CSE105 – Structure Programming

ARITHMETIC EXPRESSION, SWITCH CASE ETC…

Escape Sequences

- The `\n` code that used in format strings is called an ***escape sequence.***
- Escape sequences enable strings to contain
  - nonprinting (control) characters and
  - characters that have a special meaning
- A partial list of escape sequences:

New line	`\n`
Horizontal tab	`\t`
Black slash	`\\`
Double Quote	`\"`
Single Quote	`\'`

Escape Sequences

- A string may contain any number of escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

Escape Sequences

- Another common escape sequence is `\"`, which represents the `"` character:

```
printf("\"Hello!\"");
  /* prints "Hello!" */
```

- To print a single `\` character, put two `\` characters in the string:

```
printf("\\");
  /* prints one \ character */
```

Operator Precedence

- The arithmetic operators have the following relative precedence:

  Highest: `+` `-` (unary)
  `*` `/` `%`
  Lowest: `+` `-` (binary)

- Examples:

  `i + j * k` is equivalent to `i + (j * k)`

  `-i * -j` is equivalent to `(-i) * (-j)`

  `+i + j / k` is equivalent to `(+i) + (j / k)`

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right.
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and –) are all left associative, so

  `i - j - k` is equivalent to `(i - j) - k`

  `i * j / k` is equivalent to `(i * j) / k`

Operator Associativity

- An operator is ***right associative*** if it groups from right to left.
- The unary arithmetic operators (`+` and `–`) are both right associative, so

  `- + i` is equivalent to `-(+i)`

Side Effects

- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The = operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

Side Effects

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

- `i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

Side Effects

- An assignment of the form $v = e$ is allowed wherever a value of type v would be permitted:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
```

- "Embedded assignments" can make programs hard to read.
- They can also be a source of subtle bugs.

Lvalues

- The assignment operator requires an ***lvalue*** as its left operand.
- An lvalue represents an object stored in computer memory, not a constant or the result of a computation.
- Variables are lvalues; expressions such as `10` or `2 * i` are not.

Lvalues

- Since the assignment operator requires an lvalue as its left operand, it's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;          /*** WRONG ***/
i + j = 0;       /*** WRONG ***/
-i = j;          /*** WRONG ***/
```

- The compiler will produce an error message such as *"invalid lvalue in assignment."*

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common.
- Example:

```
i = i + 2;
```

- Using the += compound assignment operator, we simply write:

```
i += 2;    /* same as i = i + 2; */
```

Compound Assignment

- There are nine other compound assignment operators, including the following:

  `-=` `*=` `/=` `%=`

- All compound assignment operators work in much the same way:

  v `+=` *e* adds *v* to *e*, storing the result in *v*

  v `-=` *e* subtracts *e* from *v*, storing the result in *v*

  v `*=` *e* multiplies *v* by *e*, storing the result in *v*

  v `/=` *e* divides *v* by *e*, storing the result in *v*

  v `%=` *e* computes the remainder when *v* is divided by *e*, storing the result in *v*

Compound Assignment

- One problem is operator precedence: **`i *= j + k`** isn't the same as `i = i * j + k`.
- It means: `i = i * (j + k)`.

Increment and Decrement Operators

- Two of the most common operations on a variable are “incrementing” (adding 1) and “decrementing” (subtracting 1):

```
i = i + 1;
j = j - 1;
```

- Incrementing and decrementing can be done using the compound assignment operators:

```
i += 1;
j -= 1;
```

Increment and Decrement Operators

- C provides special `++` (***increment***) and `--` (***decrement***) operators.
- The `++` operator adds 1 to its operand. The `--` operator subtracts 1.
- The increment and decrement operators are tricky to use:
  - They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).
  - They have side effects: they modify the values of their operands.

Increment and Decrement Operators

- Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

Increment and Decrement Operators

- `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later."
- How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Increment and Decrement Operators

- The -- operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);    /* prints "i is 0" */
printf("i is %d\n", i);      /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 0" */
```

Increment and Decrement Operators

- When `++` or `--` is used more than once in the same expression, the result can often be hard to understand.
- Example:

```
i = 1;
j = 2;
k = ++i + j++;
```

The last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

Increment and Decrement Operators

- In contrast, executing the statements

```
i = 1;
j = 2;
k = i++ + j++;
```

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

Expression Evaluation

- Table of operators discussed so far:

Precedence	***Name***	***Symbol(s)***	***Associativity***
1	increment (postfix)	`++`	left
	decrement (postfix)	`--`	
2	increment (prefix)	`++`	right
	decrement (prefix)	`--`	
	unary plus	`+`	
	unary minus	`-`	
3	multiplicative	`* / %`	left
4	additive	`+ -`	left
5	assignment	`= *= /= %= += -=`	right

Expression Evaluation

- The table can be used to add parentheses to an expression that lacks them.

- Example: let: `b=10, c=3, d=5, e=15, f=-1`

  `a = b += c++ - d + --e / -f`

`Result: a=22, b=22, c=4, d=5, e=14, f=-1`

Expression Statements

- A slip of the finger can easily create a "do-nothing" expression statement.
- For example, instead of entering

  ```
  i = j;
  ```

  we might accidentally type

  ```
  i + j;
  ```

- Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

Relational Operators

- C's ***relational operators:***

  < less than
  > greater than
  <= less than or equal to
  >= greater than or equal to

- These operators produce 0 (false) or 1 (true) when used in expressions.
- The relational operators can be used to compare integers and floating-point numbers, with operands of mixed types allowed.

Relational Operators

- The precedence of the relational operators is lower than that of the arithmetic operators.
  - For example, `i + j < k - 1` means `(i + j) < (k - 1)`.
- The relational operators are left associative.

The "Dangling `else`" Problem

- When if statements are nested, the "dangling `else`" problem may occur:

```
if (y != 0)
  if (x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0\n");
```

- The indentation suggests that the `else` clause belongs to the outer `if` statement.
- However, C follows the rule that an `else` clause belongs to the **nearest `if` statement** that hasn't already been paired with an `else`.

The "Dangling `else`" Problem

- A correctly indented version would look like this:

```
if (y != 0)
  if (x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0\n");
```

The "Dangling `else`" Problem

- To make the `else` clause part of the outer `if` statement, we can enclose the inner `if` statement in braces:

```
if (y != 0) {
  if (x != 0)
    result = x / y;
} else
  printf("Error: y is equal to 0\n");
```

- Using braces in the original `if` statement would have avoided the problem in the first place.

Conditional Expressions

- C's ***conditional operator*** allows an expression to produce one of two values depending on the value of a condition.
- The conditional operator consists of two symbols (`?` and `:`), which must be used together:

  expr1 `?` *expr2* `:` *expr3*
- The operands can be of any type.
- The resulting expression is said to be a ***conditional expression.***

Conditional Expressions

- The conditional operator requires three operands, so it is often referred to as a ***ternary*** operator.
- The conditional expression *expr1* ? *expr2* : *expr3* should be read "if *expr1* then *expr2* else *expr3*."
- The expression is evaluated in stages: *expr1* is evaluated first; if its value isn't zero, then *expr2* is evaluated, and its value is the value of the entire conditional expression.
- If the value of *expr1* is zero, then the value of *expr3* is the value of the conditional.

Conditional Expressions

- Example:

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;            /* k is now 2 */
k = (i >= 0 ? i : 0) + j;     /* k is now 3 */
```

- The parentheses are necessary, because the precedence of the conditional operator is less than that of the other operators discussed so far, with the exception of the assignment operators.

Conditional Expressions

- Conditional expressions tend to make programs shorter but harder to understand, so it's probably best to use them sparingly.
- Conditional expressions are often used in `return` statements:

```
return i > j ? i : j;
```

Conditional Expressions

- Calls of `printf` can sometimes benefit from condition expressions. Instead of

```
if (i > j)
  printf("%d\n", i);
else
  printf("%d\n", j);
```

  we could simply write

```
printf("%d\n", i > j ? i : j);
```

- Conditional expressions are also common in certain kinds of macro definitions.

The `switch` Statement (Repeat)

- A cascaded `if` statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

The `switch` Statement

- The `switch` statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

The `switch` Statement

- A `switch` statement may be easier to read than a cascaded `if` statement.
- `switch` statements are often faster than `if` statements.
- Most common form of the `switch` statement:

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

The `switch` Statement

- The word `switch` must be followed by an integer expression—the ***controlling expression***—in parentheses.
- Characters are treated as integers in C and thus can be tested in `switch` statements.
- Floating-point numbers and strings don't qualify, however.

The `switch` Statement

- Each case begins with a label of the form

  `case` *constant-expression* :
- A ***constant expression*** is much like an ordinary expression except that it can't contain variables or function calls.
  - `5` is a constant expression, and `5 + 10` is a constant expression, but `n + 10` isn't a constant expression (unless `n` is a macro that represents a constant).
- The constant expression in a case label must evaluate to an integer (characters are acceptable).

The `switch` Statement

- After each case label comes any number of statements.
- No braces are required around the statements.
- The last statement in each group is normally `break`.

The `switch` Statement

- Duplicate case labels aren't allowed.
- The order of the cases doesn't matter, and the `default` case doesn't need to come last.
- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

The `switch` Statement

- To save space, several case labels can be put on the same line:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          printf("Passing");
          break;
  case 0:  printf("Failing");
          break;
  default: printf("Illegal grade");
          break;
}
```

- If the `default` case is missing and the controlling expression's value doesn't match any case label, control passes to the next statement after the `switch`.

The Role of the `break` Statement

- Executing a `break` statement causes the program to "break" out of the `switch` statement; execution continues at the next statement after the `switch`.
- The `switch` statement is really a form of "computed jump."
- When the controlling expression is evaluated, control jumps to the case label matching the value of the `switch` expression.
- A case label is nothing more than a marker indicating a position within the `switch`.

The Role of the `break` Statement

- Without `break` (or some other jump statement) at the end of a case, control will flow into the next case.
- Example:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Illegal grade");
}
```

- If the value of `grade` is 3, the message printed is

```
GoodAveragePoorFailingIllegal grade
```

The Role of the `break` Statement

- Omitting `break` is sometimes done intentionally, but it's usually just an oversight.
- It's a good idea to point out deliberate omissions of `break`:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0: total_grades++;
          break;
}
```

- Although the last case never needs a `break` statement, including one makes it easy to add cases in the future.

Program: Printing a Date in Legal Form

- Contracts and other legal documents are often dated in the following way:

```
Dated this __________ day of __________ , 20__ .
```

- The `date.c` program will display a date in this form after the user enters the date in month/day/year form:

```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```

- The program uses `switch` statements to add "th" (or "st" or "nd" or "rd") to the day, and to print the month as a word instead of a number.

date.c


```
/* Prints a date in legal form */

#include <stdio.h>

int main(void)
{
  int month, day, year;

  printf("Enter date (mm/dd/yy): ");
  scanf("%d /%d /%d", &month, &day, &year);

  printf("Dated this %d", day);
  switch (day) {
    case 1: case 21: case 31:
      printf("st"); break;
    case 2: case 22:
      printf("nd"); break;
    case 3: case 23:
      printf("rd"); break;
    default: printf("th"); break;
  }
  printf(" day of ");
```

```
  switch (month) {
    case 1:  printf("January");   break;
    case 2:  printf("February");  break;
    case 3:  printf("March");     break;
    case 4:  printf("April");     break;
    case 5:  printf("May");       break;
    case 6:  printf("June");      break;
    case 7:  printf("July");      break;
    case 8:  printf("August");    break;
    case 9:  printf("September"); break;
    case 10: printf("October");   break;
    case 11: printf("November");  break;
    case 12: printf("December");  break;
  }

  printf(", 20%.2d.\n", year);
  return 0;
}
```