Bit-Wise Operations
Type Qualifier `const`

- Type qualifiers restrict or qualify the way an identifier of a given type can be used
  - *Typically* come after the storage class, if any, but before the type
- `static const int x = 5;`
  - `x` is a constant int with static storage class
  - `x` can be initialized, but thereafter `x` cannot be modified
const Examples

const int n = 3;
int a[x];

• Error

const int x = 8;
int *p = &x;

• Error

const int x = 8;
const int *p = &x;

• OK, p is a pointer to a constant int
const Examples (contd.)

```c
int x;
int * const p = &x;
    • p is a constant pointer to int (p itself is a constant)
const int x = 8;
const int * const p = &x;
    • p is a constant pointer to a constant int
    • Neither p nor *p can be modified
```
Type Qualifier \texttt{volatile}

- A volatile object is one that can be modified by external processes or hardware other than the program.
  - For example, a variable that stores the value of a timer chip (either because it is located at the address of the hardware device or because it is updated on interrupts) needs to be volatile to be useful.
- Prevents compiler optimizations.
  - \texttt{x} might represent a location that can be changed by other elements of the computer system.

```c
extern int x;
void foo(void) {
    x = 3;
    while ( x != 5 ) continue;
}
```

```c
extern \texttt{volatile} int x;
void foo(void) {
    x = 3;
    while ( x != 5 ) continue;
}
```

```c
extern int x;
void foo(void) {
    x = 3;
    while ( 1 ) continue;
}
/* (x != 5) is always true */
```
Bitwise Logical Operators and Shift Operators

- Logical operators
  - Complement (~), and (&), or (|), exclusive or (^)
- Left and right shift
  - expr1 << expr2
  - expr1 >> expr2

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a &amp; b</td>
<td>a ^ b</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Computer Principles
#include <limits.h>

void bitPrint(int x)
{
    int i;
    int n = sizeof(int) * CHAR_BIT;
    /* CHAR_BIT: # of bits in char */
    int mask = 1 << (n – 1);
    for (i = 1; i <= n; i++) {
        putchar((( x & mask ) == 0 ) ? '0' : '1');
        x <<= 1;
        if (i % CHAR_BIT == 0 && i < n)
            putchar(' ');
    }
    putchar(' ');
}
Packing and Unpacking

```c
int pack(char a, char b,
         char c, char d)
{
    int p = a;
    p = (p << CHAR_BIT) | b;
    p = (p << CHAR_BIT) | c;
    p = (p << CHAR_BIT) | d;
    return p;
}

char unpack(int p, int k)
{
    int n = k * CHAR_BIT;
    unsigned mask = 0xff;
    mask <<= n;
    return ((p & mask) >> n);
}
```
C Preprocessor (CPP)

- Lines that begin with a # are called preprocessing directives
  - Communicate with the preprocessor
  - CPP does not know C
- #include <stdio.h>
- #include "filename"
#define

- `#define identifier token_stringopt`
- `#define identifier (identifier, …, identifier) token_stringopt`

Examples

- `#define PI 3.14159`
- `#define EPS 1.0e-9`
- `#define FOO(x) ((x) * (x))`
  - `FOO(i+1)` expands to `((i+1) * (i+1))`
- `#define FOO(x) x * x`
  - `FOO(i+1)` expands to `i+1 * i+1`
- `#define FOO(x) (x) * (x)`
  - `5 / FOO(k)` expands to `5 / (k) * (k)`
- `#define MIN(x, y) (((<(y)) ? (x) : (y))`
Conditional Compilation

- `#if constant_integral_expression`
- `#ifdef identifier`
- `#ifndef identifier`
  - Conditional compilation of the code that follows until `#endif` is reached
- `#undef identifier`
  - Undefine the macro defined by `#define`
- `defined identifier = defined( identifier )`
  - Operator
  - 1 if identifier is currently defined, 0 otherwise
- `#if, #ifdef, #ifndef`
- `#elif constant_integral_expression`
- `#else`
- `#endif`
#define DEBUG 1
...
#if DEBUG
    printf("debug msg\n");
#endif
==========================
#define DEBUG
...
#ifdef DEBUG
    printf("debug msg\n");
#endif

To prevent the clash of macro names, use #undef

#include “pi.h”
#undef PI
#define PI 3.14
Predefined Macros

- **__DATE__**: a string containing the current date
- **__FILE__**: a string containing the file name
- **__LINE__**: an integer representing the current line number
- **__STDC__**: if the implementation follows ANSI C, then the value is a nonzero integer
- **__TIME__**: a string containing the current time
Operator #

- # causes stringization of a formal parameter in a macro definition
- The argument to be surrounded by double quotes

```c
#define foo(x, y) \
    printf(#a " and " #b "\n")

int main(void)
{
    foo( SNU, KOREA );
    return 0;
}
```
Operator `##`

- Merges tokens

```c
#define foo(i) x ## i

... foo(1) = foo(2) = foo(3);
...

/* x1 = x2 = x3; */
```
#include <stdio.h>
#include <stdlib.h>
#if defined(NO_DEBUG)
    #define assert(x) ((void) 0)
#else
    #define assert(x)     
        if (!(x)) {      
            printf("\n%s%s\n%s%s\n%s%d\n",  
                "assertion failed: ", #x,  
                "in file ", __FILE__,  
                "at line ", __LINE__);  
            abort();
        }
#endif

File and File Pointers

- A file is a stream of characters or bits
- The identifier FILE defined in stdio.h is a structure that describes the current state of a file
- Three file pointers defined in stdio.h
  - stdin : standard input file (keyboard)
  - stdout : standard output file (screen)
  - stderr : standard error file (screen)
- File versions of printf and scanf
  - fprintf( file_ptr, control_string, other_arguments );
  - fprintf( stdout, ... ) \equiv \text{printf}( ... )
  - fscanf( file_ptr, control_string, other_arguments );
  - fscanf( stdin, ... ) \equiv \text{scanf}( ... )
- fputc, fgetc, ...
String versions of printf and scanf

```c
sprintf( char *s, const char *format, ... );
sscanf( const char *s, const char *format, ... );
```

```c
char s1[] = "23 45 67 foo";
char s2[80];
char tmp[100];
int x, y, z;
sscanf(s1,"%d%d%d%s", &x, &y, &z, tmp);
strftime(s2,"%d %d %d %s %s\n", x, y, z, tmp, tmp);
printf("%s", s2);
```
fopen() and fclose()

- For reading, writing, and appending to a file

```c
#include <stdio.h>
int main(void)
{
    int a, sum = 0;
    FILE *ifp, *ofp;

    ifp = fopen("infile", "r");
    ofp = fopen("outfile", "w");
    ...
    fclose(ifp);
    fclose(ofp);
}
```

"r" open text file for reading
"w" open text file for writing
"a" open text file for appending
"rb" open binary file for reading
"wb" open binary file for writing
"ab" open binary file for appending
"r+" open binary file for reading and writing
"w+" open text file for writing and reading
...
An Example

```c
#include <stdio.h>
#include <stdlib.h>
void foo(FILE*, FILE*);
void print_msg(char *);
int main(int argc, char **argv) {
    FILE *ifp, *ofp;
    if (argc != 3) {
        print_msg(argv[0]);
        exit(1);
    }
    ifp = fopen(argv[1], "r");
    ofp = fopen(argv[2], "w");
    foo(ifp, ofp);
    fclose(ifp);
    fclose(ofp);
    return 0;
}

void print_msg(char *s)
{
    printf("\n%s%s%s%s\n", "usage: ", s, " infile", " outfile");
}
```
Exercises

- Write a program that performs n circular shifts of an int
  - Left-shift the int by n positions
  - The high-order bits are reintroduced as the low-order bits
- Write a program that reverses the bit representation of an int
- Write a program that reverses a given string
  - Non-recursive vs. recursive
- Write a program that reverses the entire text in a text file