C Course

IIT Kanpur

Lecture 3
Aug 31, 2008

Rishi Kumar <rishik>, Final year BT-MT, CSE

Recap

- Signed and Unsigned data types in C
  - Let's consider `signed` and `unsigned int` in C.
    C allocates 2 bytes(can vary from one compiler to another)
  - For `unsigned int`,
    All bits are used to represent the magnitude.
    Thus 0 to $2^{16} - 1$ can be represented.
  - For `signed int`,
    1 bit is reserved for sign. (0 for +ve and 1 for –ve)
    Thus +ve numbers range from 0 to $2^{15} - 1$
    For –ve numbers we use 2's complements.
    What's 2's complement?

Recap

- Signed and Unsigned data types in C

  - Let's consider `signed` and `unsigned int` in C.
    C allocates 2 bytes(can vary from one compiler to another)

  - For `unsigned int`,
    All bits are used to represent the magnitude.
    Thus 0 to $2^{16} - 1$ can be represented.

  - For `signed int`,
    1 bit is reserved for sign. (0 for +ve and 1 for –ve)
    Thus +ve numbers range from 0 to $2^{15} - 1$
    For –ve numbers we use 2's complements.
    What's 2's complement?
    In 2's complement to represent a –ve number (say -x) in n bits
    - Compute $2^n - x$. Represent this magnitude as unsigned int in n bits.
    - The range is 0 to $- 2^{15}$. How?

Logical Expressions

- Formed using
  - 4 relational operators: `<, <=, >, >=`
  - 2 equality operators: `==, !=`
  - 3 logical connectives: `&&, ||, !`
- `int` type: 1(true) or 0 (false)

- Some examples are
  - If x = 8, y = 3, z = 2 what is the value of

```
x >= 10 && y < 5 || z ==2
```

Logical Expressions

- Formed using
  - 4 relational operators: `<, <=, >, >=`
  - 2 equality operators: `==, !=`
  - 3 logical connectives: `&&, ||, !`
- `int` type: 1(true) or 0 (false)

- Some examples are
  - If x = 8, y = 3, z = 2 the value of

    `x >= 10 && y < 5 || z ==2` is 1.
  - Precedence comes into picture. Remember last lecture?

Conditional Operator [`?:`]

- A conditional expression is of the form

  `expr1 ? expr2 : expr3`

  The expressions can recursively be conditional expressions.

- A substitute for `if-else`
- Example :

  `(a<b)?((a<c)?a:c):((b<c)?b:c)`

  What does this expression evaluate to?

Conditional Operator [?:]

- A conditional expression is of the form

  `expr1 ? expr2 : expr3`

  The expressions can recursively be conditional expressions.

- A substitute for `if-else`
- Example :

```
(a<b)?((a<c)?a:c):((b<c)?b:c)
```

This evaluates to min(a,b,c)

`if-else` statement

- The syntax is
  - `if(expr) stmt`
  - `if(expr) stmt1 else stmt2`

  Note that `stmt, stmt1, stmt2` can either be simple or compound or control statements.
  - Simple statement is of the form `expr;`
  - Compound statement is of the form

```
{
   stmt1;
   stmt2;
   ……..
   stmtn;
}
```

  - Control Statement: will be discussed through this lecture. involves `if-else, for, switch`, etc

    e.g- `if(expr) stmt1 else stmt2`

if-else : some examples

- ```
  x = 1; y = 10;
  if(y < 0) if(y > 0) x = 3;
  else x = 5;
  printf("%d\n", x);
  ```

  What is the output here?

- ```
  if(z = y < 0) x = 10;
  printf("%d %d\n", x, z);
  ```

  What is the output here?

if-else : some examples

- ```
  x = 1; y =10;
  if(y < 0) if(y > 0) x = 3;
  else x = 5;
  printf("%d\n", x);
  ```

  Output is : 1

  Dangling `else`: `else` clause is always associated with the closest preceding unmatched `if`.

- ```
  if(z = y < 0) x = 10;
  printf("%d %d\n", x, z);
  ```

  The above code is equiv to the following one:

  ```
  z = y <0;
  if (z) x = 10;
  printf("%d %d\n", x ,z);
  ```

  Output is: 1 0

While and do-while

- Syntax is
  - `while(expr) stmt`
    - As long as `expr` is true, keep on executing the `stmt` in loop
  - `do stmt while(expr)`
    - Same as before, except that the `stmt` is executed at least once.
- Example:

```
int i=0, x=0 ;
while (i<10) {
        if(i%3==0) {
              x += i;
              printf("%d ", x);
          }
         ++i;
 }
```

What is the output here?

While and do-while

- Syntax is
  - `while(expr) stmt`
    - As long as `expr` is true, keep on executing the `stmt` in loop
  - `do stmt while(expr)`
    - Same as before, except that the `stmt` is executed at least once.
- Example:

```
int i=0, x=0 ;
while (i<10) {
        if(i%3==0) {
             x += i;
             printf("%d ", x);
          }
         ++i;
 }
```

Output is: 0 3 9 18

`for` statement

- Syntax is
  - `for(expr1; expr2; expr3) stmt`
    - expr1 is used to initialize some parameters
    - expr2 represents a condition that must be true for the loop to continue
    - expr3 is used to modify the values of some parameters.
  - It is equiv to

```
expr1;
while (expr2) {
        stmt
        expr3;
}
```

`for` statement

- This piece of code has equivalent `for` statement as follows:

```
expr1a;
expr1b;
while (expr2) {
        stmt
        expr3a;
        expr3b;
}
```

for (expr1a, expr1b; expr2; expr3a, expr3b) stmt

- Note that in the `for` statement `expr1, expr2, expr3` need not necessarily be present. If `expr2` is not there, then the loop will go forever.

`for` statement: some examples

```
int i, j, x;
for(i=0, x=0; i<5; ++i)
    for(j=0; j<i; ++j) {
        x += (i+j-1);
        printf("%d ", x);
    }
```

What is the output here?

`for` statement: some examples

```
int i, j, x;
for(i=0, x=0; i<5; ++i)
    for(j=0; j<i; ++j) {
        x += (i+j-1);
        printf("%d ", x);
    }
```

- Output is: `0 1 3 5 8 12 15 19 24 30`

`switch` statement

- Syntax is
  - `switch (expr) stmt`
  - `expr` must result in integer value; `char` can be used(ASCII integer value A-Z: 65-90, a-z: 97-122)
  - `stmt` specifies alternate courses of action
    - `case` prefixes identify different groups of alternatives.
    - Each group of alternatives has the syntax

```
case  expr:
        stmt1
        stmt2
        ……
       stmtn
```

Note that parentheses { } are not needed in `case` block

- Multiple `case` labels

```
case expr1:case expr2 :… … …: case exprn:
                              stmt1
                              stmt2
                              ……
                              stmtm
```

`switch` statement: example

```
switch (letter = getchar()) {
    case‘a’: case ’A’: case ‘e’ : case ‘E’:
    case‘i’: case ‘I’: case ’o’ : case ‘O’:
    case‘u’: case ‘U’:
                printf(“Vowel”); break;
    default:  printf(“Consonant”);
 }
```

- Note the use of multiple cases for one group of alternative. Also note the use of `default`. Statement corresponding to `default` is always executed.

  `break` to be discussed soon.

Power of `break`

- Syntax is
  - `break;`

- used to terminate loop or exit from a `switch`.

- In case of several nested `while`, `do-while`, `for` or `switch` statements, a `break` statement will cause a transfer of control out of the immediate enclosing statement.

`break` statement: Example

```
int count =0;
while (count <=n) {
    while( c=getchar()!='\n')
    {
        if ( c =='@') break;
        … … …
    }
    ++count;
}
```

`continue` statement

- Used to bypass the *remainder* of the *current pass* through a loop.
- Computation proceeds directly to the *next* pass through the loop.
- Example:

```
for( count=1; x <=100; ++count) {
   scanf ( "%f " , &x);
   if (x < 0) {
     printf(" it's a negative no\n")
     continue;
    }
  /*computation for non-negative
   numbers here*/
}
```

`goto` statement

- Note that you can *tag* any statement in C with an identifier.
- And then, can use `goto` to directly transfer the program control to that statement .
- Example:

```
while ( x <= 10) {
        … … …
        if (x<0)   goto chkErr;
        … … …
        scanf("%f", &x);
  }
chkErr: {
         printf("found a negative value!\n");
         … … …
   }
```

- Note that use of `goto` is discouraged. It encourages logic that skips all over the program . Difficult to track the code. Hard to debug.

Questions!!