Chapter 2

The C Programming Language

(Advanced Features)

Based on material prepared by C. Gotsman & Y.M. Kimchi

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.

Based on material prepared by C. Gotsman & Y.M. Kimchi

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);
        }
                . . .
```

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  */
```

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**

  `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.

Calling a **Function** vs. Calling a **MACRO**

* Is always an **expression**	* May be a **statement** (e.g, one that requires *automatic* 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 *stack* handling)	* No calling overhead

Based on material prepared by C. Gotsman & Y.M. Kimchi

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;}
```

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)
  - The compiler **may** check for type mismatch.
  - The debugger **may** recognize the symbolic names.

Switch and Enum Types Example

```
switch (mnth)  {
     case  JAN:
     case  MAR:
                 . . .
     case  DEC:  printf("31 days");
                 break;
     case  APR:
                 . . .
     case  NOV:  printf("30 days");
                 break;
     case  FEB:  if (leap_year)
                   printf("29 days");
                 else
                   printf("28 days");
                 break;
     default:    printf("month error");
                 break;
}
```

`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.
  - 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.

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];`

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".

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 */
```

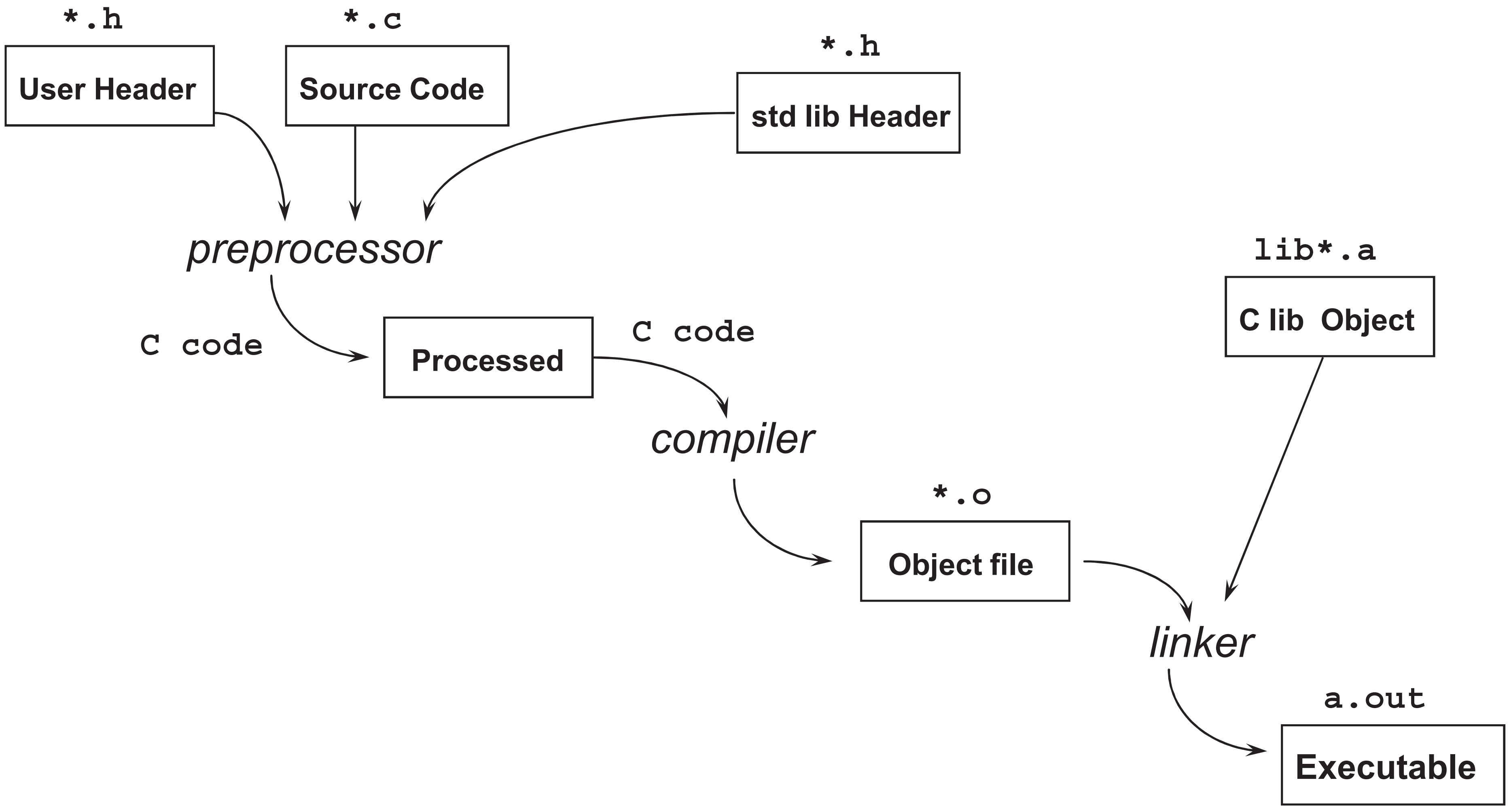
Casting (Cont.)

- 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   */
```

From **Source** to **Executable**



Based on material prepared by C. Gotsman & Y.M. Kimchi

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
```

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 */
```