3. Character Input/Output
ENEE 140

Prof. Tudor Dumitraș
Associate Professor, ECE
University of Maryland, College Park

http://ter.ps/enee140

1

Today's Lecture

- Where we've been
  - Variables and Constants
  - Arithmetic operations
  - `while` loops
- Where we're going today
  - Increment, relational and logical operators
  - Branching: `if` statement
  - Loops: `for`
  - Data types: chars
  - Input and output
- Where we're going next
  - Functions

2

2

We've Seen: Programming Concepts

- Write programs to compute quantities difficult to work out in your head
  - Programming languages provide **variables** and **arithmetic operations**
- Come up with step-by-step procedure for arriving at the result
  - Programming languages provide **loops** and **branching statements**
- Break down a problem into simpler steps
  - Programming languages provide **functions**
  - Helpful for solving problems from the top down to the small details
- Communicate with the user
  - Programming languages provide **input/output** mechanisms

3

3

Reminder: Textbook Clarifications

- If you find the K&R textbook confusing …

… Consult Steve Summit's excellent notes on the textbook:
http://www.eskimo.com/~scs/cclass/krnotes/top.html

- Linked from the class web page

… Attend the ENEE 140 lectures

… Ask questions on Piazza

5

5

Increment Operators

- We've seen

```
a = a + 1;
```
increment by assigning old value + 1

- Increment and decrement operators in C

```
a++;
```
same as a = a + 1;

```
++a;
```
same as a = a + 1;

```
a--;
```
same as a = a - 1;

```
--a;
```
same as a = a - 1;

  - There is a subtle difference between `a++` and `++a` (more on this later)

- Assignment operations also return the value assigned

```
a = 0;
b = ++a;
```
both a and b become 1

```
a = b = 0;
```
both a and b become 0

6

6

Value of Assignment Expression

- In C, an assignment expression returns the value that is assigned

```
b = (a = 0);
```
a becomes 0, and b also becomes 0

- This means that you can write things like this:

```
c = b = a = 0;
```
a, b and c become 0

7

7

Specifying Conditions

- We've seen

```
while (condition) {
    statements
}
```

statements executed repeatedly while *condition* is true

How can we specify the condition?

8

Relational and Logical Operators

- We've seen: **relational operators**, used for specifying conditions

`if (a < b) {…}`	*condition*: if a less than b
`if (a > b) {…}`	*condition*: if a greater than b
`if (a <= b) {…}`	*condition*: if a less than or equal to b
`if (a >= b) {…}`	*condition*: if a greater than or equal b
`if (a == b) {…}`	*condition*: if a equal to b
`if (a != b) {…}`	*condition*: if a not equal to b

- **Logical operators** are used for combining conditions

`if (cond1 && cond2) {…}`	*condition*: both cond1 and cond2
`if (cond1 || cond2) {…}`	*condition*: either cond1 or cond2
`if (!cond1) {…}`	*condition*: not cond1

9

Branching

- Execute statements conditionally

```
if (condition) {
    statements
}
```

statements are executed if *condition* is true

- Provide alternative to the condition

```
if (condition) {
    statements
} else {
    statements_2
}
```

statements are executed if *condition* is true

statements_2 are executed if *condition* is false

10

10

Loops

- We've seen

```
int i = 0;
while (i < 10) {
    ...
    i++;
}
```

initialize i

test !(exit condition)

increment i

- Iterate over a set of values

```
int i;
for (i = 0; i < 10; i++) {
    ...
}
```

iterate over i in [0, 10)

- **Important: every loop must have an *exit condition* that eventually becomes true**

11

11

Common Mistake: Infinite Loops

- While loop example:

```
int i = 0;
while (i < 10) {
    printf("%d\n", i);
    if (i >0) {
        i++;
    }
}
```

When is i incremented?

How many iterations does this loop execute?

- For loop example:

```
int i = 0;
for ( ; i < 10 ; ) {
    printf("%d\n", i);
}
```

you may omit any of the 3 components of a for statement …
… but you must still ensure the loop exit

12

12

Typical Exam Question

- Consider the following `while` loop:

```
int a = 0, b = 5;
while (a < b) {
    printf("%d %d\n", a, b);
    a = a + 2;
    b = b + 1;
}
```

- Write a `for` loop that does the same thing

```
for (b = 5; a < b;                ) {
    printf("%d %d\n", a, b);
}
```

14

14

Implementation Options for Conditional Execution

- How many times is the block executed?

```
if (i < 10) {
    block of statements
}
```

- How many times is the block executed?

```
while (i < 10) {
    block of statements
}
```

- How many times is the block executed?

```
for (i = 0; i <= 10; i++) {
    block of statements
}
```

(assuming that i is not modified inside the block)

16

16

Question from Quiz 2

- How many times is the printf statement in the for loop below executed:

```
for (fahr = 0; fahr <= 100; fahr = fahr + 20) {
  printf("%.2f\n", fahr);
}
```

18

18

Data Types

- We've seen

```
int a = 1;          integer variable
float b = 1.1;      floating-point variable
```

- Larger data types (can hold larger values)

```
long a = 1;         integer variable
double b = 1.1;     floating-point variable
```

- Characters

```
char c = 'A';       holds one character
char c = '\n';
```

- A data type is a set of rules for handling a certain kind of variables
  - Rules govern the interpretation of internal representations and the operations allowed
    - We will discuss the implications of `int` and `float` representations in future lectures
  - In C, you must specify the type when declaring each variable

19

19

The `char` Data Type

- Internally, characters are represented as integers
- Rules for interpreting the value of the stored data

```
char c = 'D' + 1;                  value of c is 'E'
int diff = 'c' - 'a';              value of diff is 2
if (c >= 'A' && c <= 'Z') { … }    check if c is uppercase
```

- A–Z have consecutive codes (numerical values). So do a–z and 0–9
  - The offset between the lowercase and uppercase versions of a character is always the same

```
'A' - 'a' == 'B' - 'b'
```

  - Converting a lowercase character to uppercase

```
c = c +                            add the offset of the uppercase range
```

20

20

Reading and Writing Characters

- Read one character from the input

```
int c = getchar();
```

- Write one character to the output

```
putchar(c);
printf("%c", c);
```

- Important: getchar() returns an `int` rather than a `char`
  - This allows the function to return the special value `EOF` when no more input is available

```
while (getchar() != EOF) {
    …
}
```

- **Good programming practice: Have a mechanism to indicate errors and exceptional conditions (e.g. no more input)**

22

22

More on the `char` Data Type

- Internally, characters are represented as integers
- The corresponding value of the character is determined by an encoding scheme
  - For char: American Standard Code for Information Interchange (ASCII)
  - Other encoding schemes: Unicode
- You can examine the internal encoding of characters

```
printf("%d", c);
```

- **Good programming practice: Do not rely on the internal values of the encoding**

```
c = c + 'A' – 'a';
```

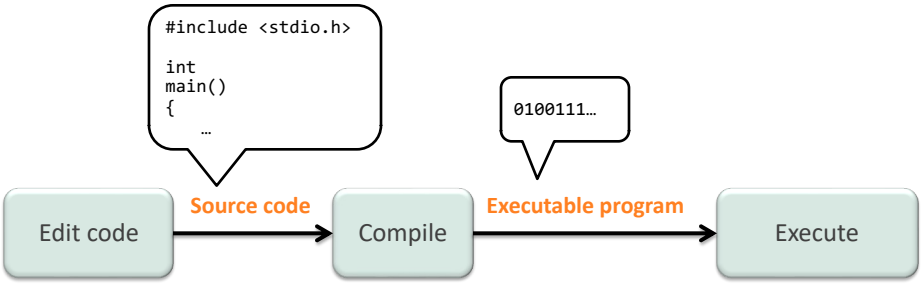
instead of `c = c - 35;`

23

23

Compiling from the Command Line

- We've seen: the programming toolchain



CLion: Build All

Command line: `gcc -o hello_world hello_world.c` `./hello_world`

Invoke these commands in the directory where your source file is located

- Good practice: compile and run your programs on GRACE (using Putty or the MacOS Terminal) before submitting

25

25

Review of Lecture

- What did we learn?

26

26

Next Steps

- Next lecture
  - Functions
- Assignments for this week
  - Read **K&R Chapters 1.7, 1.8, 7.2, 7.4, B4**
  - Weekly challenge: `temperature_conversion_function.c`
  - Homework: `enee140_lab03.pdf`, due on Friday at 11:59 pm
  - Quiz 3, due on Sunday at 11:59 am

27