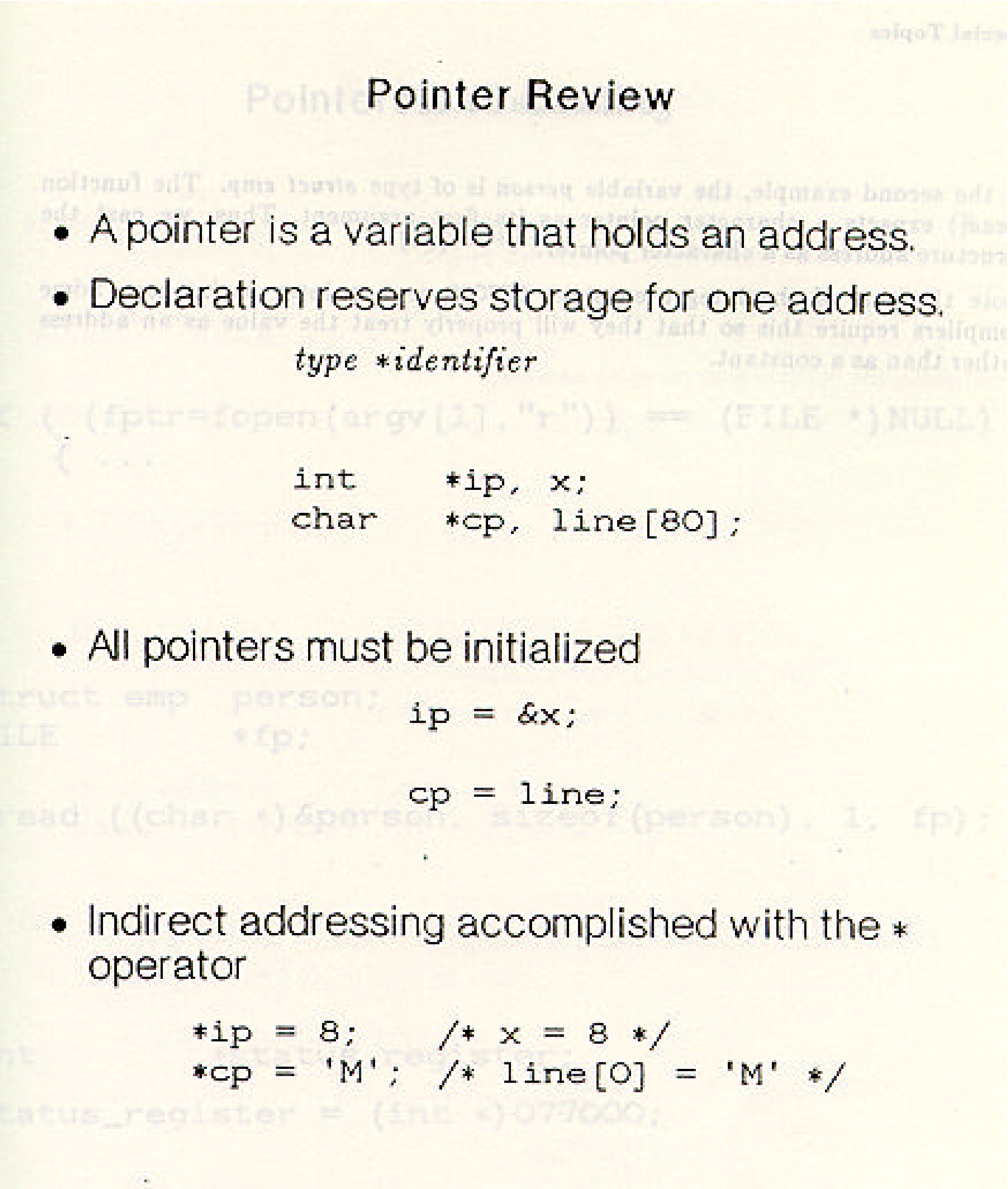
Special Topics

Unit 8 Objectives

- Use advanced pointer techniques
- Manipulate data with bit operators
- Use enumerated data types

Pointers Revisited



Pointer Review

- A pointer is a variable that holds an address.
- Declaration reserves storage for one address.

*type *identifier*

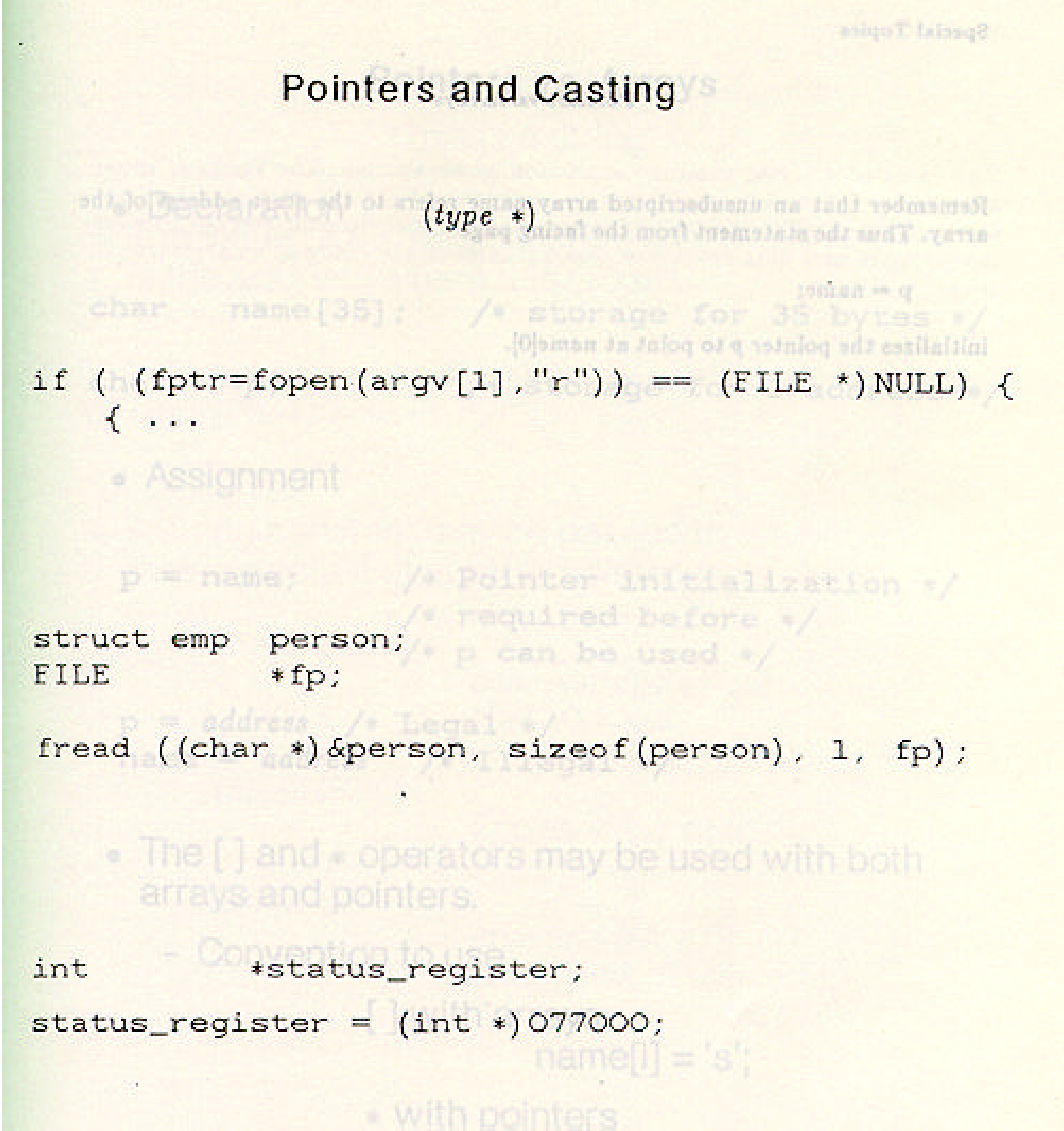
```
int     *ip, x;
char    *cp, line[80];
```

- All pointers must be initialized

```
ip = &x;

cp = line;
```

- Indirect addressing accomplished with the * operator

```
*ip = 8;    /* x = 8 */
*cp = 'M';  /* line[0] = 'M' */
```



Pointers and Casting

(*type* *)

```
if ( (fptr=fopen(argv[1],"r")) == (FILE *)NULL) {
    { ...
```

```
struct emp  person;
FILE        *fp;

fread ((char *)&person, sizeof(person), 1, fp);
```

```
int        *status_register;

status_register = (int *)077000;
```

Pointers vs. Arrays

- Declaration

```
char   name[35];    /* storage for 35 bytes */

char   *p;          /* storage for 1 address */
```

- Assignment

```
 p = name;          /* Pointer initialization */
                    /* required before */
                    /* p can be used */

 p = address   /* Legal */
 name = address   /* Illegal */
```

- The [ ] and ∗ operators may be used with both arrays and pointers.
  - Convention to use

[ ] with arrays

name[i] = 's';

∗ with pointers

p = &name[i];
∗p = 's';

String Constants

"This is a string"

- The value of a string constant expression is the address where it is stored.

```
/* printf() receives the string's address */
printf("Hello world\n");
```

- A string constant may be passed to a function that expects a character pointer.

SYNOPSIS

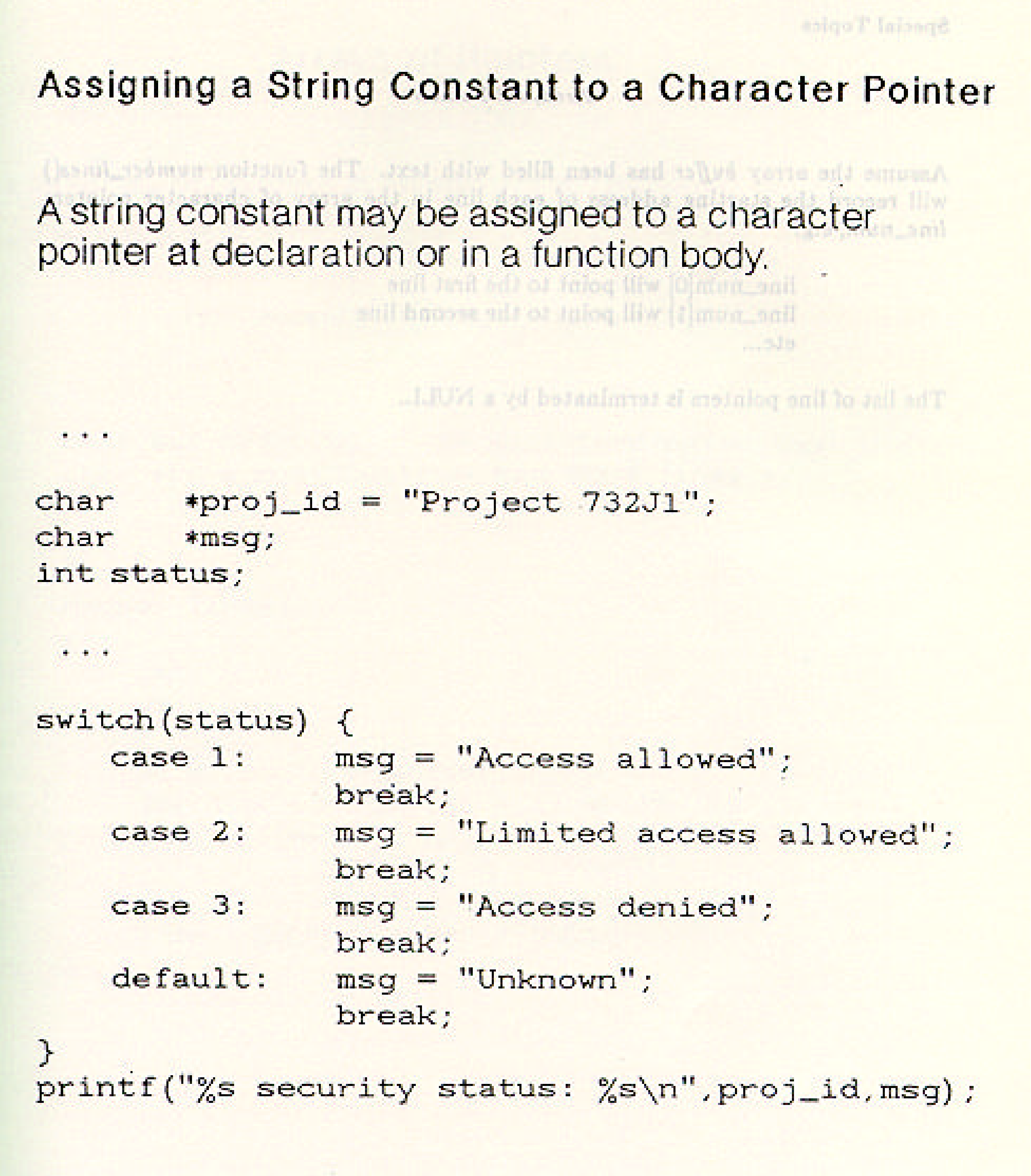
```
char *strcpy(s1, s2)
char *s1, *s2;
```

DESCRIPTION

Copies string s2 to s1.

EXAMPLE

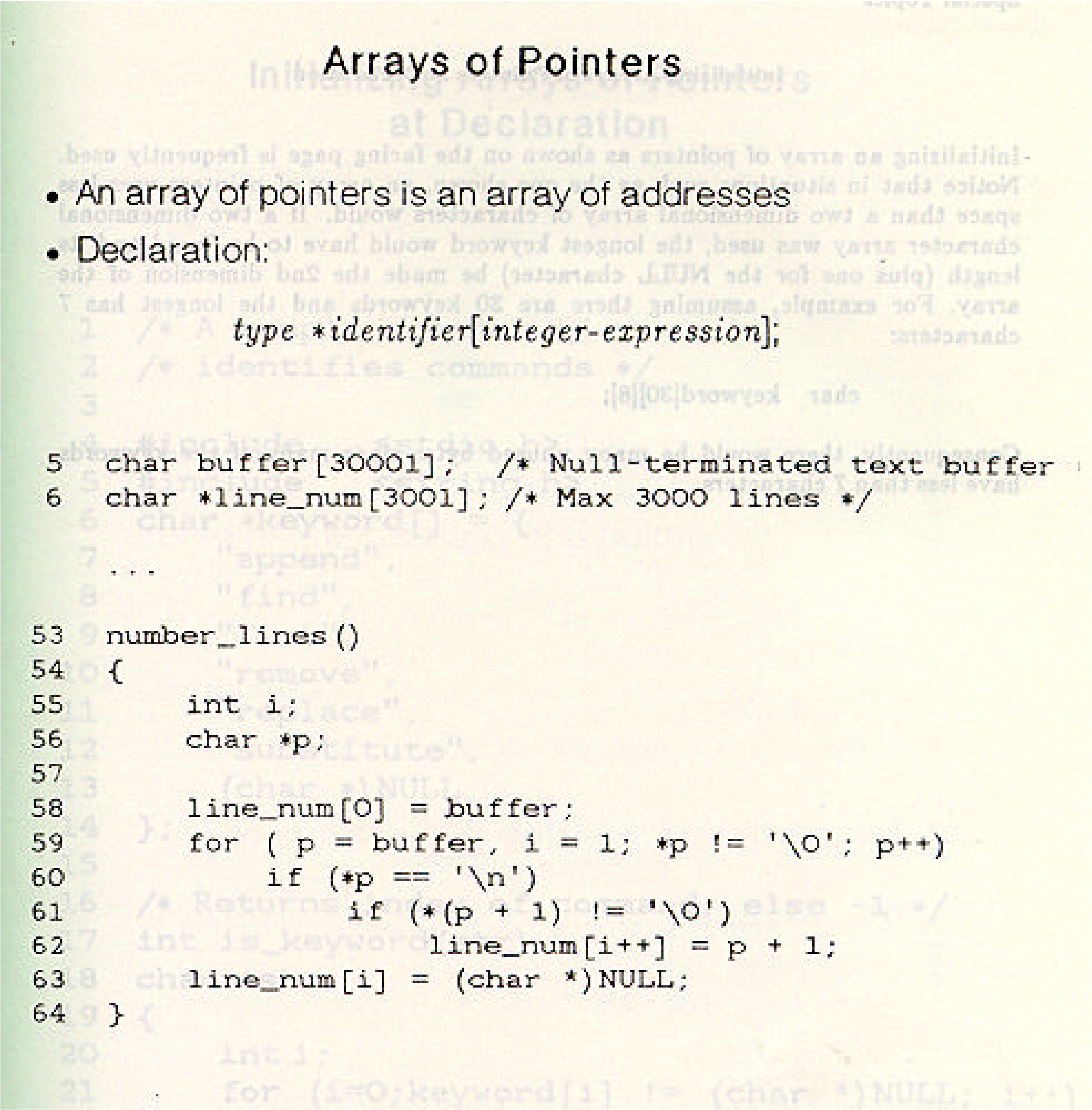
```
char line[80];
strcpy(line, "Reserved");
```



Assigning a String Constant to a Character Pointer

A string constant may be assigned to a character pointer at declaration or in a function body.

```
  ...

char    *proj_id = "Project 732J1";
char    *msg;
int status;

  ...

switch(status) {
    case 1:     msg = "Access allowed";
                break;
    case 2:     msg = "Limited access allowed";
                break;
    case 3:     msg = "Access denied";
                break;
    default:    msg = "Unknown";
                break;
}
printf("%s security status: %s\n",proj_id,msg);
```



Arrays of Pointers

- An array of pointers is an array of addresses
- Declaration:

type *identifier[integer-expression];

```
char buffer[30001];   /* Null-terminated text buffer
char *line_num[3001]; /* Max 3000 lines */

...

number_lines()
{
     int i;
     char *p;

     line_num[0] = buffer;
     for ( p = buffer, i = 1; *p != '\0'; p++)
          if (*p == '\n')
               if (*(p + 1) != '\0')
                    line_num[i++] = p + 1;
     line_num[i] = (char *)NULL;
}
```

Initializing Arrays of Pointers at Declaration

```
/* A sample scanner */
/* identifies commands */

#include    <stdio.h>
#include    <string.h>
char *keyword[] = {
    "append",
    "find",
    "list",
    "remove",
    "replace",
    "substitute",
    (char *)NULL
};

/* Returns index of command, else -1 */
int is_keyword(str)
char *str;
{
    int i;
    for (i=0;keyword[i] != (char *)NULL; i++)
        if (strcmp(str, keyword[i]) == 0)
            return(i);
    return(-1);
}
```

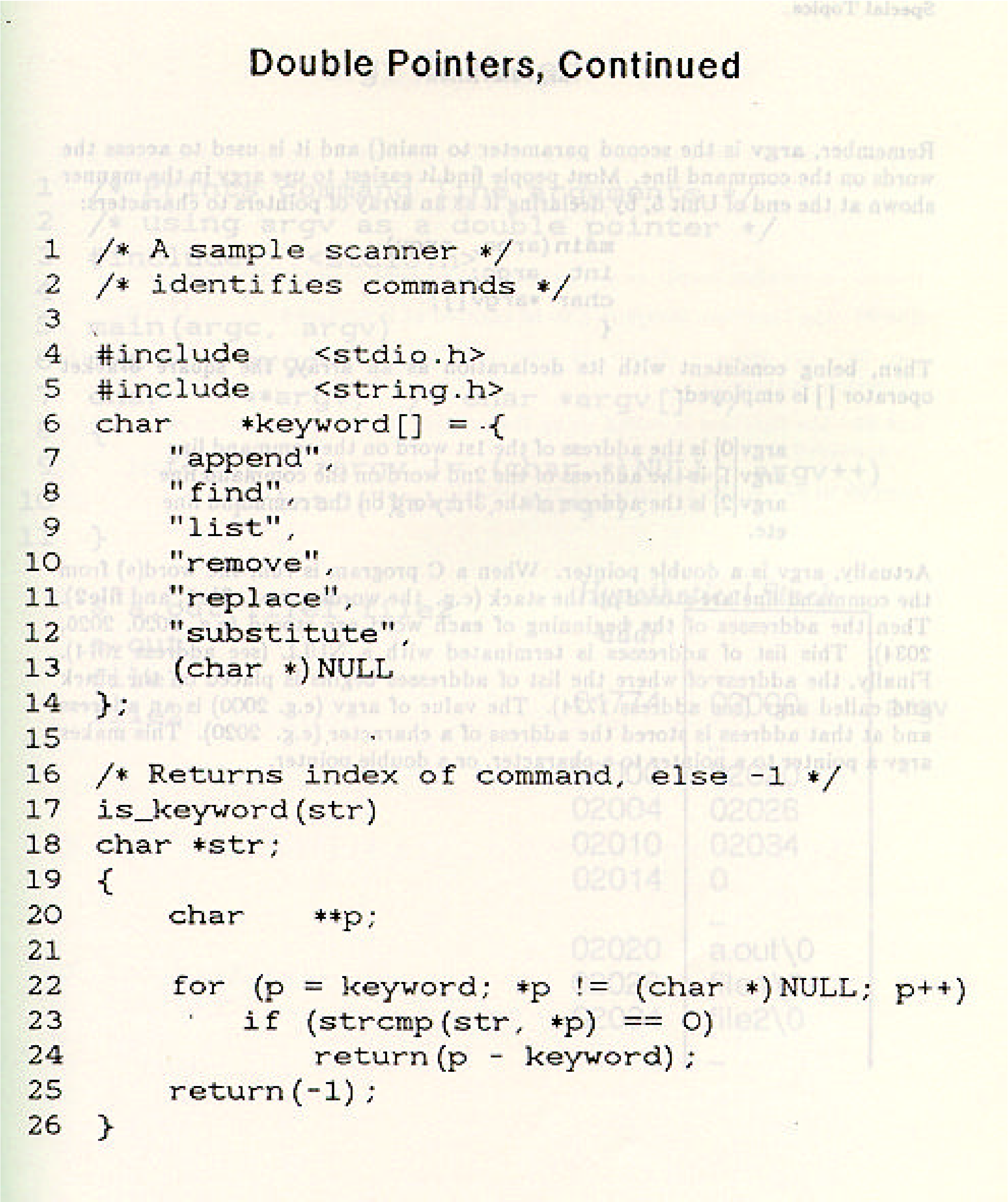
Double Pointers

- A pointer is a variable that contains the address of a variable

- A double pointer is a variable that contains the address of a pointer

- Declaration:

```
int   *p;   /* pointer */

int   **p;  /* double pointer */

int   ***p; /* triple pointer */

            /* and so on ... */
```



Double Pointers, Continued

```
/* A sample scanner */
/* identifies commands */

#include    <stdio.h>
#include    <string.h>
char    *keyword[] = {
    "append",
    "find",
    "list",
    "remove",
    "replace",
    "substitute",
    (char *)NULL
};

/* Returns index of command, else -1 */
is_keyword(str)
char *str;
{
    char    **p;

    for (p = keyword; *p != (char *)NULL; p++)
        if (strcmp(str, *p) == 0)
            return(p - keyword);
    return(-1);
}
```

argv Revisited

```
/* Prints command line arguments */
/* using argv as a double pointer */
#include    <stdio.h>

main(argc, argv)
int     argc;
char    **argv;   /* char *argv[] */
{
    for ( ; *argv != (char *)NULL; argv++)
          printf("%s\n", *argv);
}
```

```
$ a.out file1 file2
a.out
file1
file2
```

Hypothetical Stack

addr		
	...	
01774	02000	argv
	...	
02000	02020	
02004	02026	
02010	02034	
02014	0	
	...	
02020	a.out\0	
02026	file1\0	
02034	file2\0	
	...	

Pointers to Functions

```
main()
{

    ...

     int (*funptr)();

    ...

     int maxfunc();

    ...

     funptr=maxfunc; /* initialize function pointer */

    ...

     c = (*funptr)(a,b); /* c = maxfunc(a,b) */

    ...

int maxfunc(i,j)
int i,j;
{
   return( i > j ? i : j );
}
```

Example - Pointers to Functions

```
main()
{
     int append(), find(), list();
     int remove(), replace(), substitute();

     static int (*command[])() = {
          append,
          find,
          list,
          remove,
          replace,
          substitute
     };

     int i, (*fp)();
     char string[80];
     ...

     scanf("%s",string);
     ...

     i = is_keyword(string);
     if( i == -1 )
          exit(0);

     fp = command[i]; /* (*command[i])() */

     (*fp)();

     ...
}
```

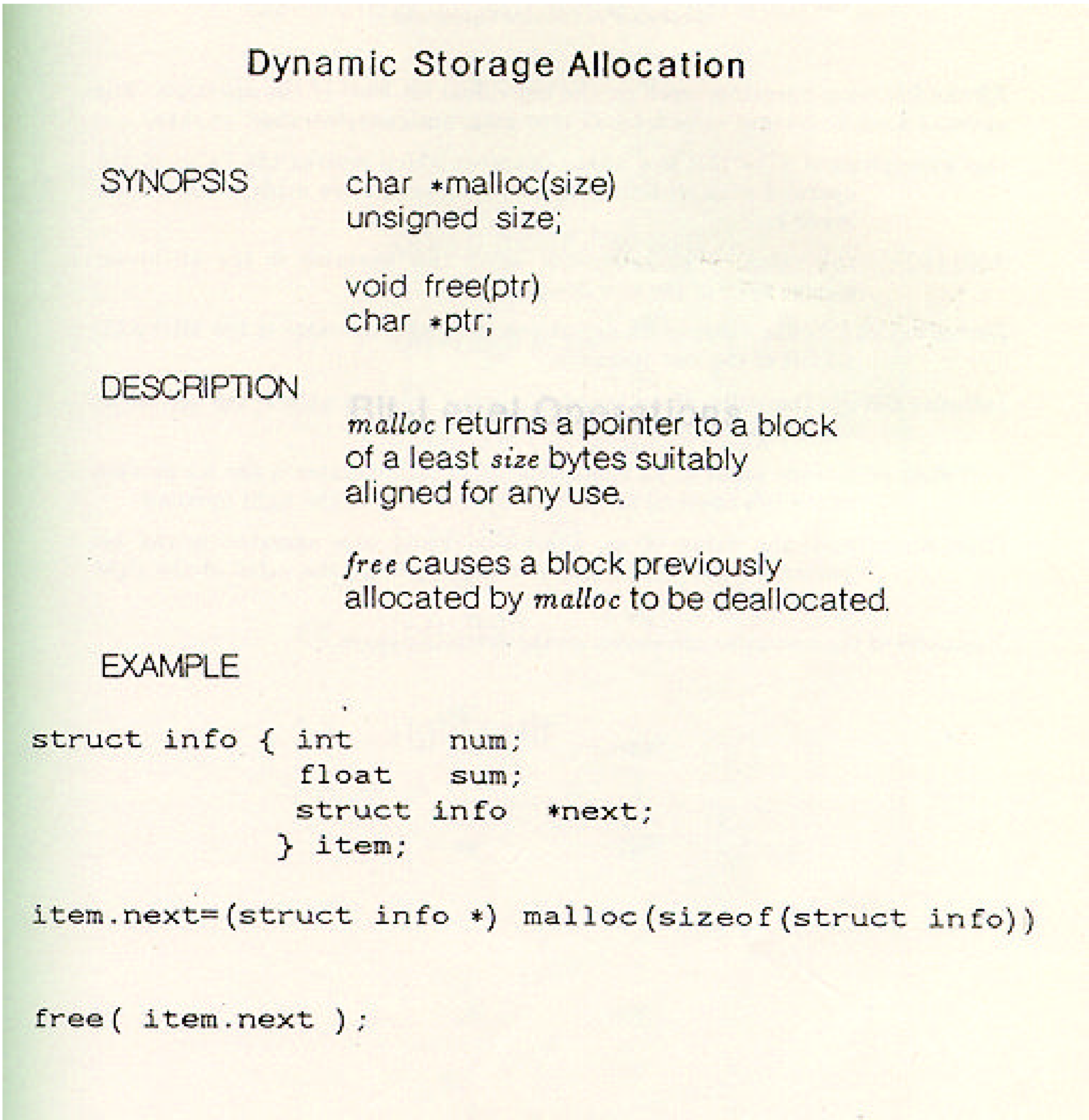
Interpreting Variable Declarations

1. Locate the identifier.

2. Look to its left and right; find the operator with the higher precedence.

3. Continue step 2, working outwards.

```
What are these declarations for?


    int  *ptr();


    int  (*ptr)();


    int  *ptr[10];


    int  (*ptr)[10];


    int   (*ptr[10])();
```



Dynamic Storage Allocation

SYNOPSIS

```
char *malloc(size)
unsigned size;

void free(ptr)
char *ptr;
```

DESCRIPTION

malloc returns a pointer to a block of a least *size* bytes suitably aligned for any use.

free causes a block previously allocated by *malloc* to be deallocated.

EXAMPLE

```
struct info { int      num;
              float    sum;
              struct info   *next;
            } item;

item.next=(struct info *) malloc(sizeof(struct info))


free( item.next );
```

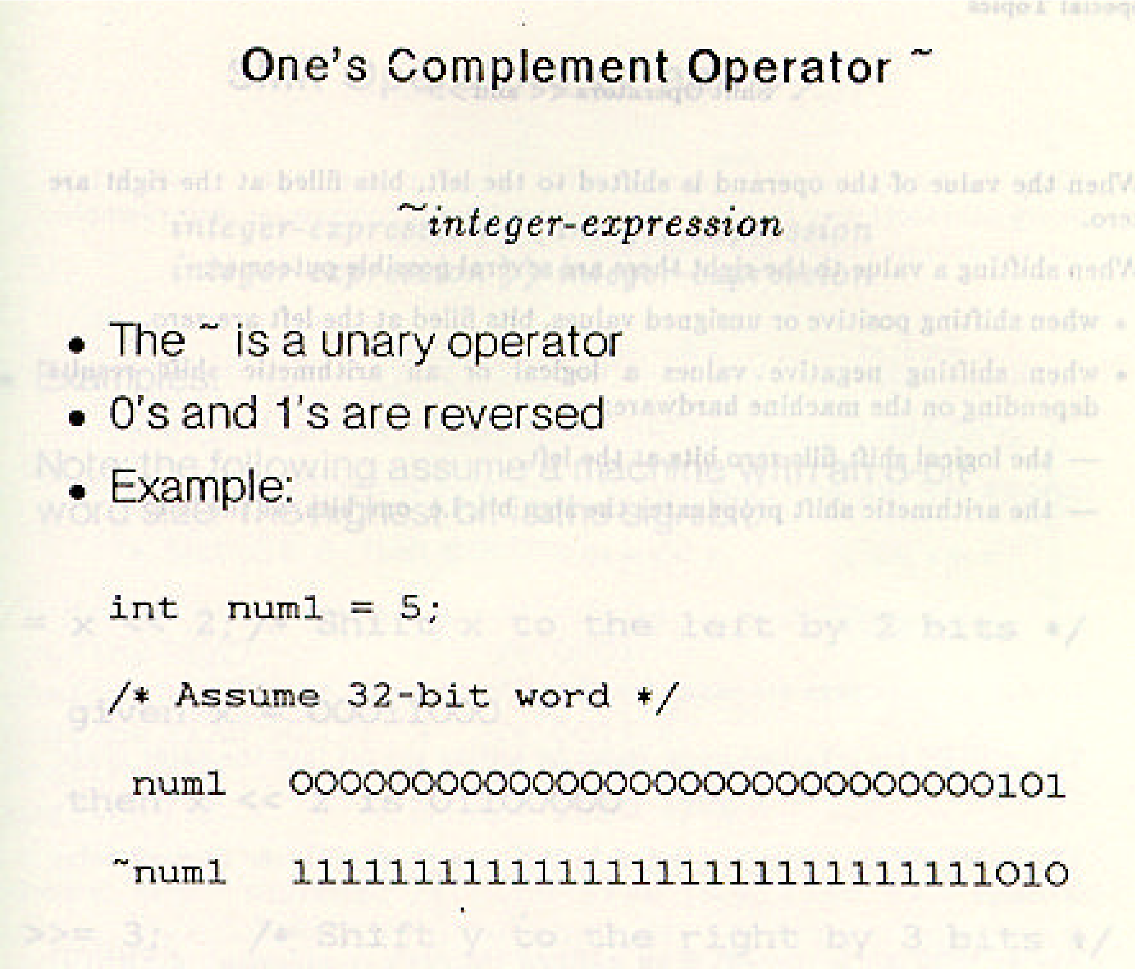
Bit-Level Operations

Overview of Bitwise Operators

~	one's complement
&	bitwise AND
^	bitwise EXCLUSIVE OR
¦	bitwise OR
<<	left shift
>>	right shift

Bitwise AND, OR, and EXCLUSIVE OR

```
Given:
      int      num1, num2;
      num1 = 5;   /* 00101 */
      num2 = 11;  /* 01011 */

Bitwise AND

             (num1 & num2)      num1    00101
                              & num2    01011
                                        -----
                                        00001

Bitwise OR

             (num1 | num2)      num1    00101
                              | num2    01011
                                        -----
                                        01111

EXCLUSIVE OR

             (num1 ^ num2)      num1    00101
                              ^ num2    01011
                                        -----
                                        01110
```

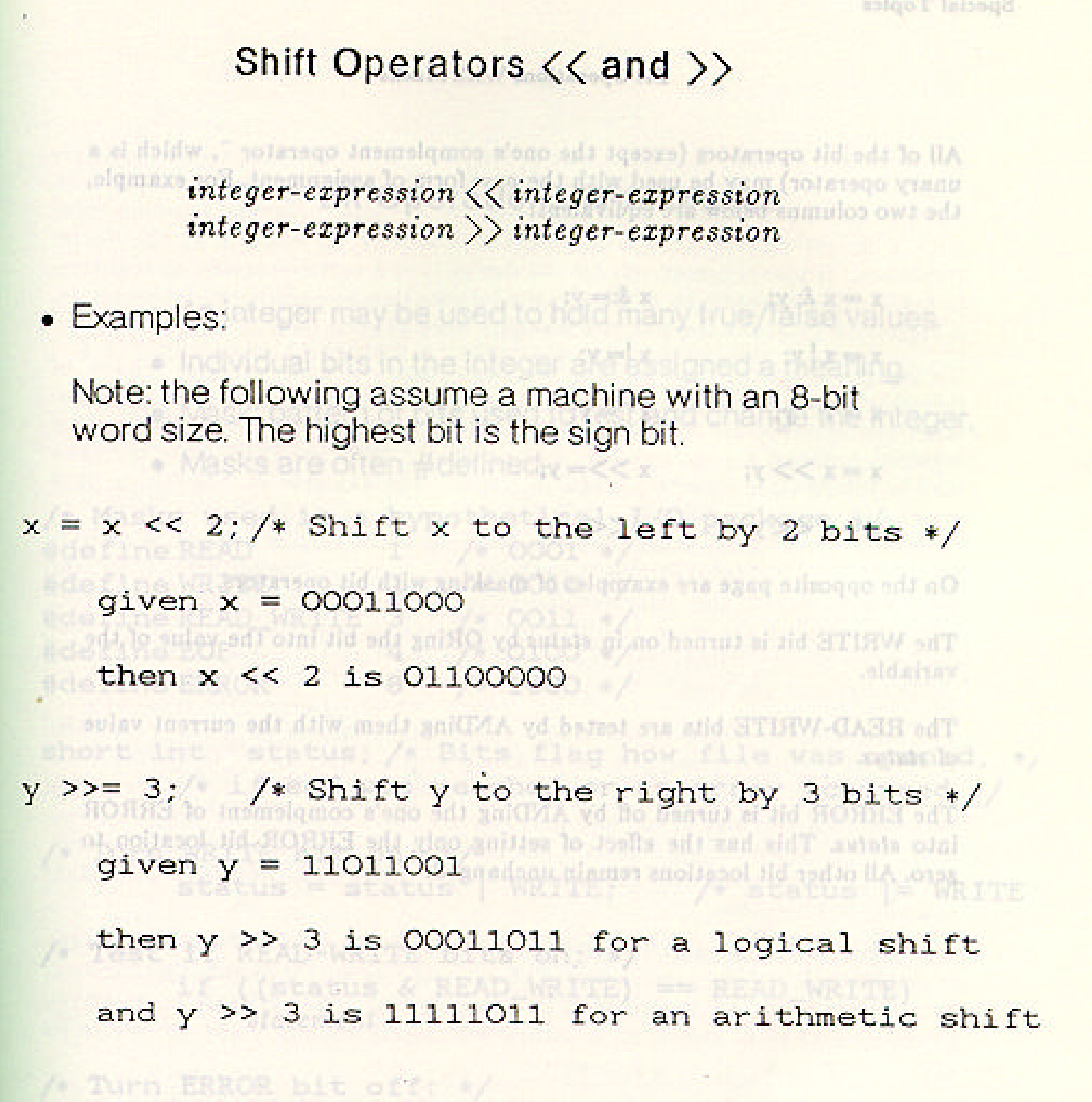


One's Complement Operator ~

~integer-expression

- The ~ is a unary operator
- 0's and 1's are reversed
- Example:

```
int  num1 = 5;

/* Assume 32-bit word */

 num1   00000000000000000000000000000101

~num1   11111111111111111111111111111010
```



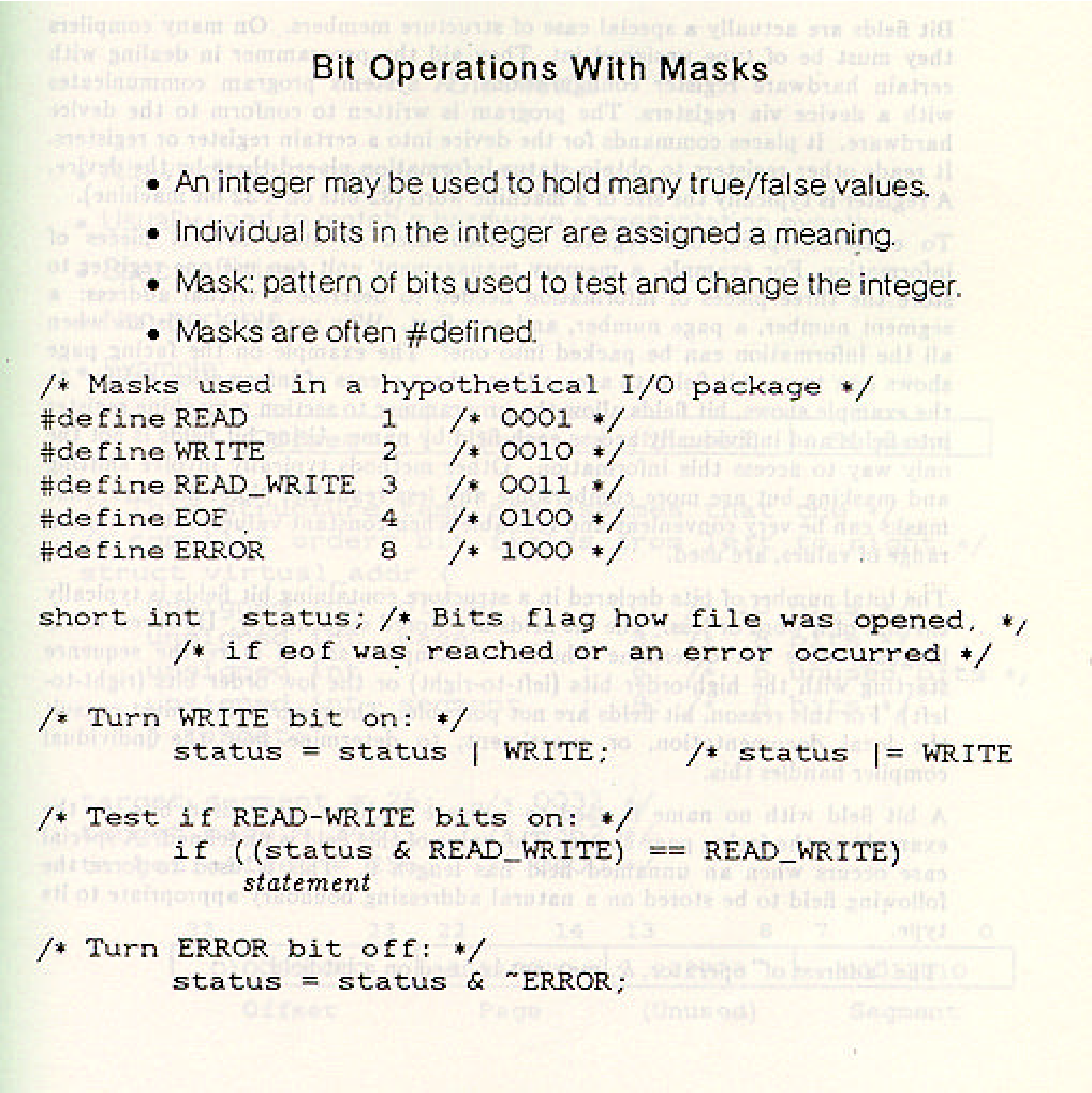
Shift Operators << and >>

integer-expression << *integer-expression*
integer-expression >> *integer-expression*

- Examples:

Note: the following assume a machine with an 8-bit word size. The highest bit is the sign bit.

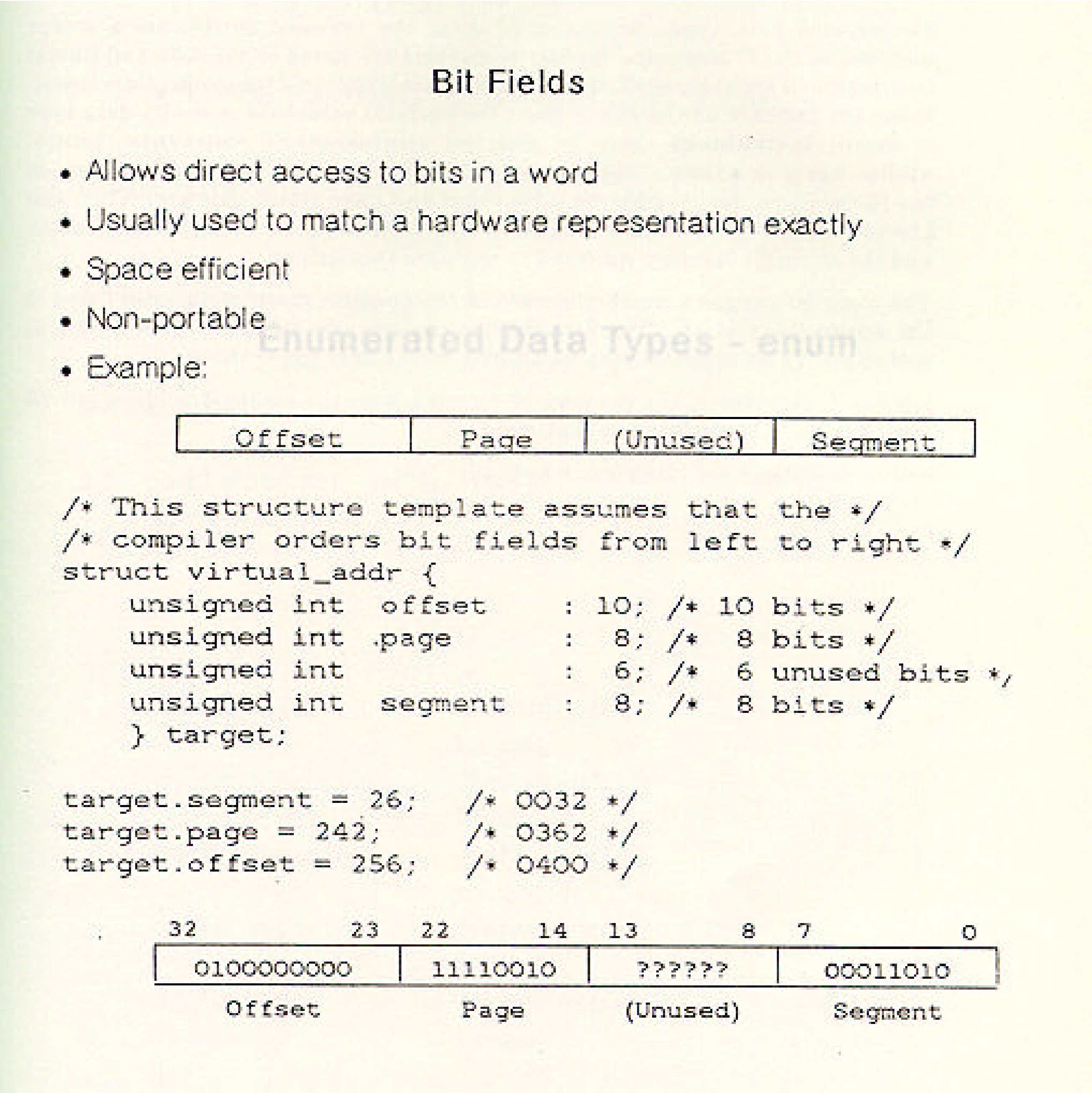
```
x = x << 2;/* Shift x to the left by 2 bits */

   given x = 00011000

   then x << 2 is 01100000


y >>= 3;   /* Shift y to the right by 3 bits */

   given y = 11011001

   then y >> 3 is 00011011 for a logical shift

   and y >> 3 is 11111011 for an arithmetic shift
```



Bit Operations With Masks

- An integer may be used to hold many true/false values.
- Individual bits in the integer are assigned a meaning.
- Mask: pattern of bits used to test and change the integer.
- Masks are often #defined.

```
/* Masks used in a hypothetical I/O package */
#define READ        1    /* 0001 */
#define WRITE       2    /* 0010 */
#define READ_WRITE  3    /* 0011 */
#define EOF         4    /* 0100 */
#define ERROR       8    /* 1000 */

short int  status; /* Bits flag how file was opened, */
       /* if eof was reached or an error occurred */

/* Turn WRITE bit on: */
       status = status | WRITE;     /* status |= WRITE

/* Test if READ-WRITE bits on: */
       if ((status & READ_WRITE) == READ_WRITE)
          statement

/* Turn ERROR bit off: */
       status = status & ~ERROR;
```



Bit Fields

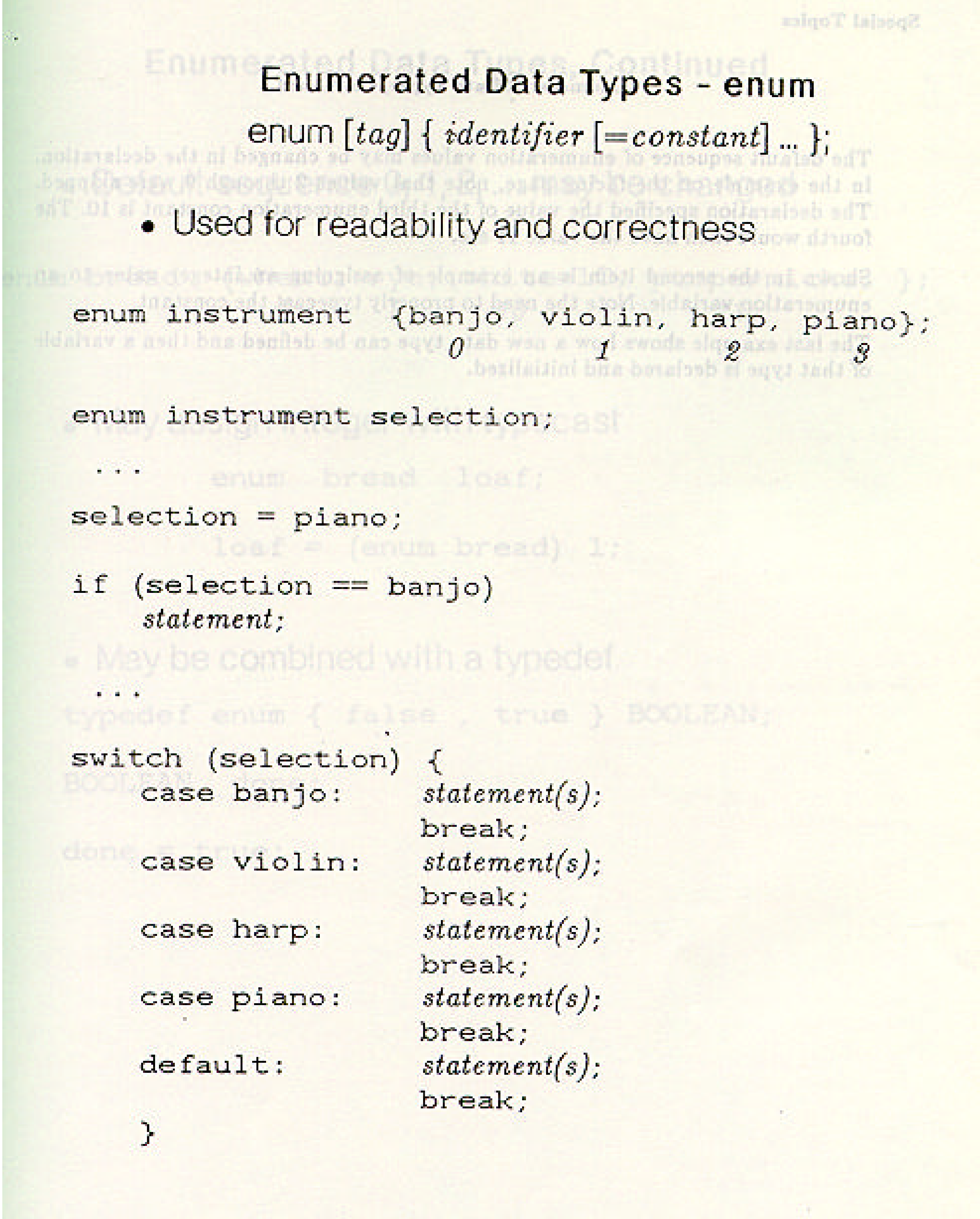
- Allows direct access to bits in a word
- Usually used to match a hardware representation exactly
- Space efficient
- Non-portable
- Example:

Offset	Page	(Unused)	Segment

```
/* This structure template assumes that the */
/* compiler orders bit fields from left to right */
struct virtual_addr {
    unsigned int  offset    : 10; /* 10 bits */
    unsigned int  page      :  8; /*  8 bits */
    unsigned int            :  6; /*  6 unused bits */
    unsigned int  segment   :  8; /*  8 bits */
    } target;

target.segment = 26;   /* 0032 */
target.page = 242;     /* 0362 */
target.offset = 256;   /* 0400 */
```

32 23	22 14	13 8	7 0
0100000000	11110010	??????	00011010
Offset	Page	(Unused)	Segment

Enumerated Data Types - enum

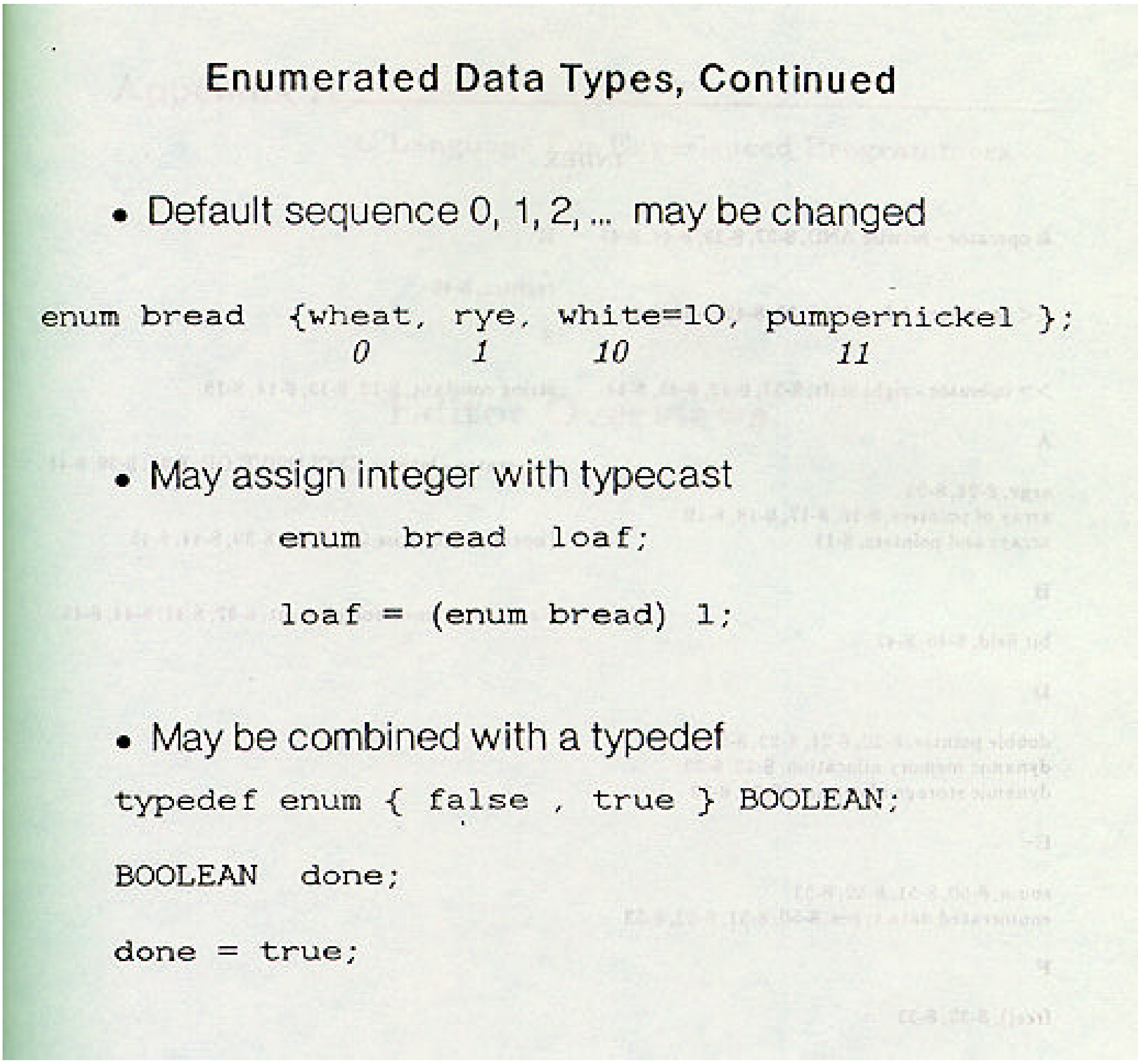


Enumerated Data Types - enum

enum [*tag*] { *identifier* [=*constant*] ... };

- Used for readability and correctness

```
enum instrument  {banjo, violin, harp, piano};
                      0       1       2      3

enum instrument selection;
 ...
selection = piano;

if (selection == banjo)
    statement;
 ...
switch (selection) {
    case banjo:      statement(s);
                     break;
    case violin:     statement(s);
                     break;
    case harp:       statement(s);
                     break;
    case piano:      statement(s);
                     break;
    default:         statement(s);
                     break;
    }
```



Enumerated Data Types, Continued

- Default sequence 0, 1, 2, ... may be changed

```
enum bread  {wheat, rye, white=10, pumpernickel };
                0      1     10            11
```

- May assign integer with typecast

```
        enum  bread  loaf;

        loaf = (enum bread) 1;
```

- May be combined with a typedef

```
typedef enum { false , true } BOOLEAN;

BOOLEAN  done;

done = true;
```