Principles of Computer Science II

Nadeem Abdul Hamid
CSC121A - Spring 2005

Lecture Slides 10 - Flow of Control

Review

- Tokens: basic syntactic units
  - Keywords, identifiers, constants, string constants, operators, punctuators
  - Separators: whitespace, operators, punctuators
- Rules of precedence and associativity
- Side-effect operators: ++ and --
- Assignment operators: =, +=, …
- Standard libraries and header files

  - Quiz: Is main a keyword?

Operators Affecting Flow of Control (most often used)

- Relational:
  - < less than
  - > greater than
  - <= less than equal
  - >= greater than equal
- Equality:
  - == equal to
  - != not equal to
- Logical:
  - ! (unary) negation
  - && logical and
  - | | logical or

C Representation of Truth

- True/false are implemented as integers in C
- False ==> 0
- True ==> anything else (i.e. any nonzero value)

```
if ( -5 ) printf("Hello");
if ( 0 ) printf("No way!");
```

- Relational, equality, and logical operators return 1 for true, 0 for false

Operator Precedence and Associativity

- (table - page 611)

Parsing Exercises

```
#include <stdio.h>

#define PRX         printf("x= %d\n", x)
#define PR(a)       printf("%d\n", a)
#define PR2(a,b)    printf("%d\t %d\n", a, b)
#define PR3(a,b,c)  printf("%d\t %d\t %d\n", a, b, c)


int main(void) {
  int w = 2, x, y, z = 2;
  x = y = z;          PR3(x, y, z);
  x = y == z;         PR3(x, y, z);
  x *= 3 + 2;         PRX;
  x = w *= y = z = 4;  PR2(w, x); PR2(y, z);

  x = 2, y = 1, z = 5;

  z += y += x ++;    PR3(x, y, z);
  z *= x += y --;    PR3(x, y, z);
  x = 5;
  z = x ++ - 1;      PR2(x, z);
  z = ++ x - 1;      PR2(x, z);
  x = 6, y = 7;
  z = x += y ++;     PR3(x, y, z);
  z = x ++ + ++ y;   PR3(x, y, z);
}
```

More Parsing

```
int w, x = 2, y = 1, z = 0;
w = x && y || z;            PR(w);
w = x || ! y || y && ! x;   PR(w);

x = 0, y = 1;
z = x ++ && y ++;           PR3(x, y, z);
z = ++ x || ++ y;           PR3(x, y, z);
w = ( x < y ? ++ x : ++ y); PR3(x, y, z);
x = y = z = 1;  ++ x && ++ y || ++ z;  PR3(x, y, z);
x = y = z = 1;  ++ x || ++ y && ++ z;  PR3(x, y, z);
x = y = z = -1; ++ x && ++ y || ++ z;  PR3(x, y, z);
```


Short-Circuit Evaluation

- With && and || operators, evaluation of the operands stops as soon as the outcome is known

  expr1 && *expr2*
  - If *expr1* is false (zero), *expr2* is not evaluated

  expr1 || *expr2*
  - If *expr1* is true (non-zero), *expr2* is not evaluated

- Watch out for side-effects in expressions

C Control Statements

```
a = b;      /* assignment statement */
a + b + c;  /* expression statement - no useful work done */
printf("%d\n", a);  /* function call */
```

- Compound statement
  - Series of declarations and statements surrounded by braces
  - *Block*: compound statement with declarations at the beginning
- Empty statement

  ```
  ;  /* an empty stmt */
  ```
- If and if-else statements

  ```
  if (expr) statement
  if (expr) statement1 else statement2
  ```
  - Watch for the dangling-else problem
- Loops

  ```
  while (expr) statement
  for (expr1; expr2; expr3) statement
  do statement while (expr);  /* notice semicolon */
  ```


Comma Operator

- Lowest precedence of all operators
  - Associates left-to-right
- *expr1*, *expr2*
  - Evaluates *expr1*, then *expr2*, returns value of *expr2* as result of entire expression
- Most commas are not operators

```
int main(void) {
  int sum, i, n = 10;

  /* sum integers from 1 to n */
  for ( sum = 0, i = 1; i <= n; ++i )
    sum += i;
  printf("Sum integers from 1 to %d = %d\n", n, sum);

  /* is this right? */
  for ( sum = 0, i = 1; i <= n; sum += i, ++i ) ;
  printf("Sum integers from 1 to %d = %d\n", n, sum);

  /* is this right? */
  for ( sum = 0, i = 1; i <= n; ++i, sum += i ) ;
  printf("Sum integers from 1 to %d = %d\n", n, sum);

  /* is this right? */
  for ( sum = 0, i = 1; sum += i, ++i <= n; ) ;
  printf("Sum integers from 1 to %d = %d\n", n, sum);

  return 0;
}
```


goto Statement

- Considered harmful
  - Primitive method of altering control flow
  - Causes unconditional jump to a labeled statement anywhere in the current function
- When to use it: never*

* In very, extremely, ultra-rare cases it can make a program significantly more efficient/greatly simplify flow of control

break and continue

- `break`: causes an exit from the innermost enclosing loop or `switch` statement

```
/* input validation using infinite loop and break */

#include <stdio.h>

int main(void) {
  int n;

  for ( n = 0;  ; ) {
    printf("Please enter a number between 1 and 10.\n");
    if ( scanf("%d", &n) == 1  &&  ( n > 0 && n <= 10 ) )
      break;
    printf("Input is out of range, try again.\n");
  }

  /* when control reaches here, the input is valid */
  printf("\nn=%d\n", n);

  return 0;
}
```

- `continue`: causes the current iteration of a loop to stop and next iteration to begin immediately

```
#include <stdio.h>

int main(void) {
  int sum, x;
  int cnt;

  sum = cnt = 0;
  printf("Enter 10 non-zero integers.\n");
  while ( cnt < 10 ) {
    int res = scanf("%d", &x);

    if ( res == 0 ) {  /* couldn't make conversion */
      printf("Please enter %d more non-zero integers.\n", 10-cnt);
      scanf("%*s");  /* eat up the unconvertible stuff */
      continue;      /* disregard zeros and bad input */
    }
    else if ( res < 0 ) {  /* input error? */
      printf("INPUT ERROR.\nPROGRAM TERMINATING.\n");
      return -1;
    }
    else if ( x == 0 ) {
      printf("Please enter %d more non-zero integers.\n", 10-cnt);
      continue;   /* disregard zeros and bad input */
    }

    sum += x;
    cnt++;
  }

  printf("\nsum = %d\n", sum);
  return 0;
}
```


Pythagorean Triples

- A set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for side1, side2, and hypotenuse - all no larger than 500.