Programming, numerics and optimization

Lecture 2: Programming basics I

Łukasz Jankowski
ljank@ippt.pan.pl

Department of Intelligent Technologies
Institute of Fundamental Technological Research
Room 4.32, Phone +22.8261281 ext. 428

March 10, 2026

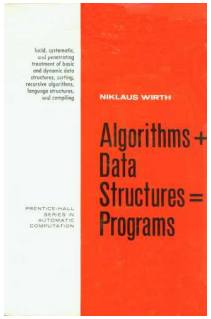
Outline

Outline

Types and variables

Niklaus Wirth

Algorithms + Data Structures = Programs



Each variable (or an object, essentially a "data piece") in a C/C++ program

- has a type assigned and
- is accessible via an identifier.

Data types

Each variable in a C/C++ program has a *type* assigned.

All data are stored in computer memory, which is organized in bytes. The type is necessary since

- variables of different types require different number of bytes for storage. Usually,
  - a single byte has to be allocated to hold a single char,
  - four bytes for an integer number,
  - eight bytes for a floating-point number, etc.
- data of different types are interpreted and processed differently. Failure to do so leads to meaningless results.

Data types

Fundamental and compound types

Types in C++

1. Fundamental
   - Integral (**signed** by default):
     char, **short**, **int**, **long** as well as
     unsigned char, **unsigned short**, **unsigned int**, **unsigned long**
   - Floating-point: **float**, **double**, **long double**
   - Boolean: **bool**
   - **void** (absence of type information)

Some of the keywords can be omitted, for example

short = **signed short** = **short int** = **signed short int**
long = **signed long** = **long int** = **signed long int**
unsigned long = **unsigned long int**

Data types

Fundamental and compound types

Types in C++

2. Compound types (built-in or user-defined):
   - Arrays
   - Functions
   - Pointers
   - References
   - Classes/structures
   - Unions
   - Enumerations

> *"Derived [compound] types are merely convenient software-engineering tools for grouping information stored in fundamental data types"*

Data types

Logical type

Originally, there were *no* logical type variables dedicated to store **true**/**false** values in C/C++ (like e.g. boolean in Pascal).
A boolean type **bool** has been added to C in 1999[1] and to C++ in 1993.

However, in C/C++ variables of *any* type (not only **bool**) can be used to represent logical values.

The rule is very simple:

value of 0:	**false**
any other value:	**true**

[1] C99 introduces a new datatype _Bool. The header stdbool.h defines **bool** as a typedef to the keyword _Bool.

Data types

void data type

A **void** data type

- represents an absence of type (and thus no data),
- is used for example when a function
  - needs no arguments or
  - does not return a value.

  and often in the context of pointers (Lecture 3).

You cannot define/declare a variable of type **void**.

Data types

Memory storage size

Size in bytes of a variable of each type is implementation-specific. It can be easily found using the following code:

example	C++ code
1	cout <<**sizeof**(**bool**) <<"bool" <<endl;
1	cout <<**sizeof**(**char**) <<"char" <<endl;
2	cout <<**sizeof**(**short**) <<"short" <<endl;
4	cout <<**sizeof**(**int**) <<"int" <<endl;
4	cout <<**sizeof**(**long int**) <<"longint" <<endl;
4	cout <<**sizeof**(**float**) <<"float" <<endl;
8	cout <<**sizeof**(**double**) <<"double" <<endl;
12	cout <<**sizeof**(**long double**) <<"long double" <<endl;

Variables

In the context of variables, several notions are important:

- Definition, identifier, declaration, initialization
- Scope and lifespan (lifetime)

Variables

Definition

- Associates a type with an identifier and allocates a proper amount of memory to store it.
- For the compiler it means: "Allocate the storage for a variable of the given type and use the given identifier to denote it."

Example

```
int result;   // storage allocation
              // and identifier assignment
cout <<result <<endl;
```

identifier		type		memory storage
result	$\longleftrightarrow$	**int**	$\longleftrightarrow$	??? (4 bytes)

Variables

Valid identifiers

An identifier in C++

- is one or a sequence of more letters, digits and _ characters,
- must begin with a letter,
- cannot match any reserved keyword (**int**, **char**, **bool**, **for**, etc.),
- is case sensitive.

Variables

Declaration

- Associates a type with an identifier.
- No memory storage is reserved (it is assumed to be reserved somewhere else).
- For the compiler it means: "If you find this identifier in the code, treat it as an integer variable."
- The identifier is associated with the storage not in compilation time, but by the linker.

```
extern int result; // defined, e.g., in a library
cout <<result <<endl; // now the compiler knows the
    // type but not the storage location
```

identifier		type		memory storage
result	$\longleftrightarrow$	**int**	$\leftarrow$ // $\rightarrow$	??? (4 bytes)

Variables

Initialization

- Assignment of an initial value to a variable/object (possibly at the moment of its definition).
- Variables in C/C++ are not initialized automatically (besides global variables, which are zeroed).

Example

```
int result;          // definition
result = 178;        // initialization
```

or

```
int result = 178; // definition + initialisation
```

identifier		type		memory storage
result	$\longleftrightarrow$	**int**	$\longleftrightarrow$	$\boxed{178}$ (4 bytes)

Variables

Scope

A variable/object is accessible via its identifier.

- Does each identifier need to be unique?
- Where is it valid? Where can it be used?

Scope of a variable is the part of code in which the compiler uniquely resolves its identifier. Within the scope, the identifier must be unique. In C++ there are three scope types:

Local scope Each pair of { } defines a new nested local scope (functions, loops etc.)

Global scope Identifiers are known throughout the entire program

Class scope Identifiers are known within a given class (object programming, Lecture 4)

Variables

Scope (example)

Global variables: result

Local variables: i, j, result

```
int result;                 //global variable (zeroed)

int main() {
    int i;                  //i not initialized
    cout <<"Global result: " <<result <<endl;
    for(i=1; i<=100; ++i) {
        int result;         //global variable is hidden
        for(int j=1; j<=100; j+=3) {
            result = i*j;
            cout <<"Local result: " <<result <<endl;
        }
    }                       //end of scope of local result
    cout <<"Global result: " <<result <<endl;
    return 0;
}
```

Variables

Lifespan

- When a variable/object is created and destroyed?
- How long does it exist (i.e., how long the memory storage remains allocated)?
  - Entire duration of a program?
  - Somewhat shorter?

A variable/object is

- created, when its definition is encountered;
- destroyed, when the execution leaves its scope (besides the variables defined with the **static** modifier).

Variables

Lifespan (example)

Global: result Local: i, j, result

```
int result;          //result allocated

int main() {
  int i;             //i allocated
  cout <<"Global result: " <<result <<endl;
  for(i=1; i<=100; ++i) {
    int result;      //result allocated (mult.)
    for(int j=1; j<=100; j+=3) { //j allocated (m.)
      result = i*j;
      cout <<"Local result: " <<result <<endl;
    }                //j destroyed
  }                  //local result destroyed
  cout <<"Global result: " <<result <<endl;
  return 0;
}                    //i destroyed
```

Constants

In C++ there are three techniques to define/use constants:

1. Literal constants (literals)
2. Modifier **const**
3. Preprocessor directives (C legacy)

Constants

Literal constants (literals)

Constant type	Literal	C++ type
Integral	'a'	**char**
	28	**int** (decimal)
	034	**int** (octal)
	0x1c	**int** (hex)
	28L	**long int**
	28uL	**unsigned long int**
Floating-point	12.3F	**float**
	−12.3	**double**
	12.	**double**
	5.2e−3	**double**
	12.3L	**long double**
	12.L	**long double**
	5.2e−3L	**long double**
Strings	"I am a string!\n"	**char***

Constants

Constant variables — modifier **const**

If the same constant is used several times throughout the code, it is convenient to define a *constant variable* using the modifier **const**.

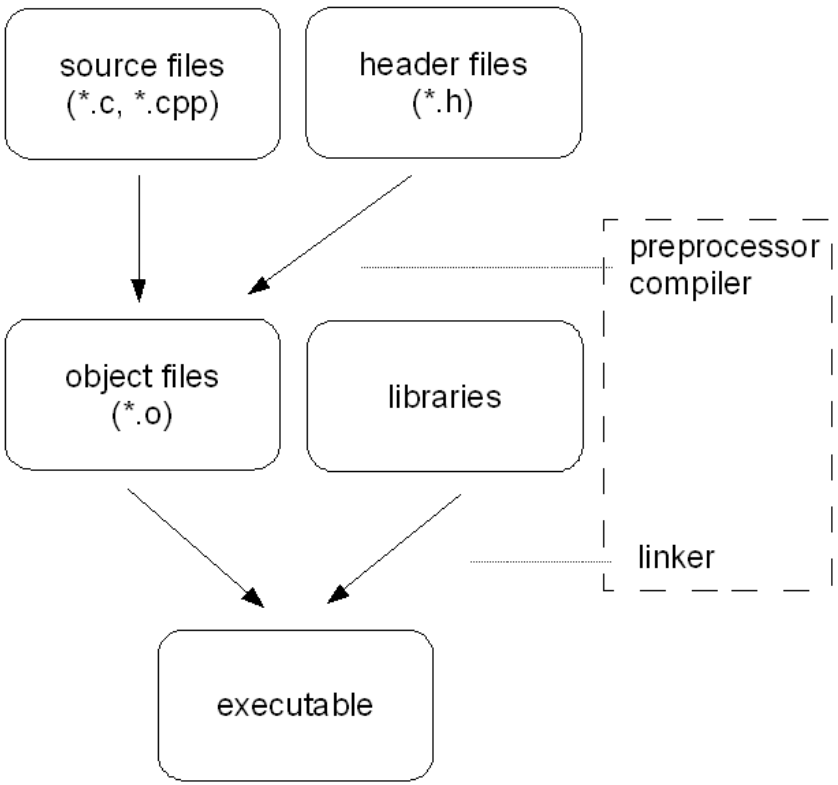
Example

```
const int age = 30;
const double height = 1.78;
```

- The compiler knows that age and height are (constant, so non-modifiable) variables, hence all scope/lifespan rules do apply.
- A constant variable must be immediately initialized to be usable (it is constant and cannot be assigned a value later).

Constants

Preprocessor directives



- The preprocessor runs before the compiler.
- Commands interpreted by the preprocessor are called *directives* and are preceded by a hash #. They are not program statements (not compiled).
- Directives extend only one line (unless backslash is used), no semicolon at the end!
- Directives allow to include files, *perform text substitutions* and test simple conditions.

Constants

Preprocessor directives

Preprocessor text substitutions can be used to define "constants".

Example

The directive

```
#define AGE 70
```

orders the preprocessor to replace in the code all the occurrences of the word AGE with 70.

It is generally better to *avoid* defining constants by **#define**:

- The replacements are blind. In slightly more complex cases it can lead to unexpected results that are hard to trace back, like in "**#define** CTOF(c) 1.8∗c+32".
  2∗CTOF(c) is expanded to 2∗1.8∗c+32 and not to 2∗(1.8∗c+32).
- They happen before the compiler runs, so without any syntax/type check. The compiler knows nothing about AGE.

Outline

(Few important) operators

- Arithmetic:

Addition	+
Subtraction	−
Multiplication	*
Division	/
Modulo	%

The type of the result depends on the type of the arguments.

Typical sources of errors

```
float x = 3/2;      // Result 1 (int -> float)
cout <<x <<endl;
x = 3./2;           // Result 1.5 (float -> float)
cout <<x <<endl;
```

```
int x = -5;
unsigned int y = 5;
cout <<x-y <<endl;  // Unsigned result
```

(Few important) operators

- Relational

Equal	guess == age
Not equal	guess != age

- Unary:

Logical NOT	!**true**	!(guess==age)	!1
Increment	++i	i++	
Decrement	--i	i--	

Example

```
int i = 10;
cout <<i++ <<endl;   //print 10, then increment
cout <<++i <<endl;   //increment, then print 12
```

(Few important) operators

- Logical:

Logical AND	(guess==age) && !error
Logical OR	(guess!=age) || error

- Assignment:

Assignment	age = 60	guess = age
Compound assignment	i+=10	i-=10
	i*=coeff	i/=coeff

Compound assignment operators are shorthands:

- i+=10 is equivalent to i=i+10,
- i/=10 is equivalent to i=i/10, etc.

(Few important) operators

- Conditional (the only three-argument operator in C/C++):

  (guess==age)? "Yes!" : "No."

```
int age = 100, guess;

cout <<"Guess my age: ";
cin >>guess;
cout <<((guess==age)?"Yes!":"No.") <<endl;
   //Note the parentheses (operator precedence)
cout <<(guess==age?"Yes!":"No.") <<endl;
   // is equivalent
```

```
cout <<guess==age?"Yes!":"No." <<endl;
   // is wrong ("<<" is stronger than "?:")
```

Outline

Control flow statements

All statements (instructions) are

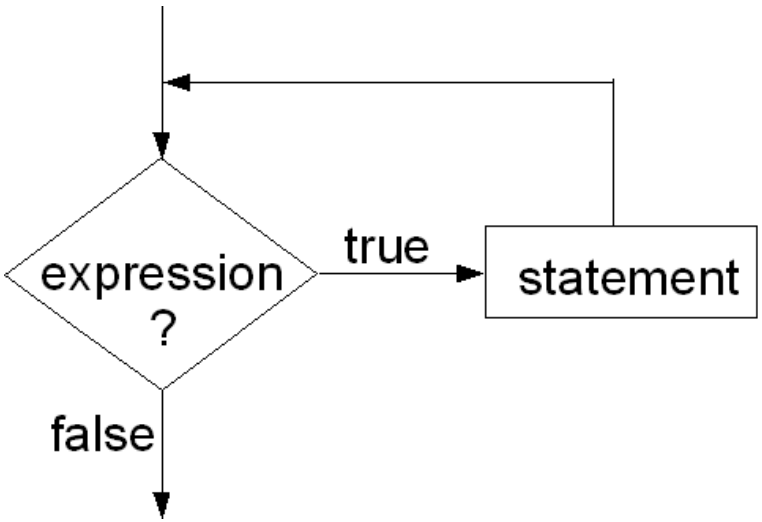
- separated by semicolons ;
- grouped within compound statements: {...}
- executed linearly (in order) unless a *control flow statement* is encountered.

Control flow types:

- linear (default)
- loop: **while**, **do** ... **while**, **for**
- branch (decision making): **if** ... **else**, **switch** ... **case**
- jump: **break**, **continue**, **goto**
- exception handling: **throw**, **try** ... **catch** ... finally .

Loops — while

```
while (expression)
    statement
```

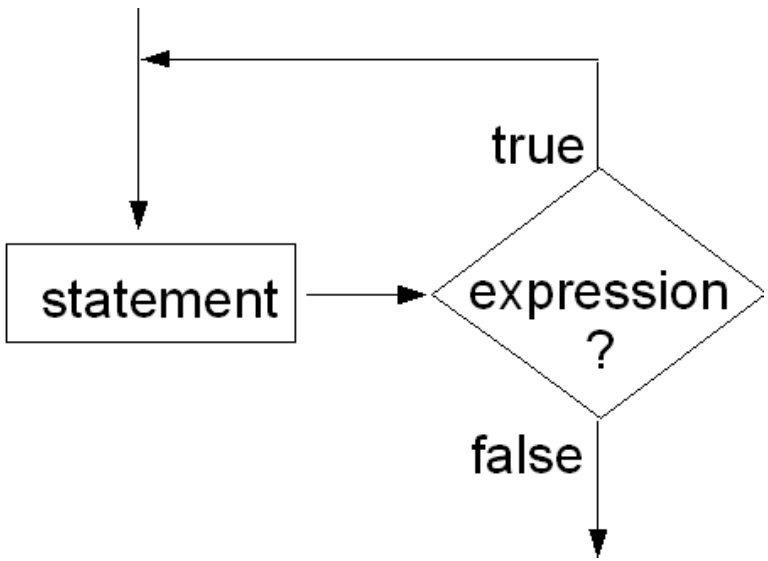



```
int age = 100;
int guess = 0;
while (age != guess) {
    cout <<"Guess my age: ";
    cin >>guess;
}
cout << "Right!" << endl;
```

1. If expression is false at the beginning, the statement is *not executed even once*.
2. The statement can be either a single statement (ending with a semicolon) or a compound statement {...}.

Loops — do... while

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

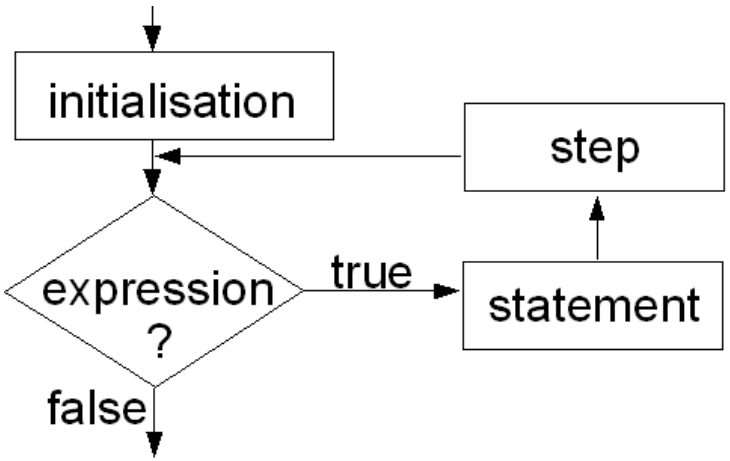



```
int age = 100;
int guess;
do {
    cout <<"Guess my age: ";
    cin >>guess;
} while (guess != age);
cout <<"Right!" <<endl;
```

1. The statement is executed *at least once*.
2. The statement can be either a single statement (ending with a semicolon) or a compound statement {...}.

Loops — for

```
for(initialisation; expression; step)
    statement
```



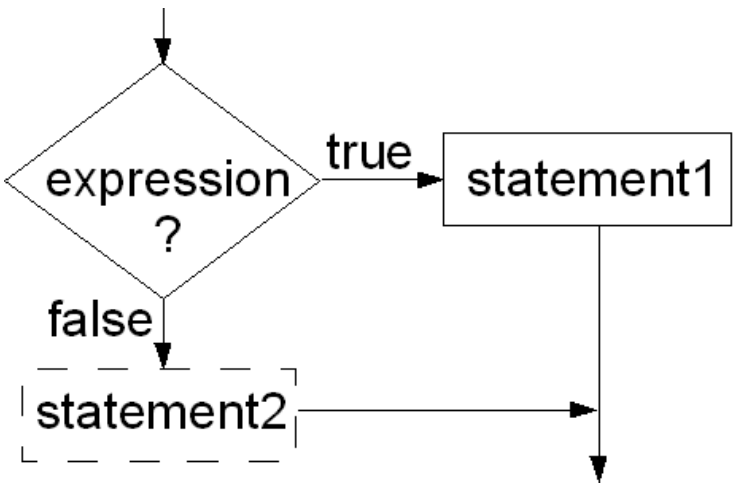

```
for(int age=1; age<=30; ++age){
    cout <<"I am " <<age;
    cout <<" years old!\n";
}
cout <<"End of loop!" << endl;
```

1. The initialization is executed only once.
2. If expression is false at the beginning, the statement *is not executed even once*.
3. Any of initialization , expression and step can be empty.
4. The statement can be a single or a compound statement.

Branches — if ... else

```
if(expression) statement1
```

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




```
int age = 100, guess;
cout <<"Guess my age: ";
cin >>guess;
if (guess<age)
    cout <<"More!\n";
else if (guess>age)
    cout <<"Lower!\n";
else cout <<"Right!\n";
```

1. The statement2 can be another **if** statement.
2. Both statements can be single or compound statements.

Branches — switch ... case

```
int guess;

cout <<"Guess my age: ";
cin >>guess;
switch (guess) {
  case 50:  cout <<"No, but my brother is 50.\n";
            cout <<"Try again." <<endl;
            break;
  case 100: cout <<"Right!\n";
            break;
  case 150: cout <<"No, but my sister is 150.\n";
            cout <<"Try again." <<endl;
            break;
  default:  cout <<"No. Try again" <<endl;
            break;
}
```

Branches — switch ... case

```
switch (guess) {
    case  50: cout <<"No, but my brother is 50!\n";
              break;
    case 100: cout <<"Right!\n";
              break;
    default:  cout <<"No.\n";
              break;
}
```

1. The expression (guess) must be of an integral type (or equivalent). Floating point numbers, etc. are not allowed.
2. If a match is found, the execution starts with the corresponding **case**.
3. The code executes until a **break** is encountered.
4. If no match is found, the execution starts with the **default** (which is an optional part and may not occur).

Jumps — break, continue

The flow of any loop can be controlled by

- **break**, which immediately quits the loop (in case of many nested loops, it quits just a single loop, not all of them)
- **continue**, which stops the execution of only the current iteration of the loop (the loop is resumed at the next iteration).

break and **continue** statements are generally considered poor programming style and usually can be avoided.

Jumps — break, continue

break and **continue** statements are generally considered poor programming style and usually can be avoided.

break, a very simplistic example

```
int age = 100;
int guess;
do {
   cout <<"Guess it: ";
   cin >>guess;
   if(guess==age) break;
} while (true);
cout <<"Right!" <<endl;
```

```
int age = 100;
int guess;
do {
   cout <<"Guess it: ";
   cin >>guess;
} while (guess != age);

cout <<"Right!" <<endl;
```

Jumps — break, continue

break and **continue** statements are generally considered poor programming style and usually can be avoided.

continue, a very simplistic example

```
int age;
for(age=1;age<=18;++age){
   cout <<age <<endl;
   if(age>=10) continue;
   cout <<"Less ";
   cout <<"than 10...\n";
}

cout <<"An adult" <<endl;
```

```
int age;
for(age=1;age<=18;++age){
   cout <<age <<endl;
   if(age<10) {
      cout <<"Less than ";
      cout <<"10...\n";
   }
}
cout <<"An adult" <<endl;
```

Jumps — goto the infamous

```
    ...
    goto foundIt;
    ...
foundIt:
    ...
```

The statement **goto** label;

- Forces the execution to jump to the statement labelled with label.
- It is poor programming style, (probably) unless it is used to leave several complicated nested loops.

Jumps — goto the infamous

goto is poor programming style, (probably) unless it is used to leave several complicated nested loops.

A very simplistic example

```
int result, i, j;
for (i=1; i<=100; ++i) {
  for (j=1; j<=100; j+=3) {
    result = i*j;
    if (result > 3211)
      goto foundIt;
  }
}
foundIt:
  cout <<i <<" x " <<j <<" = " <<result <<endl;
```

Outline

Functions

Procedural programming allows

- *reusable code* parts to be collected and separated in procedures (functions, routines) for repetitive or later use,
- programs to be structured and better readable.

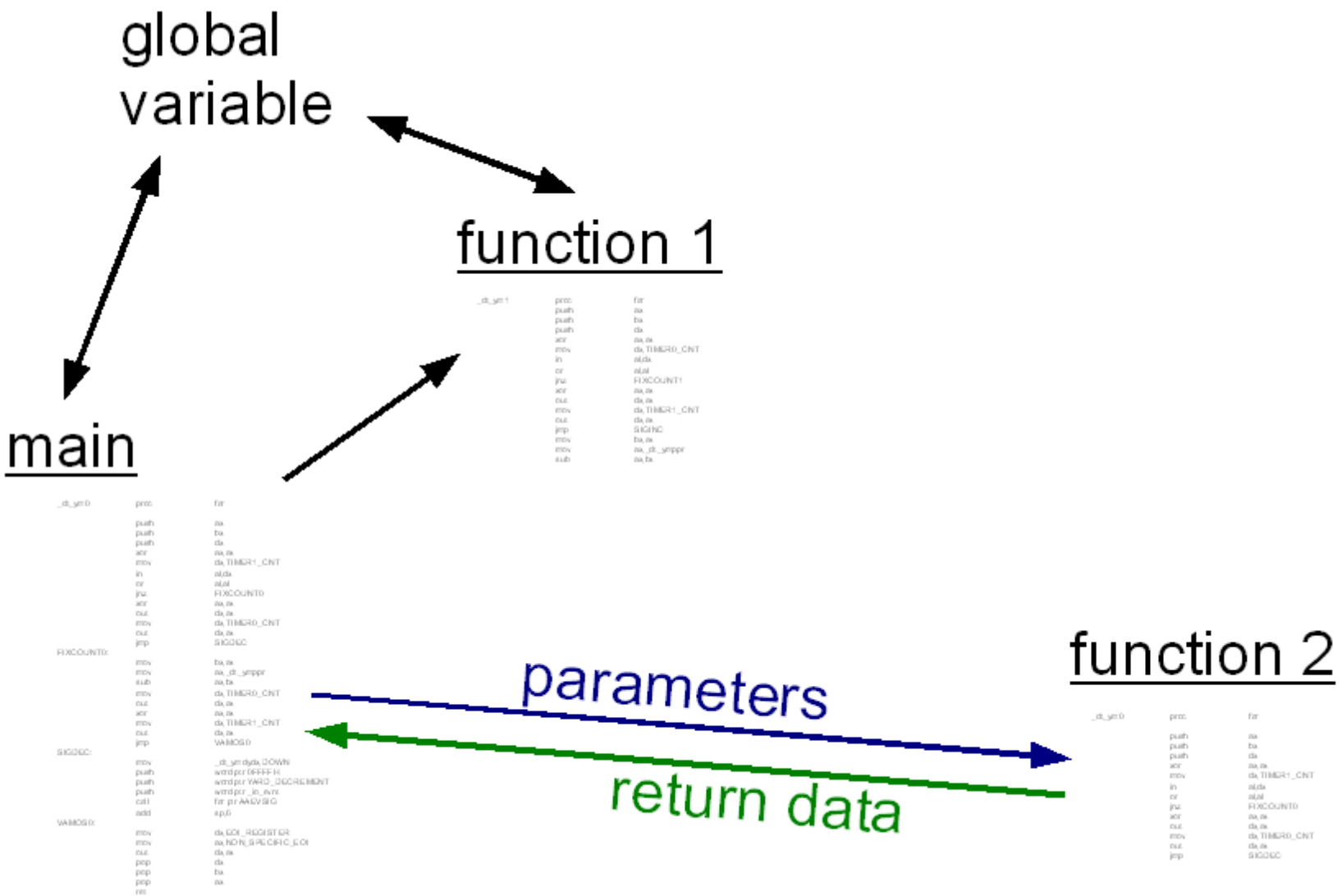
In C/C++

- all procedures are called *functions* (even if they return no data or take no arguments).
- execution of a program starts with the main() function.

In C/C++, data sharing between functions is possible via

- global variables or
- arguments and return data (which is usually preferred).

Functions


global
variable
function 1
main
function 2
parameters
return data

Definition, declaration, call

Example

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

int add(int x, int y) { //definition of add
   int r = x + y;
   return r;
}

int main() {              //definition of main
   int r;
   r = add(2,7);          //call to add
   cout <<"2 + 7 = " <<r <<endl;
   return 0;
}
```

Definition, declaration, call

```
int add(int x, int y) { //definition of add
   int r = x + y;
   return r;
}

int main() {                //definition of main
   int r;
   r = add(2,7);            //call to add
.....}
```

Note the corresponding elements

```
int       add(   int x,   int y   ) {...}   // definition
↕         ↕      ↕        ↕
r    =    add(    2,      7      );         //call
```

Definition, declaration, call

Example

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

int add(int x, int y);  // declaration of add
                        // int add(int, int);
int main() {
  int r=add(2,7);       // call already possible
  cout <<"2 + 7 = " <<r <<endl;
  return 0;
}

int add(int x, int y) { // definition of add
  return x + y;         // (later than call)
}
```

Definition, declaration, call

A source code of a program consists of a set of source files. In principle, each one of them should be doubled:

1. library.h (header file) contains all declarations (of functions, classes, etc.),
2. library.cpp contains all the corresponding definitions.

Functions defined in one file (library.c) can be used in another file by including at its beginning the corresponding header file (library.h). For your own libraries located in the current folder use

```
#include "library.h"
```

For standard libraries use

```
#include <library>
// or #include <library.h>
```

void type arguments

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

int ask(void) { //no arguments needed
   int r;
   cin >>r;
   return r;
}

int main() {    //the keyword void can be skipped
   cout <<"Your age? ";
   cout <<"Thanks, " <<ask() <<endl;
     //in call (empty) parentheses are obligatory
   return 0;
}
```

void return type

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

void printAge(int age) { // no need for
   cout <<"Age: " <<age <<endl;  // any return
       //value
}

int main() {
   printAge(100);              // return value not used
   return 0;
}
```

Passing arguments

Arguments to a function[2] can be passed

- by value
- by reference

If an argument is passed *by value*, the function operates on a copy of the originally passed data. The original data cannot be thus altered inside the function.

If an argument is passed *by reference*, the function operates on the original data, so that it can be altered inside the function[3].

[2] Essentially, the same applies to return data.

[3] In C, passing by reference corresponds to passing a pointer to a variable instead of the variable itself.

Passing arguments by value

```
int add(int x, int y) {
  x += y;
  return x;
}
```

- x and y are *copies* of a and b (in different memory storage).
- Modifications to x and y *do not* alter a and b.

```
int main() {
  int a=2, b=7;
  cout <<a <<"+" <<b <<" = ";
  cout <<add(a,b) <<" = ";
  cout <<a <<"+" <<b <<endl;
  return 0;
}
```

console:
2+7 = 9 = 2+7

Passing arguments by reference

```
int add(int &x, int &y) {
  x += y;
  return x;
}
```

- x and y are other names for a and b (x, y reference a, b).
- Modifications to x and y do alter a and b.

```
int main() {
  int a=2, b=7;
  cout <<a <<"+" <<b <<" = ";
  cout <<add(a,b) <<" = ";
  cout <<a <<"+" <<b <<endl;
  return 0;
}
```

console:
2+7 = 9 = 9+7

Passing arguments by reference

```
float doubleMe(float &x){
  return (x *= 2);
}
```

Function doubleMe()

- doubles the passed variable
- returns the doubled value.

```
int main() {
  float a=2.1;
  cout <<"a = " <<a <<endl;
  doubleMe(a);
  cout <<"a = " <<a <<endl;
  cout <<doubleMe(a) <<endl;
  return 0;
}
```

console:

a = 2.1

a = 4.2

8.4

Passing constant arguments by reference

```
void printMe(float &x) {
  cout <<x <<endl;
}
```

Wrong

```
const float a = 2.1;
printMe(a);
printMe(2.1);
```

Use the modifier const to pass constant arguments by reference

```
void printMe(const float &x) {
  cout <<x <<endl;
}
```

Wrong: const arguments cannot be modified

```
float doubleMe(const float &x) {
  return (x *= 2);
}
```

Passing arguments

Default values of arguments

Default second argument

```
int add(int x, int y=1) {
  return x+y;
}
```

- If y is omitted in a call, 1 is used instead.
- "=1" is stated *only once* (in declaration, if present)

```
int main() {
  cout <<"3+4 = " <<add(3,4);
  cout <<endl;
  cout <<"3+1 = " <<add(3);
  cout <<endl;
  return 0;
}
```

console:

3+4 = 7

3+1 = 4

Passing arguments

Default values of arguments

```
int add(int x, int y=1);

int add(int x, int y) {
  return x+y;
}
```

- If y is omitted in a call, 1 is used instead.
- "=1" is stated *only once* (in declaration, if present)

```
int main() {
  cout <<"3+4 = " <<add(3,4);
  cout <<endl;
  cout <<"3+1 = " <<add(3);
  cout <<endl;
  return 0;
}
```

console:

3+4 = 7

3+1 = 4

Overloaded functions

Several functions can have the same name, if they

- differ in argument types or
- differ in argument number.

```
void changeMe(int &x, int &y){
   int a=x;
   x=y;
   y=a;
}
```

```
int changeMe(int &x){
   return (x -= 1);
}
```

```
float changeMe(float &x) {
   return (x /= 2);
}
```

Outline

Homework 2 (15 points)

Loops and functions (Fibonacci numbers)

Definition

The first two numbers x_0 and x_1 are set arbitrarily (but usually to 0 and 1). For the successive numbers the recurrence formula is used:

$$x_n \quad \longleftarrow \quad x_{n-1} + x_{n-2}.$$

Example

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Homework 2 (15 points)

Loops and functions (Fibonacci numbers)

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

int main() {
  int x0, x1, x2, iter = 2;

  cout << "Input F(0) = ";
  cin >> x0;
  cout << "Input F(1) = ";
  cin >> x1;
  while (iter < 40) {
     x2 = x0 + x1;
     cout <<"F(" <<iter++ <<")\t=\t" <<x2 <<endl;
     x0 = x1;
     x1 = x2;
  }

  return 0;
}
```

Homework 2 (15 points)

Loops and functions (Fibonacci numbers)

1. (4 points) Rewrite the example to use the **do** ... **while** loop.
2. (4 points) Rewrite the example to use the **for** loop.
3. (4 points) Write a function `fibo(int x0, int x1, int n);` that would calculate and print first `n` Fibonacci numbers starting with `x0`, `x1`:
   - Do not forget to take into account the `n==0` and `n==1` cases.
   - Modify the `main()` function to use your `fibo (...)` function.
4. (3 points) Start with 0 and 1, and print the first 50 numbers. What happens and why?

E-mail the answer and the source code to `ljank@ippt.pan.pl`.