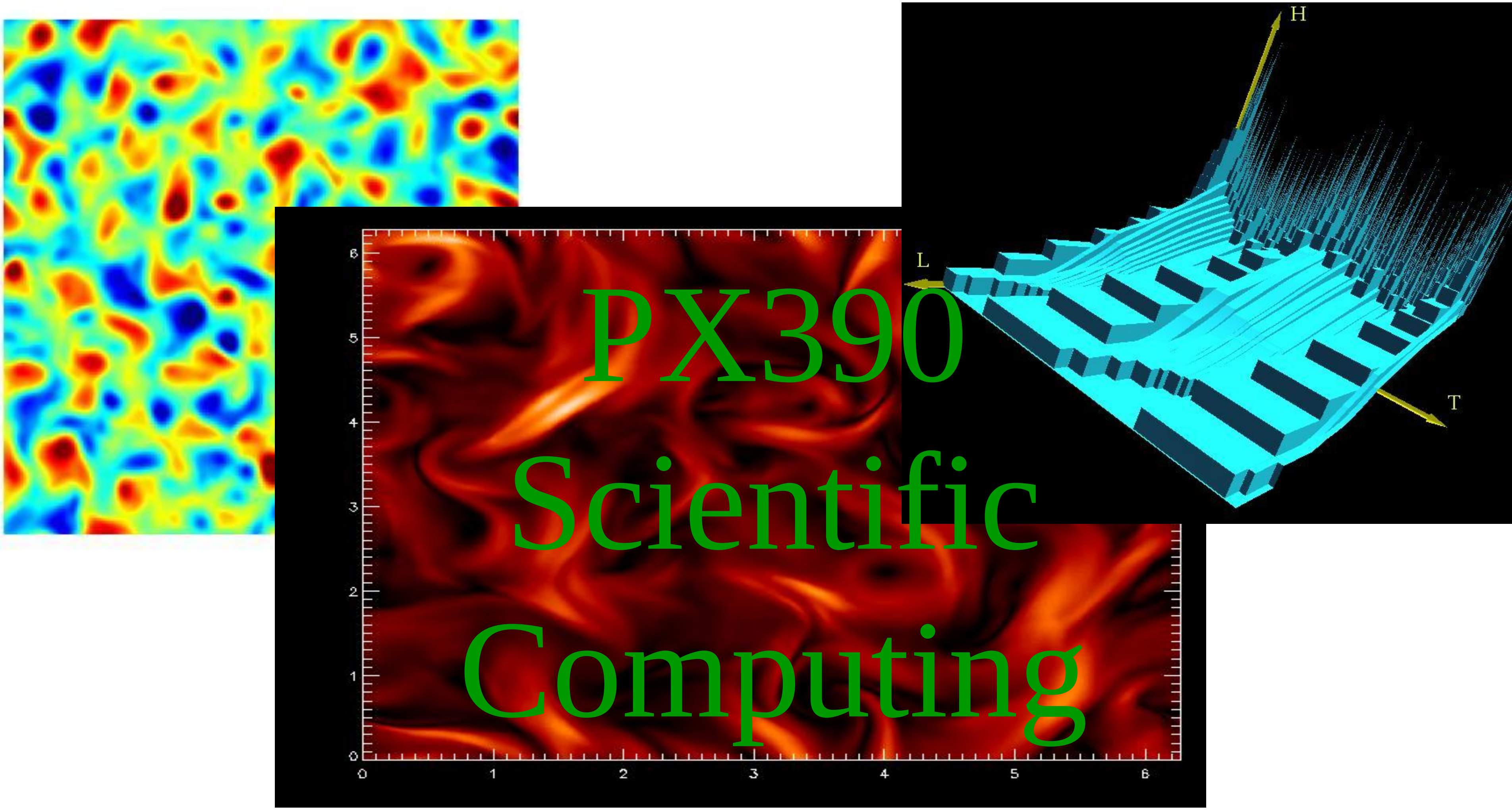


PX390 Scientific Computing

Basic structure of C program.

C Compiler

Python versus C

C is a compiled language while Python is interpreted, that is, it runs code interactively.

Python code development cycle:

Write → Run → Debug → Run

C code development cycle:

Write → Compile → Run → Debug →Compile → Run

– C compiler produces an executable program which is optimised for a native set of instructions on a given computer

– A debugger can be used to run C code ‘like an interpreter’.

Common concepts, similar syntax

Most programming languages share key constructs, such as ‘for’ loops and ‘if-else if-else’ flow control.

Python:

```
nt = 5
j = 1
for i in range(0, nt):
    j = j * i
```

C:

```
int i, j = 1;
int nt = 5;
for(i = 0; i < nt; i++)
{
    j = j * i;
}
```

- **Variables must be declared to be of specific type in C**
- Loop sets the initial value of index i, checks if it is within the valid range and increments it by1 in each iteration
- The code to be iterated in the loop is enclosed in {…}
- **Semicolon needed at the end of each line**

Functions in C

C emphasises modular development of the code. Many tasks, such as sorting, are already included as a build-in functions.

C code **must** include a function `int main(),` containing the program to be executed.

Any function other than **`main()`** must be first declared.

The declaration specifies the types of arguments (here, v1 & v2) and of the return value (here, int).

It always needs a return value

```
/* Declaration (prototype) */
int IntProduct(int v1, int v2);

int main()
{
   . . .
   int p = IntProduct(2, 4);
   . . .
}
/* Definition */
int IntProduct(int v1, int v2)
{
    int prod = v1 * v2;
    return(prod);
}
```

C program structure: example

```
/************************************
** Program description
*************************************/
/* Standard libraries always go first */
#include <stdio.h>
#include <stdlib.h>
/* Preprocessor statements define constants */
#define myPI 3.141592;
/* Function is declared */
double SphereVolume (double radius);

int main() /* Program starts here…*/
{
     double radius = -1, volume = 0;

     printf("Input the radius\n");
     while (radius < 0) scanf("%lf", &radius);
     if(radius == 0) volume = 0.0;
     else volume = SphereVolume(radius);
     return(0);
}                                  /* and ends here…*/

/* Function is defined, it returns a number */
double SphereVolume (double radius)
{
     return(0.33333*myPI*radious*radius*radius);
}
```

Ground rules

• C source file starts with inclusion of **libraries, definitions** of constants**,** followed by **function declarations.** Functions are usually defined after the **`main()`** function.

• Program commands begin at **`main()`** which can take some arguments and returns a value of integer type

• **Keywords** (int, if, etc…) are written in lower-case; **C is case sensitive**

• Statements are terminated with a **semi-colon;**

• Strings are in **double quotes**, characters in **single quotes**

• Symbols {} mark the beginning / end of a **program block**

• Any text between /* and */ is treated as a **comment**

C compiler

- **Source code** is a **text file** which includes a program written in C, file name must have extension .c (for example, *myCprog.c*)

- C compiler is a computer program that **translates source code into a set of binary sequences that can be executed** on a given computer.

- The standard name for the executable C program is **a.out .** This can be changed using option -o*filename*.

- Each operating system requires different compiler because the binary sequences are different for each platform (Windows, Mac and Unix).

- We will use GNU compiler called **gcc**. When submitting a program for assessment make sure there are no compilation errors / warnings. Use **`gcc –Wall`** to see all the compilation warnings.

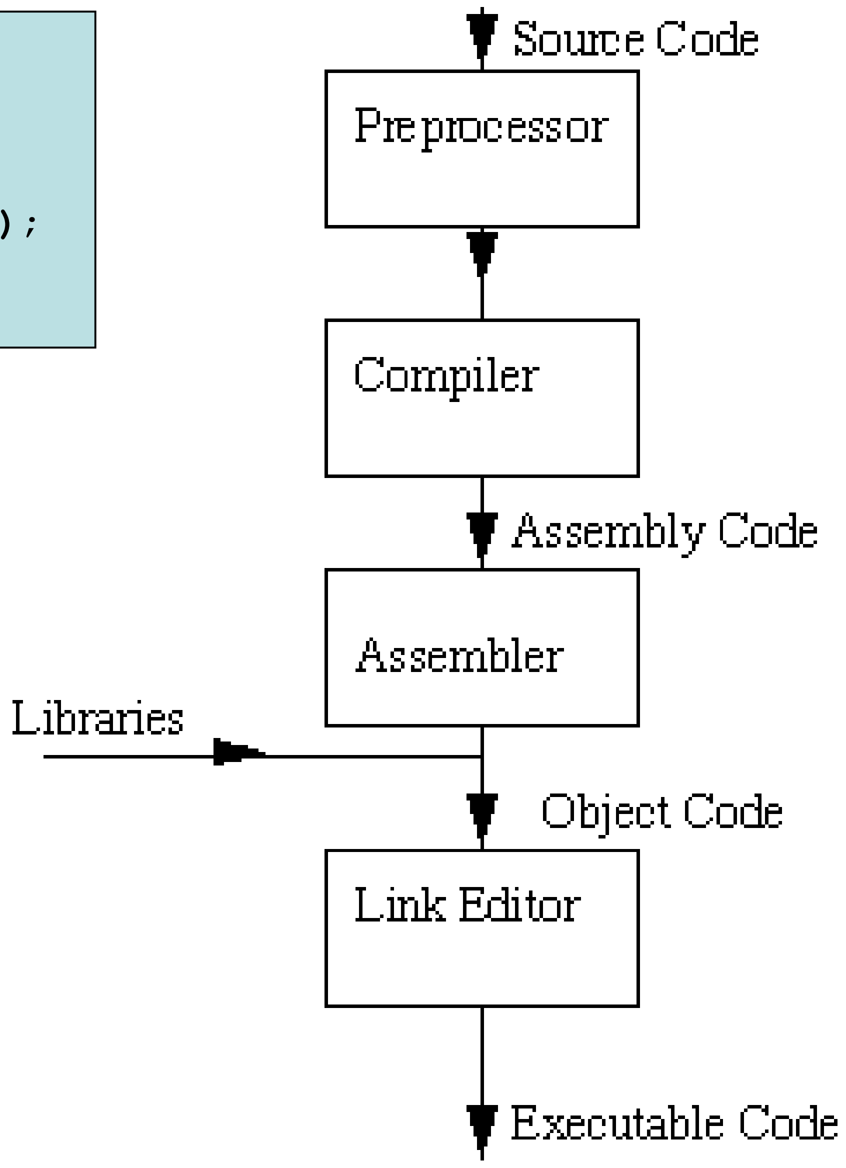
Compilation process

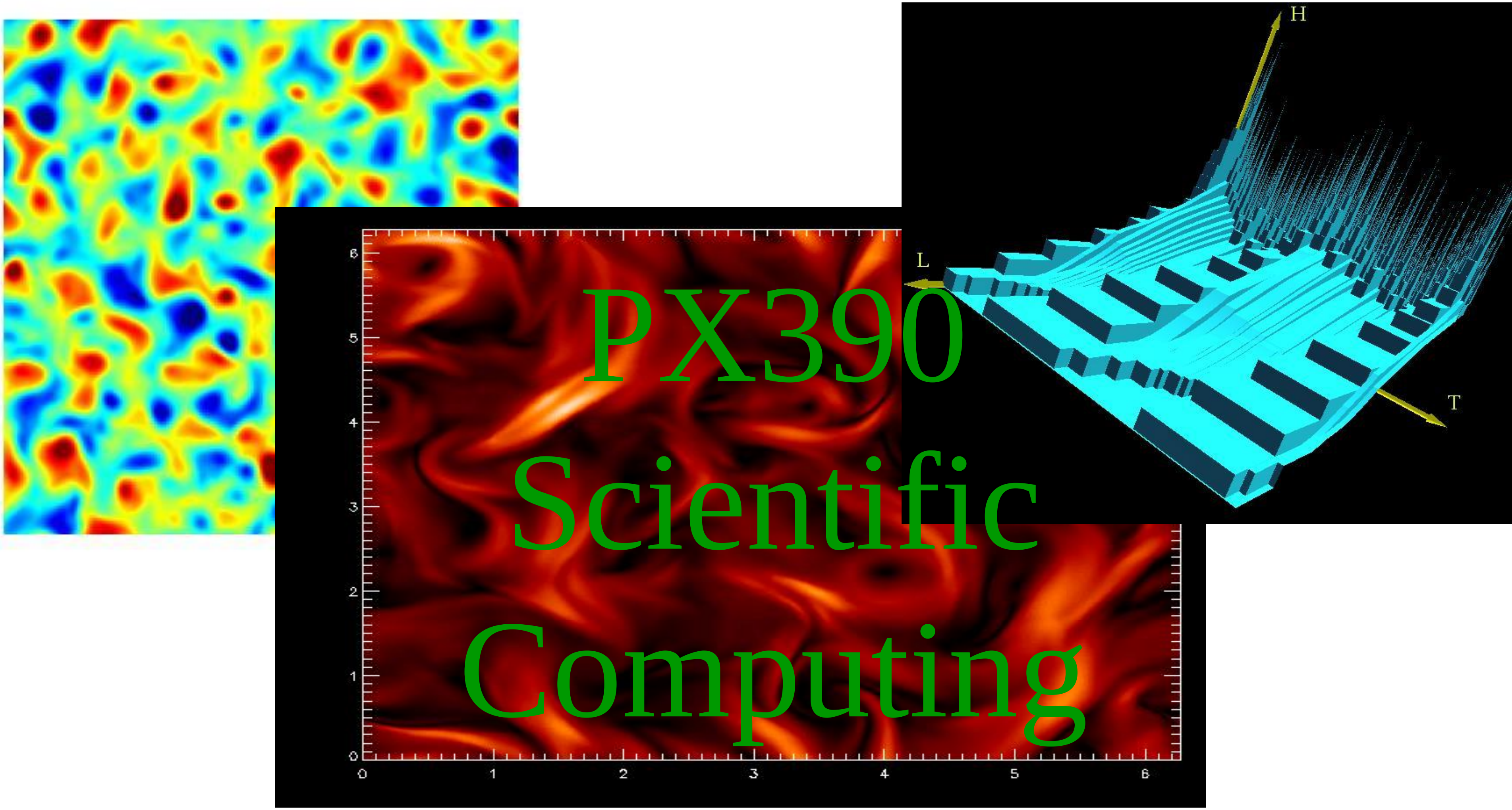
```
#include <stdio.h>

int main()
{
      printf (“Scientific Programming\n");
      return 0;
}
```

Compiler generates file *first*
unix> gcc –Wall -g –ofirst first.c

When executed program “*first*” generates the following output:
unix>./first
Scientific Programming
unix>





PX390 Scientific Computing

Variable types in C

Arrays

Data types

Data type is a fundamental concept of C language:

- any variable or function must be of specific type
- result of any expression depends on the types involved

C has three basic data types:

- **Integer: int, long int (preferred), short int, unsigned int**

A whole number within some range of mathematical integers, which may or may not allow negative values

- **Floating point: float, double (preferred)**

    Positive or negative rational number expressed as mantissa and exponent (0.mmmmm x 10^{ee}),

- **Character**

    Alphabet (upper/lower case), digits 0-9, +-%&, etc…
    Represented by ASCII code. Also escape characters.

Representation of numbers

- All numbers on the computer are represented by a finite number of bits. A **bit** is the smallest unit of memory allocation with a binary value of 0 or 1. A **Byte** is a sequence of 8 bits.

- Usually numerical types are 1, 2, 4 or 8 Bytes long, the same as 8, 16, 32 or 64 bits long.

- Example: 8 bits unsigned integers (short int): smallest value $0 = (00000000)_2$ and largest $255 = (11111111)_2$

- A number in integer representation (int, long) is **always exact**. Arithmetic between integers is always exact as long as the result is also an integer (integer division)

Negative integers: two's complement

Negative numbers 'wrap around'. First bit always indicate a sign, 0 corresponds to + and 1 corresponds to –

sign bit									
	1	0	0	1	0	1	1	1	**= -23**

Example: 8 bit signed int:

- bit string `00000001` represents 1
- bit string `01111111` represents 127
- bit string `10000000` represents -128
- bit string `10000001` represents -127
- bit string `11111111` represents -1

Is this operationally consistent? Check: add 1 to -1:

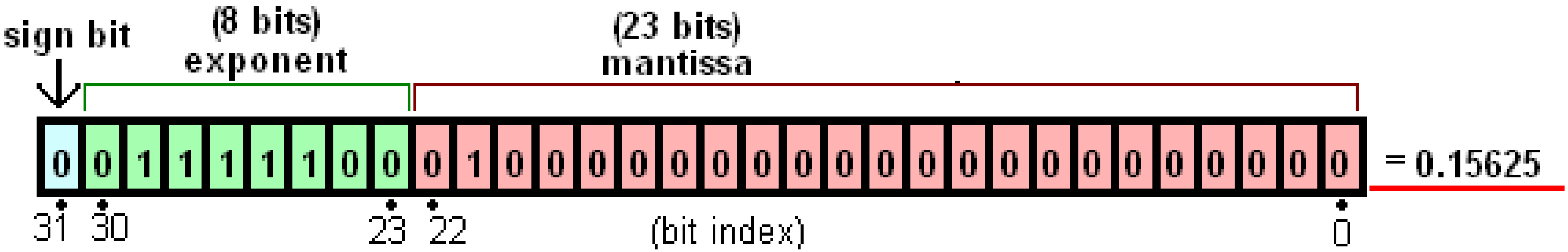
`00000001+11111111 = 100000000`

But this is now 9 bits, so ignore the first bit

`00000001+11111111 = 100000000 = 0`

Representation of real numbers

Same as 'scientific notation': Sign - Exponent - Fraction (mantissa)



Example: -5.25 = (binary) = -101.01 = $-1\bullet 2^2+0\bullet 2^1+1\bullet 2^0 + 0\bullet 2^{-1}+1\bullet 2^{-2}$

Normalised representation is 1.xxxxx • 2^n: -101.01 = -1.**0101** $\times 2^2$

Add 127 to the exponent: 127 + 2 =129 = (binary) = **10000001**

Full representation is: 1 | **10000001** | **0101**0000000000000000000

• Double precision, 64 bits: 1 bit sign, 11 bit exponent, 52 bits fraction.

• Modern languages include special values, +/- infinity and NaN.

• **Machine accuracy** ε_m: smallest floating point number which added to 1.0 produces result different than 1.0. This leads to a **round off error** in floating point arithmetic.

Floating point numbers

- **Not all floating point numbers can be represented exactly** using finite number of bits (32-float, 64-double)

- Exact comparisons may fail, **never compare two double type numbers directly**

- It is not guaranteed that 2.0 * 2.0 == 4.0 or that (a+b) + c == a + (b+c)

- Careful with constants, use library definitions e.g., π

- Operations among floating point variables are not exact, even if you use an exact analytical expressions. Results of arithmetic may depend on computer/compiler/optimisation used. Treat floating point numbers as if they have an 'error bar'.

- **Use double precision (and not single) by default.**

Type conversions

C converts float type variables to a double automatically before any operation (including passing to a function). All internal C functions return double precision numbers.

It is a good practice to explicitly cast your variables to the correct type if the resulting type is not obvious (changed).

Implicit 'casts'

- double f = (3/2);

    Results in f == 1.0 (integer division, result converted to double)

Explicit 'casts'

- double f = ((double) 3) / (double) 2);

    Results in f ==1.5 (division of two double numbers)

Precision may be lost converting floating point to integer or back again.

Arrays

An array type describes an object whose values are composed of **elements that have the same type.**

A single dimensional array of **N** elements and a specific type:

$$\texttt{type array\_name[N] = \{}a_0,\ a_1,\ a_2,\ldots,a_{N-1}\texttt{\};}$$

Two-dimensional arrays are declared in a similar way

$$\begin{aligned}\texttt{type array\_name[N][M] =}&\{a_{0,0},\ a_{0,1},\ a_{0,2},\ldots,a_{0,M-1}\},\\ &\{a_{1,0},\ a_{1,1},\ a_{1,2},\ldots,a_{1,M-1}\},\\ &\qquad\qquad *\\ &\{a_{N-1,0},\ a_{N-1,1},\ a_{N-1,2},\ldots,a_{N-1,M-1}\};\end{aligned}$$

- **Index is always zero based** (first element of any array a is a[0]).
- If a complete list of the initial values is specified, N can be omitted.

Often, the initial values are 0s and the array elements are assigned their desired values during calculations.

- **All types pack tightly in the memory when composed into arrays.**

Arrays: Examples

```
/* Array of characters */
static char name[ ] = { “John Smith” };

/* Two dimensional array of complex numbers */
double z[4096][2];
/* To assign number z= 1-2i to element 0 */
z[0][0] = 1;  z[0][1] = -2;
/* The last element */
z[4095][0] = -0.5;  z[4095][1] = 3.455;
```

Array elements are stored sequentially in a compact way.

a[0]	a[1]	a[2]	a[3]	a[4]

Two dimensional arrays are stored in the same way. Take our array z[4096][2]

z[0][0]	z[0][1]	z[1][0]	z[1][1]	z[2][0]	z[2][1]

WARNING: There is no checking of array bounds in C so you can easily go beyond the bounds and overwrite other things.

Constants

C has many predefined constants. For example, if you look in /usr/local/math.h file you will find following entries:

```
#define M_E          2.7182818284590452354     e-base of normal logarithm
#define M_LOG2E      1.4426950408889634074     log2(e)
#define M_LOG10E     0.43429448190325182765    log10(e)
#define M_LN2        0.69314718055994530942
#define M_LN10       2.30258509299404568402
#define M_PI         3.14159265358979323846    I am sure you recognize this one
```

There are two ways to define your own constants, using normal variables with keyword *constant* or using preprocessor definitions.

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

int main(void)
{
    double constant   mypi=3.141592;
    constant int    days_in_week=7;
    …..
}
```

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

#define  MY_PI  3.141592
#define MAX_ENTRY   100000
/* No semicolon after these lines!! */
int main(void)
{
    length=2*MY_PI*r;
```