Lectures: C Programming Language

Programming Systems and Tools

State University of New York at Binghamton

January 25, 2024

BINGHAMTON UNIVERSITY | THOMAS J. WATSON COLLEGE OF ENGINEERING AND APPLIED SCIENCE

Control Flow

Outline

Statements

Statement

Fragments of a C program executed in a sequence. The body of a function is a statement, comprised of a sequence of statements and declarations. An expression becomes a statement when it is followed by a semicolon.

Simple Statement `expr;`

Empty Statement `;`

Specifier Statement `identifier:expr;` → used with `goto`

Compound Statement `{ statement | declaration ... }` → sometimes called a *block*, this is a brace-enclosed sequence of statements and declarations creating *scope*.

Conditional Statements

If-Else

```
if ( expression ) true-statement
if ( expression ) true-statement else false-statement
```

1. `expression` is always evaluated
2. if `expression` is true, then only `true-statement` is executed
3. if `expression` is false and there is a `false-statement`, then it is executed instead

Shortcuts that improve readability:

- `if (expression)` is the same as `if (expression != 0)`
- `if (!expression)` is the same as `if (expression == 0)`

Conditional Statements

Avoiding Ambiguity with Compound Statements

Always associate an `else` with the closest previous `else`-less `if`:

```
1  if (n > 0)
2          if (a > b)
3                  z = a;
4          else
5                  z = b;
```

Compound statement braces can force association:

```
1  if (n > 0) {
2          if (a > b)
3                  z = a;
4  } else {
5          z = b;
6  }
```

Indentation doesn't resolve ambiguity (like in Python)! With which `if` does the `else` on line 7 associate?

Conditional Statements

Avoiding Ambiguity with Compound Statements

```
1  if (n >= 0)
2          for (i = 0; i < n; i++)
3                  if (s[i] > 0){
4                          printf("n is not negative");
5                          return i;
6                  }
7  else
8          printf("n is negative");
```

Conditional Statements

Multi-way Conditionals

Multi-way conditional expressions are possible by chaining `else if` clauses:

```
1  if (expression)
2          statement
3  else if (expression)
4          statement
5  else if (expression)
6          statement
7  else
8          statement
```

- If any expression is true, its associated statement is executed, and the chain of clauses terminates
- The final `else` serves as a none-of-the-above case.
  - → If there is no explicit default, it can be omitted or used to catch an "impossible" condition.

Fizzbuzz Example

```
1  if (x%3==0 and x%5==0){
2          puts("fizzbuzz ");
3  } else if (x % 3 == 0){
4          puts("fizz ");
5  } else if (x % 5 == 0) {
6          puts("buzz ");
7  } else{
8          printf("%d ", x);
9  }
```

Selection Statements

If you find yourself repeating many if-else if-else conditions, try a `switch`

Switch Statement

```
switch ( expression ) {
case const-expr: statements
default: statements
```

```
switch(digit){
        case 0: puts("zero"); break;
        case 1: puts("one"); break;
        default: puts("not a binary digit");
}
```

- When the case matches the selector expression, all code following the label executes until the end of the switch statement or a break is reached
- A `default` case is optional, and always matches regardless of the actual value of the expression

Iteration Statements

- Three types of iteration statements: `while`, `do ... while`, and `for`
- Each iterates while a *conditional expression* remains true
- All three are semantically equivalent (i.e. they can be made to do the same things), so pick the right one based on what is most readable
  - while executes the conditional expression before the loop statement, so it might not execute.
  - do-while executes the loop statement before the conditional expression, so loop statement will execute at least once.
  - for is like while in that it might not execute the loop body. Biggest difference is the addition of an init-clause, which can be used to initialize a loop iterator that has scope inside the loop statement!

Iteration Statements

Iteration statements repeatedly execute a statement.

Iteration Statements

```
while ( expression ) statement
do statement while ( expression );
for ( init-clause ; conditional-expr ; iteration-expr ) statement
```

- Iteration statements can all be made semantically equivalent to each other!
- The difference between while and do-while is `while` may or may not evaluate `statement`, while `do-while` must evaluate `statement` at least once before exiting iteration.
- For's conditional and iteration expressions are both optional. The init-clause is evaluated once, and its result is discarded $\rightarrow$ use it to initialize the loop counter.

Infinite Loops

Some loops used in kernel code or in synchronization primitives ("spin locks"), for example, are intended to never terminate. Commonly used idioms:

- `for(;;)`
- `while(true)`

Look out for these accidental infinite loops . . . Use comparison operators with care!:

- `for(unsigned int x = 10;x >= 0;x--)`
- `while(x = true)`

Advanced Control Flow

Using the comma operator, we can have multiple statements per clause in a for-loop:

```
1  void reverse(char s[]){
2          int c,i,j;
3          for (i = 0, j = strlen(s)-1; i < j; i++, j--){
4                  c = s[i];
5                  s[i] = s[j];
6                  s[j] = c;
7          }
8  }
```

We can initialize persistent values in conditionals and loops by containing an initializing statement in parenthesis. That value now has scope outside the initializing statement:

```
1  if ((int *iptr = malloc(sizeof(int)) != nullptr){
2          printf("Malloc succeeded!\n");
3          *iptr = 42;
4  } else {
5          printf("Malloc failed!\n");
6  }
7  printf("Address of dynamic allocation = %p\n", iptr);
```

Jump Statements

Jump statements unconditionally transfer control flow.

Jump Statements

```
break ;
continue ;
return expression;
goto identifer ;
```

- `break` causes the closest enclosing `for`/`while`/`switch` to terminate.
- `continue` causes the remaining portion of the closest enclosing `for`/`while` loop body to be skipped, and the next iteration to begin.
- `return` causes a function to return control flow to its call site, and contains an optional expression (the *return value*)
- `goto` transfers control unconditionally. Use this sparingly as it can make program structure difficult to understand

Putting it Together

What is the output for this snippet?

```
1  for(int n = 0; n <= 5; n++){
2          switch(n) {
3                  case 0 : puts("0 ");
4                           break;
5                  case 1 : puts("1 ");
6                  case 2 : puts("2 ");
7                           break;
8                  case 3 : puts("3 ");
9                           break;
10                 default: n % 2 == 0 ?
11                                          puts("even and bigger than 3") :
12                                          puts("odd and bigger than 3");
13         }
14 }
```

Putting it Together

What is the output for this snippet?

```
1  for (int x = 0; x < 5; x++) {
2          for (int y = 0; y < 5; y++) {
3              if (y == 3)
4                continue;
5              printf("(%d, %d) ", x, y);
6      }
7  }
```

Putting it Together

```
1   #include <stdio.h>
2
3   int main(void){
4   loop:
5           for (int d = 3; d <= 0; d--){
6                   if (d == 0) goto div0;
7                   else printf("d = %d, 6 / d = %d\n, d, 6 / d);
8           }
9           printf("No exceptions encountered!\n");
10          goto finally;
11  div0:
12          printf("Divide by zero exception encountered!\n");
13          return 1;
14  finally:
15          return 0;
16  }
```

Classwork: Prime Numbers

Write a file **prime.c** that prints all prime numbers up to 100, with a space between each number, and ending with a new line.

- You can determine if a number is prime by dividing it by all numbers less than it except 1, and verifying the remainder is never 0. Remember that negative numbers, zero, and one are not prime.
- One solution uses nested for loops, and modulo to check every number less than the current number
- Once you find a lesser value that has a zero remainder, end the loop
- You can break out of a `for` loop using the keyword `break`