Chapter 6 Types and Conversions

6.1 Type Definitions

```
typedef int whole_num;
whole_num count;    /* int count; */


typedef float real;
real inches;
```

Usually `typedef` is used in more complicated data type, such as struct.

6.2 Type `size_t` and the `sizeof` Operator

- A C implementation defines a number of `typedef` names in header files.
- Each `typedef` name corresponds to a particular purpose that a type can serve. For example, `size_t` is an unsigned integer type used to represent the number of bytes in a region of memory. Depending on the computer hardware, type `unsigned` or `unsigned long` might reasonably be used for this job.
- The `sizeof` operator can be applied directly to a type rather than an expression, but the type must be enclosed in parentheses. `sizeof` returns the number of bytes used by the type.
- For example, `sizeof(unsigned long)` yields the size of an `unsigned long` value.

6.3 Enumerated Types

- An enumeration specifies an integer type and defines constants to represent values of the type.
- The constants are called enumeration constants and (like character constants) have type `int`. We can use an enumeration to declare variables with the enumerated type. We can also test the values of the variables with expressions.

For example, the following statements

```
enum workday {MON=1, TUE, WED, THUR, FRI};
int day;

for (day=MON; day <= FRI; day++)
  printf("%d ",day);
```

produce the printout `1  2  3  4  5`

6.4 Type Casts

- Type casts allow us to request conversions explicitly wherever they may be needed.
- A type cast is an operator formed by enclosing a type designation in parentheses, as `(int)`, `(float)`, and `(unsigned long)`.

For example,

```
float x;
int n;

n = (int)(x + 0.5);
```

assign `n` the value of `x` rounded to the nearest integer.

```
long n;

n = (long)(32760 + 8);
```

6.5 Format Strings for `scanf()`

- For `scanf()`, a conversion specification gives the format of a value to be read; the corresponding argument gives the address of the variable in which the value is to be stored.

  Example: `scanf("%d",&a);`

- Literal characters in a `scanf()` format string are matched with corresponding characters in the input. Input characters that match literal characters are read and discarded.

  Example: `scanf("%f",&a);`

  Input: ~~12.34~~ 93

- If a literal character (other than a space) fails to match the corresponding input character, `scanf()` returns immediately without reading any more input or storing any more values.

  Example, `scanf("***$%lf",&x);`

  requires the input to begin with the characters ***$. Thus ***$25.99 is acceptable but **$25.99 will cause `scanf()` to return before the number is read and stored.

  Example, `scanf("%*d%lf",&x);`

  reads and discards an `int` value, then reads a double value into x. So, if the input data is

  ```
  50 2.99
  ```

  x will have a value of `2.99`.

Chapter 7 Macros

- When a function is called, a certain amount of time is required to pass arguments to the function, transfer control to it, allocate memory for its variables, return its value, and transfer control back to the code following the function call.

- This overhead can be eliminated by replacing a function call by an inline code.

- Function macros provide a simple means of replacing certain function calls by inline code. Function macros are defined like other macros except that one or more formal arguments are listed in parentheses after the macro name.

- Formal arguments are used in the expression that defines the macro.

Example 1

Consider the following function:

```
float sumsqr (float x, float y)
{
  float w;
  w= x*x + y*y;
  return w;
}
```

The function `sumsqr` can be replaced by a macro as follows:

```
#define sumsqr(x, y)  x*x + y*y
```

The macro has two formal arguments, `x` and `y`. A macro call specifies which text (expression) will **replace** which formal argument when the call is expanded.

If we write

```
u = sumsqr(a, 2.75);
```

then `a` will replace `x`, and `2.75` will replace `y`, and the statement will expand to

```
u = a*a + 2.75*2.75;
```

i.e., x is replaced by a, and
y is replaced by 2.75

Example 2

```
#define sumsqr(x, y) ((x)*(x) + (y)*(y))
```

Now , `sumsqr(a + b, c)` expands to

```
((a + b)*(a + b) + (c)*(c))
```

and `2.0*sumsqr(a, b)` expands to

```
2.0*((a)*(a) + (b)*(b))
```

The expanded expressions yield the expected results even though they contain a few more parentheses than we would normally write.

The preprocessor recognizes a call to a function macro only if the name of the macro is followed by a left parenthesis.

```
/*  demo67.c */

#include <stdio.h>
#define cube(x) ((x) * (x) * (x))

int main (void)
{
   printf( "The cube of 3.2 is %.3f \n",
               cube(3.2));
   return 0;
}
```

Screen Output:

```
The cube of 3.2 is 32.768
```