

# Podstawy Informatyki

**Katedra Telekomunikacji, EiT**

dr inż. Jarosław Bułat

[kwant@agh.edu.pl](mailto:kwant@agh.edu.pl)

# Plan prezentacji

- » Funkcja main()
- » Instrukcje preprocesora
- » Modyfikatory zmiennych
- » Biblioteki
- » Funkcja printf()

# main()

argumenty funkcji

# main() - argumenty

```
#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;
}
```

- » Argumenty programu:
- » argc: liczba argumentów
- » argv: tablica zawierające wskaźniki do tablic z pojedynczymi argumentami programu (wskaźnik na wskaźnik)
- » result:  
4  
ex01  
-v  
-i  
file.txt

# main() - argumenty

```

./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};
```

argv ->

<b>arg0</b>	char arg0[7]==	.	/	e	x	0	1	'\0'		
<b>arg1</b>	char arg1[3]==	-	v	'\0'						
<b>arg2</b>	char arg2[3]==	-	i	'\0'						
<b>arg3</b>	char arg3[9]==	f	i	l	e	.	t	x	t	'\0'

# main() - wartość funkcji

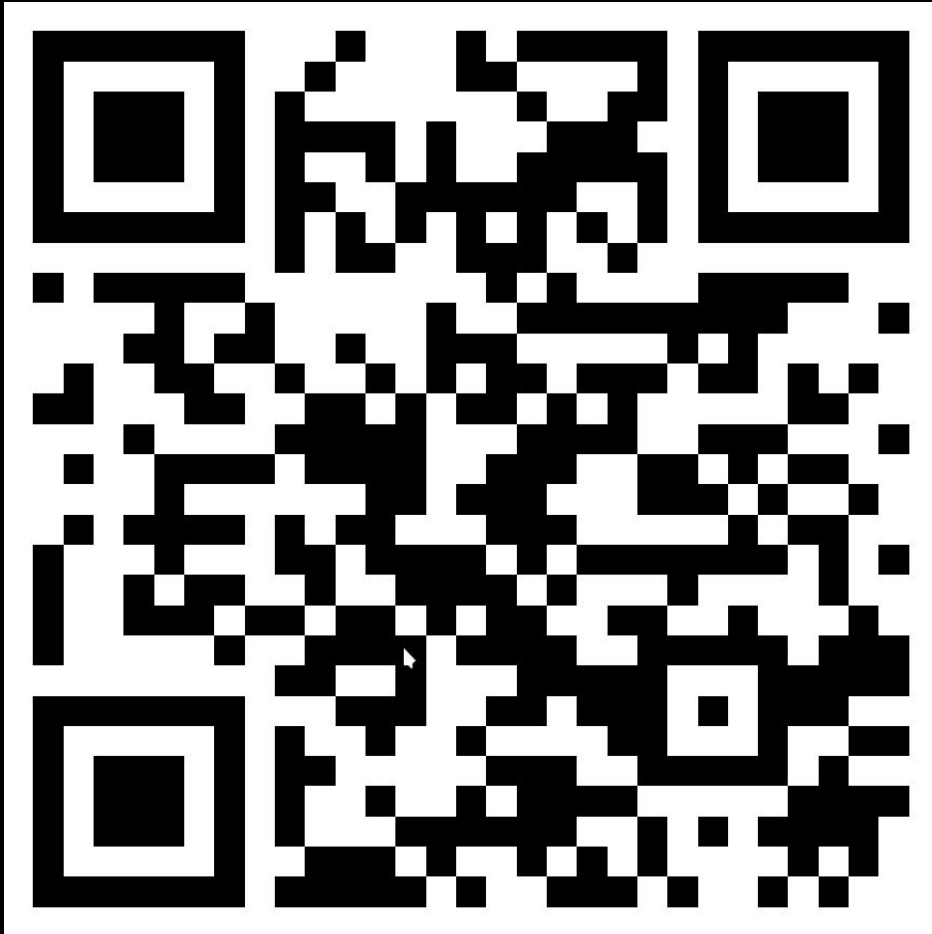
```
#include<iostream>
#include<cstdlib>
using namespace std;

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

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

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

- » Wartość zwracana przez funkcję `main()` to status błędu:
  - 0: bez błędu
  - X: kod błędu (np -1, 1, 10)
  - EXIT\_SUCCESS  
zdefiniowane w `cstdlib`
- » Umożliwia przekazanie innym programom (np. powłoce) informacji jak zakończyło się wykonywanie tego programu
- » `return 0;` jeżeli jawnie nie wywołamy



quiz

**PI08\_main**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

Pisząc w C/C++ piszesz w dwóch  
językach równocześnie

od zleceniodawcy żądaj podwójnej wypłaty 😄



# 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
```

- » Sterowanie procesem kompilacji
- » Dołączanie nagłówków
- » Pozwala wyłączyć część kodu z kompilacji
- » Pisanie programów dla różnego sprzętu
  - część kodu tylko dla ARM
  - część tylko dla CPU z SSE
- » Makrodefinicje
- » Instrukcje rozpoczynają się znakiem #
- » Wykonywane przed etapem kompilacji
- » Kompilator otrzymuje kod bez instrukcji preprocesora (instrukcje zostaną wykonane i usunięte z kodu źródłowego)

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

# #include

- » Wstawia kod źródłowy w miejsce wywołania, np. zawartość pliku:  
`/usr/include/c++/4.8/iostream`
- » Jeżeli `<XXX>` szuka XXX w:
  - `/usr/include/`
  - `-L/usr/include/c++/`
- » Jeżeli `"YYY"` szuka YYY w bieżącym katalogu, gdzie `*.cc`
- » Tylko pliki `*.h` (deklaracje)
- » Element niezbędny przy używaniu bibliotek

# #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;
}
```

- » Pozwala zdefiniować:
  - stałą
  - funkcję
  - słowo kluczowe
  - makro
- » Zdefiniowany tekst zostanie podmieniony w kodzie źródłowym
- » UWAGA: brak średnika na końcu linii !!!
- » Zazwyczaj na początku pliku źródłowego
- » KONWENCJA

# #define

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

```
#define SIZE 10
```

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

- » Sposób definicji stałej
- » Często sposób na definicję rozmiaru tablicy w językach:
  - C
  - C++ (<C++98)
- » Substytut dynamicznej deklaracji rozmiaru tablicy
- » Współcześnie nie zaleca się stosowania #define do celów deklaracji stałych:
  - nieznan jest typ
  - kompilator nie może optymalizować

# #if

```
#define INTEL
```

```
#ifdef INTEL
```

```
// ... intel specific instruction
```

```
#else
```

```
// ... AMD specific instruction
```

```
#endif
```

- » Instrukcje warunkowe
- » Jeżeli zdefiniowano wyrażenie kod będzie kompilowany
- » Definicja we własnym kodzie albo dostarczana przez kompilator lub OS:
  - \_WIN32, \_WIN64
  - \_\_APPLE\_\_, \_\_MACH\_\_
  - \_\_linux\_\_, linux
  - \_\_FreeBSD\_\_
  - unix, \_\_unix

#pragma

#line 23

#error

# #

- » Specyficzne dla kompilatora, pozwala nim sterować (jeżeli kompilator nie rozumie, są ignorowane)
- » Zmiana komunikatów podczas kompilacji
- » Przerywa proces kompilacji z błędem

# #makrodefinicje

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

- » W kodzie źródłowym, zamiast max(x,y) zostanie wstawione wyrażenie trójargumentowe
- » Bardziej bezpieczne makro:  
#define mmax(a,b) ((a)>(b)?(a):(b))

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

- » ## oznacza łączenie tekstów

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

zostanie przetłumaczone na:

cout << "test";



quiz

**PI08\_prepr**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**



# Jak zarządzać dużym kodem???

otwórz własną bibliotekę...

# biblioteka

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
```

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

#define FACTORIAL_MAX 30
```

**ex05\_factorial.h**  
interfejs, deklaracje

```
#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;
}
```

**ex05\_factorial.cc**  
implementacja, definicje

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
```

- » kompilacja, każdy \*.cc -> \*.o
- » linkowanie, wszystkie \*.o -> plik\_wykonywalny
- » Możliwa kompilacja wielowątkowa
- » \*.h jest publicznym plikiem, nawet dla komercyjnych bibliotek
- » \*.o jest plikiem z kodem maszynowym (instrukcje procesora)
- » [linkowanie tylko łączy kod](#)

# rodzaj bibliotek (linux)

## » object code (object file):

- `g++ ex05_main.cc -c` -> `ex05_main.o`
- `g++ ex05_factorial.cc` -> `ex05_factorial.o`
- `*.o` to pliki z kodem maszynowym, powstałym po kompilacji programu na konkretne CPU
- Kod z `ex05_main.cc` wymaga kodu z `ex05_factorial.cc`, mogą je połączyć uzyskując działający program:  
`g++ ex05_main.o ex05_factorial.o -o ex05`

## » Biblioteka statyczna:

- zwyczajne archiwum, zawierające kilka plików `*.o`
- `ar rcs libctest.a test1.o test2.o`

» Oba typy plików są “włączane” w kod wykonywalny czyli po kompilacji można skasować `*.o` i `*.a`

» Ułatwienie kompilacji dużych programów

# rodzaj bibliotek (linux)

## » Dynamicznie ładowane biblioteki

- ładowane po uruchomieniu programu
- nie są włączone w kod programu podczas kompilacji
- są współdzielone przez wiele programów
- biblioteki (pliki) są szukane w lokalizacjach:
  - wskazywanych przez zmienną: LD\_LIBRARY\_PATH
  - w ścieżkach zapisanych w pliku /etc/ld.so.conf
  - w /lib/ oraz /usr/lib/

## » **soname** == shared object name

- dodanie biblioteki do programu (program będzie z niej korzystał):  
`g++ -lname`
- spowoduje załadowanie biblioteki po starcie programu z lokalizacji:  
`/usr/lib/libname.so.1`

## » mechanizm obsługi bibliotek dzielonych zależy od OSu



quiz

**PI08\_lib**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

# Czy zmienne mają inne cechy oprócz typu?

TAK, mają “modyfikatory”

# Modyfikatory zmiennych

- » Zmienne mają typ, np.:
  - int, float, size\_t, struct Color (mój rodzaj zmiennych)
- » Zmienne mogą mieć też tzw. “modyfikator”:
  - const
  - static
  - auto
  - extern
  - register
  - volatile



# zmienna const

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

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

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

- » **const** to Stała
- » Trzeba **zainicjalizować** podczas deklaracji, jeżeli nie to:  
**error: uninitialized const 'stala'**
- » Próba zmiany “stałej” kończy się komunikatem:  
**error: increment of read-only variable 'stala'**
- » Zaletą stałej “**const**” nad stałą deklarowaną jako **#define**, jest TYP. Ma typ, **kompilator może optymalizować i sprawdzić czy zgadza się przypisanie.**

# zmienna const

```
#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);
}
```

- » Przekazanie do funkcji tablicy przez wskaźnik.
- » Zapobiega modyfikacji zawartości tablicy !!!
- » Próba modyfikacji w funkcji:  
error: assignment of read-only location '\* tab'
- » Dobry sposób na ograniczenie niepożądanych sytuacji
- » Jaki błąd semantyczny popełniłem w kodzie?

# zmienna const

```
#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);
}
```

- » Zawartość tablicy tab jest chroniona przed zmianą
- » Wskaźnik można zmieniać !!!
- » Mógłbym napisać:

`int const * const tab`

wtedy stały byłyby zarówno wskaźnik jak i wskazywana zawartość

# zmienna **const**

» Kilka przykładów użycia **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

» Warto przeczytać:

- <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>

» Modyfikator **const** ma różnych zastosowań

# zmienna static

```
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;  
    }  
}
```

- » „Statyczność” polega na zachowaniu wartości (stanu) pomiędzy kolejnymi definicjami tej samej zmiennej

» Rezultat:

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

# zmienna static

```
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

- » Sposób na zachowanie “stanu” procesu
- » Nie trzeba komplikować kodu zmiennymi globalnymi albo przekazywaniem stanu przez argument
- » Ma wiele znaczeń, zależy od kontekstu użycia, np:
  - globanie
  - wewnątrz funkcji
  - w klasie
  - różnice pomiędzy C i C++

# zmienna **auto**

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

- » **Anachronizm**, “spadek” po językach programowania na których C jest oparty.
- » **auto** oznacza, że zmienna jest lokalna, no ale jest lokalna więc OCB?
- » auto oznacza, automatyczne tworzenie i kasowanie jeżeli wychodzi z zasięgu (scope)
- » **C++11: zmieniło znaczenie, oznacza “zastępczy typ zmiennej” - typ zostanie dopasowany w chwili inicjalizacji**

# zmienna extern

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;
}
```

- » Umożliwia użycie zmiennej globalnej z innego pliku.
- » Nie deklaruje zmiennej tylko informuje kompilator, że taka zmienna będzie zadeklarowana i pojawi się podczas linkowania
- » Związane z bibliotekami
- » Kompilacja przykładu:  
`g++ ex14_extern.cc ex14_main.cc`



# zmienna register

- » Jeżeli możliwe umieść zmienną w rejestrze a nie na stosie (będzie do niej szybszy dostęp)
- » Nie ma żadnej gwarancji
- » Współczesny kompilator “wie lepiej”
- » W praktyce: anarchizm

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

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

# zmienna **volatile**

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

```
volatile int x = 0;
```

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

- » volatile == ulotny
- » Wyłączenie optymalizacji
  - nie zostanie zastąpiona przez stałą
  - nie zostanie umieszczona w rejestrze
  - zawsze będzie odczytywana z pamięci (wolno!!!)
- » Wąskie zastosowanie
  - współbieżność
  - sterowniki do sprzętu
- » Zapobiega błędom jeżeli zmienna zostanie zmieniona bez wiedzy “kompilatora”, np. przez inny program



quiz

**PI08\_mod**

**socrative.com**

- login
- student login

Room name:

**KWANTAGH**

# Oldschool

`printf()`

# printf()

```
#include <stdio>
```

```
int main(){  
    int x = 7;  
    float f = 1.23;  
  
    printf("%d, %f\n", x, f);  
    printf("f=%.3f\n", f);  
    printf("\npusta linia !?\n");  
}
```

- » Biblioteka standardowa C
- » Można użyć w C++ ale nie zalecane mieszanie z cout
- » **“tekst formatujący”**
  - zawiera wyświetlany tekst
  - zawiera miejsce i typ wyświetlanej zmiennej
  - “%d” oznacza wyświetl int
  - “%.3f” oznacza wyświetl float z dokładnością 3 miejsca po przecinku
- » Zmienna liczba argumentów!

# printf()

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

<b><i>specifier</i></b>	<b>Output</b>	<b>Example</b>
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.	%

Dziękuję