C in a nutshell for OGO 2.1

The C Programming Language
Brian W. Kernighan
Dennis M. Ritchie

August 2001

Jos Schoenmakers
HG 7.81
tel 4568
email G.J.Schoenmakers@tue.nl

an important aspect of
our consciousness is that we can:

- *anticipate*
- *design a scenario*
- *make a plan*
- *write a program*

a first impression

```
/* the most known C program */


#include <stdio.h>


void main()
{
 printf("hello, world");
}
```

some built-in types

int	2 bytes in Borland C++
float	4 bytes
char	character, a single byte
short	short integer
int	integer
long	long integer
float	floating point
double	double precision float

int and float

int   can be used for ranking
float can be used for physical quantities

```
void main()
{
 float fahr, celsius;

 int lower, upper, step;

 lower = 0;
 upper = 300;
 step = 20;
 fahr = lower;
 while (fahr <= upper)
   {
    celsius =(5.0/9.0)*(fahr-32.0);
    printf("%f%f", fahr,celsius);
    fahr = fahr + step;
   }
}
```

text stream

a text stream is a sequence of characters

getchar()

reads the next character from a text stream
and returns that as its value.
if there is no input, getchar returns EOF

putchar(c)

writes the value of the integer variable c
as a character.

we can interpret a stream as a number of lines:
each line consists of 0 or more characters followed by a new line

character.

```
.....................\n
.................\n
```

an assignment is an expression

```
while (next character not equals EOF)
   print it
```

```
void main()
{
 int c;

 while ((c = getchar()) != EOF)
    putchar(c);
}
```

tourniquet is stream driven.

count whites:

```
void main()
{
 int c, nh;

 nh = 0;

 while ((c = getchar()) != EOF)

    if (c == ' ' || c == '\n'
                 || c == '\t') ++nh;

 printf("%d", nh);
}
```

count digits

we use an array ndigit[0...9]

```
void main()
{
 int c, i;

 int ndigit[10];

 for (i = 0; i < 10; ++i)
    ndigit[i] = 0;

 while ((c = getchar()) != EOF)
    if (c>='0' && c<='9')
       ++ndigit[c-'0'];

 for (i = 0; i < 10; ++i)
    printf("%d", ndigit[i]);
}
```

declaration

the function must be declared before it is applied

```
long power(int, int) ;
```

application

```
void main()
{
 int n;

 for (n = 1; n <= 20; ++n)
    printf("\n%ld %ld %ld %ld",
            power(n,1),
            power(n,2),
            power(n,3),
            power(n,4));
}
```

function definitions can appear in any order

a function cannot contain other functions:
the nesting is only one deep

definition

we use 2 local variables

the essence of the repetition is: p = $base^i$

```
long power(int base, int n)
{
 int i; long p;

 p = 1;
 for (i = 0; i < n; ++i)
    p = p * base;

 return p;
}
```

expression-statement: *expression* ;

this statement just empties the stack

```
p = q;
a = b = c = 20;
++i;
printf("start\n");
```

null-statement: ;

compound-statement: { *declarations*$_{opt}$ *statements*$_{opt}$ }

```
if (n>0) {
          int i;
          for (i=0; i<n; ++i) ...
         }
```

There is no semicolon after the right brace that ends a compound

catweazle again

because cases serve just as labels,
after the code for one case is done,
execution falls through to the next
unless you take explicit action

```
switch (e)
{
 case '+': plus();   break;
 case '*': mult();   break;
 case 'i': insert(); break;
 case 'p': print();  break;
 case ';': clear();  break;

 case 'x': return;

 default : print();  break;
}
```

the programmer needs discipline,
not the programming language

goto

labeled-statement: *identifier* : *statement*
case *const-expr* : *statement*
default : *statement*

goto-statement: goto *identifier* ;

stack

the stack is simple, not robust

```
int p = 0, stack[100];

void push(int i)
{
 stack[p] = i; if (p < 99) ++p;
}

int pop(void)
{
 if (p > 0)  --p; return stack[p];
}

void print(void)
{
 int i;
 for (i = p-1; i >= 0; --i)
    printf("%d ", stack[i]);
}
```

interface stack.h

the interface is defined in the headerfile

```
#ifndef STACK
#define STACK

void push(int);
int pop();

#endif
```

application appl.cpp

the interface can be included in the application
and than compiled separately

the interface can be included in the implementation
and than compiled separately

```
#include "stack.h"

void main()
{
 push(15);
 i = pop();
}
```

implementation stack.cpp

static variables belongs to the text of the program
and are not created on the execution stack
statics are not known to the linker

the value of a local static still holds when the function is re-entered

a global static is invisible for other files

```
#include "stack.h"

static int val[100], p = 0;

void push(int i)
{
 val[p++] = i;
}
int pop()
{
 return val[--p];
}
```

operators

we split the concept variable in
2 elementary concepts

v the **object** referred by v

&v the **reference** itself : quote

***p** dereference:
the object where p points to

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

i = 2;          p = &i;
j = i;          q = p;

i = 0;          p = 0;
                p = NULL;

                *q = 5;
```

parameter passing

arguments are passed by value
so the parameters are local variables

```
void swap(int x, int y)
{
 int h; h = x; x = y; y = h;
}

swap(a, b);              /* wrong */
```

references are needed to change the outer world:

```
void swap(int *x, int *y)
{
 int h; h = *x; *x = *y; *y = h;
}

swap(&a, &b);            /* correct */
```

arrays

```
int i, a[10], *p;
```

a is an array of 10 consecutive int's
with subscripts 0 9

a is a reference

`a[i]` equals `*(a+i)`

`a` equals `&a[0]`

```
p = &a[0] ;          p points to a[0]
p = a ;                 "
i = a[0] ;           i equals a[0]
i = *p ;                "
i = *a ;                "
i = a[3] ;           i equals a[3]
i = *(p + 3) ;          "
i = *(a + 3) ;          "
```

scanf

scanf is the opposite of printf

```
int i;
char s[100];

getint(&i);

scanf("%d", &i);          whites digits white

scanf("%s", s);           whites chars white
```

a first example

the members of a structure
can be of different types

```
struct article {
                 int number;
                 char name[20];
                 float weight, length;
                };
```

now the the type article is defined,
and objects can be created:

```
struct article a1, a2;
```

operations

legal operations on structures are:
• assignment
• pass arguments to a function (by value)
• return from a function (by value)
• taking the reference &
• member selection .

construction

we have to define a constructor ourselves

```
typedef struct point Point;

Point cons(int x0, int y0)
{
 Point h;
 h.x = x0; h.y = y0;
 return h;
}
```

pointers to structures

```
Point *pp;

pp = &p ;
i  = (*pp).x ;    /* correct */

i  = *pp.x;       /* wrong   */
```

there is a special operator for pointer selection:

pp->x equals (*pp).x

precedence of . and -> is high
precedence rules:

```
()  []  .  ->
!  ++  --  +  -  *  &
```

```
++pp->x;      increment x
(++pp)->x;    increment pp before access x
(pp++)->x;    increment pp after access x
```

file access

FILE	typedef of a handle that contains information about the file
file pointer	points to a FILE

```
FILE *fopen(String name, String mode)

   "r"  read
   "w"  write
   "a"  append

int fgetc (FILE *f)
int fputc (int c, FILE *f)

int fscanf
    (FILE *f, String format, ...)
int fprintf
    (FILE *f, String format, ...)


int fclose(FILE *f)
```

```
long count(FILE *f)
{
 long nc = 0;

 while (fgetc(f) != EOF) ++nc;

 return nc;
}
```

file name is String

```
void main()
{
 long nc;
 String s = "a:\\intro2\\wc.c";

 FILE *fp;
 fp = fopen(s, "r");
 nc = count(fp);
 fclose(fp);

 printf("%ld", nc);
}
```