C Language Initial

1. Getting started

1.1 What is C ?

- C is a compiled language which was created (by Ritchie and Kernighan in 1972) for facilitating the creation of a portable operating system : **Unix** (previous operating systems were written in assembly language, hardware dependant and expensive to develop).

- Its flexibility allowed to extend its application range from the small 8-bit microcontroller to the supercomputer, and it turned to be the “reference programming language”.

- C has a little number of built-in constructs (instructions), many things relying on standard libraries which come with the compiler. (But C may be used without those libraries).

- Some design flaws in the initial C spec of 1978 (“**K&R**”) where corrected later by the “ ANSI C” in 1989 (**C89**) adopted by ISO (**C90**). Some additions were made in 1999/2000 by ISO (**C99**).

- **ANSI C** (C89) is the basis for all present days C programming.

- C++ is made of “C” and “++” : C++ is an extension of C, created (1984) with the main goal of introducing object-oriented programming, while preserving the flexibility of C - learning C++ requires learning C first !

```
/* this is a simple program,
  and this text is a comment */

// below is a preprocessor directive
// requesting the inclusion of another source file
// before compiling
#include <stdio.h>

// below is a variable declaration
// of type int, with initialization
int MyNumber = 5;

// the 'main' function, which the operating system will
// consider as the entry point
int main()
{
// a call to a library function :
printf("Hello, my number is %d\n",
        MyNumber );
// an assignment (from right to left) :
MyNumber = 77;
printf("now it is %d\n", MyNumber );
// the end
return 0;
}
```

1.2 Syntax elements

The "source code" is "free form text". This means that additional whitespace is ignored, and that the line breaks are considered ordinary whitespace, like the space character and the tabulation character.

There are two notable exceptions, where the line break is a terminator :

- the preprocessor directives (beginning with a #)
- the "C++ style" comments, (beginning with //)

The language is case-sensitive.
ANSI-C has some 32 keywords (reserved words), which are all in lower-case characters . In the example, **int**, **return** are keywords.

User identifiers like variable names and function names may contain only letters (upper and lower case), decimal digits and the underscore character, and the first character should not be a decimal digit.

The semicolon character ';' is the terminator for declarations and instructions.
The comma ',' is used as separator in lists.
The function body is enclosed in curly braces {}.

```
// two uninitialized variables declared here :
int a, b;

int main()
{
a = 12;
b = 41;
// a + b will be computed at the time of printing
printf("%d + %d = %d\n", a, b, a+b );
return 0;
}


/* this example is supposed to print :
12 + 41 = 53
 */


/* Note : spaces are optional in the source text, except in places where there is nothing else to delimit keywords and identifiers. The example below should give the same results as the one above.*/

int a,b;int main(){a=12;b=41;
printf("%d + %d = %d\n",a,b,a+b);
return 0;}
```

What about this "printf" ?

"**printf**" stands for "print formatted".

It makes use of all the difficult concepts of C, so a complete understanding of its operation will be possible only near the end of the course !

But we need it immediately to get started, so we are invited to admit that :

- it takes a "format string" as its first argument, which should be delimited by double quotes,
- other arguments may be variables or expressions,
- the format string may contain "placeholders" (beginning with '**%**') which will be substituted by the values of the arguments, at run time.
- '**\n**' (backslash n) is a convenience for creating a line break in the output without creating a line break in the source code.
- other elements of the format string will be printed verbatim.

1.3 Simple types

In C, variables have to be declared before use.
The simple variable types fall in 3 categories :

- integral numbers
- floating point numbers
- pointers (deferred to the next chapter)

Integral types differ by the size, their names are (by increasing size order) :

- **char**
- **short int** (or just **short**)
- **int** (the “default” or “preferred” type)
- **long int** (or just **long**)
- **long long** (introduced in C99)

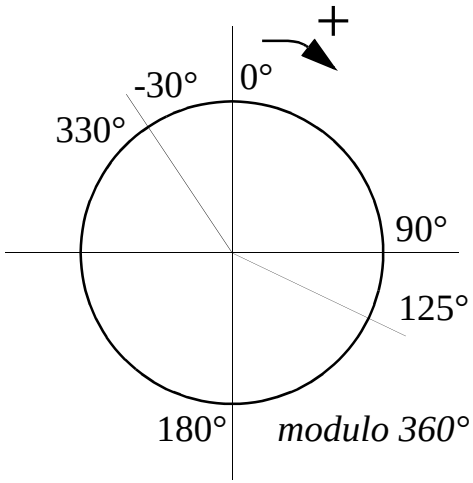
All sizes should be multiple of the **char**’s size (which is nearly always 8 bits = 1 byte).
On a given computer architecture, two types may be equivalent (**int** and **long** are the same on the 32 bits “i386”).

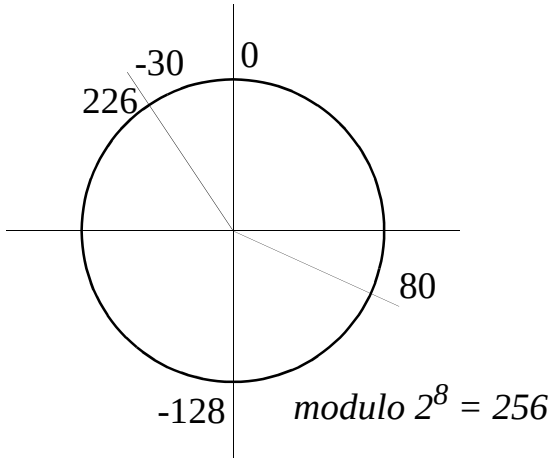
They are considered as **signed** numbers, unless explicitly qualified as **unsigned**.
(Warning : on some implementations, char is unsigned; the **signed** keyword may be used to remove any doubt)

Signed integral numbers are internally represented in **2’s complement** encoding. Addition and subtraction are performed the same way for signed and unsigned numbers. But multiplication, division and comparisons differ.

Floating point numbers or “real” numbers come in two sizes, **float** and **double** (**long double** may exist).
double is the preferred type.
Floating point numbers are always signed.

Two's complement numbers are like angles





If we subtract 155 degrees from an 125 degrees angle, we obtain an angle of -30 degrees which has the same representation as a 330 degrees angle.

8-bit example :
Subtracting 110 from 80 gives -30 which is equivalent to 226 in a modulo-256 environment.
The MSB is the "sign bit"

Digital circuits working on N-bit words perform addition and subtraction in a "modulo 2^N" environment.
The same binary word may be considered as representing a signed or unsigned value.
A negative value X has the same representation as the unsigned value $2^N - |X|$

- Unsigned values range from 0 to $2^N - 1$
- Signed values range from -2^{N-1} to $+2^{N-1} - 1$

```
int a, b, c, d;

a = 65;      // decimal
b = 0x41;    // hexadecimal
c = 'A';     // character
d = 0101;    // octal

// in the format string, %d means "signed decimal"
// what is going to be printed ?
printf("%d %d %d %d\n", a, b, c, d );

// in the format string, %u means "unsigned decimal"
// %X means unsigned hexadecimal
// what is going to be printed now ?
a = -1;
printf("%d %u %X\n", a, a, a );

// %c means print a single character the code
// of which is given
printf("%c %c\n", b, b + 1 );


/* answers :
65 65 65 65
-1 4294967295 FFFFFFFF (32-bit machine)
A B */
```

1.4 Numerical constants

Integer constants (literals) may be expressed in :

- decimal (default)
- hexadecimal (with the 0x prefix)
- octal (with just a leading 0, ***dangerous*** !)

The numerical value of the encoding of a printable character may be obtained by enclosing this character in single quotes, like 'Y' (The preferred encoding is the ASCII code).

Floating point constants may be expressed in :

- decimal fixed point format
  example : 3.14159
- decimal exponential format
  example : 1.6e-19

```
int a, b, c, s, t; double dd;

s = a + ( b + c );
// same as :
s = a + b + c;

s = ( a * b ) + c;
//same as
s = a * b + c;  // well know precedence

// different from
t = a * ( b + c );

a = 2; b = 3;

// just below, conversion is made after integer division,
// which gives zero (disappointing)
dd = a / b;
printf("2/3 = %f\n", dd );

// a better approach :
dd = a;
dd = dd / b;
printf("2/3 = %f\n", dd );
```

1.5 Expressions

- Expression may appear :
  - on the right hand side of the = sign in an assignment statement
  - as an argument in a function call
- expressions are based on operators
- expressions may be nested by means of parentheses
- parentheses may be omitted if operators precedence is “well known”
- implicit type conversion is performed silently to the broadest type present in the expression or subexpression, then to the left-hand side type
- integers are :
  - broadened by preserving sign if any
  - shrunk by discarding most significant bits (danger here)

In case of doubt : ***play safe***, perform conversion before any calculation.

```
// checking the integer division behaviour
a = 1973, b = 66;
c = (a/b)*b + a%b;
printf("%d should be %d\n", a, c );

// pre-increment and post-increment
b = 5;
printf("preinc b = %d\n", ++b );
printf("postinc b = %d\n", b++ );
printf("final b = %d\n", b );
// what will be printed ?


/* answer :
  1973 should be 1973
  preinc b = 6
  postinc b = 6
  final b = 7
*/


// BAD : illegal operations, refused by the compiler :
a + b = a + c;
a++ = 5;
// "illegal lvalue" means that the left hand side operand
// cannot receive a value
```

Arithmetic operators

operation	binary operator	assignment operator
addition	+	+=
subtraction	-	-=
multiplication	*	*=
division	/	/=
modulo (div. remainder)	%	%=

change sign (unary operator)	-

- the assignment operators store the result by overwriting the left side operand.
  Example : a += 2; is the same as a = a + 2;

- the prefix operators -- and ++ (placed before an operand) add or subtract 1 to the operand, store the result and then return it.
- the postfix operators -- and ++ (placed after an operand) save the operand value, add or subtract 1 to the operand, store the result and then return the previous value of the operand.

```
// masking some bits
// here we keep only the 6 LSBs
a  = b & 0x3F;

// force bit of weight 4 to be 1
b |= 4;

// force bit of weight 4 to be 0
c &= ( ~4 );

// complement the 4 LSBs
d ^= 0x0F;




/* Note : in binary,
0x3F is :
00000000000000000000000000111111

4 is :
00000000000000000000000000000100

~4 is :
11111111111111111111111111111011
*/
```

Bitwise Logic operators

"bitwise" means each bit of each operand will be combined with the corresponding bit of the other operand to create one bit of the result.

An example in binary notation (with the **and** function) :

```
   101101110001
 & 001111100000
 ---------------
 = 001101100000
```

In this scheme, the **and** function is performed in each column independently from the others.

operation	binary operator	assignment operator
and	&	&=
or	|	|=
exclusive or (xor)	^	^=

not (complement) (unary operator)	~

```
// swap 2 groups of 4 bits in a 8-bit data
// using b as temporary storage
b = ( a & 0x0F ) << 4;
a = ( a & 0xF0 ) >> 4;
a |= b;

// the same job, much safer because
// masking-after-shift
b = ( a >> 4 ) & 0x0F;
a = ( a << 4 ) & 0xF0;
a |= b;


N = 5;
// force bit N to be 0
// N is the rank, not the weight
c &= ~( 1 << N );
// mask to keep N LSBs
d &= ( 1 << N ) - 1;

/* Note : in binary, with N = 5,
1 << N is :
00000000000000000000000000100000
(1<<N) - 1 is :
00000000000000000000000000011111
*/
```

Shift operators

The shift operators are not symmetric. The first operand is the data word to shift, the second is the amount of shifting to be done.

Left shift means shift towards the most significant bit.

- The right side is stuffed with zeroes.

Shifting left by one bit is equivalent of multiplying by two (unless there is an overflow).

Right shift means shift towards the least significant bit.

- If the data word is unsigned, the left side is stuffed with zeroes.
- If the data word is signed and negative (having the MSB set), the left side is (usually) stuffed with ones to keep the sign.

operation	binary operator	assignment operator
shift left	<<	<<=
shift right	>>	>>=

Note : use caution when shifting right signed variables.

```
c = 0x31;
// check whether c is the ASCII
// code of a decimal digit
if (
      ( c >= '0' ) &&
      ( c <= '9' )
   ) printf("ok\n");

a = 0;

// good test
if ( a == 0 )
   printf("a is null\n");


// BAD, a common programming mistake
// legal but never prints
if ( a = 0 )
   printf("a is null\n");
// expression a = 0 returns always 0
// some compilers may give a warning
```

Relational boolean operators

These operator are mainly used for expressing conditions for conditional statements like "**if**".
They return a "boolean value", **true** or **false**.

comparison	binary operator
equal	==
lesser than	<
lesser than or equal	<=
greater than	>
greater than or equal	>=
different	!=

boolean operation	binary operator
and	&&
or	||

not (negation) (unary operator)	!

Notice the double symbols : ==, &&, ||, their meaning is totally different from the single ones.

```
// check bit of weight 8
if ( a & 8 )
   printf("bit 3 is set\n");



// watch the difference between & and &&

if ( a && b )
   printf("a and b both non zero\n");

if ( a & b )
   printf("at least 1 common bit\n");
```

Handling boolean values

Since there is no boolean type (except in C99), the **int** type is used for handling the results of boolean operations, with the following convention :

- **false** is 0
- **true** is any non-zero value, like 1 or -1

This has two practical consequences :

- the result of a comparison or boolean expression may be stored in an **int** variable for later use
- an **int** variable or expression may be used as condition for an **if** statement (or any other conditional statement)

Example :

```
if (a)
```

is equivalent to :

```
if ( a != 0 )
```

```
// tracking the min and max
min = max = 0;

   // do the following for each value of a
   if   ( a < min )
        {
        printf ("small");
        min = a;
        }
   else if ( a > max )
           {
           printf("big");
           max = a;
           }



// BAD : killing semicolon, always prints
if ( a < 0 );
   printf("negative\n");

// indentation may hide errors...
```

1.6 The if statement

This is the simplest of the execution control statements. Like with the other control statements, the code to be conditionally executed may be :

- a single statement
- a block, which is made out of a sequence of statements enclosed in curly braces { }

This is why there is nothing like “endif” in C.

The parentheses around the condition are mandatory. The **else** clause is optional.

A spare semicolon is legal as an “empty statement” (may cause subtle errors)

There is nothing wrong enclosing a single statement in braces.

In the case of nested **if**’s, each else is associated with the nearest preceding **if** *in the same block.*

```
// what will do each of these loops ?

cnt = 5;
while ( cnt )
   {
   printf("%d ", cnt );
   cnt--;
   }
printf("\n");


cnt = 5;
while ( cnt-- )
   printf("%d ", cnt );
printf("\n");




// answers :
// 5 4 3 2 1
// 4 3 2 1 0
```

1.7 The while statement

The **while** statement is the fundamental loop construct in C.

The body will be repeated while the condition is true.

The parentheses around the condition are mandatory.

The condition is tested **before** executing the body, so

- if the condition is already false, the body will not be executed at all
- if the condition is made false during the execution of the body, this will be the last execution
- if nothing in the body causes the condition to change, infinite looping is likely to occur (except if the hardware or another execution thread changes the condition).

C programmers are known to use *side effects* of the condition evaluation to perform part (or all) of the iterated task (see the second example).

```
// a simple counted loop

for ( i = 0; i < 5; i++ )
    printf("%d", i );
printf("\n");


// same as :

i = 0;
while ( i < 5 )
   {
   printf("%d", i );
   i++;
   }
printf("\n");
```

1.8 The for statement

The **for** statement is a variant of the **while** statement, which is more suitable for counted iterations but otherwise more complicated to describe.

The **for** parentheses contain three statements :

- the first one is the **initialization** : it is executed once, before anything else
- the second one is the **condition**, it is evaluated before each iteration of the loop (like with **while**)
- the third one is the **increment** action, it is performed at each iteration just after the last action of the body.

The first and the last may be empty.

If the condition is false upon entry, the initialization is done but the body and the increment action are not executed.

```
// an interactive program

printf("type Y or N\n");

do {
    // wait for keyboard input
    c = getchar();
    // force lower case
    c |= ('a' - 'A');
    }
    while (
            ( c != 'y' ) &&
            ( c != 'n' )
          );

printf("ok\n");
```

```
/* Note : forcing to lower case above is a bit tricky
It is based on the fact that in the ASCII-code
upper-case and lower-case letters differ by 1 bit only,
which is returned by the expression : ('a' - 'A')
*/
```

1.9 The do-while statement

The **do** statement is a variant of the **while** statement where the condition is evaluated after each iteration.

It is useful when the information needed by the condition does not yet exist before the first iteration.

- the body will be executed at least once
- if the condition is made false during the execution of the body, this will be the last execution
- if nothing in the body causes the condition to change, infinite looping is likely to occur (except if the hardware or another execution thread changes the condition).

Notice the semicolon after the condition !

```
// another interactive program
printf("type Y or N\n");

// limit the number of iterations
cnt = 5;

while ( cnt-- )
  {
  c = getchar();
  // filter out the linefeed
  if ( c < ' ' )
     continue;
  c |= ('a' - 'A');
  if (
       ( c == 'y' ) ||
       ( c == 'n' )
     ) break;
   printf("bad answer, try again\n");
  }
```

Break and continue

The **break** and **continue** statements are special 'jump' instructions which may appear in the body of the **while**, **for** and **do** loops.

The **break** jump causes the control to get out of the loop immediately.
If it is placed in the middle of the body, the iterated task is left "partially done".

The **continue** jump, placed in the middle of the body, causes the control to skip the remainder of the body, leaving the current task "partially done", but allowing the next iteration to proceed normally.

Note : in the case of the **for** loop, **continue** does not skip the incrementation.

In the case of nested loops, **break** and **continue** act only on the innermost loop.

(There is also a evil **goto** jump, the use of which is strongly discouraged in every programming course)

```
y = 2007;
for ( month=1; month <= 12; month++ )
  {
  // compute the length of the current month
  switch ( month )
     {
     case 1  : len = 31;   break;
     case 2  :
        if ( y % 4 ) len = 28;
        else         len = 29;
                     break;
     case 3  : len = 31;   break;
     case 4  : len = 30;   break;
     case 5  : len = 31;   break;
     case 6  : len = 30;   break;
     case 7  : len = 31;   break;
     case 8  : len = 31;   break;
     case 9  : len = 30;   break;
     case 10 : len = 31;   break;
     case 11 : len = 30;   break;
     case 12 : len = 31;   break;
     }
  printf("len=%d\n", len );
  }
```

1.10 Switch

The switch statement is a multiple comparison decision engine.

It evaluates an integer expression once, compares it sequentially to a list of constants, and starts executing the body at the first match, until encountering a **break** statement.

Caution : people usually expect the processing for each case to be exclusive, but this is the case only if each processing ends with a **break** statement.
“ Do not forget the break”

If the comparison values are not all constants, switch should be replaced by a cascade of nested **if-else**.

```
// The lazy-but-wise programmer's solution
// to the same problem

y = 2007;
for ( month=1; month <= 12; month++ )
  {
  switch ( month )
     {
     case 2 :
        if ( y % 4 ) len = 28;
        else         len = 29;
                          break;
     case 4 :
     case 6 :
     case 9 :
     case 11 : len = 30;   break;
     default : len = 31;   break;
     }
  printf("len=%d\n", len );
  }
```

Switch, default and no-break

The optional **default** case matches any value.

If the **break** statement is omitted, the program execution continues, possibly over the next cases.

This may be useful for allow several cases to share the processing (or at least the last part of the processing).

```
// incrementing an value in a circular fashion
hour = (hour < 23)?(hour+1):0;


// assigning a bounded value
// here the bounds are inclusive

y = (x>xmax)?xmax:((x<xmin)?xmin:x);


// the same

y = ( x <= xmax )
    ? ( ( x >= xmin ) ? x : xmin )
    : xmax ;
```

1.11 Conditional expressions

The conditional expression is a kind of “if-else” which can be embedded in any expression.

The conditional expression is based on the following form :

condition **?** expr1 **:** expr0

- if the condition is true (non-zero), expr1 is evaluated and its result is returned
- else expr0 is evaluated and its result returned.

Conditional expressions may be nested.

```
// a math function of 1 variable
int Square( int x )
{
return( x * x );
}

// another math function
int Cube( int x )
{
// here a function is called from an expression
return( x * Square( x ) );
}

// a void function
void PrintCube( int x )
{
printf("%d exp 3 = %d\n",
        x, Cube( x ) );
}

int main(void)
{
// calling a void function
PrintCube( 25 );
return 0;
}
```

1.12 Functions

In order to avoid duplication of code, every computer hardware supports enclosing a piece of program in a "sub-program" or "subroutine" which may be "called" from various places in the program.

Other possible name for this scheme are "procedure" and "function".
A function may be called from within an expression, where it is substituted by its "return value", behaving like a mathematical function in a mathematical formula.

In C, we have only **functions**, the role of which may be
- computing a return value
- having side effects, like writing data somewhere
- both

If the return value of a function is not needed, there is no obligation to assign it to anything

If a function has nothing to return (having only side effects), its return type may be declared as "**void**"

```c
#include <math.h>

// a function prototype
double Dist( double X1, double Y1,
             double X2, double Y2 );

int main(void)
{
printf("distance is %f\n",
       Dist( 1.0, 2.2, 3.0, 5.6 ) );
// notice the floating point constants :
// 1.0 instead of 1 which would be an integer
return 0;
}

// a function late definition
double Dist( double X1, double Y1,
             double X2, double Y2 )
{
double DX, DY;
DX = X2 - X1; DY = Y2 - Y1;
// sqrt (square root) is a library function
// the prototype of which is in math.h
return sqrt( DX * DX + DY * DY );
}
```

Function header and prototype

A function header should provide :

- on the left side, the type of the returned value (or **void** if none)
- on the right side, in parentheses, the parameters declarations, each one with its own type

The header is followed by the function body enclosed in curly braces {}.

If the function expects no parameter, the parentheses should contain nothing or the **void** keyword (empty parentheses are now considered an obsolete usage)

The compiler need a function either to be defined before use, or at least to be declared by means of a prototype.

A prototype is constructed like the function header but

- it needs to be terminated by a semicolon
- the parameter names are ignored, but may be present for legibility

```c
// a function modifying one of its arguments

void Exclamations( int N )
{

if ( N < 0 )
   {
   printf("Internal Error\n");
   return;  // <== an early return in case of error
   }

while ( N-- )
   printf("!");
printf("\n");

// <== no explicit return needed here
}



int main(void)
{
Exclamations( 9 );
Exclamations( -1 );
return 0;
}
```

Parameter passing and return

When calling a function, the parentheses should contain values or expressions compatible with the prototype (or nothing if the function is declared with no parameter).

Parameters are passed “by value”, i.e. they are copied to temporary local variables. The function is allowed to overwrite their values, but those will be discarded when returning from the function.

Passing parameters “by reference” is not supported by standard C (but it is by C++).

The **return** statement causes the control to leave immediately the function (even from nested blocks and loops). Several **return**s are allowed in a function.

In the case of a non-void function, the **return** statement is mandatory and should convey an expression which is evaluated and copied to the calling expression.

(re-entrance, recursion : see appendix A.1)

```
int g;  // <== global (file scope)

void fun( int );

int main()
{
int y;  // <== local (function scope)
   // a block
   {
   int z;  // <== local (block scope)
   g = 1; y = 3;
   z = g + y; fun( z );
   }
x = z;  // BAD : compilation error here !!
        // no z is visible in current scope
return 0;
}

void fun( int y )
{
// Note : this y is unrelated with the one defined
// in the function above
g += y;
}
```

2. Data storage

2.1 Variable scope

The **scope** of a symbol is the code region from which it is accessible, or as we say “visible”.

In C, variables may be defined only :

- anywhere outside function definitions,
- as function parameters
- at the beginning of a function body
- at the beginning of a block (enclosed in {}) within a function body

A variable defined outside of any function definition is a “**global variable**”, it is visible from any function defined in the same file (the case of multiple files will be addressed in chapter 3)

A variable defined in a function or “**local variable**”

- is visible only from within this function.
- is not visible from within functions called by this function.

A variable defined in a block is visible only from within this block, and the blocks nested inside.

```
int g, x;  //  <== global (file scope)
void fun( int );

int main()
{
int y;  //  <== local (function scope)

g = 1; x = 2; y = 3;
fun( x + y );

printf("g=%d, x=%d, y=%d\n",
        g, x, y );
return 0;
}


void fun( int y )
{
int x;  //  <== local, masks the global x
x = y * y;
g += x;
}


// what is printed ?
// g=26, x=2, y=3
```

Variable usage

It is legal to have a local variable with the same name as a global one, in this case the global variable is masked (no longer accessible) to the local code.
It is also legal to have a block scope variable with the same name of a function scope variable, in this case the global variable is masked to the block inner code.

It is generally recommended to avoid variable masking, since it makes code maintenance difficult.

It is frequently said that "**global variables are evil**". Why do people think this ?
It has been observed that extensive use of global variables make code difficult to maintain and nearly impossible to reuse. They are said to cause "namespace pollution".

But global variables are sometimes necessary - a good way to be forgiven for using them is to pack them into a few **structures** (details to follow at the end of the chapter) - this idea was the starting point of "object oriented programming".

```
int total( int incr )
{  // a static variable is initialized only once
static int cnt=0;
cnt += incr;
return( cnt );
}

// a function aware of being called for the first time
void process()
{
static int flag=0;
if   ( flag )
     printf("processing...\n");
else {
     printf("initializing..\n");
     flag = 1;
     }
}

int main()
{
printf("%d\n", total( 1 ) );
printf("%d\n", total( 3 ) );
process(); process(); process();
return 0;
}
```

2.2 Storage classes

Data storage falls into three classes :

- **static**
- **automatic**
- **dynamic**

Static variables are permanent, for the whole duration of the process. They have a fixed position in memory.

Automatic variables are created temporarily when entering a function or a block, and deleted upon exit. They are stored in a memory area called **stack** (a LIFO or Last-In First-Out storage) or in processor registers.

Dynamic storage can be created and deleted explicitly using the **malloc()** and **free()** library functions (details in chapter 4).

By default :

- **global** variables are **static**
- **local** variables are **automatic**

By explicit use of the **static** prefix, it is possible to make a local variable static. It is then stored the same way as a global variable, but has a restricted visibility.

```
typedef unsigned char Byte ;

int main() {
int a, b, i; char c; double dd;

// the example from page 7,
// fixed by type casting
a = 2; b = 3;
// dd = a / b;  <== would give zero
dd = (double)a / (double)b;
printf("2/3 = %f\n", dd );


// an example showing trouble caused by signed type
c = 255;

i = c * 2;                          // Bad
printf("i = %d\n", i );

i = (unsigned char)c * 2;           // Good
printf("i = %d\n", i );

i = (Byte)c * 2;                    // Same
printf("i = %d\n", i );

return 0; }
```

2.3 Type casting

It is sometimes necessary, and generally recommended, to use an explicit **type casting** instead of letting an implicit type conversion happen.

It also helps to make the programmer's intentions clearer or to avoid some compiler warnings.

The syntax is a type name enclosed in parentheses, just before an expression element (variable, sub-expression or constant)

A common source of trouble with signed variables is that the conversion of a signed variable to a larger one causes the most significant bits to be filled with zeroes or ones according to the sign of the value.

It is possible to define a type synonym with the **typedef** keyword, some development kits use it to create types like BYTE (8-bit), WORD (16-bit), DWORD (32-bit), uint8, unit16, uint32, uint64 for unsigned explicit-size words.

```
int main() {
unsigned int a, b;

// an example showing trouble caused by unsigned

a = 2; b = 4;
// humans expect 2 - 4 to be lesser than zero...

if ( a - b < 0 )                // fail
   printf("a - b < 0 #0\n");

if ( (int)a - b < 0 )           // fail
   printf("a - b < 0 #1\n");

if ( ((int)(a - b)) < 0 )       // OK
   printf("a - b < 0 #2\n");

if ( a - (int)b < 0 )           // fail
   printf("a - b < 0 #3\n");

if ( (int)a - (int)b < 0 )      // OK
   printf("a - b < 0 #4\n");


return 0; }
```

More type casting

Conversion from **unsigned** to **signed** and conversely does not modify the data, but changes the behaviour of the comparison, multiplication, division and right shift operators.

In the example, the behaviour of the subtract operator '-' is not changed by the casting, but the behaviour of the comparison operator '<' is.

```
int main() {
int x;
// a pointer to an integer
int * iptr;


// making iptr point to x
// (which holds no data for the moment)
iptr = &x;

* iptr = 5;
x += 2;

// what will be printed ?
printf("%d\n", (* iptr) * 3 );
printf("%d\n", *(&x) );

return 0;
}



// answer :
// 21    ( (5 + 2) * 3 )
// 7     (just x itself)
```

2.4 Pointers

A pointer is a variable which holds the memory address where the data it points to is stored.

Pointers are declared in association with the type of the data they point to, so there are as many pointer types as there are scalar types, plus one which is the void pointer.
The void pointer points to raw, untyped data.

The * operator is used to assign to or from the data which is pointed to, this is called “dereferencing”.

The & operator is used to get the address of a variable in order to assign it to a pointer (“referencing”).

Declaration syntax : the * in the pointer declaration may be interpreted as follows :

```
int * my_ptr;
```

is equivalent to

```
int (* my_ptr);
```

which means that “dereferencing **my_ptr** gives an int”.

```
// this function has to return 2 results : area and
// perimeter.
// It expects 2 integers and two pointers.
// the pointers should point to some allocated storage

void calrec( int W, int L,
             int *A, int *P )
{
*A = W * L;
*P = 2 * ( W + L );
}


int main()
{
int area, perim;

calrec( 3, 4, &area, &perim );

printf("a=%d, p=%d\n", area, perim );

return 0;
}


// prints : a=12, p=14
```

Passing pointers

We have seen that the arguments in a function call are copied to local variables, so the function cannot make any permanent change to them.

But by passing pointers to the arguments instead of the values, it is possible for the function to change the values in the calling code's data space.

This is a convenient solution for having a function providing more than one result without using any global variable.

```
int main() {
int i;
// an uninitialized array for 5 elements
int prime[5];

prime[0] = 2;
prime[1] = 3;
prime[2] = 5;
prime[3] = 7;
prime[4] = 11;

// Note : prime[5] does not exist - using it may cause
// the program to crash

// any integer expression may be used as subscript
for ( i = 0; i < 5; i++ )
    printf("%d,", prime[i] );

return 0;
}
```

2.5 One dimension arrays

An array (sometimes called vector) is a collection of values of a given type, indexed by an integer number by means of the “subscript operator” [].

In C, static and automatic arrays should be defined with a constant size (except in C99).

The index of the first element is always 0.
The index of the last element of an N-element array is therefore N-1.

There is no implicit check of the index, out-of-bounds indexing leads to corrupted data or memory fault exception, like pointer errors.

```
// a function for printing the contents of an array
void printa1( int size, int * buf )
{
int i;
for ( i = 0; i < size; i++ )
    printf("%d,", buf[i] );
printf("\n");
}

int main() {
int i;
// an initialized array - notice that the size will be
// adjusted automatically at compile time
int arr1[] = { 7, 11, 13, 17, 23 };
// an uninitialized array
int arr2[5];

// doing a copy
for ( i = 0; i < 5; i++ )
    arr2[i] = arr1[i];

// printing the contents of arr2[]
printa1( 5, arr2 );

return 0;
}
```

Array initialization and base

An array may be initialized by putting a list of values in braces {}.
In this case, the array may be declared without size information, then the compiler will derive the array size from the number of values in the list.
Otherwise, the number of values in the list must be smaller than or equal to the array size.

The name of an array may be used to get the base address of the array :

```
mypointer = myarray;
```

is equivalent to

```
mypointer = &(myarray[0]);
```

Then :

```
*mypointer;
```

is the same as

```
myarray[0];
```

The base of an array can be passed to a function, like a pointer. The array is not copied, only the first element's address is passed.

```
// a function for printing the contents of an array
void printa1( int size, int * buf )
{
int i;
for ( i = 0; i < size; i++ )
    printf("%d,", buf[i] );
printf("\n");
}

// another function doing the same
// using pointer incrementation
// - no local variable needed -
void printa2( int size, int * buf )
{
while( size-- )
    printf("%d,", *(buf++) );
printf("\n");
}

int main() {
int arr1[] = { 7, 11, 13, 17, 23 };

printa1( 5, arr1 );
printa2( 5, arr1 );
return 0;
}
```

2.6 Pointer arithmetics

The addition of an integer expression to a pointer is legal, and performs the following :
the expression is multiplied by the size of the type pointed to, and the result is added to the address value to give a new pointer.

It makes sense if the underlying data is an array : adding N to a pointer pointing to element A points to the Nth element after A.

In particular, incrementing a pointer makes it point to the next element, decrementing it makes it point to the previous element.

```
my_array[i]
```

is equivalent to

```
*(my_array + i)
```

Both syntaxes are legal with an array name as well as with a pointer name.

```
// endianness tests
int main() {
int n, i;
unsigned char * cbase;

n = 0xEFBEADDE;

cbase = (unsigned char *)&n;
for ( i = 0; i < sizeof(int); i++ )
    printf("%X", *(cbase + i) );
printf("\n");

// same thing, array style
for ( i = 0; i < sizeof(int); i++ )
    printf("%X", cbase[i] );
printf("\n");



/* prints :
 EFBEADDE
 on big-endian machines (Sun SPARC, MAC 68k)
 DEADBEEF
 on little-endian machines (intel x86)
*/
return 0; }
```

Casting pointers

It is possible to cast a pointer to another pointer type, this performs no conversion on the pointer's value but changes the way arithmetics work on it and the way dereferencing is done.

In the example we introduce the **sizeof()** operator which return the size (in char units) of any type or variable.

As we have seen a pointer and an array base name behave very similarly, but there are some differences :

- assigning to a pointer is legal, but an array base address cannot be changed
- sizeof() on a pointer returns the size of the pointer itself (which depends on the computer architecture, not on the data pointed to), while on a fixed array it returns the total size of the data

```
// pointer errors
int * fun()
{
int i; int vect[5];
// a pointer to an integer
int * iptr;

* iptr = 9;  // <== BAD, iptr points to nothing

for ( i = 0; i < 5; i++ )
    vect[i] = i * i;

for ( i = 0; i < 5; i++ )
    vect[i-1] = vect[i] * 2; // <== BAD
     // refers to vect[-1], out of bounds

iptr = vect;
iptr += 5;
* iptr = 2;   // <== BAD, out of bounds

return vect;   // <== BAD, addr of auto array
}
```

Pointer errors

Pointers are a very generous source of programming errors.
Some examples :

- dereferencing a pointer which was not previously initialized
- incrementing a pointer beyond the allocated memory
- returning a pointer to an automatic variable

A pointer error may result in :

- corrupted data possibly undetected
- fatal memory fault detected by the operating system

```
// a function returning the length of a string
// (null terminator excluded)
int str_len( char * str )
{
int len = 0;
while( *(str++) )
   len++;
return len;
}

int main()
{
// 3 arrays of char, each one with a null terminator
char * s1 = "hello ";
char s2[] =
        {'f','o','l','k','s',0 };
char s3[3];
s3[0] = 33;   // ASCII code of  '!'
s3[1] = 10;   // ASCII code of new line AKA '\n'
s3[2] = 0;

// all directly used as format string in printf
printf(s1); printf(s2); printf(s3);

// prints :  hello folks!
return 0;}
```

2.7 Character strings

In spite of the importance of the matter, there is no specific type for character strings (text) in C.

A text string is smartly handled in an **array of char**.
By convention, the last character of the string is followed by a char the value of which is zero : the "**null terminator**".
The size of such a string is limited only by the available memory, and it may contain any number of text lines.

The compiler accepts "**string constants**" which are pieces of text enclosed in double quotes :

- it stores the characters in an initialized static array of chars (translating special sequences like \n, \", \\)
- it appends the null terminator
- it returns the address of the array

This is exactly all the support the compiler gives for text.
But many library functions are available for string processing.

Now one of the secrets of **printf** may be revealed : it expects a **pointer to char** as first argument.

(ASCII code : see appendix A.11)

```
int main() {

// an array of pointers to char
// each pointing to a string
char * colors[] =
          { "red", "blue", "green" };
// a pointer to pointer to char
char ** ppc;

// dereferencing a “row” :
// (Note : in printf, %s is the placeholder for a string)
printf("%s,", colors[2] );

// dereferencing a single element
printf("%c,", colors[2][4] );
// dereferencing the same
printf("%c,", *(*(colors + 2) + 4) );

// an explicit pointer- to - pointer
ppc = colors + 1;
printf("%s,", *ppc );
printf("%c\n", **ppc );

return 0;}

// prints : green,n,n,blue,b
```

2.8 Arrays of arrays, pointers to pointers

The most general way to create a bidimensional array is to create an array of arrays, which is in fact an array of pointers, each pointing to the first element of a simple array.

A common example is the “command line arguments” which the operating system passes to the main() function as an array of strings :

`int main( int argc, char ** argv )`
argv is a pointer to pointer to char, which may be considered as the base of an array of pointers to char.
argc is the number of elements of this array, i.e. the number of strings.
Each element of this array is a pointer to the first char of a string.
The length of each of these strings can be known only by searching for the terminating null.

For example, argv[0][0] is the first char of the first command line argument (the program name itself), argv[argc-1] is a pointer to the first char of the last argument.

(rectangular array : see appendix A.2)

```
// declaring the structure
// (creates no storage for the moment)
struct Date {
int day;
int month;
int year;
char * month_name;
};

int main() {
// creating variables
struct Date today, tomorrow;

today.day = 30;
today.month = 12;
today.year = 2007;
today.month_name = "December";
// copying the whole structure at once
tomorrow = today;

// changing one value in the copied structure
tomorrow.day += 1;
// here normally we should handle the overflows...

return 0;
}
```

2.9 Structures

A **structure** is an heterogeneous aggregate data type, a container for a collection of variables of arbitrary types, called the structure **members** or **fields**.

Members may be simple types, pointers, fixed-size arrays or structures. The scope of the member names is limited to a given structure.

Assigning to or from a structure member is done by means of the '.' (dot) operator.

The total size of a structure is fixed at compilation time, it is returned by the **sizeof()** operator (due to alignment and padding, it may be greater than the sum of the member sizes).

A structure can be copied at once by assigning to a compatible structure.
As a consequence, if a structure is passed to a function or returned by a function, an entire copy is done.

```c
// declaring a structure tag
struct DateStruct1 {
int day;
int month;
int year;
char * month_name;
};

// declaring a structure type
typedef struct {
int day;
int month;
int year;
char * month_name;
} DateStruct2;

int main() {

// declaring variables
struct DateStruct1 d1;
DateStruct2 d2;

// using them...
d1.day = d2.day;

return 0; }
```

Structure declaration styles

A structure should be declared before use.
Unfortunately two redundant “styles” are in common use.

One may declare a “**structure tag**” which can be used to declare structure variables in association with the **struct** keyword.

In the other hand, one may declare a **structure type** by means of the **typedef** keyword, which can be used to declare variables and cast types the same way as any type name.

The second style is considered as more “modern”.

(unions, enums : see appendixes A.3, A.4)

```c
// creating a structured type
typedef struct {
unsigned int h;
unsigned int mn;
} Time;

// a function to add some minutes to the given time
void add_mn( Time * t,
             unsigned int mn )
{
t->mn += mn;
t->h += t->mn / 60;
t->mn %= 60;
}

int main() {
Time mytime;
Time * t = &mytime;
// it is 14h35
t->h = 14; t->mn = 35;
// 100 minutes later
add_mn( t, 100 );
printf("%d:%d\n", t->h, t->mn );

return 0;
}
```

Pointers to structures

In order to avoid unnecessary copies and to allow a function to modify the contents of a structure, **pointers to structures** are frequently used.

Dereferencing a structure member from a pointer to the structure can be done the regular way :

```c
(* my_ptr).my_member
```

or by means of the '`->`' (arrow) operator, the interest of which is purely aesthetic :

```c
my_ptr->my_member
```

Arrays of structures are welcome, and pointer arithmetics apply to pointers to structures as one would expect.

Pointer casting applies to pointers to structures too, and may be useful when dealing with structures which have a similar layout at least at the beginning of their memory span.

(pointers to functions : see appendix A.5)

3. Source code organization

C was designed for working on large projects, like a complete operating system.
For facilitating this, provisions are made to :

- split the source code in many (small) files
- compile them separately
- avoid duplicating common declarations
- combine several programming languages : assembler, C, etc...
- create and use libraries of standard precompiled functions

For this purpose were created :

- the **C preprocessor**
- the **linker** and the librarian
- the **make** utility

3.1 The C preprocessor

The preprocessor, as the name implies, processes the source text before the compilation.
Its invocation is automatically done by the compiler.
For debugging purpose, there is usually a compiler option to have a look at the preprocessor output (which is otherwise hidden).
The preprocessor performs mainly text substitution :

- file **inclusion**
- **macro** substitution, simple or with formal parameters
- **conditional** processing

The preprocessor language is **line-formatted**, unlike C itself. The preprocessor knows very little of C !

```
// including a file from a subdirectory of a
// standard include path
#include <sys/types.h>

// including a file from a subdirectory of the
// current directory
#include "hardware/uart.h"

// including a file relative to the current directory
#include "../common/structures.h"



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

specifying an include path with the GNU compiler :
gcc -I/home/shrek/project1/include


specifying an include path with the Microsoft
compiler :
cl /I"C:\proj1\inc"
or
set INCLUDE=C:\proj1\inc;%INCLUDE%

*/
```

File inclusion

The **#include** directive causes itself to be replaced by the complete contents of the designated file.
It comes in two flavours :

```
#include <file>
#include "file"
```

In the first case, the file is searched in default directories, the path of which may be :

- built-in (hard-coded) in the development system
- taken from an environment variable
- taken from a command line option

So the first flavour is convenient for standard header files.

In the second case, the file is first searched in the current directory.
The second flavour is suitable for user-defined files

In any case, absolute pathnames are not recommended.

```
// project option
#define VERBOSE
// simple constants (no ';' at the end !!)
#define PI  3.14
#define MSB32  0x80000000
#define BUFSIZE (1<<10)     // a power of 2
#define BUFMASK (BUFSIZE-1)


// updating (ok even if PI does not exist)
#undef  PI
#define PI 3.14159

// multi-line macro
#define PRIMER { 2, 3, 5, 7, 11, \
                 13, 17, 19, 23, 29 }

// using them
int buf[BUFSIZE];
int prime[200] = PRIMER;

// writing a value in the circular buffer
// (never overflows)
buf[(++i)&BUFMASK] = x;
```

Simple macros

The **#define** directive creates a macro, which is a symbol possibly associated with some contents.

- an empty macro may be useful by itself for expressing a project option, since its existence may be tested by a conditional directive
- a macro with contents may be useful for defining some constant, or some project option

Upper-case names for macros is a common, recommended usage.

The macro body ends with the line terminator, unless the terminator is "escaped" with the backslash character '\'

Alternatively, macros may be defined "from outside", by means of command-line options (option -D in the case of gcc)

Some predefined macros may exist, for example for operating system identification, compiler version, etc...

The **#undef** directive cancels a macro.

```
// identifying the operating system
#ifdef _WIN32
  #include <io.h>
#else
  #define O_BINARY 0
#endif

// handling version-specific issues
#ifdef _MSC_VER
  #if _MSC_VER < 1310
  typedef _int64 BIGLONG;
  #else
  typedef long long BIGLONG;
  #endif
#else
typedef long long BIGLONG;
#endif

// switch-like sequence
#if LEVEL == 1
printf("level 1\n");
#elif LEVEL == 4
printf("level 4\n");
#elif LEVEL == 9
printf("level 9\n");
#endif
```

Conditional processing

A portion of code may be conditionally compiled according to :

- the existence of a macro : **#ifdef** or **#ifndef**
- the value of a macro : **#if** condition

A mandatory **#endif** terminates the conditional processing.

An optional **#else** may appear before **#endif**.

Nesting conditional processings is facilitated by **#elif** which means ‘else if’.

```
// generic processing (works with many types)
#define MIN(a,b) ((a)<(b)?(a):(b))

int main() {
int u, v, w;
double X, Y, Z;

// works with ints
u = 2; v = 5;
w = MIN( u, v );

// works with doubles
X = 0.2; Y = 0.02;
Z = MIN( X, Y );

printf("w=%d, Z=%g\n", w, Z );


return 0;
}
```

Macros with parameters

A macro which has an opening parenthesis '(' appended to its name (with no space between) may accept arguments, and behave 'like a function'.

This may appear interesting for pieces of code which are too small to justify the overhead of function calling and return, or the complication of passing pointers.

It may also provide a generic, type independent processing.

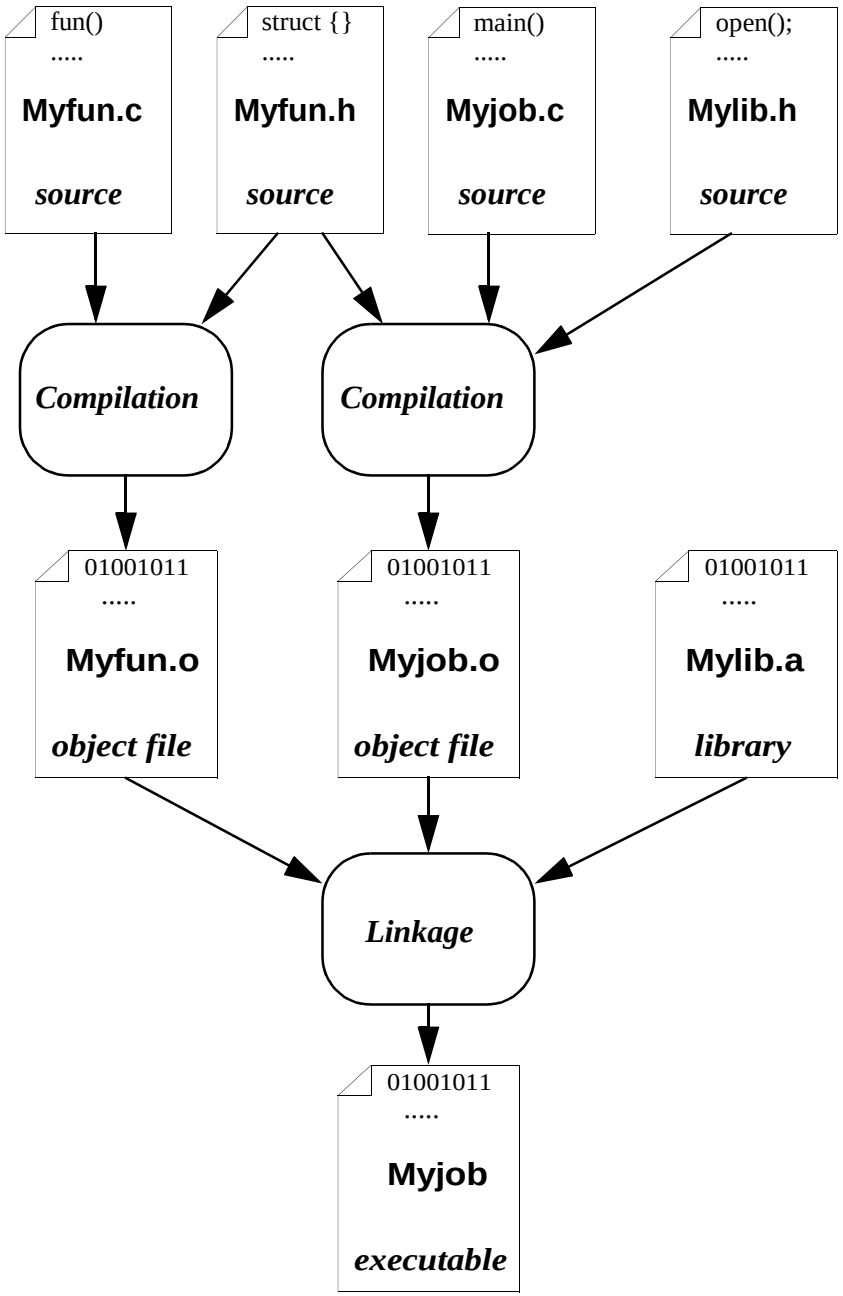
In the other hand it is known to make programs difficult to maintain, and is strongly discouraged by many authors.

Also enclosing each argument and the complete body in parentheses avoids some dangers, but not all.

To avoid these function-like macros, alternate solutions were proposed :

- inline functions (C99, recent compilers, C++)
- templates (C++)

(more on these macros : see appendix A.6)



3.2 Separate compilation

The making of a program involves 2 steps :

- the compilation itself, which produces “object code”, which is relocatable machine code
- the linking, which binds together pieces of object code to make an executable file

This allows “separate compilation”, the benefit of which is :

- saving compilation time
- possibility of mixing languages
- name space separation (to some degree)
- hiding implementation details, preserving IP

An ***object*** file contains the object code from a single compilation unit.

A ***library*** contains the results of merging several object file and creating a kind of index allowing the linker to extract the functions needed for building the project.

A ***header*** file is a source code file containing only declarations required to make use of an already compiled object file or library. The header file is intended to be included in source code.

```
/* ------------ contents of Myfun.c ----------------

#include <stdio.h>
#include "Myfun.h"

static int min( int a, int b)
{ return( a < b ? a : b ); }

void printimin()
{ printf("%d\n", min(A,B)); }

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

/* ------------ contents of Myfun.h ----------------
extern int A, B;
void printimin(void);
 ---------------------------------------------------*/

#include "Myfun.h"
int A, B;

int main() {
A = 222; B = 77;
printimin();
return 0;
}
```

Symbol scope for the linker

In order to enable the linker to make the connection between a variable or a function defined in a object file and referenced by another, the symbol should be "exported" in one side and "imported" in the other.

In C, all functions and all global variables are exported by default.

In order to prevent namespace pollution, it is possible to restrict the scope of those items (i.e. preventing their exportation) by means of the **static** keyword.
This usage of the static keyword is strongly misleading, having no relation with its meaning when used with local variables.

In C, all functions not defined in the current compilation unit are imported.

Global variables are imported only if they are declared with the **extern** keyword.

"**static** global variables may be forgiven, global variables are evil, **extern** global variables are worse"

```
// a few lines from <stdio.h>

// avoid repeating code in case of multiple inclusions
// which are practically impossible to avoid
#ifndef _STDIO_H_
#define _STDIO_H_

#include "_ansi.h"

#include <stddef.h>
#include <stdarg.h>

#include <sys/reent.h>
#include <sys/types.h>

typedef __FILE FILE;

// contents skipped...

#endif /* _STDIO_H_ */
```

Header files

A header file (.h) is made out of C source text but should not contain anything immediately generating object code, like function bodies or global variables creation.

The reason is that a header may (and generally should) be included in several units of the same project, but object code should not contain duplicated elements.

A good practice is to put all functions prototypes and relevant type definitions and constants in a header file with the same name as the source file, and to include this header in this source file itself. This way, the consistency between the function prototypes and the function definitions will be checked.

```
#! /bin/bash
# example in the Unix style

# option -c for “compile to object file”
# option -Wall for “Warnings ; all “
gcc -Wall -c Myfun.c
gcc -Wall -c Myjob.c

# option -o for naming the output file
gcc -o Myjob Myjob.o Myfun.o

# # # # # # # # # # # # # # # # # # # # # # # #

rem example in the Windows ‘.bat’ style

rem option /c for ‘compile to object file’
rem option -Wall for “Warnings ; all “
cl /Wall /c Myfun.c
cl /Wall /c Myjob.c

rem option /Fe for naming the executable file
cl /FeMyjob.exe Myjob.obj Myfun.obj
```

Building steps

In the case of a single file, the compiler may take care of calling the preprocessor, compiling and calling the linker all in a row.

In the case of multiple files, one should first invoke the compiler for each compilation unit, with some option telling not to call the linker.

A compilation unit is normally made out of a single C source file and the headers it includes.

Then the linker should be invoked, with the list of object files. This can be done through the compiler too.

The linker will search for objects first in the designated object files, then in the libraries found in directories, the path of which may be :

- built-in (hard-coded) in the development system
- taken from an environment variable
- taken from a command line option

It is also possible to use a command line option to explicitly specify the name of a library.

```
# example makefile (Unix style)
# if the file is named 'Makefile', just type make

# defining a variable
CCOPT= -Wall -O2

# the first target is the default target
Myjob : Myfun.o Myjob.o
        gcc -o Myjob Myfun.o Myjob.o

Myfun.o : Myfun.c
        gcc $(CCOPT) -c Myfun.c

Myjob.o : Myjob.c
        gcc $(CCOPT) -c Myjob.c

# type make clean to remove intermediate files
clean :
        rm *.o

# dependencies
Myfun.c : Myfun.h
Myjob.c : Myfun.h
```

3.3 The make utility

Make is a project maintenance utility, which performs tasks according to a script called “the makefile”, written in a specific language.

It differs from other script interpreters like the shell by the fact that it automatically determines what is necessary to do in order to update the project, taking into account the modification date of each file.

Each task is associated with a target, by a rule which specifies :

- a dependency list describing the prerequisites (simple files or other targets)
- a command line (to be passed to the system shell)

The base syntax is :

```
target : prerequisites
        command
```

Oddly enough, the command must be preceded by a TAB character (not a free form blank space).

Make supports internal variables, like the shell, with some syntax differences.

(more on make : see appendix A.7)

4. The standard library

Since the library function are involved in interacting with the operating system, the actual contents of the standard library are determined according to :

- The C language standards (which propose an informative list of functions)
- The operating system API standards (Application Program Interface), like POSIX
- The operating system kernel's "system calls", which are the lowest layer of I/O access.

The POSIX (Portable Operating System Interface for Computer Environments) was created by the IEEE in order to facilitate portable programming. It is merely an attempt to unify the various Unix families, but is also partially implemented in non-Unix systems.

In most cases, the standard libraries have grown to several hundreds functions.
In this document only the most used will be presented, considering the following categories :

- file I/O and console I/O (character streams)
- string functions
- dynamic memory allocation
- date and time functions
- math functions

Other interesting categories like :

- network (Ethernet, Internet)
- file system (scanning directories)
- environment

are left to the "advanced training"

4.1 Character stream IO

The stream I/O system is a layer on top of the “low level” file I/O based on system calls.
Both systems offer access to files, so confusion should be avoided :

- The stream system represents an open file with a **FILE ***, a pointer to a **FILE** structure

- The low level functions represents an open file with a “**file descriptor**” which is an **integer** number

Character streams are not used only for files but also for character devices like terminals (keyboard and text display).

Three streams are predefined and ready open by the system (of type FILE *) :
- **stdin** : standard input (keyboard)
- **stdout** : standard output (screen)
- **stderr** : standard error (screen too)

Error messages are separated from normal data to permit redirecting independently these streams to files.

Redirection may be caused from outside (by the shell) or from inside (using the freopen() function).

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

int main( int argc, char **argv ) {
FILE * my_in;
FILE * my_out;
if ( argc < 3 )
   {
   printf("filenames needed\n");
   exit(1);   // terminating...
   }
my_in = fopen( argv[1], "r" );
if ( my_in == NULL )
   {
   printf("%s not found\n",
            argv[1] );
   exit(1);   // exit is declared in stdlib.h
   }
my_out = fopen( argv[2], "w" );
if ( my_out == NULL )
   {
   printf("%s not opened\n",
            argv[2] );
   exit(1);
   }
```

Opening and closing

Opening a file as a stream is done with the **fopen()** function, the second argument of which is a control string according to:

- "**r**" : open for reading only (file should exist)
- "**w**" : open or create for writing only (file is emptied if it already exists)
- "**a**" : open or create for appending (writing new data at the end of existing data)
- "r+", "w+, "a+" : read-write access (see the manual)

This operation may occasionally fail, in this case the function returns the NULL value (the "null pointer").

It is mandatory to check this before any attempt to access to the stream - dereferencing the NULL pointer may cause the program to crash.

Closing streams with **fclose()** is not mandatory, since it is done automatically upon normal termination of the program.
But it is recommended, to make sure that all buffered data has been written to the file (**fflush()** does this too).

```
// the previous page example continued...
{
char lbuf[256];

printf("Ok to proceed ? ");
// without fflush, the displaying the message would be
// eventually delayed to the next end of line char
fflush( stdout );
fgets( lbuf, 5, stdin );
if ( lbuf[0] != 'y' )
   return(1);
// note : return exits from the current function
// while exit() terminates the program
// even from inside nested function calls

// This loop copies only the line beginning with '#'
while ( fgets( lbuf, 256, my_in ))
   {
   if ( *lbuf == '#' )
      fputs( lbuf, my_out );
   }
} // end of block
return 0;
} // end of main()
```

Character and line text I/O

Functions **putc()** and **getc()** move one character at a time.
getc() does not return a char but an int. This enables it to return the special constant EOF when reaching the end of the file (this constant would not fit in a char)

putchar() is like **putc()** on the stdout stream,
getchar() is like **getc()** on the stdin stream.

fputs() outputs a string, which is an array of char with a null terminator. (The null is not written to the file. It is recommended to never put nulls in text files).

puts() is like **fputs()** on the stdout stream, but it appends a newline character at the end of the string.

fgets() reads text until encountering a newline character, or reaching the given length minus one.
The newline char (if any) is stored and a null character is appended. It is the programmer's duty to provide an allocated buffer and indicate its capacity.
fgets() returns a null pointer when at end of file.
gets() (same for stdin) is deprecated.

```
// processing raw data by fixed-size blocks

#define BUFSIZE 1024

unsigned char src[BUFSIZE];

int cnt;
do {
   cnt = fread( my_in, src, BUFSIZE );
    //
    // some processing on src to be inserted here
    //
    wcnt = fwrite( my_in, src, cnt );
    if ( wcnt != cnt )
       {
       // write error handling here (disk full...)
       }
    }
    while ( cnt == BUFSIZE );
```

Raw data access

The library maintains a “current position” value for each open stream. It is possible to change this value in order to seek data at an arbitrary position (if the device supports seeking).

This makes more sense with files which are organized as fixed length blocks instead of line formatted text or free-form text.

Functions **fwrite()** and **fread()** are convenient for moving fixed length data blocks.
They return the count of bytes actually moved, which should always be checked.

Functions **fgetpos()** and **fsetpos()** manipulate directly the current position, as **ftell()** and **fseek()** do.

(binary files : see appendix A.8)
(“ big” files : see appendix A.9)

```c
// interactive input

#include <stdio.h>
// atoi, atof are declared in stdlib
#include <stdlib.h>

int main() {
char lbuf[256]; int pin;
double a;

printf("give your PIN number : ");
fflush( stdout );
fgets( lbuf, 40, stdin );
pin = atoi( lbuf );
printf("PIN = %d\n", pin );

printf("give the amount : ");
fflush( stdout );
fgets( lbuf, 40, stdin );
a = atof( lbuf );
printf("amount = %f\n", a );

return 0;
}
```

4.2 String processing

Parsing numbers from a text source (like a string read from a stream) can be done with the following functions :

- **atoi()** for int
- **atol()** for long
- **atoll()** for long long (if supported)
- **atof()** for double

Leading space and trailing “garbage” is ignored, and these functions perform no error checking.

```c
// construction of a formatted string in several steps

#include <stdio.h>

int main() {
int x = 24;
char lbuf[256]; int pos;

// pos is the index of the next character to be written
// it is also the index of the null terminator

pos = sprintf( lbuf, "data are " );

pos += sprintf( lbuf+pos, "%d", x );

lbuf[pos++] = ',';

x = 36;
pos += sprintf( lbuf+pos, "%d", x );

// Note : here we have to be sure to have a
// null terminator in lbuf (sprintf did put one)
fprintf( stdout, "%s\n", lbuf );
return 0;
}
// prints data are 24,36
```

Formatted output

The formatted output functions take a “format string” or “template” as one of their arguments.
The text from the format string is reproduced in the output, except the “placeholders” or “conversion specifiers” which are replaced by converted values of the subsequent arguments.

These functions accept a variable number of arguments, which is rather exceptional in C.

- **sprintf()** puts the result in a string (which should have been allocated by the calling function)
- **snprintf()** does the same, with a size safety limit
- **fprintf()** puts the result in an output stream (file or standard stream)
- **printf()** is a short-cut for **fprintf()** on **stdout**.

These functions return the number of non-null chars actually written.

The most general form is **sprintf()**, since its output can be sent to a stream, but also to a GUI (Graphics User Interface) display function, or be used as a filename.

```c
#include <stdio.h>
int main() {
double frac;
int percent, permil;

frac = 2.0/3.0;
printf("P = %f%%\n", frac * 100.0 );

percent = (int)(frac * 100.0);
printf("P = %d%c\n", percent, '%' );

permil = (int)(frac * 1000.0);
printf("P = %d.%d%%\n", permil / 10,
        permil % 10 );
// Note : see also 'precision' on next page

frac *= 100.0;
printf("F = %g\n", frac );
frac *= 100000.0;
printf("F = %g\n", frac );
return 0; }
// prints :  P = 66.666667%
             P = 66%
             P = 66.6%
             F = 66.6667
             F = 6.66667e+006
```

Conversion specifiers

Each specifier begins with the '%' character and ends with a letter specifying the type of conversion.
To insert an ordinary '%' in the output, use a "%%" sequence.

Simplified list of conversion types :

type	arg. type	output
c	char	single character
d	int	decimal
u	unsigned int	unsigned decimal
x or X	(unsigned) int	unsigned hexadecimal
e or E	double	decimal with exponent
f	double	fixed point decimal
g	double	optimized decimal
s	char *	copy of the string

```c
#include <stdio.h>
int main() {
double frac; int digits, i;
char form[32]; char *sa, *sb;

// fixed point display
frac = 2.0/3.0;
printf("p=%4.1f%%\n", frac * 100.0 );

// showing right and left justification
sa = "name"; sb = "Donald";
printf("|%9s : %-9s|\n", sa, sb );
sa = "function"; sb = "duck";
printf("|%9s : %-9s|\n", sa, sb );
sa = "job id"; i = 1234;
printf("|%9s : %-9d|\n", sa, i );

return 0; }

/* prints :
p=66.7%
|     name : Donald   |
| function : duck     |
|   job id : 1234     |

*/
```

Width and precision

The **%** sign may be optionally followed by :

- a '-' sign meaning 'left justify'
- a number representing the desired width
- a period followed by a number representing the 'precision'
- a type modifier

The desired width is in fact a minimal width, useful for producing aligned columns. Padding is done with spaces, except in the case of right-justified numbers, where leading zeroes are used if the width value has a leading zero.

For **%e**, **%E** and **%f** conversions, the precision is the number of digits after the decimal point (rounding is implied). The default precision is 6.

For the **%s** conversion, the precision is the maximum width of the string (a larger string is truncated).

The type modifier may be "l" for long or "ll" for long long (if supported).

```c
#include <stdio.h>
int main( int argc, char **argv ) {
int cnt, n1, n2;
double f; char s[256];
if ( argc < 2 )
   { fprintf( stderr, "arg ?\n" );
     exit(1); }

cnt =
 sscanf( argv[1],"%d %d", &n1, &n2 );
if ( cnt )
   printf("n1 found : %d\n", n1 );
if ( cnt > 1 )
   printf("n2 found : %d\n", n2 );

// note : below %lf stands for double
cnt = sscanf( argv[1], "%lf", &f );
if ( cnt )
   printf("f found : %g\n", f );

// Note : A program argument containing space
// should be quoted
```

Formatted input

Formatted input driven by a format string is performed by the following functions :

- **sscanf()** reads data from a string
- **fscanf()** reads data from an input stream (file or standard stream)
- **scanf()** is a short-cut for **fscanf()** on **stdin**.

Like the formatted output functions (printf family), these functions accept a format string followed by a variable number of arguments.

All of these subsequent arguments should be pointers to existing variables, prepared to accept the values resulting from the scan.

These functions return the number of successfully converted elements, or eventually the EOF constant. Blank space including tabs and newlines is skipped.

Using **scanf()** in interactive programs gives very little flexibility, **fgets()** followed by **sscanf()** is better.

```c
#include <stdio.h>
#include <string.h>

int main()
{
char sbuf[256]; int len;

strcpy( sbuf, "ello ");
strcat( sbuf, "World" );

// let us try to shift the text right by 1 position
len = strlen( sbuf );
len += 1;  // to include the null
memmove( sbuf+1, sbuf, len );

// now, insert the missing 'H'
sbuf[0] = 'H';

puts( sbuf );
return 0;
}



// prints Hello World
```

String and array copy

The length of a **null**-terminated string is returned by the **strlen()** function (the terminator is not counted).

The following string copy functions are available :

- **strcpy()** copies a string up to the **null** terminator
- **strncpy()** copies a string up to the given length, padding with **null** characters if necessary.
- **strcat()** copies the source string at the end of the destination string, overwriting the original terminator.
- **strncat()** does the same but with a length limitation.

Source and destination areas should not overlap.
Using **strcpy()** and **strcat()** is unsafe, unless the length of the source string is reliably bounded according to the available space in the destination memory.

The array copy functions are packed with the string functions, but can operate on generic data, since they accept **void** pointer and ignore **null** terminators.

- **memcpy()** : copies N bytes
- **memmove()** : same, but supports overlapping source and destination.

```c
#include <stdio.h>
#include <string.h>

int main() {

printf("%2d\n",
       strcmp("abc", "abcd") );
printf("%2d\n",
       strncmp("abc", "abcd", 3 ) );
printf("%2d\n",
       strcmp("aabc", "abc") );
printf("%2d\n",
       strcmp("abc", "Abc") );
printf("%2d\n",
       strncmp("abx", "abx", 5 ) );

return 0;
}

/* prints :
-1
 0
-1
 1
 0
*/
```

String comparison

The string comparison functions do not only check whether the strings are identical, but also tries to sort them.

They return zero if the strings are identical.
Else, they return a value which has the same sign as the difference between the first differing pair of chars, allowing kind of alphabetical ordering.

The comparison is case-sensitive.

- **strcmp()** compares strings up to the length of the shortest,
- **strncmp()** is similar but with a given length limit, allowing comparing only the first N characters.

With **strcmp()**, strings should have the same length to be considered equal.
With **strncmp()**, unequal length strings may be considered equal if they have the same “beginning”.

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

int main()
{
int *buf;
int capa;

capa = 10000;
buf = malloc( capa * sizeof(int) );
if ( buf == NULL )
   {
   fprintf( stderr,
      "malloc failed for %u bytes\n",
       capa * sizeof(int) );
   exit(1);
   }

// now we can use buf like an array of 10000 ints :
buf[9999] = 111;
//...

free( buf );

return 0;
}
```

4.3 Dynamic memory

The **malloc()** function returns a pointer to an allocated area of the requested size, or a NULL pointer if the request cannot be satisfied.

The pointer should be checked upon return from **malloc()**, since dereferencing a NULL pointer is likely to cause program crash.

The allocated memory is freed upon program termination, or explicitly by calling the **free()** function.

Changing the size of an allocated area is possible with **realloc()**. Caution : This may cause the existing data to be moved.

Calling the **free()** function with an incorrect pointer (i.e. not coming from a successful **malloc**) may cause program crash.

Losing track of the address of an allocated area prevents it to be freed, this causes “memory leak”.

(a more advanced example : see appendix A.10)

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

int main( int argc, char **argv ) {
struct tm * t; char * th;
time_t now;

now = time( NULL ); // same as time(&now)

t = localtime( &now );
if ( argc > 1 )
   t->tm_mday = atoi( argv[1] );

switch( t->tm_mday )
 {
 case 1: case 21: case 31:
                  th = "st"; break;
 case 2: case 22: th = "nd"; break;
 case 3: case 23: th = "rd"; break;
 default: th = "th";
 }
printf("%d%s of %d %d, %02dh%02d\n",
  t->tm_mday, th,
  t->tm_mon+1, t->tm_year+1900,
  t->tm_hour, t->tm_min );
return 0; }
```

4.4 Date and Time

The system keeps the date and time in a single variable of type **time_t** (usually a long int).

The current time is returned by the **time()** function.

Breaking it down into date, hour, mn etc... is done by means of the **localtime()** function, which puts the results in a **tm** structure (from time.h).

```
struct tm {
  int tm_sec;  // 0 to 59
  int tm_min;  // 0 to 59
  int tm_hour; // 0 to 23
  int tm_mday; // 1 to 31
  int tm_mon;  // 0 to 11
  int tm_year; // y - 1900
  int tm_wday; // 0 to 6
};
```

tm_year is current year - 1900.
tm_wday is such a 0 is Sunday, 1 is Monday, etc...

The reverse operation is done by **mktime()**.

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

int main() {
double pi;
// calculating PI
pi = acos(-1.0);
printf("pi = %.18f\n", pi );
pi = 4.0 * atan( 1.0 );
printf("pi = %.18f\n", pi );
// testing some FP exceptions
pi /= 0.0;
printf("bad = %.18f\n", pi );
pi = sqrt( -1.0 );
printf("bad = %.18f\n", pi );

return 0; }

/* prints
pi = 3.14159265358793116
pi = 3.14159265358793116
bad = inf
bad = nan
Note : inf (infinity) and nan (Not A Number) are
special values permitted to a float or double variable.)
*/
```

4.5 Mathematical functions

The math functions are considered part of the standard library, but sometimes packed separately.
The most important operate on double variables only.

round	nearest integer (given as double)
trunc	integral part (round towards zero)
fabs	absolute value
sqrt	square root
exp, exp10	exponential (base ***e***, base 10)
log, log10, log2	logarithm (base ***e***, 10, 2)
pow	x raised to y
sin, cos	unlimited angle
tan	may overflow
asin	-pi/2 <= angle <= pi/2
acos	0 <angle < pi
atan	-pi/2 <= angle <= pi/2

Floating point exceptions like division by 0 and function argument out of range may cause a trap (possibly terminating the process) only if such traps are enabled (which is not the case by default).

```
// misleading if
if   ( a == b )
     if   ( x & 1 )
          a = 0;
else b = 0;
// looks like the else belongs to the first if,
// but it belongs to the second

// another one
if   ( a == b )
     if   ( x & 1 )
          {
          a = 0;
          b = 3;
          };
     else b = 0;
// looks like the else belongs to the second if,
// but it belongs to the first (thanks to a ';')

// sign extension surprise
char c; int i;

c = 129;
i = c;
printf("%X \n", i );
// prints FFFFFF81
```

5. Well known pitfalls

- **if (a = 0)** : changes **a** and is always **false**
- **if** with misleading indentation (see on left side)
- **if** with spurious '**;**' (see on left side)
- converting negative **char** to **int** (see on left side)
- subtracting two unsigned expressions and test whether the result is <0
- comparing signed with unsigned expressions
- function returning a pointer to local (auto) data (invalid pointer)
- declaring variables and pointers on the same line :
  `int * p, p2;` // what is p2 ? (an int)
- using **scanf** with arguments which are not pointers
- using **scanf** with "**%f**" and a pointer to **double** (should be "**%lf**")

6. Conclusion

6.1 What is still missing (left for an advanced training)

- the const and volatile qualifiers
- the structures initialization
- the bit fields
- the wide char strings
- the functions with a variable number of arguments
- the “long jump”
- the process management (signals, threads)

6.2 Bibliography

- The C language, second version
  Brian W. Kernighan & Dennis Ritchie,
  Prentice Hall,
  Covers ANSI C (C89), unlike the first edition

- The C Book, second edition,
  Mike Banahan, Declan Brady and Mark Doran,
  available on-line (http://publications.gbdirect.co.uk/c_book/)

- C programming Wikibook, variable edition,
  anonymous authors,
  available on line (http://www.wikipedia.org)

- ISO Committee draft ISO/IEC 9899:TC2
  the C99 specification

- C Programming
  Steve Holmes, University of Strathclyde, Glasgow,
  available on line (http://www2.its.strath.ac.uk/courses/c/index.html)

- The GNU C Library Reference Manual,
  The Free Software Foundation (http://www.gnu.org)

```
/* Fibonacci sequence is defined as :
F(0) = 0;
F(1) = 1;
F(N) = F(N-2) + F(N-1) for N>1
*/

// this function computes a Fibonacci
// number by recursively calling itself
int F( int index )
{
int new;
if ( index < 2 )
   return( index );
new = F(index-2) + F(index-1);
return( new );
}

// note : this example is not the
// most efficient way to compute those
// numbers, merely an algorithmic curiosity

int main()
{
printf("%d\n", F(12) );
return 0;
}
```

A.1 Re-entrance and recursion

When the execution flow calls a function while this function is already being executed, it is said that the function is re-entered.

This may happen in case of :

- **recursion** : the function calls itself directly or through another function
- hardware **interrupt**
- **multi-thread** application

Static variables (global or local) are shared by all the instances of the function.
Automatic variables are unique for each instance, thanks to the fact that a new stack frame is created each time the function is entered.

A function is said "**reentrant"** or **thread-safe** if it can be re-entered without loss of functionality.
Using only automatic variables is the simplest way to guarantee "reentrancy".

Recursive programming is very useful for handling hierarchical data (trees) like file directories or arithmetic expressions.

```
int main() {
// a bidimensional array
int mat[3][3] = { {7,8,9},
                  {4,5,6},
                  {1,2,3}
                };
int * iptr; int row, col;

// dereferencing as an array
row = 1; col = 2;
printf("%d,", mat[row][col] );
// dereferencing as a pointer
printf("%d,", *(mat[row]+col) );
// another pointer
iptr = (int *)mat;
// the width is 3...
iptr += row * 3 + col;
printf("%d\n", *(iptr) );

// prints : 6,6,6

return 0;
}
```

A.2 Rectangular arrays

More specialized are rectangular arrays, which are made out of equal width rows.

The elements are stored in contiguous memory, row by row, in an underlying one-dimension array.
The position of a given element in this array can be derived knowing the width :
pos = (row * width) + column;
This is done automatically by the compiler when the my_array[row][column] syntax is used.
So no intermediate pointer is stored, only the payload data.

However, my_array[row] returns a pointer to the first element of a row, like if we had an array of arrays (which is kind of emulated).

But my_array itself cannot be considered as a pointer to pointers.

By cascading more subscript operators, a multidimensional array can be created.

```
// a polymorphic object..
typedef union {
// unnamed structure tags are created here
struct { int typ; char * data; } a;
struct { int typ; double data; } b;
} Poly;

void printu( Poly * u )
{
switch( u->a.typ )
  {
  case 0: printf("%s\n", u->a.data );
          break;
  case 1: printf("%f\n", u->b.data );
          break;
  }
}

int main() {
Poly p;
p.a.typ = 0;  p.a.data = "Hi";
printu( &p );
p.a.typ = 1;  p.b.data = 3.14;
printu( &p );

return 0; }
```

A.3 Unions

A union is an aggregate type containing overlapped members, laid out at the same address.

The union size is the size of its greatest member.

Union members may be structures, with usually some common initial sequence.

A “polymorphic” object can be created this way.

```
typedef enum { TEXT, REAL } PType;

typedef union {
struct { PType t; char * d; } a;
struct { PType t; double d; } b;
} Poly;

void printu( Poly * u )
{
switch( u->a.t )
  {
  case TEXT :
     printf("%s\n", u->a.d ); break;
  case REAL :
     printf("%f\n", u->b.d ); break;
  }
}

int main() {
Poly p;
p.a.t = TEXT;  p.a.d = "Cheers";
printu( &p );
p.a.t = REAL;  p.b.d = 3.14159;
printu( &p );
return 0; }
```

A.4 Enumerations

An enumerated type is a user-defined type which can take values from a list of user-defined symbols.

Using the **enum** keyword, one can define such a list of symbols which are internally bound to contiguous integer constants.

This helps making code more readable.

Unfortunately, C does not handle an **enum** as a distinct type and no checking is done on the use of the enum symbols.

It is worth noting that compiling the same program with the C++ compiler enforces a strict checking of the enums.

Notice a common usage of giving constants upper-case names.

```
// a function expecting a pointer to a void function
// taking an int argument
void execN( void (* fu)(int), int N )
{
while( N-- )
   fu( N );
}

// a function
void myfun( int i )
{ printf("%d\n", i ); }


int main() {
// a pointer to a void function
// taking unspecified arguments
void (* fp)();
fp = myfun;
// dereferencing the pointer (call myfun)
fp(55);

// passing a pointer to function
execN( myfun, 5 );

return 0; }
```

A.5 Pointers to functions

Pointers to function are useful for installing a call-back function by passing a pointer to it to some initialization function.

Pointers to functions may be members of structures, in order to associate “methods” to “objects”.

The general form for declaring a pointer to function is :

```
return_type (* my_ptr)();
```

A pair of parentheses are necessary here to bind the * (star) to the pointer name, not to the return type.

To assign a value to the pointer, just assign to it the address of a regular function, which is returned by its bare name (without ‘()’).
(Notice that the & operator does not apply here)

To dereference the pointer, which means calling the function it points to, just use the pointer name as if it were a function name :

```
my_ptr( arg1, arg2 );
```

```
// generic processing (works with many types)
#define min1(a,b) (a<b)?a:b

int main() {
int x, y, c;

c = min1(3,8);                  // OK

c = min1(3,8) + 1;              // WRONG

c = min1( 7, min1(8,3) );       // WRONG
// expands to c = (7<(8<3)?8:3)?7:(8<3)?8:3;

// improved macro
#define min2(a,b) ((a)<(b)?(a):(b))

c = min2(3,8) + 1;              // OK
c = min2( 7, min1(8,3) );       // OK
c = min1( 7, min2(8,3) );       // OK

// but not perfect...
x = 2; y = 7;
c = min2( x++, y );  // twice wrong, x and c

return 0; }
```

A.6 Macro issues

In the case of a macro with parameters, it is recommended to enclose in parentheses :

- each parameter
- the complete body of the macro

Failing to do so results in unexpected behaviour in presence of expression elements surrounding the macro invocation or contained in its arguments.

Even with this precaution, there may be situations where using the macro like if it were a C function gives incorrect results (see example).

```
# defining variables
CCOPT = -Wall -O2
C_SRC = Myfun.c Myjob.c
CPP_SRC =
EXE = Myjob

# generating the objects list
OBJS = $(C_SRC:.c=.o) \
       $(CPP_SRC:.cpp=.o)

# the first one is the default target
$(EXE) : $(OBJS)
gcc -o $(EXE) $(OBJS)

.c.o :
gcc $(CCOPT) -c $<

.cpp.o :
g++ $(CCOPT) -c $<

# dependencies
Myfun.c : Myfun.h
Myjob.c : Myfun.h
# the compiler may help finding the dependencies :
# gcc -MM Myjob.c
```

A.7 Advanced makefile

In order to simplify the maintenance of makefiles, make may provide a higher degree of automation.

For example a list of files may be generated from another one by suffix substitution :

```
OBJ =  $(C_SRC:.c=.o)
```

Multiple similar rules may be packed in one using “pattern rules” :

```
.c.o:     # classic style

%.o : %.c # modern style
```

means take each **.o** file as a target and the matching **.c** file as a prerequisite (or does nothing if the matching **.c** file does not exist).

In this case, automatic variables are generated :

- $@ for the target
- $< for the prerequisite

```
#include <stdio.h>
#include <stdlib.h>
#define sz sizeof(data)
int main( int argc, char **argv ) {
FILE * my_in; FILE * my_out;
double data;
if ( argc < 3 ) exit(1);
my_in = fopen( argv[1], "rb" );
if ( my_in == NULL ) exit(1);
my_out = fopen( argv[2], "wb" );
if ( my_out == NULL ) exit(1);

// This program reads a floating point number in binary
// and writes the same divided by 3.0
if ( fread( &data, sz, 1, my_in )
     != 1 )
   { printf("read failed\n");
     exit(1); }
printf("read %f\n", data );

data /= 3.0;
if ( fwrite( &data, sz, 1, my_out )
     != 1 )
   { printf("write failed\n");
     exit(1); }
return 0; }
```

A.8 Binary files

Binary files may contain arbitrary combinations of bits, which happens in compiled object and executable files (machine code), compressed files, multimedia files and CAD data files.

In Unix-like systems (like Linux), the standard library makes no difference between text files and binary files.

But on some non-Unix systems (like Windows), a sequence of two characters "CR-LF" is used for line termination. To preserve the operation of the C language line terminator '\n', the library performs the conversion automatically when writing and reading.

This is convenient for line-formatted text, but may corrupt any binary file, and cause erroneous positions when seeking .
To disable the conversion, append the '**b**' modifier in the second argument of fopen().

```
// checking the feasibility of a large file (> 4Gb)

int main() {
long long pos; FILE * fbig;
unsigned char buf[512];

if ( sizeof(fpos_t) < 5 )
   { printf("sorry, no big file\n");
     exit(1); }

fbig = fopen( "fatfile", "w" );
if ( fbig == NULL )
   exit(1);
// suffix LL is for a long long constant
pos = 0x110000000LL;

if ( fsetpos( fbig, &pos ) )
   { printf("fsetpos failed\n");
     exit(1); }

if ( fwrite( buf, 1, 512, fbig )
     != 512 )
   { printf("write failed\n");
     exit(1); }

fclose( fbig ); return 0; }
```

A.9 Big file access

Until recently, on 32-bit machines, the current position was stored as a long int, which limited the file size to about 2 GB.

To support larger files, the type **fpos_t** used by **fgetpos()** and **fsetpos()** may be larger than 32 bits (see stdio.h).

```c
#include <stdio.h>
#include <stdlib.h>
#define QNT 4096

typedef struct {     // a smart buffer
unsigned char * d;
int cnt; int capacity;
} MyBuf;

void addbyte( unsigned char x,
              MyBuf * b )
{
if ( b->cnt == b->capacity ) // full
   {
   if   ( b->capacity == 0 )
        b->d = malloc( QNT );
   else b->d = realloc( b->d, QNT);
   if ( b->d == NULL )
      { fprintf(stderr,"no mem\n");
        exit(1); }
   b->capacity += QNT;
   printf("%d\n", b->capacity );
   }
b->d[b->cnt++] = x;
}
// cnt and capacity must be initialized with zero
```

A.10 Smart buffer

In the example, the addbyte() function allows to append one byte in an array, the capacity of which is increased automatically to suit the needs.

Memory is allocated by chunks of an arbitrary size QNT, great enough to avoid too frequent calls to realloc().

A similar scheme is used in built-in smart arrays existing in higher level languages (C++ : vector, Perl : array).

A.11 The ASCII code

byte value (decimal)	byte value (hexa)	contents
0 to 31	0 to 1F	control characters (not printable)
32 to 47	20 to 2F	space ! " # $ % & ' ( ) * + , - . /
48 to 57	30 to 39	0 1 2 3 4 5 6 7 8 9
58 to 64	3A to 40	: ; < = > ? @
65 to 90	41 to 5A	ABCDEFGHIJKLMNOPQRSTUVWXYZ
91 to 96	5B to 60	[\ ] ^ _ ‘
97 to 122	61 to 7A	abcdefghijklmnopqrstuvwxyz
123 to 126	7B to 7E	{ | } ~
127	7F	control character DEL (non printable)
128 to 255	80 to FF	extension : letters with diacritics (example : ISO 8859-1 “latin 1” alphabet) or multi-byte code prefix (example : UTF8), or semigraphic glyphs

Control characters, detail :

decimal	hexa	contents
1 to 26	01 to 1A	CTRL-A to CTRL-Z
4	04	CTRL-D = EOT [end of text]
7	07	CTRL-G = BELL (\a)
8	08	CTRL-H = BS [backspace] (\b)
9	09	CTRL-I = TAB [tabulation] tab (\t)
10	0A	CTRL-J = LF [line feed] (\n)
12	0C	CTRL-L = FF [form feed]
13	0D	CTRL-M = CR [carriage return] (\r)
23	17	CTRL-W = [word erase]
27	1B	ESC [escape]

C Language Initial - Table of contents

(2nd edition)