Data Types, Variables, Assignments, and Preprocessing in C

010.133
Digital Computer Concept and Practice
Spring 2013

Lecture 09

Data Types, Variables, and Assignments

C Declarations

- Names (identifiers) in C
  - Case sensitive
  - Contain digits and underscores (_), but may not begin with a digit
- In C, ***all identifiers must be declared before they are used***
  - Type information for storage allocation by the compiler
  - Help the compiler to choose correct operations

Declarations (contd.)

- To declare a variable, specify the type of the variable, and then its name
  - The variable can have an initial value
  - Multiple variables can be declared after the type by separating them with commas

```
#include <stdio.h>

int area(int w, int h) {
    return w * h;
}

int main(void) {
    int width = 5, height;

    height = 10;
    printf("width: %d, height: %d, area: %d\n",
           width, height, area(width, height));

    return 0;
}
```

Data Type

- The name that specifies the kind of data values
  - Determines
    - The possible values in the category
    - The operations that can be performed on the values
    - The way the values can be stored
- Used by the compiler

Fundamental Types in C

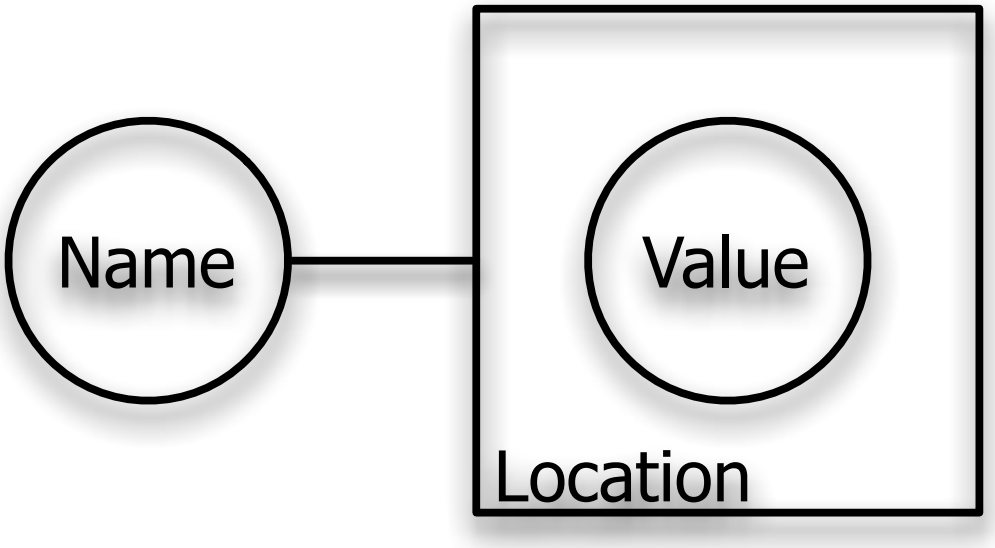
- There are several basic data types in C:
- char: a single byte holding one character
  - Depending on the compiler, either **`char` ≡ `signed char`** or **`char` ≡ `unsigned char`**
    - signed char: -128 ~ 127
    - unsigned char: 0 ~ 255
- int: an integer typically having the integer size of the host machine
- float: single-precision floating point
- double: double-precision floating point

Fundamental Types in C (contd.)

Integral types	**char**	**signed char**	**unsigned char**
	short (signed short int)	**int (signed int)**	**long (signed long int)**
	unsigned short (unsigned short int)	**unsigned (unsigned int)**	**unsigned long (unsigned long int)**
Floating types	**float**	**double**	**long double**
Arithmetic types	integral types or floating types		

C Variables

- A named storage location that contains some value
  - The name is bound to the location



C Constants

- An integer constant is a decimal integer like 75 and -492
- A floating point constant contains a decimal point like 3.14 or an exponent like 4e-1 (which means 0.4)
- A character constant is one character within single quotes like ‘a’

C Constants (contd.)

- Constants are suffixed to tell their types
- The type of an unsuffixed integer constant
  - int, long, and unsigned long
  - The system chooses the first of these types that can contain the value
  - Is 345678910123 int, long, or unsigned long?

Suffixes for integer constant		
Suffix	Type	Examples
U or u	unsigned	25U, 0x32u
L or l	long	37L, 0x27L
UL or ul	unsigned long	56ul, 234UL

C Constants (contd.)

- Any unsuffixed floating constant is of type double
- For example, 3.14159, 314.159e-2F, 0e0, 1., 314.159e-22

Suffixes for floating constant		
Suffix	Type	Examples
F or f	float	3.7F
L or l	long double	2.7L

C Constants (contd.)

- An escape sequence is used to represent a special character
- An escape sequence looks like two characters, but it represents only one
  - \n new line
  - \t horizontal tab
  - \\ backslash
  - \? question mark
  - \' single quote
  - \" double quote
- The character constant '\0' represents the character with value zero, the null character

C Constants (contd.)

- String constants
  - A sequence of characters enclosed in a pair of double quotes
    - “abc”
    - “abc\”def”
    - “abc\\def”
  - Null terminated

C Constants (contd.)

- A qualifier **`const`** can be applied to any variable declaration to specify that its value will not be changed any more
- For example, **`const float pi = 3.14;`**
  - **`pi`** cannot be assigned to a new value

Typedef

- Allows the programmer explicitly associate a type with an identifier

```
typedef  char               sval8b;
typedef  int                sval4B, sval32b;
typedef  unsigned long size_t;

sval8b  v;
sval4B  x, y;
```

C Sizeof Operator

- `sizeof (` *type* `)`
- To calculate the size of any data type
  - Returns the number of bytes needed to store a data item in *type*
- Unary operator

```
sizeof (int)
sizeof (char)
sizeof (float)
sizeof(double)
sizeof(unsigned int)
sizeof(long)
```

C Arithmetic Operators

- + Addition
- - Subtraction
- * Multiplication
- / Division
  - If both of its operands are integers, / gives the quotient
    - The value of 9/4 is 2
- % Remainder
  - It requires integer operands and gives the remainder
  - The value of 9%4 is 1

True and False in C

- 0 evaluates to false and any other number to true

C Relational Operators

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equal to
- != not equal to
- Returns either 0 (false) or 1 (true)
- For example,
  - The value of 7 < 9 is 1
  - The value of 7 > 9 is 0

C Logical Operators

- ! Logical NOT
- && Logical AND
- || Logical OR
- Returns either 1 or 0

Operator Precedence

- The unary ! operator has the same precedence as the unary + and – operators, and they have higher precedence than any binary operators
- Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the result is determined
- For example,
  - i > 0 && n / i == 10 means (i > 0) && ((n / i) == 10)
- Short circuit evaluation
  - In (i > 0 && n / i == 10), if i>0 is false, the rest needs not to be evaluated

Highest	* / %
	+ -
	< > <= >=
	== !=
	&&
Lowest	||

Type Conversion in C

- Type conversions can occur across an assignment or when the operands of a binary operator are evaluated
  - To choose a proper machine instruction to perform the operation
- An expression (such as x + y) has both a value and a type
- To perform a binary operation, its operands must be of the same type
  - When two operands are of different types, generally the more restricted operand is converted to the more general one

Type Conversion (contd.)

- Examples
  - `int i; float f;`
    - In i + f, the operand i gets promoted to a float and the expression i + f as a whole has type float
  - `double d; int i;`
    - In d = i, i is converted to double and assigned to d
- Other names: automatic conversion, implicit conversion, coercion, promotion, widening

Type Conversion (contd.)

```
char c;  short s;  int i;  long l;
unsigned u; unsigned long ul;
float f;  double d;  long double ld;

c-s/i (int)
u*3.0-i (double)
c+4 (int)
c+3.0 (double)
d+s (double)
7*i/l (long)
u*4-i (unsigned)
f*8-i (float)
8*s*ul (unsigned long)
ld + c (long double)
u-ul (unsigned long)
u-l (system dependent)
```

Type Casts

- Explicit type conversion by the programmer
- Examples
  - `(double) i`
  - `(long) ('A' + 1.0)`
  - `(double) (x = 22)`
  - `(float) i + 3`
  - `((float) i) + 3`
  - `d = (double) i / 5`

C Assignment Operation

- Denoted by =
- Assigns the value of the right-hand side operand to the storage location bound to the left-hand side operand
- For example, x = y stores the value stored in the storage location bound to y to the storage location bound to x
- A C statement ends with ;
  - For example, x = y; is an assignment statement

L-values and R-values

- L-value
  - The value that can appear in the left side of an assignment operation (address)
- R-value
  - The value that can appear in the right side of an assignment operation (value)
- A variable typically refers to the stored value
  - However, if a variable is the left operand of an assignment operation, it refers to the storage location
  - A variable's l-value is its address and r-value is the value stored in it

Block Statement

- Also known as a compound statement
  - It is a statement, but does not need to end with ;
- A group of any number of declarations and statements
  - Surrounded by a pair of curly braces
  - Treated as a single statement
- Can be nested
- A variable declared within a block has block scope
  - An identifier is active and accessible from its declaration point to the end of the block
  - If an identifier is redeclared in an inner block, the original declaration is hidden by the redeclaration in the inner block from the redeclaration point to the end of the inner block

```c
#include <stdio.h>

int area(int w, int h) {
    return w * h;
}

int main(void) {
    int width = 5, height;

    height = 10;
    printf("width: %d, height: %d, area: %d\n",
           width, height, area(width, height));

    return 0;
}
```

```c
#include <stdio.h>

int area(int w, int h) {
    return w * h;
}

int main(void)
{
    int width = 5;
    {
        int width = 6, height = 10;
        printf("width: %d, height: %d, area: %d\n",
               width, height, area(width, height));
    }

    return 0;
}
```

Increment and Decrement Operators

- C has special operators for increment (add by 1) and decrement (subtract by 1), ++ and --
- They can be used as prefix operators like ++i or postfix operators like i++
- ++i increment i before its value is used, while i++ increments after its value is used
  - Assume i has the value 7
    - n = ++i; sets n to 8
    - n = i++; sets n to 7
    - In both cases, i becomes 8
- ++ and -- have higher precedence than any binary operators

printf

- The programmer need to declare the function that is used in his/her program
- Function prototypes
  - A way of declaring functions
    - int foo(int a);
    - float bar(void);
  - Printf's prototype is included in stdio.h
- Printf is passed a list of arguments: a format string and other arguments
  - The format string contains both ordinary characters and conversion specifications (format specifiers)

printf (contd.)

- `printf("abcde");`
- `printf("%s", "abcde");`
- `printf("%c%c%c%c%c",'a','b','c','d','e');`
- `printf("%c%4c%4c", 'a', 'b', 'c');`

c	Character
d	Decimal integer
e	Floating-point number in scientific notation
f	Floating-point number
g	In the e-format or f-format, whichever is shorter
s	string

printf (contd.)

```
#include <stdio.h>

int main(void)
{
    int n = 251;
    float pi = 3.14;
    char a = 'h';

    printf("[%d]\n", n);
    printf("[%6d]\n", n);
    printf("[%-6d]\n", n);
    printf("[%f]\n", pi);
    printf("[%10.4f]\n", pi);
    printf("[%-10.4f]\n", pi);
    printf("%c\n", a);
    return 0;
}
```

```
[251]
[   251]
[251   ]
[3.140000]
[    3.1400]
[3.1400    ]
h
```

scanf

- Analogous to printf, but used for input
- scanf is passed a list of arguments: a format string and other arguments
  - Other arguments
    - Addresses
  - **`scanf("%d", &x);`**
    - The format specifier %d is matched with &x
    - Treat characters in the input stream as a decimal integer and store it at the address of x

scanf (contd.)

- When reading numbers, it skips white space (blanks, newlines, and tabs), but when reading in a character, white space is not skipped

c	Character
d	Decimal integer
f	Floating-point number (float)
LF or lf	Floating-point number (double)
s	string

scanf (contd.)

```
#include <stdio.h>

int main(void)
{
  int n;
  char a;
  float x;
  double y;

  scanf("%d %c %f %lf", &n, &a, &x, &y);
  printf("%d %c %f %f\n", n, a, x, y);
  return 0;
}
```

```
25 g 3.14 -4e-1
```

```
25 g 3.140000 -0.400000
```

Exercises

- Use the following code to print out a list of powers of 2 in decimal, hexadecimal, and octal

```
int i, val = 1;

for(i = 1; i < 35; i = i + 1) {
    printf("%15d%15u%15x%15o\n", val, val, val, val);
    val = val * 2;
}
```

Exercises (contd.)

- Compile and run the following program:

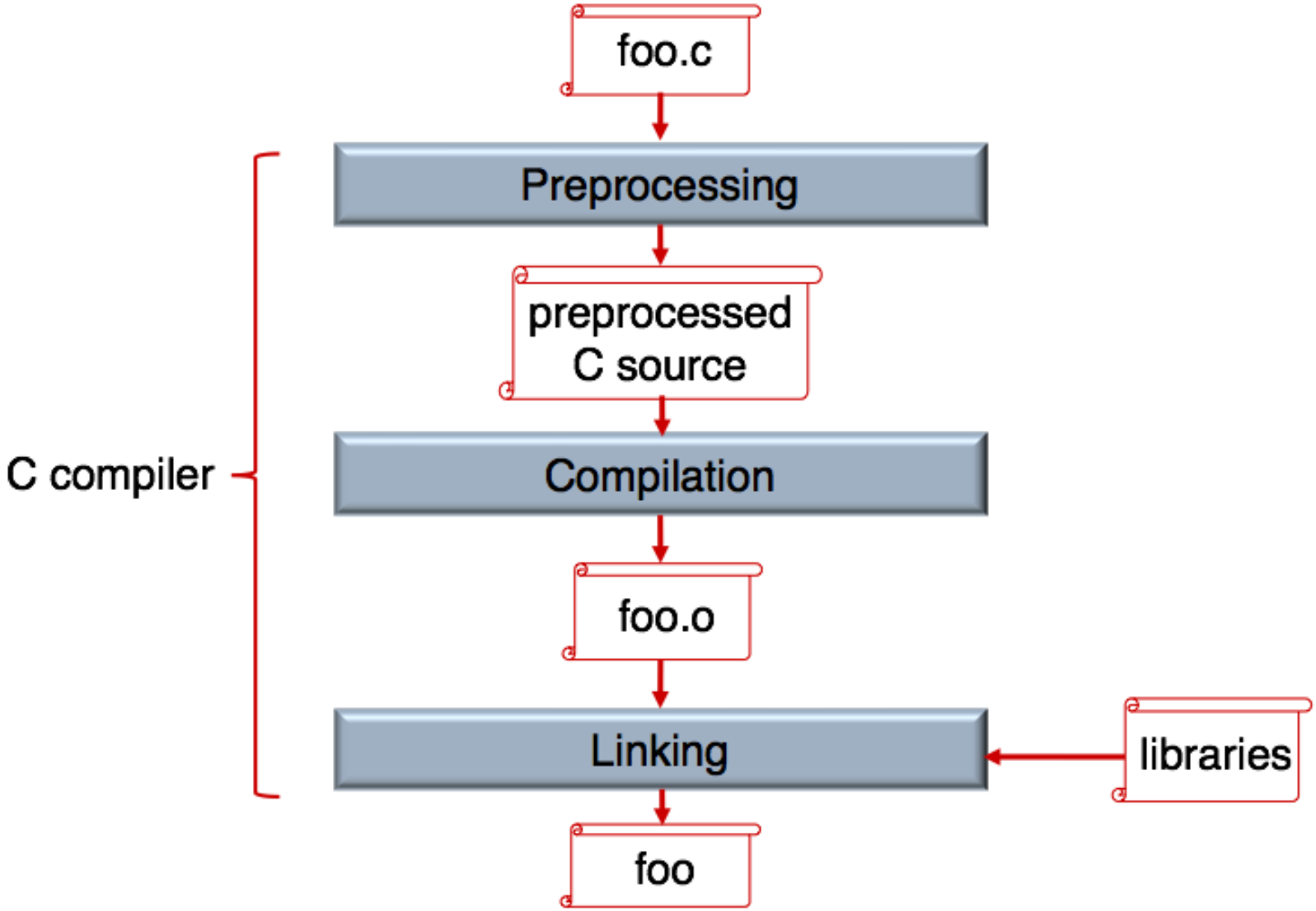
```
#include <math.h>
#include <stdio.h>
int main(void)
{
  double two_pi = 2.0 * M_PI;   /* in math.h */
  double h = 0.1;
  double x;
  for(x = 0.0; x < two_pi; x = x + h)
    printf("%5.1f: %.15e\n", x,
           sin(x) * sin(x) + cos(x) *cos(x));
  return 0;
}
```

Exercises (contd.)

- Write a program that prints a table of trigonometric values for sin(), cos(), and tan()
  - The angles in your table go from 0 to 2π in 20 steps

Preprocessing

Typical Compilation Phases



Preprocessing

- C preprocessor (cpp) is the preprocessor for the C programming language
  - Typically, it is a separate program invoked by the compiler as the first part of translation
- The preprocessor handles,
  - File inclusion (#include)
  - Macros (#define)
  - Conditional compilation (#if)
- Cpp does not know C
  - Cpp can also be used independently to process other types of files

File Inclusion

- `#include < file_name >`
- `#include "file_name"`
- The preprocessor replaces the line `#include <stdio.h>` with the system header file of that name
  - The entire text of the file `stdio.h` replaces the `#include` directive

```
#include <stdio.h>

int main (void)
{
    printf("Hello, world!\n");
    return 0;
}
```

File Inclusion (contd.)

- If the filename is enclosed within angle brackets, the file is searched for in the standard compiler include paths
- If the filename is enclosed within double quotes, the search path is expanded to include the current source directory
- C compilers and programming environments allow the programmer to define where include files can be found
- By convention, include files are given a .h extension, and files not included by others are given a .c extension
  - Not required, though

Macros

- There are two types of macros, object-like and function-like
- Object-like macros
  - **`#define`** <identifier> <replacement_token_list>
  - No arguments
  - Whenever the identifier appears in the source code, it is replaced with the replacement token list (possibly empty)
  - Conventionally used as part of good programming practice to create symbolic names for constants
- For example,
  - **`#define PI 3.14159`**
  - **`#define EPS 1.0e-9`**

Function-like Macros

- **`#define`** <identifier>(<parameter_list>) <replacement_token_list>
- Takes arguments
- Must not have any whitespace between the identifier and the first, opening, parenthesis
- If whitespace is present, the macro will be interpreted object-like
- The identifier is only replaced when the following token is also a left parenthesis that begins the argument list of the macro invocation
- The exact procedure followed by the expansion of function-like macros is dependent upon implementation

Function-like Macros (contd.)

- `#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))`

Macros in stdio.h and ctype.h

- getc() and putc() are macros defined in stdio.h
  - Read a character from a file
  - Write a character to a file
- stdin : standard input file
- stdout : standard output file
- In stdio.h
  - #define getchar() getc(stdin)
  - #define putchar() putc((c), stdout)
- In ctype.h
  - isalpha(c), isupper(c), islower(c), …, toupper(c), tolower(c), toascii(c), …

Conditional Compilation

- **`#if`** constant_integral_expression
- **`#ifdef`** identifier
- **`#ifndef`** identifier
- Conditional compilation of the code that follows until #endif is reached

```
#if /* or #ifdef or #ifndef */
     ...
#elif constant_integral_expression
     ...
#else
     ...
#endif
```

Conditional Compilation (contd.)

```
#define DEBUG 1
      …
#if DEBUG
     printf("debug msg\n");
#endif
```

```
#define DEBUG
      …
#ifdef DEBUG
     printf("debug msg\n");
#endif
```

Conditional Compilation (contd.)

- **`#undef`** identifier
  - Undefine the macro defined by **`#define`**
  - For example,

```
#include "pi.h"
    ...
#undef PI
#define PI 3.14
```

- **`defined`** identifier
- **`defined(`** identifier **`)`**
  - Operator
  - 1 if identifier is currently defined, 0 otherwise

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 should be surrounded by double quotes

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

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

Operator

- Merges tokens together
- For example, foo(1) is replaced by x1, foo(2) by x2, and foo(3) by x3 in the code below

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

    ...

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

    ...
```

Assert Macro

```
#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
```