COMPSCI 210

Control Flow

Control Flow

- Specify order of computation
- Statements and Blocks
- Conditional
  - `if-else`
  - `else-if`
  - `switch`
- Loops
  - `while`
  - `for`
  - `do-while`
  - `break`
  - `continue`
  - `goto` and label

Statement and Blocks

- A statement is an expression followed by a ';' (statement terminator)
  - `a++` is an expression
  - `a++;` is a statement
- `{ }` (braces) group declarations and statements together into a compound statement, or a block
  - `main { … }`
  - `if { … }`

`if-else`

- Formal syntax

```
if (expression)
    statement1;
else
    statement2;
```

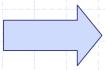
`else` part is optional

- `expression` evaluated
  - true (non zero): statement1 executed
  - false (zero): statement2 executed, if there is an `else` part

if-else

- Association in nested `if-else`

```
if (a>0)
if (a>b)
    y=a;
else
    y=b;
```



```
if (a>0)
    if (a>b)
        y=a;
    else
        y=b;
```

`else` associated with the closest `if` without `else`

```
if (a>0) {
    if (a>b)
        y=a;
}
else
    y=b;
```

use {} to change association

if-else

- Indentation may differ from what the compiler does

```
if (a>0)
    for (i=0;i<n;i++)
        if (a>b){
            printf("something…");
            y=a;
        }
else
    printf("something else");
```

`else` associated with the closest `if` without `else`

else-if

- Formal syntax

```
if (expression1)
    statement1;
else if (expression2)
    statement2;
else if (expression3)
    statement3;
…
else
    statementN;
```

`else` part is optional, gives default case, error checking

- `expressions` evaluated in order
  - false : evaluate next expression
  - true : execute associated statement, terminate

switch

- Formal syntax

```
switch (expression){
    case const-expr1: statements;
    case const-expr2: statements;
    …
    default: statements;
}
```

`case, default` can occur in any order

`default` is optional

- `expression` evaluated
  - `const-expr1` matches: statement1 executed
  - otherwise : statements associated with `default` is executed

switch

- allows several cases to be attached to a single set of actions
- use `break` to exit from `switch`
- beware of falling through to other cases

while

- Formal syntax

```
while (expression)
    statement;
```

- `expression` evaluated
  - true (non zero): statement executed, expression re-evaluated
  - false (zero): proceed to next statement

for

- Formal syntax

```
for (expr1; expr2; expr3)
    statement;
```

- `expr1` and `expr3` are assignments or function calls
- `expr2` is a relational expression
- any of `expr1`, `expr2` and `expr3` can be omitted

```
for (;;){
    ...
}
```

equivalent to infinite loop broken by `break` or `return`

for

- can be converted to `while` statement

```
for (expr1; expr2; expr3)
    statement;
```

is equivalent to

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

except for behaviour with `continue`

- use `for` when there is simple initialization and increment

do-while

- Formal syntax

```
do
    statement;
while (expression);
```

`statement` executed at least once

- `statement` is executed first
- `expression` evaluated *after* making each pass through the loop
  - true (non zero): statement executed, expression re-evaluated
  - false (zero): proceed to next statement

break

- useful for exiting loop without testing at the top or bottom
- break provides early exit from `for, while, do`
- also exit from `switch`

```
for (i=1;i<argc; i++){
    …
    if (num<0){
        …
        break;
        //jump out of loop
    }
    …
}
```

```
switch (expression){
    case const-expr1:
    statement1;
    break;
    case const-expr2:
    statement2;
    break;
    …
    default:
    statementN;
    break;
}
```

continue

- useful for continuing execution of loop without additional nesting

```
for (i=1;i<argc; i++){
     …
     if (num<0){
          printf("something…");
          continue;  //jump to next iteration of loop
     }
     //more code
     //do not need to be nested within the if-statement
     …
     …
}
```


goto and labels

- useful for jumping to a certain label location in program
- statements after label are executed

```
for (i=1;i<argc; i++){
     …
     if (num<0){
          …
          goto negativeInput;
          //jump to negativeInput
     }
     …    //code skipped
}

…         //code skipped
negativeInput:
…         //more code
```