Introduction to C

Bryn Mawr College
CS246 Programming Paradigm

History of C

- Developed during 1969-73 in the Bell Labs.
- C is a by-product of the UNIX operating system.
- Original UNIX operating system (1969) was written by Ken Thompson
- Ran on the DEC PDP-7 computer (8K words of main memory)
- Written in assembly language
- Painful to debug and hard to enhance
- B was designed by Thompson
- Dennis Ritchie developed C – meant to be an extended version of B

History of C

- Original machine (DEC PDP-11) was very small
  - 24k bytes of memory,
  - 12k used for operating systems
- Small in memory size, not actual size.

The C Language

- Currently one of the most commonly-used programming languages
- a low-level language
- small (with limited set of features), but powerful.
- C is permissive. It does not, in general, try to protect a programmer from his/her mistakes.
- very portable : compiler exists for virtually every processor
- can be error-prone, difficult to understand and modify

Programming Process

- Source code must carry extension .c
- Identifiers may be named with any valid Unix file name
  - may contain letters, digits and underscores,
  - but must begin with a letter or underscore.
  - case-sensitive:
    ten teN tEn tEN Ten TeN TEn TEN – all different

Example

```
/* helloworld.c,
   Displays a message */

#include <stdio.h>

int main() {

  printf("Hello, world!\n");
  return 0;

}
```

helloworld.c

Hello World in C

```
#include <stdio.h>

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

Preprocessor used to share information among source files

Similar to Java's import

Hello World in C

```
#include <stdio.h>

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

Program mostly is a collection of functions

"main" function special: the entry point

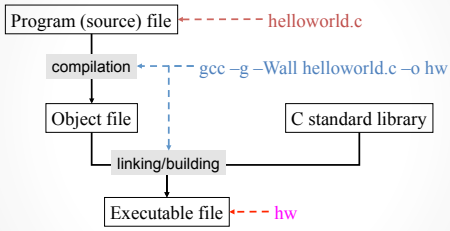
"int" qualifier indicates function returns an integer

I/O performed by a library function

The Compiler

- gcc (Gnu C Compiler)
- gcc –g –Wall helloworld.c –o hw
- gcc flags
  - -g (produce debugging info for gdb)
  - -Wall (print warnings for all events)
  - -o filename (name output file with filename, default is a.out)

Programming Process Summary



All this is done under Unix

C Program Style

- Case sensitive
- Ignores blanks
- Comments
  1. Ignored between `/*` and `*/`
  2. Comments are integral to good programming!
- All local variables must be declared before they are used !!!

Data Types

- Integer
  - C keyword: `int, short, long`
  - Range: typically 32-bit (±2 billion), 16-bit, 64-bit
- Floating-point number
  - C keyword: `float, double`
  - Range: 32-bit ($\pm 10^{38}$), 64-bit
  - Examples: `0.67f, 123.45f, 1.2E-6f, 0.67, 123.45, 1.2E-6`

In general, use double

Variables and Basic Operations

- Declaration (identify variables and type)

```
int x;
int y, z;
```

- Assignment (value setting)

```
x = 1;
y = value-returning-expression;
```

- Reference (value retrieval)

```
y = x * 2;
```

Constants

- Integer
  - const int year = 2002;
- Floating point number
  - const double pi = 3.14159265;
- Constants are variables whose initial value can not be changed.
- Comparable to `static final`

Output Functions

- Output characters

```
printf("Text message\n");
```

\n for new line

- Output an integer

```
int x = 100;
printf("Value = %d\n", x);
```

Output: `Value = 100`

Variations

- Output a floating-point number

```
double y = 1.23;
printf("Value = %f\n", y);
```

- Output multiple numbers

```
int x = 100;
double y = 1.23;
printf("x = %d, y = %f\n", x, y);
```

Output: `x = 100, y = 1.230000`

`printf` Summary

```
printf("                ",        );
```

- Text containing special symbols
  - `%d` for an integer
  - `%f` for a floating-point number
  - `\n` for a newline
- List of variables (or expressions)
  - In the order corresponding to the `%` sequence

Display Problem

- Problem
  - Precision of `double`: 15 digits
  - Precision of `%f`: 6 digits below decimal
  - Cannot show all the significant digits
- Solution
  - More flexible display format possible with `printf`

% Specification

- `%i` **`int`**, **`char`** (to show value)
- `%d` same as above (`d` for decimal)
- `%f` **`double`** (floating-point)
- `%e` **`double`** (exponential, e.g., `1.5e3`)

Formatting

- Precision `%.#f`
- Width `%#f`, `%#d`
  - Note: Entire width
- Zero-padding `%0#d`
- Left-justification `%-#d`
- Various combinations of the above

Replace # with digit(s)

Formatting Example (1)

```
%f    with 1.23456789 >1.234568<
%.10f with 1.23456789 >1.2345678900<
%.2f  with 1.23456789 >1.23<

%d   with 12345 >12345<
%10d with 12345 >     12345<
%2d  with 12345 >12345<

%f    with 1.23456789 >1.234568<
%8.2f with 1.23456789 >    1.23<
```

Formatting Example (2)

```
%d:%d     with 1 and 5 >1:5<
%02d:%02d with 1 and 5 >01:05<

%10d  with 12345 >     12345<
%-10d with 12345 >12345     <
```

formatting.c

Arithmetic Operators

- Unary: `+`, `-` (signs)
- Binary: `+`, `-`, `*` (multiplication), `/` (division), `%` (modulus, int remainder)
- Parentheses: `(` and `)` must always match.
  - Good: `(x)`, `(x - (y - 1)) % 2`
  - Bad: `(x`, `)x(`

Types and Casting

- Choose types carefully
- An arithmetic operation requires that the two values are of the same type
- For an expression that involves two different types, the compiler will cast the smaller type to the larger type
- Example: 4 * 1.5 = 6.0

Mixing Data Types

- `int` values only ⇒ `int`
  - `4 / 2` ⇒ 2
  - `3 / 2` ⇒ 1
  - `int x = 3, y = 2;`
    `x / y` ⇒ 1
- Involving a `double` value ⇒ `double`
  - `3.0 / 2` ⇒ 1.5

Assignment of Values

- `int x;`
  - `x = 1;`
  - `x = 1.5;` `/* x is 1 */` warning
- `double y;`
  - `y = 1;` `/* y is 1.0 */`
  - `y = 1.5;`
  - `y = 3 / 2;` `/* y is 1.0 */`

`int` evaluation; warning

Example

`mixingtypes.c`

```
int i, j, k, l;
double f;

i = 3;
j = 2;
k = i / j;
printf("k = %d\n", k);

f = 1.5;
l = f;                  /* warning */
printf("l = %d\n", l);  /* truncated */
```

`sizeof` and Type Conversions

- `sizeof(type)`
  - The sizeof operator returns the number of bytes required to store the given type

Implicit conversions
- arithmetic
- assignment
- function parameters
- function return type
- promotion if possible

Explicit conversions
- casting

```
int x;
x = (int) 4.0;
```

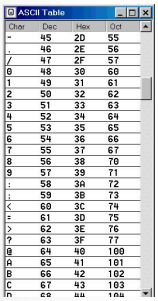
Use of `char` (character)

- Basic operations
  - Declaration: `char c;`
  - Assignment: `c = 'a';`
  - Reference: `c = c + 1;`
- Constants
  - Single-quoted character (only one)
  - Special characters: `'\n'`, `'\t'` (tab), `'\"'` (double quote), `'\''` (single quote), `'\\'` (backslash)

Characters are Integers

- A `char` type represents an integer value from 0 to 255 (1 byte) or –128 to 127.
- A single quoted character is called a "character constant".
- C characters use ASCII representation:
- `'A' = 65 … 'Z' = 'A' + 25 = 90`
- `'a' = 97 … 'z' = 'a' + 25 = 122`
- `'0' != 0 (48), '9' - '0' = 9`
- Never make assumptions of `char` values
  - Always write `'A'` instead of `65`

ASCII Table



American Standard Code for Information Interchange
A standard way of representing the alphabet, numbers, and symbols (in computers)

wikipedia on ASCII

`char` Input/Output

- Input
  - `char getchar()` receives/returns a character
  - Built-in function
- Output
  - `printf` with `%c` specification

chartypes.c

```
int main() {
  char c;
  c = getchar();
  printf("Character >%c< has the value %d.\n", c, c);
  return 0;
}
```

`scanf` Function

```
scanf("                ",          );
```

- Format string containing special symbols
  - `%d` for `int`
  - `%f` for `float`
  - `%lf` for `double`
  - `%c` for `char`
  - `\n` for a newline
- List of variables (or expressions)
  - In the order corresponding to the `%` sequence

`scanf` Function

- The function `scanf` is the input analog of `printf`
- Each variable in the list MUST be prefixed with an &.
- Ignores white spaces unless format string contains `%c`

`scanf` Function

```
int main() {
  int x;

  printf("Enter a value:\n");
  scanf("%d", &x);
  printf("The value is %d.\n",
x);
  return 0;
}
```

`scanf` with multiple variables

scanf.c

```
int main() {
  int x;
  char c;
  printf("Enter an int and a char:");
  scanf("%d %c", &x, &c);
  printf("The values are %d, %c.\n",
         x, c);
  return 0;
}
```

`scanf` Function

- Each variable in the list MUST be prefixed with an &.
- Read from standard input (the keyboard) and tries to match the input with the specified pattern, one by one.
- If successful, the variable is updated; otherwise, no change in the variable.
- The process stops as soon as **`scanf`** exhausts its format string, or matching fails.
- Returns the number of successful matches.

`scanf` Continued

- White space in the format string match any amount of white space, including none, in the input.
- Leftover input characters, if any, including one '**`\n`**' remain in the input buffer, may be passed onto the next input function.
  - Use **`getchar()`** to consume extra characters
  - If the next input function is also **`scanf`**, it will ignore '**`\n`**' (and any white spaces).

`scanf` Notes

- Beware of combining **`scanf`** and **`getchar()`**.
- Use of multiple specifications can be both convenient and tricky.
  - Experiment!
- Remember to use the return value for error checking.