Chap. 4 Functions and Program Structure

4.1 Basics of Functions

4.2 Functions Returning Non-integers **+ Function Prototype**

4.3 External Variables

4.4 Scope Rules

4.5. Header Files

4.6 Static Variables + Scope and Lifetime of Variables and Functions

4.7 Register Variables

4.8 Block Structure

4.9 Initialization

4.10 Recursion + System Calls

4.11 The C Preprocessor : C language facilities

cpp, file inclusion (#include), macro (#define), conditional (#if, #ifdef, #ifndef)

++ Example: basic STACK

++ Intro to Make

4.1 Basics of Functions (1/5)

- Each Function has the form :

```
return-type function-name (argument declarations)
{
        declarations and statements
}
```

- Various parts may be absent; a minimal function is : *dummy () {}*

- A program is just a set of definitions of variables and functions

- Communication between functions :
  - *by arguments*
  - *value returned by a function*
  - *external variables*

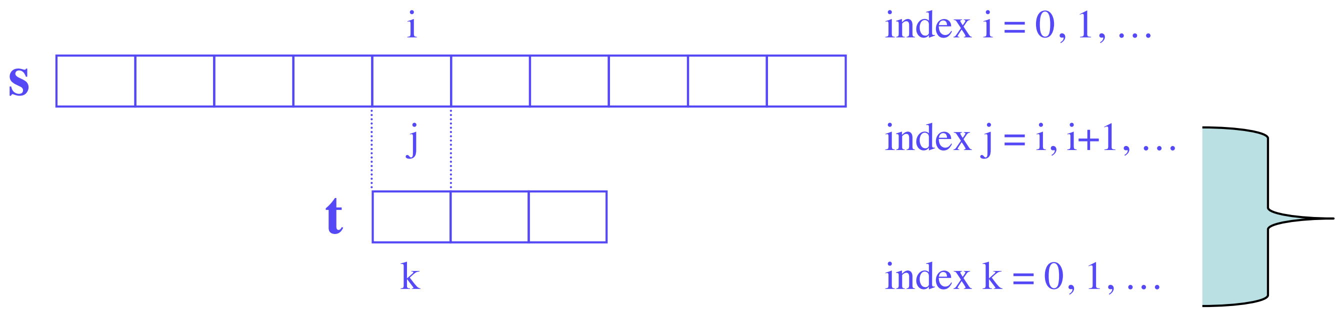
4.1 Basics of Functions (2/5) : An Example

Example: Write a program to print each line of its input that contains a particular “pattern”, i.e. a string of characters (this is a special case of the famous UNIX grep program!)

High level solution

```
while (there’s another line)          <---   getline(line)
    if (the line contains the pattern) <---   strindex(line,pattern)
        print it                        <---   printf(line)
```

Graphical solution for `strindex(s,t)` and C code excerpt



```
for (i = 0; s[i] != ‘\0’; i++)
    for (j = i, k = 0; t[k] != ‘\0’ && s[j] == t[k]; j++, k++)
        ;
```

4.1 Basics of Functions (3/5) : A Skeleton Program

```
#include <stdio.h>                      /* for printf() */
int getline(char line[]);
int strindex(char source[], char searchfor[]);
char pattern[] = "ould";    /* pattern to search for */
```

```
/* find all lines matching pattern  ------- KR p.69 */
main()
{
    char line[];

    while (getline(line) > 0)
        if (strindex(line, pattern) >= 0)
            printf("%s", line);
}
```

```
/* getline: get line into s, return length */
int getline(char s[]) {...}
```

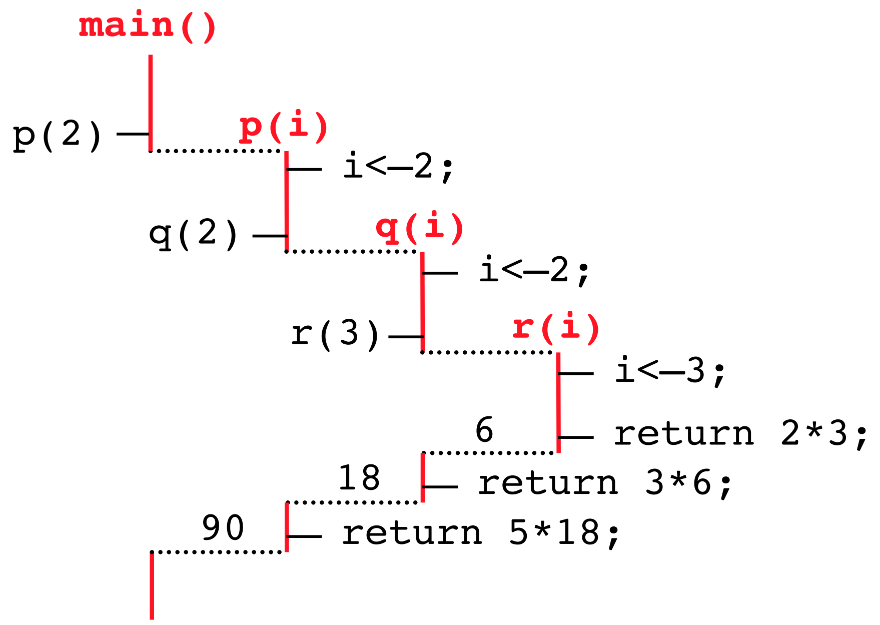
```
/* strindex: return index of t in s, -1 if none */
int strindex(char s[], char t[]) {...}
```

4.1 Basics of Functions (4/5) : Control Flow of a Function Call

A dummy program:

```
#include <stdio.h>
int r(int i) {return 2*i;}
int q(int i) {return 3*r(i+1);}
int p(int i) {return 5*q(i);}
main() { p(2); }
```

Control Flow of a Function Call:



4.1 Basics of Functions (5/5) :
** Function Declaration (Prototype) and Function Definition **

Function declaration, also named function prototype

- syntax:

```
[return_type] fct_name
            ([type [var] [, type [var]]*]);
```

- examples:

```
    int toto(char c1, char c2);
    int toto(char);  int toto();  toto();
```

- A **declaration** introduces an identifier and describes its type (what the compiler needs to accept references to that identifier)
  (*old-style C declaration*: a function declared without parameters "funct()" is treated as having an unspecified number of untyped parameters. A function with no parameters should be explicitly declared as "func(void)")
- A **definition** actually instantiates/implements this identifier.

To remember
A function can be *declared* multiples times, but *defined* only once

4.2 Functions Returning Non-integers

The **return** statement is the mechanism for returning a value:

```
    return expression;
```

If the return type is omitted, **int** is assumed.

Remark : the expression will be converted to the return type of the function if necessary (compilers might throw a warning in that case).

4.3 External Variables

A program is just a set of external objects, which are either variables or functions

External variables are defined outside any function

Functions are always external

C does not allow functions to be defined inside other functions

Internal variables to a function come into existence when the function is entered, and disappear when it is left *(unless they are defined as static)*

Remark: if an external variable is to be referred before it is defined, or if it is defined in a different file, then the `extern` declaration is mandatory
=> when `extern` is used with a variable, it is only declared not defined.

*** 4.4 Scope Rules ***

Definition

The ***scope*** of a name is the part of the program within which the name can be used.

Scope of an external variable or function

lasts from the point at which it is declared to the end of the *file* being compiled

Scope of an automatic variable (i.e. **local** variable of a function)

is the *block* in which the name is declared

Remark about the *declaration* and *definition* of external variables

A **declaration** announces the properties of a variable (its type and name), whereas a **definition** also causes storage to be allocated, e.g. :

```
extern int i;  extern char val[]; /* external decl. of i and val */
int i; char val[]; val2[8];       /* definition of i, val, val2 */
int j=1; char val3[8] = {'a','b'};/* def. and init. of j and val3 */
```

*Important property: an external object (variable, function, ...) can be **declared** multiple times (the declarations have to be identical), but only **defined** once. This is verified at compile time.*

*** 4.5 Header Files ***

As programs grow larger and larger (and include more files), it becomes increasingly tedious to have to forward declare every function you want to use that lives in a different file.

As much as possible, you want to centralize this, so that there is only one copy to get and keep right as the program evolves.

Typically, this common material is put in a *header file*, (.h), which will be included as necessary.

=> A header file contains **C declarations** and **macro** definitions to be **shared** between several source files

*** 4.6 Static Variables ***

Definition

1. The **`static`** keyword, applied to an *external variable or function*, constraints its scope to the rest of the source file *only* (it can only be accessed from the same C program file).
2. The **`static`** keyword, applied to an *automatic variable* (i.e. a variable defined locally to a function), promotes that variable to a permanent variable. (e.g., it remains in existence rather than coming and going each time the function is activated)

Usage

```
static char buf[BUFSIZE]; /* buffer for ungetch */
static int  bufp = 0      /* next free position in buf */
```

Remark

1. A **static external variable** is a "glocal variable with permanent storage" (*local* relative to the file in which it is declared, and *global* relative all functions defined to the rest of the file; *glocal* is a contraction of global and local)

2. A **static automatic variable** is a "local variable with permanent storage" (*local* relative to the function in which it is declared)

Static for External Variables

- If the external variable is to be visible within only one .c file, you should declare it as static.
- If the external variable is to be used across multiple .c files, you should not declare it as static. Instead you should declare it as extern in the files that need it.

Summary (1/2) : *** Scope and Lifetime of Variables and Functions ***

Characterization of var and fct scope (visibility)

Global (G): the scope is the whole program (more precisely from the point on where the var/fct has been declared to the end of the source file)

Semi-Global (SG) or Glocal: scope is in between global and local, e.g. the file where the var/fct has been declared and defined

Local (L): scope is the enclosed braces {} *(only for var, not possible for fct)*

Characterization of var lifetime

Permanent (P): lifetime is the same as the one of the program

Semi-Permanent (SP): lifetime is in between permanent and volatile (e.g. via malloc/free)

Volatile (V): lifetime is the same as the one of the enclosed braces {}, or of the associated function

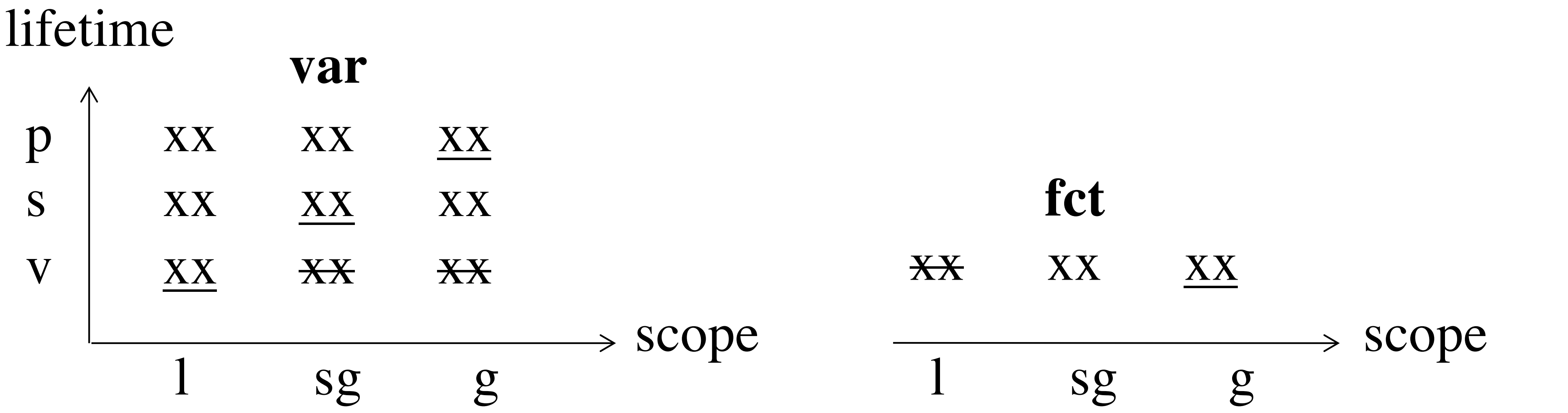
Characterization of fct lifetime

The lifetime of a fct corresponds always to the lifetime of the associated fct call.

Summary (2/2) :
*** Scope and Lifetime of Variables and Functions ***

```
int x;               // global/permanent - data segment
static int y;        // glocal/permanent - data segment

int f() {            // global            - text+stack segment
   int x;            // local/volatile    - stack segment
   static int y;     // local/permanent   - data segment
   int g() {}        // local, but not allowed in C
}
static int h() {} // glocal             - text+stack segment
```



Underlined: most common usage.   Strikethrough: doesn't exist in C.

4.7 Register Variables

Definition

A **register declaration** advises the compiler that the variable in question will be heavily used.

Remarks

1. The idea is that register variables are to be placed in machine registers (i.e. a small very rapid but very expensive memory).

2. The compilers are free to ignore the advice.

3. The register declaration can *only* be applied to automatic variables and to the formal parameters of a function.

Usage

```
void f(register unsigned m, register long n) {
   register int i;
   ...
}
```

4.8 Block Structure

Variable can be defined in a block-structured fashion

Declarations of variables (including initializations) may follow the left brace that introduces *any* compound statement, not just the one that begins a function.

Function cannot be defined in a block-structured fashion

Functions may not be defined within other functions.

Example

```
if (n>0) {
    int i;  /* declare a new integer i */
    for (i=0; i<n; i++)
        ...
}
```

The scope of the variable `i` *is the "true" branch of the if; this* `i` *is unrelated to any* `i` *outside the block.*

Remark. Let `av` and `sv` be an automatic resp. static variable declared and initialized in a block. Then `av` is initialized *each* time the block is entered and `sv` only the *first* time.
(Hint: `av` is a local variable *without* permanent storage and `sv` is a local variable *with* permanent storage!)

4.9 Initialization

- External and static variables: initialization at startup
  - *The initialization is done once, conceptually before the program begins execution*
  - *The initializer must be a constant expression*
  - *The value is initialized to zero in the absence of explicit initialization*
- Automatic and register variables: initialization at run time
  - *The initialization is done during the execution of the program, that is each time the function or block is entered*
  - *The initializer can be any expression*
  - *The value is undefined (i.e. garbage) in the absence of explicit initialization*
  - *When there are fewer initializers in a brace-enclosed list than the supposed number of elements, or fewer characters in a string literal used to initialize an array of known size, then the remainder of the aggregate is initialized implicitly as for its static counterpart*

Usage

```
int x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */

int binsearch(int x, int v[], int n) {
   int low=0, high=n-1, mid;
   …}
```

4.10 Recursion

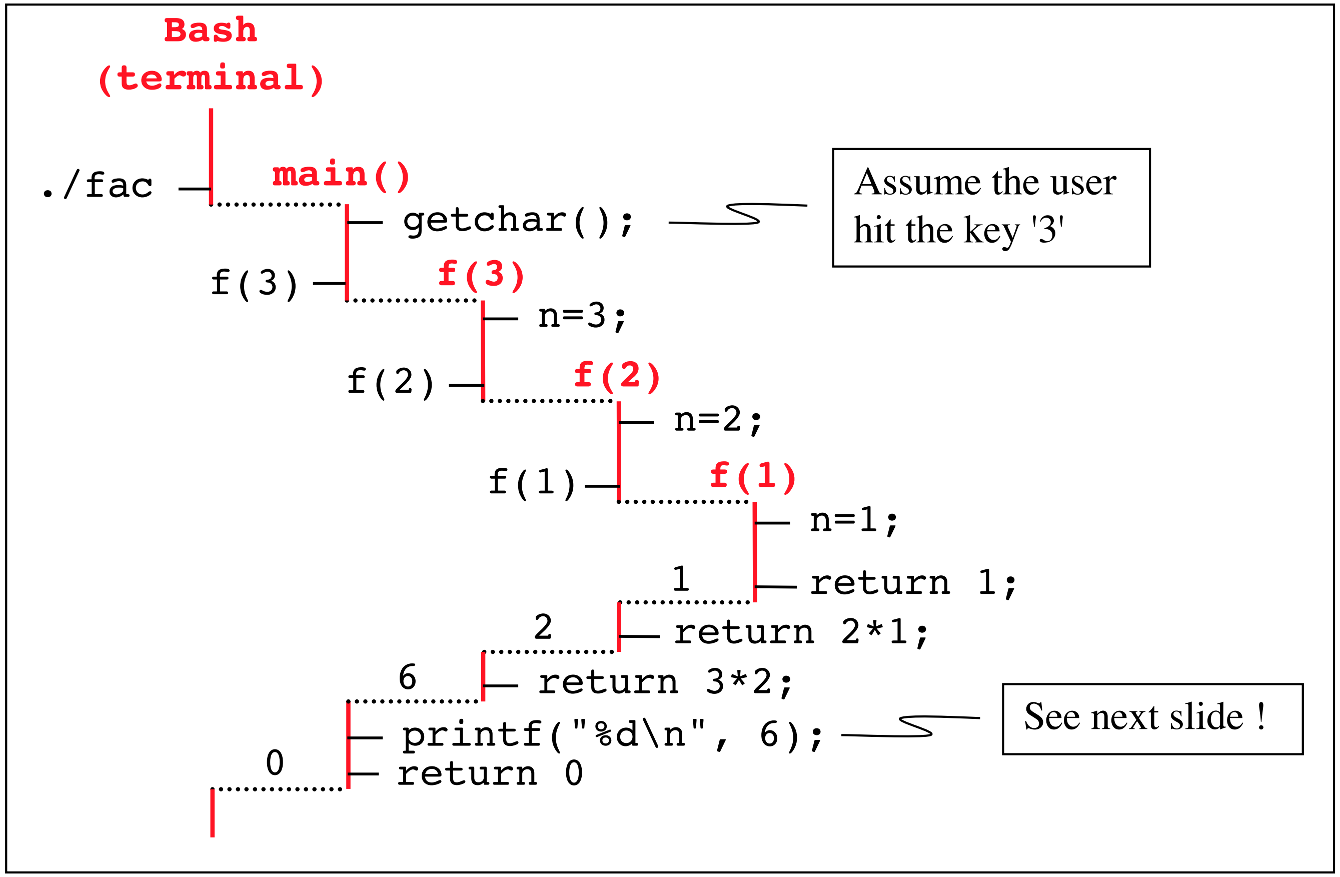
fac.c (factorial test program)

```
#include <stdio.h>
int f(int n) {if (n>1) return n*f(n-1); else return 1;}
main () {printf("%d\n", f(getchar()-'0')); return 0;}
```

In Bash (i.e. terminal)

```
% gcc -o fac fac.c
% ./fac
```

Control Flow of a Recursive Function Call (for % ./fac)

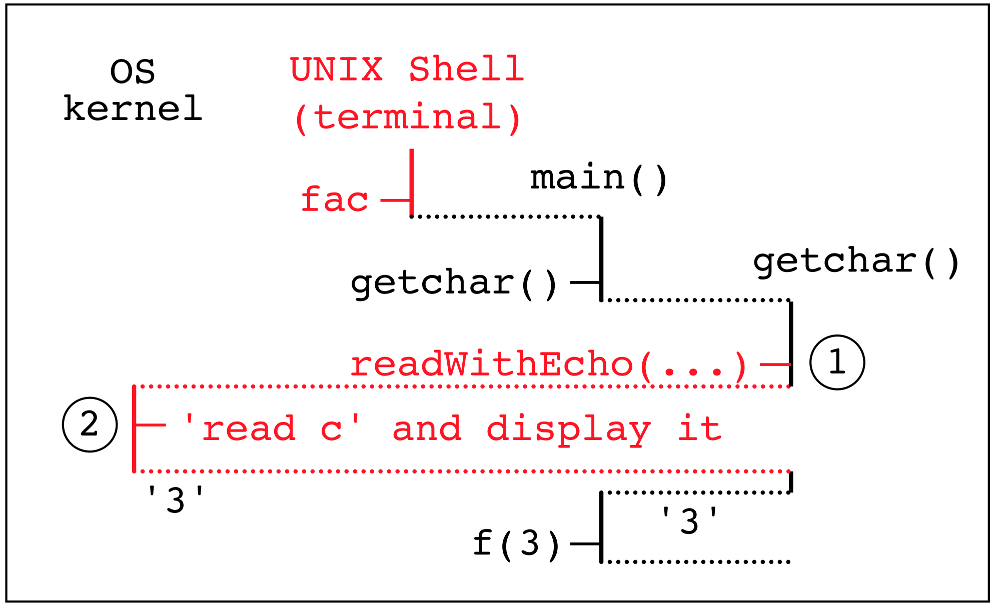


Try (hit 3 when needed)

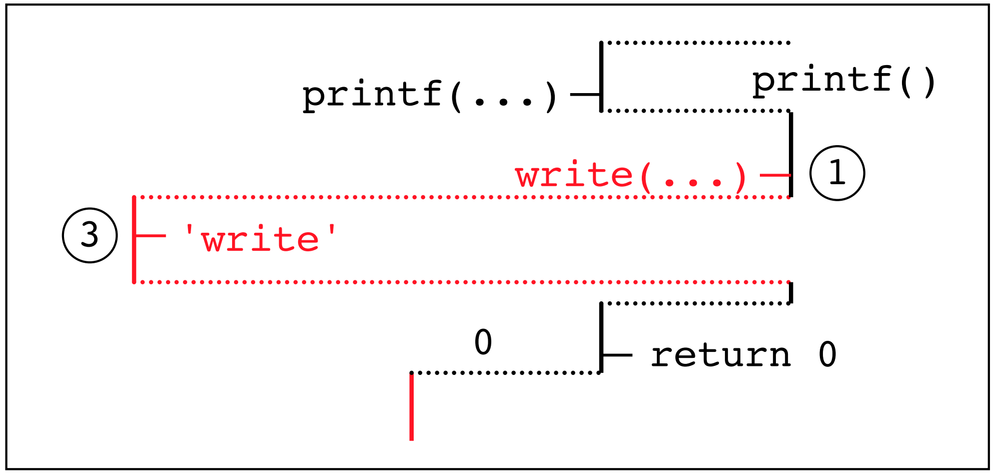
```
% ./fac ; ./fac
% ./fac && ./fac
% ./fac || ./fac
% ./fac | ./fac
```

UNIX Shell and System Calls

getchar()



printf()



(1) System call

(2) OS kernel is in charge to (1) read the next character from the standard input file, i.e. from the keyboard (assume the user hit the key '3'), (2) display it on the terminal window and (3) return it to getchar()

(3) OS kernel is in charge to write the received value to the standard output file, i.e. the terminal window

4.11 The C Preprocessor : C language facilities (1/3)

Ref: *"Macros and C Preprocessor: a first contact" tutorial*

• When you invoke gcc without options, it initiates a sequence of four basic operations:
(1) Preprocessing, (2) Compilation, (3) Assembly, (4) Linking.

• For a multifile program, (1) - (3) are performed on each individual source code file, creating an **object file** for each source code file; finally (4) combines all object codes.

• The **cpp** preprocessor allow you, among other things, to display the result of a *preprocessing* operation.

Example. Consider we have the following text in the 'macro.txt' file:

```
#define square(X)  (X)*(X)
square(2)
square(x+1)
```

and then invoke the cpp preprocessor:

```
% cpp macro.txt
(2)*(2)
(x+1)*(x+1)
%
```

Note that 'macro.txt' is not a correct C source code file; but it is fine for preprocessing.

4.11 The C Preprocessor : C language facilities (2/3)

File inclusion. The `#include` directive works by directing the C preprocessor to replace the directive by the contents of the corresponding file

Usage:
```
#include <file> // for system header files
#include "file" // for header files of your own program
```

Example:
```
#include <sdtio.h>

#include "my_stdio.h"
```

Macro. A *macro* is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the text contents of the macro.

Usage: `#define name replacement-text`

Example:
```
#define square(x) (x)*(x)
#define square(x) x*x  // what happens if square(z+1) is called?
```

Remark. A macro can be *defined* multiple times if the definitions are identical. The preprocessor will only complain if the definitions do not match. So it is not necessary to wrap up a macro with `#ifndef` in a header file.

4.11 The C Preprocessor : C language facilities (3/3)

Conditional. A conditional is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. For example, a program may need to use different code depending on the machine or operating system it is to run on.

There are 3 conditional directives: `#if`, `#ifdef` *and* `#ifndef`

A Typical Conditional Usage

```
#ifndef _CONST_H
#define _CONST_H
   const int buffer_size = 1024;
#endif /* _CONST_H */
```

assuming that this code is stored in the header file `const.h`.

Hint. In a complex program, it is hardly possible to avoid an indirect double file inclusion:

```
include <const.h>
include <const.h>
```

Without `#ifndef` the compiler will complain, as `buffer_size` would be *defined* twice.

Solution: wrap up this variable definition with the conditional directive `#ifndef`

Basic STACK: specification and implementation

Specification (.h file)

```
void stack_put(int e);
int stack_get();
```

Implementation : data structure (.c file)

```
/*--- sketch of the data structure ---------
           -------------------------------
           | * | * | * | * |   |   |   |
           -------------------------------
             |              top
            base
----------------------------------------------*/

//--- shared variables
```

```
#define CAPACITY 4            /* CAPACITY of the stack */
static int base[CAPACITY];   /* stack storage */
static int top = 0;           /* pseudo pointer referring
                              /* to the next free entry
```

Implementation : subroutines (.c file)

```
//--- stack subroutines
```

```
void stack_put(int e)
{
  if (top==CAPACITY) exit(EXIT_FAILURE); /* stack is full
  base[top++] = e;
}
```

```
int stack_get()
{
  if (top == 0) exit(EXIT_FAILURE); /* stack is empty
  return base[--top];
}
```

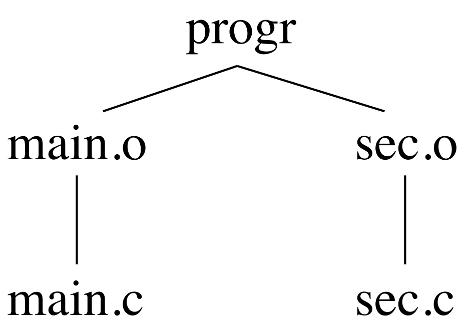
make (1/2) : Short Description + Example + Syntax

Short Description

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.

Example of a Makefile

```
(1) progr : main.o sec.o
(2)   gcc -o progr main.o sec.o
(3) main.o : main.c
(4)   gcc -c main.c
(5) sec.o : sec.c
(6)   gcc -c sec.c
```



Common Usage

```
% make
% make <target>
```

Syntax of a Makefile Rule

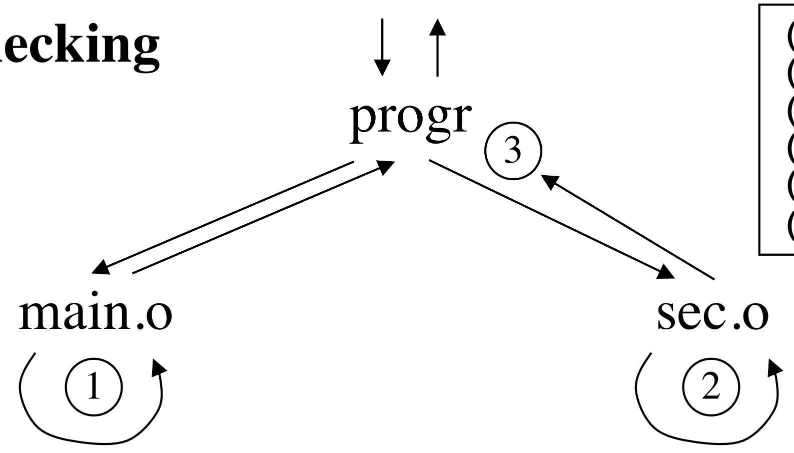
<target> : [<target's prerequisites>]<newline> *# Dependency line*

<tab><command line><newline> *# Show how to build the target out of their prerequisites*

[<tab><command line><newline>]

make (2/2) : Execution Model

Dependency checking



```
(1) progr : main.o sec.o
(2)   gcc -o progr main.o sec.o
(3) main.o : main.c
(4)   gcc -c main.c
(5) sec.o : sec.c
(6)   gcc -c sec.c
```

Execution Tree

① <u>if</u> (main.o don't exist) <u>then</u> <execute cmd (4)>
<u>else if</u> ($t_{main.o} < t_{main.c}$) <u>then</u> < execute cmd (4)> // main.o is out-of-date
<u>else</u> <do nothing>

② Similar to ①

③ <u>if</u> (progr don't exist) <u>then</u> <execute cmd (2)>
<u>else if</u> (($t_{progr} < t_{main.o}$) or ($t_{progr} < t_{sec.o}$)) <u>then</u> < execute cmd (2)> // progr is out-of-date
<u>else</u> <do nothing>

Remark 1. 'make' echoes each command to the standard output before executing it.
Remark 2. If <target's prerequisites> is absent the associated command lines are executed.