Engineering Computing I

The C programming Language

Chapter 4

Functions and Program Structure

Functions

- ❑ Functions break large computing tasks into smaller ones
- ❑ Functions enable programmers to build on what others have done instead of starting over from scratch.
- ❑ Appropriate functions hide details of operation from parts of the program that don't need to know about them thus clarifying the whole, and easing the pain of making changes.
- ❑ C has been designed to make functions efficient and easy to use
- ❑ C programs generally consist of many small functions rather than a few big ones.

Basics of Functions

To begin with, let us design and write a program to print each line of its input that contains a particular ‘‘pattern’’ or string of characters. (This is a special case of the UNIX program **grep**.) For example, searching for the pattern of letters ‘‘**ould**’’ in the set of lines

Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart’s Desire!

will produce the output

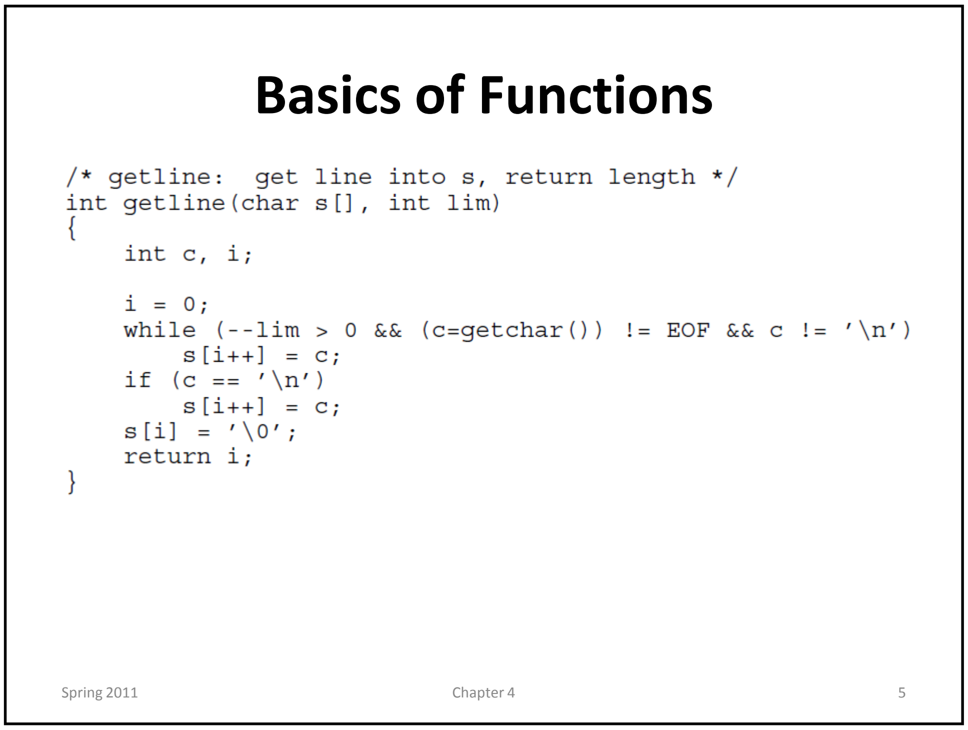
```
while (there’s another line)
    if (the line contains the pattern)
        print it
```

Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart’s Desire!

Basics of Functions

```
while (there’s another line)
    if (the line contains the pattern)
        print it
```

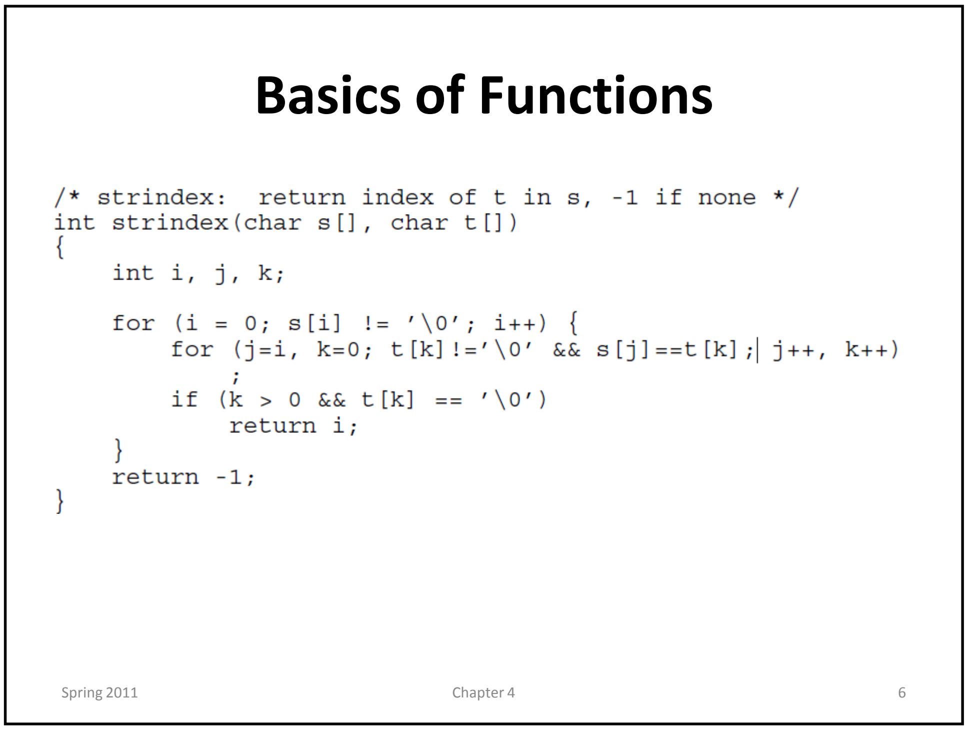
```
/* find all lines matching pattern */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}
```



Basics of Functions

```
/* getline:  get line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}
```

Spring 2011 Chapter 4 5



Basics of Functions

```
/* strindex:  return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Spring 2011 Chapter 4 6

Function Definition

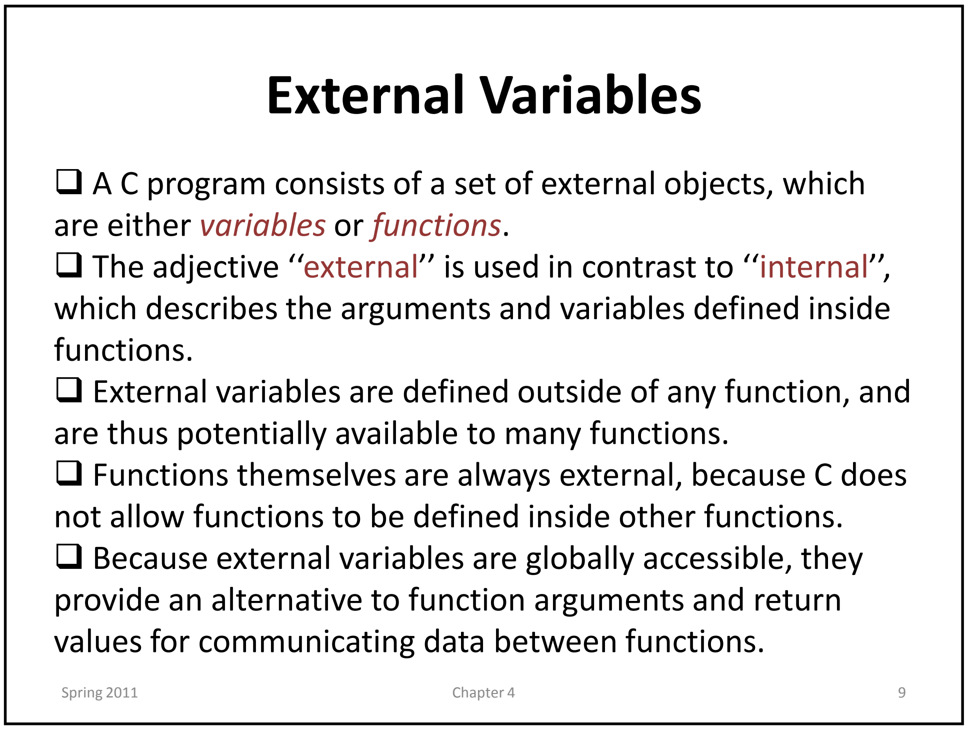
```
return-type function-name(argument declarations)
{
    declarations and statements
}
```

Minimal Function

```
dummy() {}
```

Functions Returning Non-integers

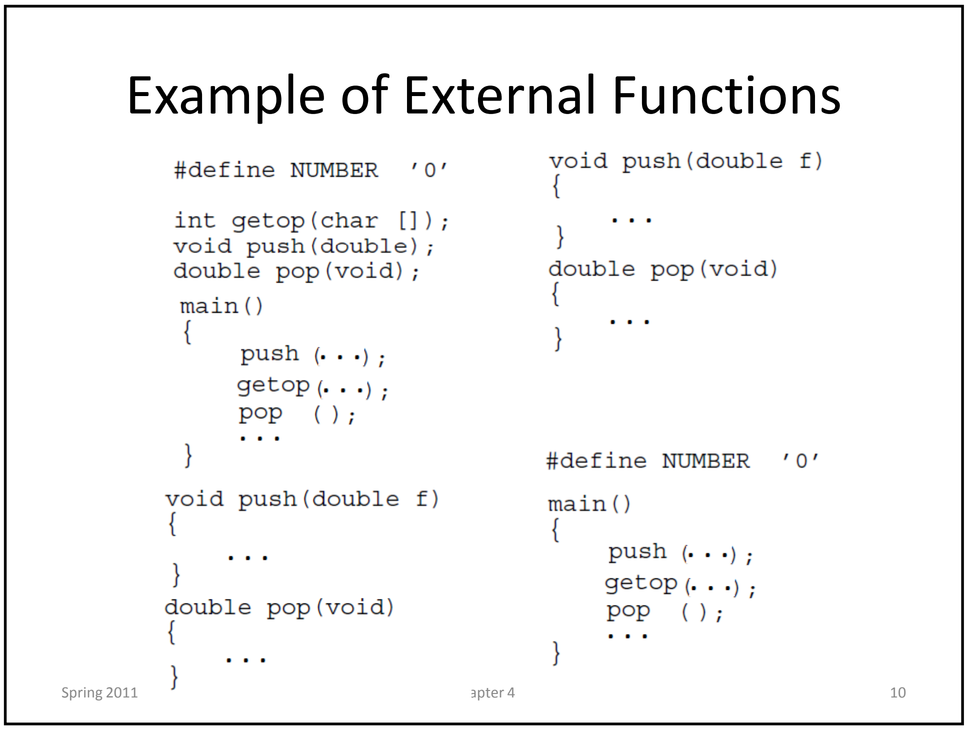
```
#include <ctype.h>

/* atof:  convert string s to double */
double atof(char s[])
{
    double val, power;
    int i, sign;

   ......
   ......
   ......
    return sign * val / power;
}
```



External Variables

❑ A C program consists of a set of external objects, which are either *variables* or *functions*.
❑ The adjective ''external'' is used in contrast to ''internal'', which describes the arguments and variables defined inside functions.
❑ External variables are defined outside of any function, and are thus potentially available to many functions.
❑ Functions themselves are always external, because C does not allow functions to be defined inside other functions.
❑ Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions.

Spring 2011 Chapter 4 9



Example of External Functions

```
#define NUMBER  '0'

int getop(char []);
void push(double);
double pop(void);
 main()
 {
        push (...);
        getop(...);
        pop  ();
        ...
 }
void push(double f)
{
      ...
}
double pop(void)
{
      ...
}
```

```
void push(double f)
{
      ...
}
double pop(void)
{
      ...
}
```

```
#define NUMBER  '0'

main()
{
      push (...);
      getop(...);
      pop  ();
      ...
}
```

Spring 2011 apter 4 10

Example of External Variables

in file1:

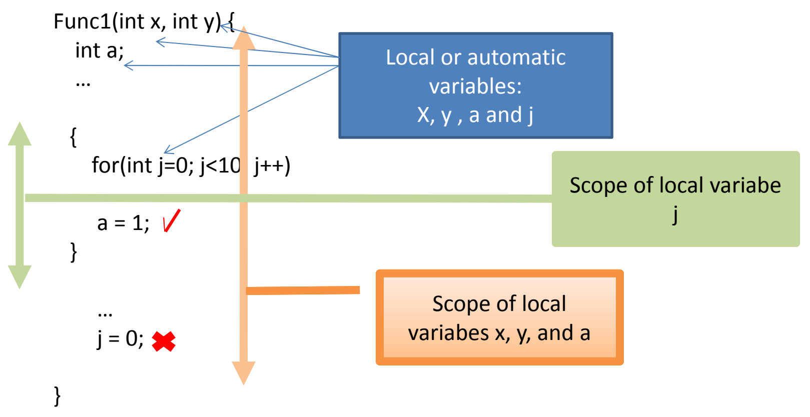
```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

in file2:

```
int sp = 0;
double val[MAXVAL];
```

Scope Rules

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



```
Func1(int x, int y){
  int a;
  ...

  {
    for(int j=0; j<10  j++)

    a = 1; ✓
  }

    ...
    j = 0; ✗

}
```

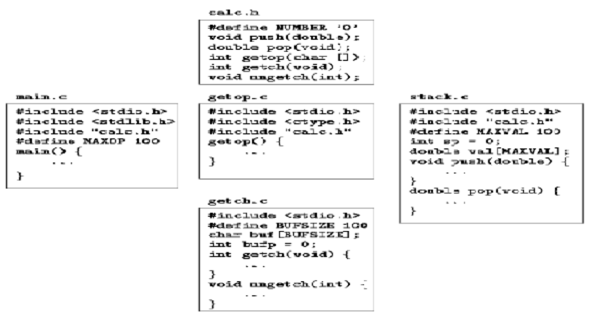
Local or automatic variables: X, y , a and j
Scope of local variabe j
Scope of local variabes x, y, and a

Multiple Source Files

```
extern  int d, e f;
int a;
void func1(args1);
void func2(args2) ;
extern void func3(args1);
extern void func4(args2) ;
main(…) { …}
int b;
extern  int c;
func1(args1) {…}
int c;
func2(args1) {…}
```

```
int d;
extern void func1(args1);
extern void func2(args2) ;

int e;
extern int f;
func3(args1) {…}
int f;
 func4(args1) {…}
```

Header Files



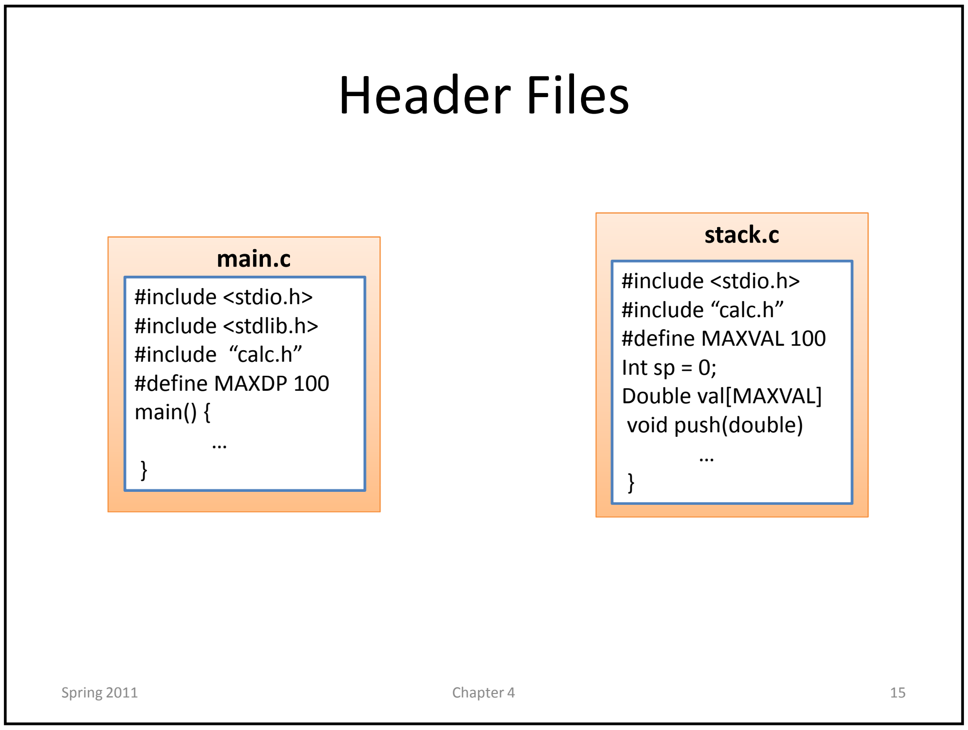
calc.h
```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c
```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c
```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c
```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c
```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```



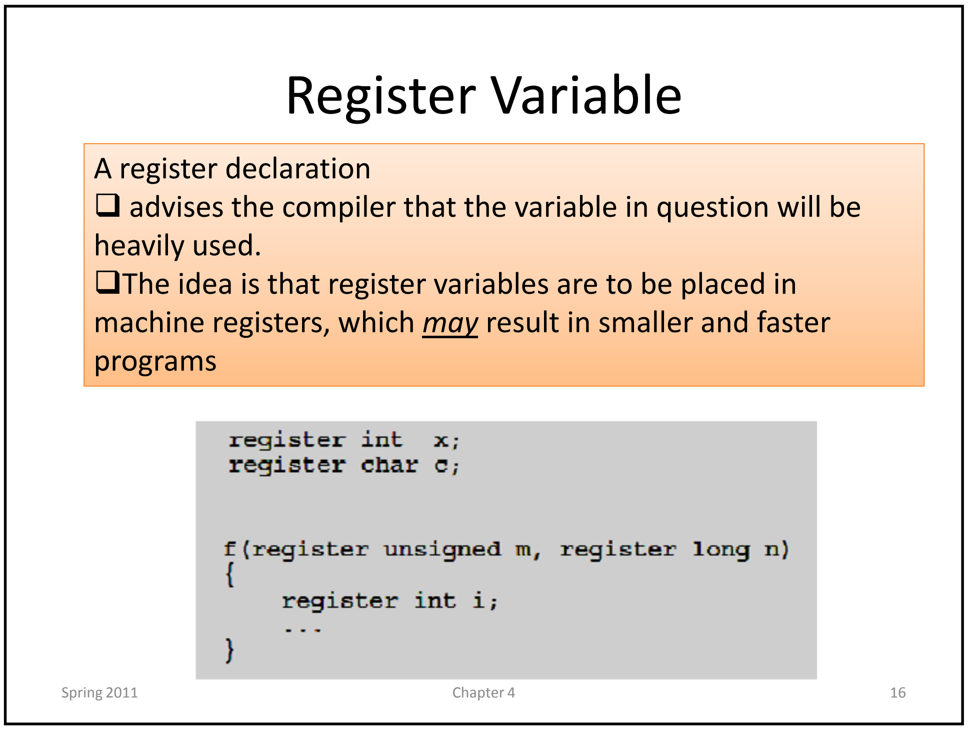
Header Files

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include  “calc.h”
#define MAXDP 100
main() {
          …
 }
```

stack.c

```
#include <stdio.h>
#include “calc.h”
#define MAXVAL 100
Int sp = 0;
Double val[MAXVAL]
 void push(double)
          …
 }
```



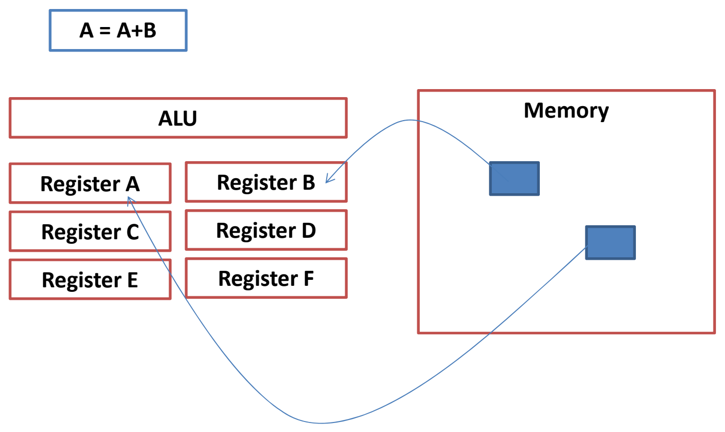
Register Variable

A register declaration

- ❑ advises the compiler that the variable in question will be heavily used.
- ❑The idea is that register variables are to be placed in machine registers, which *may* result in smaller and faster programs

```
register int  x;
register char c;


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

Register Variable



Block Structure

```
if (n > 0) {
    int i;  /* declare a new i */

    for (i = 0; i < n; i++)
        ...
}
```

Initialization

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

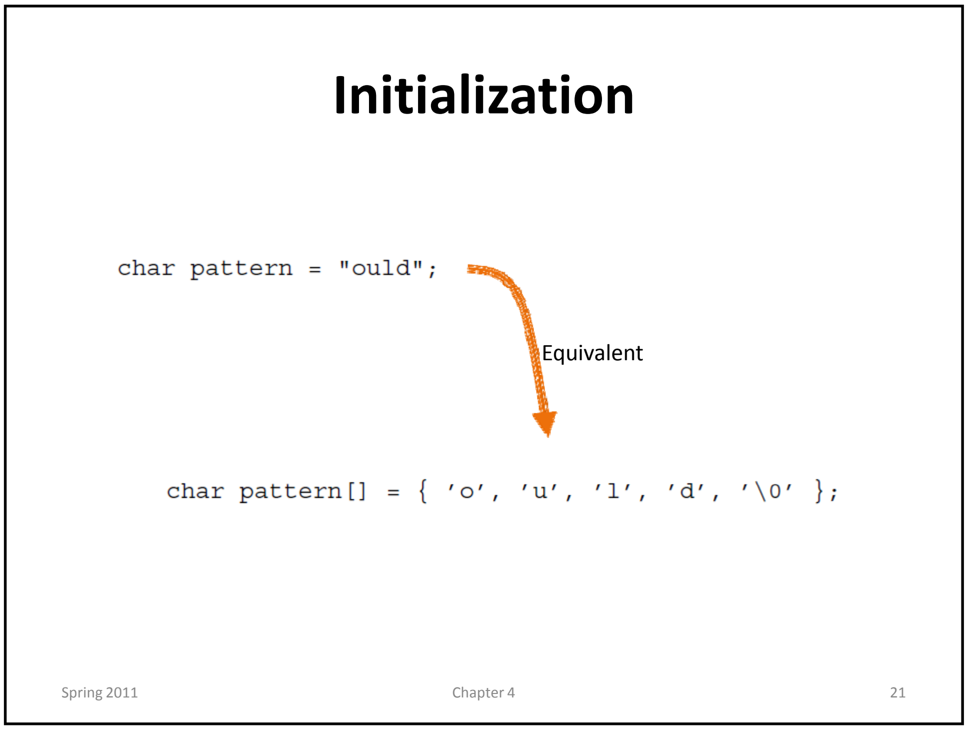
```
int low, high, mid;

low = 0;
high = n - 1;
```

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

Initialization

Size is omitted

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```



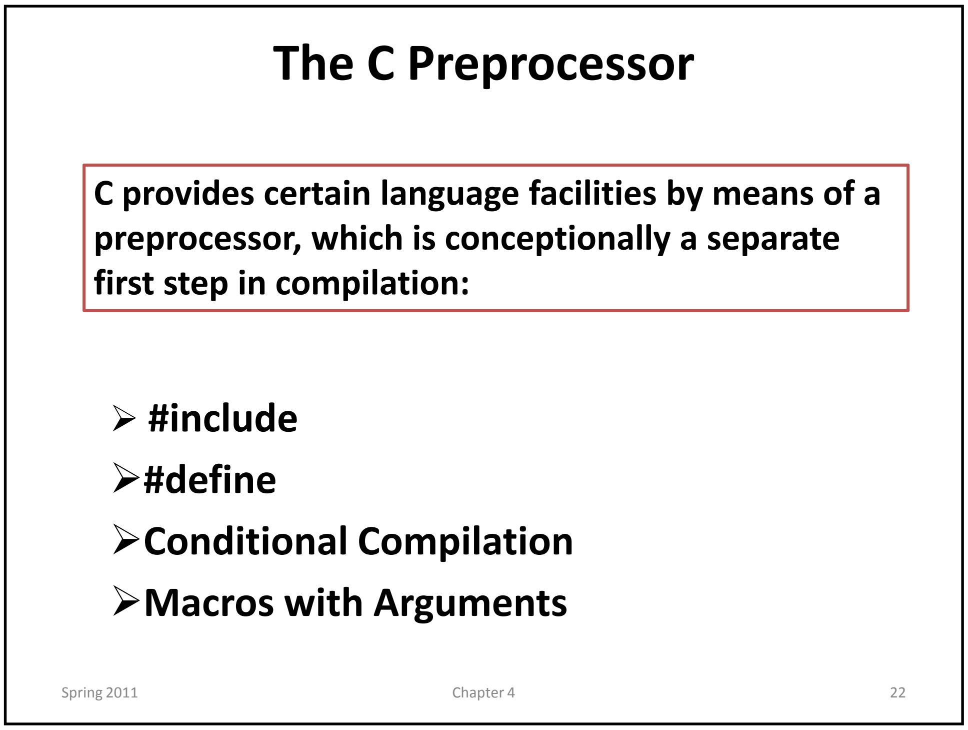
Initialization

```
char pattern = "ould";
```

Equivalent

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

Spring 2011 Chapter 4 21



The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptionally a separate first step in compilation:

- **#include**
- **#define**
- **Conditional Compilation**
- **Macros with Arguments**

Spring 2011 Chapter 4 22

File Inclusion

```
#include "filename"

#include <filename>
```

❑They include common #define statements and extern declarations and function prototypes
❑There are often several #include lines at the beginning of a source file
❑#include is the preferred way to tie the declarations together for a large program
❑It guarantees that all the source files will be supplied with the same definitions and variable declarations
❑when an included file is changed, all files that depend on it must be recompiled

Macro Substitution

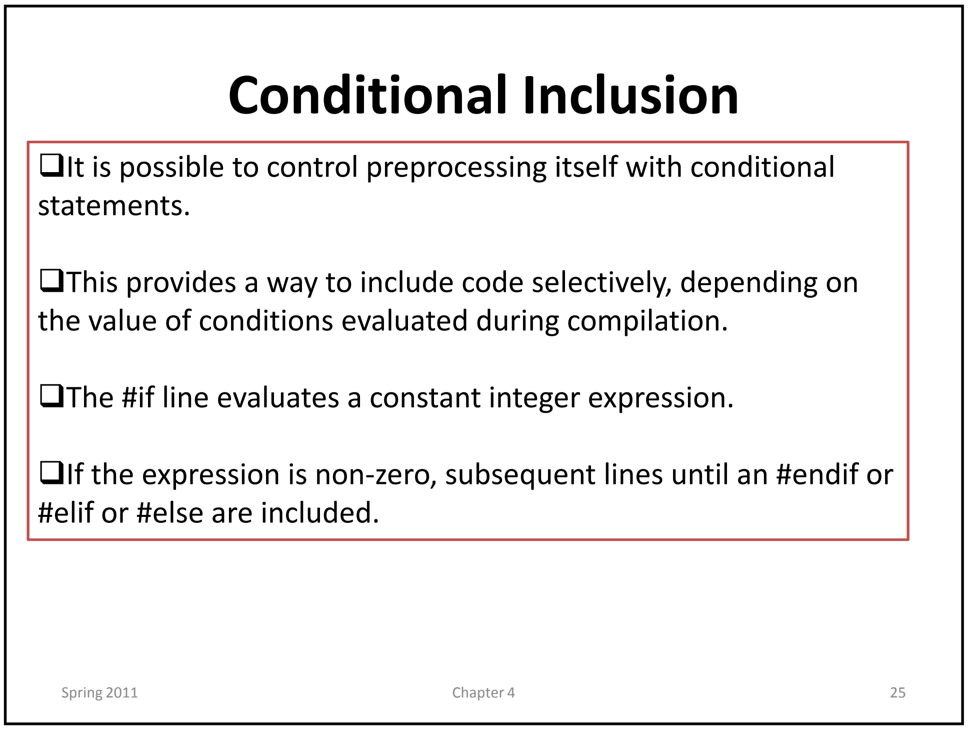
#define *name replacement text*

name = replacement text

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

```
x = max(p+q, r+s);  ➡  x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

```
#undef getchar
```



Conditional Inclusion

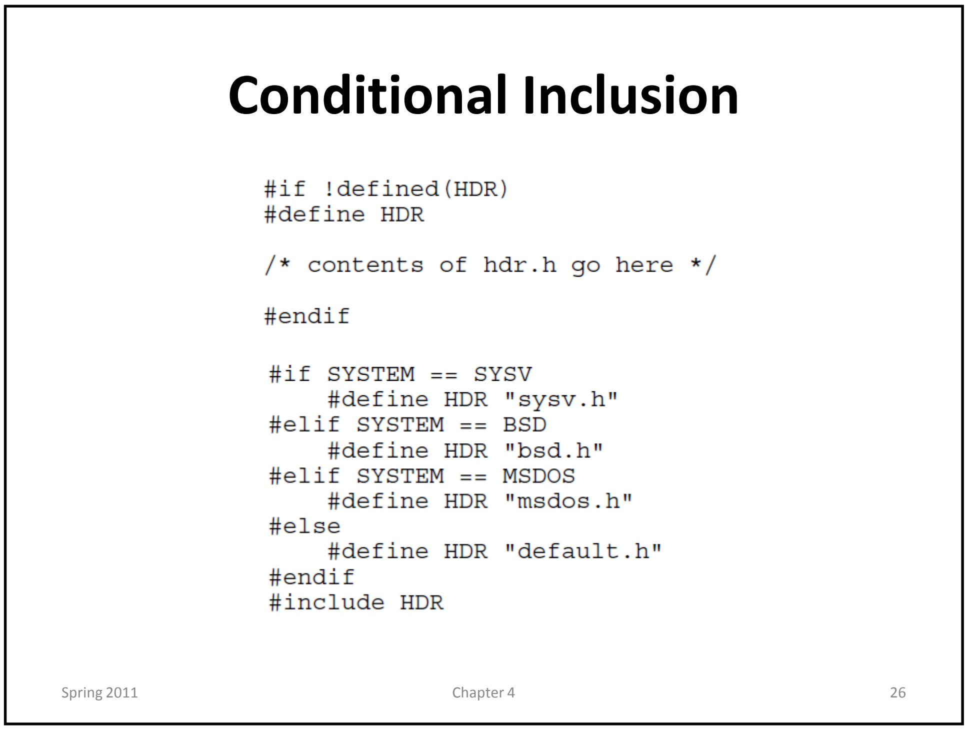
❑It is possible to control preprocessing itself with conditional statements.

❑This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

❑The #if line evaluates a constant integer expression.

❑If the expression is non-zero, subsequent lines until an #endif or #elif or #else are included.

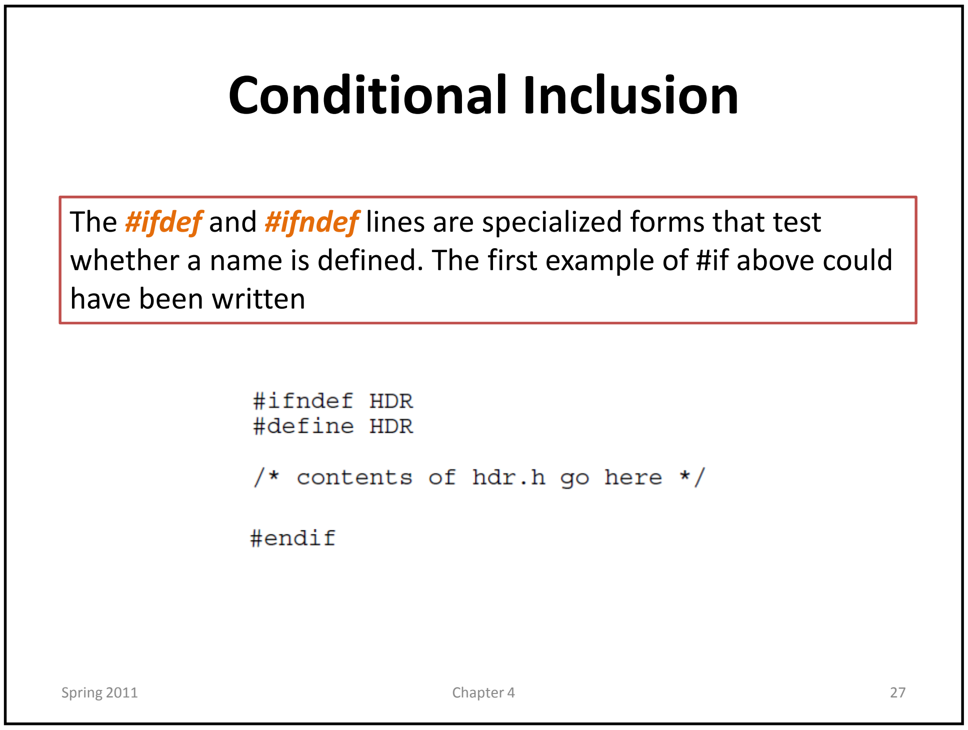
Spring 2011 Chapter 4 25



Conditional Inclusion

```
#if !defined(HDR)
#define HDR

/* contents of hdr.h go here */

#endif

#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Spring 2011 Chapter 4 26



Conditional Inclusion

The ***#ifdef*** and ***#ifndef*** lines are specialized forms that test whether a name is defined. The first example of #if above could have been written

```
#ifndef HDR
#define HDR

/* contents of hdr.h go here */

#endif
```

Spring 2011 Chapter 4 27