INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION

INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY
c/o INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS 34100 TRIESTE (ITALY) VIA GRIGNANO, 9 (ADRIATICO PALACE) P.O. BOX 586 TELEPHONE 040-224572 TELEFAX 040-224575 TELEX 460449 APH I

SMR/643 - 20

SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992

ADVANCED C
(Part II)

A. NOBILE
International Centre for Theoretical Physics
Strada Costiera 11
34100 Trieste
Italy

THE C PREPROCESSOR

Already met:

```
#include
#define
#ifdef
#ifndef
#endif
```

ANSI greatly expanded it.
Here only elementary usage

Takes code containing preprocessors directives
Transforms it into legal C without them

Works line by line (C does not care newlines)
Does not obey scope rules:
definition holds from definition point to
end of file

--> USE SPARINGLY

Introduces C-specific things

--- > NOT A GENERAL-PURPOSE MACRO SYSTEM

#define

2 versions : function-like and not

```
#define EOF     (-1)
```

.....

```
    if ( c ==  EOF)
```

is translated into

```
    if ( c == -1)
```

```
#define FMAC(a,b) a * b /*poor, see later*/
```

```
    FMAC( p->data, q[4])
```

is translated into

```
    p->data * q[4]
```

```
#define max(x,y) ((x)>(y)?(x):(y))
```

```
#define UPPER(c) ((c)-'a'+'A')  /*ASCII only */
```

RESCANNING

```
#define EOF (-1)
#define readc(c)
    ((c=getchar())!=EOF)
```

PRACTICAL RULE :
macro names (not function macros)
are all uppercase, C identifiers are lower case or mixed case


```
#define A a,b
#define strange(x) x-1
strange(A)
```

strange should be replaced
it has one argument , **A**
A should be replaced
A becomes **a,b**
strange now has 2 arguments?
ANSI says **no**; try yours
and what if

```
#define strange(A) something
```

Etc. : DON'T TRY

WARNING

```
#define PA (a)
```

This one defines PA to be (a), not PA(a) being nothing. SPACE BETWEEN NAME OF MACRO AND (

WARNING

```
#define FILENAME myprog.c
printf ("compiled from FILENAME\n");
```

Does not work : strings are a single object to preprocessor

```
#define FILENAME "myprog.c"
printf ("compiled from %s\n" , FILENAME);
```

ANSI has defined specific operators for this kind of situation.

WARNING

```
FMAC ( p+q, l+m)
```
becomes
```
p + q * l + m
```
probably wrong.

```
#define DOUBLE(x)   x+x

3* DOUBLE(x)
```
becomes
```
3* x + x
```
wrong

Correct format
```
#define DOUBLE(x) ( (x) + (x) )
#define FMAC(a,b) ( (a) * (b) )
```

WHY TO USE FUNCTION MACROS ?
- increase readability
- faster to evaluate than real functions

#undef ***identifier***

Causes the definition to be forgotten

Ex.
```
#include <stdio.h>
#undef BUFSIZE
#define BUFSIZE 1024
```

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

#define NAME "file"
#include NAME
```

Includes can be nested

Conditional compilation

```
#ifdef identifier
#ifndef identifier
#if constant expression
#else
#endif
```

- To select pieces of code that are machine dependent
- To turn on-off parts of code used for debugging

```
#define M6809
......
#ifdef M6809
typedef long int Int;
#endif
#ifdef M68020
typedef int Int;
#endif
```

or (UNIX 1983 Source)

```
typedef struct {
#if vax || u3b
    int  _cnt;
    unsigned char * _ptr;
#else
    unsigned char * _ptr;
    int  _cnt;
#endif
    unsigned char *_base;
    char _flag;
    char _file;
} FILE
```

- **vax || u3b** is a constant expression. If defined and not 0, expression not 0, etc.

WARNING

```
#define NULL 0
#if NULL
```

would fail.

C LIBRARIES

DEFINED BY ANSI STANDARD

NOT REQUIRED:
- required in "hosted" systems
- can be missing in standalone systems ("bare" C)

STANDARD HEADERS: contain macro names and types used by standard libraries

WARNING:
- identifiers defined in standard headers are reserved . Should not be redefined or reused (like all the C keywords)
- names starting with _ are reserved

<assert.h>	<math.h>	<stdio.h>
<ctype.h>	<setjmp.h>	<stdlib.h>
<float.h>	<signal.h>	<string.h>
<limits.h>	<stdarg.h>	<time.h>
<locale.h>	<stddef.h>	

Sections of the library:

I-O	**<stdio.h>**
String handl.	**<string.h>**
Debugging:	**<assert.h>**
Character handl.:	**<ctype.h>**
Time, date:	**<time.h>**
General utilities	**<stdlib.h>**
Implementation	**<limits.h>**
	<float.h>
	<locale.h>
Exceptions	**<signal.h>**
	<setjmp.h>
Var. num. of arg.	**<stdarg.h>**
Math.	**<math.h>**

ASSERTION CHECKING

```
#include <assert.h>
......
    assert ( a>b );
.....
```

if **a>b** nothing;
else
    print text of expression (**a>b** in this case)
        , file name, line number
    abort
If NDEBUG defined, expression not evaluated and test performed

```
#define NDEBUG
#include <assert.h>
```

all **assert** turned off

EXCEPTION HANDLING AND NON-LOCAL TRANSFER

EXCEPTION: occurs unexpectedly or infrequently (error conditions)
can be originated outside program control

- hardware exception: division by 0
- user exception : interrupt key

In C : signals
-can be generated by the hardware or by the software
-cause the execution to be transferred to a *signal handler*
-programs can establish their own signal handlers
-a default signal handler (implementation dependent) is always available
-program can send a signal to themselves

NOTE : sending signal to another program is an O.S. problem

ANSI defines a minimum of 8 signals: more are implementation dependent.

They are **int**, defined in **<signal.h>**

SIGABRT	calling **abort** library function
SIGFPE	illegal floating point operation
SIGILL	illegal instruction
SIGINT	interrupt (from keyboard?)
SIGSEGV	illegal memory reference
SIGTERM	software termination (sent by another program?)

Default signal handler, called SIG_DFL, defined in **<signal.h>**, typically aborts the program

Alternative signal handler, called SIG_IGN, ignores the signal.

User defined signal handler:

```
int handler(sig_number)
int sig_number;{
    ....
}
```

ANSI::

```
void handler(int sig_number){
.....
}
```


NOTE : SIG_DFL , SIG_IGN are actually pointers to a hadler, therefore have type

```
int  (*SIG_DFL)();
```

Pointer to function returning int,

or, in ANSI

```
void  (*SIG_DFL)(int)
```

Handlers gets associated to signals by calling signal

```
#include <signal.h>
int fpe_handler();
main()
{
    ...
    signal(SIGFPE, fpe_handler);
}
```

`signal` returns a pointer to the old handler

```
#include <signal.h>
..
int int_handler();
{
    int (*old)() ;/* place to keep
                  old to old handler*/
    ...
    /* install handler only if
       signal not ignored */
    if (( old=signal(SIGINT, SIG_IGN)) !=
       SIG_IGN )
       signal(SIGINT,int_handler);
    ...
}
int_handler(sig_num)
int signum;
{
...
}
```

Later, **old** can be used to reinstall the original handler.

COMMENT

definition of signal is

```
typedef int (*func)() HANDLER;
HANDLER signal (sig, handler)
int sig;
HANDLER handler;
```

Try to write it without `typedef`!

HANDLER STRUCTURE

- First, call `signal` again (usually, signals go back to default handler every time raised)
- Do whatever needed
- Either `return` (execution continues from point of exception) or jump somewhere else with `longjmp`

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#define PR putchar(':')
int float_err(), keyboard_intr();
/*handlers*/

float a,b, result; char opr; /*global
->visible from handlers*/
main(void)
{
    signal(SIGFPE, float_err);
    signal(SIGINT, keyboard_intr);

    while(PR,scanf("%f%c%f",&a,&opr,&b)
!= EOF)
    {
       switch(opr){.....
          /* calculator as in lect. 1
*/
          ....
       }
    }
}
```

```
closing( message, exit_code)
char *message;
int exit_code;
{.....
}
float_err(i)
int i;
{
    signal (SIGFPE, float_err);
    printf(" Floating point error\n");
    a=b=1.0;/*safe value*/
}
keyboard_intr(i)
int i;
{
    signal(SIGINT, keyboard_intr);
    printf("Do you want to quit? Y or
N:");
    switch (getchar()) {
        case 'Y' : case 'y' :
            closing("Regular end", 0);
        default: printf("continue\n");
    }
}
```

- handlers can exit, or return; return resumes execution from exception point

- problem : scanf returns EOF if interrupted.
  resuming execution within **scanf** is not continuing.
  -- > continue from a safer location

---

longjmp, setjmp

```
#include <setjmp.h>
/* defines type jmp_buf */
jmp_buf env;

int setjmp(jmp_buf env);
```

- stores in **env** all the information to resume execution from the point it is called

WARNING : not a checkpoint!

- returns zero

```
longjmp(env,val)
jmp_buf env;
int val;
```

- if **env** filled by **setjmp**, jumps to the return point of setjmp, but returning **val; (if val == 0,** returns 1.)

Example:

```
#include <setjmp.h>
jmp_buf env; /* global !*/
main(){
    int v;
    ......
    if ( v=setjmp(env) )
        printf(" Coming from longjmp "
            " value = %d", v);
......
}
other_function(){
...
    longjmp(env, 1 );
....
}
```

FINAL EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setbuf.h>
#define PR putchar(':')
int float_err(), keyboard_intr();
/*handlers*/

float a, b, result; char opr; jmp_buf
env;
 /*global to be visible from
handlers*/
main(void)
{
    signal(SIGFPE, float_err);
    (void)signal(SIGINT,keyboard_intr);
    (void)setjmp(env);

    while(PR,scanf("%f%c%f",&a,&opr,&b)!=
    EOF)
    {
        switch(opr){
            ......
        }
    }
}
closing( message, exit_code)
char *message;
int exit_code;{
.....
}
```



```
float_err(i)
int i;{
    signal (SIGFPE, float_err);
    printf(" Floating point error\n");
    a=b=1.0;/*safe value*/
}
keyboard_intr(i)
int i;{
    signal(SIGINT, keyboard_intr);
    printf("Do you want to quit? Y or N:");
    switch (getchar()) {
        case 'Y' : case 'y' :
            closing("Regular end", 0);
        default: printf("continue\n");
        longjmp(env,1);
    }
}
```