Methodology and Programming Techniques

Department of Telecommunications, IEiT

dr inż. Jarosław Bułat

kwant@agh.edu.pl

Outline

- Function main()
- Preprocessor instructions
- Variable modificators
- Library
- Function printf()

AGH

main()

function arguments

main() - arguments

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

// ./ex01 -v -i file.txt

//int main(int argc, char *argv[]){
int main(int argc, char **argv){

    cout << argc << endl;
    cout << argv[0] << endl;
    cout << argv[1] << endl;
    cout << argv[2] << endl;
    cout << argv[3] << endl;
}
```

- Application arguments:
- argc: number of arguments
- argv: array containing pointers to arrays with single program arguments (pointer to pointer)
- result:

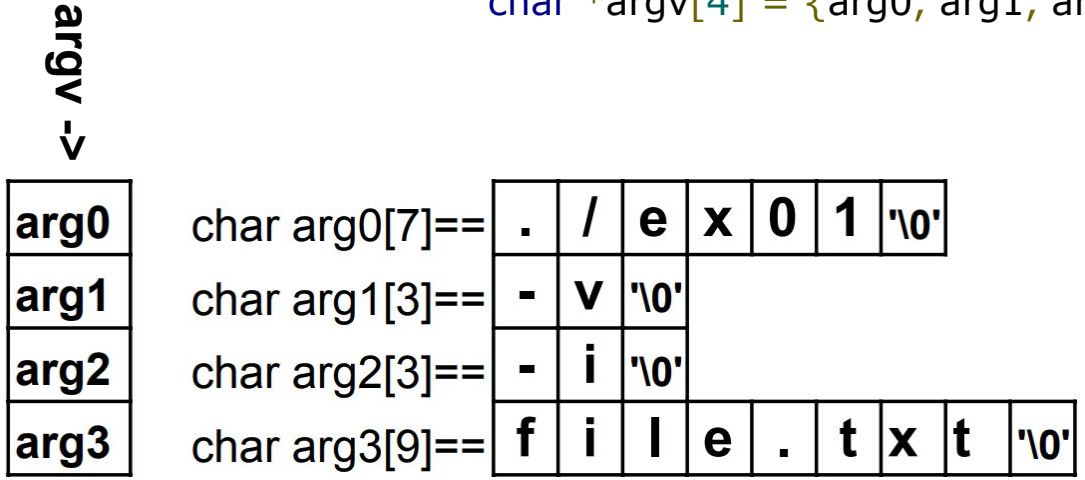
  4
  ex01
  -v
  -i
  file.txt

main() - arguments

```
./ex01 -v -i file.txt
int main(int argc, char **argv){
...
}
```

```
char arg0[7] = "./ex01";
char arg1[3] = "-v";
char arg2[3] = "-i";
char arg3[9] = "file.txt";

char *argv[4] = {arg0, arg1, arg2, arg3};
```



main() - type of function

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

// ./ex01 -v -i file.txt

//int main(int argc, char *argv[]){
int main(int argc, char **argv){

    // return 0;
    return EXIT_SUCCESS;
}
```

- The value returned by the main() function is the error status:
  - 0: without error
  - X: error code (e. -1, 1, 10)
  - EXIT_SUCCESS defined in cstdlib
- Allows transferring to other programs (eg the shell) information how the execution of this program has ended
- return 0; if we do not explicitly call

AGH

When writing in C/C++, you write in two languages simultaneously

ask the employee for a double payment 

C/C++ preprocessor

```
#include <iostream>
#include "ex01.h"

#define SIZE 10
#define EX01_H_

#ifdef SIZE
#undef SIZE
#endif

#if SIZE>10
#error COULD NOT PROCESS
#else
//...
#endif

#define ADD(a,b) a+b

#pragma
```

- Controlling the compilation process
- Including headers
- Allows to disable some code from compilation
- Writing programs for various hardware
  - part of the code only for ARM
  - part only for CPU with SSE
- Macros
- Instructions start with a sign: #
- Executed before the compilation stage
- The compiler receives the code without the preprocessor instructions (instructions will be processed and removed from the source code)

#include

```
#include <iostream>
#include "ex01.h"
```

- Inserts the source code in place of the call, e.g. file content: /usr/include/c++/4.8/iostream
- If <XXX> is looking for XXX in:
  - /Usr/include/
  - -L/usr/include/c++/
- If “YYY” searches for YYY in the current directory, where *.cc
- Only *.h files (declarations)
- Necessary when using libraries

#define

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

#define SIZE 10

int main(){
    cout << SIZE << endl;
    cout << "SIZE" << endl;
}
```

```
// ....
using namespace std;

int main(){
    cout << 10 << endl;
    cout << "SIZE" << endl;
}
```

- It allows to define:
  - constant
  - functions
  - keyword
  - macro
- The defined text will be replaced in the source code
- NOTE: no semicolon at the end of the line!!!
- Usually at the beginning of the source file
- CONVENTION

#define

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

#define SIZE 10

int main(){
    int tab[SIZE];
}
```

- Method of constant definition
  A common way to define the size of an array in languages:
  - C
  - C++ (<C ++ 98)
- Substitute for dynamic array size declaration
- Today, it is **not recommended** to use #define for the purposes of the constant declaration:
  - the type is unknown
  - the compiler can not optimize

#if

```
#define INTEL


#ifdef INTEL
// .... intel specific instruction
#else
// .... AMD specific instruction
#endif
```

- Conditional statements
- If the expression is defined, the code will be compiled
- Definition in your own code or provided by the compiler or OS:
  - _WIN32, _WIN64
  - __APPLE__, __MACH__
  - __linux__, linux
  - __FreeBSD__
  - unix, __unix

# #

#pragma

- Specific to the compiler, it allows you to control it (if the compiler does not understand it, it is ingrown)

#line 23

- Changing messages during compilation

#error

- Aborts the compilation process with an error

#macro

```
#define mmax(a,b) a>b?a:b
```

- In the source code, instead of max(x, y, the three-argument expression will be inserted
- more save macro:
  `#define mmax(a,b) ((a)>(b)?(a):(b))`

```
#define glue(a,b) a ## b
```

- ## means joining texts

```
glue(c,out) << "test";
```

will be translated into:

```
cout << "test";
```

How to manage a large code???

open your own library...

library

ex05_main.cc

```cpp
#include <iostream>
#include "ex05_factorial.h"
using namespace std;

int main(){
    for (int x = 0; x < 10; ++x) {
        cout << x << "! = ";
        cout << factorial(x) << endl;
    }
}
```

```
g++ ex05_main.cc -c
g++ ex05_factorial.cc -c
g++ ex05_main.o ex05_factorial.o -o ex05
./ex05

g++ ex05_main.cc ex05_factorial.cc -o ex05
```

ex05_factorial.h interface, declarations

```cpp
/**
 * calculate factorial
 * @param x argument
 * @return x!
 **/
double factorial(int x);

#define FACTORIAL_MAX 30
```

ex05_factorial.cc implementation, definition

```cpp
#include "ex05_factorial.h"

double factorial(int x){
    if (x==0) {
        return 1;
    } else if (x>FACTORIAL_MAX){
        return -1;
    }

    double result=1;
    for (int i = 2; i <= x; ++i) {
        result *= i;
    }
    return result;
}
```

library

ex05_main.cc

```
#include <iostream>
#include "ex05_factorial.h"
using namespace std;

int main(){
    for (int x = 0; x < 10; ++x) {
        cout << x << "! = ";
        cout << factorial(x) << endl;
    }
}
```

```
g++ ex05_main.cc -c
g++ ex05_factorial.cc -c
g++ ex05_main.o ex05_factorial.o -o ex05
./ex05

g++ ex05_main.cc ex05_factorial.cc -o ex05
```

- compilation, each *.cc -> *.o
- linking, all *.o -> executable
- multithreaded compilation possible
- *.h publicly available, event with (close-sourced) library
- *.o is a machine code (CPU instructions)
- linking only “copy” the code

type of library (linux)

- **object code (object file):**
  - g++ ex05_main.cc -c -> ex05_main.o
  - g++ ex05_factorial.cc -> ex05_factorial.o
  - *.o are files with machine code, created after compilation of the program for a specific CPU
  - The code from ex05_main.cc requires a code from ex05_factorial.cc, I can combine them with a running program: g++ ex05_main.o ex05_factorial.o -o ex05
- **Static library**:
  - an ordinary archive, containing several *.o files
  - ar rcs libctest.a test1.o test2.o
- Both types of files are included in the executable code, so after compilation, you can delete *.o i *.a
- Makes possible efficient compilation of large programs

type of library (linux)

- **Dynamically loaded libraries**
  - loaded after starting the program
  - they are not included in the program code during compilation
  - they are shared by many programs
  - libraries (files) are searched in locations:
    - indicated by the variable: LD_LIBRARY_PATH
    - in paths saved in the /etc/ld.so.conf file
    - w /lib/ oraz /usr/lib/
- **soname** == shared object name
  - adding a library to the program (the program will use it):
    g++ -lname
  - will load the library after starting the program from the location:
    /usr/lib/libname.so.1
- **the mechanism for handling shared libraries depends on the OS**

Do variables have other characteristics than type?

YES, they have "modifiers"

Variable modifiers

- Variables have a type, e.g.
  - int, float, size_t, struct Color (my own type of variable)
- Variables can also have so-called "Modifier":
  - const
  - static
  - auto
  - extern
  - register
  - volatile

const variable

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

// #define stala 10
const int stala = 10;

int main(){
    cout << stala << endl;
    // cout << stala++ << endl;
}
```

- constant variable
- You have to initialize during the declaration, if not then:
  error: uninitialized const ‘stala’
- An attempt to change a constant ends with a message:
  error: increment of read-only variable ‘stala’
- The advantage of const over the constant created as #define, is a TYPE. Compiler can optimize and check if use is correct

const variable

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

int sumOfTable(int const *tab, size_t size) {
    int sum = 0;
    for (size_t i = 0; i < size; ++i) {
        sum += tab[i];
    }

    return sum;
}

int main(){
    size_t size = 100;
    int tab[size];
    cout << sumOfTable(tab, size);
}
```

- Passing to the array function by the pointer.
- It prevents modification of the array content!!!
- An attempt to modify in the function:
  error: assignment of read-only location '* tab'
- A good way to reduce unwanted situations
- What semantic error did I make in the code?

const variable

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

int sumOfTable(int const *tab, size_t size) {
    int sum = 0;
    for (size_t i = 0; i < size; ++i) {
     //sum += tab[i];
        sum += *tab++;
    }

    return sum;
}

int main(){
    size_t size = 100;
    int tab[size];
    cout << sumOfTable(tab, size);
}
```

- The content of the tab table is protected from change
- The pointer can be changed!!!
- I could write:

int const * const tab

then both the pointer and the indicated content would be constant

const variable

- Some examples of using **const**
  int* - pointer to int
  int const * - pointer to const int
  int * const - const pointer to int
  int const * const - const pointer to const int
- It's worth reading:
  - https://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const
  - https://stackoverflow.com/questions/10091825/constant-pointer-vs-pointer-on-a-constant-value
- "**const**" is ambiguous, has many uses, you have to be careful when using it!

static variable

```
int sumStatic(int arg) {
    static int result = 0;
    result += arg;
    return result;
}

int sum(int arg) {
    int result = 0;
    result += arg;
    return result;
}

int main(){
    for (int i = 0; i < 10; ++i) {
        cout << sumStatic(i);
        cout << " " << sum(i) << endl;
    }
}
```

- "Staticity" maintaining the value (state) between consecutive declaration of the same variable
- Result:

```
0 0
1 1
3 2
6 3
10 4
15 5
21 6
28 7
```

static variable

```
int sumStatic(int arg) {
    static int result = 0;
    result += arg;
    return result;
}

int sum(int arg) {
    int result = 0;
    result += arg;
    return result;
}

int main(){
    for (int i = 0; i < 10; ++i) {
        cout << sumStatic(i);
        cout << " " << sum(i) << endl;
    }
}
```

```
0 0
1 1
3 2
6 3
10 4
15 5
21 6
28 7
```

- A way to preserve the "state" of the process
- Allows not to complicate the code with global variables or passing the state through an argument
- Meaning, depends on the context of use, e.g.
  - global implication
  - inside the function
  - in class
  - differences between C and C +

auto variable

```
int main(){
   auto int x = 0;
   cout << x << endl;
}
```

- Anachronism, "inherit" from programming languages on which C is based
- auto means that the variable is local, but is local so…
- auto means automatic creation and deletion if it goes out of range (scope)
- C++11: changed the meaning, it means "substitute for this variable" - the type will be matched at the time of initialization

extern variable

plik: ex14_extern.cc

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

int x = 7;
```

plik: ex14_main.cc

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

extern int x;

int main(){
    cout << x << endl;
}
```

- Allows you to use a global variable from another file
- It does not declare a variable, it only informs the compiler that such a variable will be declared and will appear during linking
- Related to libraries

- Compilation of example:
  g++ ex14_extern.cc ex14_main.cc

register variable

- If possible, place the variable in the register instead of on the stack (it will be faster to access it)
- There is no guarantee
- The modern compiler "knows better"
- In practice: anarchism

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

int main(){
    for (register int x = 0; x < 10; ++x) {
        cout << "super fast iterator ;-) ";
        cout << x << endl;
    }
}
```

volatile variable

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

volatile int x = 0;

int main(){
   cout << x << endl;
}
```

- Disabling optimization
  - will not be replaced by a constant
  - will not be placed in the register
  - will always be read from memory (slowly !!!)
- Specific application
  - concurrency
  - hardware drivers
- Prevents errors if the variable is changed without the knowledge of the "compiler", eg. by another program

Oldschool

printf()

printf()

```
#include <cstdio>

int main(){
    int x = 7;
    float f = 1.23;

    printf("%d, %f\n", x, f);
    printf("f=%.3f\n", f);
    printf("\nempty line ?!?\n");
}
```

- Standard library C
- Can be used in C++ but not recommended mixing with “cout”
- "Formatting text"
  - contains the displayed text
  - contains the place and type of displayed variable
  - "%d" means display int
  - "%.3f" means display the float with an accuracy of 3 decimal places
- Variable number of arguments!

printf()

Przykłady: http://www.cplusplus.com/reference/cstdio/printf/

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

Thank you