Język C

Kierunek: Informatyka Stosowana (*Wydział FAIS UJ*)

Wykład: Wtorek, 12.15 s. A-1-06

Forma zaliczenia kursu: Egzamin pisemny (test wyboru) * **

* Warunkiem przystąpienia do egzaminu jest wcześniejsze ***zaliczenie ćwiczeń*** (lub w uzasadnionych przypadkach: *zgoda prowadzącego ćwiczenia*)

[⇒ OCENA KOŃCOWA: *0.5*ocena z ćwiczeń + 0.5*wynik egzaminu* **]**

Cel dydaktyczny:

Poznanie elementów składniowych standardowego języka *ANSI C*; nauka podstaw programowania strukturalnego w tym języku, z naciskiem na *czytelny styl programowania*

Wymagania wstępne:

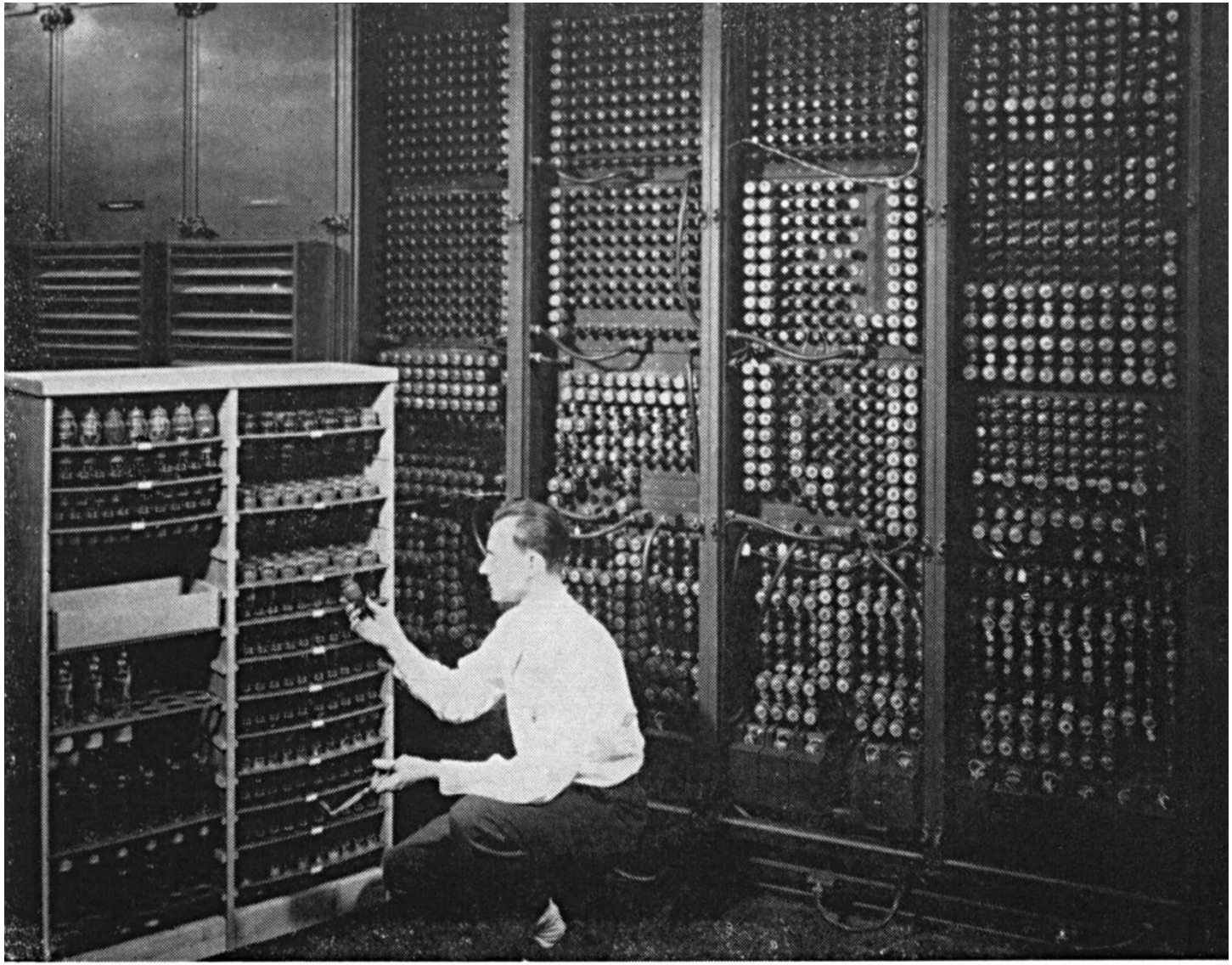
Matematyka/logika na poziomie szkoły średniej; zamiar sprawnego przyswajania wiedzy matematycznej (algebra/analiza), która będzie konieczna do zrozumienia części *przykładów* …

Zagadnienia szczegółowe: Typy danych, deklaracje, wyrażenia, instrukcje, funkcje, dyrektywy preprocesora, operacje bitowe, wskaźniki, pliki, wejście/wyjście, C w systemie UNIX/Linux, …

Literatura (+źródła):

1. Brian W. Kernighan, Dennis M. Ritchie, *Język ANSI C*, WNT Warszawa 2000. **<== PODSTAWA KURSU!**
2. Stephen Prata, *Język C. Szkoła programowania.* Wydanie VI. Wydawnictwo Helion, Gliwice, 2016.
3. Dawn Griffiths, David Griffiths, *C. Rusz głową!*, Wydawnictwo Helion, Gliwice, 2013.
4. K. N. King, *Język C. Nowoczesne programowanie*. Wydanie II Wydawnictwo Helion, 2011
5. Wykłady **online** + *materiały dla Grup 2. i 5.:* https://th.if.uj.edu.pl/~adamr/zadania/zadania.html
6. ORAZ *inne źródła cytowane w kolejnych wykładach …*

Pierwszy komputer ...



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Pierwszy komputer …

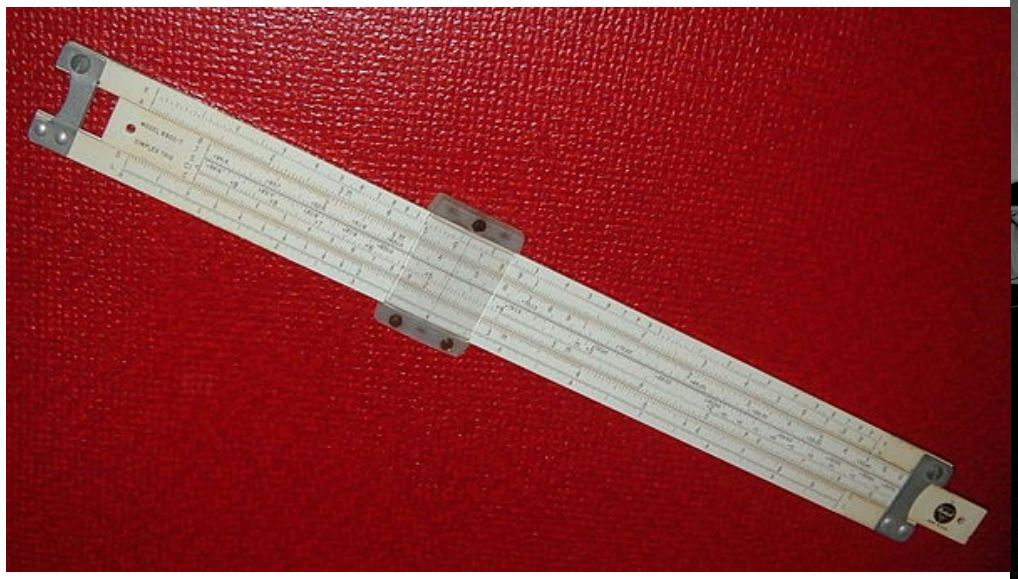
ENIAC (ang. *Electronic Numerical Integrator And Computer* – *Elektroniczny*, Numeryczny Integrator i Komputer) – skonstruowany w latach 1943–1945 przez J.P. Eckerta i J.W. Mauchly'ego z Uniw. Pensylwanii, USA. Używany do 1955. [Źródło: Wikipedia]

=> **Dla dociekliwych**: A co to znaczy "*Integrator*"?

… Konkurenci do tytułu:

Brytyjski *Colossus*, niemieckie maszyny Konrada Zusego; komputer ABC (ang. *Atanasoff-Berry Computer*), zbudowany w Iowa State University przez Johna Vincenta Atanasoffa i Clifforda Berry'ego w latach 1937–1942. [Źródło: Wikipedia]

A jak liczono wcześniej?





W ramach *Projektu Manhattan* (1942–46) zespół ok. 1000 rachmistrzów (“*a human computer*”) wyposażonych w kartki, ołówki i >>**suwaki logarytmiczne**<< prowadzi symulacje numeryczne reakcji łańcuchowej (=> *algorytm Metropolisa i Ulama*).

Komputer to maszyna cyfrowa ...

Pozycyjne systemy liczenia:

$$c_3c_2c_1c_0,c_{-1}c_{-2}\ldots = c_3p^3 + c_2p^2 + c_1p^1 + c_0p^0 + c_{-1}p^{-1} + c_{-2}p^{-2} + \ldots$$

gdzie: p - *podstawa,* $c_i = 0, 1, \ldots, p-1$ — *cyfry*

Przykłady:

p=10 - system dziesiątkowy, cyfry: 0,1,…,9

p=2 - s. dwójkowy (*binarny*), cyfry: 0,1

p=8 - s. ósemkowy (*oktalny*), cyfry: 0,1,…,7

p=16 - s.szesnastkowy (*heksadecymalny*), cyfry: 0,1,…,9,A,B,C,D,E,F

Przykładowe deklaracje w języku C:

```
const int a = 1977; /* Stała całkowita, dziesiątkowo */
const int a = 03671;  /* To samo,  ó s e m k o w o  */
const int a = 0x7B9;  /* To samo, szesnastkowo  */
```

Czy można liczyć inaczej? [*Jeśli na dodatek nie lubimy suwaków…*]

Rozkład na *czynniki pierwsze*: $1977 = 3 * 659 = 3^1 * 659^1$

Zapisujemy *w y k ł a d n i k i* w rozkładzie na czynniki pierwsze (także jeśli są zerami, aż do największego czynnika):

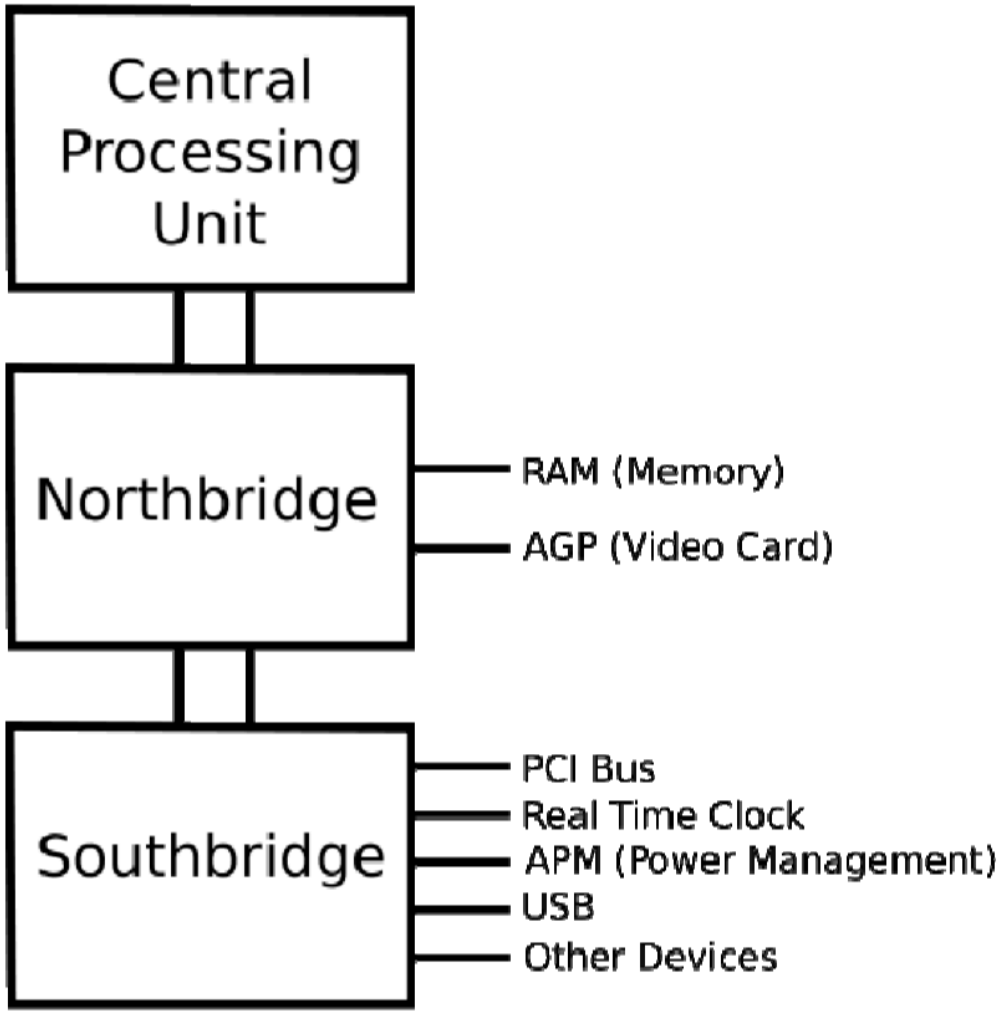
$$0_2\ 1_3\ 0_5\ 0_7\ 0_{11} \ldots 0_{647}\ 0_{653}\ 1_{659} = 1977$$

$$0_2\ 0_3\ 0_5\ 1_7\ 1_{11} = 77$$

Mnożenie w tym systemie wykonujemy *d o d a j ą c* wykładniki:

$$0_2\ 1_3\ 0_5\ 1_7\ 1_{11} \ldots 0_{647}\ 0_{653}\ 1_{659} = 1977 \times 77 = 152\ 229$$

Architektura komputera



Architektura komputera

Mostek północny (ang. *north bridge*):

wymiana danych między pamięcią a procesorem, np. sterowanie magistralą AGP i PCI

Mostek południowy (ang. *south bridge*):

współpraca z urządzeniami we/wy, takimi jak dysk, inna pamięć masowa, karty rozszerzeń (PCI), port szeregowy, port równoległy, złącze klawiatury, złącze myszy, modem, interfejs FDD, wyjście audio itp.

=> A po co j e s z c z e kompilator ???

Przykładowa konwersacja z komputerem w asemblerze:

(Nieco wcześniej programowano bezpośrednio w jęz. wewnętrznym …)

```
mov al,7    <= do rejestru “al” wstaw liczbę ‘7’
add al,12   <= dodaj ’12’
mov [500],al  <=  pod adres “500” wstaw
                  zawartość “al”
```

=> *Rachmistrzowie to mieli życie …*

Kompilator to specjalny program, który tłumaczy nasz *kod źródłowy* (napisany np. w języku C, C++, fortran, python, julia, …) na kod zrozumiały dla maszyny.

=> A jak skompilowano pierwszy kompilator?

Po co dzisiaj uczyć się języka C?

<u>Współczesne maszyny cyfrowe</u>:



, ... ,



Programy w C można pisać na Earth Simulator-a i na iPhone-a …

Cechy języka C:

- J. C daje niemal pełną kontrolę nad działaniami maszyny (np. *przydział* i *zwalnianie* pamięci — tzw. dynamiczna alokacja, adresy, wskaźniki, *jednowymiarowe* tablice)
- W uproszczeniu: C to ***współczesny asembler*** (działa na tzw. *niskim poziomie*)
- Z powyższego wynikają DWIE ważne konsekwencje:

  a) **Dobry programista** C zarazem dobrze zna (*i rozumie*) działanie maszyny i systemu operacyjnego, wie co jest *wykonalne* a co nie
  [=> dodatkowo: C to tzw. "*mały język*", w którym stosunkowo łatwo zostać dobrym programistą :-)]

b) J. C stwarza możliwość tworzenia małych i szybkich (*wydajnych*) aplikacji bez “bagażu” często produkowanego w innych językach.

[=> Aplikacje na smartfony i urządzenie mobilne]

[=> Także obliczenia *wielkiej skali*, w przypadku których “mały i szybki” może po prostu oznaczać “jedyny wykonywalny” na istniejących maszynach]



, ... ,



=> Język C ma DWA końce …

5 zasad szczęśliwego programisty:

1) Komputer to (*tylko i aż …*) **maszyna** [*chociaż cyfrowa …*]
2) Komputer jest **sługą** człowieka [*niech się komp. męczy, nie ja!*]
3) Komputer się **nigdy nie myli** *
4) Komputer wykonuje **tylko i dokładnie** to, co człowiek mu poleci **
5) Komputer jest **zawsze szybszy** od człowieka

*) Jednak liczy ze *skończoną* dokładnością (*zob. reprezentacja liczb zmiennopozyzyjnych w maszynie*); niekiedy wynik obliczeń na pierwszy rzut oka sprawia wrażenie, jakby “komputer się pomylił”.

**) To *człowiek* nie zawsze jest świadomy, co dokładnie polecił …

Trochę więcej o języku C

- C (*ANSI C*) jest językiem *ogólnego zastosowania;* początkowo blisko związany z systemem UNIX (unix-y i wiele programów działających pod nimi były pisane w C) i nazywany *"językiem programowania systemowego"*.
- Język C nie jest jednak przywiązany do konkretnego systemu operacyjnego; nadaje się do pisania programów z wielu dziedzin.
- Wiele idei C pochodzi z języków BCPL i B (powstał nawet spór, czy następca C ma się nazywać "D" czy też "P"…)
- Języki BCPL i B to tzw. *języki beztypowe*, np deklaracje:

```
let V = vec 10    /* w BCPL */
auto V[10];       /* w B */
```

alokują w komórce V adres *pierwszej z ciągu* kolejnych 10 komórek.

Więcej o początkach języka C:

https://www.bell-labs.com/usr/dmr/www/chist.html

[*Tam też dociekliwi znajdą m.in. odpowiedź na pytanie:*
==> *Jak skompilowano pierwszy kompilator ? …*]

W języku C - mamy różnorodne *typy danych*:

Typy podstawowe - znaki (*char*), liczby całkowite (*int* | *long*), l.zmiennopozycyjne (*float* | *double* | *long double*); typ wyliczeniowy (*enum*).

Typy pochodne - tworzymy za pomocą wskaźników, tablic, struktur i unii. [*napis=tablica znaków:* char s[20]="Plus ratio quam vis";]

Wyrażenie: operator + argumenty; [np. "a = a+2"]

Każde wyrażenie w C (także np. *przypisanie* lub *wywołanie funkcji*) może pełnić rolę instrukcji. ["a=2" zwraca wartość "2" !]

Wskaźniki w C pozwalają wykonywać obliczenie na adresach w sposób niezależny od maszyny. [wsk=&V[0]; *(wsk+3) == V[3]]

Konstrukcje sterujące w C (*programowanie strukturalne*): grupowanie instrukcji ({ … }), instr. warunkowa (if – else if – else), instr. wyboru (switch), pętle (while, for, do–while), przerwanie pętli (break), powrót do początku pętli (continue)

Funkcje w C mogą zwracać wartości typów podstawowych, struktury, unie i *wskaźniki* (*tablice — NIE!*).
Funkcje można wywoływać *rekurencyjnie*, ale nie można ich *zagnieżdżać* (f-cja może jednak zawierać dowolnie zagnieżdżone bloki instrukcji)
Zmienne lokalne f-cji: tworzone na nowo przy każdym wywołaniu.

C jest językiem “niskiego poziomu”:

Operuje na obiektach (tj. *znakach, liczbach, adresach*) zbliżonych do tych, które *realnie* istnieją w pamięci maszyny
[np. tablica = *ciągły, jednowymiarowy blok komórek pamięci*]

W języku C **nie ma operacji na obiektach złożonych**
(jak ciągi znaków, listy, tablice); jedynie *struktury* mogą być kopiowane w całości.

Przydział pamięci możliwy jest wyłącznie poprzez definicje statyczne i stos [=> zmienne lokalne funkcji]; nie ma “sterty” (ang. *heap*) ani “odśmiecania” (ang. *garbage collection*)

Nie ma także narzędzi wejścia-wyjścia
(np. instrukcji READ | WRITE)

=> **Czy zatem w ogóle w C można napisać sensowny program** ?!

Standard ANSI C określa liczne funkcje bibliotek
(stdlib.h, stdio.h, string.h, math.h, …), które uzupełniają wiele spośród w/w braków. KAŻDY program w C używa bibliotek.

=> **Dla porównania 2 napisów muszę wywoływać jakieś funkcje?!**
[TAK, chyba że wolimy porównywać znak po znaku …]

***Minimalizm* standardu C** daje jednak wymierne korzyści:
"mały język" stosunkowo łatwo przyswoić w całości; programista szybko jest w stanie używać *pełnej mocy* języka.
(Łatwo też stworzyć praktycznie *doskonały* kompilator …)

W języku C przyjmuje się, że **programista wie co robi.**
Np. warunek "if (a=7) {…}" jest FORMALNIE POPRAWNY i …
zawsze prawdziwy (!) [Zapewne poeta miał na myśli "if (a==7)"…]

[*Wskazówka praktyczna*: Lepiej pisać "if (stała == zmienna)" wtedy zamiana "==" na "=" spowoduje błąd na etapie kompilacji.]

Prosty program w C (*w całości!*)

```
#include <stdio.h>   /* Plik nagłówkowy, zawiera */
/* np. deklaracje funkcji “printf” i “scanf” */

main()
{
  printf(“Hej! Jestem Twoim programem!\n”);
}
```

Kompilujemy (UNIX/Linux):

```
$ gcc mojprogram.c -o mojprogram  <enter>
```

Inny sposób otrzymanie tego samego wyniku:

```
printf("Hej! "); printf("Jestem "); printf("Twoim ");
printf("programem!");
printf("\n");   /* "\n" TO ZNAK KOŃCA WIERSZA */
```

Aby mieć pewność, że program został napisany **w czystym C:**

```
$ gcc -ansi -pedantic mojprogram.c
```

Uwaga: Opja -ansi kompilatora gcc jest równoważna -std=c89.
Inne spotykane standardy (*także "ANSI"!*) języka C to *C99, C11,* oraz *C17/C18.* [*Standard C23 – planowany wkrótce…*].

Na tym wykładzie: ANSI == C89 (!!!)

Kompilatory C pracujące w systemie WINDOWS:

http://www.bloodshed.net

http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2_setup.exe

[*kompilator Dev-C++*]

http://sourceforge.net/projects/orwelldevcpp/

Kompilator Orwell Dev-C++ (wersje 5.4.2, 5.5.0)

Mac OS: Instalujemy pakiet *Xcode*, oraz *command-line tools*:

```
$ sudo xcode-select –install
```

Inne — *też darmowe*! — kompilatory języka C *działające m.in. po systemem WINDOWS:*

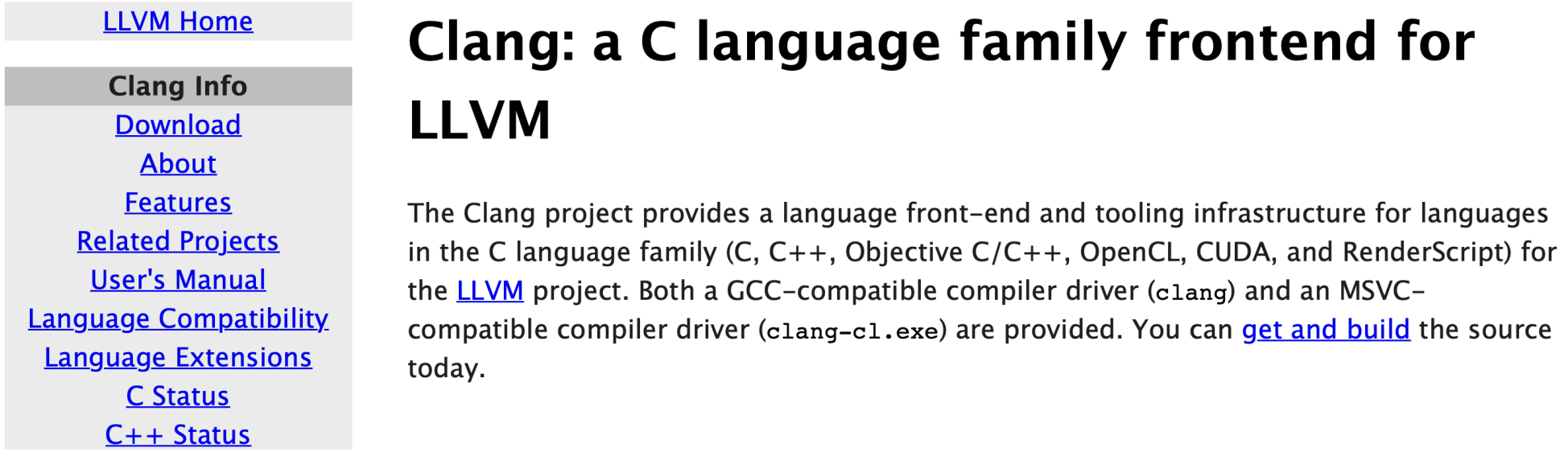
IDE (ang. *integrated development environment*) **działające online** [*nie trzeba niczego instalować!*]:

https://www.onlinegdb.com

Wielosystemowe IDE dla C/C++: **„CLion”** (*dostępne za darmo dla posiadaczy konta email w domenie* ***uj.edu.pl***):

https://www.jetbrains.com/student/

Kompilatory z rodziny Clang:



LLVM Home

Clang Info
Download
About
Features
Related Projects
User's Manual
Language Compatibility
Language Extensions
C Status
C++ Status

Clang: a C language family frontend for LLVM

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project. Both a GCC-compatible compiler driver (`clang`) and an MSVC-compatible compiler driver (`clang-cl.exe`) are provided. You can get and build the source today.

Patrz: https://clang.llvm.org

=> *Wiele cech wspólnych z gcc; bardziej ortodoksyjne „ansi c”, za to rozbudowany interfejs z podpowiedziami dot. możliwych błędów.*

```
test.c:8:8: warning: using the result of an assignment as a condition without parenthe
ses [-Wparentheses]
  if (a=7) {
      ~^~
test.c:8:8: note: place parentheses around the assignment to silence this warning
  if (a=7) {
      ^
     (  )
test.c:8:8: note: use '==' to turn this assignment into an equality comparison
  if (a=7) {
      ^
      ==
1 warning generated.
```

==> ***Doskonały wybór do nauki porządnego programowania w C!***

Co to znaczy “dobry styl programowania”?

- Określamy nasz priorytet:
  Optymalizacja pamięci, czasu wykonania, czasu opracowania programu, kosztów konserwacji programu, …
- **Prostota, umiar, elegancja** …
  Warunki: `if(i)` `if(i!=0)` `if((i!=0)!=0)` znaczą to samo *dla kompilatora*, ale już niekoniecznie *dla człowieka*
  (*np. autora programu we własnej osobie, po upływie 1 roku …*)
- Program w trakcie pisania podlega nieustannym zmianom
  (*a “pisanie programu” nigdy się nie kończy …*)
- **Nie nadużywamy** *zmiennych globalnych* i *instrukcji skoku*
  *(żeby kózka nie skakała, to by … **program napisała**!)*

Dobry styl programowania (cz.2)

- Używamy nawiasów w złożonych wyrażeniach
  (także kiedy "nie trzeba"!)
- `/* Zamieszczamy LICZNE komentarze */`
- Nie stosujemy *ekstrawaganckich* warunków [np. if((i!=0)!=0)]
- Korzystamy z funkcji bibliotecznych
  [*użycie:* `!strcmp(s,"TAK");` *z nagłówka <string.h> będzie* ***zawsze lepsze*** *niż porównywanie znak napisów po znaku …*]
- Używamy starannie dobranych nazw zmiennych
  [w kalendarzu: "dzien", "miesiac", "rok" lepsze niż "x", "y", "z"]
- Zostawiamy **oddzielny wiersz** dla nawiasów klamrowych: "{", "}"

Dobry styl programowania (cz.3)

- Używamy stałych symbolicznych (*makrodefinicje*)

```
#define Dwa 2
#define cisnienie P
```

- Staramy się, aby nasze funkcje były *możliwie* małe
  (*jeśli długą funkcję można podzielić na mniejsze – DZIELIMY!*)
- Stosujemy **wcięcia** w tekście programu dla *uwypuklenia* znaczenia bloków
- Zaznaczamy koniec funkcji `/* ODPOWIEDNIM KOMENTARZEM */`

Słowa kluczowe

Następujące symbole (=*słowa kluczowe*) są zastrzeżone* i nie mogą być redefiniowane (np. jako nazwy zamiennych, funkcji, lub etykiety)

auto	else	int	typedef
break	entry	long	switch
case	enum	register	union
char	extern	return	unsigned
continue	float	short	void
default	for	sizeof	while
do	goto	static	
double	if	struct	

Dodatkowo, standard ANSI zastrzegł: const signed volatile

*) *Czy to oznacza, że **każdy inny** symbol może być użyty w moim programie np. jako nazwa zmiennej?*

NIESTETY NIE, implementacja bibliotek standardowych oznacza *zajęcie* znacznej liczby innych symboli na zmienne i funkcje *globalne* (a funkcje w C są **zawsze** globalne).

Konkretna implementacja może zajmować więcej nazw niż wymaga standard; np. funkcja `atoi` (zamieniająca napis na *int*) opisana w rozdz.3. *Kernighana i Ritchie'go* (s.78), jest już zajęta w implementacji gcc (w bibliotece <sdtlib.h>) !

Na ogół, nazwy zmiennych/stałych globalnych są tak dobrane, aby trudno było w nie trafić przypadkowo, np.:

```
#define _GLIBCXX_TR1_STDLIB_H 1
```

Kiedy “komputer się myli”, czyli o reprezentacji liczb

(“Komputer to maszyna!”)

Pamięć komputera: ciąg *komórek*, 1 komórka = 1 bajt = 8 bitów

(8 cyfrowa l.dwójkowa = 2 cyfrowa l.szesnastkowa, z przedziału 0 ÷ 255, LUB: 0 ÷ 0xFF)

Większe liczby => *więcej bajtów potrzebnych do zapisu …*

Typy liczbowe (w języku C):

całkowite: `char` — 1 bajt/8 bitów,

zakres: 0 ÷ 2^8-1=255 (jeśli *unsigned**; signed:* -128 ÷ 127)

`int` — ***co najmniej**** 2 bajty/16 bitów; 0 ÷ 2^{16}-1=65635

*) *Zazwyczaj typ* `int` *odzwierciedla rozmiar l.całkowitej wynikający z architektury maszyny, współcześnie: 32 lub 64 bajty.*

**)*Przykładowa pełna deklaracja zmiennej:* `unsigned int i;`

Typy zmiennopozycyjne:

`float` — pojedynczej precyzji, 32 bity (*co najmniej!*), ~7 cyfr, zakres:

$-3.4 \times 10^{38} \div -3.4 \times 10^{-38}$, 0, $3.4 \times 10^{-38} \div 3.4 \times 10^{38}$

double — podwójnej precyzji, 64 bity (*co najmniej!*), ~15 cyfr, zakres:

$-1.7 \times 10^{308} \div -1.7 \times 10^{-308}$, 0, $1.7 \times 10^{-308} \div 1.7 \times 10^{308}$

Kwalifikatory `short` (krótki) oraz `long` (długi) modyfikują typy podstawowe, np. deklaracja: `long int i;` wymusza 32 bity.

(W tym przykładzie słowo `int` ***może zostać pominięte****.)*

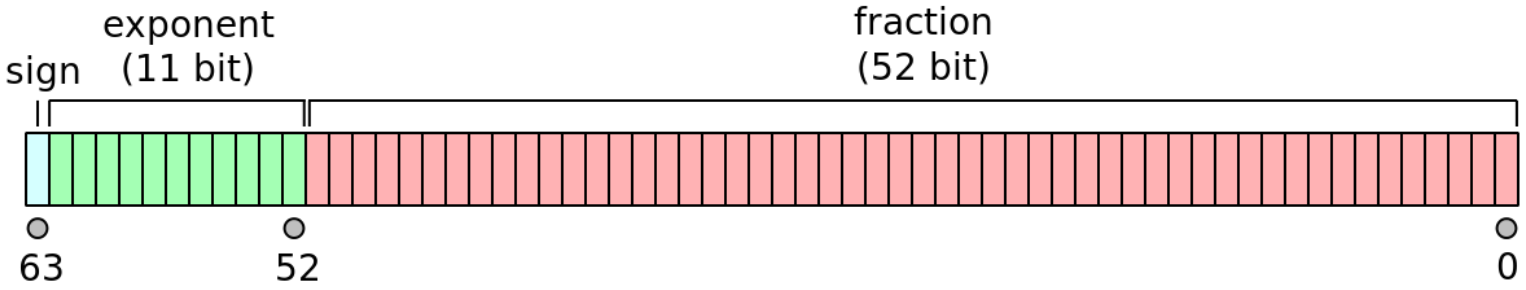
A zatem: Kiedy “komputer się myli” ?

Obliczenia zmiennopozycyjne wykonywane są ze ***skończoną dokładnością***. *Matematycznie*, np. typ `double` z dodawaniem (i odejmowaniem) *nie stanowi grupy!*

Rozważmy działania: $(2x10^{-20} + 1) - 1 == 0$

$2x10^{-20} + (1 - 1) == 2x10^{-20}$

(*W pierwszym przypadku następuje tzw.* **przepełnienie mantysy**)



Jednostki leksykalne języka C

słowa kluczowe: `int, goto, if, else, …`

stałe*: `0xF85  2.17e+10  'a'  '7'`

łańcuchy znaków: `"Tekst"` <=> tablicy:

`{'T','e','k','s','t','\0'}`

identyfikatory: `A-Z, a-z, 0-9`

(pierwszy znak: zawsze litera lub `'_'`)

operatory: `=  +  *  +=  &&  ,  ?:`

separatory: `' '  ( )  [ ]  { }  ;`

*) *znaki specjalne:* `'\n', '\t', '\"', '\\', …`

Operatory w języku C *(uwagi ogólne)*

- Podobnie jak w matematyce, *mnożenie i dzielenie ma pierwszeństwo przed dodawaniem i odejmowaniem*
- Kilka operatorów ma niefortunnie określone priorytety *(nie ma języka bez wad ...)*
- Dlatego, **kolejność innych działań** najlepiej wymuszać za pomocą nawiasów okrągłych: `( )`
- *Nie jest określona* **kolejność obliczania wartości argumentów** operatora, np. w instrukcji: `x=f()+g();`
  funkcja f może być wykonana jako pierwsza, ***ale nie musi!***
- Aby wymusić pożądaną kolejność – *wynik pośredni należy przychować w zmiennej tymczasowej*

Priorytety i łączność operatorów w C

Operatory	Łączność
() [] -> .	lewostronna
! ~ ++ -- + - * & (*typ*) sizeof	prawostronna
* / %	lewostronna
+ -	lewostronna
<< >>	lewostronna
< <= > >=	lewostronna
== !=	lewostronna
&	lewostronna
^	lewostronna
|	lewostronna
&&	lewostronna
||	lewostronna
?:	prawostronna
= += -= *= /= %= ^= |= <<= >>=	prawostronna
,	lewostronna

Jednoargumentowe operatory +, -, * oraz & mają priorytet wyższy niż ich odpowiedniki dwuargumentowe.