CEng-140

Selective & Repetitive Structures

Today

- Relational (<, <=, >, >=, ==, !=)
- Logical Operators (&&, ||)
- Changing the flow of the program
  - Conditional statements
  - Conditional expressions

No Boolean type in C

- In C, there is no Boolean values!
  - Integers are used for representing truth
  - 0 value means **False**
  - Any **nonzero** value means **True**

Relational Operators & Expressions

- 6 rel operators for **comparing** values of expressions:

  < <= > >= == !=

  – Can be applied to any arithmetic operands
- Value of a relational expression is of type **int**
- If comparison is true, value of rel exp is **1**, otherwise **0**

Operator	Type	Associativity
+ - ++ --	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
= *= /= %= += -=	Binary	Right to left

Examples

- Semantics may not be clear at first sight:
- a = b + c <= d + e == c – d
  - Better use parentheses for clarification!
  - a = (((b + c) <= (d + e)) == (c – d))
- Semantics may not always be intuitive
  - 3 < 5 < 2 evaluates to ?
  - 5 < n < 10 evaluates to ?

Logical Operators & Expressions

- **&& || !**
  - Can be applied to any arithmetic operands
- Value of a logical expression is of type **int**
- Value of logical exp is **1** or **0,** depending on the logical value of the operands

Operator	Type	Associativity
+ - ++ -- !	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
&&	Binary	Left to right
||	Binary	Left to right
= *= /= %= += -=	Binary	Right to left

Logical AND (&&)

- Recall: Value of logical exp is **1** or **0,** depending on the logical value of the operands
- exp1 && exp2

exp1	exp2	exp1 && exp2
Non-zero (TRUE)	Non-zero (TRUE)	1 (TRUE)
Non-zero (TRUE)	Zero (FALSE)	0 (FALSE)
Zero (FALSE)	Non-zero (TRUE)	0 (FALSE)
Zero (FALSE)	Zero (FALSE)	0 (FALSE)

int a, b, c;

a = b = c = 10;

a && b + c ⇨ **evaluates to ?**

a && b – c ⇨ **evaluates to ?**

Logical NOT (!)

- !exp
- Evaluate *exp*,
- if it is 0 → value of logical expr is 1
- if it is non-zero → value of logical expr is 0

int a, b, c;

a = b = c = 10;

!a evalautes to ?

!(a-b) evalautes to ?

How do we determine the value of:

!a >= b && c / d

Logical Operators & Expressions

- Logical AND and OR operations are **always** evaluated **conditionally from left to right**
  - Called **short-circuited** (or, **based-on-need)** evaluation

    a = b = c = 10;

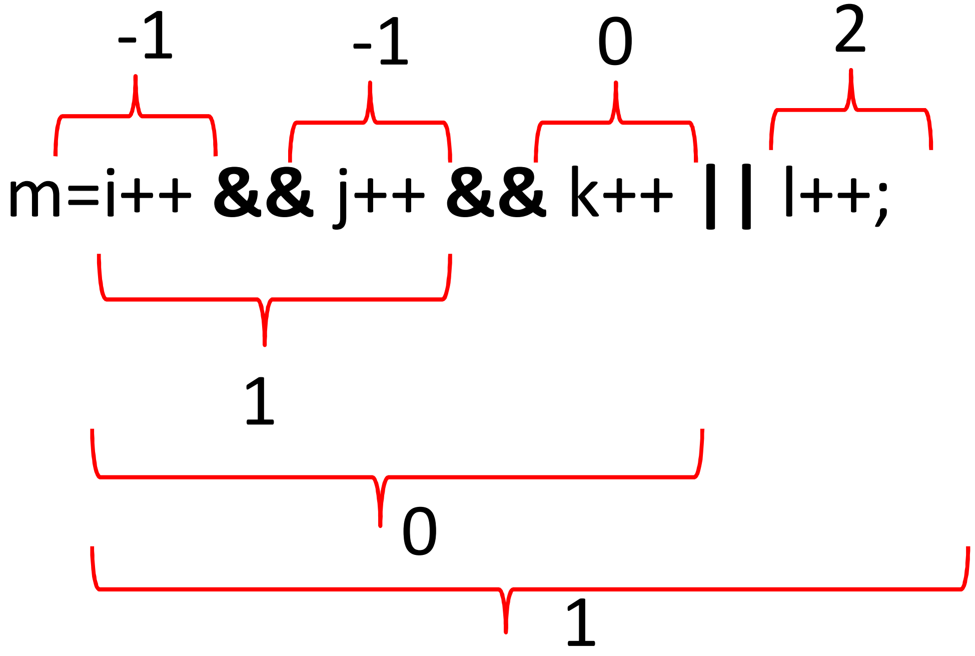
    (b-c) && a → a is not evaluated; as b-c is 0 so exp. value is 0
  - Why is it good?
  - (a != 0) && (b / a > 10) can be written safely, because if a is 0 the second expression is never evaluated and you don't get a division by 0 error!
  - Must be **very careful** for short-circuited eval of expressions with **side-effects**

    a = b = c = 10;

    --a + --b * --c **--a || --b && --c**

Examples

- int main(void)
  {
    int i=-1, j=-1, k=0, l=2, m;



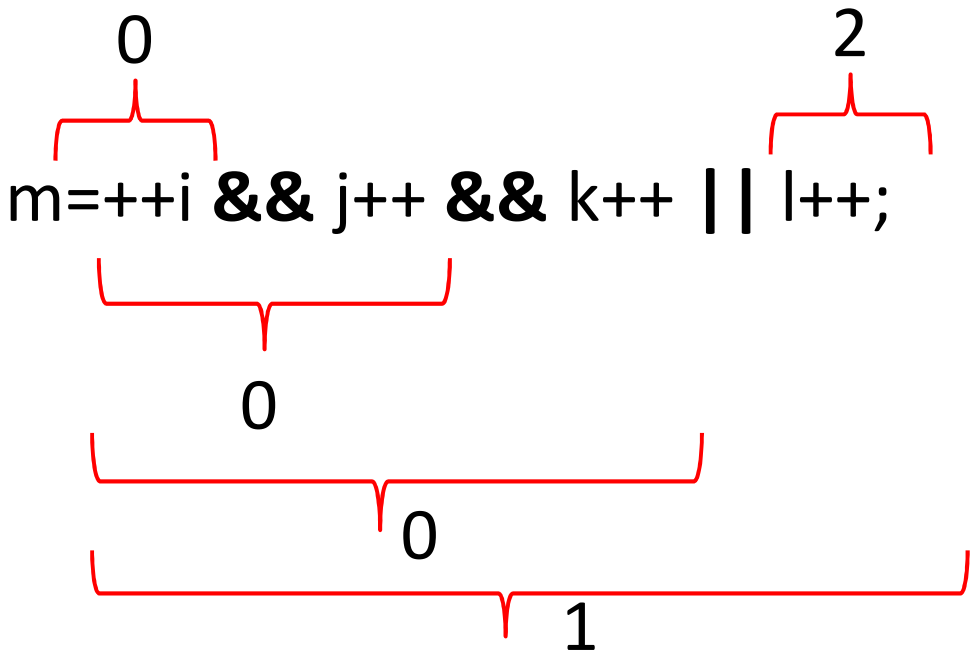
m=i++ **&&** j++ **&&** k++ **||** l++;

i	-10
j	-10
k	01
l	23
m	

printf("%d %d %d %d %d",i,j,k,l,m); }

Examples

- int main(void)
  {
     int i=-1, j=-1, k=0, l=2, m;



```
m=++i && j++ && k++ || l++;
```

```
printf("%d %d %d %d %d",i,j,k,l,m); }
```

i	-10
j	-1
k	0
l	23
m	1

Examples

-
  ```
  int main(void)
  {
    int i=-1, j=-1, k=0, l=2, m;
    m=i++ && j++ && k++ || l++;
    printf("%d %d %d %d %d",i,j,k,l,m);
  }
  ```

-
  ```
  int main(void)
  {
    int i=10;
     i=!i>14;
     printf ("i=%d",i);
  }
  ```

Selective Structure

- Allows changing the sequential order,
- It consists of a test for a condition followed by alternative paths that the program can follow

Condition

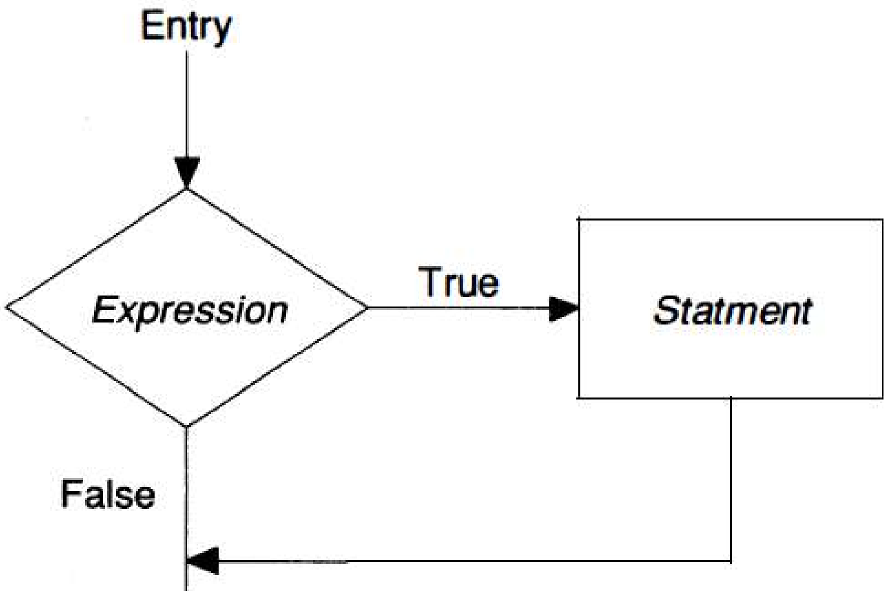
- For changing the flow o

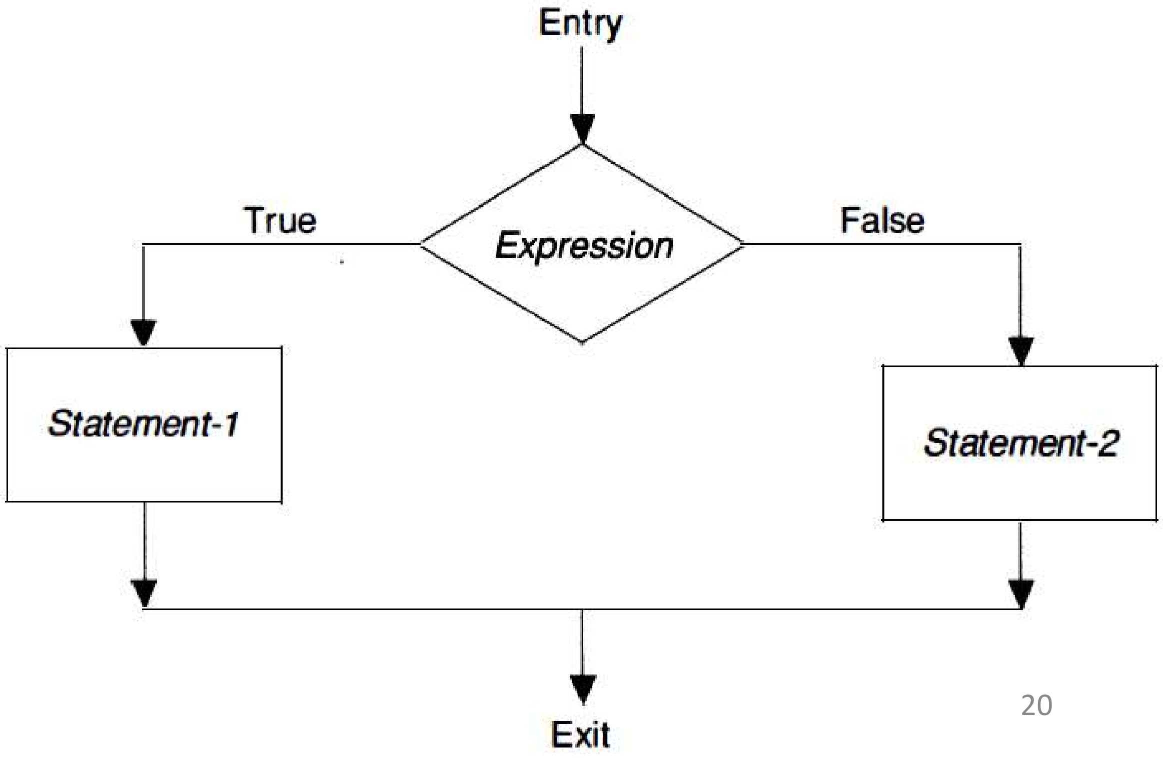
- if statements

```
if (expr)
   stat1
statN
```

if the e
otherw
immedi

```
if (expr)
   stat1
else
   stat2
statN
```





Be careful!

- Common mistake with if statements
- **if** (a = 10) { … }
- **if** (a == 10) { … }

Nested if statements

- Can be nested (no limit!), better use **braces** for blocks!

```
if (expr)
{  stat1
   …
   statK
}
else if (expr)
{  statP
   ….
   if (…)
   { … }
   else
   { … }
}
```

```
if (a > b)
{
    printf(“a is bigger”);
}
else
{
    if (a < b)
    {
        printf(“b is bigger”);
    }
    else
    {
        printf(“a = b”);
    }
}
```

```
if (a > b)
    printf(“a is bigger”);
else
    if (a < b)
        printf(“b is bigger”);
    else
        printf(“a = b”);
```

Sequence of nested if’s

```
if (expr1)
  if (expr2)
     if (expr3)
       ….
         if (exprN)
          {
               stat1
               …
               statK
          }
```

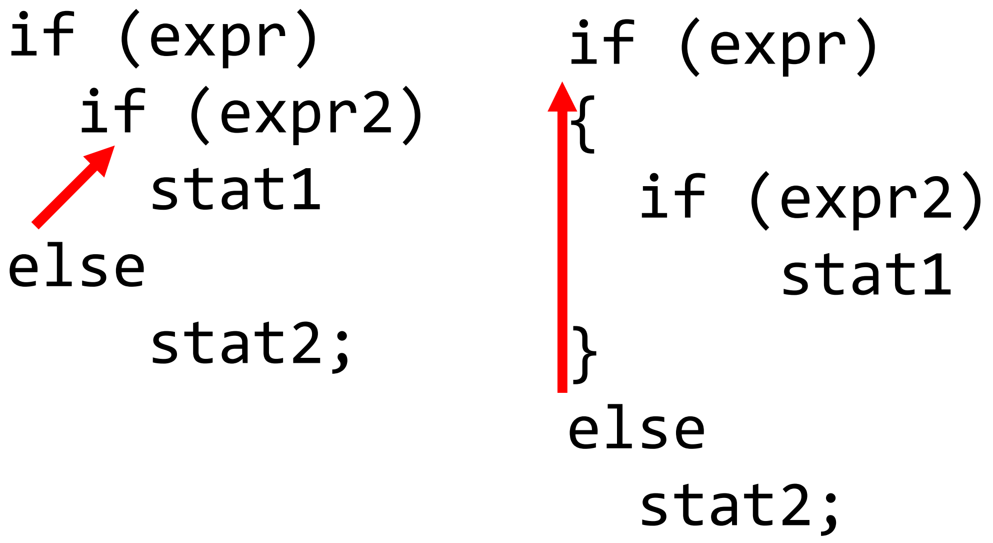
↔

```
if (expr1 && expr2 … && exprN)
{
   stat1
    …
   statK
}
```

What property of AND expr evaluation makes these two equivalent?

Dangling else

- An else is associated with the closest if without an else!



```
if (expr)
   if (expr2)
      stat1
else
      stat2;
```

```
if (expr)
{
   if (expr2)
         stat1
}
else
   stat2;
```

Multi-way conditionals: if-else ladder

- Same as nested if-else, just to improve readability and make it clear that it is a multi-way decision.

```
if (a > b)
   printf(“a is bigger”);
else
   if (a < b)
      printf(“b is bigger”);
   else
      printf(“a = b”);
```

```
if (a > b)
   printf(“a is bigger”);
else if (a < b)
   printf(“b is bigger”);
else
   printf(“a = b”);
```

Multi-way conditionals: if-else ladder

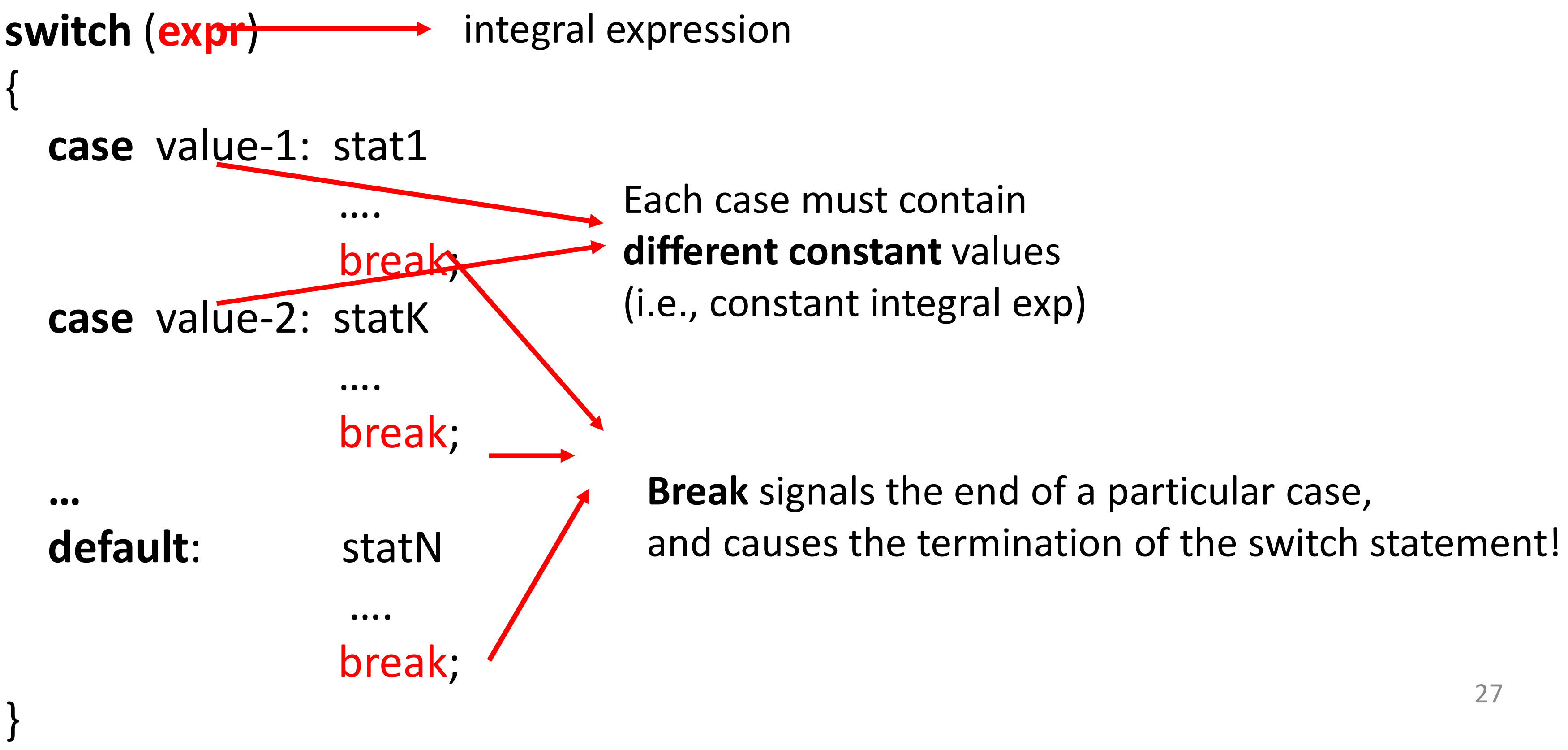
```
if (a > b)
   printf(“a is bigger”);
else if (a < b)
   printf(“b is bigger”);
else
   printf(“a = b”);
```

Each if-else has another if-else in the else block except the last else

- if one of the expr in conditions is true, corresponding stat is executed and terminates the chain; otherwise the final else is executed

Constant multi-way conditional : switch statements

- Useful when each test in a multiway if statement checks for a different value of the **same expr**
  - Use switch when we want constant multiway decision



```
switch (expr)
{
   case  value-1:  stat1
                   ....
                   break;
   case  value-2:  statK
                   ....
                   break;

   ...
   default:         statN
                    ....
                   break;
}
```

integral expression

Each case must contain
different constant values
(i.e., constant integral exp)

Break signals the end of a particular case,
and causes the termination of the switch statement!

Example

```
int main(void)
{
  int i;
  scanf ("%d",&i);
  switch (i)
  {
   default: printf("not 1-2-3");
            break;
   case 1: printf("one");
           break;
   case 2: printf("two");
           break;
   case 3: printf("three");
           break;
  }
}
```

Constant multi-way conditionals: switch statements

```
switch (expr)
{
   case  value-1:  stat1
                   ….
                   break;
   case  value-2:  statK
                   ….
                   break;
   …
   default:         statN
                    ….
                   break;
}
```

Important:

1) if no break, the execution continues
   (called fall-through)

2) Case order is not important

3) Last break is good
   (maybe you later add other cases)

4) This is different from if-else ladder!

- Can be faster
- But, has limitations: we only have constant values of the same exp.

Example

```
int main(void)
{
  int i;
  scanf ("%d",&i);
  switch (i)
  {
   case 1:  printf("one");
            break;
   case 2:  printf("two");
   default: printf("not 1-2-3");
            break;
   case 3:  printf("three");
            break;
  }
}
```

Conditional Expression Operator

- Conditional expression:
  - Expr **?** True-expr **:** False-expr
  - int a = x > 10 ? 1 : 0;
- Right-to-left associative.
  - X = c ? a : d ? e : f;
- Precedence:
  - '?' and ':' bracket the expression. True-expr can have operators of any precedence without parentheses.
  - The False-expr part has lower precedence than all operators except '=' and ','
  - So, c ? X = a : X = b will give an error!

    c ? X = a : (X = b) is fine

Repetitive Structure

- Allows a sequence of program statements to be executed several times
- Involves:
  - An entry point that may include intialization
  - A loop continuation condition
  - A loop body
  - An exit point

While Loop

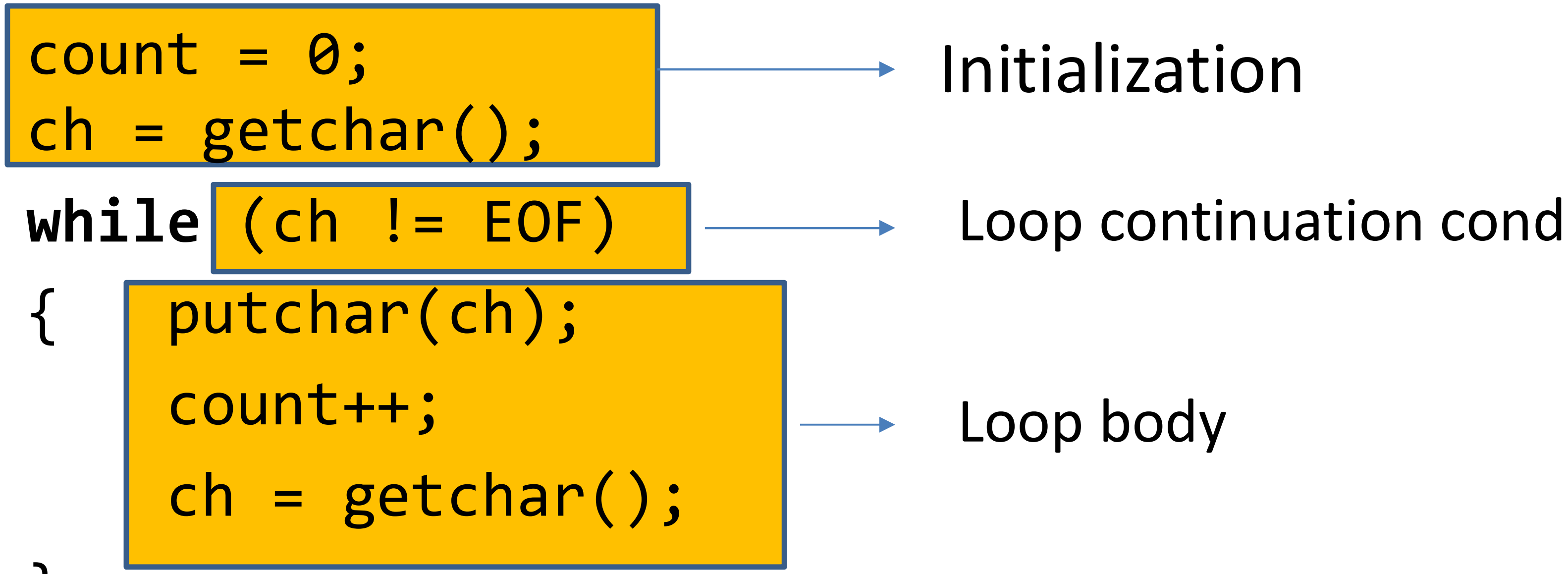
- **Pre-test loop:** loop continuation condition is tested before the loop body is executed (the body may never be executed)

```
Initialization;
while (expr)
      stat1;
```

```
Initialization;
while (expr)
{
      stat1
      …
      statN
}
```

Example

- Read chars, print and count them



```
int ch, count;

count = 0;
ch = getchar();
while (ch != EOF)
{   putchar(ch);
    count++;
    ch = getchar();
}
printf("total %d\n", count);
```

Initialization

Loop continuation cond

Loop body

Example

```
int ch, count;

count = 0;
ch = getchar();
while (ch != EOF)
{
    putchar(ch);
    count++;
    ch = getchar();
}
printf("total %d\n", count);
```

```
S> ./a.out
ab\n
..
CTRL-D
S> ./a.out
a\n
..
b\n
..
c\n
..
CTRL-D
```

Example

- Read chars, print and count them

```
int ch, count;

count = 0
ch = getchar();
while (ch != EOF)
{   putchar(ch);
    count++;
    ch = getchar();
}
printf("total %d\n", count);
```

```
int ch, count;

count = 0;


while ((ch = getchar()) != EOF)
{   putchar(ch);
    count++;
}
…
```

Example

- Factorial

```
int N, fact = 1;

scanf(“%d”, &N);
while (N > 0)
{ fact *= N--;  }
```

Bad examples: what is wrong?

```
while (x = 1)
{
        x = getchar();
}
```

```
x = 0.0;
while (x != 1.0)
{
        x += 0.005;
}
```

```
int i= 0, sum = 0, n;
while (i < 25)
{
   scanf(“%d”, &n);
   sum += n;
}
```

```
int i= 0, sum = 0, n;
scanf(“%d”, &n);
while (n != 0)
{
   sum += n;
   n--;
}
```

Do-while loop

- **Post-test loop**

```
Initialization;
do
    statement
while (expr);
```

```
Initialization;
do
{
        statement;
        statement;
        statement;
} while (expr);
```

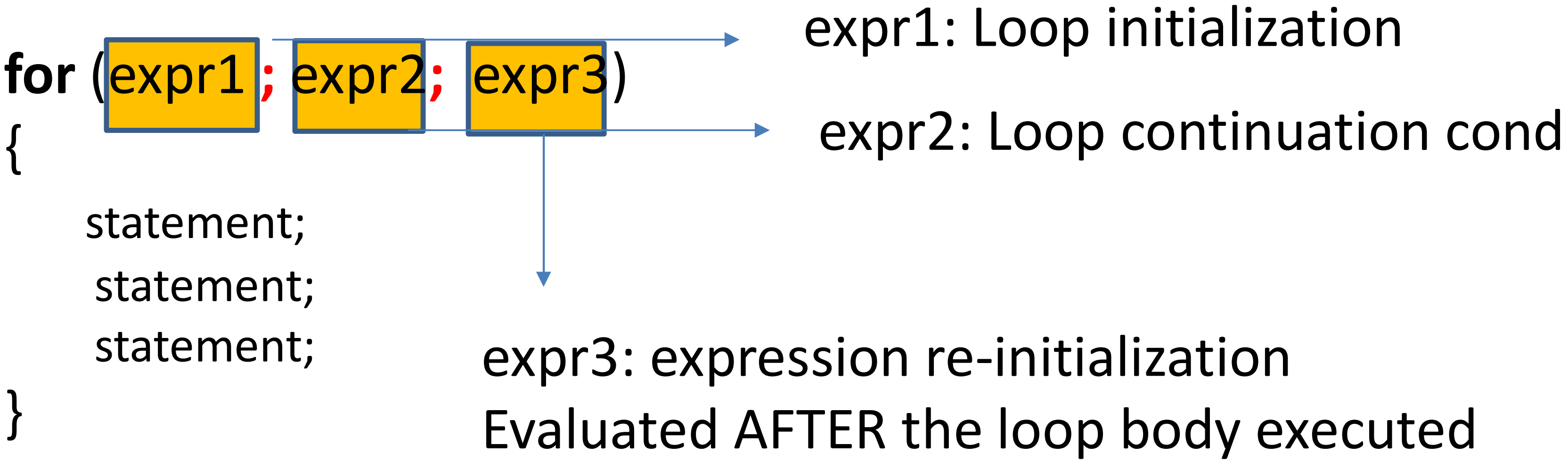
```
int number, digits;

digits=0;
scanf(“%d”, &number);

do
{
   number /= 10;
   digits++;
}
while (number > 0)
```

For loop

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



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

expr1: Loop initialization

expr2: Loop continuation cond

expr3: expression re-initialization
Evaluated AFTER the loop body executed

Example

```
int N, i;

scanf(“%d”, &N);

for (i = 0; i < N; i++)
     printf(“i: %d\n”, i);
```

```
i= 0;
while (i < N)
{
     printf(…);
     i++;
}
```

How can we write it using a while loop?

For – While Equivalence

```
for ( expr1 ; expr2;  expr3 )
{
      stat1
      stat2
      ….
      statN
}
```

```
expr1;
while (expr2)
{           stat1
            stat2
      ….
            statN
            expr3;
}
```

For Loop

- All 3 expr are optional but you must have ( ) and ; ;

- expr1: omit if initialization is done before loop
- expr3: omit if re-init is done in the loop body
- expr2: if omitted, would be an infinite loop -> you must somehow stop the loop
    - We will soon learn **break**

Example

```
int N, i;

scanf(“%d”, &N);

for (i = 0; i < N; i++)
     printf(“i: %d\n”, i);


i = 0;

for ( ; i < N; )
{    printf(“i: %d\n”, i);
     i++; }
```

Do-While and For Equivalence

```
do
{
        stat1;
        stat2;
        …
        statN;
} while (expr);
```

```
for (x=1; x; x = (expr))
{
                stat1;
                stat2;
                …
                statN;
}
```

For Loop

- You can have loops with complex expressions

```
for (i=0, j=M; i < N && j > 0; i++, j--);
```

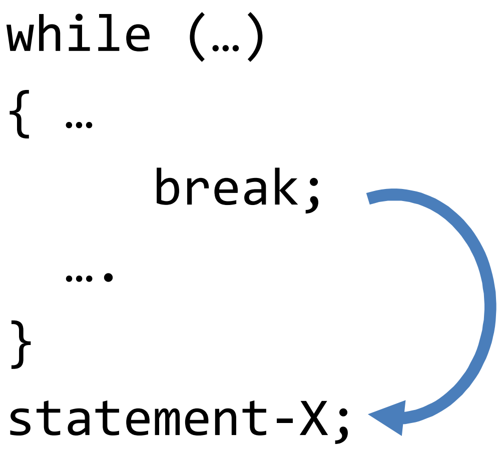
Nested Loops

- You can have loops within loops:

```
for (i=0; i<N; i++)
{
   for (j=0; j<N; j++)
   {
       ….
   }
}
```

break;

- Stop the loop/iteration and continue with the statement after the loop.
- Usable with while, for and do-while



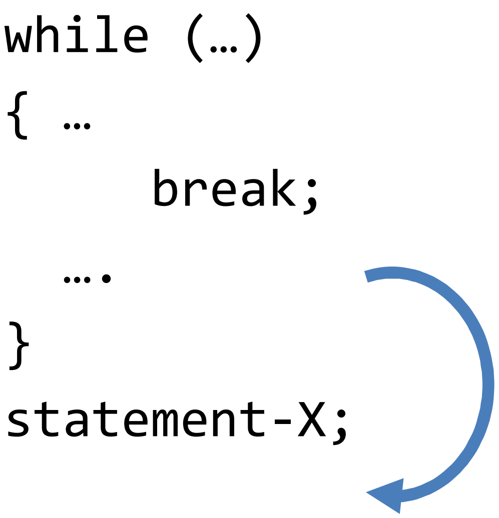
```
while (…)
{ …
      break;
  ….
}
statement-X;
```

```
int c;

while (1)
{
       c = getchar();
      if (c == EOF)
             break;
       putchar(c);
}
```

break;

- When located in nested loops, the only loop interrupted is the one whose body contains the break statement
- Using the break statement outside of a **loop body** or **switch** statement is illegal!



```
while (…)
{ …
      break;
  ….
}
statement-X;
```

continue;

- Skips the remaining statements in the loop and continues with the “loop head”.
  - if while loop, continues with the loop continuation cond
  - if for loop, first execute re-init statement and the loop continuation cond
- Usable with while, for and do-while

```
while (…)
{ …
      continue;
  ….
}
```

```
sum = 0;
for (i=1; i<N; i++)
{
     if (i%5 == 0)
          continue;
     sum = sum + i;
}
```

null statement

- C allows a statement with just ; to be placed wherevever a statement can appear
  - Has no effect
  - Needed bec of syntax

```
for (count =0; getchar() ! =EOF; count++)
     ;
```

Comma operator

- Used to combine related expressions into one
- The compound expression is
  - evaluated from left to right
  - type and valueof the result are the type and value of the **right operand**
  - (i.e., value of left operand(s) are discarded, they are evaluated only for side effects)
- Comma operator has the LOWEST precedence

Comma operator

```
int t, ,x y;

t = x;

x = y;

y= t;

/* instead: */

t = x, x = y, y= t;
```

```
while ((ch = getchar()) != EOF)
{    putchar(ch);
     count++;
}

/* instead: */

while (ch = getchar(), ch != EOF)
{    putchar(ch);
     count++;
}
```

Example

```
int main()
{  int x;
   while (1)
   {   printf("Enter input: \n");
      scanf("%d", &x);
      switch (x)
      {
         default: printf("error, type again\n"); continue;
         case 1: printf("case1\n"); break;
         case 0: printf("case0\n");
       }
      printf("Thanks for correct input: %d\n",x);
      break;
   } /* end of while */
} /* end of main */
```

Arrays

Today

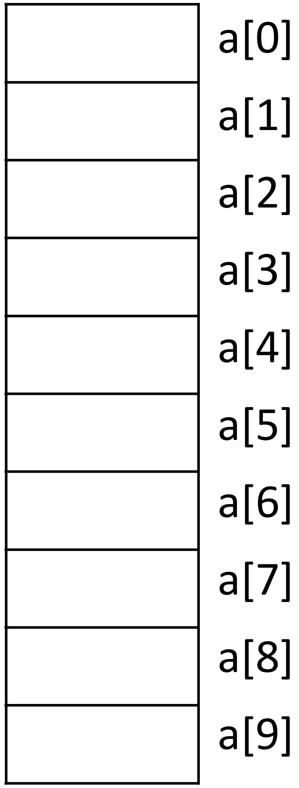
- Collection of data
  - Arrays: An ordered finite collection of items of the same type
    - İndividual data items are elements
  - Can be as many dimensions as you want

Arrays of Numerical Values

- Array Declaration:
  - *type* name[expr1] [expr2] …[exprN];
  - Each expr denotes the length in a dimension
  - Each expr must be **constant** integral expr

  (can include symbolic constants etc)

  - Arrays are NOT dynamic in ANSI C!
- Ex: int a[10];
  - Length: 10
- Ex: float b[20];
  - Length: 20



Accessing Array Elements

/* declaration */

int a[10];

/* Access: **any integral expression** can be used as a subscript: */ a[3*i]

/* We can use the elements of an array like a variable in **expressions***/

int b = a[8];

int c = 25 + a[2] - a[8] / a[0];

/* Like a variable, we can **assign** values to the elements */

a[2] = 25;   a[i] += 25 – a[2]++;

Notes

- C does not check array boundaries:
  - If you try to access an array's element with negative index or with an index which is bigger than its length, you **may** get a **run-time error (behaviour is undefined)**
- Arrays cannot be copied like this:
  - int a[10], b[10];
  - a = b; ➔ error!
  - Correct way: for(i=0; i < 10; i++) a[i] = b[i];
- Arrays cannot be automatically initialized to a value:
  - int a[10];
  - a = 0; ➔ error!
  - Correct way: for(i=0; i < 10; i++) a[i] = 0;

Initializing Arrays

```
int a[3] = {1, 2, 3};
float c[3] = {.1, 2.2, 0.3};
char letters [3] = {‘a’, ‘b’,’c’}
```

```
/* The following two are equivalent */
int a[3] = {1, 2, 3};
int a[] = {1, 2, 3}; /* C derives lengtsh from the initializers */
```

```
/* If the number of initializers is less than the size of the array, the remaining
   ones are set to zero; if more, compile error! */
int a[8] = {1, 2, 3};  ➔ int a[8] = {1, 2, 3, 0, 0, 0, 0, 0};
```

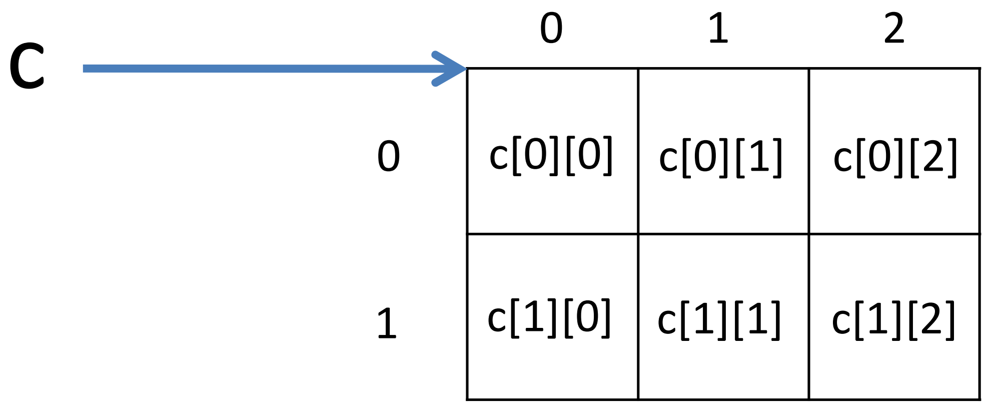
Strings: Arrays of characters

A character array may be initialized to a string constant; then array also has the terminating NULL in the string!

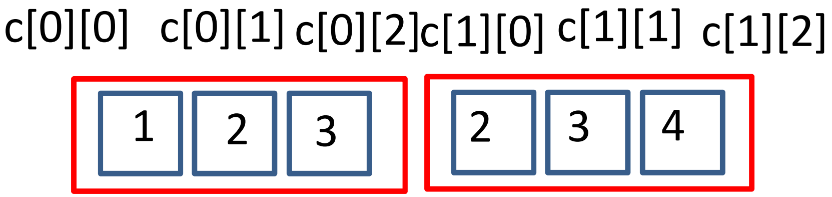
- char a[3] = “AB”; ➔ char a[3] = {‘A’, ‘B’, ‘\0’};
- char a[] = “AB”; ➔ char a[3] = {‘A’, ‘B’, ‘\0’};
- char b[2] = “AB”; ➔ char b[2] = {‘A’, ‘B’};
  - You cannot use string functions on b since it does not have an ending mark, i.e., ‘\0’.
- char a[2] = "ABC"; /* compile error */

Multi-dimensional Arrays

```
int a[] = {1, 2, 3};
int c[2][3] = {
                {1, 2, 3},
                {2, 3, 4}
                };
```

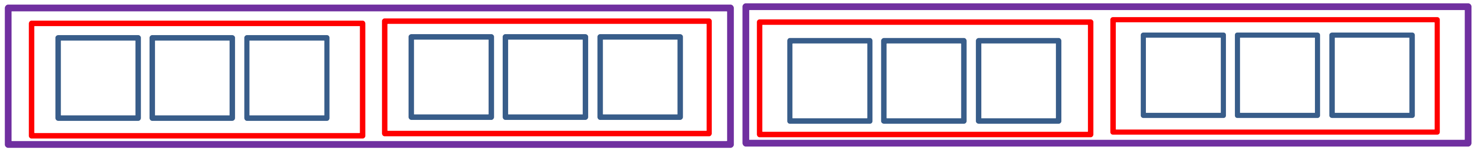


	0	1	2
0	c[0][0]	c[0][1]	c[0][2]
1	c[1][0]	c[1][1]	c[1][2]



c[0][0]	c[0][1]	c[0][2]	c[1][0]	c[1][1]	c[1][2]
1	2	3	2	3	4

```
int d[2][2][3] = {
          { {1, 2, 3},       {2, 3, 4}},
          { {5, 6, 3},       {7, 8, 4}}
          };
```



Arrays as Function Arguments

Arrays as Function Arguments

Passing **Array Elements** as Arguments

```
#include <math.h>
double cuberoot(double x)
{ return pow(x, 1.0/3.0); }

int main(void)
{ double x, z[3] = {8, 27, 125};
  printf( "%f %f", cuberoot(z[1]), z[1] ); }
```

- Like simple variables, **individual array elements** are **passed by value**.
  - Values copied into parameters and can’t be changed by the called function.

- Passing **Arrays** as Arguments

```
double cuberoot(double x)
{ return pow(x, 1.0/3.0); }

void array_cuberoot(double x[3])
{ int i;

   for (i=0; i<3; i++)
      x[i] = cuberoot(x[i]); }

int main(void)
{ double z[3] = {8, 27, 125};
  array_cuberoot(z);
  /* Output if we print elements of array z here?*/ }
```

To pass **entire array** as argument, use just the **name** of it (no [] etc). in the function call!

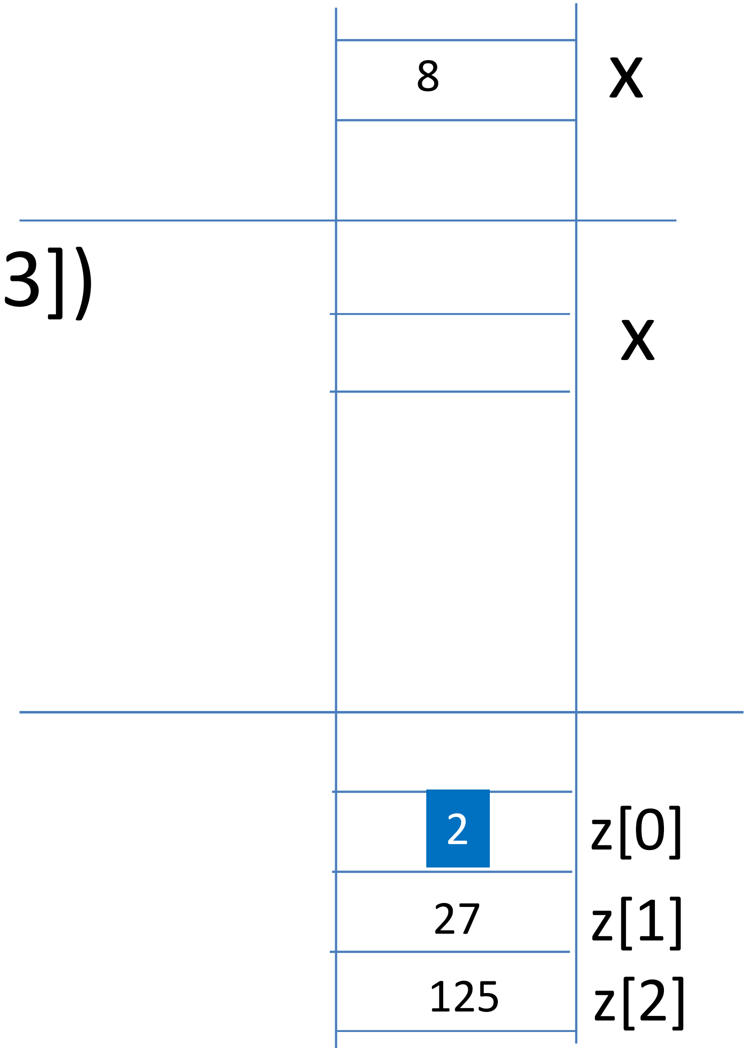
- Passing **Arrays** as Arguments

```
double cuberoot(double x)
{ return pow(x, 1.0/3.0); }

void array_cuberoot(double x[3])
{ int i;

   for (i=0; i<3; i++)
      x[i] = cuberoot(x[i]); }

int main(void)
{ double z[3] = {8, 27, 125};
  array_cuberoot(z);
   /* Output if we print elements of array z here?*/ }
```



Passing **Arrays** as Arguments

- When array is passed as an argument to a function, the **address of the beginning of the array** is passed (copied) to function,
- **but** the elements of the array are not copied to the function.
- Thus, any reference to the parameter array name inside the function indeed refers to the elements of the argument array!

Passing Arrays as Arguments

- Since only the **address of the beginning of the array** is passed (copied) to function, there is no need to declare the array length specified in brackets (for 1-D arrays) and compiler will ignore it.
  - So, we can specifty the **1D array parameter** as
    - void **array_cuberoot**(double x[])
  - To be able to know the array length in the function, pass the length as another parameter
    - void **array_cuberoot**(double x[], int length)

Passing Arrays as Arguments

A more general function that can work with a 1D array of any length

```
void array_cuberoot(double x[], int length)
{ int i;

  for (i=0; i<length; i++)
    x[i] = cuberoot(x[i]); }
```

However, when we declare a **multi-dim array** as a **parameter**, we must still specify **all** but the first dimension!