Computer Programming 1

Instructions

Outline

- Program structure
- Instruction
  - Simple instructions
  - Conditional instructions
  - Iterative instructions

Structure of a program

- A program consists of a set of functions, possibly divided into several files
  - The function that constitutes the main program must necessarily be called main
- Each program contains a list of instructions: simple or structured instructions

Example

```
(…)
int main() {
  int x=2, y=6, z;
  z=x*y;
  return 0;
}
```

Simple instructions

- Simple instructions are the basis of more complex instructions (structured instructions)
- They are always terminated by a semicolon ";"
- They can be distinguished in:
    - definitions / declarations (declaration-statement) of
        - Variables, e.g., int x,y,z;
        - Constants, e.g., const int kilo=1024;
    - expressions (expression-statement)
        - input, e.g., cin >> x
        - output, e.g., cout << 3*x
        - assignment, e.g., x=2*(3-y)
        - arithmetic, e.g., (x-3)*sin(x)
        - logics, e.g., x==y && x!=b
        - constants, e.g., 3*12.7
        - conditional (… next slides)

      Each expression followed by “;” is also an instruction

Conditional expression

- Syntax: exp1 ? exp2 : exp3;
  - If exp1 is true it is exp2 otherwise it is exp3

Example

```
Price = value * weight * (weight>10) ? 0.9 : 1;
```

*If weight is greater than 10, then: Price = value * weight * 0.9*

*Otherwise: Price = value * weight * 1*

Example of use of conditional expression:

{ ../IF_THEN_ELSE_SWITCH/**conditional_expression**.cc}

Structured instructions

- Structured statements allow you to specify complex actions
- They are distinguished in
  - Composed instructions (compound statement)
  - Conditional instructions (conditional statements)
  - Iterative instructions (iteration-statement)
  - Jump instructions (jump-statement)

Composed Instructions

- Transform a sequence of statements into a single statement
  - The sequence is delimited by '{' and ‘}’
  - The delimited sequence is called a **block**
- Definitions can appear anywhere in the block
  - they are visible only inside the block
  - can access externally defined objects
  - in case of identical identifiers, the innermost one prevails

Example

```
        {
           int a= 4;
           a *= 6;
           char b=‘c’;
           b += 3;
        }
```

Example of block and scope:

{ ../IF_THEN_ELSE_SWITCH/visibility.cc|visibility0.cc}

Composed Instructions

```
using namespace std;
#include <iostream>

int main() {
  int n = 44;
  cout << "External block: n = " << n << endl;
  {
    cout << "Internal block 1: n = " << n << endl;
    int n = 3;
    cout << “Internal block 2: n = " << n << endl;
  }
  cout << "External block: n = " << n << endl;

  return 0;
}
```

```
$ g++ visibility0.cc
$ ./a.out
External block: n = 44
Internal block 1: n = 44
Internal block 2: n = 3
External block: n = 44
```

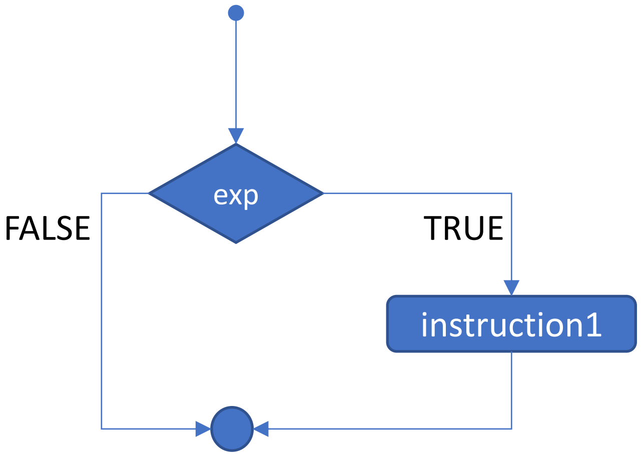
{ ../IF_THEN_ELSE_SWITCH/visibility0.cc}

The conditional instruction *if-then*

- The simple instruction "if-then"
  - Syntax:

```
if (exp)
        instruction1
```

  - Meaning:
    - If exp is true, then instruction1 is executed
    - Otherwise, nothing is executed
  - Note: instruction1 can consist in another complex instruction (e.g., another if-then block)



Example

```
if (x != 0)
    y=1/x;
```

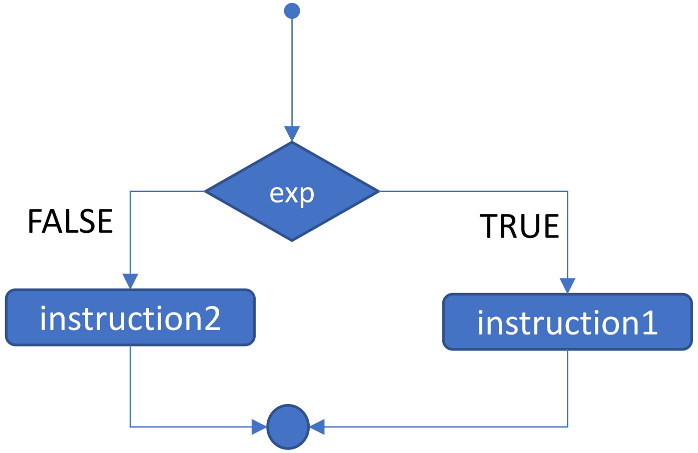
Example of if-then:

{ ../IF_THEN_ELSE_SWITCH/divisibility.cc}

The conditional instruction *if-then-else*

- The composed instruction "if-then-else"
  - Syntax:

    if (exp) instruction1
    else instruction2
  - Meaning:
    - If exp is true, then instruction1 is executed
    - Otherwise, instruction2 is executed
  - Note: instruction1 and instruction2 can consist in complex instructions (e.g., another if-then-else block)



Example

```
if ( x<0 )
    y=-x;
else
    y=x;
```

Example of if-then-else:

{ ../IF_THEN_ELSE_SWITCH/divisibility2.cc}

Nested *if*

- In if-then and if-then-else blocks, instruction1 and instruction2 can be complex instructions (e.g., a block, another if-then-else block etc)
- The nesting of if-then-else and the use of logical operators allow us to compose quite complex decisional structures

Example

```
if ( x<0 ) {
    y=-x;
    if ( y == 5)
        y=1;
} else
    y=x;
```

Example of nested if-then-else:

{ ../IF_THEN_ELSE_SWITCH/eq_1grade.cc}

Example of reverse order:

{ ../IF_THEN_ELSE_SWITCH/eq_1grade_2.cc}

Example with logical operators:

{ ../IF_THEN_ELSE_SWITCH/eq_1grade_3.cc}

Note

- Code indentation is important!
    - It increase the code readability
    - It helps us to read and understand the code

Nested *if*

```
1.    using namespace std;
2.    #include <iostream>

3.    // Computation of solutions for ax + b = 0
4.    int main() {
5.      float a,b;
6.      cout << "Insert a and b : ";
7.      cin >> a >> b;
8.      if (a==0) {
9.        if (b==0)
10.         cout<<"Infinite solutions\n";
11.       else {
12.         cout<<"A solution does not exist\n";
13.          }
14.     } else
15.       cout<<"Solution: "<<-b/a<<endl;
16.
17.     return 0;
18.   }
```

```
$ g++ eq_1grade.cc
$ ./a.out
Insert a and b :
5 10
Solution: -2
```

line	a	b	*Std out*
5	-	-	
6			Insert a and b
7	*5*	*10*	
8	5	10	
15			Solution: -2
17			

{ ../IF_THEN_ELSE_SWITCH/eq_1grade.cc}

Nested *if*

```
using namespace std;
#include <iostream>

// Computation of solutions for ax + b = 0
int main() {
  float a,b;
  cout << "Insert a and b : ";
  cin >> a >> b;
  if (a==0) {
    if (b==0)
      cout<<"Infinite solutions\n";
    else {
      cout<<"A solution does not exist\n";
       }
  } else
    cout<<"Solution: "<<-b/a<<endl;

  return 0;
}
```

```
$ g++ eq_1grade.cc
$ ./a.out
Insert a and b :
5 10
Solution: -2
```

```
$ g++ eq_1grade.cc
$ ./a.out
Insert a and b :
0 1
A solution does not exist
```

```
$ g++ eq_1grade.cc
$ ./a.out
Insert a and b :
0 0
Infinite solutions
```

{ ../IF_THEN_ELSE_SWITCH/eq_1grade.cc}

Code indentation

Note

- Code indentation is important!
  - It increase the code readability
  - It helps us to read and understand the code

Example

```
if ( x<0 ) {
    y=-x;
    if ( y == 5)
        y=1;
} else
    y=x;
```

Example

```
if ( x<0 ) {
y=-x;
if ( y == 5)
y=1; } else
y=x;
```

```
#include <iostream>
int main(){std::cout<<"C/C++ source code Formatter"; return 0;}
```

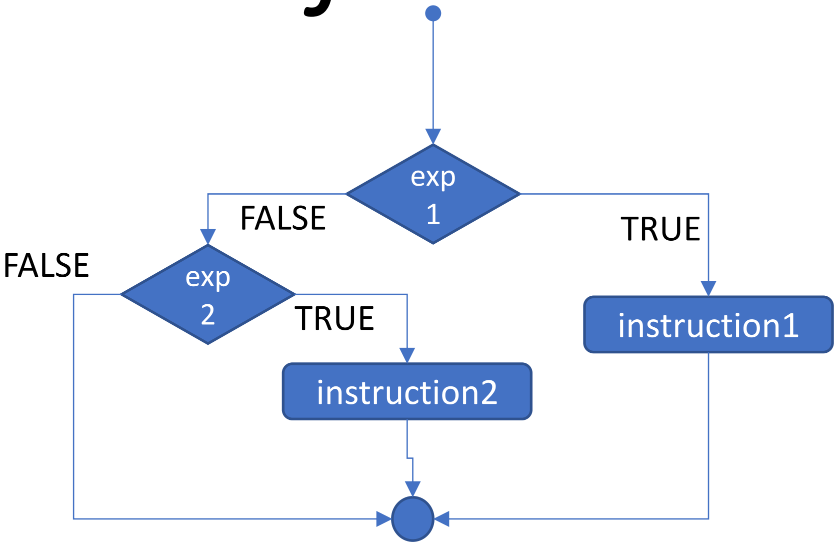
```
#include <iostream>
int main(){
    std::cout<<"C/C++ source code Formatter";
    return 0;
}
```

The conditional instruction *if-then-elseif-else*

- The composed instruction “if-then-elseif-else”
  - Syntax:

    ```
    if (exp1) instruction1
    else if (exp2) instruction2
    else instruction3
    ```

  - Meaning:
    - If exp1 is true, then instruction1 is executed
    - If exp1 is false and exp2 is true, then instruction2 is executed
    - Otherwise, instruction3 is executed
  - Note: instruction1/2/3 can consist in complex instructions (e.g., another if-then-else block)



Example

```
if ( x<0 )
    y=-x;
else if ( x>0 )
    y=x;
else
    y=0;
```

Example:

{ ../IF_THEN_ELSE_SWITCH/eq_1grade3.cc} { ../IF_THEN_ELSE_SWITCH/eq_1grade4.cc}

Nested *if*

```
1.      using namespace std;
2.      #include <iostream>

3.      // Computation of solutions for ax + b = 0
4.      int main() {

5.       float a,b;
6.        cout << "Insert a and b : ";
7.        cin >> a >> b;
8.        if ((a==0)&&(b==0))
9.          cout<<"Infinite solutions\n";
10.       else if ((a==0)&&(b!=0))
11.         cout<<"A solution does not exist\n";
12.       else
13.         cout<<"Solution: "<<-b/a<<endl;
14.       return 0;

15.     }
```

```
$ g++ eq_1grade3.cc
$ ./a.out
Insert a e b :
5 10
Solution: -2
```

line	a	b	*Std out*
5	-	-	
6			Insert a and b
7	*5*	*10*	
8	5	10	
10	5	10	
13			Solution: -2
14			

{ ../IF_THEN_ELSE_SWITCH/eq_1grade3.cc}

Nested *if*

```
using namespace std;
#include <iostream>

// Computation of solutions for ax + b = 0
int main() {

 float a,b;
  cout << "Insert a and b : ";
  cin >> a >> b;
  if ((a==0)&&(b==0))
    cout<<"Infinite solutions\n";
  else if ((a==0)&&(b!=0))
    cout<<"A solution does not exist\n";
  else
    cout<<"Solution: "<<-b/a<<endl;
  return 0;

}
```

```
$ g++ eq_1grade3.cc
$ ./a.out
Insert a and b :
5 10
Solution: -2
```

```
$ g++ eq_1grade3.cc
$ ./a.out
Insert a and b :
0 1
A solution does not exist
```

```
$ g++ eq_1grade3.cc
$ ./a.out
Insert a and b :
0 0
Infinite solutions
```

{ ../IF_THEN_ELSE_SWITCH/eq_1grade3.cc}

Examples about nested if-then-else

Example of alternatives to the user:

{ ../IF_THEN_ELSE_SWITCH/conversion.cc}

Example of alternatives to the user:

{ ../IF_THEN_ELSE_SWITCH/conversion2.cc}

Example of “Dangling else”:

{ ../IF_THEN_ELSE_SWITCH/dangling_else.cc}

Example of “Dangling else” (2):

{ ../IF_THEN_ELSE_SWITCH/dangling_else2.cc}

Example of the typical error:

{ ../IF_THEN_ELSE_SWITCH/ifeq_err.cc}

Example of correct version:

{ ../IF_THEN_ELSE_SWITCH/ifeq_corr.cc}

Example of minimal between two numbers:

{ ../IF_THEN_ELSE_SWITCH/min.cc}

Example of minimal among three numbers :

{ ../IF_THEN_ELSE_SWITCH/min2.cc}

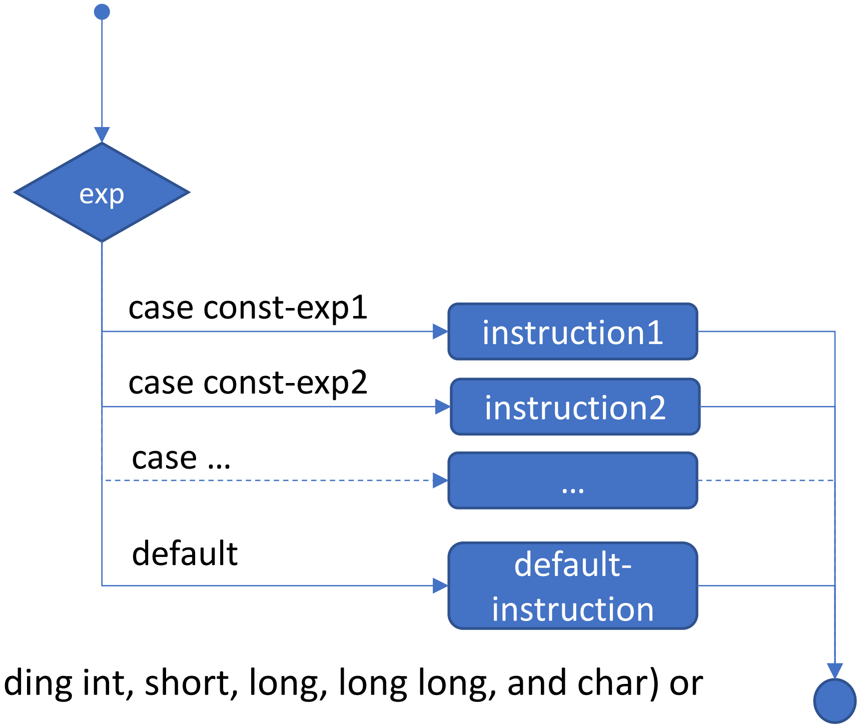
Example of choices among multiple values:

{ ../IF_THEN_ELSE_SWITCH/simple_calc.cc}

The conditional instruction *switch (1)*

- The conditional instruction “switch”
- Syntax:

```
switch (exp) {
        case const-exp1:
                        instruction1; break;
        case const-exp2:
                        instruction2; break;
        …
        default:
                        default-instruction;
        }
```



- The execution:
  1. execution/computation of the expression exp
     - exp can return a value of type integer (including int, short, long, long long, and char) or enum.
  2. execution of the instruction that corresponds to the computed alternative
  3. execution of the default instruction, in case it is defined and in case no other expression correspond
  4. otherwise, nothing is executed
- Note: instruction1 can consist in another complex instruction

Example

```
char operator=‘+’
switch(operator) {
  case ‘+’:
      // code block1
      break;
  case ‘*’:
      // code block2
      break;
  default:
     // code block
}
```

Example of choice among multiple values with the switch:

{ ../IF_THEN_ELSE_SWITCH/simple_calc2.cc}

The conditional instruction *switch (2)*

```
char grade = 'D';
switch(grade) {
    case 'A' :
        cout << "Excellent!" << endl;
        break;
    case 'B' :
    case 'C' :
        cout << "Well done" << endl;
        break;
    case 'D' :
        cout << "You passed" << endl;
        break;
    case 'F' :
        cout << "Better try again" << endl;
        break;
    default :
        cout << "Invalid grade" << endl;
}
cout << "Your grade is " << grade << endl;
```

1.Definition of the variable grade with initialization to the char ‘D’

2.Evaluation of the variable grade

Note case ‘B’ (no break) and case ‘C’

3.Selection of the case to be executed according to the evaluation of grade

4.Execution of the selected case statements

5.Break in the switch execution

6.Execution of the flow outside the switch statement

Multiple choices with the switch (1)

- If after the last instruction of an alternative there is no the break instruction, the next alternative is also executed
- This behavior is not recommended but may be justified in some cases

Example

```
enum days { monday, tuesday, wednesday, thursday, friday, saturday, sunday };
int workingHours =0;
days day= monday;
switch ( day )
{ case monday : case tuesday : case wednesday : case thursday :
  case friday :
        workingHours +=8; break ;
   case saturday : case sunday:
        break ;
}
cout << workingHours << endl;
```

```
$ g++ simple_switch_enum_1.cc
$ ./a.out
8
```

{ ../IF_THEN_ELSE_SWITCH/ simple_switch_enum_1.cc}

Multiple choices with the switch (2)

- If after the last instruction of an alternative there is no the break instruction, the next alternative is also executed
- This behavior is not recommended but may be justified in some cases

Example

```
enum days { monday, tuesday, wednesday, thursday, friday, saturday, sunday };
int workingHours =0;
days day= saturday;
switch ( day )
{ case monday : case tuesday : case wednesday : case thursday :
   case friday :
            workingHours +=8; break ;
    case saturday : case sunday:
           break ;
}
cout << workingHours << endl;
```

```
$ g++ simple_switch_enum_1.cc
$ ./a.out
0
```

{ ../IF_THEN_ELSE_SWITCH/ simple_switch_enum_1.cc}

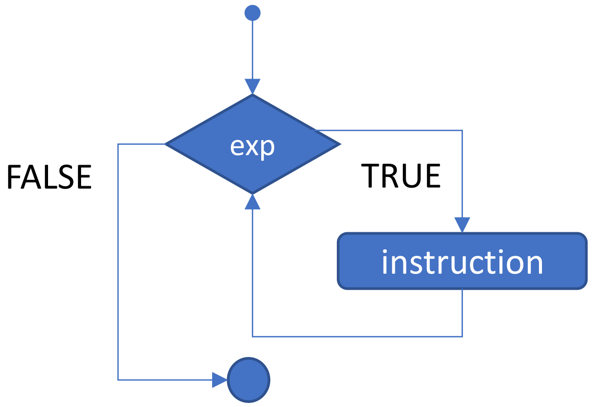
The iterative instruction *while (while-do)*

- The iterative instruction: "while-do"
- Syntax: while (exp) { instruction }

  where:

  exp is a Boolean expression

  instruction can be a complex instruction



- The execution:
  1. Execution/computation of the expression exp
  2. If exp is true, instruction is executed, and the execution is repeated until exp is true (if exp is false, instruction is not executed)
- Attention
  - Instruction could be never executed
  - It is possible to generate infinite loops

Example

```
...
int index = 0;
while ( index<5 ) {
  //block of instructions
  index=index+1;
}
```

Note

- Typically, exp contains, at least, a variable (named **control variable** of the cycle) that is modified in instruction, thus allowing the variable to converge towards ***false***

While-do

```
using namespace std;
#include <iostream>

int main() {

  int n,index,sum;
  cout << "How many integers do you want to sum? ";
  cin >> n;

  index = 1;
  sum = 0;

  while (index<=n) {
    sum += index;
    index++;
  }
  cout  << "Sum = "  << sum   << endl;
  return 0;

}
```

```
$ g++ sumintegers_while.cc
$ ./a.out
How many integers do you want to sum?  3
Sum = 6
```

{ ../LOOPS/sumintegers_while.cc}

While-do

```
1.    using namespace std;
2.    #include <iostream>

3.    int main() {

4.      int n,index,sum;
5.      cout << "How many integers do you want to sum? ";
6.      cin >> n;

7.      index = 1;
8.      sum = 0;

9.      while (index<=n) {
10.       sum += index;
11.       index++;
12.     }
13.     cout  << "Sum = "  << sum   << endl;
14.     return 0;

15.   }
```

line	n	index	sum	*Std out*
4	-	-	-	
5	-	-	-	How many…
6	*3*	-	-	
7	3	1	-	
8	3	1	0	
9	3	1	0	
10	3	1	0+1	
11	3	2	1	
9	3	2	1	
10	3	2	1+2	
11	3	3	3	
9	3	3	3	
10	3	3	3+3	
11	3	4	6	
9	3	4	6	
13	3	4	6	Sum=6

{ ../LOOPS/sumintegers_while.cc}

Examples about the use of the while (1)

Example of operation repetition (increasing counter):

{ ../LOOPS/printhello.cc}

Example of decreasing counter:

{ ../LOOPS/printhello2.cc}

Example of infinite loop:

{ ../LOOPS/printhello_infloop.cc}

Example of sum with accumulator:

{ ../LOOPS/sumintegers_while.cc}

Example of product with accumulator:

{ ../LOOPS/fact_while.cc}

Example of exit condition different from a counter:

{ ../LOOPS/divisible.cc}

Examples about the use of the while (2)

Example of repetition of menu command

{ ../LOOPS/conversion3_while.cc}

- Example of sum with accumulator, with counter:

  { ../LOOPS/series_while.cc}

- Example of sum with accumulator, with conditional exit:

  { ../LOOPS/series_while1.cc}

- Example of “cin loops”:

  { ../LOOPS/cin_loop.cc}

- Example of “cin loops” with fail:

  { ../LOOPS/cin_loop_equivalent.cc}

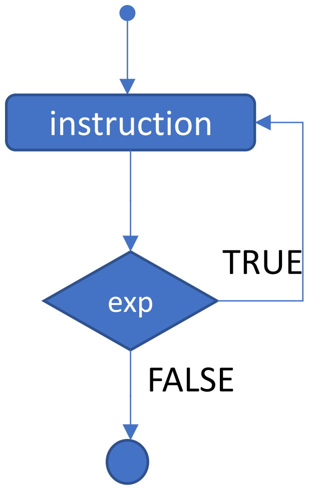
The iterative instruction *do (do-while)*

- The iterative instruction “do-while”
- Syntax: do { instruction } while (exp);

  exp is a Boolean expression

  instruction can be a complex instruction

- The execution:
  1. execution of instruction
  2. execution/computation of the expression exp
  3. if exp is true, the repetition of the execution of the do instruction
- Instruction is always executed at least once
- Typically, among the different iterative instruction is the less used one



Example

```
…
int index = 0;
do {
   //block of instructions
   index=index+1;
}
while ( index<5 );
```

Do-while

```
using namespace std;
#include <iostream>

int main() {

  int n,index,sum;
  cout << "How many integers do you want to sum? ";
  cin >> n;

  index = 1;
  sum = 0;

  if (n>=1)    // conditions to avoid N=0
    do {
      sum += index;
      index++;
    } while (index<=n);

  cout  << "Sum = "  << sum   << endl;
  return 0;

}
```

```
$ g++ sumintegers_do.cc
$ ./a.out
How many integers do you want to sum?  3
Sum = 6
```

{ ../LOOPS/sumintegers_do.cc}

Do-while

```
1.    using namespace std;
2.    #include <iostream>

3.    int main() {

4.      int n,index,sum;
5.      cout << "How many integers do you want to sum? ";
6.      cin >> n;

7.      index = 1;
8.      sum = 0;

9.      if (n>=1)    // conditions to avoid N=0
10.       do {
11.         sum += index;
12.         index++;
13.       } while (index<=n);

14.     cout  << "Sum = "  << sum   << endl;
15.     return 0;

16.  }
```

line	n	index	sum	*Std out*
4	-	-	-	
5	-	-	-	How many…
6	3	-	-	
7	3	1	-	
8	3	1	0	
9	3	1	0	
11	3	1	0+1	
12	3	2	1	
13	3	2	1	
11	3	2	1+2	
12	3	3	3	
13	3	3	3	
11	3	3	3+3	
12	3	4	6	
13	3	4	6	
14	3	4	6	Sum=6

{ ../LOOPS/sumintegers_do.cc}

Examples about the use of the do

- Example of sum with accumulator (do)

  { ../LOOPS/sumintegers_do.cc}

- Example of menu command (do)

  { ../LOOPS/conversion3_do.cc}

- Example of base conversion:

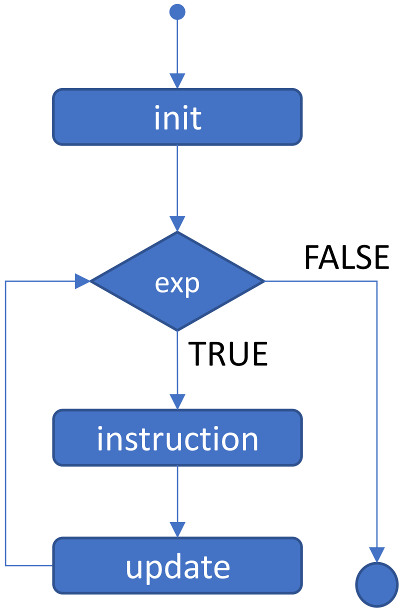
  { ../LOOPS/base.cc}

The iterative instruction *for*

- The iterative instruction: “for”
- Syntax: for (init; exp; update) { instruction }

  init is an initialization instruction for the control variable
  exp is a Boolean expression
  update is the update instruction for the control variable
  instruction can be a complex instruction

- The execution:
  1. execution of init
  2. execution/computation of the expression exp
  3. If exp is true, instruction is executed, then update is executed,
  4. repeat from step.2



Example

```
…
for (int index=0; index<5; index++) {
   //block instructions
   // index can be used here
}
```

- Note: The control variable is defined and initialized internally to the cycle
  - E.g., for (int index=0; index<MAXDIM; index++) {<index occurs only in this block>}
- Typically, among the different iterative instruction is the most used one

for

```
using namespace std;
#include <iostream>

int main() {

  int n,sum;
  cout << "How many integers do you want to sum? ";
  cin >> n;

  sum = 0;
  for (int index=0; index<=n; index++){  //index defined here
    sum += index;
  }

  cout  << "Sum = "  << sum   << endl;
  return 0;

}
```

```
$ g++ sumintegers_for.cc
$ ./a.out
How many integers do you want to sum?  3
Sum = 6
```

{ ../LOOPS/sumintegers_do.cc}

for

```
1.     using namespace std;
2.     #include <iostream>

3.     int main() {

4.       int n,sum;
5.       cout << "How many integers do you want to sum? ";
6.       cin >> n;
7.
8.       sum = 0;
9.       for (int index=0; index<=n; index++){
10.        sum += index;
11.      }
12.
13.      cout  << "Sum = "  << sum   << endl;
14.      return 0;

15.  }
```

line	n	index	sum	*Std out*
4	-		-	
5	-	-	-	How many…
6	*3*	-	-	
8	3	-	0	
9	3	0	0	
10	3	0	0+0	
9	3	1	0	
10	3	1	0+1	
9	3	2	1	
10	3	2	1+2	
9	3	2	3	
10	3	3	3+3	
9	3	4	6	
13	3		6	Sum=6

{ ../LOOPS/sumintegers_do.cc}

Cycle for and cycle while

```
for (init; exp; update)
{
   instructions
}
```

equivalent

```
{  init,
   while (exp) {
      instruction
      update;
     };
}
```

Example

```
for (int index=1; index<0; index++)
{
   x *=2;
}
```

equivalent

```
{  int index=1;
   while (index<10) {
      x *=2;
      index++;
    };
}
```

Examples about the use of the *for*

- Example of product with accumulator (for)
  { ../LOOPS/fact_for.cc}
- Example of sum with accumulator, number of iterations (for)
  { ../LOOPS/series_for.cc}
- Example of sum with accumulator, with exit condition (for)
  { ../LOOPS/series_for1.cc}
- Example of nested for
  { ../LOOPS/doublefor.cc}
- Example of multiple initial condition with for
  { ../LOOPS/series_for1_2init.cc}
- Example of multiple initial condition & multiple exit with for
  { ../LOOPS/series_for1_2init2.cc}
- Example of increment as input given by the user
  { ../LOOPS/minmax.cc}
- Example of double increment
  { ../LOOPS/doublecontrol.cc}

The jump instruction

- **Jump instructions:** break, continue, return 0;
  - How you should not program in C/C++!!!
  - Not use jump instructions!!!

The jump instruction: *break*

- The instruction break ends the cycle

```
1.    while (… ) {
2.       …
3.       break;
4.       ….
5.    }
6.    …
```

Flow:

- from (3) directly to (6) – end of the while block execution

- It needs to be avoided!
- It is better to use an exist condition

Example of simple break (while):

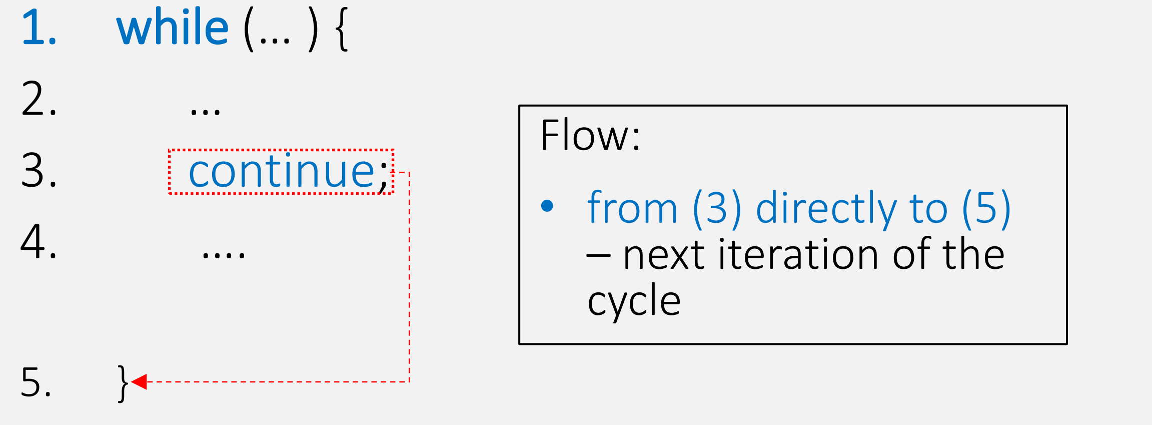
{ ../LOOPS/break_while.cc}

Example of how to avoid a break (while):

{ ../LOOPS/nobreak_while.cc}

The jump instruction: *continue*

- The continue instruction ends the iteration of the cycle under execution and goes to the next iteration



```
1.   while (... ) {
2.        ...
3.        continue;
4.         ....

5.   }
```

Flow:

- from (3) directly to (5) – next iteration of the cycle

- In case of cycle for, the update instruction is skipped
- It needs to be avoided!
- It is better to use of an if instruction

Example of simple continue (while):

{ ../LOOPS/continue.cc}

Example of how to avoid a continue (while):

{ ../LOOPS/nocontinue.cc}

The *return* instruction

- The return instruction ends the cycle and the whole function

```
1.    int main () {
2.    …
3.       while (… ) {
4.          …
5.          return 0;
6.           ….
7.    }
8.    }
```

Flow:

- from (5) directly to (8)
  – end of the function

- It needs to be avoided!
- It is better to use an exist condition

Example of simple return (while):

{ ../LOOPS/return_while.cc}

Example of how to avoid a return (while):

{ ../LOOPS/noreturn_while.cc}