

Chapter 1

The C Programming Language (Advanced Features)

MACRO Definitions

#define *identifier* *token-sequence*

#define *identifier* (*parameter-list*) *token-sequence*

- No space is allowed between the identifier and the parentheses.
- Every appearance of *<identifier>* in the code is replaced by the *<token-sequence>* prior to compilation. If *arguments* are involved, they replace the *parameters*.
- **Common Practice:**
MACRO identifiers are all CAPITAL letters
- **A Useful Rule:** Constants present in software, with the possible exception of 0, 1, will backfire. USE MACROS.



11

MACRO examples

```
#define MAX_STR_LEN 20
#define IS_UPPER(c) ((c) >= 'A' && (c) <= 'Z')
#define TO_LOWER(c) (IS_UPPER(c)? (c) - 'A' + 'a' : (c))
    . . .

char arr[MAX_STR_LEN + 1], *str;
    . . .
for (str = arr; *str != '\0'; str++) {
    *str = TO_LOWER(*str);
}
    . . .
```

12

MACRO PITFALLS

Several problems may arise while calling the MACRO

```
#define SQR(x) x * x
```

- **Operator Precedence ERROR**

- External

```
(int)SQR(a)    is expanded to
(int)a * a     which casts the left operand only
```

solution: Put parentheses (or braces) around MACRO.

- Internal

```
SQR(a + b)    is expanded to
a + b * a + b which is definitely
not (a + b)*(a + b)
```

solution: Put parentheses around MACRO arguments.

```
#define SQR(x) ((x) * (x)) /* is SAFE */
```



13

MACRO PITFALLS (Cont.)

- Side Effects ERROR

`SQR(++i)` is expanded to
`((++i) * (++i))` which may increments `i` twice
 (and the result is *undefined*)

- Unnecessary Function Calls (actually, this is *side-effect* too)

`SQR(very_difficult_func(a,z,t))`
 will evaluate the function twice, before multiplication.

NO General Solutions for the above two problems:

Be **wise** while defining a MACRO
 and **cautious** while calling a MACRO

14

Calling a Function vs. Calling a MACRO

- | | |
|---|--|
| * Is always an expression | * May be a statement (e.g. one that requires <i>automatic</i> variables). |
| * Will not change arguments and side-effects are fully controlled | * May have unexpected side-effects |
| * Can always return a newly created object | * May require an argument to carry a newly created object |
| * Limited to fixed type arguments | * May operate unchangeably on arguments of varying types |
| * Saves executable code
Easier maintenance | * Code is duplicated though maintenance is easy |
| * May be passed as an argument to functions | * Cannot be passed as an argument |
| * Function call overhead (for <i>stack</i> handling) | * No calling overhead |



15

When is a **MACRO** better than a **Function** ?

Rules of Thumb:

- Operation required is **short, simple**, and (maybe) used in **different locations**.
- Operation required is **short, simple**, and is used **intensively**.
- Operation required is performed on variety of **different types**.

Examples for last case:

```
#define MAX(a, b)      ((a) > (b) ? (a) : (b))
#define SWAP(type, a, b) { type t = a; a = b; b = t;}
```

16

C99: Inline functions

- The new C standard (**ANSI+ISO+IEC+9899-1999**) allows **inline** functions (this feature was taken from C++)
- A function is **inlined** when its code is expanded at the **point of the call to the function** instead of creating a **function-call**.
- In that respect, inlining a function has the same effect as using a **MACRO** except one major difference
Inlining is a job done by the compiler
- Consequently, it behaves like a function with all other respects, **except code duplication** and **calling overhead**



17

Inline Functions (cont.)

```
inline int max(int a, int b)
{return (a) > (b) ? (a) : (b)}
```

- The function definition should be in a header (if it is used across several files)
- The function is restricted to a single set of types (as denoted in its definition)
 - Arguments of other types will be converted (if permissible)
 - If you need it for different types (without conversion), use **MACRO**
- Since the function must be defined before use it subverts top-down presentation
- The compiler may refuse to inline a function (either completely or in selected cases)

18

Enumerable Types

- **Types** that consist of certain **integral values**, which are carried by **symbolic names**.
The **names** are **more important** than the actual values.
- **Enum definitions**

```
enum bool { FALSE, TRUE };
enum month { JAN = 1, FEB, /* ... */ , DEC };
enum colors { WHITE = 1, BLACK, GREEN = 8, RED };
```
- **Using enum types**

```
enum bool b[SIZE], t = FALSE;
```
- **enum vs. #define** (enum is superior)
 - It obeys C's **scoping rules**.
 - The compiler **may** check for type mismatch.
 - The debugger **may** recognize the symbolic names.



19

MACRO and Enum – a Comparison

```
enum day {SUN = 1, MON, TUE, WED, THR, FRI, SAT};
void f(void)
{ enum day d = TUE, /* o.k. */
  e = 3; /* syntax error */
  enum bool {FALSE, TRUE};
  enum bool b = FALSE, /* o.k. */
  c = 1; /* syntax error */
  #define RED 3;
  int color = 3; /* o.k. */
}
void g(void)
{ enum day d = TUE; /* o.k. */
  enum bool b = FALSE; /* syntax error */
  int color = RED; /* o.k. */
}
```

20

const type qualifier

```
const int fixed = expression ; /* No changes after initialization */
```

- When is **const** preferred over **enum** or a **MACRO** ?
 - Its value is decided at run time (and is of **any** type).
 - It obeys C's scoping rules.
 - It is used where its address (& operator) is required.
 - It must be recognized by the compiler/debugger.
 - In *trying* to force a function not to modify an array argument, or any argument that is passed by its address.

Example: If a function is *defined* as

```
int scalar_product(const int vec1[], const int vec2[]);
```

the compiler *may* check that no assignment of the form

```
vec1[i] = exp is evaluated in the function body.
```



21

typedef Declarations

C provides a facility for creating new data type names.

```
typedef int Length;      /* Defining */
Length l, lvec[SIZE];   /* Using */
    will make l of type int, and lvec of type array of int.
typedef enum bool bool; /* Shortening */
```

- **Typedefs** are far from being **MACROs**.

```
typedef char Buf[BUF_SIZ];
Buf buffer, buf_array[SIZE];
    will make buffer - an array (of size BUF_SIZ),
    and buf_array - an array (of size SIZE)
                                of arrays (of size BUF_SIZ),
    i.e, equivalent to char buf_array[SIZE][BUF_SIZ];
```

22

Why typedef ?

- **Easy modification of data types**

Example: Certain **int** variables are used for carrying flags. Later, the software became more complicated, and we want to change these variables into type **long**. Had these variables been declared being of type **Flag** (with "**typedef int Flag;**"), all can be done by modifying the **typedef** statement.

- **Meaningful names for data types**

In the example above, wherever we see the declaration "Flag var;" we understand that 'var' is going to be used as a "flag carrier".



23

Casting

Is a way to force an expression to be evaluated to a certain type

Example:

```
int    i = 6;
double d = 2.9;
```

- The following three expressions are evaluated to three different values :

```
i/d ( == 2.0689)  (int)(i/d) ( == 2)  i/(int)d ( == 3)
```

- Here we force an argument of a function to be of the correct type:

```
d = sqrt((double)i);  /* a documentation benefit too */
```

24

Casting (Cont.)

- There are cases where we have to declare pointers without prior knowledge about the type they will point to.
- The type `void *` (i.e. a pointer to void) is used as a **generic pointer type**. In a mixed type **pointer expression**, conversion is automatic.
 - However, **casting** is necessary when pointers are dereferenced.

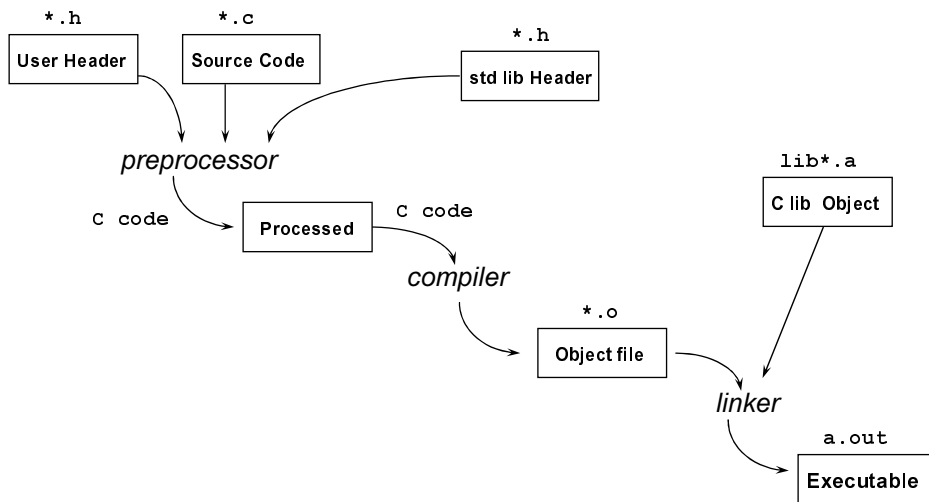
Example:

```
int    i;
double d, e;
void   *gpt0 = &i, *gpt1 = &d;      /* Causes no problem */
e = *gpt0 + *gpt1;                  /* Is impossible */
e = *(int *)gpt0 + *(double *)gpt1; /* Is the solution */
```



25

From Source to Executable



26

The C Preprocessor

- **MACRO definitions**

```
#define a macro definition
#undef identifier
```

- **File Inclusion**

```
#include < file-name >
#include " file-name "
```

- **Conditional Compilation**

```
#if constant-expression
#ifdef identifier /* #if defined(identifier) */
#ifndef identifier /* #if !defined(identifier) */

#elif constant--expression
#else
#endif
```



27

The C Preprocessor (Cont.)

A common use of the `#ifndef` command is in **header files**. It is, usually, harmful to include a header file more than once. Since file inclusion is transitive, a file may inadvertently be included more than once, through inclusion of other files. A strong mechanism that prevents multiple inclusion, is that each file defines a unique MACRO-identifier once included, and "refuses" inclusion if this identifier is `#defined`.

Example: (a header file named "list.h")

```
#ifndef LIST_H
#define LIST_H    /* Prevents entering here in
                  future inclusions */
    (Here comes the content of the header file)
#endif    /* LIST_H */
```

28

Structures

Syntax: `struct [name] {fields-list}`

- Define **new types** that are built from several, simpler types.
- Allow access to each component.
- Almost everything that can be done with built-in types (`int`, `double`), is legal for structures.
 - Legal: Arrays of structures, functions that return structure values, etc.
 - Illegal: Overloading predefined *arithmetical / logical* operations.
- Structures can be **nested**.



29

Structures (Cont.)

Example: Complex Numbers

```

struct complex {
    double r, i;
};

typedef struct complex complex;

#define CMLPX_ADD(x,y,z) {(z).r = (x).r + (y).r; \
                          (z).i = (x).i + (y).i; \
                          }

complex ComplexAdd(complex x, complex y)
{ complex z;
  z.r = x.r + y.r;
  z.i = x.i + y.i;
  return z;
}

```

30

Structures (Cont.)

Example:

```

struct person {
    unsigned long id;
    char *name;
    short gender;
    struct {
        char *str_n_num,
        *city;
        unsigned zip;
    } address;
    struct person *father;
    /* The above is legal because 'father' is only a pointer */
};

```



31

Structures (Cont.)

Usage:

```

unsigned long id, zip;
char *name1, *name2;
struct person per_var, *per_ptr;
    . . .
per_var.id = id;
(*per_ptr).id = ++id;
per_ptr->name = name1;
per_var.address.zip = per_ptr->address.zip;
per_ptr->father->name = per_var.father->name;

```

- **Note:** The field operators (`->`, `.`) have the same precedence (the highest among all operators), and are associated *left to right*.

32

Structures (Cont.)

- **Structures may be initialized:**

```
complex z = {2.3, -.4};
```

- **Structures returned by functions, may be used as such:**

```
double x = ComplexAdd(z1, z2).r;
```

- **Structure declarations do not constitute executable code:**

They merely describe a template (a blueprint)
to be used in building objects of a specific type.

This blueprint describes the memory layout of these objects.



Addresses and Pointers

- Every object in the computer memory has an **address**.
- **Some** of the objects in a **C** program may be referenced through the address of their location in memory.
 - Expressions like `&var`, are evaluated to the **address** of `var`.
- The **address operator**, `&`, cannot be applied to objects that have a temporary location in the memory (examples are: explicit constants, compound expressions).
- Addresses can be stored in variables of type **pointer to . . .**

Addresses and Pointers (Cont.)

- When `pvar` is a pointer variable carrying an address, the **dereferencing** (or **indirection**) operator, `*`, is used to extract the value stored in that address (via the expression `*pvar`).
- The character `*`, is also used for the **declaration** of pointer type variables.



35

Addresses and Pointers (Cont.)

Example:

```
int i, *pi;          /* pi - a pointer to integer      */
                    /* Another words, *pi is an int */
i = 3; pi = &i;     /* Now, (*pi == 3) */
*pi = 2;           /* Now, ( i == 2 ) */
```

Memory Image

Address 0x6414 0x6480
 i

	3
--	---

0x6414	
--------	--

 pi (After line 2, above)

Address 0x6414 0x6480
 i

	2
--	---

0x6414	
--------	--

 pi (After line 3, above)

36

Addresses and Pointers (Cont.)

- In order to dereference a pointer, it **must be known** to which **type** it refers.
- Objects of different types may occupy spaces of **different size**, e.g, char, int, float, double.

Example: (Showing how pointers differ from integers.)

char c[N];

--	--	--	--	--	--	--	--	--	--

char *pc = &c[0]; (*(pc+1)==c[1]);
0 1

int i[N];

--	--	--	--	--	--

int *pi = &i[0]; (*(pi+1)==i[1]);
0 1

double d[N];

--	--	--	--	--

double *pd = &d[0]; (*(pd+1)==d[1]);
0 1

- It is **illegal** to compare two pointers, **unless** they are known to point to a **single object** (e.g, an array), or to be **NULL**. Illegal comparisons are many times **possible**, but the results may be **surprising**.



37

Pointers and Arrays

Following a declaration like `double weight[LEN], *pw;` we have:

- `weight[i]` is an expression of type **double** that refers to the value stored in the *i*'th entry of the array.
- `weight` is an expression of the type **LEN-size array of double**.
- Under most circumstances, the expression `weight` is evaluated **as if** it was of type **pointer to double** that refers to the address of the first element of the array.
 - This means that `weight == &weight[0]` is always **TRUE**.
 - However, `sizeof(weight)` will **not** evaluate to `sizeof(double*)`
- **Fact:** The **C** compiler always translates an "array expression" like `weight[i]` into its equivalent "pointer expression" `*(weight+i)`

38

Pointers and Arrays (Cont.)

- However, when dealing with an object of type **array of ...** using the "array style expression" `weight[i]` is **recommended**, stressing the fact that we deal with an **actual array**.
- Assigning `pw = weight` will make the expression `pw[2]` have the value `weight[2]`
- The main **practical difference** between `pw` and `weight` is that `weight` **acts as a constant**, and `pw` **is a variable**, i.e, `weight` cannot be assigned to.

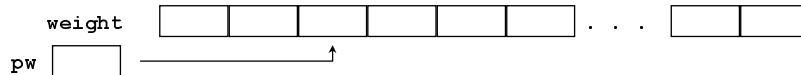


39

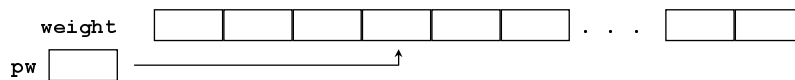
Pointers and Arrays (Cont.)

Example:

```
pw = weight + 2;
```



```
pw++; (now pw[1] is weight[4])
```



BUT `weight++;` **is illegal.**

40

Pointers and Function-Arguments

- In the C language, function arguments are passed only **by value**.
- It means that a variable that is used as an argument at a function call, will always retain its value when the function returns to the caller.

Example:

```
int i = 3, j = 4;
swap(i,j);
/* i is still 3,
 * j is still 4.
 */

void swap(int a, int b)
{ int t = a;
  a = b; b = t;
  return;
}
```



41

Pointers and Function-Arguments (Cont.)

- By using pointers as function arguments this restriction is overcome.

Example:

```

int i = 3, j = 4;
swap(&i, &j);
/* i is now 4,
 * j is now 3.
 */
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b; *b = t;
    return;
}

```

- This makes the call to `swap()`, practically a **call by reference**.
- In both cases, `a, b`, are **local** variables, which are initialized to the values of the arguments at the function call.

42

Arrays as Function-Arguments

An exception to the C “call by value” paradigm is for array parameters

- When an array is used as an argument at a function call, the entire array is not copied, but only its address is passed.
 - For a one-dimensional array of integers `vec[SZ]`, that is used as an argument at a function call, `f(vec)` and `f(&vec[0])` are synonyms.
 - In that case, at the **definition** or **declaration** of the function `f`, `f(int *arr)` and `f(int arr[])` are synonyms too.
 - If part of the array is to be transferred at the function call, then, e.g. `f(vec+2)` and `f(&vec[2])` are synonyms.
 - For multidimensional arrays, the above discussion is true for the first (leftmost) coordinate only.



43

Pointers to Functions

There are cases where there is a **function call** in a command, but there is no prior knowledge **which function** is to be called.

Example:

```
void *v1, *v2;
. . .
if (compare(v1,v2) == 0) {
. . .
```

v1, v2 may point to **integers** or **strings**, or other types. An appropriate **compare** function should be called, according to the **type** of the objects pointed to by v1, v2.

44

Pointers to Functions (Cont.)

This is how we deal with the situation:

```
void *v1, *v2;
enum {INT, STR} type;          /* coding actual type */
int (*compare)(void*,void*); /* a pointer to function */
. . .
switch(type) {
    case INT: compare = int_cmpr;
               break;
    case STR: compare = str_cmpr; /* calls strcmp() */
               break;
. . .
}
if ((*compare)(v1,v2) == 0) { /* compare(v1,v2) is legal too */
. . .
```

```
int int_cmpr(void *p, void *q)
{ return *(int*)p - *(int*)q;}
```



45

Pointers to Functions (Cont.)

- Another case of using a pointer to function is when a function is used as an **argument**, passed to another function (in principle, this is similar to the previous case: had we known a-priori a single function that would be used, there was no need to put it as an argument).

Example:

- A definition of a function which uses a function as an argument:

```
char*
string_manipulation(char s[], int (*chr_manip)(int))
{ for (int i=0; i < strlen(s); i++) /* C99 initialization */
  s[i] = chr_manip(s[i]);          /* A given operation on a char */
  return s;
}
```

- A call to that function:

```
char str[10] = "aBcD";
. . .
string_manipulation(str, tolower);
```

46

Dynamic Memory Allocation

- **C** allows general purpose dynamic memory allocation off the heap, restricted only by the amount of memory available at run time.
- There are three predefined functions for this: (in <stdlib.h>)

```
void *ptr = malloc(num_bytes_to_allocate);
void *z_ptr = calloc(num_of_obj, size_of_1_obj);
void *new_ptr = realloc(old_ptr, new_size_in_bytes);
```

- If memory allocation fails, these functions return a **NULL** pointer.



47

Dynamic Memory Allocation

(Cont.)

Example:

```
int *vec, *new;

if ((vec = (int*)malloc(arr_lng*sizeof(int))) == NULL) {
    fprintf(stderr,"cannot allocate\n");
    exit(1);    /* a MACRO will save you from code duplication */
}

if ((new = (int*)realloc(vec,new_lng*sizeof(int))) == NULL) {
    fprintf(stderr,"cannot allocate\n");
    exit(1);
} /* Now, vec points "nowhere" */
```

Casting is not obligatory

- **Dynamically allocated memory (only)** can be returned to the system using the function `free(old_ptr)`, which returns **void**.

48

Dynamic Memory Allocation Example

```
#include <stdio.h>
#include <stdlib.h> /* For malloc(), etc. */
#include <ctype.h> /* For tolower(), etc. */
#define LNG1 (6)
#define LNG2 (7)

char*
string_manipulation(char *s, int (*chr_manip)(int))
{
    while (*s != '\0') {
        *s = chr_manip(*s);
        s++;
    }
    return s;
}
```



49

Dynamic Memory Allocation Example (Cont.)

```

int
main()
{
    char *str, *new_str, *str1;

    str = (char *) malloc(LNG2);

    while (printf("Enter 6-char string\n")
           && fgets(str,LNG2,stdin)) {
        printf(" %s ==>> ",str);
        string_manipulation(str, tolower);
        printf("%s\n",str);
    }
}

```

50

Dynamic Memory Allocation Example (Cont.)

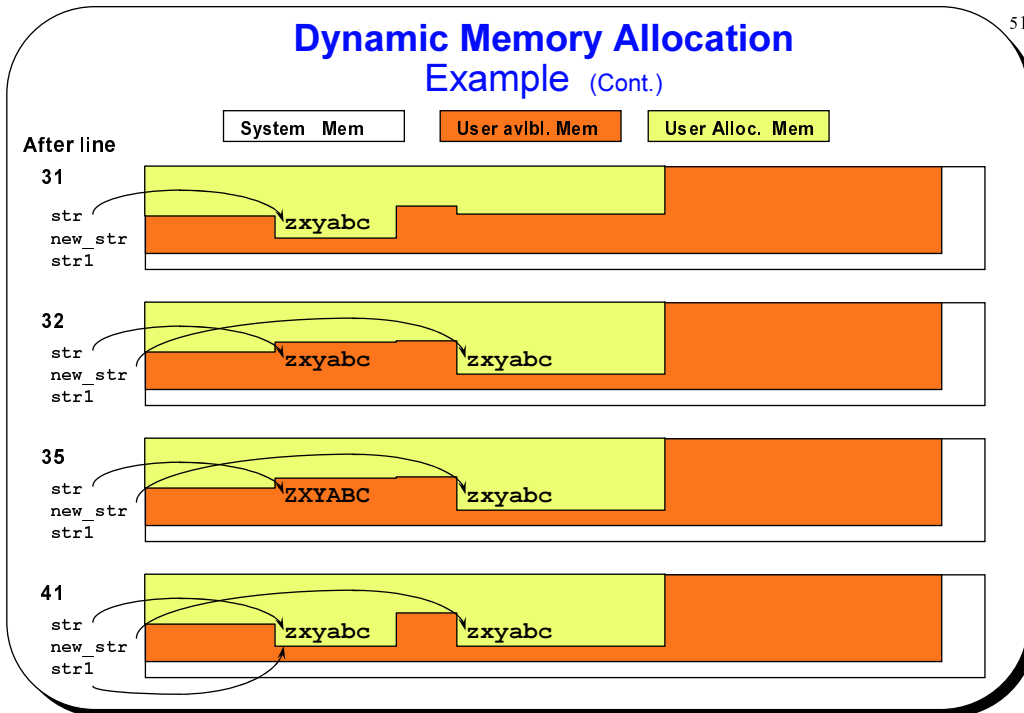
```

new_str = realloc(str, LNG2+LNG1);
printf("line %d:\tnew_str = %s\n", __LINE__, new_str); (line 33)
printf("line %d:\t    str = %s\n", __LINE__, str);    (line 34)
string_manipulation(str, toupper);
printf("line %d:\tnew_str = %s\n", __LINE__, new_str); (line 36)
printf("line %d:\t    str = %s\n", __LINE__, str);    (line 37)

str1 = malloc(LNG2);
printf("line %d:\t    str1 = %s\n", __LINE__, str1);  (line 40)
string_manipulation(str, tolower);
printf("line %d:\t    str1 = %s\n", __LINE__, str1);  (line 42)
return 0;
}

```





52

Dynamic Memory Allocation Example (Cont.)

A Sample Output

```

Enter 6-char string
  CdfsER ==>> cdfser
Enter 6-char string
  ZYXABC ==>> zyxabc
Enter 6-char string
line 33:      new_str = zyxabc
line 34:          str = zyxabc   This is a FREE memory area !!!
line 36:      new_str = zyxabc
line 37:          str = ZYXABC   A FREE memory is manipulated
line 40:          str1 = ZYXABC  This area is ALLOCATED again
line 42:          str1 = zyxabc  and manipulated by an OLD pointer
    
```

