601.220 Intermediate Programming

Control flow

Outline

- logical operators
- conditionals (`if` and `switch`)
- assignment and unary operators
- loops

Logical operators

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

https://www.tutorialspoint.com/cprogramming/c_operators.htm

Logical operators example

```c
// logical_op.c:
#include <stdio.h>
int main(void)
{
    int a = 5, b = 5, c = 10, result = 0;
    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) equals to %d \n", result);
    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) equals to %d \n", result);
    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) equals to %d \n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) equals to %d \n", result);
    result = !(a != b);
    printf("!(a != b) equals to %d \n", result);
    result = !(a == b);
    printf("!(a == b) equals to %d \n", result);
}
```

```
$ gcc logical_op.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
(a == b) && (c > b) equals to 1
(a == b) && (c < b) equals to 0
(a == b) || (c < b) equals to 1
(a != b) || (c < b) equals to 0
!(a != b) equals to 1
!(a == b) equals to 0
```

Conditionals (`if`)

- Suppose a represents some boolean expression (that is, a can be interpreted as having either value true or value false).

```
if (a) printf("a is true\n");
if (a) {
    printf("a is true\n");
}

if (a) {
    printf("a is true\n");
}
else {
    printf("a is false\n");
}

a ? printf("a is true\n") : printf("a is false\n");
```

Conditionals (switch)

```
switch (integer expr) {

case c1: stmt1;  // execution starting point for c1
case c2: stmt2;
         break;  // exits switch block
case c3:
case c4: stmt3;
         stmt4;  // executes stmt3, stmt4 and
                 // stmtlast for matches of c3 or c4
default: stmtlast;   // if no case matches

}
```

Switch statement example

```c
// switch_example.c:
#include <stdio.h>

int main () {
   char grade = 'B';
   switch(grade) {
      case 'A' :
         printf("Excellent!\n");
         break;
      case 'B' :
      case 'C' :
         printf("Well done\n");
         break;
      case 'D' :
         printf("You passed\n");
         break;
      case 'F' :
         printf("Better try again\n");
         break;
      default :
         printf("Invalid grade\n");
   }
   printf("Your grade is  %c\n", grade);
}
```

```
$ gcc switch_example.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Well done
Your grade is  B
```

Zoom poll!

What output is printed by the following C program if the user enters the input 42?

```c
#include <stdio.h>
int main(void) {
  int x;
  printf("Enter an integer: ");
  scanf("%d", &x);
  switch (x) {
    case 1:  printf("Ok\n");
             break;
    case 17: printf("Not bad\n");
             break;
    case 42: printf("Great!\n");
    default: printf("Huh?\n");
  }
  return 0;
}
```

A. `Ok`

B. `Not bad`

C. `Great!`

D. `Great!` followed by `Huh?`

E. `Huh?`

Compound assignments

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 3.11 | Arithmetic assignment operators.

Increment and decrement

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 3.12 | Increment and decrement operators

Loop summary

- `while(boolean expression) { statements }`
  - Iterates ≥ 0 times, as long as expression is true
- `do { statements } while(boolean expression);`
  - Iterates ≥ 1 times; always once, then more times as long as expression is true
- `for(initialize; boolean exp; update) { stmts }`
  - `initialize` happens first; usually declares & assigns "index variable"
  - Iterates ≥ 0 times, as long as boolean expression is true
  - Right after `stmts`, `update` is run; often it increments the index variable (`i++`)
- `break` immediately exits loop
- `continue` immediately proceeds to next iteration of loop

An example for loop

```c
// for_example.c:
#include <stdio.h>
int main(void) {
    for(int i = 0; i < 10; i++) {
        printf("%d ", i);
    }
}
```

A question for you: why using `i++` but not `++i`? Which one is preferred?

Zoom poll!

What output is printed by the following C program?

```c
#include <stdio.h>
int main(void) {
  for (int i = 1; i <= 10; i + 2) {
    printf("%d ", i);
  }
  return 0;
}
```

A. `2 4 6 8`

B. `2 4 6 8 10`

C. `1 3 5 7`

D. `1 3 5 7 9`

E. None of the above

A loop that reads in values until no more are available

```c
// sum.c:
#include <stdio.h>
int main(void) {
    int sum = 0;
    int addend;  //addend's value is undefined to start
    //read as many integers as we can
    while (scanf("%d", &addend) == 1) {
        //accumulate the sum of all numbers
        sum += addend;
    }
    //output the sum
    printf("%d\n", sum);
    return 0;
}
```

This contnues to scan even when you press enter. To signal end-of-input, press `Ctrl-D` (possibly twice).

Less desirable loop to read in input

```c
// sum_less_clean.c:
#include <stdio.h>
int main(void) {
    int sum = 0;
    while (1) {
        int addend = 0;
        if(scanf("%d", &addend) != 1) {
            break; // immediately exit loop
        }
        sum += addend;
    }
    printf("%d\n", sum);
    return 0;
}
```

The loop on the previous slide is preferred, since the loop body is cleaner. The code is more easy to follow, and less prone to errors.