf("The age is %d.\n",BillyBob.age);
  printf("The height is %.1f.\n",BillyBob.height);
  printf("The weight is %.1f.\n",BillyBob.weight);
  return 0;
}
```

Passing a struct to a function example

Using structs in functions is entirely the same as any other data type and this is the case with all C syntax - it is possible to have arrays of structs and pointers to structs etc.

c/struct3.c

```
#include<stdio.h>

struct person
{
        int age;
        double height;
        double weight;
};

void printDetails(struct person A)
{
        printf("The age is %d.\n",A.age);
        printf("The height is %.1f.\n",A.height);
        printf("The weight is %.1f.\n",A.weight);
}

int main(void)
{
        struct person BillyBob={34,1.93,85.0};
        printDetails(BillyBob);
        return 0;
}
```

Example: computation of Fibonacci numbers

In Python:

python/fibrec.py

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

import time
starttime = time.time()
print(fib(36))
print("{} seconds".format(time.time() - starttime))
```

`f(32)` took 2.1s (on Lyceum) and `f(36)` took 15.3s

In Matlab:

matlab/fibrec.m

```
function fibrec()

tic
disp(fib(36))
toc

function res=fib(n)

if n==0
  res=0;
else
  if n==1
    res=1;
  else
    res = fib(n-1)+fib(n-2);
  end
end
```

Lyceum: Matlab 7.7 (2008)

`fib(32)` took 72 sec (34 times slower than Python),

`fib(36)` took 462 sec (30 times slower than Python)

c/fibrec.c

```c
#include <stdio.h>
#include <time.h>
/* Compute Fibonacci number n
   (recursively */
int fib(int n) {
  if (n==0)
    return 0;
  else
    if (n==1)
    return 1;
    else
      return fib(n-1) + fib(n-2);
}

int main(void) {
  time_t starttime;
  starttime = time(NULL);
  printf("fib(36)=%d\n",fib(36));
  printf("Time taken: %4.2g\n",(double) (time(NULL)-starttime));
  return 0;
}
```

`fib(36)` took 0.33 sec.

In summary, we find the relative speed *for this example* (all on Lyceum for `fib(36)`)

- Python: 15.3s $\rightarrow$ factor 45 slower than C
- Matlab: 462s $\rightarrow$ factor 1400 slower than C

See also slides 160 to 175 for a more detailed study on execution speed.

Fibonacci faster

c/fibarray.c

```c
#include <stdio.h>
#define N 36
long fib(int n) {
  long fibs[N+1];
  long i;
  fibs[0]=0; fibs[1]=1;
  for (i=2; i<=N; i++) {
    fibs[i]=fibs[i-1]+fibs[i-2];
  }
  return fibs[N];
}

int main(void) {
  printf("fib(%d)=%ld\n",N,fib(N));
  return 0;
}
```

```
fib(36)=14930352
```

Nice, but requires use of array. In program above, there is a maximum size hard coded (due to the array).

We can get around this with *dynamic memory allocation* ($\rightarrow$ laboratory session).

A speed comparison example

Example content: The upper boundary of a two-dimensional geometric shape is described by $y(x) = \sqrt{1 - x^2}$ for $-1 \le x \le 1$, and the lower boundary by y $=$ 0. If mirrored along the x-axis, this shape would be a circle with radius 1 and area π.

We use numerical integration of $y(x)$ to approximate π with $h(y(a) + y(b) + 2\sum_{i=1}^{n-1} y(x_i))$ where $a = -1$, $b = 1$, and $x_i = a + ih$ and $h = (b - a)/2$. (There are of course other ways of determining π.)

We use the (relatively) simple composite trapezoidal rule with $n = 10,000,000$ subdivisions to integrate $f(x) = y(x) = \sqrt{1 - x^2}$ from $a = -1$ to $b = 1$. (There are more efficient ways of computing such an integral numerically, but here we are more interested in the computational speed than the algorithmic sophistication.)

We will solve the same problem in Matlab, Python and C. They all produce the same answer (for the same number of subdivisions).

Overview: Making Python code faster

We assume that a code has been written (or prototyped) in Python.

Often only few lines or functions of a simulation code are taking most of the execution time. Once we have found those (profiling is key), we can use multiple techniques to speed up existing Python code:

Option 1: 'Within' Python:

- inline functions, reduce creation and deletion of objects (and in general: choose the fastest algorithm)
- Use compiled libraries (such as numpy: for numpy we need to 'vectorise' our code, i.e. express the problem in matrices)

Option 2: Exploit compiled code (here C):

- Include C-code strings in Python ($\rightarrow$ weave)
- Link to (existing?) C libraries ($\rightarrow$ ctypes)
- Compile Python code ($\rightarrow$ Cython)

Other tools available: Swig, Boost, f2py, can also use Cython to access C-libraries, and much more.

Python, conceptually clear

python/pi.py

```
import math
def f(x):
    return math.sqrt(1. - x * x)

def pi(n):
    s = 0.5 * (f(-1) + f(1))
    h = 2. / n
    for i in range(1, n):
        x = -1 + i * h
        s += f(x)
    return s * h * 2

import time
starttime = time.time()
b = pi(10000000)
stoptime = time.time()
print("%.15g, %g" % (b, math.pi - b))
print(math.pi)
print("Needed %g seconds" % (stoptime - starttime))
```

Output:

```
3.14159265348482, 1.04977e-10
3.14159265359
Needed 10.8633 seconds
```

Execution time:

Lyceum: 14.5s

MacBook Pro: 11s

Python, function $f(x)$ inline

Reducing the overhead of calling the function can make code faster.

python/pi_inlined.py

```
import math
def pi(n):
    s = 0
    h = 2. / n
    for i in range(1, n):
        x = -1 + i * h
        s += math.sqrt(1. - x * x)
    return s * h * 2

import time
starttime = time.time()
b = pi(10000000)
stoptime = time.time()
print("%.15g, %g" % (b, math.pi - b))
print("%.15g" % math.pi)
print("Needed %g seconds" % (stoptime - starttime))
```

Output:

```
3.14159265348482, 1.04977e-10
3.14159265358979
Needed 6.681 seconds
```

Lyceum: 8.2s

MacBook Pro: 6.6s

Python, using numpy

Delegate some of the operations to numpy (which uses compiled code).

python/pi_array.py

```python
import time
import math
import numpy

def pi(n):
    x = 1. - numpy.linspace(0, 2, n + 1)
    y = numpy.sum(numpy.sqrt(1 - numpy.power(x, 2)))
    s = numpy.sum(y)
    return s * 4. / n

starttime = time.time()
b = pi(10000000)
stoptime = time.time()
print("%.15g, %g" % (b, math.pi - b))
print(math.pi)
print("Needed %g seconds" % (stoptime - starttime))
```

Output:

```
3.14159265348482, 1.04976e-10
3.14159265359
Needed 1.14756 seconds
```

MacBook Pro: 1.14s

Lyceum: 1.36s

Mixing C and Python

A common strategy is to replace those (and only those) parts of a program which consume most of the CPU time by fast (i.e. compiled) code. There are a number of Python tools supporting the mixing of C and Python (including Cython, Pyrex, Swig, Boost [in particular for C++], Weave). Here we use the `weave.inline` tool that comes with `scipy`:

python/pi_weaveinline.py

```
import time, math
from scipy import weave         # this is a Python program
def pi(n):
    my_C_code = """
          double s=0;       /* this is C code */
          double h=2./n;      /* in a string    */
          double x;
          long i;
          for (i=1; i<n; i++) {
            x = -1+i*h;
            s += sqrt(1.-x*x);
          }
          return_val = s*h*2;  /* last line C code */
          """
    return weave.inline(my_C_code, ['n'], compiler='gcc')

if __name__ == "__main__":  # main program (Python)
    starttime = time.time()
    b = pi(10000000)
    stoptime = time.time()
    print("%.15g, %g" % (b, math.pi - b))
    print math.pi
    print("Needed %g seconds" % (stoptime - starttime))
```

Output on Lyceum:

```
[fangohr@lyceum ~/tmp]$ python pi_weaveinline.py
<weave: compiling>
creating /tmp/fangohr/python24_intermediate/compiler_c9100a4644ba0ca4d2d2a1b35e817f54
3.14159265348482, 1.04977e-10
3.14159265359
Needed 1.57684 seconds
[fangohr@lyceum ~/tmp]$ python pi_weaveinline.py
3.14159265348482, 1.04977e-10
3.14159265359
Needed 0.205193 seconds
```

Note that this is slow the first time we execute it (as the c-code string is compiled into a C program) but very fast the second time. (MacBook Pro: 0.17s)

C

`[Code missing]` $\rightarrow$ lab exercise

Measurements:

Lyceum: 0.42s (0.20s if using -O3)

MacBook Pro: 0.27s (0.16 if using -O3)

Matlab

matlab/pi.m

```
function pi()

tic;
  disp(piEuler(10000000));
toc;

function res=f(x);
  res=sqrt(1.-x*x);

function s=piEuler(n)
s = 0.5*(f(-1)+f(1));
h=2./n;
for i=1:n-1
  x = -1+i*h;
  s = s+f(x);
end
s = 2*s*h;
```

Lyceum: 64s

MacBoo Pro: (Unknown)

Matlab, exploiting arrays

matlab/pi_array.m

```
tic
n=10000000;
x=-1:2./(n+1):1;
y=sqrt(1-x.^2);
s=sum(y);
disp(s*4./n);
toc
```

Lyceum: 0.21s

MacBoo Pro: (Unknown)

Speed example, summary

Use Lyceum numbers:

Language	Seconds	Factor
C (-O3)	0.2	1.00
Python weave	0.21	1.05
Matlab array	0.21	1.05
C	0.4	2.00
Python array	1.3	6.50
Python inline	8.2	41.00
Python plain	14.5	72.50
Matlab plain	64.0	320.00

Need interpretation warning on next slide.

Which one is fastest?

Actual performance will depend on

- Problem to solve, and implementation
- Hardware
- Version of OS, Compilers, Interpreters
- Other activity on the system

Example on previous slide shows that Matlab can be very fast if used well.

Not all performance possibilities have been exploited.

Cython — a modern approach to effective simulation code development

From http://www.cython.org: Cython is a language that makes writing C extensions for the Python language as easy as Python itself.

Cython gives you the combined power of Python and C to let you

- write Python code that calls back and forth from and to C or C++ code natively at any point.
- easily tune readable Python code into plain C performance by adding static type declarations.
- integrate natively with existing code and data from legacy, low-level or high-performance libraries and applications.

- interact efficiently with large data sets, e.g. using multi-dimensional NumPy arrays.

- use combined source code level debugging to find bugs in your Python, Cython and C code.

- quickly build your applications within the large, mature and widely used CPython ecosystem.

Another summary: *Cython is Python with C data types*.

Making Python code fast using Cython

- Central idea is to *automatically* translate Python code into C code (through the *Cython compiler*, and to compile the resulting C code using a (normal) C-compiler).
- This provides a modest performance gain over executing the Python code with the Python interpreter.
- We can improve the performance significantly if we can tell the Cython compiler which data types Python objects will be. For numeric code, this is often straightforward.

This is best demonstrated through an example:

- We stick to our previous example `python/pi.py`, and gather the following files in the directory `bench` for benchmarking performances:

bench/pi.py

```python
import math


def f(x):
    return math.sqrt(1. - x * x)


def pi(n):
    s = 0.5 * (f(-1) + f(1))
    h = 2. / n
    for i in xrange(1, n):
        x = -1 + i * h
        s += f(x)
    return s * h * 2
```

- We have replaced `range` with `xrange` for clarity (xrange does not actually create the list as it is a generator, and thus needs much less memory).

The command to translate the python file `pi.py` into C is:

```
cython pi.py
```

- For clarity of the following steps, we copy our file `bench/pi.py` and save it under the new name `bench/cy_simple.pyx`, and execute

```
cython cy_simple.pyx
```

  Cython files (i.e. the files that we will compile to C code using the Cython compiler) have by convention the extension `pyx`.

  In this particular case, the code is both valid Python and valid Cython.

- The resulting file `cy_simple.c` is long (> 2000 lines) and difficult to read (show only beginning):

bench/cy_simple.c

```c
/* Generated by Cython 0.19.1 on Sun Nov 24 23:10:45 2013 */

#define PY_SSIZE_T_CLEAN
#ifndef CYTHON_USE_PYLONG_INTERNALS
#ifdef PYLONG_BITS_IN_DIGIT
#define CYTHON_USE_PYLONG_INTERNALS 0
#else
#include "pyconfig.h"
#ifdef PYLONG_BITS_IN_DIGIT
#define CYTHON_USE_PYLONG_INTERNALS 1
```

```
#else
#define CYTHON_USE_PYLONG_INTERNALS 0
#endif
#endif
#endif
#include "Python.h"
#ifndef Py_PYTHON_H
    #error Python headers needed to compile C extensions, please install developmer
#elif PY_VERSION_HEX < 0x02040000
    #error Cython requires Python 2.4+.
#else
#include <stddef.h> /* For offsetof */
#ifndef offsetof
#define offsetof(type, member) ( (size_t) & ((type*)0) -> member )
#endif
#if !defined(WIN32) && !defined(MS_WINDOWS)
  #ifndef __stdcall
    #define __stdcall
  #endif
  #ifndef __cdecl
```

- To actually compile this into executable code which we can use from python, it is easiest to use the `pyximport` module as in the following example:

bench/call_cy_simple.py

```
import time

import pyximport       # this allows to import pyx modules as
pyximport.install()    # if they were normal python modules
                       # (cython will compile them automatically
                       # at that point)
import cy_simple       # access to compiled cython module

print cy_simple.pi(10000)   # call pi from compiled code
print cy_simple.__file__    # print the module's file

# Compare cython with plain python version
import pi
print pi.__file__

n = 10 ** 7
t_start = time.time()
value_python = pi.pi(n)
print "Native Python needs %f seconds" % (time.time() - t_start)
t_start = time.time()
value_cython = cy_simple.pi(n)
print "(Simple) Cython needs %f seconds" % (time.time() - t_start)
if value_cython == value_python:
    print("Both compute the same value: %.16f" % value_python)
```

which produces this output (on MacBook Air 1.7GHz Inter Core i5, Sept 2012):

```
3.14158932743
/Users/fangohr/.pyxbld/lib.macosx-10.5-i386-2.7/cy_simple.so
Native Python needs 6.832224 seconds
(Simple) Cython needs 5.180885 seconds
Both compute the same value: 3.1415926534848162
```

Discussion

- the cython version is actually compiled code (because the `cy_simple` module is a shared object file with name ending in `.so`).
- both the cython and the python version compute exactly the same number
- the cython version is slightly faster.

Can we do better?

Annotating the c-code that Cython produced

Cython provides the `-a` switch which stands for Annotate. It produces html output that shows

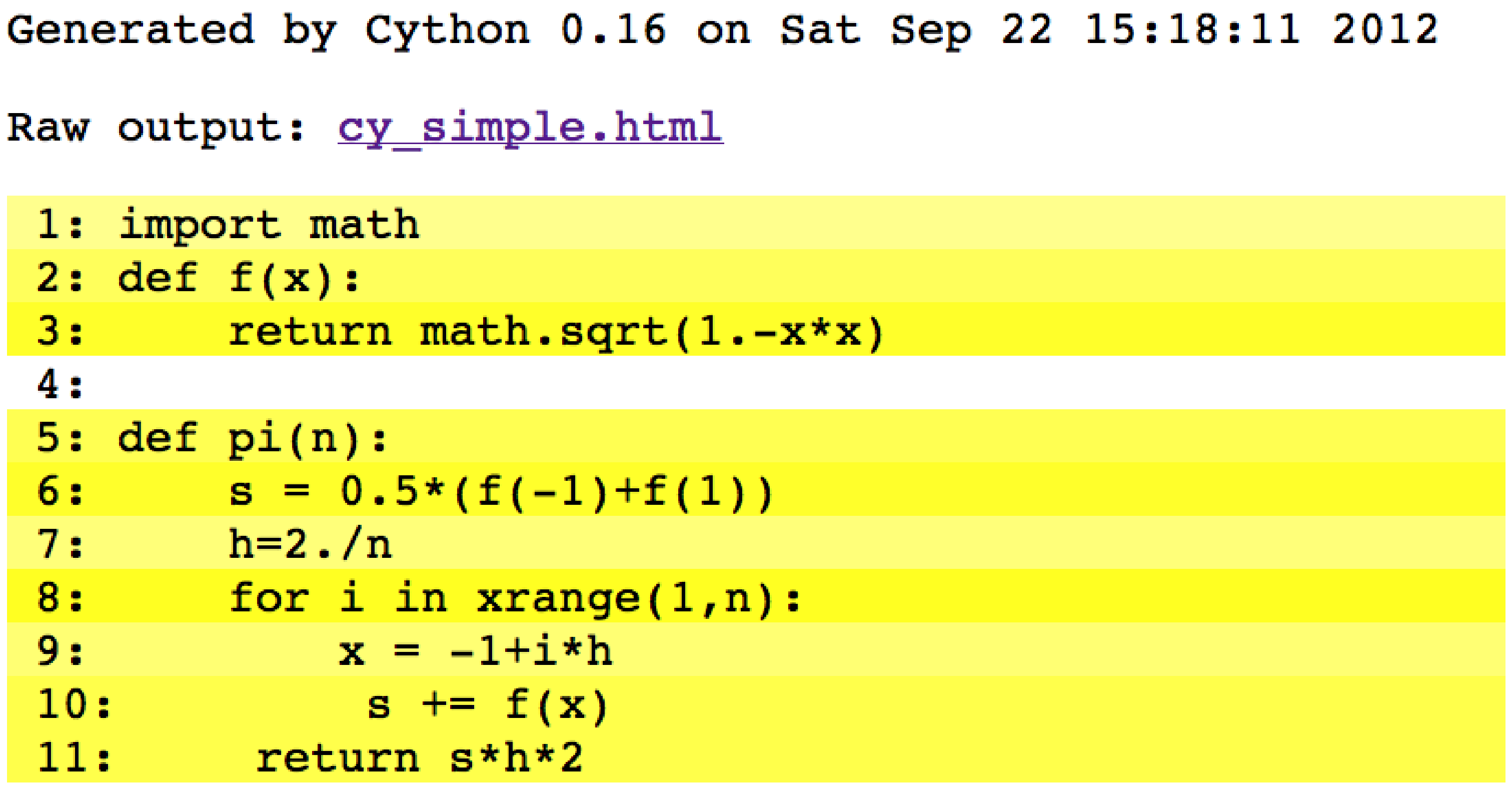
- the python source on white or yellow background
- line numbers can be double clicked and will reveal the C-code required to represent each line
- the background color represents the closeness of the resulting code to Python (yellow) and C (white).

We can use this to identify which parts of the code are slow (more yellow).

For a given pyx file, for example `cy_simple.pyx`, the command to produce the annotated html output is

```
fangohr$ cython -a cy_simple.pyx -o HTML/cy_simple.html
```

Example:



```
Generated by Cython 0.16 on Sat Sep 22 15:18:11 2012

Raw output: cy_simple.html

import math
def f(x):
    return math.sqrt(1.-x*x)

def pi(n):
    s = 0.5*(f(-1)+f(1))
    h=2./n
    for i in xrange(1,n):
        x = -1+i*h
        s += f(x)
    return s*h*2
```

Step 2: declaring types of variables

- The Cython compiler can not know what object types the variables in the code will be. (Python is a dynamically typed language.)
- The compiled code that is flexible enough to deal with any object type, is fairly slow: running such generic code is essentially what the python interpreter has to do.
- To make the cython-compiled c code faster, we have to restrict the object types to particularly (c) types.

Cython provides keywords that we can use in `pyx` to fix the data types.

We modify `bench/cy_simple.pyx` to become `bench/cy_opt.pyx` and include lines 7 to 9 to fix the type of `s`, `h`, and `x` to be `double`, and variable `i` to be of type `int`:

bench/cy_opt.pyx

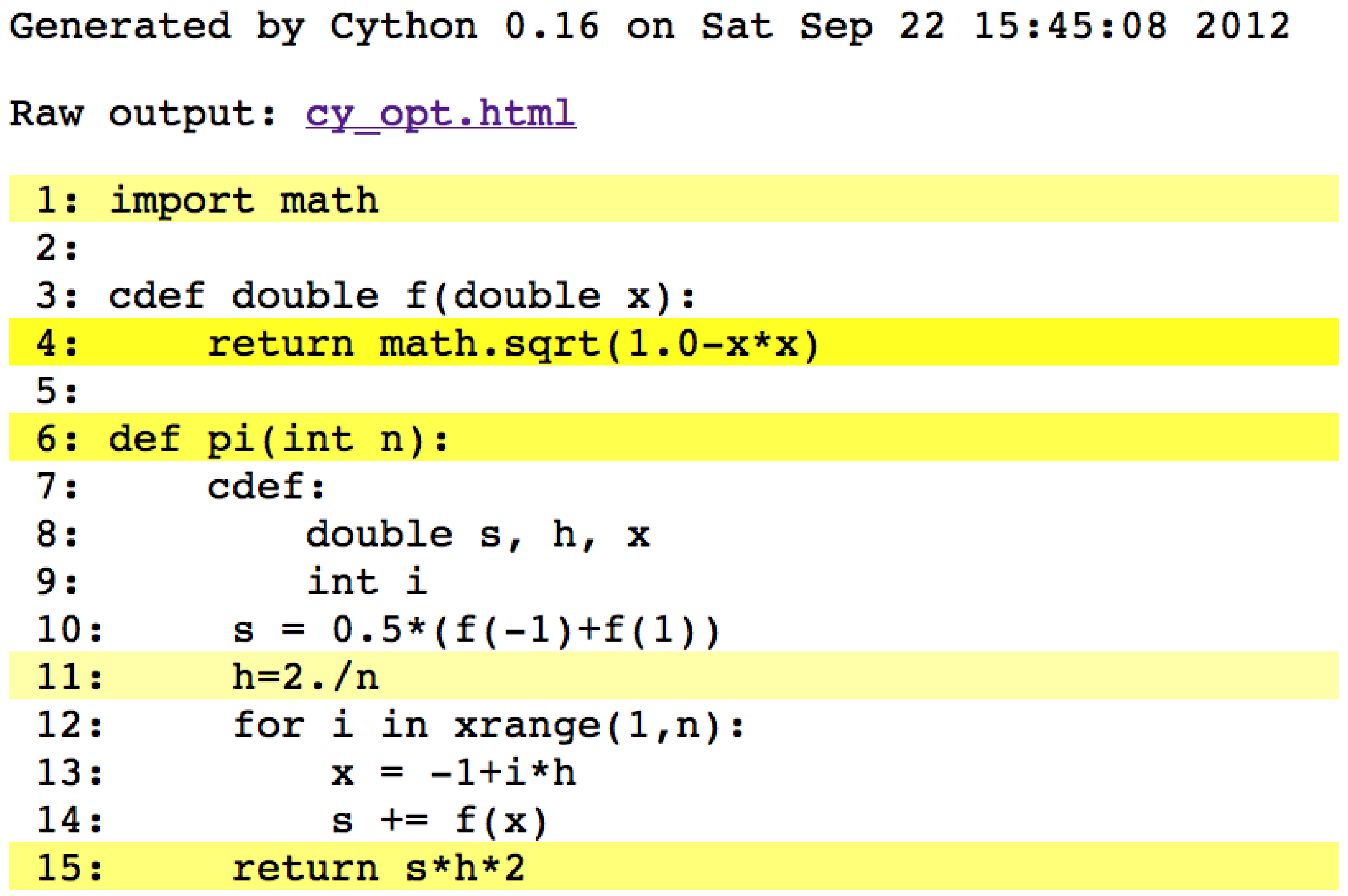
```
import math

cdef double f(double x):
    return math.sqrt(1.0-x*x)

def pi(int n):
    cdef:
        double s, h, x
        int i
    s = 0.5*(f(-1)+f(1))
    h=2./n
    for i in xrange(1,n):
        x = -1+i*h
        s += f(x)
    return s*h*2
```

.

We annotate the code again:



Generated by Cython 0.16 on Sat Sep 22 15:45:08 2012

Raw output: cy_opt.html

```
import math

cdef double f(double x):
    return math.sqrt(1.0-x*x)

def pi(int n):
    cdef:
        double s, h, x
        int i
     s = 0.5*(f(-1)+f(1))
     h=2./n
     for i in xrange(1,n):
         x = -1+i*h
         s += f(x)
     return s*h*2
```

and see that

- lines 10, 12, 13, 14 are truly white, and will thus execute at native C speed.

- In line 11, the code is still checking whether ZeroDivision might take place. Cython has a switch to disable this, however, the line is only executed once.

New performance data:

```
basic python:        6.8 seconds -> 4.0
naive cython:        5.0 seconds -> 2.9
typed cython:        1.7 seconds -> 1.0
```

A substantial amount of CPU time goes into the calculation of square roots. At the moment, we are using the sqrt function from the mathematics module. While this is very likely written in C as well, we need to interface from Cython's C code to Python's math module to then calculate a single square root using the (compiled C) code in the math module.

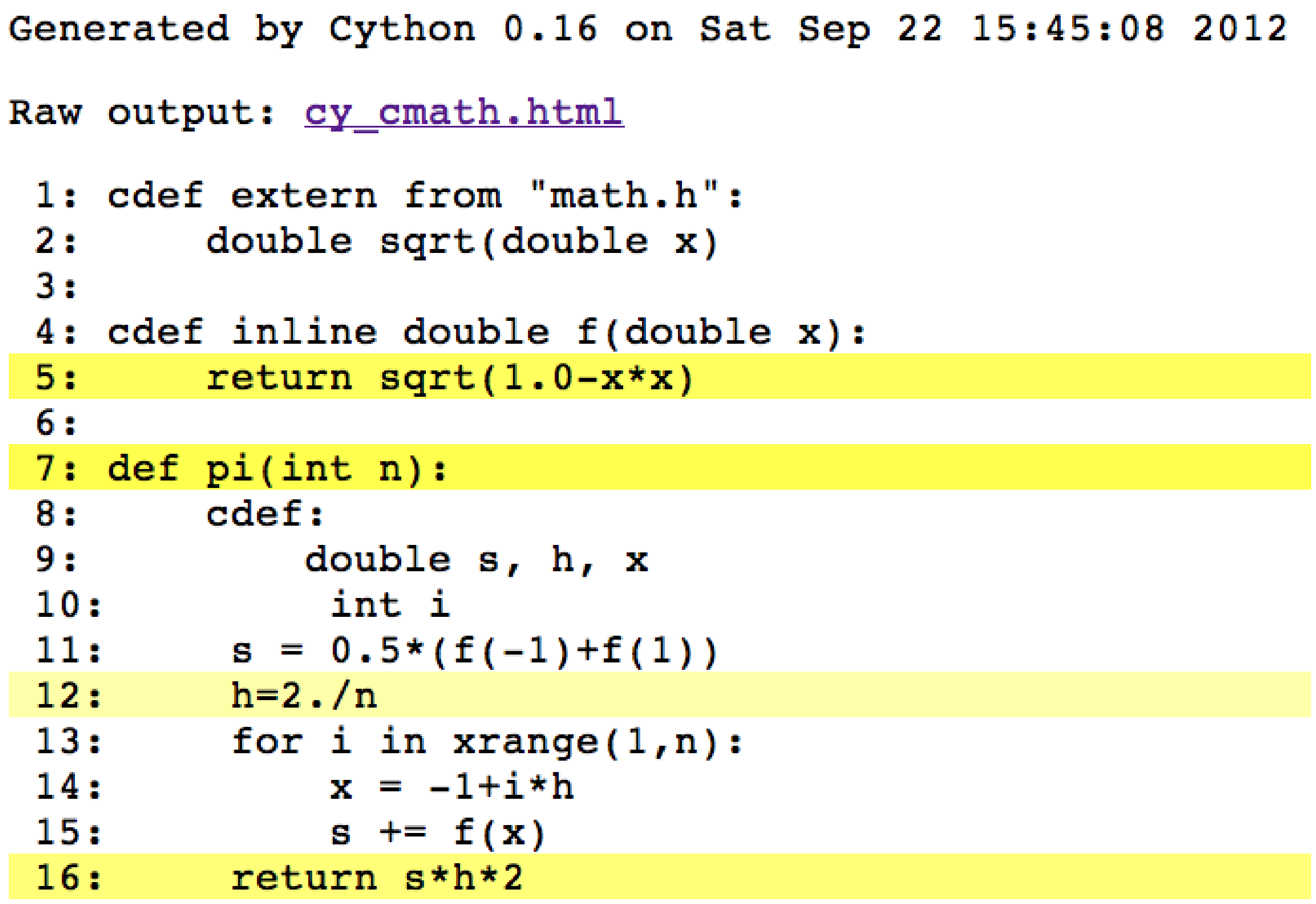
It would be faster if we could use the sqrt function from the C libraries immediately from Cython's C code:

bench/cy_cmath.pyx

```
cdef extern from "math.h":
    double sqrt(double x)

cdef double f(double x):
    return sqrt(1.0-x*x)

def pi(int n):
    cdef:
        double s, h, x
        int i
    s = 0.5*(f(-1)+f(1))
    h=2./n
    for i in xrange(1,n):
        x = -1+i*h
        s += f(x)
    return s*h*2
```



```
Generated by Cython 0.16 on Sat Sep 22 15:45:08 2012

Raw output: cy_cmath.html

cdef extern from "math.h":
    double sqrt(double x)

cdef inline double f(double x):
    return sqrt(1.0-x*x)

def pi(int n):
    cdef:
        double s, h, x
        int i
    s = 0.5*(f(-1)+f(1))
    h=2./n
    for i in xrange(1,n):
        x = -1+i*h
        s += f(x)
    return s*h*2
```

The performance figures are now

```
basic python:       6.841452 seconds -> 43.04261  (pi.py}
naive cython:       5.201624 seconds -> 32.72572  (cy_simple.pyx}
typed cython:       1.827127 seconds -> 11.49527  (cy_opt.pyx}
cmath cython:       0.158946 seconds -> 1.0       (cy_cmath.pyx}
```

With relatively little effort (adding a few `cdef` static type statements, and interfacing of `sqrt` from C library), we have made our code 40 times faster using Cython.

Overview performance

Finally, we also include the performance of native C code (compiled with different optimisation flags) and the `scipy.weave` code into the following table:[3]

```
    Method                   Time             Factor
    ====================     ============     ======
    basic python:            6.887 seconds -> 43.88
   xrange python:            6.710 seconds -> 42.75
    naive cython:            5.133 seconds -> 32.70
    typed cython:            1.845 seconds -> 11.75
C (no opt flags):            0.249 seconds ->  1.58
         C (-O3):            0.158 seconds ->  1.00
         C (-O2):            0.158 seconds ->  1.00
    cmath cython:            0.157 seconds ->  1.00
    weave python:            0.157 seconds ->  1.00
```

[3] The script `bench/benchmark.py` produces this table. These numbers are produced on Mac Book Air, Sept 2012, Mac OS X 10.8.1, 1.7 GHz Intel Core i5.

- Optimised Cython code can run as fast as optimised C code.
- (Note: the order of the last four lines can vary from run to run – they are essentially of the same speed.)

How can we write even faster code?

- With compiled C code, we have pretty much done as much as we can to make our code fast.

  (Not entirely true: FEEG6002 will teach more details on serial optimisation.)

- To gain further speed-up, we need to parallelise using

  ▷ multiple cores per CPU
  ▷ multiple CPUs per board
  ▷ multiple machines connected
  ▷ GPU cards (that host hundreds of cores)
  ▷ emerging hybrid architectures

GPUs

- GPUs are programmed in specialised languages. Well known are CUDA (Nvidia-specific) and OpenCL (generic).
- GPUs have many computational cores but not a lot of GPU RAM (typically 2-4GB max at the moment)
- cores are fast, but communication to CPU RAM is very slow
- if problem fits into GPU RAM, speed ups of factor 10 to 100 can be achieved

GPU computing example: OpenCL code

Timings taken on a Ubuntu Linux machine with a GeForce GTX 560 Ti GPU with 256 cores, and Intel Core2 CPU 6420 @ 2.13GHz 2, using double floats on GPU ($n = 10^7$):

```
        xrange python:       5.6759 seconds ->  897.104
         basic python:       5.6001 seconds ->  885.130
         naive cython:       4.3711 seconds ->  690.881
         typed cython:       1.7268 seconds ->  272.923
     C (no opt flags):       0.4600 seconds ->   72.705
              C (-O2):       0.3700 seconds ->   58.480
         cmath cython:       0.2744 seconds ->   43.364
         weave python:       0.2685 seconds ->   42.435
              C (-O3):       0.2600 seconds ->   41.094
PyOpenCL inc transfer:       0.0072 seconds ->    1.137
PyOpenCL exc transfer:       0.0063 seconds ->    1.000
```

- we can gain another factor ≈ 50 if we run the calculation on the GPU.

- The card used here (GeForce GTX 560 Ti GPU) costs about £150 (summer 2012).

- If we use single precision floats on the GPU (instead of double), the execution time is reduced by another factor ≈ 2.

Acknowlegdements: Justin Diviney has written and provided the OpenCL code (bench/OCL.py) used in these timings, and helped with the Cython and benchmarking programme.

PyOpenCL example

- There is a very convenient tool called PyOpenCL that helps with writing OpenCL code: only the parts ("kernels") of the calculation that need be fast are written in OpenCL; the surrounding glue is in Python. For example:

bench/pyopencl–example.py

```
import numpy
import pyopencl as cl

# create host side vectors
A_host = numpy.array([2, 4, 6, 5, 0, 8, 3], dtype = numpy.float32)
B_host = numpy.array([1, 1, 0, 5, 6, 3, 1], dtype = numpy.float32)
C_host = numpy.array([0, 0, 0, 0, 0, 0, 0], dtype = numpy.float32)

# OpenCL C source code for parallel addition of two vectors A and B
kernel = """
__kernel void vector_add(__global float* A,
                         __global float* B,
                         __global float* C)
{
    // each kernel instance has a different global id
    int i = get_global_id(0);
    C[i] = A[i] + B[i];
}
"""

# OpenCL routines and objects
context = cl.create_some_context()
queue = cl.CommandQueue(context)
program = cl.Program(context, kernel).build()

# create device side vectors and copy values from host to device memory
A_dev = cl.Buffer(context, cl.mem_flags.COPY_HOST_PTR, hostbuf = A_host)
B_dev = cl.Buffer(context, cl.mem_flags.COPY_HOST_PTR, hostbuf = B_host)
C_dev = cl.Buffer(context, cl.mem_flags.COPY_HOST_PTR, hostbuf = C_host)

# run the kernel (see string sourceCode above)
program.vector_add(queue, A_host.shape, None, A_dev, B_dev, C_dev)

# enqueue data transfer from device to host memory
cl.enqueue_read_buffer(queue, C_dev, C_host).wait()

print(C_host)

# Maxim Skripnik, Hans Fangohr, Southampton 2012
```

when run produces the output

```
fangohr@osiris:$ python pyopencl-example.py
[  3.    5.    6.   10.    6.   11.    4.]
```

PyOpenCL homepage: http://mathema.tician.de/software/pyopencl

Tidying up a few loose ends in C programming

- writing to files
- reading from files
- interchangealbe nature of `*a` and `a[]`
- parsing command line arguments

Writing to files

c/write_to_file.c

```c
 1 #include<stdio.h>
 2 #include<time.h>
 3 int main(void) {
 4   FILE *f;             /* pointer to file */
 5   if ( (f=fopen("myfile.txt", "w"))==NULL) {
 6     printf("Cannot open 'myfile.txt' for writing");
 7     return -1;
 8   }
 9   fprintf(f,"We can now print to the file f using Fprintf\n");
10   fprintf(f,"For example, a number %d and %d and %d, and\n", 1, 2, 42);
11 
12   if  (fclose(f) != 0 ) {
13     fprintf(f,"File could not be closed.\n");
14     return -1;
15   }
16   return 0;
17 }
```

- Line 5: attempt to open a file for Writing
- if successful, `fopen` returns a pointer different from `NULL`

- line 7: return -1 to OS to signal failure if we couldn't open file
- line 12: `fclose` returns 0 if successful

Note: There are other ways of writing data to a stream, including `fputc` to write a character.

Reading from files

c/read_from_file.c

```c
#include<stdio.h>
#include<stdlib.h>
int main(void) {
  FILE *f;                      /* pointer to file */
  char c;
  if ( (f=fopen("myfile.txt","r"))==NULL) {
    printf("Cannot open 'myfile.txt' for reading");
    return -1;
  }
  /* read file content char by char and print to stdout */
  while ( (c=fgetc(f)) != EOF) {
    printf("%c", c);
  }
  if  (fclose(f) != 0 ) {
    printf("File could not be closed.\n");
    return -1;
  }
  return 0;
}
```

Note: There are other ways of reading data, including `fscanf`.

`a*` and `a[]`

Can use `*a` and `a[]` interchangeably in function parameter definitions:

c/array–and–pointer.c

```c
#include <stdio.h>

void print_array1( int a[]) {
  int i;
  for (i=0; i<5; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
}

void print_array2( int *a) {
  int i;
  for (i=0; i<5; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
}

int main(void) {
  int a[5];
  int i;
  for (i=0; i<5; i++) a[i]=i*i;
  print_array1(a);
  print_array2(a);
  return 0;
}
```

Output:

```
0 1 4 9 16
0 1 4 9 16
```

Reading command line arguments

The `main()` function in general takes two intput arguments: an `int argc` which provides the total number of arguments, and `char *argv[]`, an array of pointers to strings (i.e. to an array of chars).

c/args.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for (i=0; i<argc; i++) {
    printf ("Argument %d = '%s'\n", i, argv[i]);
  }
  return 0;
}
```

```
$ ./args This is a test
Argument 0 = './args'
Argument 1 = 'This'
Argument 2 = 'is'
Argument 3 = 'a'
Argument 4 = 'test'
```

Dynamic memory allocation for 1d, 2d and 3d arrays (Version 1)

The following code has kindly been provided by Dr Ian Bush, NAG http://www.nag.co.uk/about/ibush.asp to demonstrate how to allocate memory for 1d, 2d and 3d arrays so that it is arranged *contiguously* in memory.

The following test program demonstrates how to use the routines in `c/array_alloc.c`.

First, we compile `array_alloc.c` into an object file (`array_alloc.o`)

```
gcc -ansi -pedantic -Wall -c -o array_alloc.o array_alloc.c
```

and then we compile the executable `array_alloc_demo` using this object file `array_alloc.o`

```
gcc -ansi -pedantic -Wall \
    -o array_alloc_demo array_alloc.o array_alloc_demo.c
```

c/array_alloc_demo.c

```c
/* Simple example program to show use of the array_alloc routines */
/* Ian Bush, 2010 for SESG6025 */

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

#include "array_alloc.h"

#define NROWS 3
#define NCOLS 5

int main( void ) {

  float **array_float;
  char  **array_char;

  double ***t;

  int i, j, k;

  /* First we will deal with a two dimensional array of floats */

  /* Allocate a NROWS by NCOLS array */
  array_float = alloc_2d_float( NROWS, NCOLS );

```

```
/* Check the allocation worked */
if( array_float == NULL )
  /* It failed! Exit the program */
  exit( EXIT_FAILURE );

/* Zero the array - this shows how to access the elements */
for( j = 0; j < NROWS; j++ )
  for( i = 0; i < NCOLS; i++ )
    array_float[ j ][ i ] = 0.0;

/* Print out the adress of each element of array
   Note how the 4 byte floats are contiguous in memory */
printf("Addresses of the elements of the %i by %i two dimensional array:\n",
        NROWS, NCOLS );
for( j = 0; j < NROWS; j++ )
  for( i = 0; i < NCOLS; i++ )
    printf( "a[ %i ][ %i ] is at %lu\n", j, i,
             ( unsigned long ) &( array_float[ j ][ i ] ) );
printf( "Note that the %i byte floats are contiguous in memory\n",
        (int) sizeof( **array_float ) );
printf( "\n" );

/* Now do exactly the same, but this time with chars */

/* Allocate a NROWS by NCOLS array */
array_char = alloc_2d_char( NROWS, NCOLS );
```

```

  /* Check the allocation worked */
  if( array_char == NULL )
    /* It failed! Exit the program */
    exit( EXIT_FAILURE );

  /* Zero the array - this shows how to access the elements */
  for( j = 0; j < NROWS; j++ )
    for( i = 0; i < NCOLS; i++ )
      array_char[ j ][ i ] = 0.0;

  /* Print out the adress of each element of array
     Note how the 1 byte chars are contiguous in memory */
  printf("Addresses of the elements of the %i by %i two dimensional array:\n",
          NROWS, NCOLS );
  for( j = 0; j < NROWS; j++ )
    for( i = 0; i < NCOLS; i++ )
      printf( "a[ %i ][ %i ] is at %lu\n", j, i,
              ( unsigned long ) &( array_char[ j ][ i ] ) );
  printf( "Note that the %i byte chars are contiguous in memory\n",
          (int) sizeof( **array_char ) );
  printf( "\n" );


  /* Finally a 3 dimensional example */
  t = alloc_3d_double( 2, 3, 4 );
  if( t == NULL )
    exit( EXIT_FAILURE );
  printf( "Addresses of the elements ");
  printf( "of the %i by %i by %i three dimensional array of doubles:\n",
          2, 3, 4 );
  for( k = 0; k < 2; k++ )
    for( j = 0; j < 3; j++ )
      for( i = 0; i < 4; i++ )
        printf( "a[ %i ][ %i ][ %i ] is at %lu\n", k, j, i,
                ( unsigned long ) &( t[ k ][ j ][ i ] ) );

  return EXIT_SUCCESS;
}
```

The output from the program reads:

```
Addresses of the elements of the 3 by 5 two dimensional array:
a[ 0 ][ 0 ] is at 140408843680272
a[ 0 ][ 1 ] is at 140408843680276
a[ 0 ][ 2 ] is at 140408843680280
a[ 0 ][ 3 ] is at 140408843680284
a[ 0 ][ 4 ] is at 140408843680288
a[ 1 ][ 0 ] is at 140408843680292
a[ 1 ][ 1 ] is at 140408843680296
a[ 1 ][ 2 ] is at 140408843680300
a[ 1 ][ 3 ] is at 140408843680304
```

```
a[ 1 ][ 4 ] is at 140408843680308
a[ 2 ][ 0 ] is at 140408843680312
a[ 2 ][ 1 ] is at 140408843680316
a[ 2 ][ 2 ] is at 140408843680320
a[ 2 ][ 3 ] is at 140408843680324
a[ 2 ][ 4 ] is at 140408843680328
Note that the 4 byte floats are contiguous in memory

Addresses of the elements of the 3 by 5 two dimensional array:
a[ 0 ][ 0 ] is at 140408843680368
a[ 0 ][ 1 ] is at 140408843680369
a[ 0 ][ 2 ] is at 140408843680370
a[ 0 ][ 3 ] is at 140408843680371
a[ 0 ][ 4 ] is at 140408843680372
a[ 1 ][ 0 ] is at 140408843680373
a[ 1 ][ 1 ] is at 140408843680374
a[ 1 ][ 2 ] is at 140408843680375
a[ 1 ][ 3 ] is at 140408843680376
a[ 1 ][ 4 ] is at 140408843680377
a[ 2 ][ 0 ] is at 140408843680378
a[ 2 ][ 1 ] is at 140408843680379
a[ 2 ][ 2 ] is at 140408843680380
a[ 2 ][ 3 ] is at 140408843680381
a[ 2 ][ 4 ] is at 140408843680382
Note that the 1 byte chars are contiguous in memory
```

```
Addresses of the elements of the 2 by 3 by 4 three dimensional array of doubles:
a[ 0 ][ 0 ][ 0 ] is at 140408843680384
a[ 0 ][ 0 ][ 1 ] is at 140408843680392
a[ 0 ][ 0 ][ 2 ] is at 140408843680400
a[ 0 ][ 0 ][ 3 ] is at 140408843680408
a[ 0 ][ 1 ][ 0 ] is at 140408843680416
a[ 0 ][ 1 ][ 1 ] is at 140408843680424
a[ 0 ][ 1 ][ 2 ] is at 140408843680432
a[ 0 ][ 1 ][ 3 ] is at 140408843680440
a[ 0 ][ 2 ][ 0 ] is at 140408843680448
a[ 0 ][ 2 ][ 1 ] is at 140408843680456
a[ 0 ][ 2 ][ 2 ] is at 140408843680464
a[ 0 ][ 2 ][ 3 ] is at 140408843680472
a[ 1 ][ 0 ][ 0 ] is at 140408843680480
a[ 1 ][ 0 ][ 1 ] is at 140408843680488
a[ 1 ][ 0 ][ 2 ] is at 140408843680496
a[ 1 ][ 0 ][ 3 ] is at 140408843680504
a[ 1 ][ 1 ][ 0 ] is at 140408843680512
a[ 1 ][ 1 ][ 1 ] is at 140408843680520
a[ 1 ][ 1 ][ 2 ] is at 140408843680528
a[ 1 ][ 1 ][ 3 ] is at 140408843680536
a[ 1 ][ 2 ][ 0 ] is at 140408843680544
a[ 1 ][ 2 ][ 1 ] is at 140408843680552
a[ 1 ][ 2 ][ 2 ] is at 140408843680560
a[ 1 ][ 2 ][ 3 ] is at 140408843680568
```

Array allocation tools header file

c/array_alloc.h

```


/* 1d routines */
char   *alloc_1d_char  ( int ndim1 );
int    *alloc_1d_int   ( int ndim1 );
float  *alloc_1d_float ( int ndim1 );
double *alloc_1d_double( int ndim1 );

/* 2d routines */
char   **alloc_2d_char  ( int ndim1, int ndim2 );
int    **alloc_2d_int   ( int ndim1, int ndim2 );
float  **alloc_2d_float ( int ndim1, int ndim2 );
double **alloc_2d_double( int ndim1, int ndim2 );

/* 3d routines */
char   ***alloc_3d_char  ( int ndim1, int ndim2, int ndim3 );
int    ***alloc_3d_int   ( int ndim1, int ndim2, int ndim3 );
float  ***alloc_3d_float ( int ndim1, int ndim2, int ndim3 );
double ***alloc_3d_double( int ndim1, int ndim2, int ndim3 );
```

Array allocation tools implementation

c/array_alloc.c

```

/*
   Simple wrappers for allocating arrays so that the elements are
   contiguous in memory

   The following data types are supported: char, int, float, double

   The following dimensionality of arrays is supported: 1, 2, 3

   The naming convention for routines is

   alloc_<dimensionality>_<type>

   where <dimensionality is of the form nd, where n is the number of
   dimensions, and <type> is the data type

   Each function returns an pointer with the appropriate degre of
   indirectino for the dimensionality required . On error NULL is
   returned.

   The arguments to the functions are the dimensions in natural order.
```

```

  Ian Bush, 2010 for SESG6025, University of Southampton,
  very minor modifications by Hans Fangohr.

*/

#include <stdlib.h>

char *alloc_1d_char( int ndim1 ) {
  return malloc( ndim1 * sizeof( char ) );
}

int *alloc_1d_int( int ndim1 ) {
  return malloc( ndim1 * sizeof( int ) );
}

float *alloc_1d_float( int ndim1 ) {
  return malloc( ndim1 * sizeof( float ) );
}

double *alloc_1d_double( int ndim1 ) {
  return malloc( ndim1 * sizeof( double ) );
}

char  **alloc_2d_char ( int ndim1, int ndim2 ) {
  char  **array2 = malloc( ndim1 * sizeof( char  * ) );
  int i;
  if( array2 != NULL ){
    array2[0] = malloc( ndim1 * ndim2 * sizeof( char  ) );
    if( array2[ 0 ] != NULL ) {
      for(i = 1; i < ndim1; i++)
        array2[i] = array2[0] + i * ndim2;
    }
    else {
      free( array2 );
      array2 = NULL;
    }
  }
  return array2;
}

int **alloc_2d_int( int ndim1, int ndim2 ) {
  int **array2 = malloc( ndim1 * sizeof( int * ) );
  int i;
  if( array2 != NULL ){
    array2[0] = malloc( ndim1 * ndim2 * sizeof( int ) );
    if( array2[ 0 ] != NULL ) {
      for(i = 1; i < ndim1; i++)
        array2[i] = array2[0] + i * ndim2;
    }
    else {
      free( array2 );
```

```
        array2 = NULL;
      }
    }
    return array2;
}

float **alloc_2d_float( int ndim1, int ndim2 ) {
    float **array2 = malloc( ndim1 * sizeof( float * ) );
    int i;
    if( array2 != NULL ){
      array2[0] = malloc( ndim1 * ndim2 * sizeof( float ) );
      if( array2[ 0 ] != NULL ) {
        for(i = 1; i < ndim1; i++)
          array2[i] = array2[0] + i * ndim2;
      }
      else {
        free( array2 );
        array2 = NULL;
      }
    }
    return array2;
}

double **alloc_2d_double( int ndim1, int ndim2 ) {
    double **array2 = malloc( ndim1 * sizeof( double * ) );
    int i;
```

```
  if( array2 != NULL ){
    array2[0] = malloc( ndim1 * ndim2 * sizeof( double ) );
    if( array2[ 0 ] != NULL ) {
      for(i = 1; i < ndim1; i++)
        array2[i] = array2[0] + i * ndim2;
    }
    else {
      free( array2 );
      array2 = NULL;
    }
  }
  return array2;
}

char   ***alloc_3d_char ( int ndim1, int ndim2, int ndim3 ) {
  char   *space = malloc( ndim1 * ndim2 * ndim3 * sizeof( char ) );
  char ***array3 = malloc( ndim1 * sizeof( char ** ) );
  int i, j;

  if( space == NULL || array3 == NULL )
    return NULL;

  for( j = 0; j < ndim1; j++ ) {
    array3[ j ] = malloc( ndim2 * sizeof( char * ) );
    if( array3[ j ] == NULL )
      return NULL;
```

```c
    for( i = 0; i < ndim2; i++ )
      array3[ j ][ i ] = space + j * ( ndim3 * ndim2 ) + i * ndim3;
  }
  return array3;
}

int   ***alloc_3d_int   ( int ndim1, int ndim2, int ndim3 ) {
  int   *space = malloc( ndim1 * ndim2 * ndim3 * sizeof( int   ) );
  int  ***array3 = malloc( ndim1 * sizeof( int   ** ) );
  int i, j;

  if( space == NULL || array3 == NULL )
    return NULL;

  for( j = 0; j < ndim1; j++ ) {
    array3[ j ] = malloc( ndim2 * sizeof( int * ) );
    if( array3[ j ] == NULL )
      return NULL;
    for( i = 0; i < ndim2; i++ )
      array3[ j ][ i ] = space + j * ( ndim3 * ndim2 ) + i * ndim3;
  }
  return array3;
}

float  ***alloc_3d_float ( int ndim1, int ndim2, int ndim3 ) {
  float  *space = malloc( ndim1 * ndim2 * ndim3 * sizeof( float ) );
  float  ***array3 = malloc( ndim1 * sizeof( float ** ) );
  int i, j;

  if( space == NULL || array3 == NULL )
    return NULL;

  for( j = 0; j < ndim1; j++ ) {
    array3[ j ] = malloc( ndim2 * sizeof( float * ) );
    if( array3[ j ] == NULL )
      return NULL;
    for( i = 0; i < ndim2; i++ )
      array3[ j ][ i ] = space + j * ( ndim3 * ndim2 ) + i * ndim3;
  }
  return array3;
}

double ***alloc_3d_double( int ndim1, int ndim2, int ndim3 ) {
  double *space = malloc( ndim1 * ndim2 * ndim3 * sizeof( double) );
  double ***array3 = malloc( ndim1 * sizeof( double** ) );
  int i, j;

  if( space == NULL || array3 == NULL )
    return NULL;

  for( j = 0; j < ndim1; j++ ) {
    array3[ j ] = malloc( ndim2 * sizeof( double* ) );
    if( array3[ j ] == NULL )
      return NULL;
    for( i = 0; i < ndim2; i++ )
      array3[ j ][ i ] = space + j * ( ndim3 * ndim2 ) + i * ndim3;
  }
  return array3;
}
```

Dynamic memory allocation for 1d, 2d and 3d arrays (Version 2)

The following code has kindly been provided by Dr David Henty, EPCC https://www.epcc.ed.ac.uk/about/staff/dr-david-henty to demonstrate how to allocate memory for 1d, 2d and 3d arrays so that it is arranged *contiguously* in memory.

The code works slighly differently to what we have seen on slide 210: (i) this works for arrays of arbitrary dimension and (ii) only a single call of the `free` function is required to de-allocate the memory. However, the code is more complex.

The following test program demonstrates how to use the routines in `c/arralloc/arralloctest.c`.

First, we compile `arralloc.c` into an object file (`arralloc.o`)

```
gcc -ansi -pedantic -Wall -c -o arralloc.o arralloc.c
```

and then we compile the executable `arralloctest` using this object file `arrayalloc.o`

```
gcc -ansi -pedantic -Wall \
    -o arralloctest arralloc.o arralloctest.c
```

c/arralloc/arralloctest.c

```c
/* Code provided by David Henty, EPCC David Henty,
The University of Edinburgh,
HPC Training and Support, Edinburgh EH9 3JZ, UK,
for FEEG6002 at the University of Southampton, 2014 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#include "arralloc.h"

void printlinearbuffer(float *x, int n);
void print3Darray(float ***x, int ni, int nj, int nk);

int main(void)
{
  int nx, ny, nz;
  int i, j, k;

  float ***array;

  nx = 2;
  ny = 4;
  nz = 3;

  array = arralloc(sizeof(float), 3, nx, ny, nz);

  for (i=0; i < nx; i++)
    {
      for (j=0; j < ny; j++)
        {
          for (k=0; k < nz; k++)
            {
              array[i][j][k] = k + j*nz + i*nz*ny;
            }
        }
    }

  printf("array[%d][%d][%d] = %f\n", nx/2,ny/2,nz/2, array[nx/2][ny/2][nz/2]);

  /*
   *  The following call is correct - pass address of first data element
   */

  printf("\nArray passed correctly to a function expecting a linear buffer\n\n");

  printlinearbuffer(&(array[0][0][0]), nx*ny*nz);

  /*
   *  The following call is INCORRECT - you should NOT pass the array itself
   */

  printf("\nArray passed INCORRECTLY to a function expecting a linear buffer\n\n");

  printlinearbuffer((float *) array, nx*ny*nz);

  /*
   *  The following call is correct - can pass address of array provided that
   *  the function being called expects a 3D array and not a linear buffer
   */

  printf("\nArray passed correctly to a function expecting a 3D array\n\n");

  print3Darray(array, nx, ny, nz);

  /*
   *  Entire multidimensional array is deallocated with a single free
   */

  free(array);
}

void printlinearbuffer(float *x, int n)
{
  int i;

  for (i=0; i < n; i++)
```

```c
    {
      printf("x[%d] = %f\n", i, x[i]);
    }
}

void print3Darray(float ***x, int ni, int nj, int nk)
{
  int i, j, k;

  for (i=0; i < ni; i++)
    {
    for (j=0; j < nj; j++)
      {
      for (k=0; k < nk; k++)
        {
          printf("x[%d][%d][%d] = %f\n", i, j, k, x[i][j][k]);
        }
      }
    }
}
```

The output from the program reads:

```
array[1][2][1] = 19.000000

Array passed correctly to a function expecting a linear buffer
```

```
x[0] = 0.000000
x[1] = 1.000000
x[2] = 2.000000
x[3] = 3.000000
x[4] = 4.000000
x[5] = 5.000000
x[6] = 6.000000
x[7] = 7.000000
x[8] = 8.000000
x[9] = 9.000000
x[10] = 10.000000
x[11] = 11.000000
x[12] = 12.000000
x[13] = 13.000000
x[14] = 14.000000
x[15] = 15.000000
x[16] = 16.000000
x[17] = 17.000000
x[18] = 18.000000
x[19] = 19.000000
x[20] = 20.000000
x[21] = 21.000000
x[22] = 22.000000
x[23] = 23.000000
```

```
Array passed INCORRECTLY to a function expecting a linear buffer

x[0] = -443251030725245272064.000000
x[1] = 0.000000
x[2] = -443252156625152114688.000000
x[3] = 0.000000
x[4] = -443253282525058957312.000000
x[5] = 0.000000
x[6] = -443253704737524023296.000000
x[7] = 0.000000
x[8] = -443254126949989089280.000000
x[9] = 0.000000
x[10] = -443254549162454155264.000000
x[11] = 0.000000
x[12] = -443254971374919221248.000000
x[13] = 0.000000
x[14] = -443255393587384287232.000000
x[15] = 0.000000
x[16] = -443255815799849353216.000000
x[17] = 0.000000
x[18] = -443256238012314419200.000000
x[19] = 0.000000
x[20] = 0.000000
x[21] = 1.000000
x[22] = 2.000000
x[23] = 3.000000
```

```
Array passed correctly to a function expecting a 3D array

x[0][0][0] = 0.000000
x[0][0][1] = 1.000000
x[0][0][2] = 2.000000
x[0][1][0] = 3.000000
x[0][1][1] = 4.000000
x[0][1][2] = 5.000000
x[0][2][0] = 6.000000
x[0][2][1] = 7.000000
x[0][2][2] = 8.000000
x[0][3][0] = 9.000000
x[0][3][1] = 10.000000
x[0][3][2] = 11.000000
x[1][0][0] = 12.000000
x[1][0][1] = 13.000000
x[1][0][2] = 14.000000
x[1][1][0] = 15.000000
x[1][1][1] = 16.000000
x[1][1][2] = 17.000000
x[1][2][0] = 18.000000
x[1][2][1] = 19.000000
x[1][2][2] = 20.000000
x[1][3][0] = 21.000000
x[1][3][1] = 22.000000
```

```
x[1][3][2] = 23.000000
```

Array allocation tools header file

c/arralloc/arralloc.h

```
/* Code provided by David Henty, EPCC David Henty, The University of Edinburgh,
HPC Training and Support, Edinburgh EH9 3JZ, UK,
for FEEG6002 at the University of Southampton, 2014 */

void *arralloc(size_t size, int ndim, ...);
```

Array allocation tools implementation

c/arralloc/arralloc.c

```
/*****************************************************************************
 * Alloc         Interface functions to dynamic store allocators.           *
 * arralloc()    Allocate rectangular dope-vector (ie using pointers) array *
 *****************************************************************************/

/* Code provided by David Henty, EPCC David Henty, The University of Edinburgh,
HPC Training and Support, Edinburgh EH9 3JZ, UK,
for FEEG6002 at the University of Southampton, 2014 */

/*========================== Library include files ===========================*/
#include <stddef.h>
#include <stdarg.h>
/*##include <malloc.h> */
#include <stdlib.h>
/*========================== Library declarations ============================*/
/* char *calloc(); */
/* char *malloc(); */
/*========================== External function declarations ==================*/
#ifdef  DEBUG
int     malloc_verify();
int     malloc_debug();
```

```
#endif
/***************************************************************************
 *  ~arralloc.  Allocate a psuedo array of any dimensionality and type with   *
 *  specified size for each dimension.  Each dimension is                    *
 *  an array of pointers, and the actual data is laid out in standard 'c'    *
 *  fashion ie last index varies most rapidly.  All storage is got in one    *
 *  block, so to free whole array, just free the pointer array.             *
 *  array = (double***) arralloc(sizeof(double), 3, 10, 12, 5);             *
 ***************************************************************************/

/* ALIGN returns the next b byte aligned address after a */
#define ALIGN(a,b)       (int*)( (((long)(a) + (b) - 1)/(b))*(b) )

/* If on an I860 align arrays to cache line boundaries */
#ifdef I860
#define MIN_ALIGN 32
#else
#define MIN_ALIGN 1
#endif

/*---------------------------------------------------------------------*/


void    subarray(align_size, size, ndim, prdim, pp, qq, dimp, index)
size_t  align_size;     /* size of object to align the data on */
size_t  size;           /* actual size of objects in the array */
```

```
int       ndim, prdim;     /* ndim- number of dim left to do */
                           /* prdim - no of pointers in previous iteration */
void      ***pp, **qq;     /* pp - pointer to previous level of the array */
                           /* qq - pointer to start of this level */
int *dimp, index;
{
   int  *dd = ALIGN(qq,align_size);     /*aligned pointer only used in last recursion
   int  **dpp = (int**)pp;
   int i,       dim = dimp[index];

   if(ndim > 0)            /* General case - set up pointers to pointers  */
   {
      for( i = 0; i < prdim; i++)
         pp[i] = qq + i*dim;     /* previous level points to us */

      subarray(align_size, size, ndim-1, prdim*dim,
                          (void***)qq,    /* my level filled in next */
                          qq+prdim*dim,   /* next level starts later */
                          dimp, (index+1) );
   }
   else                    /* Last recursion - set up pointers to data   */
      for( i = 0; i < prdim; i++)
         dpp[i] = dd + (i*dim)*size/sizeof(int);
}
```

```c

/*---------------------------------------------------------------------*/

/*
 * if REFS is defined the va macros are dummied out. This is because the
 * GreenHills va_arg macro will not get past the cref utility.
 * This way the call tree can still be constructed. Do NOT under
 * any circumstance define REFS when compiling the code.
 */

#if REFS
   #undef va_start
   #undef va_arg
   #undef va_end
   #undef va_list
   #define va_list int
   #define va_start( A , B) ( A = (int) B)
   #define va_end( A ) ( A = 0 )
   #define va_arg( A , T ) ( A = (T) 0)
#endif

void *arralloc(size_t size, int ndim, ...)
{
   va_list      ap;
   void         **p, **start;
   int          idim;
```

```
    long            n_ptr = 0, n_data = 1;
    int             *dimp;
    size_t          align_size;

    va_start(ap, ndim);

    if( size % sizeof(int) != 0 )  /* Code only works for 'word' objects */
       return 0;
    /* we want to align on both size and MIN_ALIGN */
    if( size > MIN_ALIGN )
    {
         align_size = size;
    }
    else
    {
         align_size = MIN_ALIGN;
    }
    while( (align_size % size) || (align_size % MIN_ALIGN) )
    {
         align_size++;
    }
    /*
     * Cycle over dims,  accumulate # pointers & data items.
     */
    if ( NULL == (dimp=(int *)malloc( ndim * sizeof(int) )))
         return 0;
```

```c

    for(idim = 0; idim < ndim; idim++)
    {
        dimp[idim] = va_arg(ap, int);
        n_data *= dimp[idim];
        if( idim < ndim-1 )
            n_ptr  += n_data;
    }
    va_end(ap);


    /*
     *  Allocate space  for pointers and data.
     */
    if( (start = (void**)malloc(
                    (size_t)((n_data*size)+align_size+(n_ptr*sizeof(void**)))) == 0)
        return 0;
    /*
     * Set up pointers to form dope-vector array.
     */
    subarray(align_size, size, ndim-1, 1, &p, start, dimp, 0);
    free( dimp );

    return (void*)p;
}
```

```
changeset:   785:fed77264ceb4
tag:         tip
user:        Hans Fangohr [MBA] <fangohr@soton.ac.uk>
date:        Sun Oct 02 20:27:52 2016 +0100
summary:     tidy up of white space, pre 2016-2017 delivery
```