Introduction to programming
2014 - 2015

Corina Forăscu

corinfor@info.uaic.ro

http://profs.info.uaic.ro/~introp/

Course 2: agenda

- Operators (cont.)
- Conversions
- Input-output
- Instructions
- Functions

Operators

- Assignment operator (=)
- Arithmetic operators (+, -, *, /, %)
- Compound assignment (+=, -=, *=, /=, %=)
- Increment and decrement (++, --)
- Relational and comparison operators (==, !=, >, <, >=, <=)
- Logical operators (!, &&, ||)

Bitwise operators (&, |, ^, ~, <<, >>)

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

Bitwise operators - examples

```
01001000 &
10111000 =
--------
00001000
```

```
72 & 184 = ?
```

```
01001000 |
10111000 =
--------
11111000
```

```
72 | 184 = ?
```

```
01001000 ^
10111000 =
--------
11110000
```

```
72 ^ 184 = ?
```

Bitwise operators - examples

```
leftArg << numberBitsToLeft
```

```
7 << 1; //14
//00000111 becomes 00001110
```

```
leftArg >> numberBitsToRight
```

```
7 >> 1; //3
//00000111 becomes 00000011
```

```
~ argument
```

```
~ 0; // 11111111
```

Conditional ternary operator (? :)

condition ? *result-1* : *result-2*

- The `condition` is evaluated (true / false)
- If `condition` is `true` (any non-zero integer), the entire expression evaluates to `result-1`, and `result-2` is not evaluated.
- If `condition` is `false` (zero), the entire expression evaluates to `result-2`, and `result-1` is not evaluated.
- grouping: right –to-left

Conditional operator ?: Example

```
7==5+2 ? 4 : 3
x >= 0 ? x : y
x > y ? x : y
x > y ? x > z ? x : z : y > z ? y : z
```

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

void main(){
    int a=1, b=2, c=3;
    int x, y, z;
    x = a?b:c?a:b;
    y = (a?b:c)?a:b; /* left grouping */
    z = a?b:(c?a:b); /* right grouping */
    cout<< "x=" << x << "\ny=" << y << "/nz="<< z;
}
/* x=2 y=1 z=2 */
```

sizeof()

- accepts one parameter (can be a `type` or a `variable`) and returns the size in bytes of that type or object

```
sizeof(int),        sizeof(double);
sizeof(b*b-4*a*c), sizeof(i);
```

sizeof(char)<= sizeof(short)<= sizeof(int)<= sizeof(long)

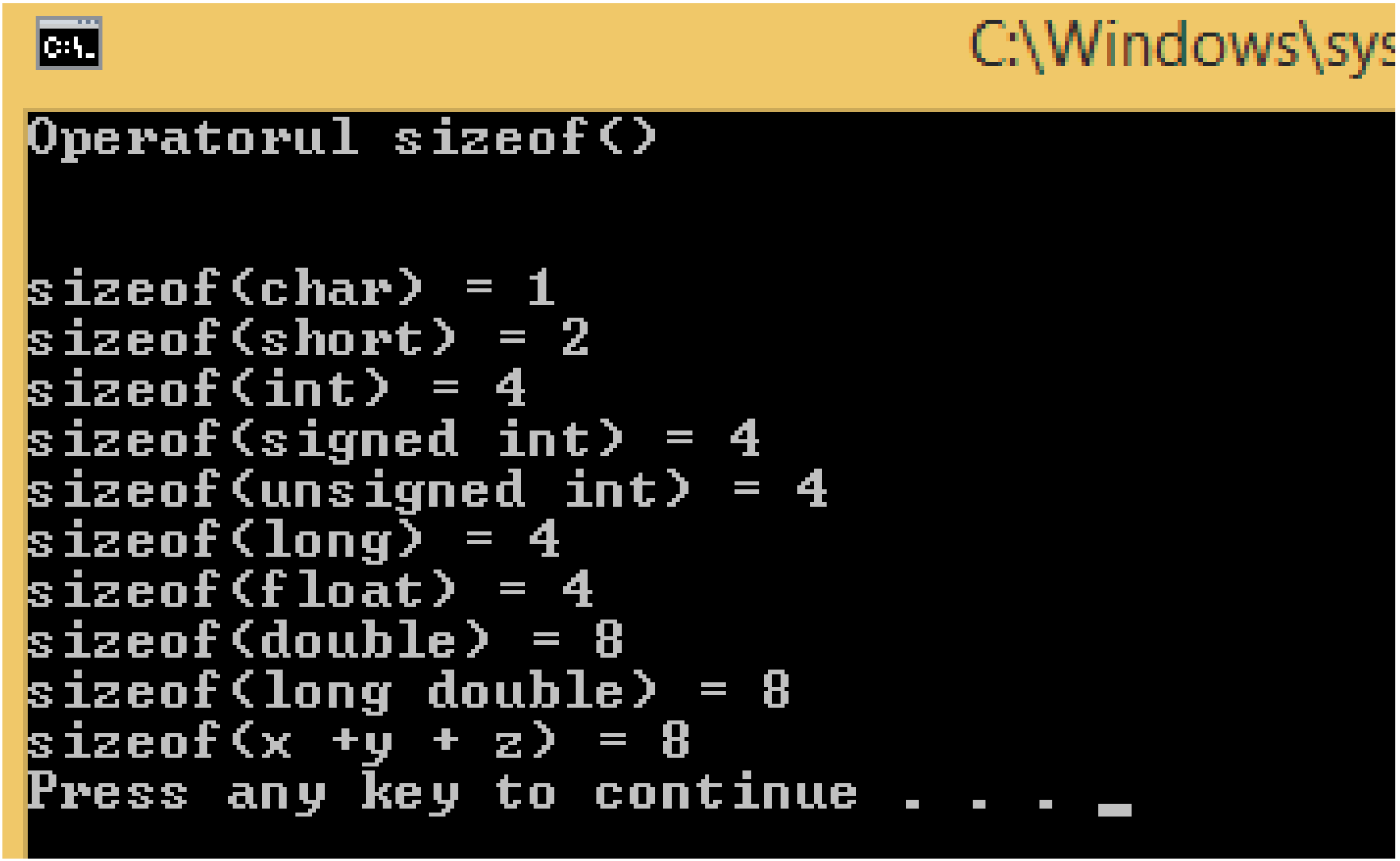
sizeof(signed) = sizeof(unsigned) = sizeof(int)

sizeof(float) <= sizeof(double) <= sizeof(long double)

sizeof() - example

```
#include <iostream>
using namespace std;
void main(){
int x = 1; double y = 9; long z = 0;
cout << "Operatorul sizeof()\n\n\n";
cout << "sizeof(char) = " << sizeof(char) << endl;
cout << "sizeof(short) = " << sizeof(short) << endl;
cout << "sizeof(int) = " << sizeof(int) << endl;
cout << "sizeof(signed int) = " << sizeof(signed int) << endl;
cout << "sizeof(unsigned int) = " << sizeof(unsigned int) << endl;
cout << "sizeof(long) = " << sizeof(long) << endl;
cout << "sizeof(float) = " << sizeof(float) << endl;
cout << "sizeof(double) = " << sizeof(double) << endl;
cout << "sizeof(long double) = " << sizeof(long double) << endl;
cout << "sizeof(x +y + z) = " << sizeof(x + y + z) << endl;
}
```

sizeof() - example



Comma operator (,)

expresWithComma ::= *expr1*, *expr2* , … , *expr-n*

- It is used to separate two or more expressions that are included where only one expression is expected
- All expressions are evaluated, from left to right
- The returned type and value are those of the last expression
- It has the smallest precedence

```
a = 1, b = 2 , ++k + 1;
k != 1, ++x * 2.0 + 1;
for(suma = 0, i = 1; i <= n; suma += i, ++i);
a = (b=3, b+2);
```

typedef

typedef *standard_type* new_name;

The typedef declaration creates an alias that can be used anywhere in place of a (possibly complex) type name:

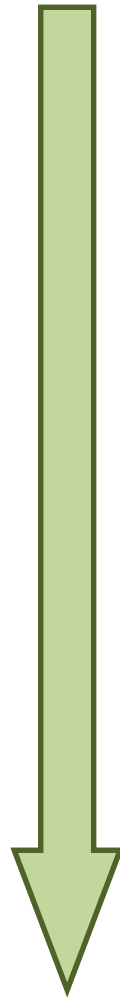
```
typedef short age;
typedef unsigned long ulong;
```

The new identifier can be further used to declare variables, functions

```
age v1, v2;
ulong numberBig;
unsigned long anotherBigNumber
```

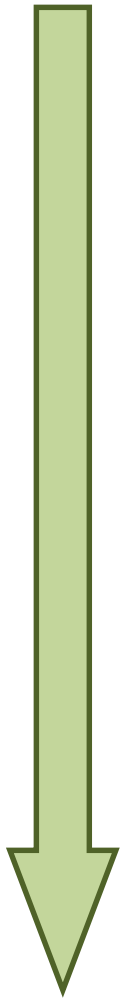
Precedence of operators

Lev	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(type)	C-style type-casting	
4	Pointer-to-member	.* ->*	access pointer	Left-to-right



Precedence of operators

Lev	Precedence group	Operator	Description	Grouping
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or	|	bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction	||	logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -= >>= <<= &= ^= |=	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right



Conversions

- Conversions can occur
  - **explicitly** as the result of a **cast** or
  - **implicitly** as required by an operation.
- Conversions are generally required for the correct execution of a program.
- -> lost or misinterpreted data
- Conversion of an operand value to a compatible type causes no change to the value or the representation

Explicit conversions (*type-casting*)

- **(new_type) expression** // c-like cast notation
- **new_type (expression)** // functional notation

- For expression, it returns a value of type new_type.

- Can add ".0" to literals to force precision arithmetic, but not for variables.

```
(long)('A' + 1.0)
(int)(b*b-4*a*c)
(double)(x+y)/z
float(x*y/z)
x / (float)2
17.0 / 5     //equals 3.4 in C++
```

Type-cast example

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

int main(void){
  int i, j; double x, y, z, t;
  i = 5/2; x = 5/2;
  y = (double)(5/2);  j = (double)5/2;
  z = (double)5/2;   t = 5./2;
  cout << i << ", " << x << ", ";
  cout << y << ", " << j << ", ";
  cout << z << ", " << t << ", " << endl;
  return 0;
}
/* 2, 2, 2, 2, 2.5, 2.5 */
```

Implicit conversions - Arithmetic conversions

- If there is no **`unsigned`** object, everything is converted to the "highest" type, in descending order:

`long double, double, float, long int, int`

- The conversion rules for **`unsigned`** operators depend on (may vary with) the implementation.

Implicit conversion: integer promotion

- The operations are performed starting with the **`int`** type, hence **`char`** and **`short`** are promoted to **`int`.**
- Type-conversion: in an assignment (var = exp), the type of the right expression is converted to the type of the left variable
- In type -conversion there might appear:
    - Loss of precision (**`double ->float ->long int`**)
    - Loss of significant bits (**`long ->int`**)
    - indetermination

Example

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

int main(void){
  char c1 = -126, c2;           /* c1 = 10000010    */
  unsigned char c3, c4 = 255;   /* c4 = 111111111   */
  short s1, s2 = -32767; /* s2=10000000 00000001   */
  short s3 = -1, s4;      /* s3 = 11111111 11111111 */
  s1 = c1;
  cout << "c1=" << (int)c1 << " s1=" << s1 << endl;
  c2 = s2;
  cout << "c2=" << (int)c2 << " s2=" << s2 << endl;
  c3 = s3;
  cout << "c3=" << (int)c3 << " s3=" << s3 << endl;
  s4 = c4;
  cout << "c4=" << (int)c4 << " s4=" << s4 << endl;
  return 0;
}
```

Example (solution)

c1 = 10000010 **s1 = c1;**

c1 = -126, s1 = -126

s2=10000000 00000001 **c2 = s2;**

c2 = 1, s2 = -32767

s3 = 11111111 11111111 **c3 = s3;**

c3 = 255, s3 = -1

c4 = 111111111 **s4 = c4;**

c4 = 255, s4 = 255

```
char c1 = -126, c2;
unsigned char c3, c4 = 255;
short s1, s2 = -32767;
short s3 = -1, s4;
```

Console Input / output

- I/O objects `cin`, `cout`, `cerr` defined in the C++ library `<iostream>`

```
cin >> var;          /* read and store in var from cin */
```

- Must input "to a variable“ of arithmetic or string type (no literals allowed for `cin`)

```
cout << expr;        /* write expr at cout */
```

- Any data can be outputted (variables, constants, literals, pointers, expressions), but no chars

- Cascading – multiple values in one instruction:

```
cin >> var1 >> var2 ... >> varN;
cout << var1 << var2 ... << varN;
```

Input / output

- Reading an int `cin >> variable`

```
std::cout << “Give the value of a: ”;
std::cin >> a;
```

- Displaying an int `cout << expressions`

```
int a = 10;
std::cout << a;
```

Input / output: example - characters

```
/* Character constants */
#include <iostream>
using namespace std;

int main()
{
   char a, b, c, d;
   a = 'A'; b = 65; c = '\101'; d = '\x41';
   cout << a << b << c << d << endl;
   cout << a << (int)a;
   cout << oct << (int)a << hex << (int)a;
   return 0;
}
```

```
A A A A
A 65 101 41
```

Input / output: example – ASCII codes

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

int main ()
{
   short c;
   for(c = 0; c <= 127; c++){ // for(c='a'; c<='z'; c++)
      cout << "ASCII code: " << (int)c;
      cout << ", character: ";
      cout << (char)c << endl;
   }
   return 0;
}
```

Instructions

- statement: ; *expresssion*;
- Compound statement (block): {*declarations instructions*}
- Conditionals: `if` `if-else` `switch-case`
- Loops: `for` `while` `do-while`
- Sequence interruption:
  `continue;` `break;` `return expr;`
- Unconditional jump: `goto`

Statement

instr_expression ::= {*expression*}$_{opt}$;

- Examples:

```
a = b;
a + b + c;
;
cout << a;
sizeof(int);
```

Compound (block) statement

{{*statements* | *instructiuns*}$_{0+}$}

{ *statement1; statement2; … statement3;* }

- The entire block is considered a single statement (composed itself of multiple substatements)
- a generic statement in a flow control statement can be either a simple statement or a compound statement.

Compound (block) statement: examples

```
{
  int a=3, b=10, c=7;
  a += b += c;
  cout << a << ", " << b <<  ", " << c; // ?, ?, ?
}
```

```
if (x > y){
   int temp;
   temp = x; x = y; y = temp;
   cout << x << y;
}
```

```
{
    int a, b, c;
    {
        b = 2; c = 3; a = b += c;
    }
    cout << "a= " << a <<endl;
} // ?
```

Selection statements

if şi if-else

instr_if ::= `if (<boolean_expression>) {<yes_statement>;}`

instr_if-else ::= `if (<boolean_expression>) {<yes_statement>;} else {<no_statement>;}`

boolean_expression consists of :

- Arithmetical expressions
- Comparison operators: `==, !=, <, <=, >, >=`
- Logical conectors: `&&, ||, !`

Selection statements: examples

```
if(b == a)
  area = a*a;
```

```
if(a%2) if(b%2) p = 1; else p = 2;
```

```
if(x < y)
  min = x;
else
  min = y;
```

```
if(a%2){
  if(b%2)
    p = 1;
}
else p = 2;
```

Selection statements: examples

```
int i, j, k, l, max;
if(i>j)
      if(k>l)
            if(i>k) max = i;
            else max = k;
      else
            if(i>l) max = i;
            else max = l;
else
      if(k>l)
            if(j>k) max = j;
            else max = k;
      else
            if(j>l) max = j;
            else max = l;
```

“Dangling else Problem”

```
int a=1, b=2; // b=3
if (a == 1)
      if (b == 2) // b=2
            cout << "*****\n";
else
      cout << "ooooo\n";
```

- The rule is: **`else`** is attached to the nearest **`if`**!
- !! Equality operator vs. assignment operator!!

Multiway `if-else`

```
if ( condition-1 ) {
     instructions-1;
 }
 else if ( condition-2 ) {
     instructions-2;
 ...
 }

 else if ( condition-n ) {
     instructions-n;
 }
 else {
instructions-for-all-other-possibilities;
 }
```

If-else. Example

```
int main(void){
    float x, y, result;
    char op;
    cout << "Expression:(number operator number – NO WHITESPACES)\n";
    cin >> x >> op >> y;
    if(op == '+')
          result = x+y;
    else if(op == '-')
           result = x-y;
    else if(op == '*')
           result = x*y;
    else if(op == '/')
            result = x/y;
    else{
   cout << "Error in writing the expression!";
          return 1;
    }
    cout << “The result is : " << rezultat << "\n";
    return 0;
}
```

The switch statement

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
  .
  .
  default:
     default group of statements
}
```

```
!! case n:
!! case (1..3):
```

Execution "falls thru" until **break**

switch

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

```
if (x == 1)
 {
  cout << "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
  cout << "value of x unknown";
  }
```

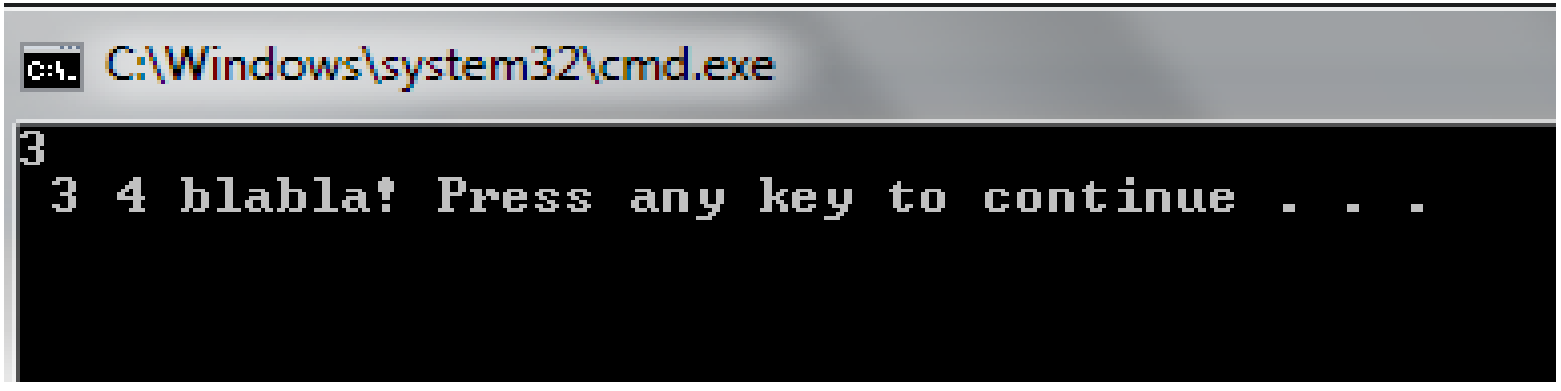
Biggest use: MENUs

- Provides clearer "big-picture" view
- Shows menu structure effectively
- Each branch is one menu choice

switch. Example

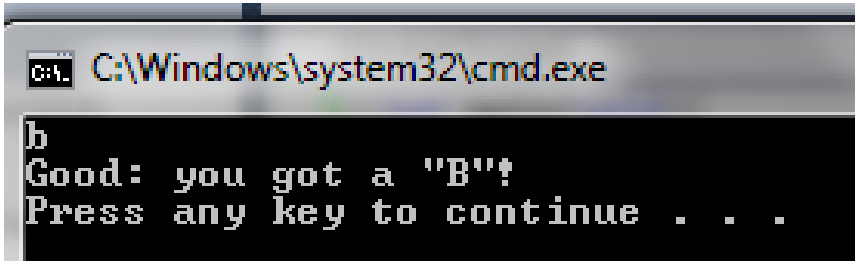
```
int i;
cin >> i;
switch(i){
case 1: cout << " 1";
case 2: cout << " 2";
case 3: cout << " 3"; // break;// ???
case 4: cout << " 4";
default: cout << " blabla! ";

//3
```



switch. Example (2)

```
char eval;
cin >> eval;
switch (eval) {
  case 'A':
  case 'a':
  cout << "Excellent: you got an \"A\"!\n";
  break;
  case 'B':
  case 'b':
  cout << "Good: you got a \"B\"!\n";
  break;
  }
```



Loops

- 3 Types of loops in C++
  - **`while`**
    - Most flexible
    - No "restrictions"
  - **`do-while`**
    - Least flexible
    - Always executes loop body at least once
  - **`for`**
    - Natural "counting" loop

while

```
while(boolean_expression) statement;

while (boolean_expression){
        statement;
}
 next-statement;
```

- *boolean_expression* is evaluated:
  - If it is true, *statement* is executed and the control is passed to the beginning of the **while** statement
  - If it is false, *next-statement* is executed.

- *statement* is executed 0 or more times.

while. Example

```
int n, i=1, factorial=1; // Initialization
cin>>n;
while (i++<n) // Loop condition, update expression
    factorial *= i; // Loop body
cout<<factorial;
```

do..while

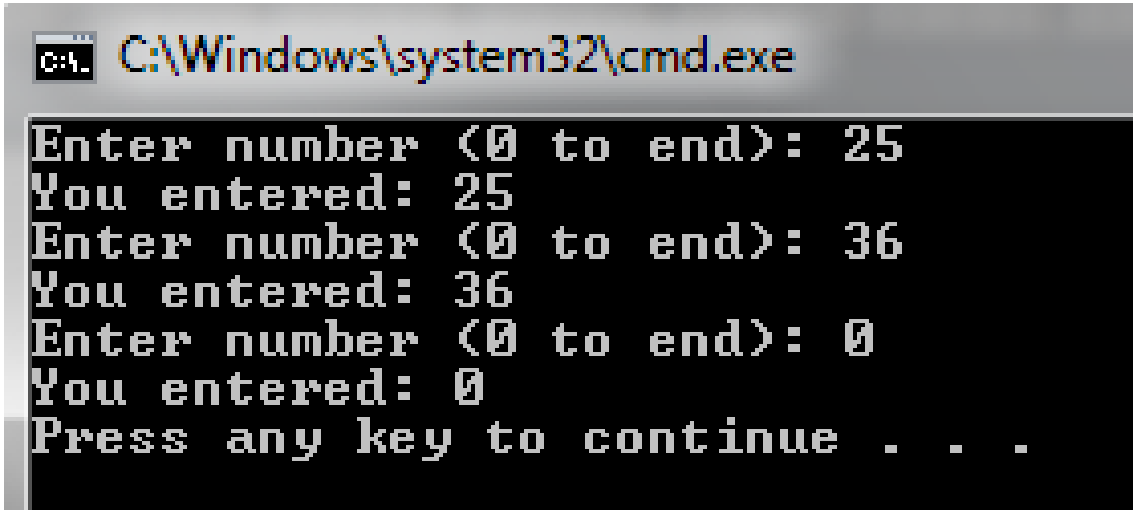
```
do statement while(condition);

do{
        statement
} while (condition);
next_statement;
```

- ***statement*** is executed (at least once).
- ***condition*** is evaluated*:*
  - If it is `true`, the control is passed at the beginning of the **`do..while`**;
  - If it is false, ***`next_statement`*** is executed.

do..while. Example

```
unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
```



Example - calculator

```
int main(void){
    float x, y, result;
    char op, c;
    int ERROR;
    cout << "Calculator for expressions of form \n number operator number \n";
    cout << "Use the operators + - * / \n";

do{
       ERROR = 0;
       cout << "Expression: ";
       cin >> x >> op >> y;
       switch(op){
         case '+': result = x+y; break;
         case '-': result = x-y; break;
         case '*': result = x*y; break;
         case '/': if(y != 0) result = x/y;
                   else {cout << "Division through zero!\n"; ERROR = 1;}
                   break;
         default : {cout << "Unknown operator!\n"; ERROR = 1;}
       }
       if(!ERROR)
    cout << x << " " << op << " " << y << " = " << result;
       cin.sync();
       do{cout << "\n Do you want to continue (y/n)?"; c = getchar();
       } while (c != 'y' && c != 'n');
  } while (c != 'n');
  cout << "See you later!\n";
  return 0;
}
```

for

```
for (expr-initial; expr-cond; expr-in/decrement)
statement;

for (expr1; expr2; expr3){
        statements;
    }
    next_statements
```

- One, two or even all the three expressions can miss; the two separators (;) are compulsory.

for

- If *statement* does not include a **continue** and *expr-cond* is indicated, then **for** is equivalent with :

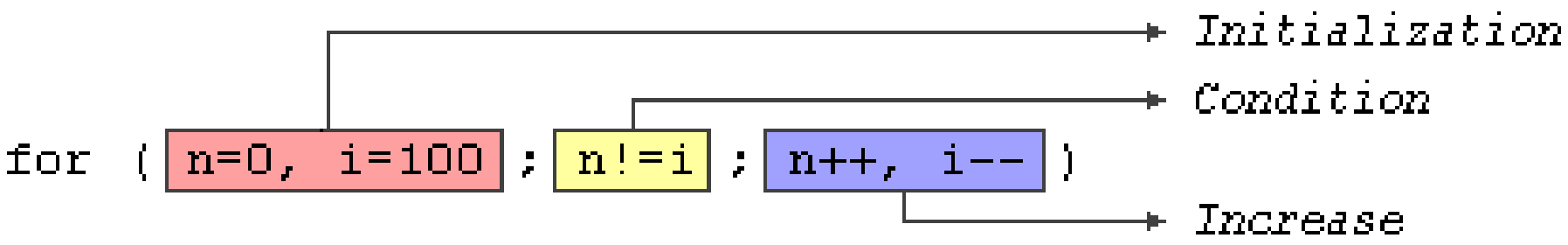
```
expr-init;
while(expr-cond){
        statement;
        expr-in/decrement;
}
next_statement
```

- If *statement* includes a **continue**, this is passing the control to the *expr-in/decrement*.

for. Examples

```
int n;
cin>>n;
  for (; n>0; n--) {
    cout << n << ", ";
  }
  cout << "GATA!\n";
```

```
C:\Windows\system32\cmd.exe
5
5, 4, 3, 2, 1, GATA!
Press any key to continue . . .
```



for. Examples

```
i = 1;
suma = 0;
for(;i <= N;++i) suma += i;

i = 1;
suma = 0;
for(;i <= N;) suma += i++;

i = 1;
suma = 0;
for(;;) suma += i++; // infinit loop
```

Sequence interruptions

- **`break;`**
  - It refers to the nearest loop or `switch` statement.
  - The loop / switch is exited from and the control is passed to the next statement

- **`continue;`**
  - It refers to the nearest loop (`for`, `while`, `do..while`).
  - The current execution is interrupted and the control is passed to the next loop

- **`goto;`** Edsger Dijkstra (1964) GOTO considered harmful
  - It permits to jump to a given section of the program, previously identified through its starting point

- **`return expr;`** *or* **`return;`**
  - In functions, it interrupt the execution and the control is passed to the appellant, including the returning of the *expr* expression.

Examples: sequence interruptions

```
int n;
 for (n=10; n>0; n--)
 {
   cout << n << ", ";
   if (n==3)
   {
     cout << "countdown aborted!";
     break;
   }
 }
```

```
for (int n=10; n>0; n--) {
   if (n==5) continue;
   cout << n << ", ";
 }
 cout << "FIRE!\n";
```

```
int n=10;
 loop:
 cout << n << ", ";
 n--;
 if (n>0) goto loop;
 cout << "FIRE!\n";
 return 0;
```

Example: for..continue

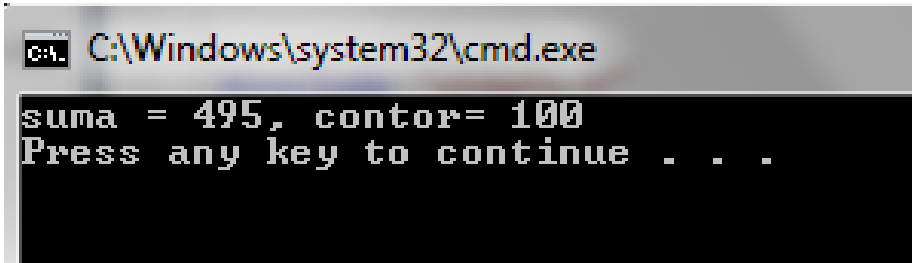
```
 int i, sum=0;
   for(i = 1; i<=N; i++){
if(i%3 != 0) continue;
sum+=i;
   }
   cout << “sum = ” << sum; /* suma multiplilor de 3 până la N */
```

Tip: iterative statements

- The comparative operators are preferred in the iteration control, instead of non-equality operators

```
int main(){
int contor = 0;
double suma = 0.0, x;
for(x = 0.0; x != 9.9; x += 0.1){
        suma += x;
        ++contor;
}
cout << "suma = " << suma << ", contor= " << contor << "\n";
return 0;
}
```

- Use **`x < 9.9`** instead of **`x != 9.9`**



Functions

```
int sum(int n)
{
  int s = 0;
  int i;
  for(i=1; i<=n; ++i)
    s += i;
  return s;
}
```

Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  - Divide & Conquer
  - Readability
  - Re-use
- Your "definition" can go in either:
  - Same file as main()
  - Separate file so others can use it, too

Components of Function Use

- 3 Pieces to using functions:
  - Function Declaration/prototype
    - Information for compiler
    - To properly interpret calls
  - Function Definition
    - Actual implementation/code for what function does
  - Function Call
    - Transfer control to function

Function Declaration

- Also called function prototoype
- An "informational" declaration for compiler
- Tells compiler how to interpret calls
  - Syntax:
    ```
    <return_type> FuncName(<formal-parameter-list>);
    ```
  - Example:

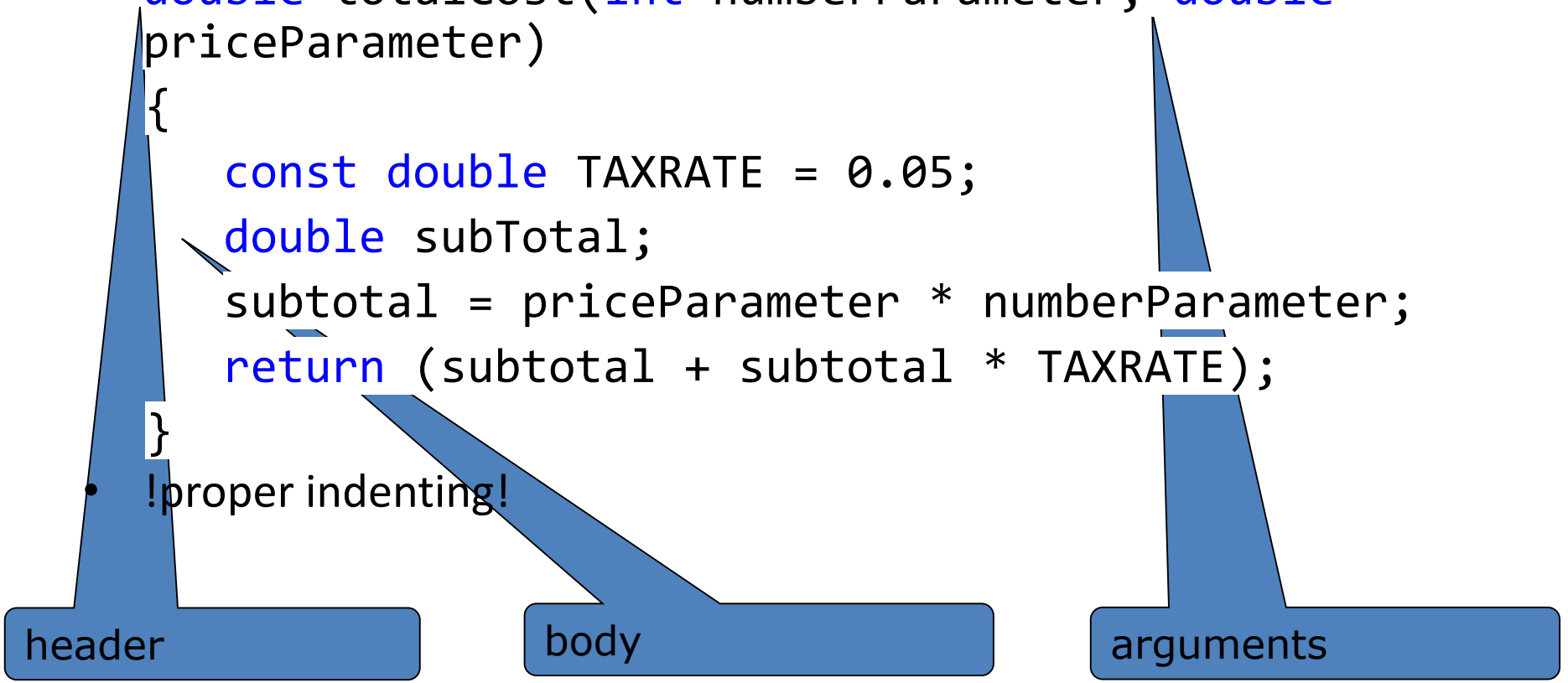
    ```
    double totalCost(int numberParameter, double
    priceParameter);
    ```
- Placed before any calls
  - In declaration space of `main()`
  - Or above `main()` in global space

Function Definition

- Implementation of function
- Just like implementing function `main()`
- Example:

```
double totalCost(int numberParameter, double
priceParameter)
{
   const double TAXRATE = 0.05;
   double subTotal;
   subtotal = priceParameter * numberParameter;
   return (subtotal + subtotal * TAXRATE);
}
```

- !proper indenting!



Function Definition Placement

- Placed after function `main()`
  - NOT "inside" function `main()`!
- Functions are "equals"; no function is ever "part" of another
- Formal parameters in definition
  - "Placeholders" for data sent in
    - "Variable name" used to refer to data in definition
- return statement
  - Sends data back to caller

Function Call

- Just like calling a predefined function

```
double bill;
bill = totalCost(number, price);
```

- Arguments here: number, price
  - Recall arguments can be literals, variables, expressions, or combination
  - In function call, arguments often called "actual arguments“ because they contain the "actual data" being sent

Functions: example

```
void swap(int x, int y){
    int temp = x; x = y; y = temp;
    cout << "x=" << x<< ",y=" << y<< "\n";
}

int main(void){
    int a = 2, b = 3;
    swap(a, b);    // x = 3, y = 2
    cout << "a=" << a<< ",b=" <<b <<"\n";
    // a = 2, b = 3
```

Glossary

- operator
- precedence
- priority
- Grouping
- Conversions: implicit / explicit (type casting)
- Input / output
- (Block)statement

Glossary

- branching statements
  - if-else, switch
- loop statements
  - while
  - do-while
  - for
- branching statements
  - if-else, switch
- Sequence interruption:
  - continue; break; return;
- Unconditional jump: goto
- functions
  - Declaration, definition, call