Chapter 3
Processing and Interactive Input

**Assignment**

- **Storing a value in a variable**
  - Variable = Expression
    - `i = j = k = 0;` // equivalent to `i = (j = (k = 0))`
  - Expression
    - Constant: `10, 3.14, 'A', 'A', 'hello', 0x10FA, 012`
    - Variable: `x, count, sum`
    - Function: `sin(x)`
    - Arithmetic: `10 + size, x - y, 2 + sqrt(2)`
  - Operation assignment: `+=, -=, *=, /=, %=, ^=, <<=, >>=`
    - `variable op= expression;` // variable = variable op expression
    - `x += 10;` // `x = x + 10`
    - `i = 2;` // `i = i - 2;`
    - `a = b;` // `a = a / b;`
    - `index %= size;` // `index = index % size;`
    - `mask <<= 2;` // `mask = mask << 2;`

**Type Conversions**

- **Implicit and Explicit**
  - chars and ints convert back and forth
  - Small types automatically "promoted" to large types in arithmetic expressions
  - If `a` and `b` are ints, `a/b` is an int (truncates if necessary)
  - Explicit conversions are called type casts: `(type)expression`
    - Required to convert from a large type to a small type and to force conversions for /
      - `int i = (int)(3.14159); // always truncates`
      - `int i = (int)(3.14159 + 2.7);`
      - `unsigned int i = (unsigned int)75;`
      - `double x = (double)1 / 3;`

**Operator Examples**

- **Unusual operators**
  - Auto increment and decrement
    - `Target must be a variable`
    - `int i, j = 10;`
      - `i = j++;` // `i` is 10, `j` is 11
      - `i = ++j;` // `i` is 11, `j` is 12
      - `i = j--;` // `i` is 10, `j` is 9
      - `i = --j;` // `i` is 9, `j` is 9
  - `++`, `--`, `i++`, `i--` are legal (i.e., assignment is not required)
  - May be embedded in expressions
  - Often used in array indexes

**Math Library**

- **General functions**
  - Most parameters and return types are type double
    - `sqrt(x)`
    - `pow(x,y)`
    - `exp(x)`
    - `exp2(x)`
    - `log(x)`
    - `log10(x)`
    - `fabs(x)`
    - `abs(x)`
  - `x` and return value are type int

- **More general functions**
  - `floor(x)`
    - largest integral value not greater than `x`
  - `ceil(x)`
    - smallest integral value not less than `x`
  - `fmin(x, y)`
    - smallest of `x` and `y`
  - `fmax` (largest of `x` and `y`)
  - `round`
    - round to nearest integer, away from zero
      - returns a double value
  - `remainder(x,y)` (floating point remainder)
    - return `x - n*y`, where `n` is the value `x / y`, rounded to the nearest integer. If this quotient is ½, it is rounded to the nearest even number.
Math Library

Trigonometric functions

- All angular measures are in radians
- \( \sin(x) \)
- \( \cos(x) \)
- \( \tan(x) \)
- \( \sin(x) \)
- \( \cos(x) \)
- \( \tan(x) \)

Math Library

Hyperbolic functions

- \( \sinh(x) \)
- \( \cosh(x) \)
- \( \tanh(x) \)
- \( \sinh(x) \)
- \( \cosh(x) \)
- \( \tanh(x) \)

Formatted Printing with \texttt{printf}

Standard library output function

\begin{verbatim}
#include <stdio.h>  // (prototype and constants, etc.)

printf("format string", arg1, arg2, arg3 ...);
- The format string is required; other arguments are optional
- Arguments can be constants, variables, expressions, etc.
- Format specified (e.g., \texttt{%-d}) is placeholder - replaced by value
- \texttt{%-d or %-d (int, short, long)}; \texttt{%-d (long)}
- \texttt{%-d (string)}
- Excess arguments (no corresponding format specifier) are ignored. Control characters without corresponding arguments produce undefined results

printf("The answer is: \%d and \%dn", ans1, ans2);
\end{verbatim}

The Control String

Output specification

- Contains
  - Characters that are printed without modification
  - Format specifiers: minimally a percent sign (\%) & a conversion specifier

- Format specifiers
  - \texttt{%-position flag \%fieldWidth \%precision \%length}
  - \texttt{position flag fieldWidth precision length conversion}
    - \texttt{position flag}
      - \texttt{- (dash, hyphen, or minus sign) left justify output (used w/ field width)}
      - \texttt{0} (zero) pad with 0's (default is pad with blanks)
      - \texttt{+} precede numbers with + or -
      - \texttt{(blank or space) precede numbers with space or -}

Simple \texttt{printf} Examples

Common usage

\begin{verbatim}
printf("\%d \%d \%ld", 10, counter, 10+counter);
\end{verbatim}

\begin{verbatim}
printf("i*\pi = \%f \sin(i*\pi) = \%15.10f\n", i*\pi, sin(i*\pi));
\end{verbatim}

\begin{verbatim}
printf("A = \%d in ASCII\n", 'A');
\end{verbatim}

\begin{verbatim}
printf("Can't open file \"%s\\n", filename);
\end{verbatim}

\begin{verbatim}
printf("Test Score: \%f\%\n", tscore);
\end{verbatim}

\begin{verbatim}
printf("The Earth weighs \%e tons", 6.65e24);
\end{verbatim}
**Data Input with scanf**

"Standard" library input function

- Used to read data from the keyboard
- `#include <stdio.h>` (prototype, constants, etc.)
- Excess arguments (no corresponding control character) are ignored. Control characters without corresponding arguments produce undefined results
- Arguments **must** be pointers (i.e., addresses); these are usually found with the address of operator (&)

```c
scanf("%d", &size);
scanf("%f", &balance);
scanf("%lf", &debt); /* must use 1 for double */
scanf("%tc", &answer);
```

**#include and Header Files**

Merging files

- The contents of #included files are read into and merged with the contents of the file in which the directive appears
- Header files end with .h and contain declarations
- `#include <fileName>`
  - Searches for a file in the standard directory for include files
  - Unix: /usr/include
- `#include "fileName"
  - Searches for a file in the current directory
- `#includes` can be nested to any depth

**#define**

Symbolic constants

- Textual replacement from the point of definition & beyond
- Mnemonic for "magic" numbers and strings
- Consistency (one place for change)
- Examples (note, they are not terminated with a semicolon)
  - `#define LINELEN 256`
  - `#define LONGLINE 2 * LINELEN`
  - `#define FALSE 0`
  - `#define TRUE 1`
  - `#define OMODE "rt"
  - `#define E_MESS "Unable to open file for reading"
  - `#define u/long unsigned long`

**#define**

Parameterized macros

- Textual replacement from the point of definition & beyond
  - Fast: inline expansion avoids function call and return
  - Works simultaneously with multiple data types
- `#define MAX(x, y) (((x) > (y)) ? (x) : (y))`
  - Any number of parameters (arguments)
  - No spaces in parameter list
  - `MAX(a+b, c-10)` → (((a+b) > (c-10)) ? (a+b) : (c-10))
- Problems
  - Can't take the address of a macro (can't be a function argument)
  - May cause code explosion if macro is too big
  - Subject to nasty side effects: MAX(a++, b++) → (((a++) > (b++)) ? (a++) : (b++))

**The Preprocessor In Action**

Substitutes and copies text

- `#define SIZE 100` other stuff
- progh.h
- `#include <stdio.h>`
- `#include "prog.h"`
- using namespace std;
- void main()
  - { printf("%d", SIZE); }
- prog.c
- stdio.h
- void main()
  - { printf("%d", 150); }
- temp file
- compiler
- other stuff
Using Assertions

- #include <assert.h>
- assert(expression)
  ▶ if expression is true, execution continues
  ▶ otherwise, prints assertion failed with file name & line number & exits
  ▶ good for program development but not for final error checking
- Turn off assertions (but leave code in place)
  ▶ #define NDEBUG before #include
  ▶ use the compiler option -DNDEBUG
  /* #define NDEBUG */ /* uncomment to turn off assertions */
- #include <assert.h>
- assert(count < 10);
...

Enumerations

- Symbolic constants
  ▶ An enumeration is a set of named integers
  ▶ Provide mnemonic names for “magic” numbers
  ▶ Passing and returning encoded values
  ▶ Creating bit values, which may be or’ed together
- enum tag { enumeration_list } variable_list;
  ▶ tag is optional and follows the rules for creating a legal identifier
  ▶ variable_list is optional and often omitted; if present, it defines
    variables of type enum tag
  ▶ enumeration_list is a comma separated list if legal identifiers
  - the first identifier is assigned the value 0
  - the value of one or more identifiers may be specified
  - unless specified, the value of an identifier is always 1 greater than the last

enum Examples

"Magic numbers" and bit-patterns

```c
enum commands {
  e_Unknown, /* commands typed at keyboard */
  e_quit, /* unknown command entered */
  e_search, /* close files, quite database */
  e_input, /* search for a person */
  e_import, /* input a new person */
  e_dump, /* import data from another file */
  e_help, /* dump data base in ascii/text */
};
```

```c
enum { uread = 1, uwrite = 2, uexe = 4, gread = 8, gwrite = 16,
  gexe = 32, oread = 64, owrite = 128, oexe = 256
};
```

typedef Statements

Creating type aliases

- A typedef statement creates an alias or new name for an
  existing type (i.e., does not create a new type)
  ▶ A “shorthand” notation
  ▶ More mnemonic or self documenting
  ▶ Improve portability by creating machine-dependent types (see
    <sys/types.h>)
- General format: typedef old_name new_name;
  ▶ typedef unsigned long ulong;
  ▶ typedef int boolean;
  ▶ typedef unsigned int size_t;
  ▶ typedef char * string;