Flow of Control
Flow of Control

- Sequential flow of control
  - Statements in a program are executed one after another in the order they placed in the program
Relational Operators

- relational_expression ::= expr < expr | expr > expr
  | expr <= expr | expr >= expr

- Two operands, the result is 0 or 1 (int)

- Precedence of the relational operators is less than that of the arithmetic operators

- Examples

```c
char  c = 'w';
int  i = 1, j = 2, k = -3;
double  x = 6e+22, y = 0.01;

'a' + 3 < c \equiv (\text{\char `'a' + 3}) < c =
- i - 6 * j >= k + 2
  \equiv ((- i) - (6 * j)) >= (k + 2)

2 < j < 6 \equiv (2 < j) < 6

x - 4.23 <= x + y \equiv (x - 4.23) <= (x + y)

x < x + y \equiv x < (x + y)
```
Equality Operators

- equality_expression ::= expr == expr | expr != expr

- Two operands, the result is 0 or 1 (int)

- Precedence of the equality operators is less than that of the arithmetic operators

- Examples

```
int i = 1, j = 2, k = 3;
i == j ≡ j == i
i != j ≡ j != i
i+j+k == -3 * -k
≡ ((i+j)+k) == ((-2)*(-k))
```
Logical Operators

- `logical_expression ::= logical_negation_expr
  | logical_or_expr
  | logical_and_expr`

- `logical_negation_expr ::= ! expr`
- `logical_or_expr ::= expr || expr`
- `logical_and_expr ::= expr && expr`
- The result is 0 or 1 (int)

| expr1 | expr2 | ! expr1 | expr1 && expr2 | expr1 || expr2 |
|-------|-------|---------|----------------|---------------|
| 0     | 0     | 1       | 0              | 0             |
| 0     | 1     | 1       | 0              | 1             |
| 1     | 0     | 0       | 0              | 1             |
| 1     | 1     | 0       | 1              | 1             |
Short-circuit Evaluation

- The evaluation process stops as soon as the outcome 0 or 1 is known
- `expr1 && expr2`
  - When `expr1` has value 0, then the evaluation of `expr2` will not occur
- `expr1 || expr2`
  - When `expr1` has value 1, then the evaluation of `expr2` will not occur
Compound Statement

- A series of declarations and statements surrounded by braces
  - A compound statement itself is a statement
  - A block
- `compound_statement ::= '{' { declaration }* { statement }* '}'`
- Examples
  ```
  {
    a = 1;
    {
      b = 2;
      c = 3;
    }
  }
  ```
Expression Statement

- expression_statement ::= { expr }opt ;
- Examples
  
  a = b;
  a + b;
  ;
  printf(“%d\n”,x);
If Statement

- **if** _statement_ ::= if ( _expr_ ) _statement_
  - If _expr_ is nonzero, then _statement_ is executed
  - Otherwise, _statement_ is skipped and control passes to the next _statement_

- **if** _else**_ _statement_ ::= if ( _expr_ ) _statement_
  else _statement_
  - Else attaches to the nearest **if**

**Examples**

```c
if (x == y)
    if (y == z)
        printf("%d\n", x);
else
    printf("%d\n", y);
```
while Statement

- `while_statement ::= while ( expr ) statement`
  1. `expr` is evaluated
  2. If it is nonzero, then `statement` is executed
  3. Control is passed back to the beginning of the `while` statement

- Statement is executed repeatedly until `expr` is zero

- Examples

```c
while ( i < n ) {
    f = f * i;
    i = i + 1;
}
```
for Statement

- `for_statement ::= for ( expr ; expr ; expr ) statement`
  1. The first `expr` is evaluated
  2. The second `expr` is evaluated
  3. If it is nonzero, then statement is executed
  4. The third `expr` is evaluated
  5. Control is passed back to the beginning of the while statement
     - The evaluation of the first `expr` is skipped

- Examples
  ```
  for ( i = 1; i <= n; i = i + 1 ) {
    f = f * i;
  }
  for ( i = 1; i <= n; i = i + 1 )
    f = f * i;
  ```
Comma Operator

- `comma_expression ::= expr , expr`
  1. The first `expr` is evaluated
  2. The second `expr` is evaluated
  3. It has the value and the type of the second `expr`

**Examples**

```c
for ( f=1, i=1; i <= n; i=i+1 ) {
    f = f * i;
}

for ( f=1, i=1; i <= n; i=i+1, f=f*i );

for ( f=1, i=1; i <= n; f=f*i, i=i+1 );
```
do Statement

- `do_statement ::= do statement while ( expr );`
  1. Statement is executed
  2. The `expr` is evaluated
  3. If it is nonzero, then control is passed back to the beginning of the `do` statement

Examples

```cpp
i = 0;
sum = 0;
do {
    sum = sum + i;
i = i + 1;
} while ( i > 0 );
```
Break Statement

- break;
- Causes an exit from the innermost enclosing loop or switch statement
- Examples

```c
while(1) {
    scanf("%lf", &x);
    if (x < 0.0)
        break;
    printf("%f\n", x*x);
}
```
continue Statement

- `continue;`
- Causes the current iteration of a loop to stop and causes the next iteration of the loop begin
- Examples

```c
while(1) {
    scanf("%lf", &x);
    if (x < 0.0)
        continue;
    printf("%f\n", x*x);
}
```
switch Statement

- switch_statement ::= switch ( integral_expression ) '{' 
  { case_statement | default_statement | switch_block } '}'

- case_statement ::= { case constant_integral_expression : }+ statement

- default_statement ::= default : statement

- switch_block ::= '{' { declaration_list }opt { case_group}* {default_group}opt '}'

- case_group ::= { case constant_integral_expression : }+ { statement }+

- default_group ::= default : { statement }+

1. The integral_expression is evaluated

2. Goto the caselabel having a constant value that matches the value, or if not found, go to the default label, or if there is no default label, terminate the switch statement

3. Terminate the switch statement when a break statement is encountered, or terminate the switch statement by falling off the end
switch Statement (contd.)

- **Examples**

```c
switch( c ) {
    case 'a':
    case 'A':
        a_cnt = a_cnt + 1;
        break;
    case 'b':
    case 'B':
        b_cnt = b_cnt + 1;
        break;
    default:
        other_cnt = other_cnt + 1;
}
```
Conditional Operator

- `conditional_expression ::= expr  ? expr : expr`
  1. The first `expr` is evaluated.
  2. If it is nonzero, then the second `expr` is evaluated. Otherwise the third `expr` is evaluated.
  3. The value and the type of the conditional expression as a whole are the value and the type of the second or the third `expr` evaluated.

Examples

```
x = ( y < z ) ? y : z;
if (y < z) x = y;
else x = z;
```
goto statement

- labeled_statement ::= { label : }+ statement
- label ::= identifier
- goto label ;
  - Control is unconditionally transferred to a labeled statement
- In general, goto should be avoided
- Examples

  goto error;
  ... 
  error: {
    printf("error\n");
    exit(1);
  }
getchar() and putchar()

- getchar() – gets a value from the keyboard and returns the value
- putchar(c) – the value of c is written to the standard output in the format of a character
- Both defined in stdio.h

```c
int c;
while ( ( c = getchar() ) != EOF ) {
    putchar( c );
}
```
int c;
while ( ( c = getchar() ) != EOF )
  if ( c >= 'a' && c <= 'z' )
      putchar( c + 'A' - 'a' );
  else
      putchar( c );
Increment and Decrement Operators

- ++, --
- Unary operators
  - ++a
    - Increment the stored value of a
    - The value of ++a is
  - a++
    - Increment the stored value of a
    - The value of a++ is the value before incrementing
- -- is similar to ++
Exercises

• The sequence of Fibonacci numbers is defined recursively by,
  \[ f_0 = 0, \ f_1 = 1, \ f_{n+1} = f_n + f_{n-1} \text{ for } n = 1, 2, \ldots \]

• Write a C program that computes a sequence of \( n \) Fibonacci numbers

<table>
<thead>
<tr>
<th>( n )</th>
<th>Fibonacci num</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
Exercises

- Write a program to check the proper pairing of parentheses