Overview of C, Part 2

CSE 130: Introduction to Programming in C
Stony Brook University

Integer Arithmetic in C

- Addition, subtraction, and multiplication work as you would expect
- Division (/) returns the whole part of the division (the quotient)
  - 12 / 3 is 4
  - 15 / 2 is 7
- Modulus (%) returns the remainder
  - 12 % 3 is 0
  - 15 % 2 is 1

Short hand Operators

- Some operators are shortcuts for others
    - Ex. +=, -=, *=, /=, %=, ++, and --
- `x += 5;`
  is the same as
  `x = x + 5;`
- `y++;`
  is the same as
  `y = y + 1;`

The Increment (++) and Decrement (--) Operators

- When used by themselves, **`y++`** and **`++y`** have identical results
  - In an expression, they have different results

- The relative order of the operator matters:

  `y++`: use y’s current value, then increment it

  `++y`: increment y, then use the new value

- The same is true for decrement (--)

Operator Precedence

- Precedence rules specify the order in which operators are evaluated
- Remember PMDAS:
  - Parentheses, Multiplication, Division, Addition, Subtraction
- Associativity determines left-right order

Precedence Examples

- 3 - 8 / 4

- / has the highest precedence, so we compute 8 / 4 first, then subtract the result from 3

- Equivalent expression: 3 - (8 / 4)

What is the value of 3 * 4 + 18 / 2?

Precedence Examples

5 - 3 + 4 + 2 - 1 + 7

- + and - have equal precedence, so this expression is evaluated left to right:

```
(((((5 - 3) + 4) + 2) - 1) + 7)
```

- The innermost parentheses are evaluated first

Parentheses

- Parentheses can be used to force a different order of evaluation:
- 12 - 5 * 2 produces 2
- (12 - 5) * 2 produces 14

Expression Examples

What do the following expressions evaluate to?

1 + 2 * 3

(1 + 2) * 3

13 % 5

23 % 4 * 6

More Expression Examples

- 27.0 / 6.0
- 27.0 / 6
- 27 / 6
- Given:

  `int x = 5;`

  - `int y = x++ * 6;`
  - `int y = ++x * 6;`

`printf()` and `scanf()` revisited

- Each of these functions takes a list of *arguments* (input values):
  - a *control string*
  - an optional list of other arguments (data)
- The control string determines how the other arguments are displayed

Control Strings

- A control string may contain one or more ***conversion specifications*** (formats)
  - conversion specifications are replaced (or substituted) by the arguments that follow the control string, in order
  - They begin with a `%` and end with a conversion character
- For example, the statement

  ```
  printf(“%s”, “abc”);
  ```

  will replace “%s” with “abc” in the final output

`printf()` Conversion Characters

Conversion character	How the corresponding argument is printed
c	as a character
d	as a decimal integer
e	as a floating-point number in scientific notation
f	as a floating-point number
g	in the e-format or f-format, whichever is shorter
s	as a string

Three Equivalent Statements

```
printf(“abc”);


printf(“%s”, “abc”);


printf(“%c%c%c”, ‘a’, ‘b’, ‘c’);
```

Fields

- A *field* is the area where an argument is printed
  - The *field width* is the number of characters that make up the field
- Field width can be specified as an integer between the `%` and the conversion character
- For example,

  ```
  printf(“%c%3c%5c”, ‘a’, ‘b’, ‘c’);
  ```

  will print

  ```
  a  b    c
  ```

Control Strings for `scanf()`

- `scanf()` is used to collect user input from the keyboard

- It is called with a control string and a list of *addresses*

- The control string conversion specifiers describe how the input stream characters should be interpreted

- The addresses correspond to the memory locations where variables are stored

Parsing Data

- **`scanf()`** will skip whitespace (tabs, blanks, and newlines) when reading in numbers
- Whitespace is ***NOT*** skipped when **`scanf()`** is reading in characters

`scanf()` Conversion Characters

Conversion character	How input stream characters are converted
c	as a character
d	as a decimal integer
f	as a floating-point number (float)
lf or LF	as a floating-point number (double)
s	as a string

```
#include <stdio.h>

int main(void)
{
  char c1, c2, c3;
  int i;
  float x;
  double y;

  printf(“\n%s\n%s”, “Input three characters,”,
         “an int, a float, and a double:  “);

  scanf(“%c%c%c%d%f%lf”, &c1, &c2, &c3, &i, &x, &y);
  printf(“\nHere is the data that you typed in:\n”);
  printf(“%3c%3c%3c%5d%17e%17e\n\n”,
         c1, c2, c3, i, x, y);
  return 0;
}
```

Return Values

- **`printf()`** and **`scanf()`** each return an integer value when they complete

- **`printf()`** returns the number of characters printed, or a negative value if an error occurred

- **`scanf()`** returns the number of successful conversions or the system-defined end-of-value.

Flow of Control

Control Flow

- Normally, C programs are executed sequentially
- We can alter this process using *conditionals* (which provide alternative actions) and *loops* (which repeat groups of statements)

Conditions

- Conditional statements execute a test to determine which path to follow

- This test consists of an expression that is evaluated

- Normally, this expression compares two or more values

True and False Values

- Any expression with a non-zero value is considered to be true
  - Ex. 1, 3.14159, -23
- An expression is only false if its value is 0
- Common programming error: using ‘=’ (assignment) instead of “==” (equality)
  - Ex. `if (x = 5)`

Relational Operators

Operator	Meaning	Example
<	Less than	age < 30
>	Greater than	height > 6.2
<=	Less than or equal to	taxable <= 20000
>=	Greater than/equal to	temp >= 98.6
==	Equal to	grade == 100
!=	Not equal to	number != 250

The `if` Statement

- General form:

```
if ( condition )
    statement (or block of statements) to be
    executed if condition is true
```

- Ex.

```
if (length < 2)
    printf(“Too short!\n”);
```

The `if-else` Statement

- Select one of two possible execution paths, based on the result of a comparison

General format:

```
 if ( expression )
   statement block 1
 else
   statement block 2
```

Compound Statements

- `if` and `else` only execute a single following statement

- We can get around this by enclosing multiple statements in curly braces
  - The resulting block is called a *compound statement*

- Style suggestion: always use curly braces around the body of an `if` or `else` clause

```
if (key == ‘F’)
{
  contemp = (5.0/9.0) * (intemp - 32.0);
  printf(“Converted to Celsius\n”);
}
else
{
 contemp = (9.0/5.0) * intemp + 32.0;
 printf(“Converted to Fahrenheit.\n”);
}
```

Iterative Programming

- Many programs perform the same task many times
    - Operations are repeated on different data
- Ex.Adding a list of numbers
- Ex. Displaying a menu of options
- Repetitive tasks are specified using *loops*

Loop Elements

- All loop constructs share four basic elements:

  1. Initialization
  2. Testing the loop condition
  3. The loop body (the task to be repeated)
  4. The loop update

- The order of these elements may vary

Initialization

- This section of code is used to set starting values

- For example, setting a total to 0 initially

- This can be done as part of the loop, or separately before the loop code begins

Loop Tests

- Test expressions are used to determine whether the

  loop should execute (again)

- Tests compare one value/variable with another

- If the test evaluates to TRUE, then the loop will execute another time

Loop Update

- This step changes the value(s) of the loop variable(s) before the loop repeats
- Ex. moving to the next item to process
- This can be done explicitly as part of the loop, or it can be done inside the loop body

`while` Loops

- `while` loops can execute an arbitrary number of times
- Order of execution:

1. Initialization
2. Loop condition test
3. Loop body
4. Loop update

General Form

```
initialization
while ( loop condition test )
{
    loop body
    loop update
}
```

`while` Loop Example

```
int countDown = 5;
while (countDown >= 0)
{
  printf(“%d...”, countDown);
  countDown--;
}
```

Loop Output

```
5...4...3...2...1...0...
```

Another Example

```
int root = 0;
while (root < 10)
{
  root += 1;
  printf(“%d * %d = ”, root, root);
  printf(“%d\n”, root * root);
}
```

root	output
0	1 * 1 = 1
1	2 * 2 = 4
2	3 * 3 = 9
3	4 * 4 = 16
4	5 * 5 = 25
5	6 * 6 = 36
6	7 * 7 = 49
7	8 * 8 = 64
8	9 * 9 = 81
9	10 * 10 = 100

`for` Loops

- `for` loops execute a fixed number of times

- Order of execution:

  1. Initialization
  2. Loop condition test
  3. Loop body
  4. Loop update

General Form

```
for ( initialization ;
      loop condition test ;
      loop update )
{
   loop body
}
```

`for` Loop Example

```
int i;
for (i = 0; i < 10; i++)
{
 printf(“%d ”, i);
}
```

Loop Output

```
0123456789
```

Another Example

```
int nextNumber, i, sum = 0;
for (i = 0; i < 5; i++)
{
  printf(“\nEnter a number: ”);
  scanf(“%d ”, nextNumber);
  sum += nextNumber;
}
```

i	nextNumber	sum
-	-	0
0	2	2
1	15	17
2	5	22
3	7	29
4	3	32
5	-	32