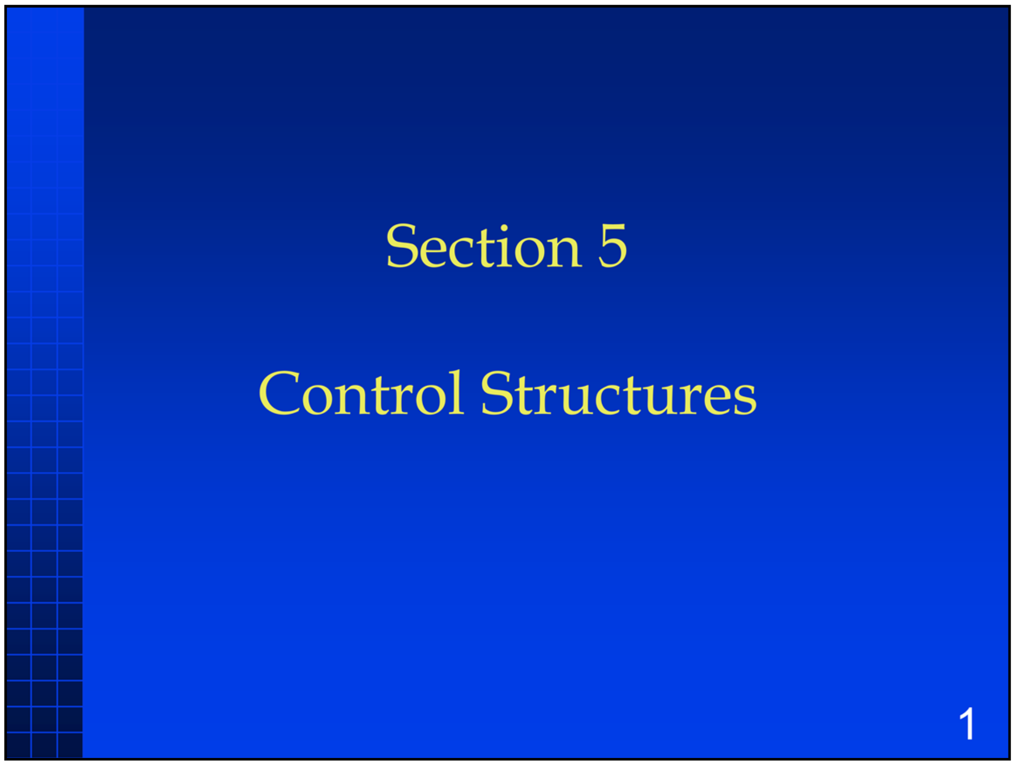
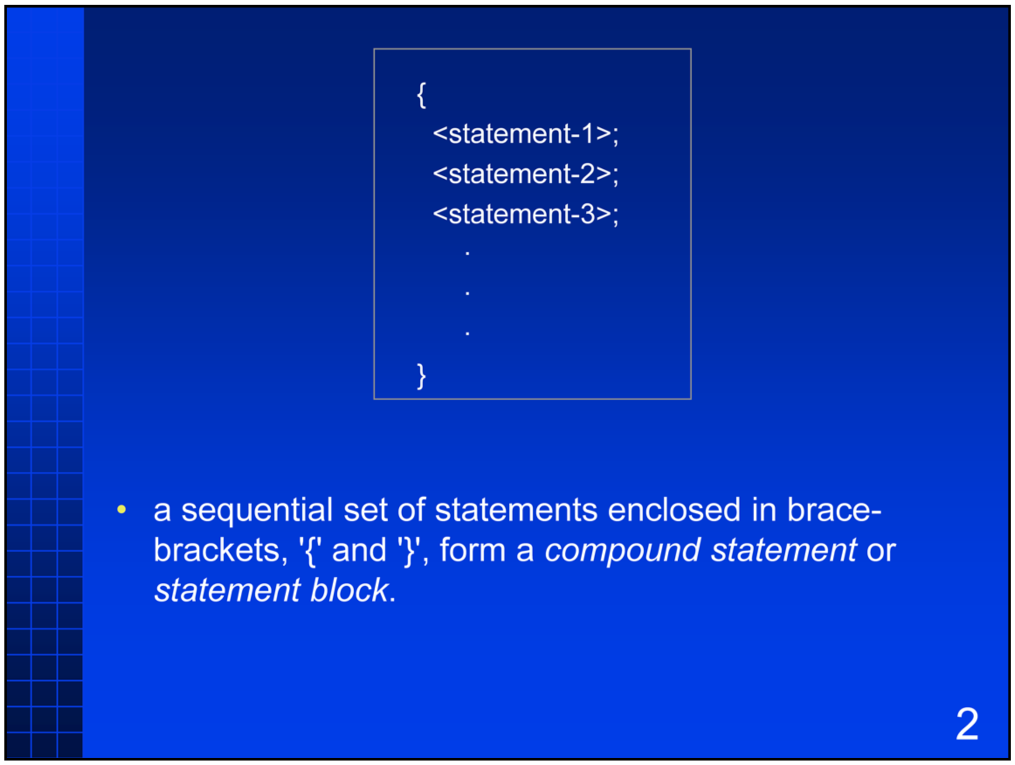


Section 5

Control Structures

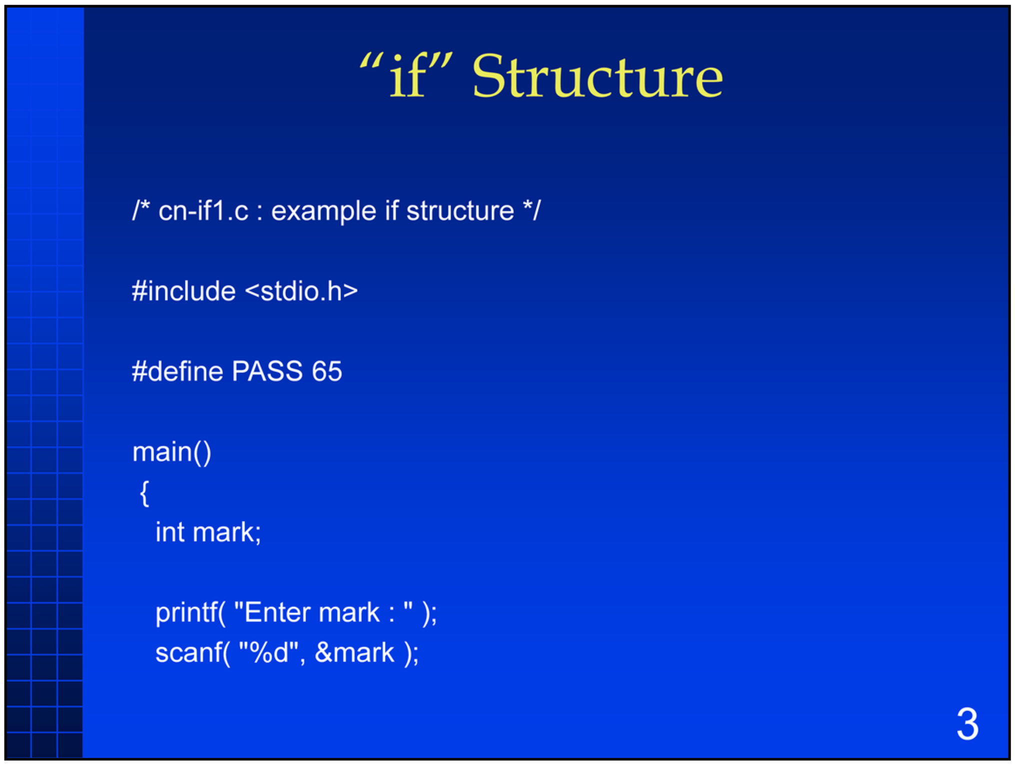
Control structures, control execution sequence of statements in program.



```
{
  <statement-1>;
  <statement-2>;
  <statement-3>;
      .
      .
      .
}
```

- a sequential set of statements enclosed in brace-brackets, '{' and '}', form a *compound statement* or *statement block*.

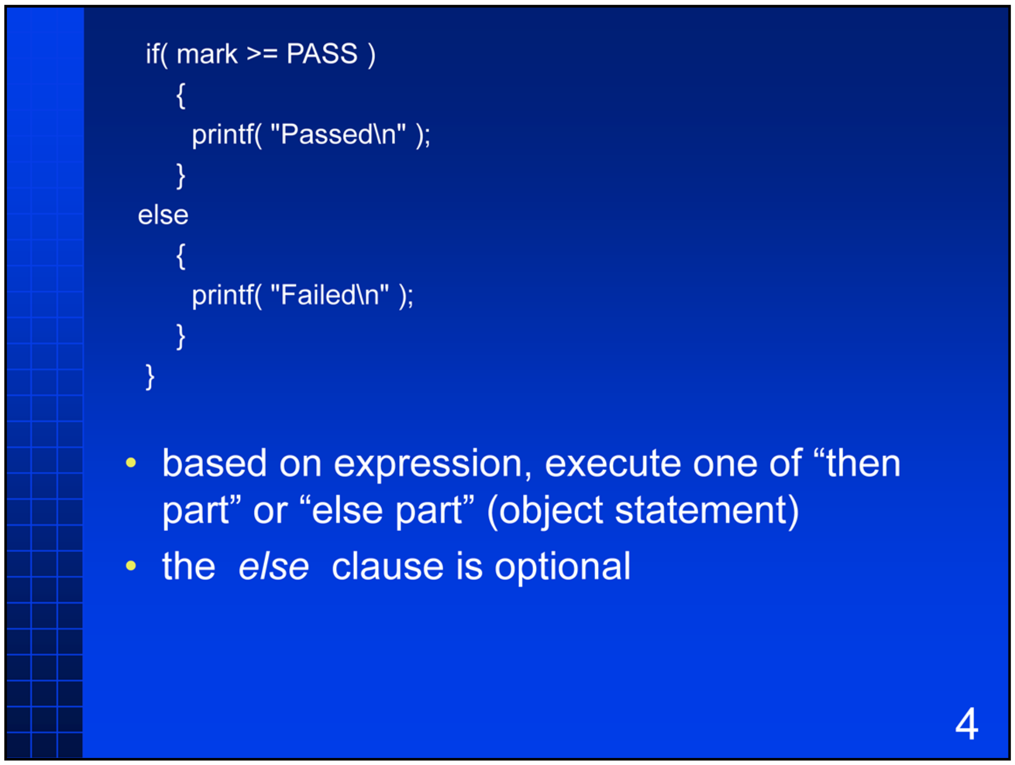
semi-colons after statements, never after braces



“if” Structure

```
/* cn-if1.c : example if structure */

#include <stdio.h>

#define PASS 65

main()
 {
  int mark;

  printf( "Enter mark : " );
  scanf( "%d", &mark );
```

if statement -- choose between two alternative actions called object statements

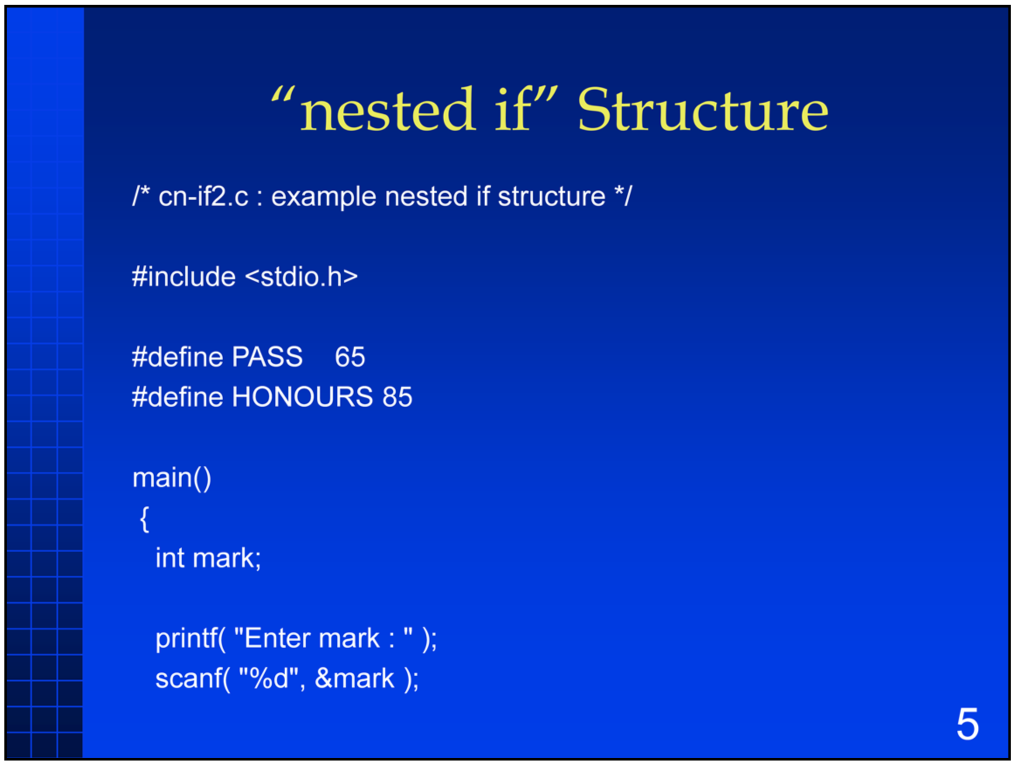


```
 if( mark >= PASS )
    {
      printf( "Passed\n" );
    }
else
    {
      printf( "Failed\n" );
    }
 }
```

- based on expression, execute one of “then part” or “else part” (object statement)
- the *else* clause is optional

object statements: then part and else part

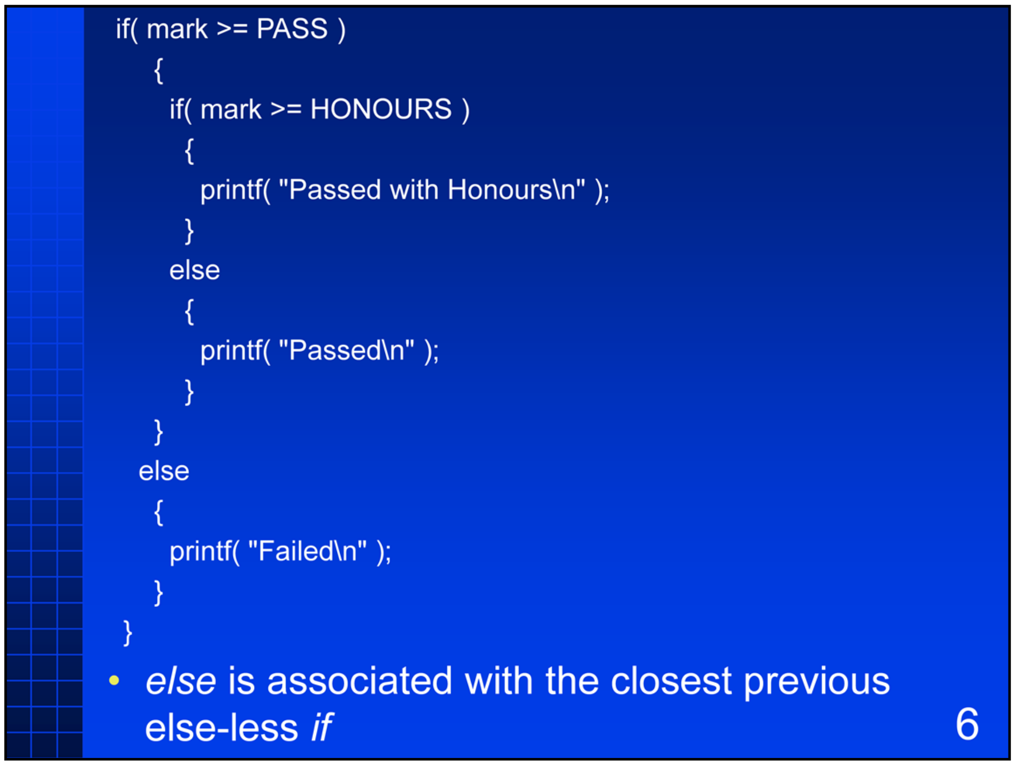
objects are single statements, use braces to form compound statement



“nested if” Structure

```
/* cn-if2.c : example nested if structure */

#include <stdio.h>

#define PASS    65
#define HONOURS 85

main()
 {
   int mark;

   printf( "Enter mark : " );
   scanf( "%d", &mark );
```

increase complexity, decisions within decisions represented by nested if statements



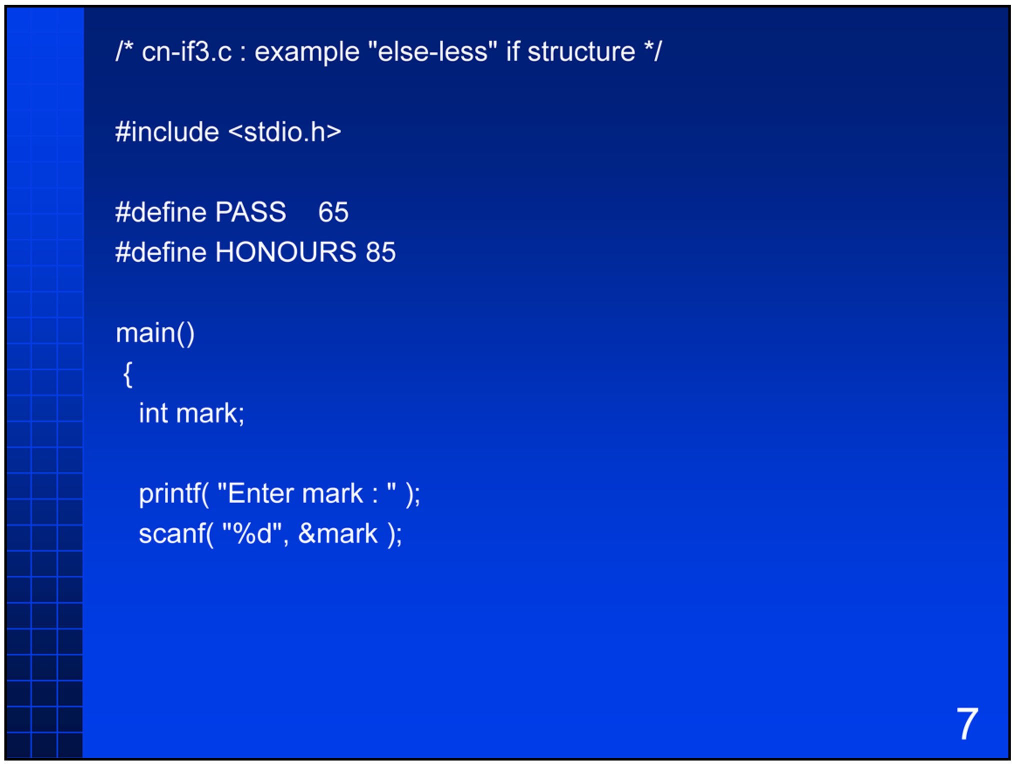
```
if( mark >= PASS )
   {
     if( mark >= HONOURS )
       {
         printf( "Passed with Honours\n" );
       }
     else
       {
         printf( "Passed\n" );
       }
   }
 else
   {
     printf( "Failed\n" );
   }
}
```

- *else* is associated with the closest previous else-less *if*

if-else associations cannot “cross” braces -- if-else must associate within same compound statement. (nesting level)

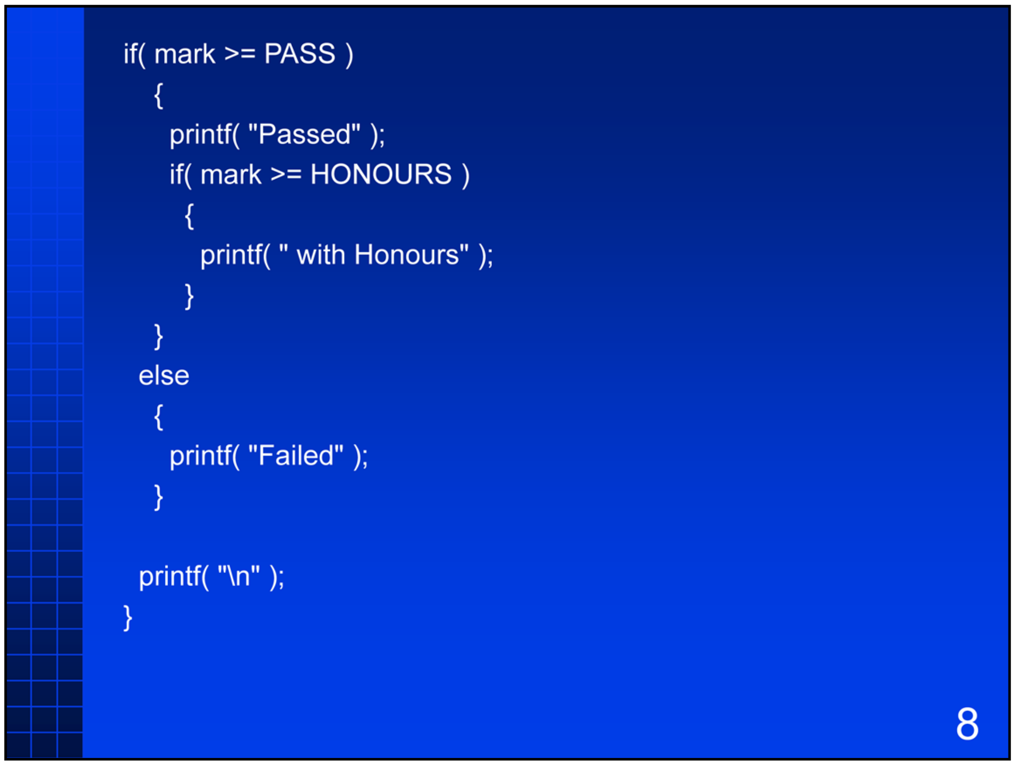
Eg: get rid of first else; 2nd else doesn’t associate with 2nd if because wrong level

Illustration following



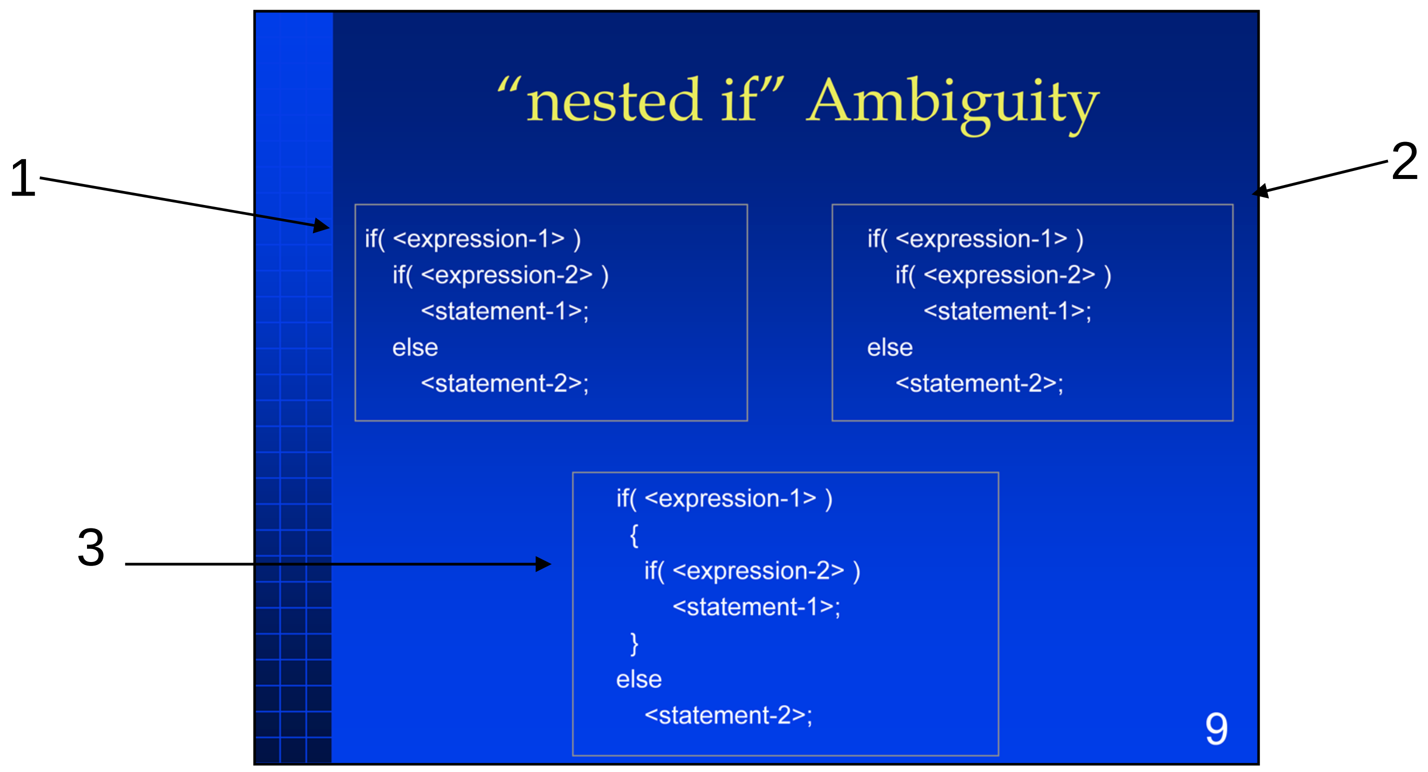
```
/* cn-if3.c : example "else-less" if structure */

#include <stdio.h>

#define PASS    65
#define HONOURS 85

main()
 {
   int mark;

   printf( "Enter mark : " );
   scanf( "%d", &mark );
```

Braces are necessary to resolve ambiguitues.
classic problem in this style of language called “dangling else”



```
if( mark >= PASS )
  {
    printf( "Passed" );
    if( mark >= HONOURS )
     {
       printf( " with Honours" );
     }
  }
 else
  {
    printf( "Failed" );
  }

 printf( "\n" );
}
```

this illsustrates situation just mentioned -- which if does else go with?
use of braces makes it clear: don’t cross boundaries



"nested if" Ambiguity

```
if( <expression-1> )
    if( <expression-2> )
        <statement-1>;
    else
        <statement-2>;
```

```
if( <expression-1> )
    if( <expression-2> )
        <statement-1>;
else
    <statement-2>;
```

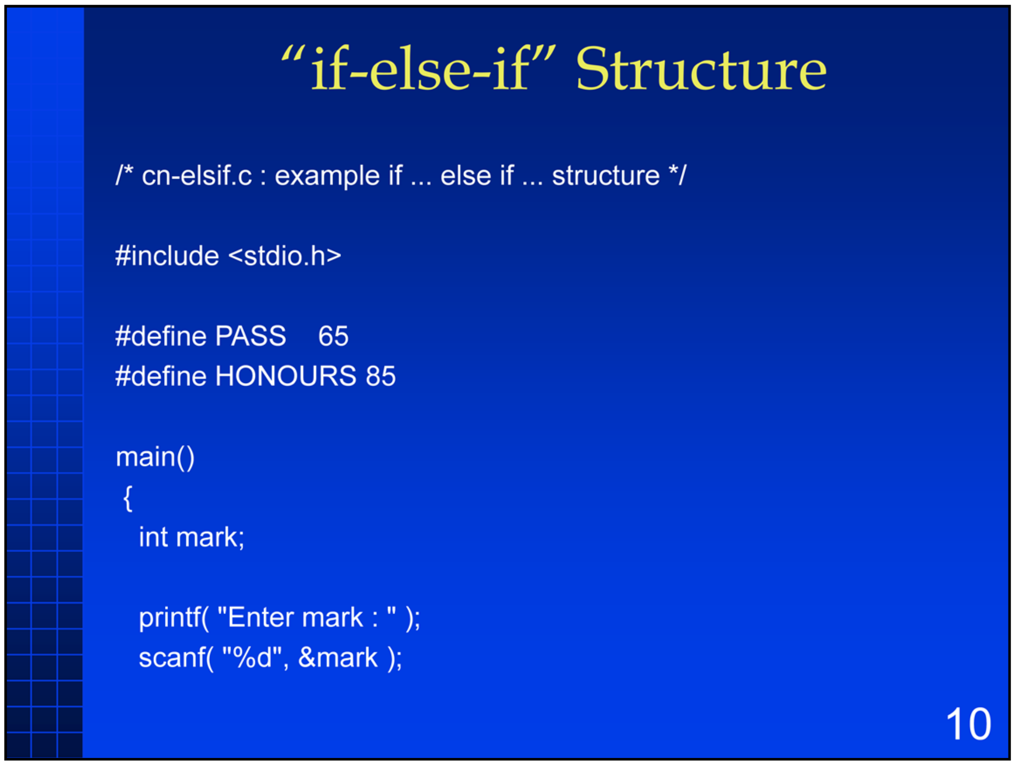
```
if( <expression-1> )
  {
    if( <expression-2> )
        <statement-1>;
  }
else
    <statement-2>;
```

9

1) correct structure (no braces) for nesting: else goes with closest if

2) misleading indentation: still no braces present, so association is same as before

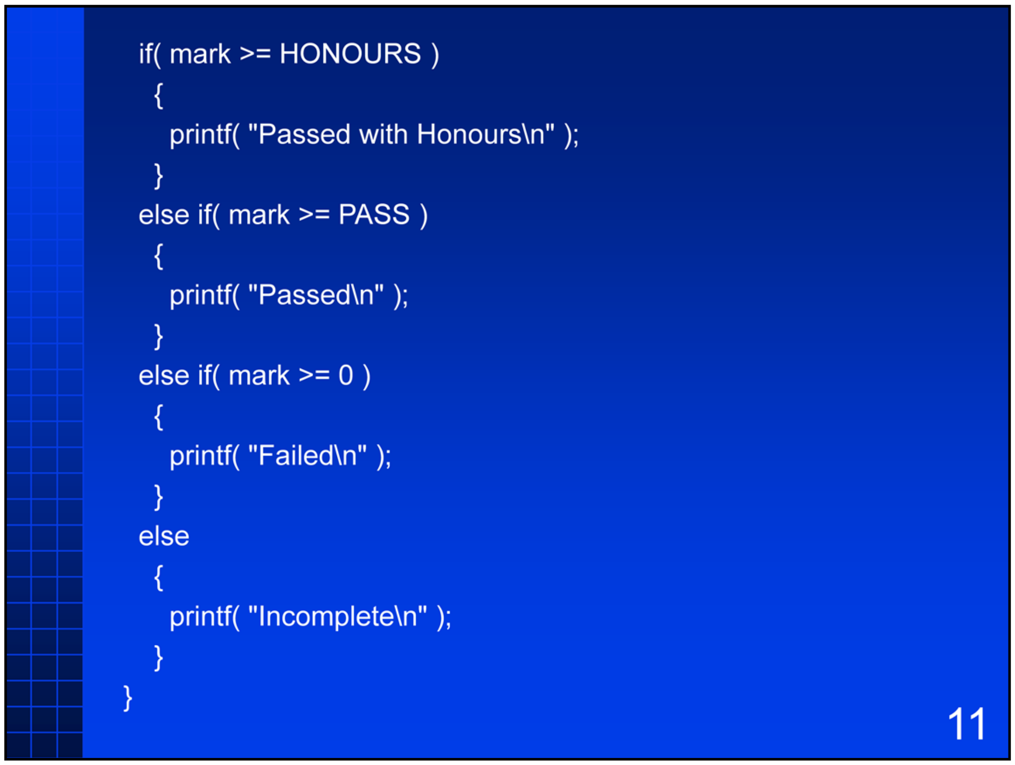
3) must have braces to associate else with outer if



“if-else-if” Structure

```
/* cn-elsif.c : example if ... else if ... structure */

#include <stdio.h>

#define PASS    65
#define HONOURS 85

main()
 {
   int mark;

   printf( "Enter mark : " );
   scanf( "%d", &mark );
```

Choose one action from a set of actions



```
if( mark >= HONOURS )
  {
    printf( "Passed with Honours\n" );
  }
else if( mark >= PASS )
  {
    printf( "Passed\n" );
  }
else if( mark >= 0 )
  {
    printf( "Failed\n" );
  }
else
  {
    printf( "Incomplete\n" );
  }
}
```

- C has no explicit “elseif”, so use cascading if-else-if; presentation is very stylized, but functionally equivalent.

True form:

```
if( blah )
 {
   asdfasdf
 }
else
 {
   if( blah )
    {
      asdfasdf
    }
   else
    {
      if( blah )
       {
         asdfasdf
       }
    }
 }
```

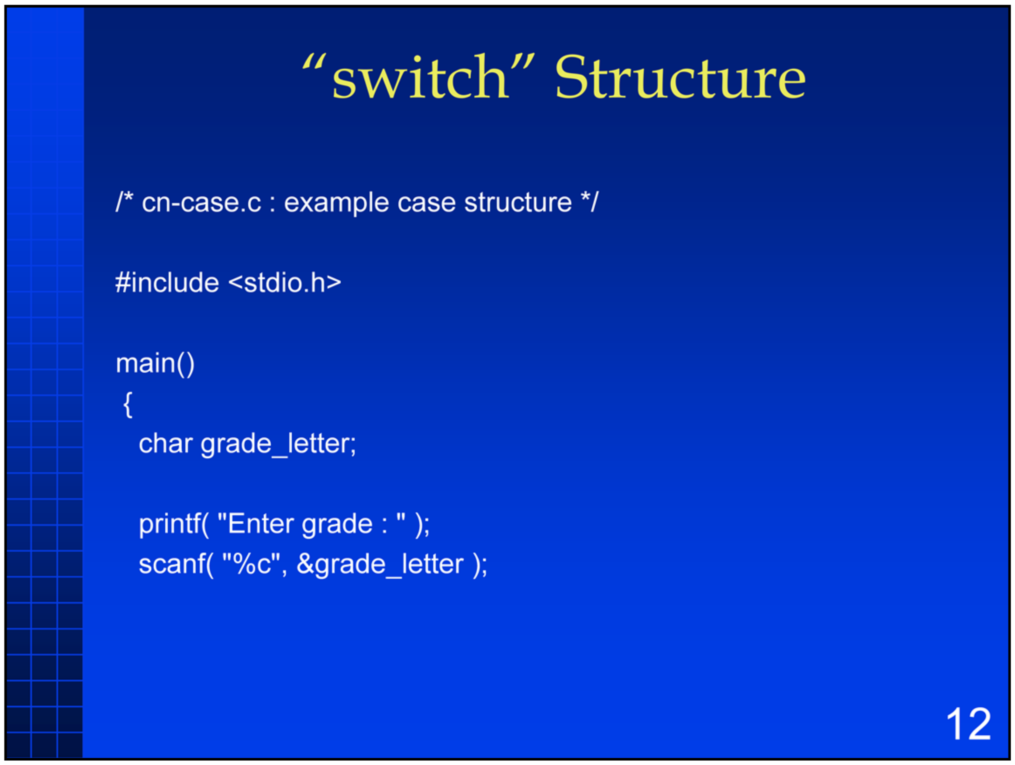
eliminate brace preceding if:

```
if( blah )
 {
   asdfasdf
 }
else
   if( blah )
    {
      asdfasdf
    }
   else
      if( blah )
       {
         asdfasdf
       }
```

rearrange indendation and spacing

```
if( blah )
 {
   asdfasdf
 }
else if( blah )
 {
    asdfasdf
 }
else if( blah )
 {
   asdfasdf
 }
```

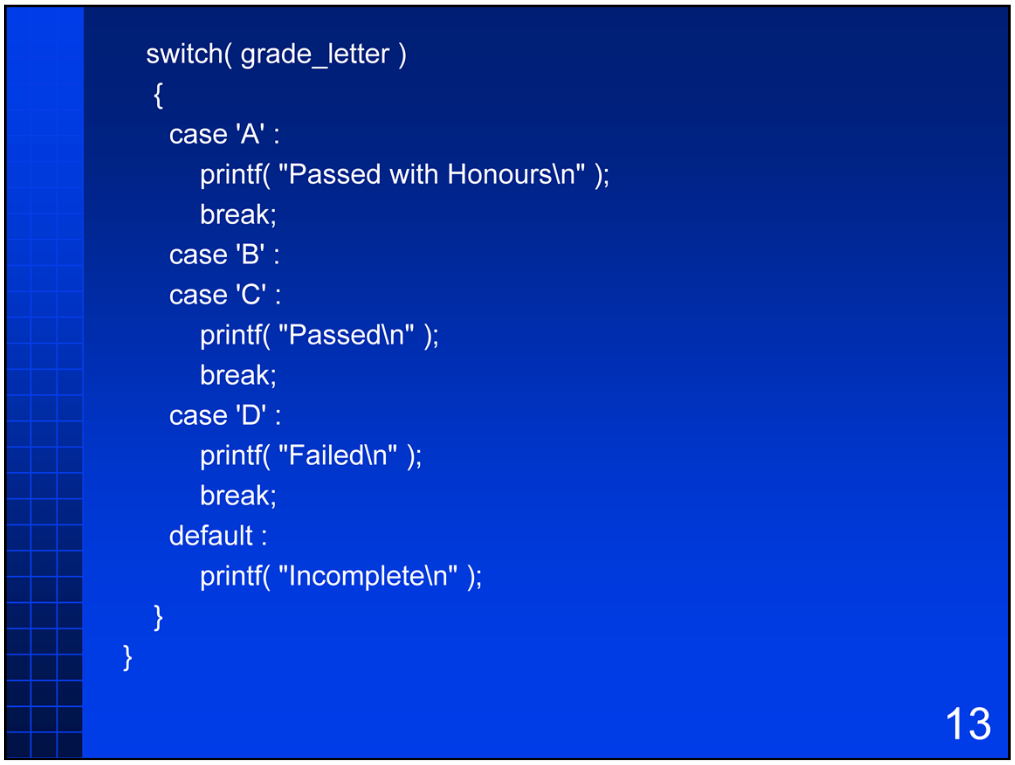
- always uses braces to avoid problems



"switch" Structure

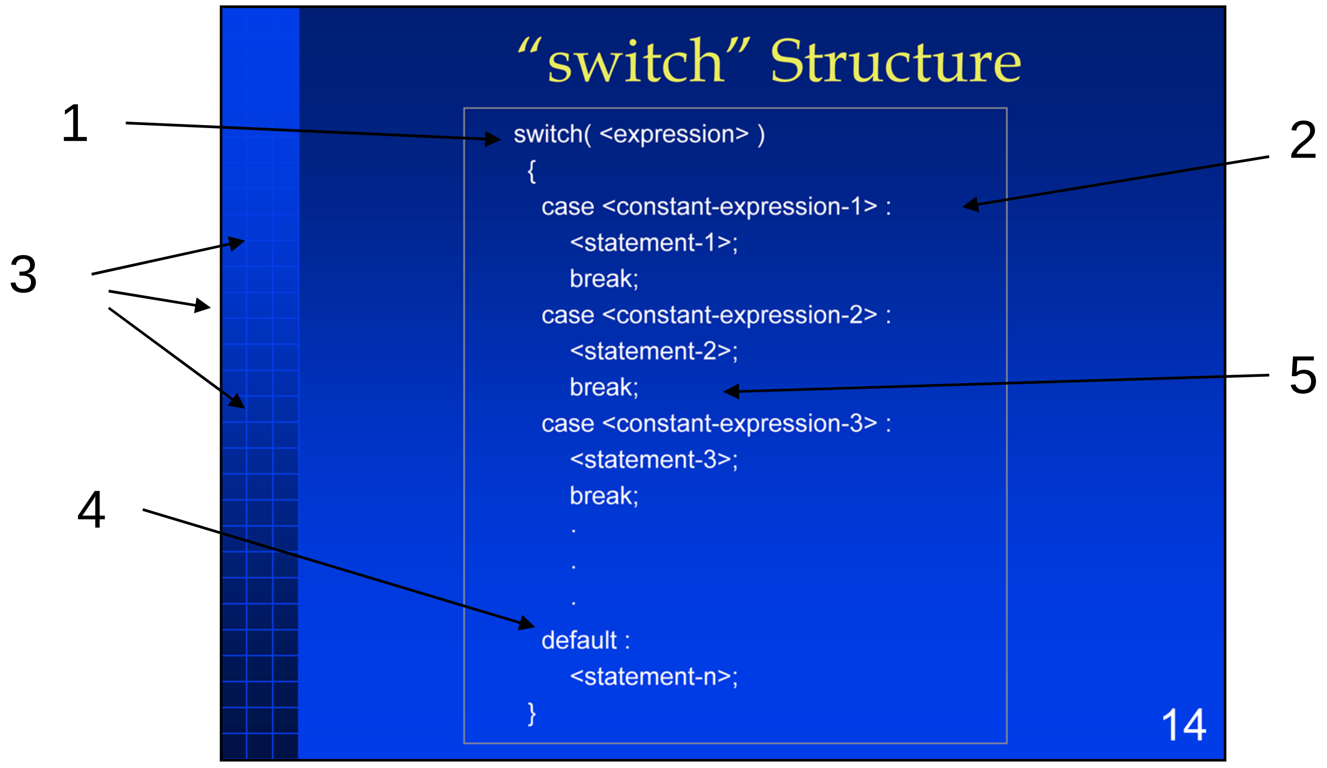
```
/* cn-case.c : example case structure */

#include <stdio.h>

main()
 {
   char grade_letter;

   printf( "Enter grade : " );
   scanf( "%c", &grade_letter );
```

if-else-if is very common, explicit statement for handling situation called “switch”, analogous to “case” statement in other languages



```
    switch( grade_letter )
     {
       case 'A' :
           printf( "Passed with Honours\n" );
           break;
       case 'B' :
       case 'C' :
           printf( "Passed\n" );
           break;
       case 'D' :
           printf( "Failed\n" );
           break;
       default :
           printf( "Incomplete\n" );
     }
}
```

an example of a switch statement
mechanics: evaluate expression, find matching case label.
execute statements sequentially, if break found, goto end of switch
if none found goto default (aka otherwise)



"switch" Structure

```
switch( <expression> )
 {
   case <constant-expression-1> :
       <statement-1>;
       break;
   case <constant-expression-2> :
       <statement-2>;
       break;
   case <constant-expression-3> :
       <statement-3>;
       break;
       .
       .
       .
   default :
       <statement-n>;
 }
```

more formally:

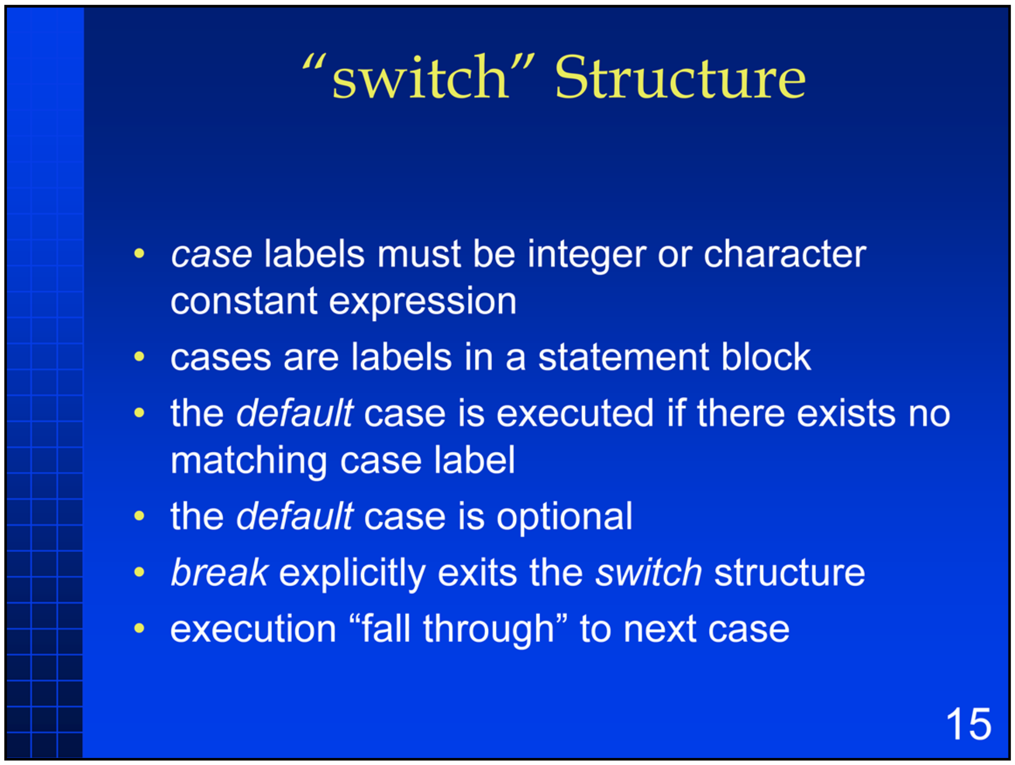
1) expression evaluates to one of the case labels

2) case labels must evaluate to constants complie-time. only one permitted (but note fall-though technique)

3) switch object statement is a statement block (sequence), control transfers to indicated label (like a goto); execution proceeds from that point forward. braces not required within cases, since not object statements

4) if no match, go to default case. if no default, do nothing

5) break transfers control to end of switch object. if no break, "fall through" to next case. Comes from "computed goto" and "label in statement sequence" idea. Allows multiple labels per action



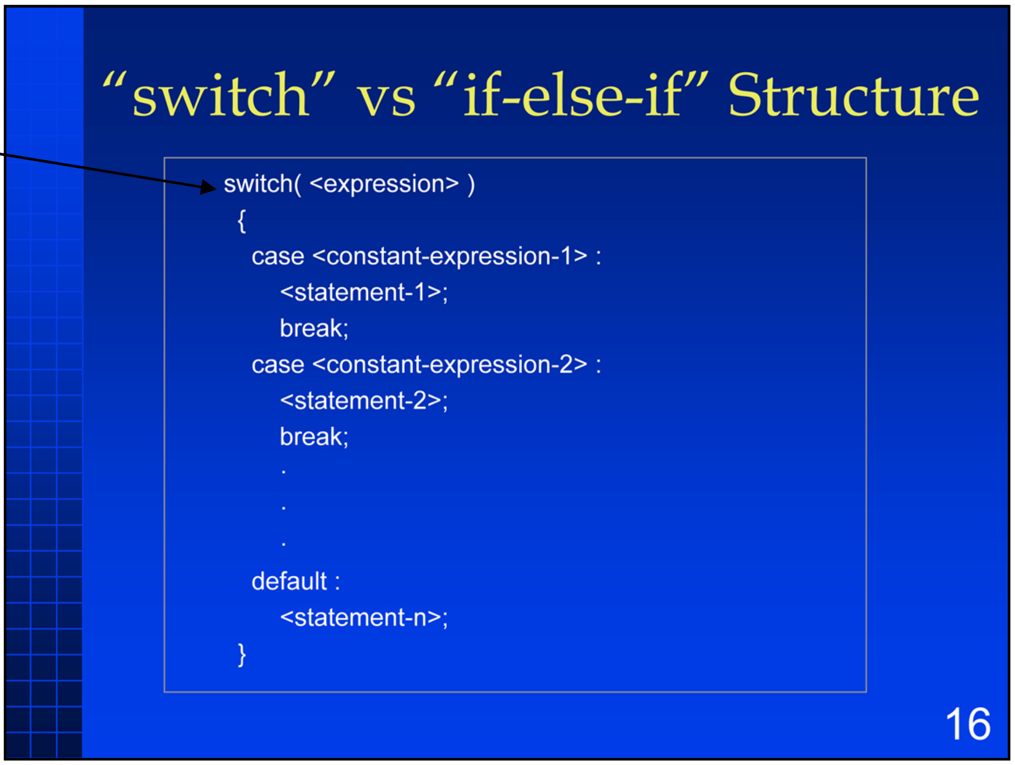
“switch” Structure

- *case* labels must be integer or character constant expression
- cases are labels in a statement block
- the *default* case is executed if there exists no matching case label
- the *default* case is optional
- *break* explicitly exits the *switch* structure
- execution “fall through” to next case

15

summary, as noted

1



“switch” vs “if-else-if” Structure

```
switch( <expression> )
 {
   case <constant-expression-1> :
       <statement-1>;
       break;
   case <constant-expression-2> :
       <statement-2>;
       break;
       .
       .
       .
   default :
       <statement-n>;
 }
```

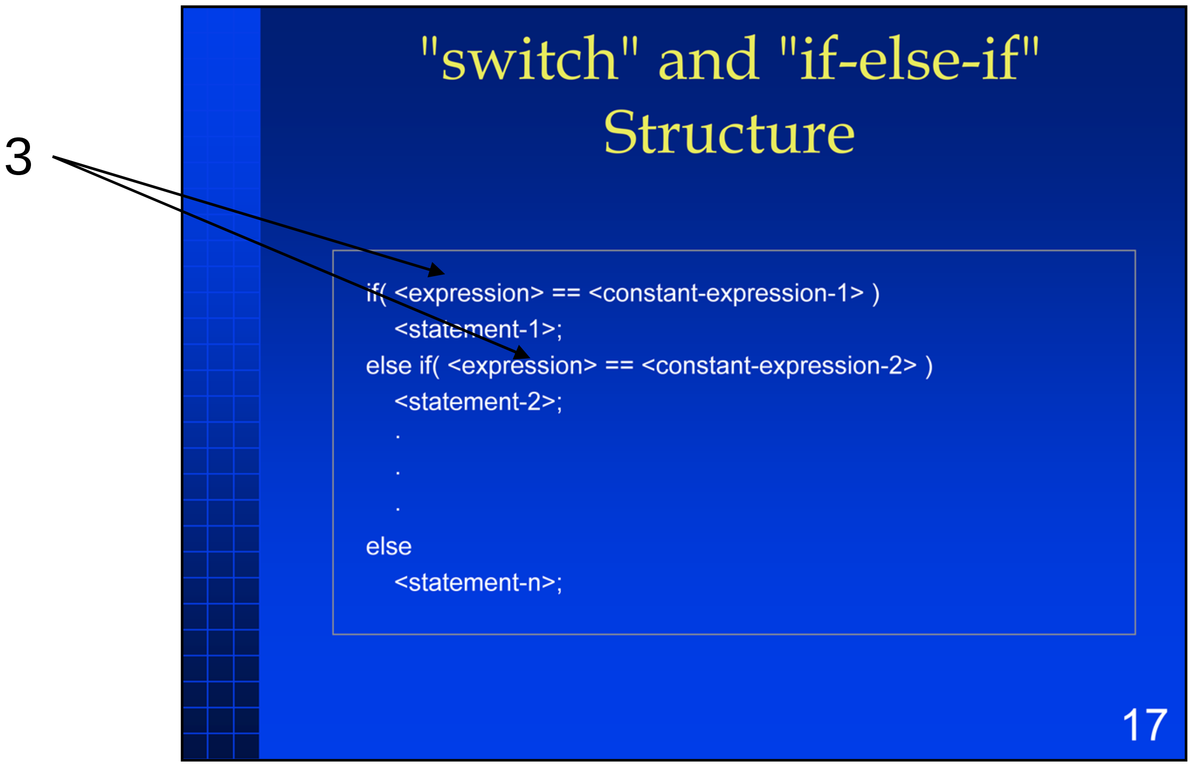
16

difference between switch and if-else-if, mostly style and taste. switch can sometimes express “choose one of” idea (dense cases)

some practical differences:

1) expression evaluated once

- some compilers may be able to generate branch table or other special optimization.
- duplicate cases easy to identify
- but: missing break can be pesky



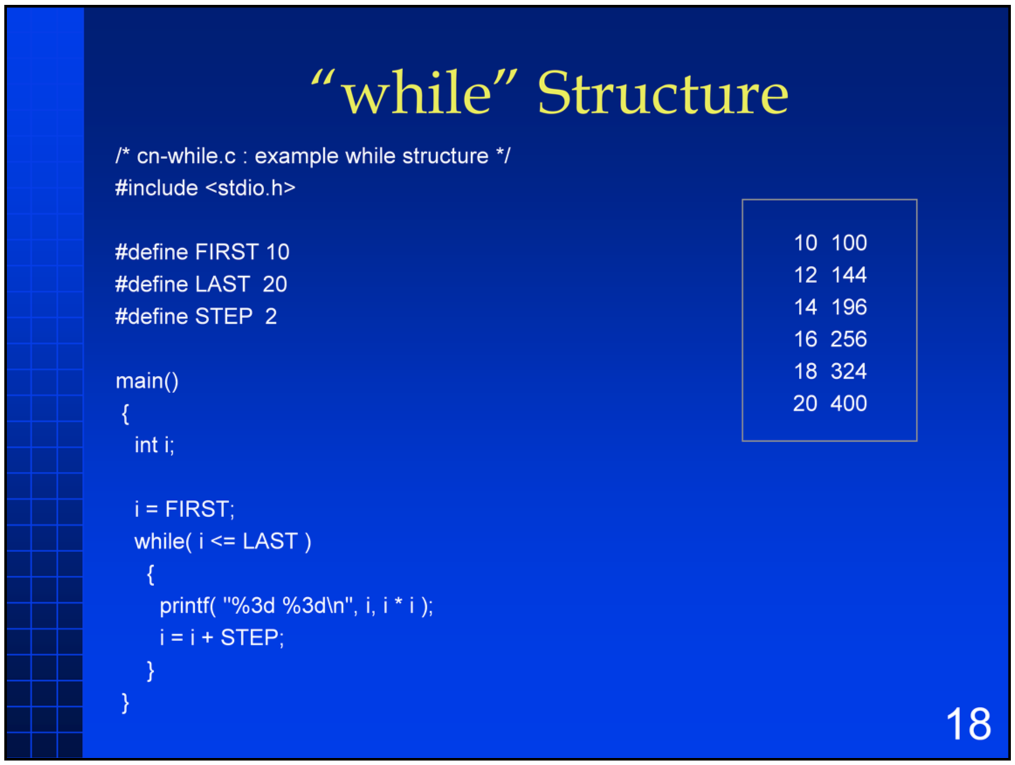
3

"switch" and "if-else-if" Structure

```
if( <expression> == <constant-expression-1> )
    <statement-1>;
else if( <expression> == <constant-expression-2> )
    <statement-2>;
    .
    .
    .
else
    <statement-n>;
```

3) expression evaluate lots (every time up until match) -- could be expensive, if optimizer cannot hoist common code.

- necessary if non-constant tests
- good for sparse cases
- not so good if same action in multiple places: requires “or” expressions that can become messy

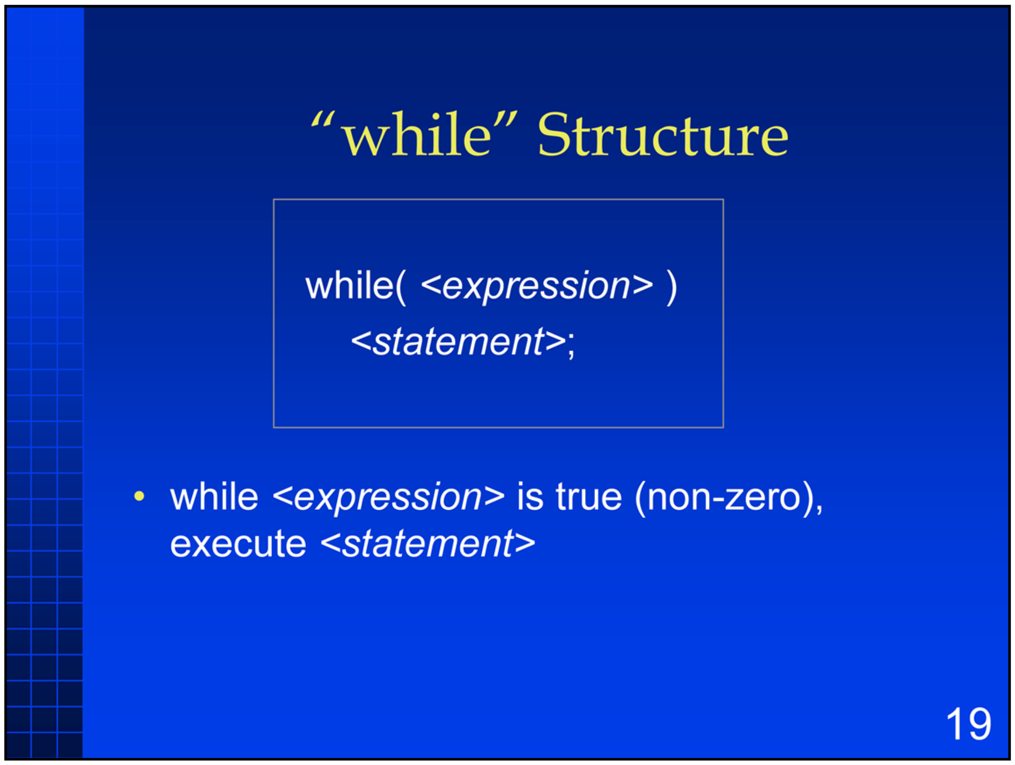


“while” Structure

```
/* cn-while.c : example while structure */
#include <stdio.h>

#define FIRST 10
#define LAST  20
#define STEP  2

main()
 {
   int i;

   i = FIRST;
   while( i <= LAST )
     {
       printf( "%3d %3d\n", i, i * i );
       i = i + STEP;
     }
 }
```

```
 10  100
 12  144
 14  196
 16  256
 18  324
 20  400
```

looping construct to repeat statement



“while” Structure

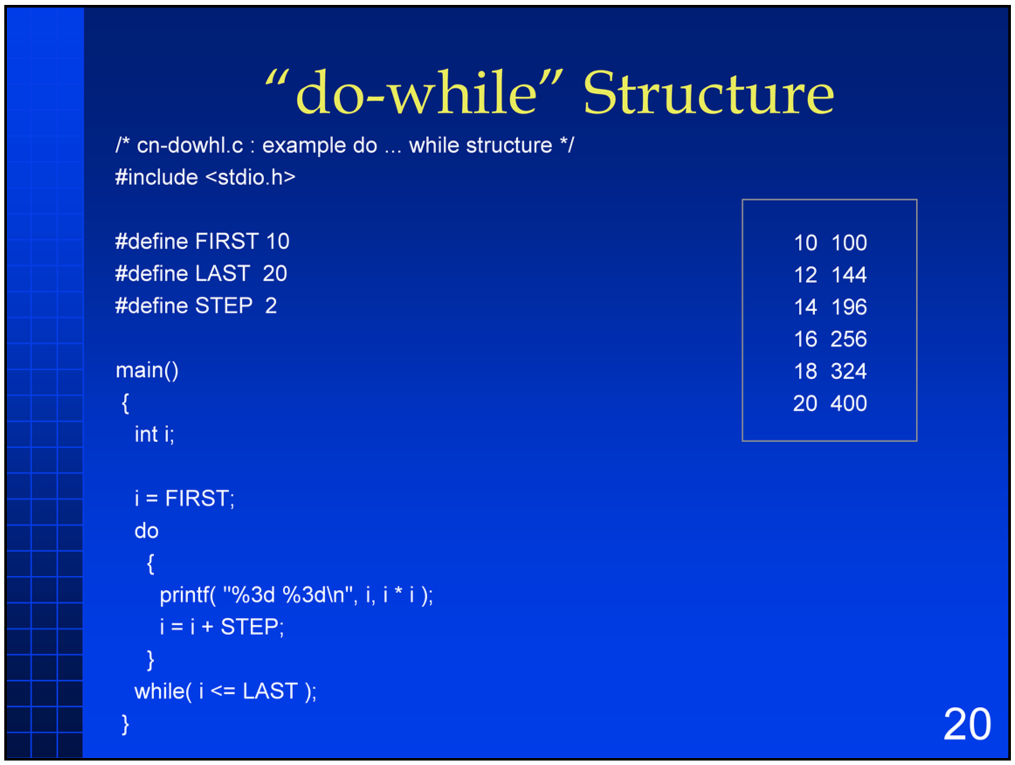
```
while( <expression> )
    <statement>;
```

- while <*expression*> is true (non-zero), execute <*statement*>

expression evaluates to false (zero) or true (non-zero)

statement can be a compound statement

if expression initially false, never executes statement



“do-while” Structure

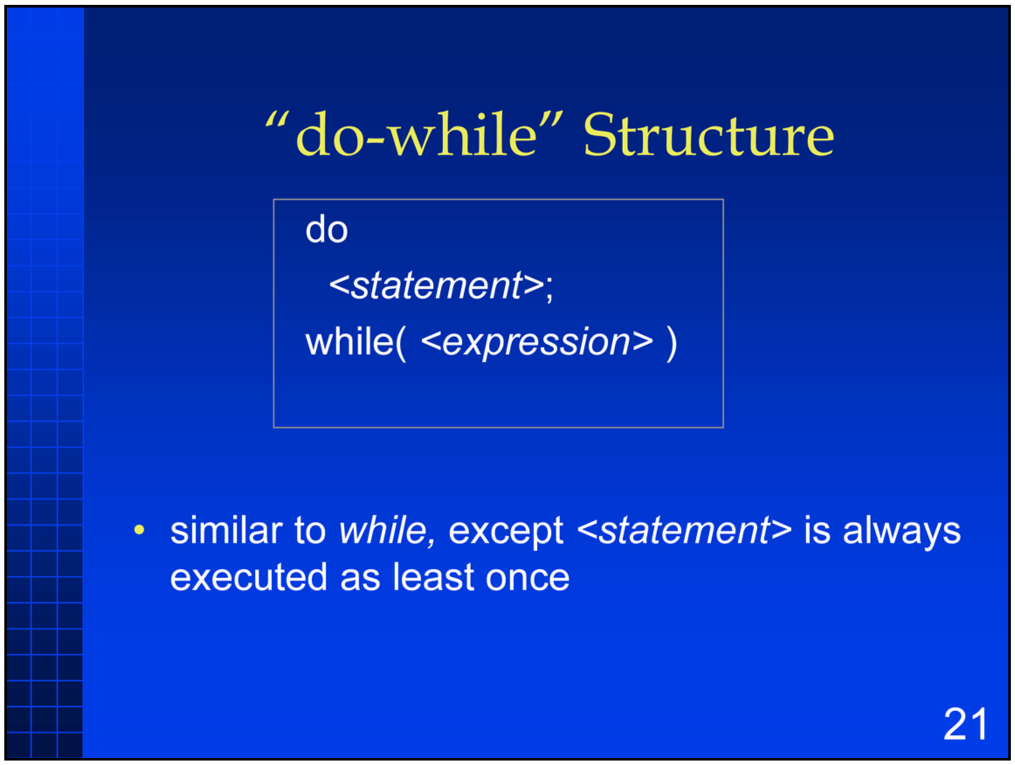
```
/* cn-dowhl.c : example do ... while structure */
#include <stdio.h>

#define FIRST 10
#define LAST  20
#define STEP  2

main()
 {
  int i;

  i = FIRST;
  do
   {
    printf( "%3d %3d\n", i, i * i );
    i = i + STEP;
   }
  while( i <= LAST );
 }
```

```
 10  100
 12  144
 14  196
 16  256
 18  324
 20  400
```

variation of while: do-while

upside-down while, iteration test at end of loop

loop object always executes at least once



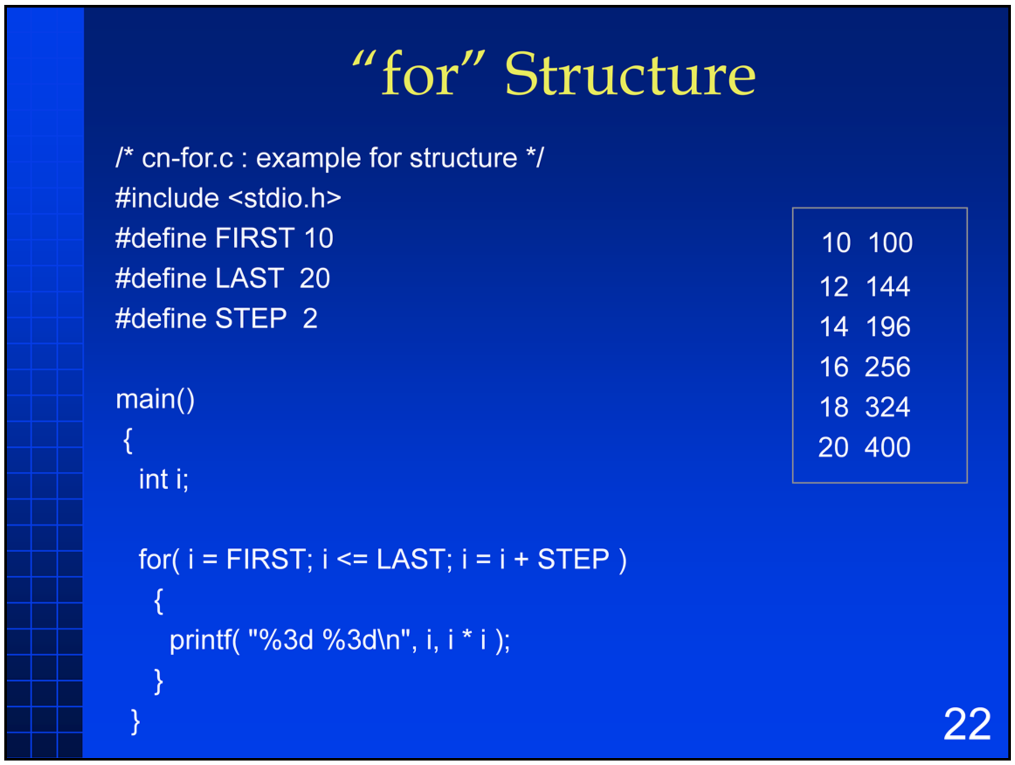
“do-while” Structure

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

- similar to *while,* except <*statement*> is always executed as least once

21

not much difference between the two, useful if computation of expression depends on execution of statement



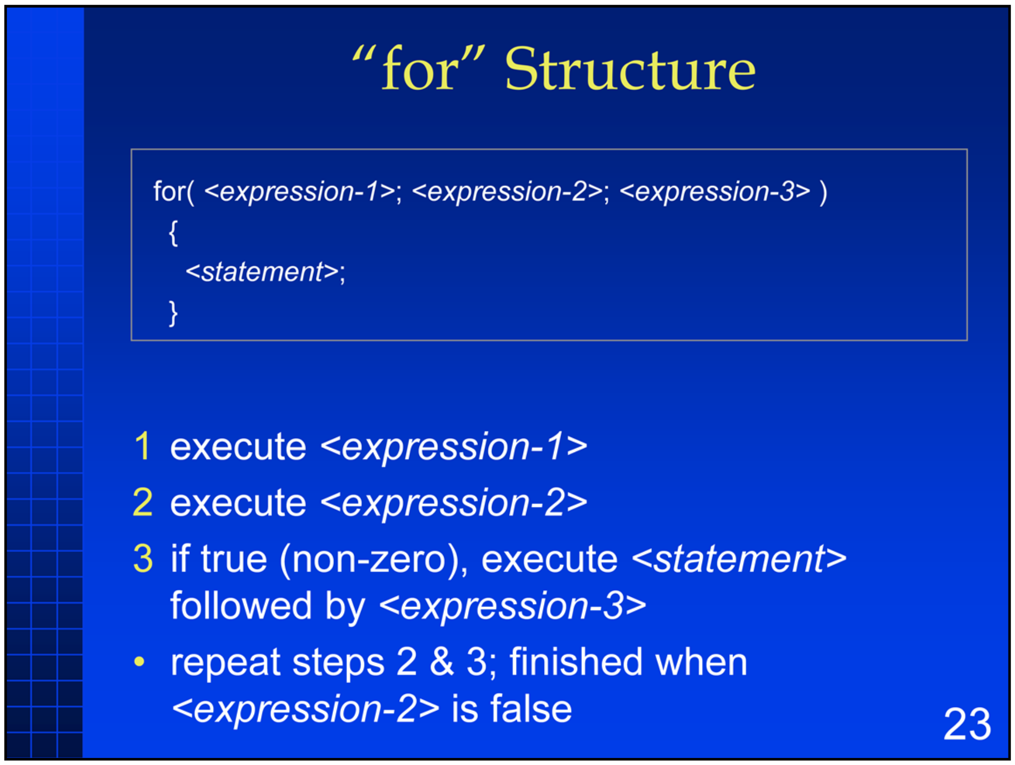
“for” Structure

```
/* cn-for.c : example for structure */
#include <stdio.h>
#define FIRST 10
#define LAST  20
#define STEP  2

main()
{
  int i;

  for( i = FIRST; i <= LAST; i = i + STEP )
   {
     printf( "%3d %3d\n", i, i * i );
   }
}
```

```
 10  100
 12  144
 14  196
 16  256
 18  324
 20  400
```

22

ordinary for statement as already seen -- nothing new



"for" Structure

```
for( <expression-1>; <expression-2>; <expression-3> )
  {
    <statement>;
  }
```

1 execute <*expression-1*>

2 execute <*expression-2*>

3 if true (non-zero), execute <*statement*> followed by <*expression-3*>

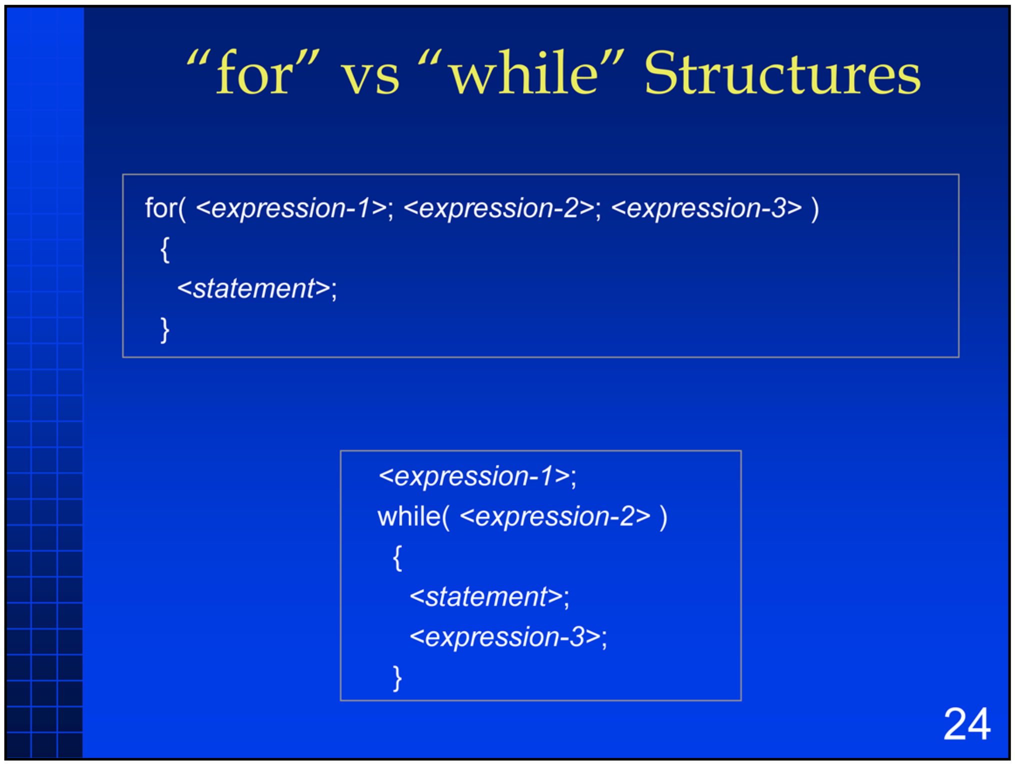
- repeat steps 2 & 3; finished when <*expression-2*> is false

three parts: initialization, loop control, incrementor

three steps: initialize, test termination, do statement and increment

not dependent on integral steps, no associated variables (eg read a file)

might never execute statement or expr3



“for” vs “while” Structures

```
for( <expression-1>; <expression-2>; <expression-3> )
 {
   <statement>;
 }
```

```
<expression-1>;
while( <expression-2> )
 {
   <statement>;
   <expression-3>;
 }
```

24

not much difference

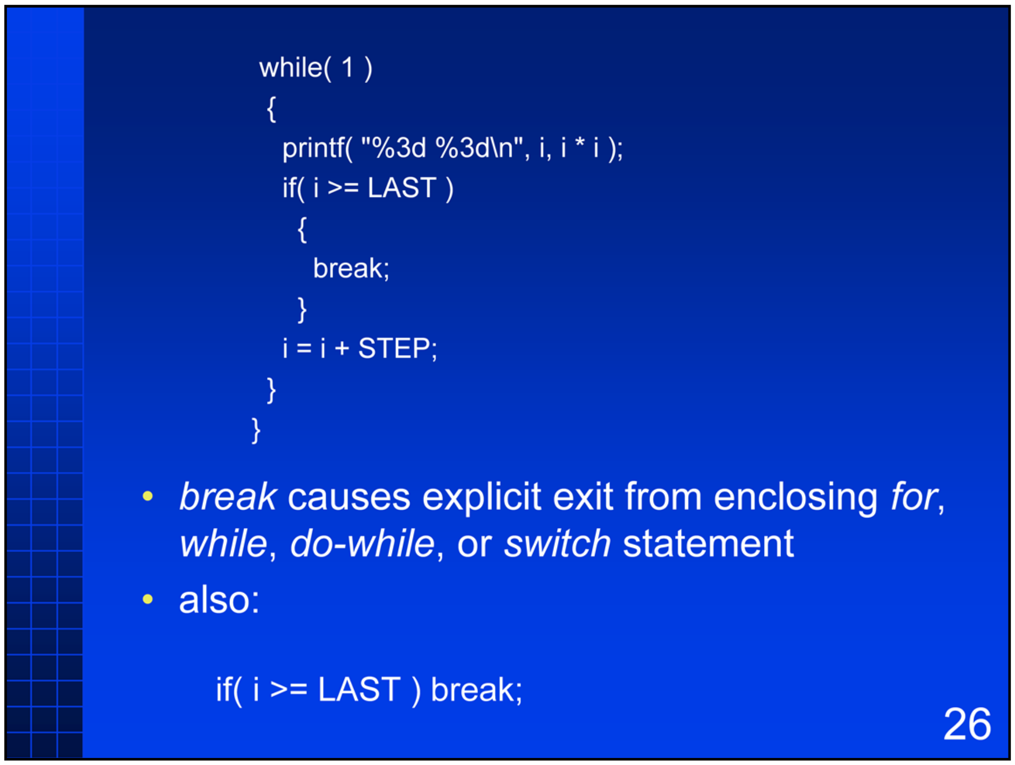
for stmt guarantees that incrementor will be done after the stmt; syntax make incrementor very explicit

while relies on user to implement incremenotr and put in proper place (nb continue statement)

“break” Statement

```
/* cn-break.c : example break statement */

#include <stdio.h>

#define FIRST 10
#define LAST  20
#define STEP  2

main()
 {
   int i;

   i = FIRST;
```

```
 10  100
 12  144
 14  196
 16  256
 18  324
 20  400
```

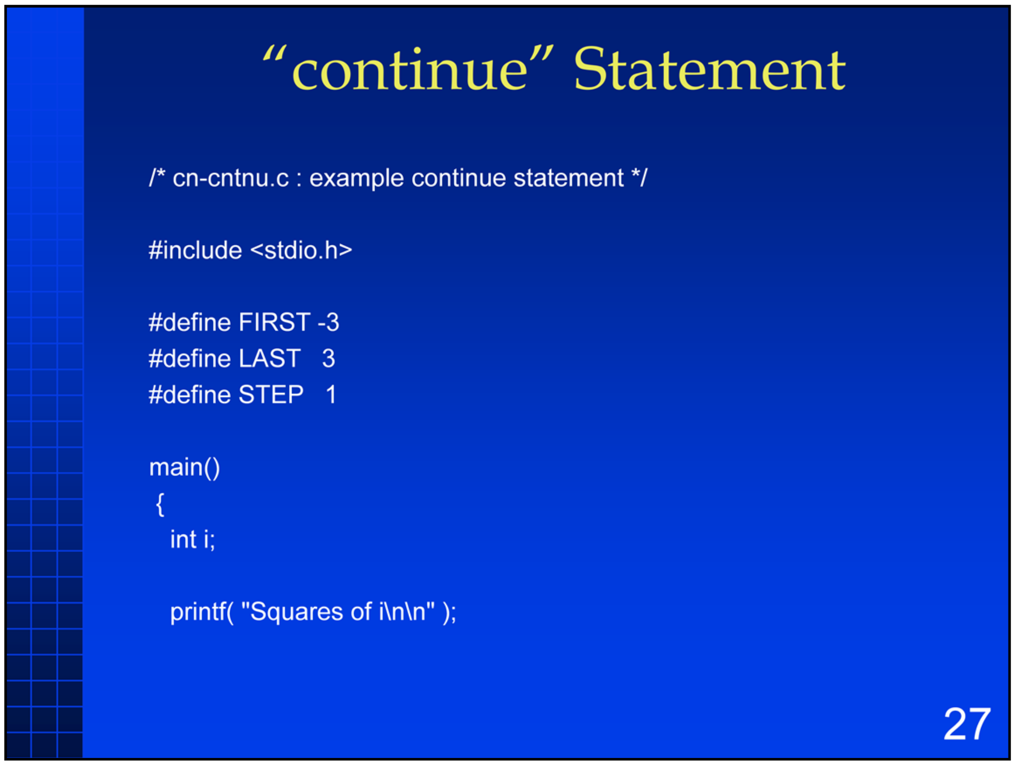


```
while( 1 )
 {
   printf( "%3d %3d\n", i, i * i );
   if( i >= LAST )
     {
       break;
     }
   i = i + STEP;
 }
}
```

- *break* causes explicit exit from enclosing *for*, *while*, *do-while*, or *switch* statement
- also:

```
if( i >= LAST ) break;
```

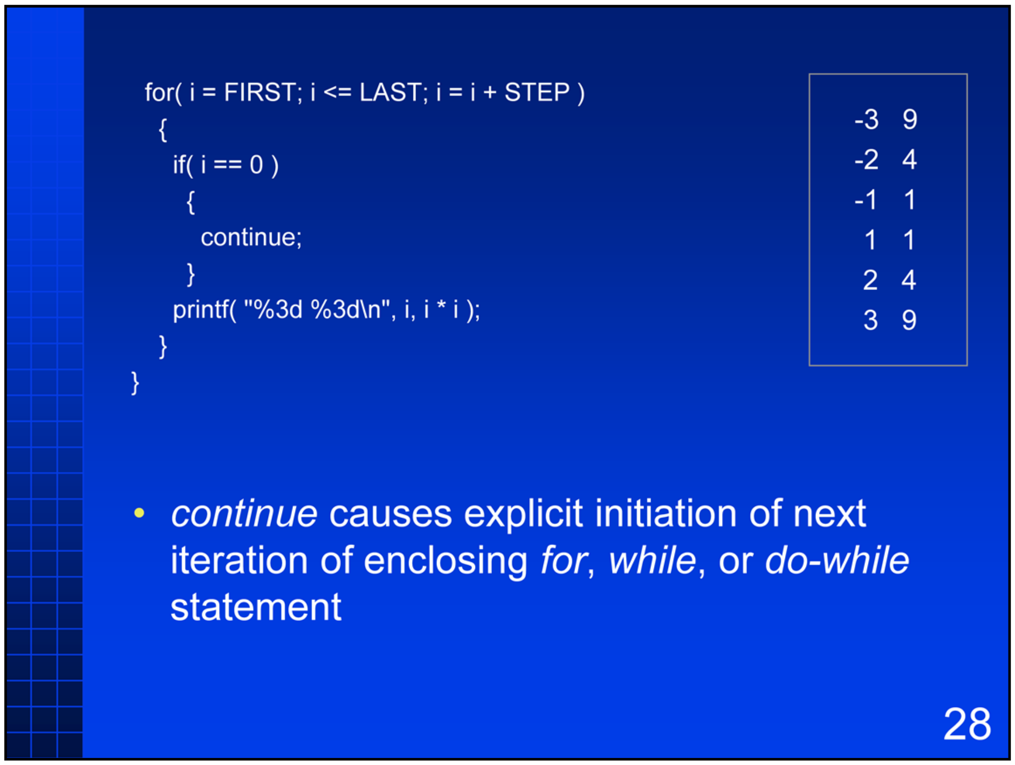
use break to get out of loops & switch
not if statement
one level at a time



"continue" Statement

```
/* cn-cntnu.c : example continue statement */

#include <stdio.h>

#define FIRST -3
#define LAST   3
#define STEP   1

main()
 {
   int i;

   printf( "Squares of i\n\n" );
```

continue: somewhat of a novelty?



```
for( i = FIRST; i <= LAST; i = i + STEP )
  {
    if( i == 0 )
      {
        continue;
      }
    printf( "%3d %3d\n", i, i * i );
  }
}
```

```
-3   9
-2   4
-1   1
 1   1
 2   4
 3   9
```

- *continue* causes explicit initiation of next iteration of enclosing *for*, *while*, or *do-while* statement

“go around again” for all looping structures (closest enclosing, no way to be explicit)

in for stmts, proceeds directly to for incrementor expression, then test

in while stmts, goes directly to top and tests: if incrementor not placed carefully, problem

ok for use with for, generally dangerous for others