```
string sInput;
int iLength, iN;
double dblTemp;
bool again = true;

while (again) {
    iN = -1;
    again = false;
    getline(cin, sInput);
    system("cls");
    stringstream(sInput) >> dblTemp;
    iLength = sInput.length();
    if (iLength < 4) {
        again = true;
        continue;
    } else if (sInput[iLength - 3] != '.') {
        again = true;
        continue;
    } while (++iN < iLength) {
        if (isdigit(sInput[iN])) {
            continue;
        } else if (iN == (iLength - 3) ) {
```

CRASH COURSE ON C

Enrico Bertolazzi

C LANGUAGE ORIGIN

- Developed at Bell Laboratories (begin of 70) by Dennis Ritchie (a pioneer of UNIX)
- First implementation on DEC PDP-11 on UNIX OS
- By many year the "De-Facto Standard" of the language is contained in the book by Kernighan e Ritchie.
- In 1983 the ANSI committee define the first standard for C language
- Next evolved in **C89**, C90, C95, C99, C11, C18

C FEATURES

- Is an intermediate level language
  Its easy to combine high level formalist with low level instruction (like the assembly)
- Easy low level manipulation of bit, byte, word pointers
  C is target for OS development and/or hardware interface
- C for many year was a good choice for the development of general purpose library, compilers, OS, low level interface driver, …

C CHARACTERISTIC

- C is a structured language
  - C has the ability to store information and use instruction that are hidden
- C is primitive language
  - No feature are built in in the language

    I/O is done by libraries

    MATH is done by libraries

    STRING manipulation is done by libraries

    …
- C is a compiled language
  - There are dialect of C that are interpreted

    https://gitlab.com/zsaleeba/picoc (PICO C)

    http://www.softintegration.com (CH interpreter)
- C is free (mainly)
  - The C compiler is universally availble in any OS, the GNU C is available virtually on any hardware.

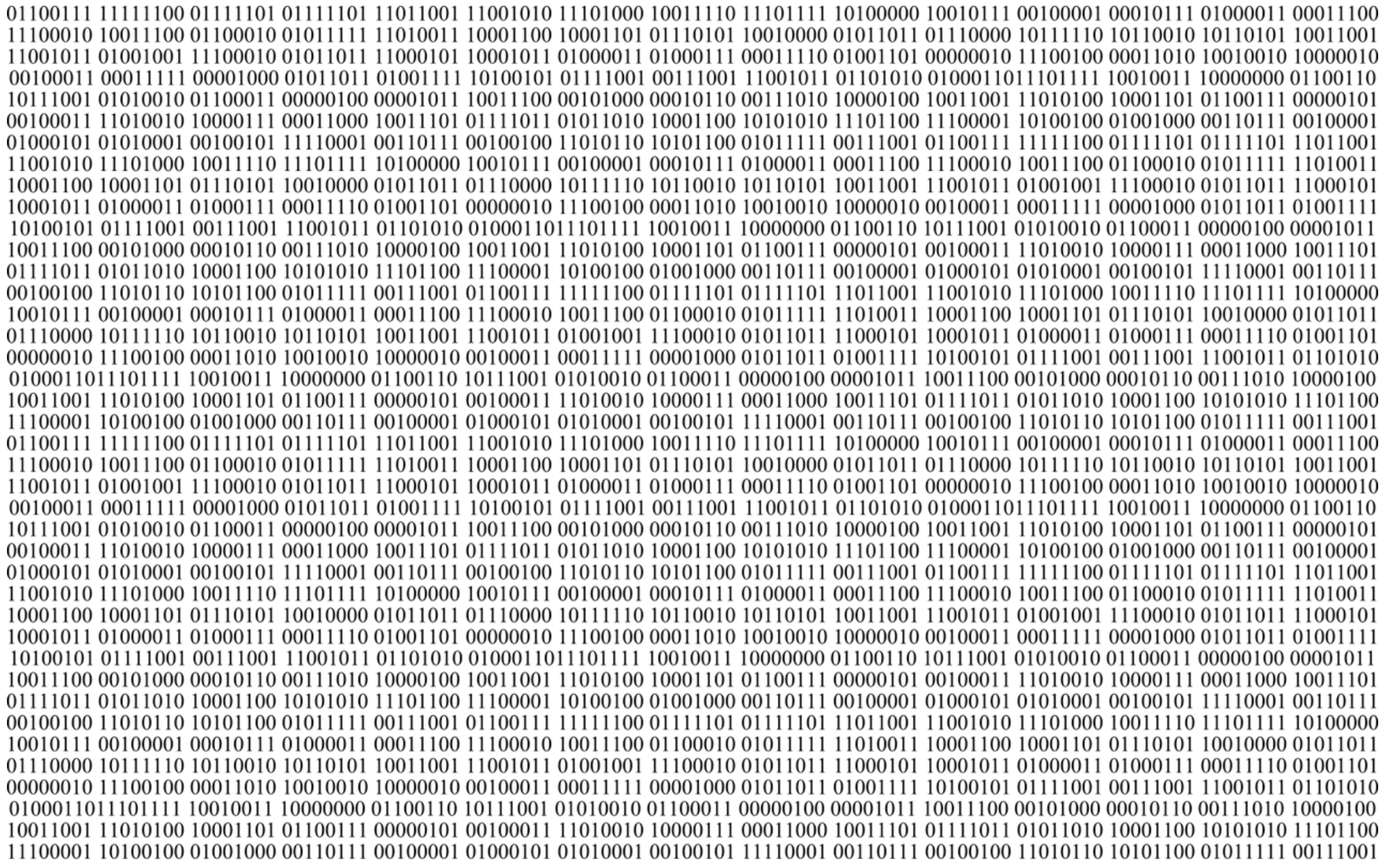
C MAIN PROGRAM

A C program consists of a single function named **main.**

The function must return an integer

```
// I am a single line comment
/*
  I am a multi line comment
*/

int // the function return a value of type “integer”
main() {

  /*\
   | fill lines by user code
   |
  \*/

  return 0;
}
```

C BASIC DATA TYPE AND OPERATOR



C KEYWORDS

- Type of data
  void, char, int, float, double, enum, struct, union
- Modification of data size/storage
  signed, unsigned, short, long, extern, static, register, cost, volatile
- Conditions and loops
  if, else, do, while, for, continue, break, goto, switch, case, default
- Operatos
  assign, **=**, logical **||** (or) **&&** (and), **!** (negation)
  arithmetic, **+**, **-**, *****, **/**, **%** (remainder only for integer)
  comparison **>**, **<**, **>=**, **<=**, **==**, **!=**
  bit manipulation, **&**, **|**, **~**, **>>**, **<<**
- Other keywords
  main, return, typedef, sizeof

SIZE OF MAINLY USED DATA

Data Type	Keyword	Size
character	char	Generally 1 byte
integer	int	Generally 4 bytes
floating point	float	Generally 4 bytes
double floating point	double	Generally 8 bytes
valueless	void	no size

SIZE MODIFIER

Modifier	apply to
long	int, double
short	int
signed	char, int
unsigned	char, int

Notice that
unsigned int
can be shorted to
unsigned

short int and **long int**
can be shorted int
short and **long**
respectively

MEMORY STORAGE SPECIFICATION

- **extern**

  In front of a variable declaration specify that the data is not stored in this unit but somewhere else. The data will be available to the unit code after the linking phase

- **static**

  The data is available ONLY on the function in the specific source code. No external symbol are generated

- **register**

  Suggest to the compiler that the data is important and must be stored in a manner as fast as possible, for example store it on a CPU register

EXTRA INFORMATION ON DATA

- **const**

  Declare the contest of the data is not modified in the current function/program. This can help compiler optimization.

- **volatile**

  The data declared volatile can change without the code explicitly access it. This is the case for example of accessing memory related to hardware units.

CONSTANTS

- **characters**

  a character is defined a letter/number between quote, e,g,

  - ‘a’, ‘h’, ‘1’

  special ascii character like new line, tab have special code prefixed by \

  - ‘\n’ new line
  - ‘\t’ tab
  - ‘\r’ return

  any ascii code can be inserted with **** followed by **ON** where **N** is an octal integer. For example the ascii character ‘+’ that correspond to the ascii character number 43 (56 in octal) can be accessed with ‘\056’

- **strings**

  are simply a consecutive sequence (array) of character terminated by the special character ‘\0’ (zero). They are denoted by double quote:

  “mikey mouse”

  “pluto” (‘p’, ‘l’, ‘u’, ’t’, ‘o’, ‘\0’)

CONSTANTS

- **integer**

  are written as usual, can specify special end character to define unsigned (U) long (L), long long (LL)

  123 (normal integer)

  123U (unsigned integer)

  123UL (unsigned long integer)
  123L (signed long integer)

  Special prefix are used to insert octal or exadecimal number
  0xFF (the number 255 in decimal notation 15+16*15)
  077 (the number 63 in decimal notation 7+8*7)

- **floating point number**

  constant number with dot are by default double precision (**double**), single precision (**float**) must be terminate by **f**

  123.0 (double number)
  123**f** or 123.0**f** (float number)

STRING CONSTANTS

- C has not an intrinsic string type (no operation on strings are defined at language level)
- C permits to define compactly sequence of character (null terminated) to be used as string
- The notation is a double quoted sequence of characters

Character	Meaning
\b	Backspace
\0	Null
\f	Form feed
\\	Backslash
\n	New line
\v	Vertical tab
\r	Carriage return
\a	Alert
\t	Horizontal tab
\NNN	NNN Octal constant
\"	Double quote
\xNN	Hexadecimal constant
\'	Single quote

Bit Manipulation

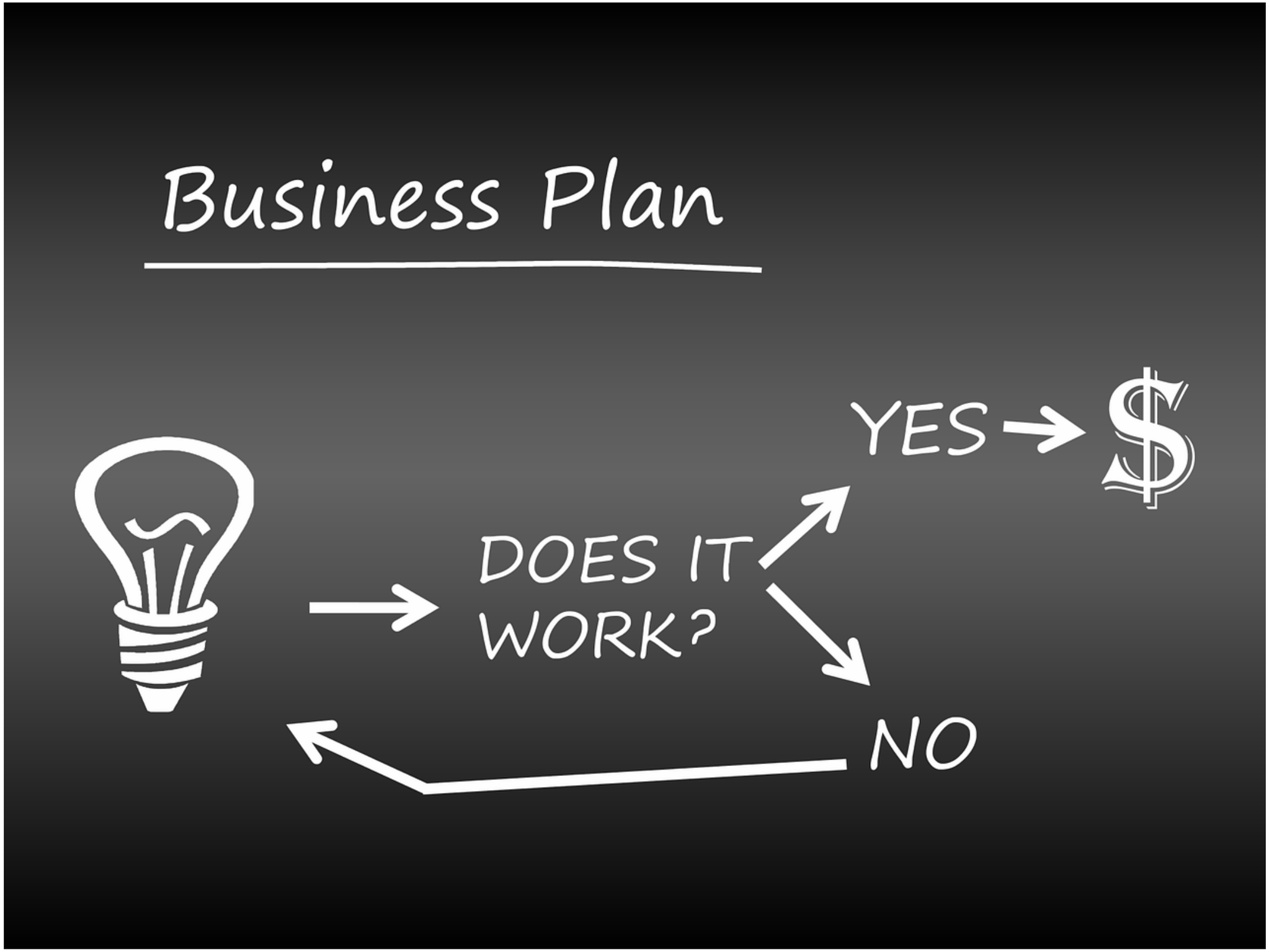
Operator	Meaning
&	AND
|	OR
>>	Shift right
<<	Shift left
^	Exclusive OR
~	One's complement

EXTRA OPERATORS

- sizeof
  - pseudo function, return at compile time the size of the argument in bytes
- &
  - return the address of the operand
- *
  - in front of a pointer or address deference the data (get the content of teh pointed data)
- , (comma)
  - used to combine expression contained between (and).
- . (dot)
  - access the field of a structure
- (type)
  - Casting operator, reinterpret or convert a data

C SHORTCUT

Shortcut	Expanded
a += b	a = a+b
a &&= b	a = a && b
a <<= b	a = a << b



C FLUX CONTROL

The "if"

```
if( expression) istruzione;
else            istruzione;
```

Example

```
b = a * c;
if(b)
   printf("%d\n",b);
else
   printf("b == zero\n");
```

The "if" combined with multiple else

```
if(expression) instruction;
else if (expression) instruction;
else if (expression) instruction;
....
else if (expression) instruction;
else instruction;
```

```
Instruction ca be also a block

{
   instruction;
   instruction;
   instruction;
}
```

INLINE/EMBEDDED "IF"

```
( expression ? TRUE : FALSE )
```

```
a = c > d ? 3 : z;

// its equivalent to

if ( c > d ) a = 3;
else         a = z;
```

SWITCH STATEMENT

```
switch ( expression ) {
   case constant1:
      sequence of statements;
      break;
   case constant2:
      sequence of statements;
      break;
      ...
   default:
      sequence of statements;
}
```

switch statements can be nested

For cycle

```
for(init;test;increment)
  statement;
```

Example

```
x = 10;
for ( y = 0 ; y != x ; ++y )
  printf("%d",y);
```

While cycle

```
while (test) statement;
```

Example

```
x = 10;
y = 10;
while ( y != x ) {
   printf("%d",y);
   ++y ;
}
```

Do cycle

```
do { statements } while ( test );
```

Example

```
x = 10;
y = 10;
do {
   printf("%d",y);
   ++y ;
} while ( y != x ) ;
```

In do-while cycle the test is done at the end, thus at least one cycle is performed

EARLY EXIT FROM THE LOOP

any cycle can be modified by using `break` and `continue`

```
i = 1;
for (;;) { // infinity loop
   ++i;
   if ( i%4 == 0 ) continue;
   // continue -> jump to next iterate
   ....
   if ( i%3 == 0 ) break;
   // break -> exit from the loop
}
```

LAST RESOURCE "GOTO"

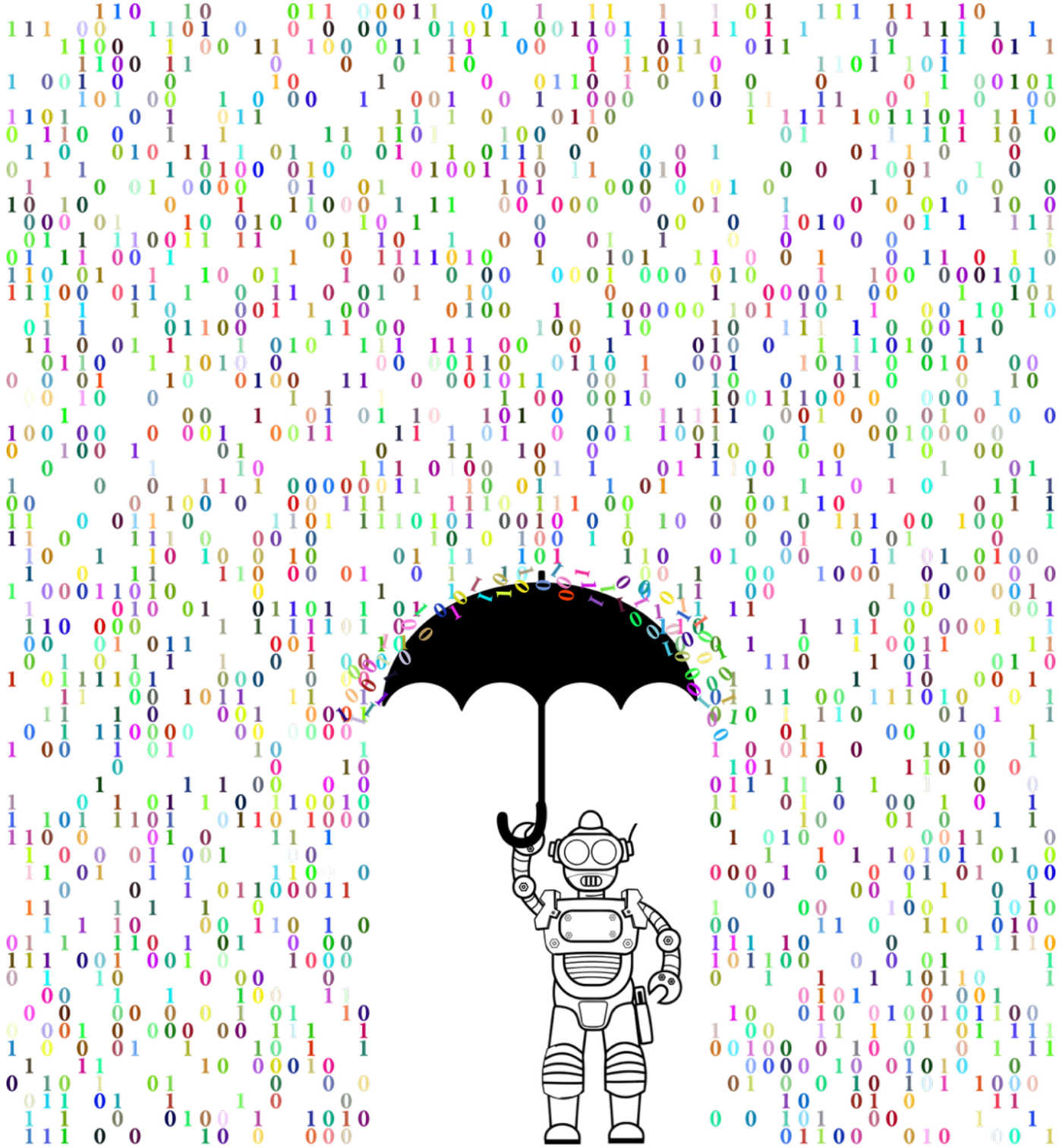
any cycle can be modified by using break and continue

```
  i = 1 ;
  for (;;) { // infinity loop
      ++i ;
      if ( i%4 == 0 ) goto skip;
      // next iterate
   }
skip:
   .....
```

RETURN (FROM A FUNCTION)

- Normally placed at the end of a function to return the computed value
- Can be put everywhere in the code
- in case the function is declared void, return is used without arguments

```
int
fibonacci( int n ) {
   if ( n <= 2 ) return 1;
   return fibonacci(n-1)+
          fibonacci(n-2);
}
```



VECTOR AND POINTERS

POINTERS

Pointers are variable that store the address of a data, not the data itself

```
int    a = 100; // define the variable a and initialize with 100
int * pa = &a;  // pa is a pointer pointing to the addess of a

*pa = 20; // * deference the address of pa, now a is set to 20!

float   vec[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
float * pv;
for ( pv = vec; pv < vec+10; ++pv )
  printf( “value = %f\n”, *pv );

// the loop print 1, 2, 3, …, 10
// vec+10 the address of vec[10] (past to the last element)
```

POINTERS ARITHMETICS

Pointers can be moved compared etc

```
float * pa;
float * pb;
float * pc;

pa = pb + 1; // if pb contains 0x00F0 then pa contains 0x00F4 (4 bytes more)
pc = pa;     // pc = 0x00F4
++pc;        // pc incremented to 0x00F8 (4 bytes more)
int diff = pc - pb; // diff = 2 not 0x00F8 - 0x00F0 = 8!

float e = *pa;     // extract the contents of the memory pointed by pa
float f = *(pa+3); // extract the contents of the memory pointed by pa+3
float g = pa[3];   // shortcut for *(pa+3)
```

ARRAYS ARE SPECIAL POINTERS

Vectors/array are pointer to a block of memory. This pointer is not modifiable

```
float a[100]; // define an array of 100 float numbers
float * pb;   // pb is a pointer to a float
a  = pb;      // ERROR: a cannot be modified
pb = a;       // OK: now pb point to the same piece of memory of a

a[2] == pb[2];
a[4] == pb[4];

++a; // /ERROR: a cannot be modified
```

STRINGS

STRING ARE POINTERS TO CHAR..

String are not native in C, are simply a pointer to memory that contains characters **null** terminated

```
char s[] = “mery poppins”;

// s is a pointer to a vector of char

s[0]  == ‘m’;
s[4]  == ‘ ‘;
s[11] == ’s’;
s[12] == ‘\0’; // null character, end of a string

s[13] ->
// out of the string can contains anything or return a bad address error
```

String operation are contained in the standard C library

String operations are function of the standard C library.

To access the library include the header **string.h**

```
// include function definition for string manipulation
#include <string.h>

int
main() {
  char s[] = “mery”;
  char t[] = “poppins”;
  char st[100];

  // use strcat the signature is
  // char *strcat(char *dest, const char *src)
  // char *strcpy(char *dest, const char *src);

  strcpy( st, s );   // now st contains “mery”
  strcat( st, “ “ ); // now st contains “mery ”
  strcat( st, t );   // now st contains “mery poppins”

  return 0;
}
```

PRINCIPAL STRING OPERATION

String operations are function of the standard C library.
To access the library include the header **string.h**

- char *strcpy(char *dest, const char *src);
  copy the string src to dest (no memory check)
- char *strcat(char *dest, const char *src);
  append the string src to dest
- int strcmp(const char * a, const char * b);
  compare the contents of string a with string b. Return -1 if a < b, +1 if a > b and 0 if a == b.
- size_t strlen(const char *s);
  return the length of the string s, size_t is normally unsigned in
- char *strstr(const char *str, const char *substr)

  return the pointer of the first match of substr in str, if there is no match return **NULL**, the null pointer

TYPEDEF AND NEW TYPES

STRUCT

Struct is a way to collect many types in one object

```
struct name_of_the_struct {
  int    a;
  float  b;
  double c[100];
};

// usage

struct name_of_the_struct S;

S.a    = 1;
S.b    = 3.45f;
S.c[0] = 1.0;
```

BITFIEDS

In a struct you can access integer data using portion of bits. Its better to use bit manipulation…

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int funny_int:9;
};

// usage

struct packed_struct S;

S.f1        = 1;
S.f4        = 0;
S.funny_int = 23;
```

UNIONS

Sometimes is useful to view the same portion of memory as different types.

```
union overlapped_data {
  double f;
  int    a[2];
  char   c[8];
};

// usage

struct overlapped_data S;

// S.f, S.a and S.c are on the same block of memory

S.f = 1.34;

S.c[0]; // the bytes get part of the floating point
S.c[1]; // representation of “S.f”
```

ENUM

Sometimes it is useful have a sequence of number stored on variables that cannot be changed

```
enum {
   FIRST,
   SECOND,
   THIRD
};

// FIRST  is the constant 0
// SECOND is the constant 1
// THIRD  is the constant 2

// sequence can be changed

enum {
   A = 20,
   B,
   C = 100
};

// A  is the constant 20
// B  is the constant 21
// C  is the constant 100
```

TYPEDEF

Complex data type have long declaration, typedef make an alias

```c
typedef struct {
  int  a;
  char b;
} the_struct_type;

the_struct_type a_struct;

// define a vector of pointer to char (10 elements)
char const * strs[10];

typedef char const * vector_of_pointer_to_char[10];

// define a vector of pointer to char (10 elements)
vector_of_pointer_to_char strs1;

typedef double real_type; // real_type is an alias of double
typedef int    integer;   // integer is an alias of int
```

CASTING

CAST OPERATOR

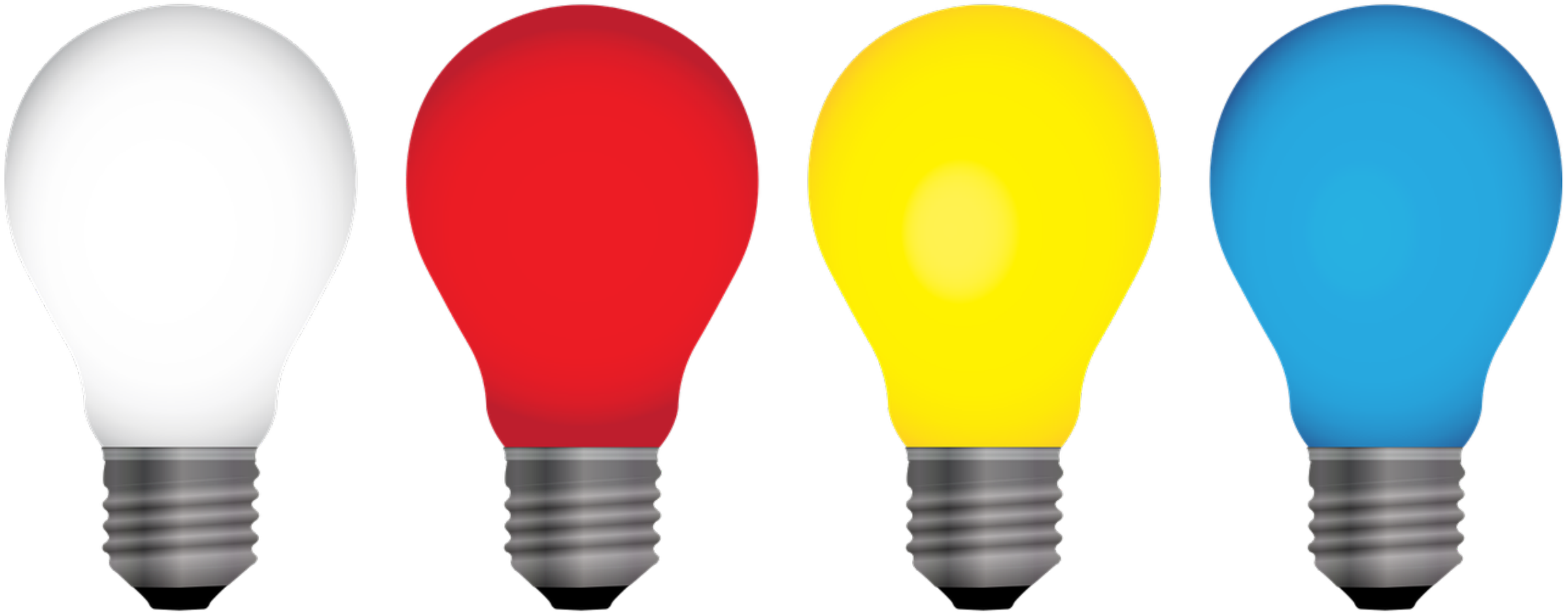
Sometime you need to change a type to another type

```
double a = 1.5;
int    b = (int) a; // conversion to int, a truncation happen

int    c = 23;
double d = c;          // warning issue due to implicit conversion
double e = (double)c; // conversion to double without warning

double   vec[100];
double * ptr  = vec+12; // now ptr point to &vec[12]
float  * ptr1 = vec+3;  // error issue due to different pointer type
float  * ptr2 = (float*)(vec+3); // ok, the pointer is converted
                                 // mo matter if the think has few sense
char   * ptr3 = (char*)(vec+12); // OK

// mixed precision operation
float  w = 123.0f;
double z = 23.34;
double res1 = z*w;          // w is converted to double
double res2 = z*(double)w;
float  res3 = z*w; // w is converted to double, result is converted to float
float  res3 = (float)z*w;  // z is converted to float, no more conversion
```

CAST OPERATOR

Casting is mandatory for dynamic allocation

```c
#include <stdlib.h> /* calloc, exit, free */

int
main () {
  int i,n;
  int * pData;

  // do something

  pData = (int*) calloc( i, sizeof(int) ); // allocate memory

  // do something

  free (pData); // free allocated memory
  return 0;
}
```



SPLIT YOUR CODE IN MANY FILES

USE HEADER TO DEFINE PROTOTYPE

A good library split code and header with the prototype of the library functions

```c
// file factorial.h
// declare the prototype
// of the function factorial

extern int factorial( int n );
```

```c
// file factorial.c
#include “factorial.h”
int
factorial( int n ) {
  if ( n <= 1 ) return 1;
  return n*factorial(n-1);
}
```

header file declaring the prototype of function factorial

body file implementing the function factorial

```c
// file main.c
#include “factorial.h”

int
main() {
  int n, fn;
  printf(“insert a number\n”);
  scanf(“%d”,&n);
  fn = factorial( n );
  printf(“%d! = %d\n”, n, fn);
  return 0
}
```

main program that use the function factorial

EXTERN/STATIC

extern tell the compiler that a function prototype will be imnplemented in another unit.

```c
// file functions.h
// declare the prototype of the
// functions factorial and fibonacci

extern int factorial( int n );
extern int fibonacci( int n );
```

```c
// file functions.c
#include “functions.h”
int factorial( int n ) { … }
int fibonacci( int n ) { … }

static int local_fun( int m ) { … }
```

```c
// file main.c
#include “functions.h”

// local_fun is not accessible
// cannot be linked
int
main() {
   int n, fn;
   printf(“insert a number\n”);
   scanf(“%d”,&n);
   fn = fibonacci( n );
   printf(“%F(d) = %d\n”, n, fn);
   return 0
}
```

header file declaring the prototypes

body file implementing the functions

main program that use the functions

MEMORY ALLOCATION

MALLOC/FREE

Often a program cannot known in advance the memory requirements

```c
#include <stdlib.h> /* calloc, exit, free */

int
main () {
  int i,n;
  int * pData;

  // do something

  int n = 100;
  // allocate memory for 100 integers
  pData = (int*) malloc( n * sizeof(int) );

  // do something

  free (pData); // free allocated memory
  return 0;
}
```

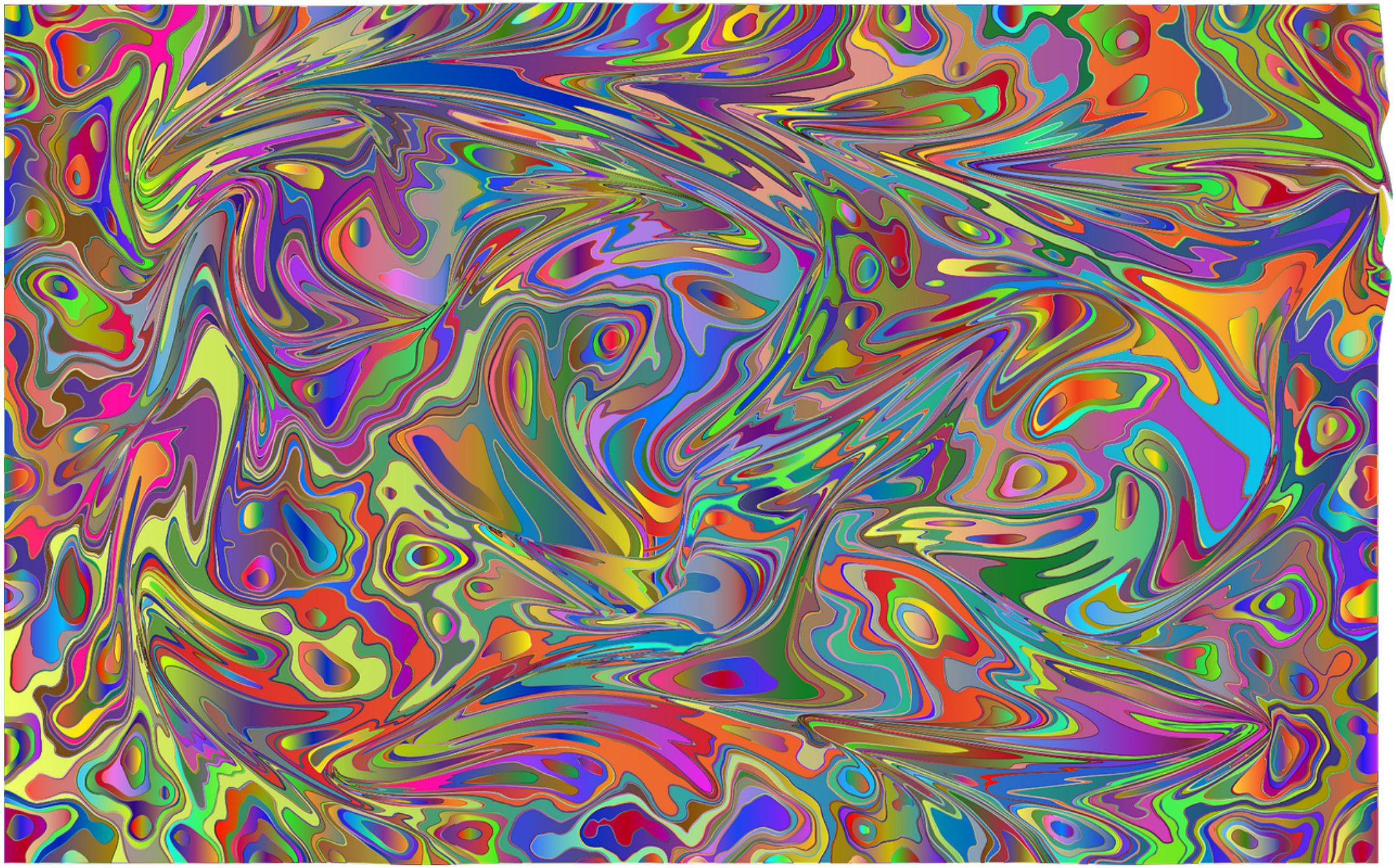
CALLOC/FREE

Often a program cannot known in advance the memory requirements

```c
#include <stdlib.h> /* calloc, exit, free */

int
main () {
  int i,n;
  int * pData;

  // do something

  int n = 100;
  // allocate memory for 100 integers
  pData = (int*) calloc( n, sizeof(int) );

  // do something

  free (pData); // free allocated memory
  return 0;
}
```

PRINCIPAL MEMORY ALLOCATION FUNCTIONS

Function	Purpose
malloc(n)	allocate memory for a block of n bytes, retrun a pointer to the memory block
realloc(ptr, n)	loically equivalent to free(ptr) followed by malloc(n)
calloc(n, size)	equivalent to malloc(n * size) followed by a zero initialize of the block of memory
free(ptr)	free the allocated memory pointer by ptr. If ptr point to a **NOT** allocated piece of memory an error is issued



ARGUMENTS FOR MAIN

ARGUMENT FOR MAIN

A compiled C code is a command for the OS that may use argument. Like ls -l, ps -e, cat pippo…

The syntax is

```
int main( int argc, char const argv[] ) { … }
```

An integer containing
the number of argument passed.
For example

cmd arg1 arg2

argc = 3

A vector of pointer to C-strings.
For example

cmd arg1 arg2

argc[0] point to “cmd”
argc[1] point to “arg1”
argc[2] point to “arg2”

ARGUMENT FOR MAIN

A compiled C code is a command for the OS that may use argument. Like ls -l, ps -e, cat pippo…

```c
// file main.c
#include “factorial.h”

int
main( int argc, char const * argv[] ) {
  // argc count the number of argument
  // argv is a vector of pointer to char
  // (C-string) with the arguments

  if ( argc != 2 ) {
    printf(“usage %s n\nn is an integer”, argv[0]);
    return 1; // error
  }

  sscanf( argv[1], “%d”, &n );

  printf( “computed %d! = %d\n” );
  return 0;
}
```

```c
// file factorial.h
// declare the prototype
// of the function factorial

extern int factorial( int n );
```

header file with the prototype

main program

compile link and execute

```
> gcc main.c factorial.c -o runme
>
> ./runme 10
computed 10! = 3628800
```

THE I/O

PRINTF = PRINT FORMATTED

I/O is done by function library

```c
// file main.c
#include <stdio.h> // include function definition for I/O

int
main() {
  for ( int i = 0; i <= 5; ++i )
    printf( “i = %d, i*i = %d\n”, i, i*i);
  return 0;
}
```

printf write on standard output with formatting.

The argument are formatted using %CODE where CODE is how the arguments is maniopulated

```
> gcc main.c –o runme
>
> ./runme
0, 0
1, 1
2, 4
3, 9
4, 16
5 25
```

PRINTF = PRINT FORMATTED

I/O is done by function library

```
// file main.c
#include <stdio.h> // include function definition for I/O

int
main() {
  for ( int i = 0; i <= 5; ++i )
    printf( “i = %5d, i*i = %-5d\n”, i, i*i);
  return 0;
}
```

%d means format an integer as a decimal number

%5d reserve 5 space for printing

%-5d also left justify

```
> gcc main.c –o runme
>
> ./runme
    0, 0
    1, 1
    2, 4
    3, 9
    4, 16
    5, 25
```

SCANF

```c
// file main.c
#include <stdio.h> // include function definition for I/O

int
main() {
  double a[2];
  printf("input the numbers\n");
  for ( int i = 0; i < 2; ++i ) scanf( "%lf", &a[i]);
  printf("you entered\n");
  for ( int i = 0; i < 2; ++i ) printf( "a[%d] = %lf\n", i, a[i] );
  return 0;
}
```

```
> gcc main.c -o runme
>
> ./runme
input the numbers
1
34.45
23
you entered
a[0] = 1
a[1] = 34.45
a[2] = 23
```

PRINTF/SCANF FORMATTING CODES

- The formatting is done by % followed by a letter

  %d = decimal number

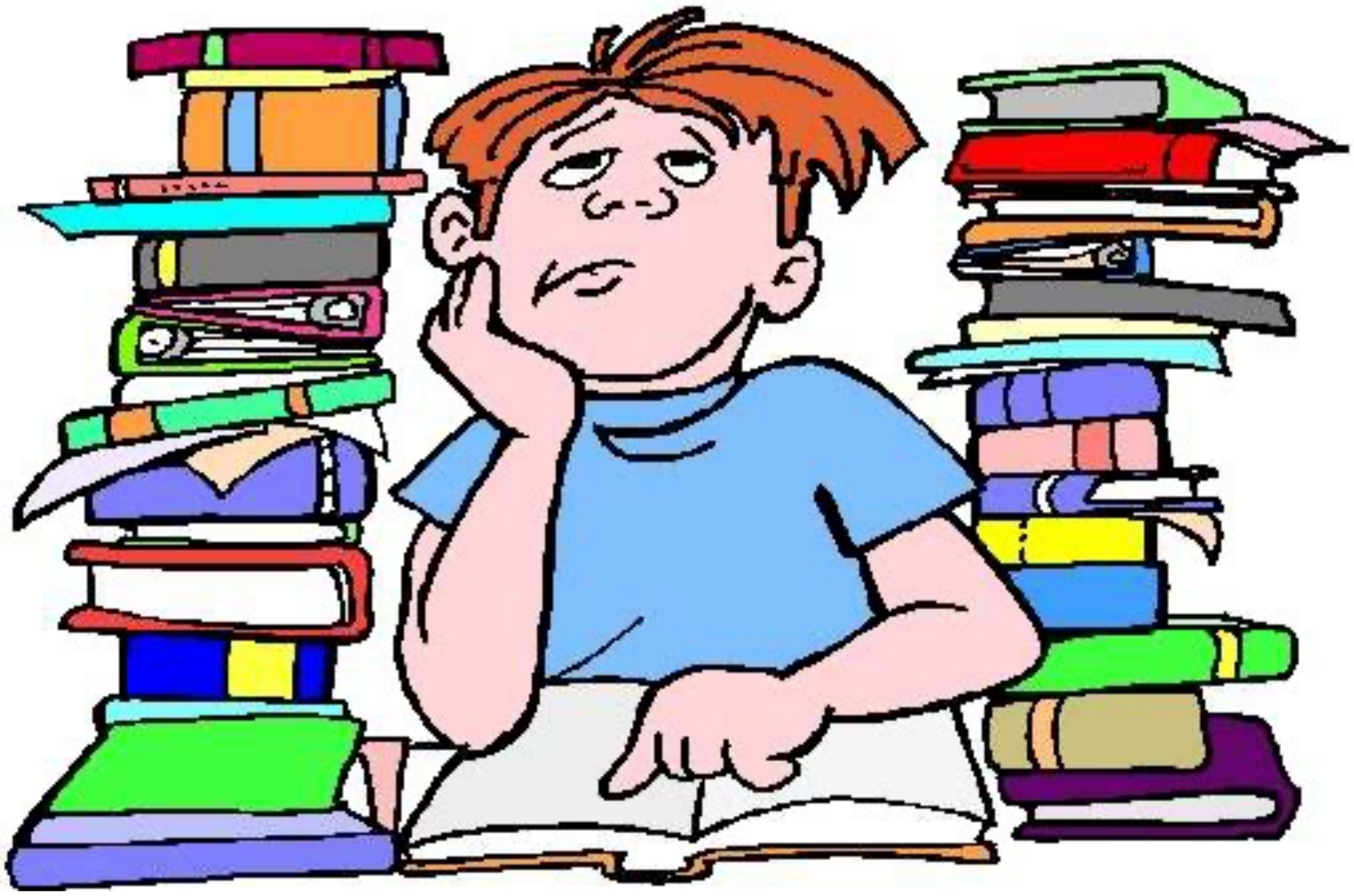
  %f = float number (%lf for double)

  %e = float formatted using exponential form 0.0012f = 1.2e-3
  %le = double formatted using exponential form 0.0012 = 1.2e-3
  %g = float formatted normally or exponential, choose the better looking (%lg for double)

- %[n] reserve n character for the print, for example
  printf("*%5g*\n", 12.5f) print
  __12.5 (two more space on the left)

- %- left align, for example
  printf("*%-5g*\n", 12.5f) print
  12.5__ (two more space on the right)

- for a full list of the formatting code read at
  http://www.cplusplus.com/reference/cstdio/printf/?kw=printf
  http://www.cplusplus.com/reference/cstdio/scanf/?kw=scanf



THE PREPROCESSOR

PREPROCESSOR

- C is a compiled language, it means that the source code is translated to machine code and linked with system/user libraries to produce an executable (or another library)
- Before to start the compiling phase (the translation of the C code to a machine code) the source code is processed/filtered by preprocessor that produce a new source code
- Preprocessor is used normally to include definition and prototype from system and user libraries but the macros mechanism permits to use conditional programming so that the same source code can adapt to work well on different architectures or operative systems.
- Macros are widely used to produce "inline" functions or named constants

#DEFINE

- #define MACRO_NAME
  - The preprocessor define the macros **MACRO_NAME** every time it find the string **MACRO_NAME** is replaced with the text on the right in this case nothing.
- #define MAX(A,B) ((A) > (B) ? (A) : (B))
  - In when the compiler find MAX(i+1,j-1) replace with ((i+1) > (j-1) : (i+1) : (j-1))
- #define NAME_USER “pippo”
  - In when the compiler find NAME_USER replace with “pippo”

before preprocessing

```
#define USER “pippo”
#define FUN(A) (A)*(A)

printf( “Username: “ USER “\n” );
int i = 12;
printf( “i*i = “, FUN(i) );
```

after preprocessing

```
printf( “Username: “ “pippo” “\n” );
int i = 12;
printf( “i*i = “,(i) * (i) );
```

consecutive constant strings are concatenated

```
printf( “Username: pippo\n” );
int i = 12;
printf( “i*i = “,(i) * (i) );
```

#INCLUDE

- **#include "filename"**
  - The preprocessor read the file "filename" and expand as is written in the same place
- **#include <filename>**
  - same as previous but "filename" is searched in system directories Normally **/usr/include** but depend on the OS or the compiler

file a_file.h

```
extern void fun( int i );
extern void fun2( int i );
extern void fun3( int i );
```

before preprocessing

```
#include "a_file.h"

int
main() {
  int i=1;
  fun(i);
  return 0;
}
```

after preprocessing

```
extern void fun( int i );
extern void fun2( int i );
extern void fun3( int i );

int
main() {
  int i=1;
  fun(i);
  return 0;
}
```

#IFDEF #IFNDEF #ELSE #ENDIF

- #ifdef SOMETHING
  - expand the block is SOMETHING is a defined macros
- #ifndef SOMETHING
  - expand the block is SOMETHING is **NOT** a defined macros

before preprocessing

```
#ifdef PIPPO
  int   a = 1;
  float b = 2;
#else
  int   a = 100;
  float b = 1.3;
#endif
```

after preprocessing
and PIPPO is defined

```
int   a = 1;
float b = 2;
```

after preprocessing
and PIPPO is NOT defined

```
int   a = 100;
float b = 1.3;
```

AVOID DOUBLE INCLUSION TRICK

file.h

```
#ifndef FILE_HH
#define FILE_HH

int a = 1;
double fun( int x );
#define ODD( A ) (((A)%2) == 1 )

#endif
```

before preprocessing

```
#include “file.h”
#include “file.h”

int
main() {
  int a = 1;
  if ( ODD(a) ) {
    printf(“%d is odd\n”, a );
  } else {
    printf(“%d is evan\n”, a );
  }
  return 0;
}
```

after preprocessing

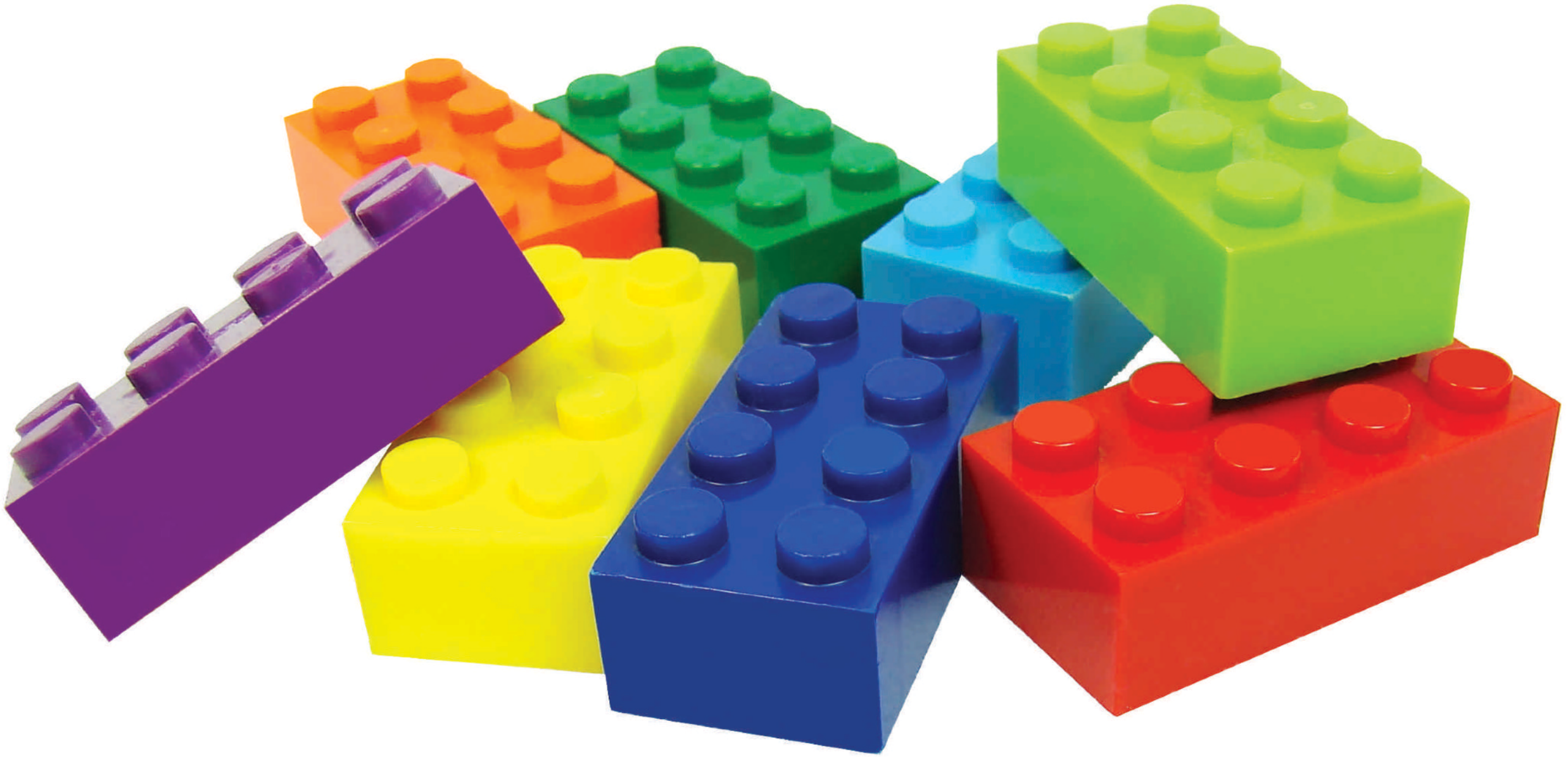
```
int a = 1;
double fun( int x );
#define ODD( A ) (((A)%2) == 1 )

int
main() {
  int a = 123;
  if ( ( ( (a) % 2) ) == 1 ) ) {
    printf(“%d is odd\n”, a );
  } else {
    printf(“%d is evan\n”, a );
  }
  return 0;
}
```

OTHER PREPROCESSOR COMMAND

- #if defined(MACRO)
  is the same of #ifdef MACRO, but can combine things more complex
- #if defined(A) && !define(B)
- #elseif
  used in the #ifdef block
- #line NUMBER
  change the line number used in the debug message
- #pragma
  pass special command to the compiler, its compiler dependent.
  Read the manual of the compiler for the list of pragmas
- #undef MACRO
  if the macro **MACRO** is defined will be "cancelled"

SPECIAL PREPROCESSOR COMMAND

- #define A(X) #X #X
  the # convert symbol to string, in this example A(pippo) is expanded to “pippo” “pippo”
- #define CONCAT(B,C) B ## C
  perform string concatenation, in this example CONCAT(“pippo”,”pluto”) is expanded to “pippopluto”
- Predefined macros
  - __LINE__
  - __FILE__
  - __DATE__
  - __TIME__
  - __STDC__
  - more others…



C STANDARD .H FILES