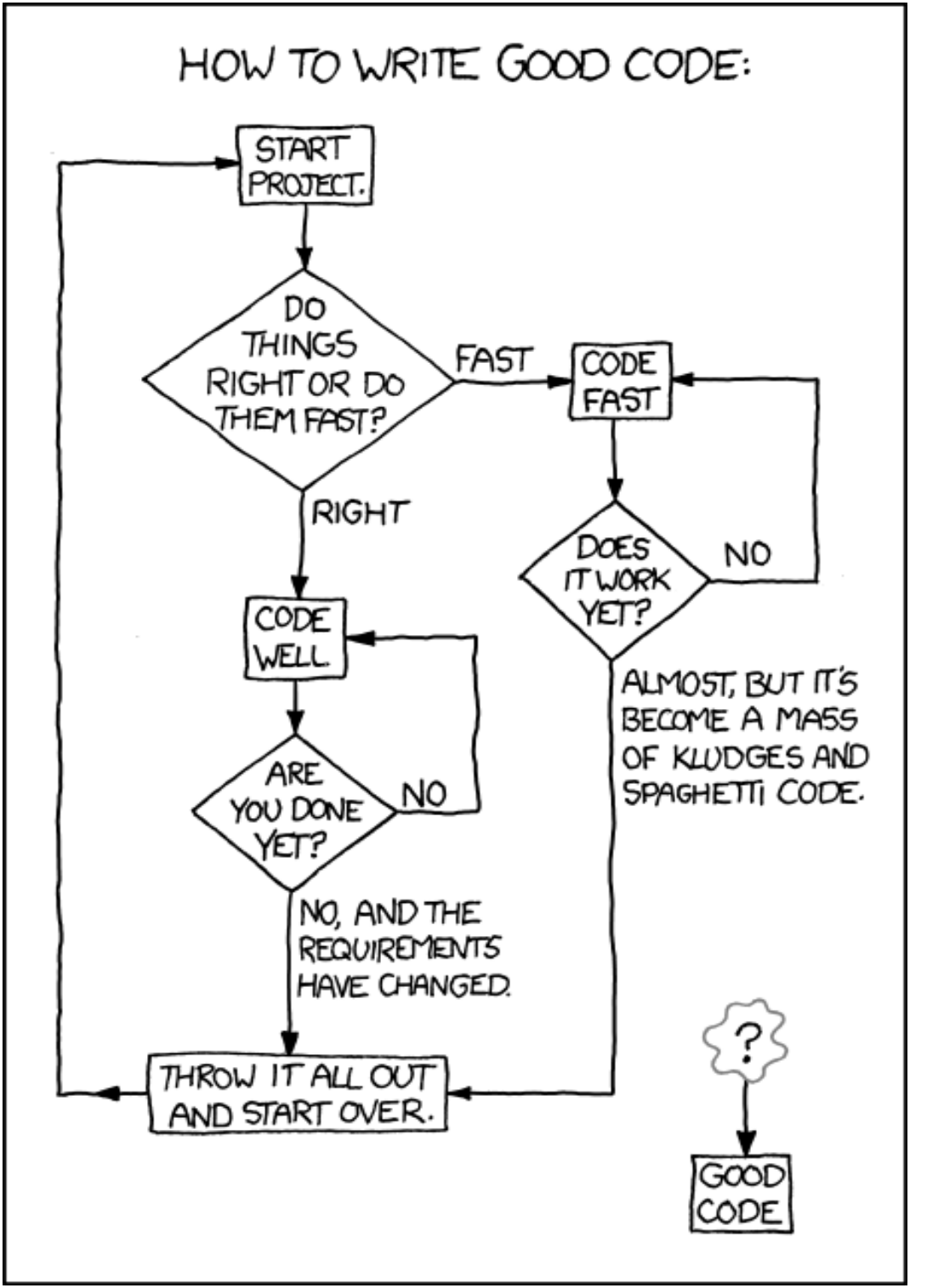

HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Wojciech Myszka

Informatyka I — RAM031042 . Wprowadzenie do informatyki — MCM032102. wer. 16 z drobnymi modyfikacjami!

2023-04-16 14:00:54 +0200
kompilowane: 16 kwietnia 2023

People seem to equate programming with coding, and that's a problem. Before you code, you should understand what you're doing. If you don't write down what you're doing, you don't know whether you understand it, and you probably don't if the first thing you write down is code. If you're trying to build a bridge or house without a blueprint—what we call a specification—it's not going to be very pretty or reliable. That's how most code is written. Every time you've cursed your computer, you're cursing someone who wrote a program without thinking about it in advance.

Leslie Lamport, 2014

Spis treści

1. Wprowadzenie

1.1. Wstęp

Na podstawie slajdów, które przygotowałem do wykładów:

1. *Informatyka I* dla Automatyki i Robotyki,
2. *Wprowadzenie do informatyki* dla Mechatroniki,

postanowiłem opracować coś w rodzaju skryptu. Nie tyle może skryptu co bryku[1]. Zawiera on treść wszystkich slajdów plus dodatkowe komentarze, czasami przybliżające trudniejsze treści. W żadnym wypadku nie jest to **kompletny kurs programowania w języku C**, a raczej pomoc wyjaśniająca te rzeczy, na które nie starczyło na wykładzie miejsca.

1.2. Program zajęć

Program zajęć jest przy okazji planem tego bryku. Zwracam uwagę, że zajęcia dotyczą języka ANSI C. Tak też są ustawione kompilatory w laboratorium komputerowym, w którym prowadzone są zajęcia praktyczne.

1. Wprowadzenie. Algorytm. Schematy blokowe. Idea programowania strukturalnego.
2. Struktura programów w C. Identyfikator, typy danych (typy fundamentalne: całkowite, rzeczywiste, znakowe, logiczny), deklaracja i inicjalizacja zmiennych, definiowanie stałych. Komunikacja poprzez konsolę. Operatory: arytmetyczne, logiczne, inkrementacji, dekrementacji, przypisania. Obliczanie wartości wyrażeń.

[1] Po angielsku mogłoby się to nazywać *Lecture Notes*.

3. Struktury sterowania obliczeniami: rozgałęzienia i skoki, pętle pojedyncze i zagnieżdżone. Instrukcje proste i złożone; instrukcje warunkowe, wyrażenia warunkowe, instrukcje iteracyjne.
4. Preprocesor: dyrektywy, makrodefinicje[2].
5. Funkcje: budowa funkcji, argumenty funkcji, wynik wykonania funkcji, definicje i deklaracje globalne, argumenty funkcji main, rekurencja.
6. Tablice (tablice jedno i wielowymiarowe), łańcuchy znaków.
7. Wskaźniki. Pamięć dynamiczna.
8. Struktury danych, unie: deklaracja struktury, definiowanie zmiennej strukturalnej, tablice struktur, wskaźniki a struktury danych.
9. Operacje na plikach: otwieranie, zamykanie plików, czytanie i zapisywanie do plików.
10. Formatowanie w operacjach wejście/wyjście. Binarne wejście/wyjście.
11. Operacje na łańcuchach znaków.
12. Programowanie strukturalne w praktyce: podział programu na moduły, struktury danych, kompilacja.
13. Programy pomocnicze: diff, make, systemy rcs i cvs, debugger. Zarządzanie wersjami. Środowiska zintegrowane.

Każdy z bloków zaplanowany jest na około dwie godziny zajęć, ale z powodów różnych (godziny Rektorskie, Dziekańskie, czy inne sytuacje awaryjne) — może to się zmieniać.

Dokładnie ten sam program (w założeniu) realizowany jest również na kierunku Automatyka i Robotyka. Ze względu na mniejszą liczbę godzin wykładu, zakłada się, że studenci będą musieli poświęcić więcej czasu na samodzielne studia.

1.3. Po co

Może się pojawić pytanie *Po co uczyć się programowania podczas studiów na Wydziale Mechanicznym?* Przy czym to, że jest to Wydział Mechaniczny oraz, że zajęcia są obowiązkowe — zostawiam na później. Tu zajmę się tylko tym czym warto się zajmować.

Pozwolę sobie przytoczyć tu 12 przewidywań na temat przyszłości programowania [24].

[2] To, z braku czasu, studenci AiR powinni opanować samodzielnie.

1. Procesory graficzne (GPU) będą naszymi następnymi procesorami. Procesory bardzo potaniały. Oferują nam co najwyżej kilka rdzeni (ang. *core*). Natomiast dobre karty graficzne mają nieprawdopodobnie duże możliwości obliczeniowe — podczas ich normalnej pracy sa one niezbędne do tworzenia dynamicznie zmieniającego się obrazu. Programiści coraz chętniej sięgają po ich możliwości obliczeniowe. Warto o tym pamiętać.
2. Bardzo wiele przyszłego programowania dotyczyć będzie baz danych (*big data*). Coraz więcej normalnych, wydawałoby się, operacji wymagać będzie dostępu do baz danych. I to nie zwykłych baz danych tylko ogromnych zbiorów kiepsko strukturyzowanych danych — *big data*.
3. JavaSript do wszystkiego. Po pierwsze nie należy mylić języka JavaScript (używanego głównie przez przeglądarki internetowe) z językiem Java... Po drugie powyższe stwierdzenie nie oznacza, że zanikną inne języki programowania.
   Natomiast trzeba pamiętać, że coraz więcej aplikacji używanych jest w przeglądarce. Używanie JavaScriptu i HTML5 może być wystarczające w bardzo wielu zastosowaniach.
4. Android na każdym urządzeniu.
5. Internet rzeczy — kolejne nowe platformy.
6. Open source na wiele sposóbów.
7. Oparte na platformie WordPress aplikacje webowe.
8. Programy zostaną zastąpione przez „wtyczki" (plug-ins).
9. Niech żyją polecenia wydawane w terminalu!
10. Nie liczmy na dalsze upraszczanie językow programowania.
11. Programowaniem zajmować będą się programiści z krajów o najniższym koszcie pracy.
12. W dalszym ciagu szefostwo nie będzie rozumieć o co chodzi z tym programowaniem.

I jeszcze jedna historyjka

A modern luxury car has something close to 100 million lines of software code in it, running on 70 to 100 microprocessors. The navigation system of the Mercedes-Benz S-Class alone exceeds 20 million lines of code.

Manfred Broy, of the Technical University, Munich, told IEEE Spectrum that software and electronics can make up 35 to 40 percent

wer. 16 z drobnymi modyfikacjami!

of the cost of a premium car today. At $10 a line, a cost he calls too low, 100 million lines represent $1 billion of investment for each car.

According to consultant Frost & Sullivan, those 100 million lines of code will rise to 200 or 300 million within a few years.

Wzywamy samochody do przeglądu...

And the software that controls the "drive-by-wire" accelerators of Toyota and Lexus vehicles is one potential culprit in the tangled collection of issues, allegations, and recalls of many of those vehicles for so-called "sudden acceleration" problems.

The NHTSA's mission is to "save lives, prevent injuries, reduce vehicle-related crashes." If it cannot properly analyze those systems, or even understand at a deep-code level how they work, then the agency is useless at overseeing the entire "Safety" part of its mandate.

The agency has an annual budget of more than $800 million, and it employs 635 of people. That not a single one of them is an EE or software engineer borders on the criminally insane.

Na podstawie [22].

1.4. Slajdy

Slajdy do wykładu dostępne są na stronach WWW poświęconych zajęciom.

— Automatyka i Robotyka: http://www.immt.pwr.wroc.pl/~myszka/InformatykaI/ARE0004/
— Mechatronika: http://www.immt.pwr.wroc.pl/~myszka/InformatykaI/INM0610/

1.5. Podstawowa literatura

Bardzo trudno wskazać jakieś **bardzo dobre książki** do języka C. Jest ich za dużo. Ten wykład oparty jest głownie na książce napisanej przez autorów języka C [10], warto też zajrzeć do książki niejako pomocniczej [21] — zawierającej rozwiązania i dyskusje ćwiczeń. Można też korzystać ze strony internetowej [8]. Dostępna jest ogromna liczba zasobów internetowych z

zasobami Wikibooks [2] na czele, ale do klasycznych internetowych zasobów należą [20, 16] — rzeczy, które powstały w toku udzielania odpowiedzi w Internecie na pytania zadawane przez początkujących[3]. Jeżeli chodzi o programowanie nieco ogólniej to warto zajrzeć do [19] natomiast źródłem inspiracji dla wielu algorytmów prezentowanych tu był znakomnity cykl książek Donalda E. Knutha [13].

1.6. Kilka słów o notacji

W bryku, na slajdach do wykładu oraz na stronach www poświęconych laboratoroim/projektowi pojawiają się (czasami) dziwne znaczki:

— w tych miejscach, gdzie chodzi o zaakcentowanie, że jest tam spacja używany jest ten znak „␣”.

— w kliku przykładach na końcu linii, która ma kontynuację w następnym wierszu umieszczałem znak \. Jest to konwencja czasami stosowana podczas wprowadzania długich poleceń (tak w systemach unixowych jak i języku C). Można to sobie wytłumaczyć tak: *znak nowej linii oznacza przejście do nowej linii, dodanie znaku backslash nadaje mu znaczenie specjalne: niby przechodzimy do nowej linii, ale jej tam nie ma.*
Konieczność posiadania takiej możliwości najlepiej tłumaczy poniższy kod:

```
#include <stdio.h>

int main(int argc, char **argv)
{
        printf("Ala␣ma␣kota␣\
␣␣␣␣␣␣␣␣i␣psa");
        return 0;
}
```

Powyższe skompiluje się poprawnie (choć odstęp przed i będzie trochę za duży). Natomiast poniższe — nie:

[3] Ten element kultury Internatu popada dziś w zapomnienie, ale trzeba pamiętać, że w czasach gdy nie było Wikipedii ogromne rzesze internautów gromadziły wiedzę z różnych dziedzin i udostępniały ją w Usenecie. Serwis pod adresem http://www.faqs.org/faqs/ gromadzi to co udało się z tego dziedzictwa zachować.

```
#include <stdio.h>

int main(int argc, char **argv)
{
        printf("Ala␣ma␣kota
␣␣␣␣␣␣␣␣i␣psa");
        return 0;
}
```

Choć najlepszym rozwiażaniem będzie chyba to:

```
#include <stdio.h>

int main(int argc, char **argv)
{
        printf("Ala␣ma␣kota␣"
        "i␣psa");
        return 0;
}
```

Tak na marginesie — zwracam uwagę jak bardzo przydaje się specjalny znaczek na oznaczanie odstępów!

2. Programowanie

2.1. Cel zajęć

Celem[1] tych zajęć jest nauczenie Państwa:

1. Programowania
2. Programowania w języku C

W pierwszej kolejności należałoby odpowiedzieć na pytanie „Po co!?" Krótka odpowiedź na tak zadane pytanie jest następująca:

— *Bo tak!*

Dłuższa odpowiedź będzie jakoś taka:

— *Rada Wydziału Mechanicznego (Studium Mechatroniki) w swej mądrości zdecydowała, że każdy absolwent AiRu/Mechatroniki, kończący studia I stopnia powinien znać co najmniej jeden język programowania. Jako język podstawowy uznano język C.*

Zdaję sobie sprawę, że ogarnięcie obu tych rzeczy na raz jest bardzo trudne. Być może nawet (w pierwszej chwili) niemożliwe. Co więcej niekoniecznie trzeba się starać na raz ogarniać obie.

Kolejne, naturalne pytanie będzie jakoś takie:

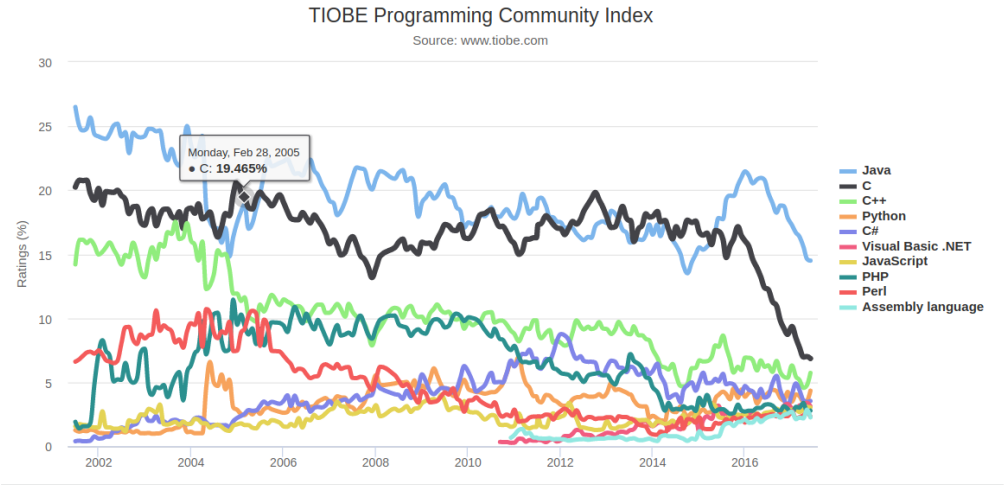
Czemu język C?

Tu odpowiedź jest znacznie bardziej skomplikowana. Odpowiedzi można szukać bądź w historii jezyków programowania lub w różnych portalach dla programistów. Ja wybrałem TIOBE Company[2]

Na rysunku 2.1 przedstawiona jest „popularność" różnych języków programowania. Język C od wielu lat znajduje się w czołówce (aktualnie na drugim miejscu, nawet jeżeli jego popularność spada).

[1] No dobra. Ja nie wiem co jest Państwa celem. Być może macie, po prostu, sporo czasu...

[2] TIOBE is specialized in assessing and tracking the quality of software. We measure the quality of a software system by applying widely accepted coding standards to it.



Rysunek 2.1. Popularność różnych języków programowania według TIOBE

2.2. Programowanie

Co to jest programowanie? Nie potrafię znaleźć dobrej, oficjalnej, definicji tego słowa (czy procesu). Pozwalam sobie zatem zacytować za (angielską) Wikipedią:

The purpose of programming
is to find a sequence of instructions that will automate performing a specific task or solving a given problem. The process of programming thus often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Co na nasze przetłumaczyć można jakoś tak: *Celem programowania jest znalezienie sekwencji poleceń pozwalających zautomatyzowanie wykonanie zadania lub rozwiązanie zadanego problemu. Dlatego też proces programowania wymaga wiedzy z zakresu wielu dziedzin, w tym dobrej znajomości problemu, różnych specjalizowanych algorytmów i logiki formalnej.*
2023-04-16 14:00:54 +0200

Warto też popatrzeć na motto tego bryku, sformułowane przez Lesliego Lamporta[3]:

Programming

People seem to equate programming with coding, and that's a problem. Before you code, you should understand what you're doing. If you don't write down what you're doing, you don't know whether you understand it, and you probably don't if the first thing you write down is code. If you're trying to build a bridge or house without a blueprint—what we call a specification—it's not going to be very pretty or reliable. That's how most code is written. Every time you've cursed your computer, you're cursing someone who wrote a program without thinking about it in advance.

Co na nasze tłumaczy się jakoś tak: *Ludzie zwykli utożsamiać* programowanie *z* kodowaniem *i to jest problem. Zanim zaczniesz kodować powinieneś rozumieć co robisz. Jeżeli nie zapiszesz sobie co zamierzasz osiągnąć, nie wiesz nawet czy to rozumiesz i, najprawdopodobniej, nie rozumiesz, jeżeli zaczynasz od pisania kodu. Jeżeli rozpoczniesz budowę mostu albo domu bez szczegółowego planu (tu nazywamy to specyfikacją) to nie ma szans żeby był on jakoś specjalnie ładny czy solidny. Za każdym razem kiedy klniesz na komputer — przeklinasz kogoś, kto napisał program bez wcześniejszego przemyślenia co on ma robić.*

Można powiedzieć, że programowanie przypomina trochę budowanie czegoś z klocków. Jak się chce osiągnąć oszałamiający efekt — trzeba mieć dokładny plan. W przeciwnym razie powstające dzieło trzeba wielokrotnie przerabiać i wielokrotnie burzyć to co już osiągnięto, żeby poprawić jeden klocek, zmienić jego kolor, usytuowanie czy orientację.

2.3. Na czym polega programowanie?

— Mamy do rozwiązania problem

[3] Leslie Lamport oprócz tego, że jest twórcą systemu składu tekstów LaTeX (którego używam do składu slajdów i składu tego bryku) jest cenionym informatykiem i laureatem nagrody Alana Turinga za „wkład to teorii i praktyki rozproszonych i równoległych systemów obliczeniowych" przyznanej mu w roku 2013. Zaliczany również do grona pionierów informatyki.

— Decydujemy, że użyjemy komputera Zastanawiamy się, jak komputer może nam pomóc
  — jest jakaś gotowa „aplikacja” która się nada?jeżeli tak — użyjemy jej
  — jeżeli nie — musimy ją stworzyć.
— *???* Jak się do tego zabrać?

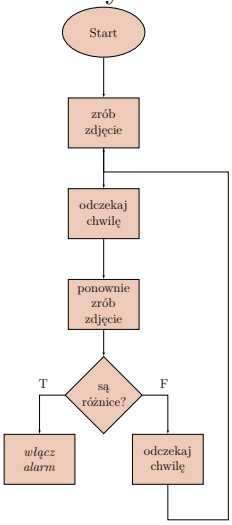
2.3.1. Wykrywanie intruza

Weźmy jakiś prosty przykład. Na początek załóżmy, że mamy komputer z kamerą i chcemy użyć go do nadzoru jakiegoś pomieszczenia. Trzeba obmyślić jakiś sposób[4] postępowania.

Przyjąć można, że kamera co jakiś czas wykonuje fotografię nadzorowanego obszaru i porównuje ją z poprzednim zdjęciem. Jeżeli zauważy jakieś różnice — zgłasza alarm. Jeżeli nie — kontynuuje pracę.

Zaproponowany algorytm jest bardzo naiwny. Jako zadanie domowe mogę polecić zastanowienie się nad jakimś lepszym, uwzględniającym zmiany oświetlenia, cienie i inne mogące pojawić się problemy.

Zaczniemy od schematu blokowego.



Kwestie do rozstrzygnięcia:

1. Jak robić fotografię?
2. Jak „odczekać chwilę” i ile to „chwila”?

[4] Ten „sposób“ to będzie nasz algorytm. Zakładam, że to pojęcie nie jest Państwu obce.

3. Jak stwierdzić czy są różnice?
4. Co zrobić po zasygnalizowaniu alarmu?

2.3.2. Największy wspólny dzielnik

Algorithm

In mathematics, computing, linguistics and related subjects, an **algorithm** is a sequence of finite instructions, often used for calculation and data processing. It is formally a type of effective method in which a list of well-defined instructions for completing a task will, when given an initial state, proceed through a well-defined series of successive states, eventually terminating in an end-state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as probabilistic algorithms, incorporate randomness.

Można to przetłumaczyć jakoś tak: *W matematyce, informatyce, lingwistyce i dziedzinach pokrewnych **algorytm** to ciąg precyzyjnie zdefiniowanych instrukcji, używany często w obliczeniach i podczas przetwarzania danych. Z formalnego punktu widzenia jest to efektywna metoda, w której ciąg dobrze zdefiniowanych poleceń startując z pewnego stanu początkowego i przechodząc przez kolejne stany pośrednie prowadzi ostatecznie do stanu końcowego. Przejście z jednego stanu do drugiego nie musi być deterministyczne, niektóre algorytmy (nazywane probabilistycznymi) dopuszczają losowość.*

Można powiedzieć, że programowanie przypomina trochę budowanie czegoś z klocków. Jak się chce osiągnąć oszałamiający efekt — trzeba mieć dokładny plan. W przeciwnym razie powstające dzieło trzeba wielokrotnie przerabiać i wielokrotnie burzyć to co już osiągnięto, żeby poprawić jeden klocek, zmienić jego kolor, usytuowanie czy orientację.

Blockly

— Blockly to też język programowania!
— Ma ograniczoną liczbę bloczków.
— Wydaje się, że dobrze trenuje myślenie nad tym co ma być zrobione.
— Pozwala zapomnieć o niezrozumiałej, czasami, składni języka programowania.
— *Trochę marudny.*
— *Nawet dosyć prosty program potrafi być spory (na ekranie).*

wer. 16 z drobnymi modyfikacjami!

Wcielenia Blockly

Google Blockly występuje w dwu „wcieleniach”:

1. Gotowe aplikacje przygotowane z jego użyciem: https://blockly-games.appspot.com/.
2. Wersja dla deweloperów: https://developers.google.com/blockly/ zawierająca szereg informacji.

Instalacja Google Blockly...

...na lokalnym komputerze jest bardzo prosta:

1. Ściągnąć musimy plik:
   — zip z adresu https://github.com/google/blockly/archive/master.zip
   — lub użyć programu git w formie: `git clone https://github.com/google/blockly.git`
2. Ściągnięte pliki (gdy nie korzystamy z programu git) należy „rozpakować” (powinna zostać utworzona kartoteka o nazwie `blockly-master`
3. Przechodzimy do „podkartoteki” `/demos/code/` i oglądamy w przeglądarce plik `index.html`

Tak na marginesie — Na serwerach github dostępna jest również wersja „gier” zrobionych w Blockly: https://github.com/google/blockly-games.

W żadnym wypadku nie twierdzę, że Blockly może zastąpić język C. Języki różnią się znacznie (pomijając wygląd) zestawem udostępnionych konstrukcji programistycznych. Ale, wydaje mi się, że pozwalają łatwiej ogarnąć ideę programowania (czyli składania konstrukcji programu z elementarnych bloczków) i — ze względu na charakter zmniejszają liczbę błędów formalnych (konstrukcja poleceń, średniki).

Opanowanie języka C (albo kolejnego, innego języka) polega na opanowaniu tych wszystkich subtelnych różnic między językami programowania...

Alternatywa dla Blockly

Jest bardzo wiele podobnych, „obrazkowych” języków programowania. Obszerna lista znajduje się na stronie Wikipedii: https://en.wikipedia.org/wiki/Visual_programming_language

1. App Inventor IDE for Android apps from MIT.
2. Code.org K-12 computer science.

3. Wonder Workshop Robots for play and education.
4. Gameblox Introduction to Game Design from MIT.
5. Made with Code Encouraging girls to code.
6. Code Spells Programming in a virtual world.
7. BlocksCAD 3D printing.
8. Lil'Bot Self-balancing robot.
9. Custom Packer Human-robot packing system.
10. MII Scratch Stwórz historyjki, gry i animacje

Parę uwag dodatkowych

1. Po napisaniu programu zostanie on automatycznie przetworzony do jednej z postaci wynikowych:
   — JavaScript
   — Python
   — PHP
   — Dart.
   Wystarczy wybrać odpowiednią pozycję z menu poziomego. Tak przygotowane programy można uruchamiać.
2. Konwersja do XML (ostatnia pozycja na liście tego menu) pozwala zapisać tworzony program. Po wyświetleniu zawartości, wystarczy skopiować ją do notatnika i zapisać na dysku. Później wystarczy plik w notatniku otworzyć i przekopiować zawartość do „pustego" obszaru XML.
3. Istnieje (bardzo niedoskonała) wersja programu blockly, która generuje od w języku C: http://hcilab.cju.ac.kr/blockly/apps/webc/index.html

Alternatywne języki programowania

Jak dla kogoś C będzie za trudny, a wzgardzi C — może zająć się programowaniem w języku Brainfuck. Jest bardzo prosty. Do dyspozycji jest 8 (słownie 8) poleceń.

Znak	Znaczenie	Odpowiednik w C
>	zwiększa wskaźnik o 1	++p
<	zmniejsza wskaźnik o 1	−−p
+	zwiększa o 1 w bieżącej pozycji	++(∗p)
−	zmniejsza o 1 w bieżącej pozycji	−−(∗p)
.	wyświetla znak w bieżącej pozycji (ASCII)	putchar(∗p)
,	pobiera znak i wstawia go w bieżącej pozycji (ASCII)	∗p=getchar()
[	skacze bezpośrednio za odpowiadający mu [, jeśli w bieżącej pozycji znajduje się 0	**while**(∗p){
]	skacze do odpowiadającego mu [	}

Zakłada się dodatkowo, że program w Brain** ma do dyspozycji bardzo dużą pamięć (zapewnienie kompletności w sensie Turinga może wymagać założenia, że pamięć ta, podobnie jak taśma maszyny Turinga, jest nieskończenie wielka). Organizacja pamięci jest bajtowa (program operuje na pojedynczych bajtach — choć to nie musie być konieczne wymaganie).

Brainfuck

1. Pomijając kwestie, że nie wszystko tu może być dla Państwa jasne (wskaźniki, które do końca będą niejasne) język jet bardzo skąpy
2. można w nim pisać różne programy.

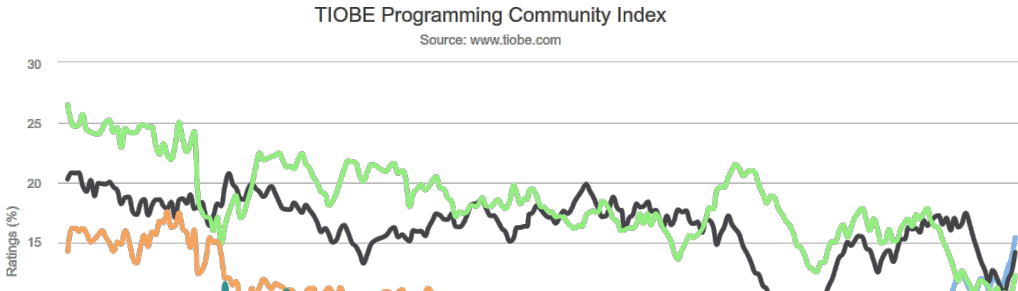
```
,[.,]
```

3. Co tłumaczy się:
   — czytaj jeden znak
   — jak zero to kończ zero (znak o kodzie zero) oznacza, że nie ma już nic do czytania
   — w przeciwnym razie wydrukuj go
   — czytaj znak
   — wróć do poprzedniego nawiasu

Najpopularniejsze języki...

...za TIOBE

2023-04-16 14:00:54 +0200



wer. 16 z drobnymi modyfikacjami!

3. Algorytmy

3.1. Wstęp

Programowanie to czynność wtórna w stosunku do tworzenia algorytmów. I to ich tworzenie sprawia prawdziwą przyjemność. Rozpoczniemy od przypomnienia podstawowych instrukcji. Mogą być przydatne informacje zawarte w wykładzie z Technologii Informacyjnych[1].

3.2. Algorytm

Algorytm

Słowo „algorytm" jest bardzo nowe (w pewnym sensie).

Pochodzi od nazwiska Muḥammad ibn Mūsā al-Khwārizmī — perskiego matematyka (IX w) i pierwotnie oznaczało (każde) obliczenia w dziesiętnym systemie obliczeniowym.

Algorytm to jednoznaczny przepis przetworzenia w skończonym czasie pewnych danych wejściowych do pewnych danych wynikowych (Wikipedia). W tym znaczeniu pojawił się w latach pięćdziesiątych XX wieku.

3.2.1. Przykład algorytmu

Algorytm Euklidesa

[1] https://kmim.wm.pwr.edu.pl/myszka/dydaktyka/technologie-informacyjne/

Oto jedna z wersji algorytmu Euklidesa:*Dane są dwie dodatnie liczby całkowite m i n, należy znaleźć ich* ***największy wspólny dzielnik (NWD)*** *tj. największą dodatnią liczbę całkowitą, która dzieli całkowicie zarówno m jak i n.*

1. *[Znajdowanie reszty] Podziel m przez n i niech r oznacza resztę z tego dzielenia. (Mamy $0 \leqslant r < n$.)*
2. *[Czy wyszło zero?] Jeśli $r = 0$ zakończ algorytm; odpowiedzią jest n.*
3. *[Upraszczanie] Wykonaj $m \leftarrow n$, $n \leftarrow r$ i wróć do kroku 1.*

Alternatywą dla tego algorytmu może być naiwne postępowanie odtwarzające istotę zagadnienia. Popatrzmy na problem: największy wspólny dzielnik dwu liczb. I zacznijmy analizować od końca:

1. Mamy zatem dwie liczby: m i n.
2. Znaleźć należy ich dzielniki. Niech M oznacza wszystkie dzielniki[2] liczby m, a N wszystkie dzielniki liczby n.
3. Wspólne dzielniki zawarte będą w zbiorze $W = M \cap N$ będącym częścią wspólną zbiorów M i N.
4. Na koniec, wreszcie trzeba znaleźć w zbiorze W element maksymalny[3].

Teraz trzeba już tylko zaproponować algorytmy składowe:

1. Znajdowania wszystkich dzielników liczby (i tworzenia „zbioru” wszystkich dzielników)[4].
2. Znajdowania części wspólnej dwu zbiorów (iloczynu).
3. Znajdowania elementu maksymalnego w zbiorze.

Zwracam uwagę, że zaproponowany przez Euklidesa algorytm jest znacznie prostszy: na każdym kroku wykonywane są **jednakowe** i bardzo proste czynności. To nie jedyne sformułowanie algorytmu Euklidesa.

Inne sformułowanie może być takie: szukamy NWD dla dwu liczb: a i b; większą z liczb a i b zastępujemy różnicą $\max(a, b) - \min(a, b)$. Procedurę powtarzamy aż obie liczby będą takie same.

---

[2] Zbiór wszystkich dzielników.

[3] Element maksymalny w zbiorze (uporządkowanym) to taki element x, że w zbiorze nie ma większych od niego.

[4] Ten algorytm będzie stosowany dwukrotnie: raz dla liczby m i raz dla n.

3.2.2. Cechy algorytmu

Skończoność

Po pierwsze **powinien być skończony**; oznacza to, że po skończonej (być może bardzo dużej) liczbie kroków algorytm się zatrzyma[5]. Pytanie pomocnicze: Co gwarantuje, że algorytm Euklidesa zakończy się w skończonej liczbie kroków?

Procedura, która ma wszystkie cechy algorytmu poza skończonością nazywana jest *metodą obliczeniową*. Podaj przykłady metod obliczeniowych realizowanych przez rzeczywiste komputery.

Dobre zdefiniowanie

Po drugie **powinien być dobrze zdefiniowany**. Każdy krok algorytmu musi być opisany precyzyjnie. Wszystkie możliwe przypadki powinny być uwzględnione, a podejmowane akcje dobrze opisane[6]. Oczywiście język naturalny nie jest wystarczająco precyzyjny — może to prowadzić do nieporozumień. z tego powodu używa się bardziej formalnych sposobów zapisu algorytmów, aż po języki programowania...

Dane wejściowe

Po trzecie **powinien mieć precyzyjnie zdefiniowane dane wejściowe**. Pewne algorytmy mogą nie mieć danych wejściowych (mieć zero danych wejściowych). Dane wejściowe to wartości, które muszą być zdefiniowane zanim rozpocznie się wykonanie algorytmu.

Dane wyjściowe

Po czwarte **zdefiniowane dane wyjściowe**. Daną wyjściową algorytmu Euklidesa jest liczna n która jest naprawdę największym wspólnym dzielnikiem danych wejściowych. Osobną sprawą jest pokazanie skąd wynika, że wynik algorytmu Euklidesa jest rzeczywiście NWD liczb m i n.

---

[5] Ale sama skończoność to jednak za mało — z praktycznego punktu widzenia **dobry** algorytm powinien gwarantować, że obliczenia zostaną zakończone w skończonym ale **rozsądnym** czasie!

[6] Zwracam też uwagę, że algorytmy kucharskie nie są odpowiednio precyzyjne: co to znaczy „lekko podgrzej”?

Efektywność

Po piąte algorytm **powinien być określony efektywnie** to znaczy jego operacje powinny być wystarczająco proste by można je (teoretycznie?) wykonać w skończonym czasie z wykorzystaniem kartki i ołówka.

3.2.3. Przykładowe algorytmy

1. Przepis kucharski,
2. Algorytm Euklidesa (patrz 3.2.1),
3. Wyszukiwanie osoby najwyższej/najniższej
4. Algorytm Euklidesa (drugi wariant — tak zwany „Algorytm B" patrz rozdz. 3.11)

3.2.4. Sposób zapisu algorytmów

1. Język naturalny
2. Schemat blokowy
3. Tablica decyzyjna
4. Maszyna Turinga(???)
5. Język programowania

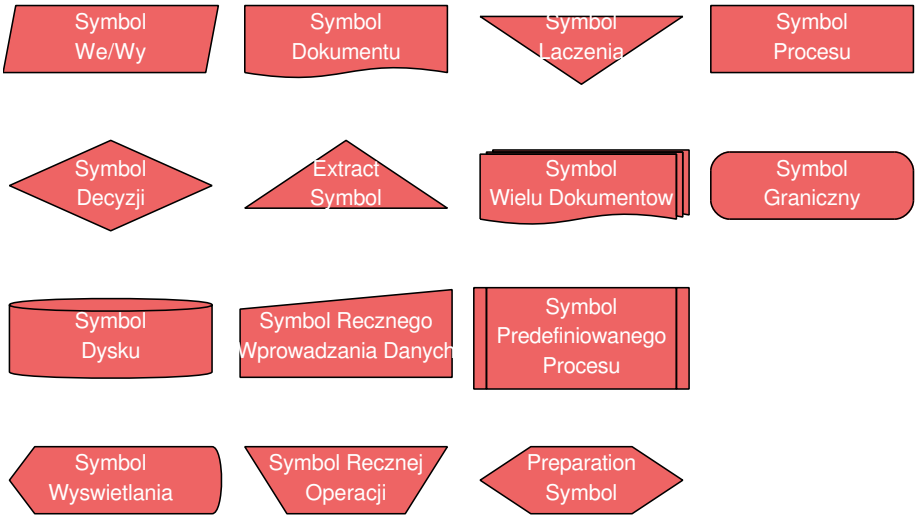
3.3. Schematy blokowe

Schemat blokowy to starodawny sposób przedstawiania zależności pomiędzy wszystkimi czynnościami algorytmu. Idea jest bardzo prosta: w blokach umieszczamy wszystkie czynności, a za pomocą linii (ze strzałkami) wskazujemy kolejność wykonania czynności. W przypadku gdy przedstawiony algorytm stwarza jakiekolwiek problemy bardzo dobrze jest naszkicować go na papierze.

Na rysunku 3.1 przedstawione są najpopularniejsze symbole bloków, ale od ślepego trzymania się kształtów ważniejsza jest próba rozrysowania schematu.

UML — Unified Modeling Language

UML (*Unified Modeling Language* — ujednolicony język modelowania) to graficzny język do obrazowania, specyfikowania, tworzenia i dokumentowania elementów systemów informatycznych.



Rysunek 3.1. Symbole najczęściej używane na schematach blokowych

Umożliwia standaryzację sposobu opracowywania przekrojów systemu, obejmujących obiekty pojęciowe, takie jak procesy przedsiębiorstwa i funkcje systemowe, a także obiekty konkretne, takie jak klasy zaprogramowane w ustalonym języku, schematy baz danych i komponenty programowe nadające się do ponownego użycia.

Wiele informacji na temat UML znaleźć można w internecie. Polecam też stronę http://www.uml.org/ lub książkę [14].

3.4. Programowanie

Najbogatszym źródłem inspiracji do tworzenia algorytmów jest matematyka, gdzie bardzo wiele metod rozwiązywania konkretnych zadań od razu jest podstawą do tworzenia algorytmów. Pomijając różne kłopoty ze zrozumieniem podstaw teoretycznych poruszanej materii, warto szukać tam inspiracji do wprawek w programowaniu.

Aby pokazać poszczególne fazy budowy algorytmu na podstawie wzorów matematycznych rozpatrzmy problem szukania miejsc zerowych wielomianu kwadratowego.

3.4.1. Prosty przykład

Mamy rozwiązać zadanie:

$$ax^2 + bx + c = 0$$

Metoda:

1. Wylicz $\Delta = b^2 - 4ac$
2. Jeżeli $\Delta < 0$ — nie ma pierwiastków rzeczywistych; koniec
3. Oblicz $x_1 = \dfrac{-b - \sqrt{\Delta}}{2a}$
4. Oblicz $x_2 = \dfrac{-b + \sqrt{\Delta}}{2a}$

3.4.2. Realizacja

1. **Wprowadź** a, b, c
2. **Wylicz** Delta=b * b – 4 * a * c
3. **Jeżeli** Delta <0 **to**
   a) **wypisz tekst** "Nie ma pierwiastków rzeczywistych"; ***koniec***
   w przeciwnym razie
   a) x1=(–b – sqrt(Delta))/(2 * a)
   b) x2=(–b + sqrt(Delta))/(2 * a)
4. **Wypisz** "x1 = ", x1, " x2 = ", x2
5. ***koniec***

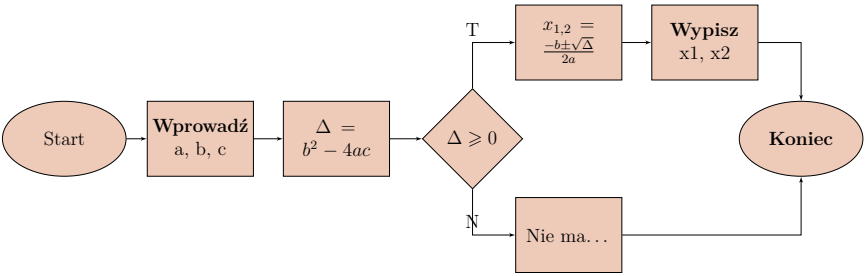
Program nie jest fajny bo ma dwa końce...

Bardzo ważne (w przypadku złożonych programów) jest panowanie nad przebiegiem programu. Zwykle zakończenie obliczeń związane jest z wykonaniem pewnych czynności organizacyjnych. Warto więc zadbać aby **zawsze** były one wykonywane. Sprzyja temu takie rozplanowanie algorytmu, aby program miał jeden *koniec*. Nie jest to specjalnie istotne w przypadku prostych, szkolnych programów. Ale gdy program korzysta z nietypowych urządzeń zewnętrznych — istnieje groźba, że kolejne programy nie będą mogły z nich korzystać, gdy program skończy obliczenia bez wykonania pewnych czynności związanych z obsługą takich urządzeń.

1. **Wprowadź** a, b, c
2. **Wylicz** Delta=b * b – 4 * a * c
3. **Jeżeli** Delta <0 **to**

a) **wypisz tekst** "Nie ma pierwiastków rzeczywistych"
w przeciwnym razie
a) x1=(–b – sqrt(Delta))/(2 * a)
b) x2=(–b + sqrt(Delta))/(2 * a)
c) **Wypisz** "x1 = ", x1, " x2 = ", x2
4. **koniec**

3.4.3. A schemat blokowy?



Na koniec zostaje jeszcze tylko jedna decyzja projektowa: jakiego typu powinny być zmienne. W programie występują następujące zmienne: $a, b, c, Delta, x1, x2$. Powinno być oczywiste że wszystkie te zmienne przechowywać będą mogły „liczby rzeczywiste”[7]. Tak wiec wszystkie zmienne powinny być zmiennoprzecinkowe.

3.4.4. A program w C?

Napisanie na tej podstawie programu w C pozostawiam wnikliwemu czytelnikowi jako zadanie domowe.

3.5. Idea programowania strukturalnego

Idea programowania strukturalnego

1. Jest na ten temat piękna teoria — Dijkstra [3].
2. Wielu teoretyków programowanie strukturalne rozumie nieco inaczej niż Dijkstra.
3. W pewnym uproszczeniu **programowanie strukturalne** to pewne rozszerzenie **programowania proceduralnego**.

[7] Jak wiemy, komputery nie znają liczb rzeczywistych, najbliższym ich przybliżeniem są liczby zmiennoprzecinkowe.

4. Polega na po podziale kodu na dobrze zdefiniowane moduły (bloki).
5. Komunikacja między modułami odbywa się za pomocą precyzyjnie określonych interfejsów.
6. Wedle innych to stosowanie konstrukcji języków programowania takich jak pętle i instrukcje warunkowe, oraz...
7. ...unikanie instrukcji skoku (goto) oraz wielokrotnych punktów wejścia i wyjścia z kodu bloku (podprogramu).

3.5.1. Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia

Załóżmy, że mamy wyznaczyć pierwiastek stopnia n z liczby w, czyli znaleźć taką liczbę x, że:

$$x^n = w \tag{3.1}$$

lub inaczej:

$$x^n - w = 0 \tag{3.2}$$

Jeżeli oznaczymy $f(x) = x^n - w$ to zadanie to można zapisać ogólniej: należy znaleźć takie x, że $f(x) = 0$.

Jeżeli zadanie dodatkowo uprościmy zakładając:
— funkcja ma dokładnie jedno miejsce zerowe,
— jest różniczkowalna,
— jej pochodna w całym przedziale jest albo dodatnia albo ujemna;
to możemy naszkicować rysunek (rys. 3.2) ilustrujący nasze zadanie.

Zaczynamy w punkcie g; wartość funkcji w tym punkcie wynosi $f(g)$. Musimy w jakiś sposób zdecydować w którym kierunku należy wykonać następny krok. Zauważmy, że możemy w tym celu wykorzystać pochodną (czerwona, przerywana linia na poprzednim rysunku). Jeżeli przybliżymy funkcję za pomocą pochodnej (stycznej do funkcji, przechodzącej przez punk $(g, f(g)$ to następnym przybliżeniem będzie punkt przecięcia stycznej z osią x.

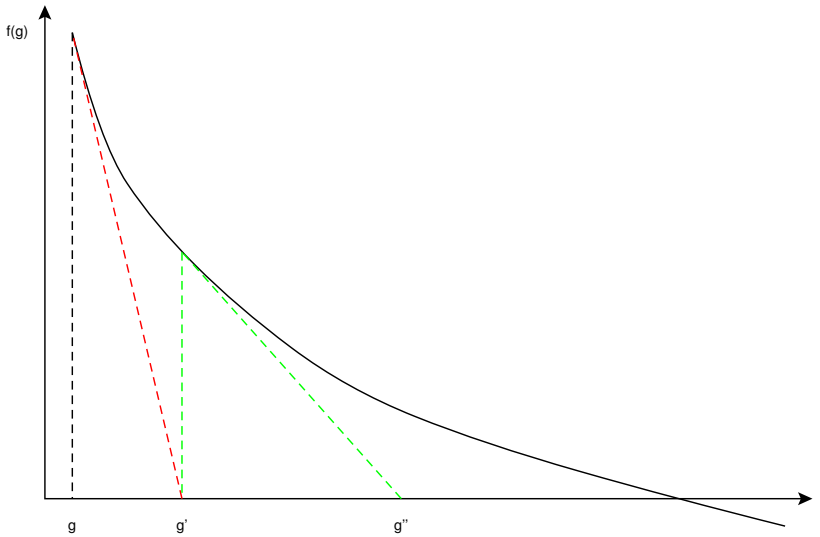
Z równania prostej mamy:

$$\frac{f(g) - 0}{g - g'} = f'(g) \tag{3.3}$$

czyli

$$\frac{f(g)}{f'(g)} = g - g' \tag{3.4}$$

wer. 15 z drobnymi modyfikacjami!



Rysunek 3.2. Przykład funkcji spełniającej założenia oraz ilustracja pierwszych kroków algorytmu

i dalej

$$g' = g - \frac{f(g)}{f'(g)} \tag{3.5}$$

Jeżeli zauważymy, że $f(x) = x^n - w$ oraz, że $f'(x) = nx^{n-1}$ to kolejne przybliżenie wyliczane będzie ze wzoru:

$$g' = g - \frac{g^n - w}{ng^{n-1}} \tag{3.6}$$

albo

$$g' = \frac{ng^n - g^n + w}{ng^{n-1}} = \frac{(n-1)g^n + w}{ng^{n-1}} = \frac{1}{n}\left((n-1)g + \frac{w}{g^{n-1}}\right) \tag{3.7}$$

Gdy $n = 2$, wówczas

$$g' = \frac{1}{2}\left(g + \frac{w}{g}\right). \tag{3.8}$$

Umawiamy się, że program kończy pracę gdy kolejna poprawka g' nie różni się zbytnio od poprzednio wyliczonej wartości g, czyli $|g - g'| < \varepsilon$.

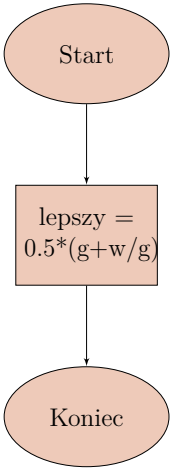
3.5.2. Realizacja programowa

Program będzie się składał z trzech części:

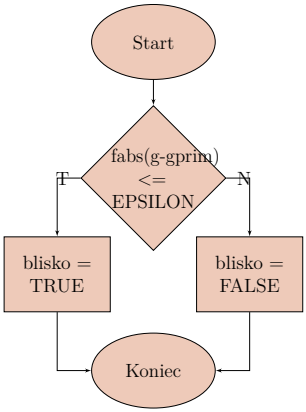
1. `blisko(g, gprim)` — funkcja o wartościach logicznych sprawdzająca czy $|g - g'| \leqslant \varepsilon$,
2. `lepszy(n, w, g)` — funkcja rzeczywista wyliczająca następne, lepsze przybliżenie pierwiastka,
3. `pierwiastek(n, w, g)` — funkcja (rzeczywista) wyliczająca pierwiastek stopnia n z w zaczynając od przybliżenia g.

Uwaga: Dalszy przykład zakłada $n = 2$

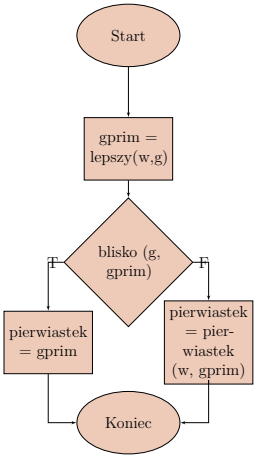
Realizacja programowa
`lepszy(w, g)`



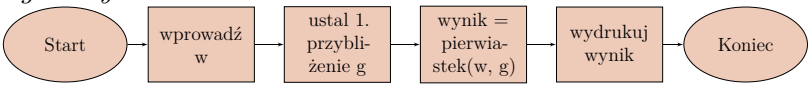
Realizacja programowa
`blisko(g, gprim)`



Realizacja programowa
`pierwiastek(w, g)`



Realizacja programowa
Program główny



Metoda Newtona-Raphsona wróci raz jeszcze w rozdziale **??** (gdzie omawiane są funkcje).

3.6. Czemu C?

Na koniec kilka uwag ogólniejszych o tym czemu studenci Wydziału Mechanicznego Politechniki Wrocławskiej muszą[8] się uczyć programowania w C.

— Nie wiem!
— Jest bardzo popularny.
— Stosunkowo prosty.
— C++ (jeden z podstawowych języków obiektowych) „wyrasta” z C.
— *Dosyć niskiego poziomu.*
— *Deczko „hakerski”.*

Dwie ostatnie cechy powodują, że język C jest trudny w nauce. Pisząc programy nie trzeba korzystać z tych wszystkich hakerskich cech, ale trzeba je rozumieć w programie, który dostanie się do modyfikacji...

3.7. O programach, programowaniu, C...

W dalszej części będzie kilka przykładów-żarcików zaczerpniętych z zasobów języka Python. Część z nich ma konotacje humorystyczne — środowisko programistów ma tendencję do tworzenia różnych, dziwnych czasami żartów.

Hello World — wersja łatwa

```
1 #!/usr/bin/env python
2
3 # example HelloWorld.py on terminal
4
5 print("Hello World!")
```

Powyższy program jest jakimś odpowiednikiem tych aplikacji, które tworzymy na laboratorium. Są to bardzo proste programy, nie używające środowiska graficznego. Dzięki temu mogą być one prymitywnie proste i pozwalają poznać **istotę** programowania.

Hello World — wersja trudna

[8] Po pierwsze nie muszą. Sami wybrali takie studia.

```
#!/usr/bin/env python

# example helloworld.py

import pygtk
pygtk.require('2.0')
import gtk

class HelloWorld:

    # This is a callback function. The data arguments
    # are ignored in this example. More on callbacks
    # below.
    def hello(self, widget, data=None):
        print "Hello World"

    def delete_event(self, widget, event, data=None):
        # If you return FALSE in the "delete_event"
        # signal handler, GTK will emit the "destroy"
        # signal. Returning TRUE means you don't want
        # the window to be destroyed. This is useful
        # for popping up 'are you sure you want to quit?'
        # type dialogs.
        print "delete event occurred"

        # Change FALSE to TRUE and the main window will
        # not be destroyed with a "delete_event".
        return False

    def destroy(self, widget, data=None):
        print "destroy signal occurred"
        gtk.main_quit()

    def __init__(self):
        # create a new window
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        # When the window is given the "delete_event"
        # signal (this is given by the window manager,
        # usually by the "close" option, or on the
        # titlebar), we ask it to call the delete_event()
        # function as defined above. The data passed to
        # the callback function is NULL and is ignored
        # in the callback function.
        self.window.connect("delete_event", self.delete_event)
```

```
46
47          # Here we connect the "destroy" event to a signal
48          # handler.  This event occurs when we
49          # call gtk_widget_destroy() on the window,
50          # or if we return FALSE in the "delete_event"
51          # callback.
52          self.window.connect("destroy", self.destroy)
53
54          # Sets the border width of the window.
55          self.window.set_border_width(10)
56
57          # Creates a new button with the label "Hello
58          # World".
59          self.button = gtk.Button("Hello World")
60
61          # When the button receives the "clicked" signal,
62          # it will call the function hello() passing it
63          # None as its argument.  The hello() function
64          # is defined above.
65          self.button.connect("clicked", self.hello, None)
66
67          # This will cause the window to be destroyed by
68          # calling gtk_widget_destroy(window) when
69          # "clicked".  Again, the destroy signal could
70          # come from here, or the window manager.
71          self.button.connect_object("clicked",
72                                     gtk.Widget.destroy, self.window)
73
74          # This packs the button into the window (a GTK
75          # container).
76          self.window.add(self.button)
77
78          # The final step is to display this newly
79          # created widget.
80          self.button.show()
81
82          # and the window
83          self.window.show()
84
85      def main(self):
86          # All PyGTK applications must have a gtk.main().
87          # Control ends here and waits for an event to
88          # occur (like a key press or mouse event).
89          gtk.main()
90
```

wer. 15 z drobnymi modyfikacjami!

```
# If the program is run directly or passed as an argument
# to the python interpreter then create a HelloWorld
# instance and show it
if __name__ == "__main__":
    hello = HelloWorld()
    hello.main()
```

Wersja „trudna” to aplikacja okienkowa. Żeby ją zaprogramować trzeba nauczyć się ogromnej liczby szczegółów dotyczących rzeczy drugorzędnych z punktu widzenia sposobu konstruowania algorytmu, czy użycia podstawowych konstrukcji języka... Program staje się większy. Komentarze są po to, żeby mniej-więcej zrozumieć co jest robione. ale większość kodo to tworzenie i niszczenie okienek oraz podejmowanie działań w odpowiedzi na zdarzenia czyli na to wszystko co może zrobić użytkownik używający myszy.

Silnia na wiele sposobów

— Newbie programmer

```
#Newbie programmer
def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(x - 1)
print factorial(6)
```

— First year programmer, studied Pascal

```
#First year programmer, studied Pascal
def factorial(x):
    result = 1
    i = 2
    while i <= x:
        result = result * i
        i = i + 1
    return result
print factorial(6)
```

— First year programmer, studied C

```
#First year programmer, studied C
def fact(x): #{
    result = i = 1;
    while (i <= x): #{
        result *= i;
```

```
6           i += 1;
7       #}
8       return result;
9   #}
10  print(fact(6))
```

— First year programmer, SICP

```
1  #First year programmer, SICP
2  @tailcall
3  def fact(x, acc=1):
4      if (x > 1): return (fact((x - 1), (acc * x)))
5      else: return acc
6  print(fact(6))
```

SICP — Structure and Interpretation of Computer Programs[9]

— First year programmer, Python

```
1  #First year programmer, Python
2  def Factorial(x):
3      res = 1
4      for i in xrange(2, x + 1):
5          res *= i
6      return res
7  print Factorial(6)
```

— Lazy Python programmer

```
1  #Lazy Python programmer
2  def fact(x):
3      return x > 1 and x * fact(x - 1) or 1
4  print fact(6)
```

— Lazier Python programmer

```
1  #Lazier Python programmer
2  f = lambda x: x and x * f(x - 1) or 1
3  print f(6)
```

— Python expert programmer

```
1  #Python expert programmer
2  import operator as op
3  import functional as f
4  fact = lambda x: f.foldl(op.mul, 1, xrange(2, x + 1))
5  print fact(6)
```

[9] SICP to popularny w latach osiemdziesiątych podręcznik programowania opracowany na MIT.

— Python hacker

```
#Python hacker
import sys
@tailcall
def fact(x, acc=1):
    if x: return fact(x.__sub__(1), acc.__mul__(x))
    return acc
sys.stdout.write(str(fact(6)) + '\n')
```

— EXPERT PROGRAMMER

```
#EXPERT PROGRAMMER
import c_math
fact = c_math.fact
print fact(6)
```

— ENGLISH EXPERT PROGRAMMER

```
#ENGLISH EXPERT PROGRAMMER
import c_maths
fact = c_maths.fact
print fact(6)
```

— Web designer

```
#Web designer
def factorial(x):
    #------------------------------------------------
    #--- Code snippet from The Math Vault ---
    #--- Calculate factorial (C) Arthur Smith 1999 ---
    #------------------------------------------------
    result = str(1)
    i = 1 #Thanks Adam
    while i <= x:
        #result = result * i #It's faster to use *=
        #result = str(result * result + i)
           #result = int(result *= i) #??????
         result str(int(result) * i)
        #result = int(str(result) * i)
         i = i + 1
    return result
print factorial(6)
```

— Unix programmer

```
#Unix programmer
import os
def fact(x):
```

```
    os.system('factorial ' + str(x))
fact(6)
```

— Windows programmer

```
#Windows programmer
NULL = None
def CalculateAndPrintFactorialEx(dwNumber,
                                 hOutputDevice,
                                 lpLparam,
                                 lpWparam,
                                 lpsscSecurity,
                                 *dwReserved):
    if lpsscSecurity != NULL:
        return NULL #Not implemented
    dwResult = dwCounter = 1
    while dwCounter <= dwNumber:
        dwResult *= dwCounter
        dwCounter += 1
    hOutputDevice.write(str(dwResult))
    hOutputDevice.write('\n')
    return 1
import sys
CalculateAndPrintFactorialEx(6, sys.stdout, NULL, NULL,
                             NULL, NULL, NULL, NULL,
                             NULL, NULL, NULL, NULL,
                             NULL, NULL, NULL, NULL,
                                               NULL, NULL)
```

— Enterprise programmer

```
#Enterprise programmer
def new(cls, *args, **kwargs):
    return cls(*args, **kwargs)

class Number(object):
    pass

class IntegralNumber(int, Number):
    def toInt(self):
        return new (int, self)

class InternalBase(object):
    def __init__(self, base):
        self.base = base.toInt()

    def getBase(self):
```

wer. 15 z drobnymi modyfikacjami!

```
17          return new (IntegralNumber, self.base)
18
19  class MathematicsSystem(object):
20      def __init__(self, ibase):
21          Abstract
22
23      @classmethod
24      def getInstance(cls, ibase):
25          try:
26              cls.__instance
27          except AttributeError:
28              cls.__instance = new (cls, ibase)
29          return cls.__instance
30
31  class StandardMathematicsSystem(MathematicsSystem):
32      def __init__(self, ibase):
33          if ibase.getBase() != new (IntegralNumber, 2):
34              raise NotImplementedError
35          self.base = ibase.getBase()
36
37      def calculateFactorial(self, target):
38          result = new (IntegralNumber, 1)
39          i = new (IntegralNumber, 2)
40          while i <= target:
41              result = result * i
42              i = i + new (IntegralNumber, 1)
43          return result
44
45  print StandardMathematicsSystem.getInstance(
46                  new (InternalBase,
47                  new (IntegralNumber, 2))).calculateFactorial(
48                  new (IntegralNumber, 6))
```

3.8. Ewolucja języków programowania

Kolejny żarcik pokazujący jak z czasem zmieniało się podejście twórców języków programowania do ich opisowości. O ile możemy uznać, że printf("%10.2f", x); wygląda nieco tajemniczo, to alternatywy potrafią być męczące...

— 1980: C

```
1  printf("%10.2f", x);
```

— 1988: C++

```
cout << setw(10) << setprecision(2) << showpoint << x;
```

— 1996: Java

```
    java.text.NumberFormat formatter =
    java.text.NumberFormat.getNumberInstance();
    formatter.setMinimumFractionDigits(2);
    formatter.setMaximumFractionDigits(2);
    String s = formatter.format(x);
    for (int i = s.length(); i < 10; i++)
                        System.out.print(' ');
    System.out.print(s);
```

— 2004: Java

```
    System.out.printf("%10.2f", x);
```

— 2008: Scala and Groovy

```
    printf("%10.2f", x)
```

3.9. Proces tworzenia programu

1. Potrzeba (Zadanie).
2. Pomysł.
3. Algorytm.
4. Program.
5. Kompilacja (przetworzenie z postaci kodu źródłowego do postaci pośredniej).
6. Uruchomienie.
7. (Ewentualnie) odpluskwianie...
8. Poprawa błędów (ewentualnie zmiana algorytmu)

Cały problem, który mamy na tych zajęciach jest taki, że w większości Państwo nie czują potrzeby zaprogramowania czegokolwiek...

Popatrzmy na taki przykład. Opisuje człowiek zasadę wyznaczania daty Świąt Wielkiej Nocy [23]. I można zaproponować zaprogramowanie tego (w języku C). Z drugiej strony trzeba chwil kilka żeby znaleźć, na przykład, to: [4]. Czy warto przepisać program? Czy warto go skompilować i uruchomić, żeby sprawdzić czy działa dobrze? Jaka jest zależność między tablicami opisanymi w artykule, a kodem programu?

wer. 15 z drobnymi modyfikacjami!

Otóż moje zdanie jest takie: Największą wartość ma to co **czynnie** wymyślimy sami. Bezmyślne skorzystanie z gotowca to rozwiązanie najgorsze. Skorzystanie z gotowca i (**skuteczna**) próba zrozumienia jak działa — nie jest działaniem bezsensownym.

3.10. Zadania domowe

1. Co gwarantuje, że algorytm Euklidesa zakończy się w skończonej liczbie kroków?
2. Podaj przykłady metod obliczeniowych realizowanych przez rzeczywiste komputery.
3. Przerysować schematy blokowe Metody Newtona dla przypadku ogólnego (to znaczy dla dowolnego n).
4. Narysować schemat blokowy „Algorytmu B" (poniżej!)

3.11. Algorytm B

Innym sposobem realizacji zadania znajdowania Największego Wspólnego Dzielnika dwu liczb jest algorytm B, prezentowany w dalszej części. Jest to „binarny algorytm Euklidesa". Czemu binarny okaże się później. Natomiast już tu można zauważyć, że jedyne operacje wykonywane to dzielenie przez dwa, zmiana znaku, odejmowanie, sprawdzamie czy liczba jest parzysta oraz podnoszenie dwójki do potęgi. Wszystkie te operacje mogą być zrealizowane w elementarny sposób, z wykorzystaniem najproszych poleceń w kodzie maszynowym.

Algorytm B
Dla danych dodatnich liczb całkowitych u i v algorytm ten znajduje **największy wspólny dzielnik**.

B1 Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.

B2 Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku B4. W przeciwnym razie przyjmij $t \leftarrow u$.

B3 (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.

B4 Jeśli t jest parzyste to przejdź do B3.

B5 Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.

B6 Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku B3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.

Zachęcam do zapoznania się z algorytmem. Zapoznanie powinno polegać na wybraniu kilku pary liczb (dwucyfrowa i trzycyfrowa) i wykonaniu wszystkich obliczeń „na piechotę” tak, żeby móc później odtworzyć go z pamięci...

3.12. Program w C

3.12.1. Program

Tak wygląda program w języku C

```
1 /* Hello World in C, Ansi-style */
2 #include <stdio.h>
3 int main(void)
4 {
5       puts("Hello World!");
6 }
```

Pierwsza linia to komentarz, czyli taki tekst, który nie jest uwzględniany przez kompilator. Komentarz zaczyna się od dwu znakowego nawiasu /*, a kończy dwu znakowym nawiasem zamykającym */. Inny rodzaj komentarza to komentarz jednolinijkowy. Rozpoczyna się od dwu znaków ciach[10] i kończy wraz z końcem linii. Mały przykład:

```
1     a = b / 2;   // Dzielenie całkowitoliczbowe!
```

Pierwsza część linii jest traktowana przez kompilator normalnie, druga jest ignorowana.

Polecenie #include (linia 2) służy do włączenia zawartości zewnętrznego pliku. Będzie o tym mowa mowa, w którymś z następnych wykładów. Patrz rozdział 4.

Linia trzecia definiuje początek funkcji (o tym, też będzie mowa później). Przyjęta konwencja mówi tyle, że zaraz po załadowaniu programu do pamięci, jego wykonywanie rozpoczyna się od wykonania funkcji main.

Pomiędzy nawiasami klamrowymi (linie 4 i 6) zawarta jest treść programu, nazywana często z angielska *body*.

A tak (ten sam) w Pascalu

```
1 program test;
2 {rozne deklaracje}
3 begin
4         writeln('to jest program');
5 end.
```

[10] Po angielsku slash.

Jak widać wersja paskalowa jest bardzo podobna. Główna różnica jest taka, że rolę nawiasów kalmrowych spełniają słowa **begin** i **end**.

W przypadku Blockly nie ma właściwie pojęcia program. Staje sie nim cokolwiek umieścimy w obszarze roboczym (nawet jeżeli nie ma to specjalnego sensu).

3.12.2. Co składa się na program?

Najogólniej na sprawę patrząc program składa się z:

1. Pewnej **struktury**;
2. **Poleceń** wykonywanych przez procesor;
3. Obiektów (zwanych **zmiennymi**) służących do przechowywania danych, wyników i wartości pośrednich uzyskanych podczas obliczeń;
4. **Stałych** używanych podczas obliczeń i do inicjowania wartości zmiennych;

1. **Struktura** to część „organizacyjna”. Bardzo ważna, mocno sformalizowana. Zajmować się nią będziemy nieco później.
2. Polecenia (rozkazy) wykonywane przez komputer (procesor) wydają się być najważniejsze, ale nie mają żadnego znaczenia bez zmiennych.
3. Same **zmienne** nie znaczą wiele i muszą być rozpatrywane łącznie ze kodem.
4. Praktycznie prawie zawsze w programie pojawią się **stałe**. Podobnie jak w matematyce to wartości, które nie zmieniają się w trakcie pracy programu.

Prosty przykład:

```
1  /*
2      Hello World in C, Ansi-style
3  */
4  # include <stdio.h>
5  int main (void)
6  {
7      int z;
8      z = z + 1;
9      puts ("Hello World!"); // druk
10 }
```

wer. 14 z drobnymi modyfikacjami!

— W powyższym kodzie linie o numerach 4–6 oraz 10 to struktura programu.
— Linie 1–3 to komentarz (cokolwiek istotne dla człowieka i zupełnie nierozpoznawalne przez komputer) wieloliniowy. Znaki /∗ i ∗/ to „nawiasy" otwierający i zamykający komentarz. W lini o numerze 9 po znakach // znajdujący się tekst również jetst komentarzem[11].
— Linie 5 i 7 to deklaracje: głównego programu (w pierwszym przypadku) i zmiennej w drugim. main i z to identyfikatory, odpowiednio, programu i zmiennej.
— W linii 7 i 8 występuje zmienna.
— Jedynka (1) w linii 8 oraz tekst ("Hello␣World") w lini 9 to stałe (liczbowa oraz tekstowa).
— puts w linii 9 to wywołanie funkcji[12].

Zwracam uwagę, że wszystkie nawiasy (okrągłe, klamrowe, kwadratowe czy „kątowe": <> albo komentarza) muszą występować w parach, a zakresy ich działania nie mogą się przecinać.

3.12.3. Zmienne

Pojęcie zmiennej jest jednym z najważniejszych pojęć używanych w algorytmice. Zmienne to specjalne obiekty „przechowywane" w pamięci komputera zawierające informacje, które przetwarza algorytm.

Każda zmienna wyposażona jest w unikatową nazwę.

Każda zmienna zajmuje w pamięci komputera jeden lub więcej bajtów, jej zawartość jest zapisywana binarnie, natomiast interpretacja ciągu zer i jedynek zależy od umownego[13] typu zmiennej.

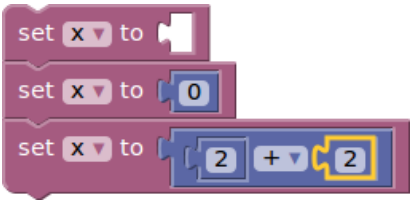
Zmienne

1. Każdy program operuje na pewnych obiektach służących do przechowywania bieżących wartości.
2. Obiekty te nazywa się **zmiennymi**.
3. Każda zmienna musi mieć jakąś nazwę. **Pierwszym znakiem nazwy powinna być litera**, używać można również cyfr i znaków podkreślenia.

[11] Ten rodzaj komentarza nie jest objęty żadnym standardem ale usanjcjonowany praktyką — większośc kompilatorów rozpoznaje go.

[12] Będzie o tym mowa później!

[13] Umownego w tym sensie, że programista decyduje jaki to będzie typ i zawartość zmiennej jest interpretowana zgodnie z tym typem.



Rysunek 3.3. „Instrukcja” podstawienia w blockly w kilku wariantach

4. W odróżnieniu od różnych innych pojemników — te do przechowywania wartości rozróżniają typ przechowywanej wartości.
5. Jak zmienna przechowuje jedną wartość — nazywa się **zmienną prostą**, gdy potrafi przechować więcej wartości — **złożoną**.

Najważniejsza operacja na zmiennej to przypisanie jej wartości. W języku C zapisywana jest tak:
zmienna = wyrażenie;.

W zapisie tym są dwie strony: „lewa” i „prawa”. Komputer najpierw „wylicza” wartość **prawej** strony, a następnie tę wartość przypisuje (podstawia do) zmiennej występującej po stronie lewej. W blockly, aby uniknąć nieporozumień stosowany jest inny „zapis” (rys. 3.3).

3.12.4. Typy danych

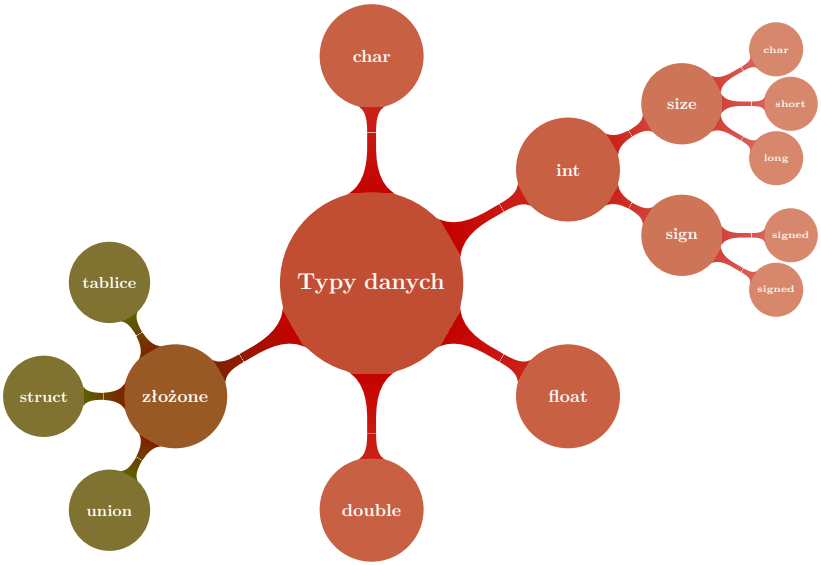
Dostępne w języku C typy danych podsumowano na rysunku 3.4. W części prawej i centralbej umieszczono typy proste (to znaczy takie, które pozwalają na przechowywanie pojedyncej wartości), natomiast w części lewej są typy złożone — pozwalają na przechowywaniu kilku wartości tego samego typu (tablice) lub różnych typów — struktury.

Typ danych określa rodzaj informacji przechowywanej w zarezerwowanym dla zmiennej obszarze pamięci operacyjnej. Powinno być jasne, że wszystkie informacje przechowywane są w postaci dwójkowej (zwanej czasami binarną). Natomiast sposób interpretacji bitów może być różny. Była o tym mowa na technologiach informacyjnych.

Tekst

Bardzo często nawet nie zastanawiamy się w jaki sposób komputer przechowuje w pamięci literki. Zastosowano tu jedno z prostszych rozwiązań: każdemu znakowi przyporządkowano odpowiednią kombinację bitów. I tak:

wer. 14 z drobnymi modyfikacjami!



Rysunek 3.4. Typy danych

— litera „A” to 1000001 czy raczej 01000001,
— natomiast litera „a” to 1100001 (01000001),
a
— znak „!” to 100001 (00100001).

I tak z każdym znakiem dostępnym z klawiatury.

Ciągi zer i jedynek można traktować jako liczby całkowite, zatem literze „A” odpowiada kombinacja bitów 1000001 albo 65 (konwertując do wartości dziesiętnej), „a” to 97, a wykrzyknik („!”) — 33.

Nieco inaczej jest ze znakami specjalnym czy akcentowanymi. Do ich zapisu potrzeba znacznie dłuższych ciągów bitów (i na razie nie będziemy się nimi zajmowali).

Reprezentacja binarna: teksty

1. Każdy znak tekstu to jeden bajt.
2. Zmienna tych **char** zajmuje jeden bajt.
3. Reprezentacją binarną znaku ASCII jest liczba całkowita (bez znaku) o wartości równej kodowi ASCII tego znaku.

Liczby całkowite

W przypadku liczb całkowitych trzeba tylko ustalić na ilu bitach mają być kodowane. Standardowo przyjmuje się, że kodowane są na czterech bajtach. Używane jest kodowanie uzupełnieniowe do dwu, w którym najbardziej znaczący bit (czyli ten po prawej stronie) zarezerwowany jest do przechowywania znaku (0 to + a 1 to −).

Największa wartość (dodatnia), którą można zakodować na czterech bajtach to 01111111 11111111 11111111 11111111 czyli $2^{31}-1$ albo 2 147 483 647
11111111 11111111 11111111 11111111 to -1
a 10000000 00000000 00000000 00000000 to liczba najmniejsza (ujemna, czyli -2^{31} albo −2 147 483 648), którą można zapisać na czterech bajtach.

Natomiast jeżeli zrezygnować ze znaku liczby (i kodowania uzupełnieniowego do dwóch) możliwe do zakodowania na 4 bajtach (32 bitach) liczby będą z zakresu od 0 do
11111111 11111111 11111111 11111111 czyli od zera do 4 294 967 295.

Zazwyczaj posługujemy się wartościami kodowanymi w układzie dziesiętnym, ale czasami znacznie wygodniej jest użyć do zapisu stałych całkowitych układu szesnastkowego. W takim zapisie 1234 (dziesiętnie) to 10011010010 dwójkowo, 2322 ósemkowo i 4d2 (4D2 — wielkość liter nie ma znaczenia) szesnastkowo.

Reprezentacja binarna: liczby całkowite

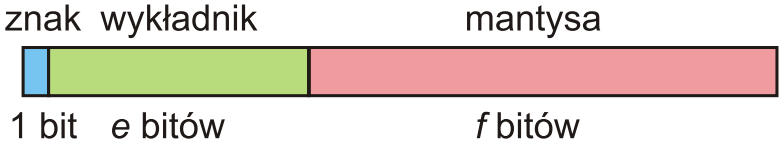
1. Każdy liczba całkowita tekstu to jeden, dwa, cztery lub osiem bajtów...
2. Zmienna typu **int** zajmuje cztery bajty (czyli 32 bity).
3. Liczby całkowite (ze znakiem) reprezentowane są w zapisie uzupełnieniowym do dwu.
4. Najbardziej znaczący bit to bit znaku (1 oznacza liczbę ujemną, 0 dodatnią)
5. Liczby całkowite bez znaku reprezentowane są w kodzie binarnym.

Liczby zmiennoprzecinkowe

Kolejną kategorią danych, które mogą być kodowane dwójkowo to liczby zmiennoprzecinkowe (taki odpowiednik[14] liczb rzeczywistych).

Sposób kodowania tych liczb definiuje norma IEEE 754. Liczba zmiennoprzecinkowa, kodowana jest zgodnie ze schematem przedstawionym na

[14] Tak na prawdę to komputery mogą zapisywać tylko liczby wymierne i to z ograniczonego zakresu, nigdy rzeczywiste.



Typ	Znak	e	f	$\sum$	bin.	dzieś.	C	suffix
Half (IEEE 754-2008)	1	5	10	16	11	≈3.3	—	—
Single	1	8	23	32	24	≈7.2	float	F
Double	1	11	52	64	53	≈15.9	double	
x86 ext. prec.	1	15	64	80	64	≈19.2	—	—
Quad	1	15	112	128	113	≈34.0	long double	L

Rysunek 3.5. Reprezentacja liczby zmiennoprzecinkowej zgdobnie ze standardem IEEE

rysunku3.5.[15] Język C wykorzystuje tylko trzy z podanych typów: **float**, **double** i **long double**. Aby wprowadzać stałę zmiennoprzecinkowe tych typów. trzeba do kązdej dodać przyrostek (suffix) F dla **float** i L dla **long double**.

— Znak to znak całej liczby.
— Wykładnik kodowany jest jako liczba dwójkowa, zapisywana na ośmiu bitach i kodowana w układzie uzupełnieniowym do dwu; przyjmuje wartości z zakresu od -128 do 127[16].
— Mantysa to ciąg bitów rozwinięcia dwójkowego liczby. Pamiętać należy, że liczba (praktycznie) **zawsze** zapisywana jest w postaci znormalizowanej, to znaczy takiej, gdzie przed kropką jest cyfra 1.

Reprezentacja binarna: liczby zmiennoprzecinkowe

1. Każda liczba zmiennoprzecinkowa zajmuje 4, 8 lub 16 bajtów.
2. Dokładnie sposób zapisu liczb definiuje norma IEEE 754.
3. Stosuje się zapis wykładniczy w postaci $znak * mantysa * 2^{cecha}$.
4. Znak to jeden bit, cecha (wykładnik) to 8, 11 albo 15 bitów (binarna liczba całkowita ze znakiem); mantysa (ułamek binarny) — pozostałe bity.

[15] e i f na ilustracji określają liczbę bitów przeznaczoną na wykładnik i mantysę liczby.

[16] Tak, na prawdę sprawa jest jeszcze bardziej skomplikowana, żeby ułatwić prównywanie liczb. Więcej szczegółów można znaleźć w [6].

5. Liczby zapisywane są w postaci znormalizowanej (przed kropką dziesiętną musi być 1 — cyfra różna od zera).
6. Specjalne „wartości” $+\infty$, $-\infty$ oraz NaN (Not a Number) mają również swoje reprezentacje binarne; takie wyniki mogą pojawić się w przypadku „dzielena przez zero”.

Blockly

W języku Blockly w zasadzie nie wystepuje nic takiego jak typy danych. Choć nie należy typów mieszać. Jeżli przyjmiemy, że zmienna X przechowje wartość numeryczną, a zmienna y — tekstową to najlepiej żeby było tak do końca programu.

3.12.5. Nazwy

Nazwy zmiennych

1. Każdy identyfikator musi być unikatowy!
2. Wielkie i małe litery są ważne.
3. Nazwy identyfikatorów powinny zaczynać się od litery i składać z liter i cyfr.
4. Znak _ (podkreślenie) traktowany jest jak litera i może służyć do tworzenia bardziej czytelnych nazw.
5. Znaku _ nie powinno się stosować na początku nazwy — bardzo wiele nazw systemowych zaczyna się od niego.
6. Nie należy używać polskich liter w nazwach.
7. Tradycyjnie zmienne pisze się małymi literami, a różnego rodzaju stałe — wielkimi.
8. Słowa kluczowe (**int**, **void**, **if**, **else**,...) są zarezerwowane i nie mogą być używane jako nazwy.

Słowa kluczowe języka C

— **auto**	— **break**	— **case**	— **char**
— **const**	— **continue**	— **default**	— **do**
— **double**	— **else**	— **enum**	— **extern**
— **float**	— **for**	— **goto**	— **if**
— **int**	— **long**	— **register**	— **return**
— **short**	— **signed**	— **sizeof**	— **static**

wer. 14 z drobnymi modyfikacjami!

— **struct** — **switch** — **typedef** — **union**
— **unsigned** — **void** — **volatile** — **while**

Powyższe słowa są zarezerwowane i nie mogą być używane jako identyfikatory!

3.12.6. Deklaracje zmiennych

Wszystkie używane w programie obiekty muszą być zadeklarowane przed miejscem pierwszego użycia. *Zadeklarowane* znaczy tyle co przedstawione z nazwy i typu: to jest Pan X, a to jest Pani Y. X i Y to nazwy natomiast Pan i Pani to typy. W przypadku niektórych obiektów pojawiać się będą dodatkowe parametry. Będę o nich mówił we właściwym czasie.

Deklaracje zmiennych

Język C zna następujące typy zmiennych:

1. **char** znakowe (elementarną jednostką informacji jest jeden znak).
2. **int** całkowite.
3. **float** zmiennoprzecinkowe („rzeczywiste”).
4. **double** zmiennoprzecinkowe, podwójnej precyzji.
5. **void** pusty typ; nie można zadeklarować zmiennej takiego typu, ale może być on wykorzystany do zwrócenia uwagi, że funkcja nie zwraca wartości lub, że nie przyjmuje parametrów.

Przykład deklaracji: *typ nazwa*;

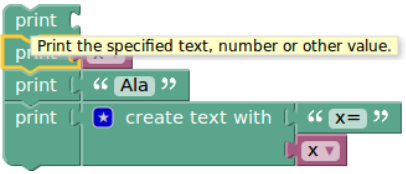
```
1        int a;
2        char b;
3        float c;
4        double d;
```

Każda deklaracja składa się z dwu części: pierwsza (**int**, **char**, **float**, **double**) to typ przechowywanej zawartości, druga (a, b, c, d) to nazwa zmiennej.

Wyprowadzanie wartości zmiennych

Wyprowadzanie wartości zmiennych

printf("Jakiś tekst %*cośtam* tekst", *zmienna*);



Rysunek 3.6. Wyprowadzani wartości

Na potrzeby pierwszych zajęć, przed dokładniejszym poznaniem funkcji **printf** przedstawiam podstawowe informacje na temat wyprowadzenia wartości zmiennej na ekran.

1. Funkcja **printf** ma zmienną liczbę argumentów: jeden jest obowiązkowy, pozostałe są nieobowiązkowe (opcjonalne).
2. Pierwszym argumentem powinien być napis (stała tekstowa) zawierająca zawartości, która ma być skopiowana na ekran. Wystąpienie wśród te zawartości znaku specjalnego % (procent) powoduje specjalne działanie funkcji **printf**.
3. Na jej podstawie pobierana jest zawartość kolejnego parametru z listy argumentów funkcji **printf**. Zawartość ta sformatowana zgodnie z kolejnymi znakami ciągu ropoczynającego się od znaku % wstawiana jest na jego miejsce.
4. Liczba pół %*cośtam* musi być taka sama jak liczba opcjonalnych argumentów funkcji printf.

```
1 printf("x = %d\n", x);
2 printf("x = %x\n", x);
```

Każda z tych instrukcji najpierw wyprowadzi napis x = , a następnie zawartość zmiennej x skonwertowaną do postaci dzieiętnej (w pierwszym przypadku) i skonwertowaną do postaci szesnastkowej w drugim.

W przypadku blockly wyprowadzanie wartości odbywa się z wykorzystaniem polecenia print (rys. 3.6). Blok *create text with* pozwala połączyć różne wartości w jeden tekst.

Ponieważ uruchamianie programów odbywa się w przeglądarce, po konwersji do JavaScript, przeglądarka przy kolejnym komunikacie daje możliwość zblokowania kolejnych komunikatów. Nie należy tego robić!

Teraz będzie nudno...

wer. 14 z drobnymi modyfikacjami!

. . . ale bez tego się nie da.

Omówię wszystkie typy

1. wskazując jakie dane przechowują
2. opisując jak wyglądają stałe omawianego typu
3. mówiąc jakie operacje można na nich wykonywać
4. w jaki sposób wyprowadzamy wartości tych typów

3.12.7. Typ znakowy — char

1. Zajmuje jeden bajt w pamięci komputera.
2. Typ służący do przechowywania znaków (ewentualnie liczb z zakresu $0 \ldots 255$ albo $-128 \ldots 127$).
3. Każdy znak przechowywany jest jako odpowiadający mu kod ASCII.
4. Znaki zapisujemy w pojedynczych cudzysłowach (w odróżnieniu od tekstów — gdzie cudzysłowy są podwójne).
5. `'a' ; 'A' ; '7' ; '!' ; '$'`
6. Znaki „specjalne” (trudne do uzyskania z klawiatury!):
   — `'\a'` — alarm (sygnał akustyczny terminala),
   — `'\b'` — backspace (usuwa poprzedzający znak),
   — `'\f'` — wysuniecie strony (np. w drukarce),
   — `'\r'` — powrót kursora (karetki) do początku wiersza,
   — `'\n'` — znak nowego wiersza,
   — `'\"'` — cudzysłów,
   — `'\''` — apostrof,
   — `'\\'` — ukośnik wsteczny (backslash),
   — `'\t'` — tabulacja pozioma,
   — `'\v'` — tabulacja pionowa,
   — `'\?'` — znak zapytania (pytajnik),
   — `'\ooo'` — liczba zapisana w systemie oktalnym (ósemkowym), gdzie 'ooo' należy zastąpić trzycyfrową liczbą w tym systemie,
   — `'\xhh'` — liczba zapisana w systemie heksadecymalnym (szesnastkowym), gdzie 'hh' należy zastąpić dwucyfrową liczbą w tym systemie,
   — `'\unnnn'` — uniwersalna nazwa znaku, gdzie 'nnnn' należy zastąpić czterocyfrowym identyfikatorem znaku w systemie szesnastkowym. 'nnnn' odpowiada dłuższej formie w postaci '0000nnnn',

— `'\unnnnnnnn'` — uniwersalna nazwa znaku, gdzie 'nnnnnnnn' należy zastąpić ośmiocyfrowym identyfikatorem znaku w systemie szesnastkowym.

7. *Napisy przechowujemy w tablicach typu znakowego!* — Zmienne złożone.

Działania na zmiennych znakowych

— Takie jak na zmiennych całkowitych!
— Działania wykonywane są na wartościach ASCII odpowiadających literom.

Przykład

```
char a = 'B';
char b = a / 2;
char c = 'Z'/2;
char d = a + 1;
```

Drukowanie zmiennej znakowej

1. putchar(zmienna_znakowa);
2. albo
   printf("Wartość␣zmiennej␣znakowej␣wynosi␣=␣%c\n",␣zmienna_znakow

Przykład

```
#include <stdio.h>
int main ()
{
    char a = 'B';
    char b = a / 2;
    putchar(a);  // albo printf('%c', a);
    putchar('\n');
    putchar(b);  // albo printf('%c', b);
    putchar('\n');  // albo printf('\n', a);
    return 0;
}
```

wer. 14 z drobnymi modyfikacjami!

3.12.8. Typ całkowity int

1. Typ służy do przechowywania liczb całkowitych
2. Stałe całkowite:
   — Liczby można zapisywać w układzie dziesiętnym: `1; 123; 48; -579` itd.,
   — Liczby można zapisywać w układzie ósemkowym (zaczynając od zera): `01; 0123; 047; 0579` — w ostatnim przypadku będzie błąd[17]!
   — Liczby można zapisywać w układzie szesnastkowym (zaczynając od `0x`): `0xffff, 0xabcdef, 0Xabcd, 0xABCD`.

Dodatkowo, gdy korzystamy z typów **unsigned int** albo **long int** można używać odpowiednich „przyrostków" (U, L) do zdefiniowania stałych tego typu. Czyli, na przykład, 0XFUL to stałą tylu **long unsigned int** o wartości 15.

Działania na liczbach całkowitych

1. Dodawanie (nic ciekawego): +,
2. Odejmowanie (nic ciekawego): −,
3. Mnożenie (nic ciekawego): ∗,
4. Dzielenie (*uwaga na kłopoty*): /,
   — wynik dzielenia dwu liczb typu całkowitego jest zawsze liczbą całkowitą!
   — 5 / 2 w wyniku daje 2;
   — 2 / 3 ∗ 3 daje w wyniko 0 (zero!);
5. Reszta z dzielenia (modulo): %,
6. Działania na bitach:
   — iloczyn bitowy: & ,
   — suma bitowa: |,
   — różnica symetryczna: $\widehat{}$,
   — negacja: ∼.

Drobna uwaga na temat negacji (∼). Jest to negacja **bitów** liczby. Zatem, jeżeli mamy x = 1 to w zapisie bitowym liczba wygląda jakoś tak:
000...001
(same zera i jedynka na końcu).

[17] Początkowe zero oznacza, że liczba jest ósemkowa; w takich liczbach dopuszczalne są cyfry od 0 do 7, a w liczbie występuje cyfra 9.

Dokonajmy teraz zamiany każdego zera na jedynkę, a jedynki na zero: 111...110

Otrzymana wartość (w przypadku „standardowych" zmiennych **int** jest liczbą ujemną: bit znaku równy jest 1. A jej wartość w układzie (kodzie) uzupełnień do dwóch to -2[18].

Tak na marginesie ~0 to −1. Operacja wykonywana przez operator ~ nazywana też jest „negacją w kodzie uzupełnień do jedności".

Warto też zwrócić uwagę, że taki rodzaj negazji zamienia liczbę parzystą na nieparzystą[19].

Dzielenie liczb całkowitych

I jeszcze raz: Są dwie operacje: / \%.
W przypadku argumentów całkowitych:
— Dzielenie wykonywane jest jako dzielenie całkowitoliczbowe!
— 3/5 daje w wyniku 0
— $3/5*5$ daje w wyniku 0 (bo operacje wykonywane są od lewej do prawej z powodu jednakowych priorytetów mnożenia i dzielenia).
— \% oznacza resztę z dzielenia całkowitoliczbowego

$$7 \,\%\, 3 = 1$$

bo

$$7 = 2 * 3 + 1$$

oraz

$$3 \,\%\, 7 = 3$$

bo

$$3 = 0 * 7 + 3$$

Jeszcze o dzieleniu liczb całkowitych

Jeszcze parę uwag o dzieleniu całkowitoliczbowym. Popatrzmy na taki program:

```
1 #include <stdio.h>
2 int main ( )
3 {
4  for ( int i = b; i >= 0; i-- )
5    {
```

[18] Przypominam algorytm zmiany znaku: składa się on z dwu etapów: negacji bitów (po zanegowaniu dostaniemy wartość „oryginalną", czyli 1), oraz zwiększenia wartości o 1.

[19] Wszystkie liczby parzyste mają jako najmniej znaczący pit zero, po negacji staje się on równy 1.

```
6       c = a / i;
7       if ( c != poprzednie )
8         {
9           poprzednie = c;
10          printf ( "%d/%d=%d\n", a, i, c );
11        }
12    }
13  return 0;
14 }
```

Program ten dzieli liczbę a (równą na początku 50) przez kolejne liczby startując ze stu i kończąc na zerze. Wynik dzielenie drukuje tylko wtedy, gdy różni się od poprzedniego. Sugeruję przepisanie programu i uruchomnienie go, ale można też popatrzeć co dzieje się u mnie:

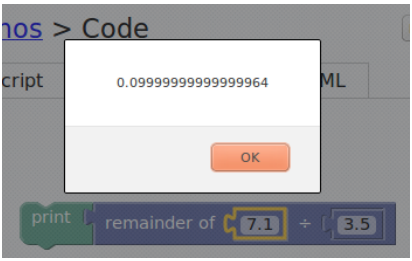
Wyniki programu

```
1  50/100=0
2  50/50=1
3  50/25=2
4  50/16=3
5  50/12=4
6  50/10=5
7  50/8=6
8  50/7=7
9  50/6=8
10 50/5=10
11 50/4=12
12 50/3=16
13 50/2=25
14 50/1=50
15 Floating point exception (core dumped)
```

Zwracam uwagę na komunikat („Floating point exception”). Floating Point znaczy tyle co liczby zmiennoprzecinkowe. Skąd one tutaj? W przypadku blockly trudno mówić o liczbach „całkowitych”, stąd nie ma całkowitoliczbowego dzielenia. Jest natomiast operacja reszty z dzielenia (reminder, patrz rys. 3.7), działą ona natomiast w sposób uogólniony i pozwala na wykonanie operacji $y = 7.1 \% 3.5$. Wynik to (jak można się domyślać) 0.1...

Operatory bitowe

2022-03-03 20:42:10 +0100



Rysunek 3.7. Uogólniona reszta z dzielenia w programie blockly

1. & I (AND),
2. | LUB (OR),
3. ^ różnica symetryczna (exclusive OR), żadne tam podnoszenie do potęgi!
4. << przesuń w lewo,
5. >> przesuń w prawo,
6. ~ dopełnienie do jednego (przełącza wartość bitu: z 0 na 1, a z 1 na 0). *Operator jednoargumentowy!*

Argumenty **całkowitoliczbowe** traktowane cą jako ciągi bitów, a operacje wykonywane są na poszczególnych bitach.

Tu kilka uwag na temat operatorów bitowych. Mogą być one (wbrew pozorom) bardzo przydatne w życiu programisty. Trzeba tylko ciepło pomyśleć o zapisie dwójkowym.

Przykłady:

1. **Sprawdzanie parzystości** (liczy całkowitej).
   Jak wiadomo ostatni bit (najmniej znaczący bit) liczby parzystej to zero, a nieparzystej to 1[20]. Zatem liczba 17 to 00010001. Aby „wyciąć" ostatni bit trzeba na liczbę nałożyć „maskę" która wszędzie za wyjątkiem ostatniej pozycji będzie miała zera: 00000001. Czyli 00000001 & 00010001 da w wyniku 00000001. Natomiast 00000001 & 00010010 da w wyniku 00000000.
2. **Potęgi dwójki** (2^k).
   Zapis dwójkowy jest tak sympatyczny, że waga k–tego bitu to 2^k. Zatem 00001000 to jedynka na pozycji trzeciej[21]. Żeby osiągnąć taką konfigurację

[20] Czemu?

[21] Liczymy od strony **prawej**, a liczenie (zgodnie z tradycją języka C) zaczynamy od zera!

bitów wystarczy jedynkę na pozycji zerowej ($2^0 = 1$) przesunąć w lewo k razy. Czyli wykonać operację 1 << k.

3. **Dzielenie/mnożenie przez dwa.**
   Podobnie jak w układzie dziesiętnym dzielenie/mnożenie przez 10 w układzie dwójkowym mnożenie/dzielenie przez dwa to przesunięcie całej liczby w lewo/prawo[22].

Wyprowadzenie wartości całkowitej

1. Format dziesiętny: %d (%l — long, %h — short, %u — unsigned);
2. Format ósemkowy: %o;
3. Format szesnastkowy: %x;

Jest też specyfikacja %i. Jej działanie podczas wyprowadzania zachowuje się identycznie jak %d. Pewne różnice występują podczas wprowadzania (w funkcji scanf), ale o tym będzie mowa później.

3.12.9. Typ zmiennoprzecinkowy — float

1. Typ służy do przechowywania danych zmiennoprzecinkowych (zmiennopozycyjnych) czyli, tak zwanych rzeczywistych.
2. Liczby zapisujemy z **kropką** dziesiętną: `1.5f; 23.387f; 3.f`
3. Albo używając zapisu „naukowego”: `3e6f; 2.34E8f; 3.14e-5F`
4. Zwracam uwagę na literę f (F) na końcu! Bez niej stałe zmiennoprzecinkowe są traktowane jako typu **double**.
5. Zapis liczb zgodny z normą IEEE 754 na 32 bitach:
   — 7,22 cyfr dziesiętnych (23 bity),
   — zakres $10^{38,23}$.

Dwa programy

```
1 #include <stdio.h>
2 int main()
3 {
```

[22] Przy przesuwaniu w lewo, liczba uzupełniana jest zerami z lewej strony. Podczas przesuwania w prawo duplikowany jest bit znaku. Pamiętać też należy, że operacja ta działa zgodnie z oczekiwaniem jedynie dla liczb dodatnich. W przypadku liczb ujemnych wynik dla liczb nieparzystych zaokrąglany jest „w złą stronę”. Generalnie dokumentacja każdego kompilatora, dla każdej architektury powinna definiować wynik działania tej operacji dla liczb ujemnych.

```
4     float x = 10.3f;
5     printf("%30.20g\n", x);
6     return ( 0 );
7 }
```

```
10.300000190734863281
```

```
1 #include <stdio.h>
2 int main()
3 {
4     double x = 10.3;
5     printf("%30.20g\n", x);
6     return ( 0 );
7 }
```

```
10.300000000000000711
```

O liczbach typu **float** zapomnijmy!

3.12.10. Typ zmiennoprzecinkowy — double

1. Typ służy do przechowywania danych zmiennoprzecinkowych podwójnej precyzji. Liczba zajmuje podwójną ilość miejsca ale też jest dokładniejsza (więcej cyfr po przecinku).
2. Zapis liczb zgodny z normą IEEE 754 na 64 bitach:
   — 15,95 cyfr dziesiętnych (53 bity),
   — zakres $10^{307,95}$.
3. Oprócz typu **double** jest jeszcze **long double** (16 bajtów, 128 bitów). Stałe muszą być wyposażone w przyrostek L.

long double

```
1 #include <stdio.h>
2 int main()
3 {
4     long double x = 10.3L;
5     printf("%35.30Lg\n", x);
6     return ( 0 );
7 }
```

```
10.3000000000000000001734723476
```

Typ **long double** to (teoretycznie) najczęściej liczby 16-bajtowe (128 bitów). Ale najprawdopodobniej 80 bitowe!

wer. 14 z drobnymi modyfikacjami!

Operacje zmiennoprzecinkowe

1. dodawanie,
2. odejmowanie,
3. mnożenie,
4. dzielenie.

Operacje, generalnie, starają się jak najlepiej realizować to co znamy z matematyki. Ale są też i pewne dziwactwa. Na przykład taki program jest najzupełniej poprawny i nie generuje żadnych błędów ani podczas kompilacji ani podczas wykonania:

```
1 #include <stdio.h>
2 int main()
3 {
4     double a, b, c, d;
5     a = 10.;
6     b = 0.;
7     c = a / b;
8     printf("%f\n", c);
9     a = -10.;
10    d = a / b;
11    printf("%f\n", d);
12    d += 10;
13    printf("%f\n", d);
14    d = c + d;
15    printf("%f\n", d);
16    d = d + 5;
17    printf("%f\n", d);
18    return 0;
19 }
```

Oto rezultat wykonania programu:

```
1 1: inf
2 2: -inf
3 3: -inf
4 4: -nan
5 5: -nan
```

inf i −inf to nieskończoność (dodatnia i ujemna), a nan i −nan to Not a Number (nie liczba)[23].

Wyprowadzanie liczb zmiennoprzecinkowych

1. format stałoprzecinkowy:
   %f (albo %*szerokosc_pola*.*liczba_cyfr_po_przecinku***f**) na przykład %10.3f,
2. format zmiennoprzecinkowy: %e,
3. format „mieszany”: %g.

Format „g” jest bardzo wygodny gdy czcemy wyprowadzać liczby, których wartość może zmieniać się w bardzo szerokim zakresie. Generalnie jego idea sprowadza się do podjęcia decyzji ile cyfr znaczących chcemy wyprowadzić i określenia szerokości pola, na którym ma pojawić się liczba.

Wartość będzie drukowana albo jako liczba „z kropką dziesiętną” albo jako liczba w zapisie wykładniczym (liczby bardzo małe lub bardzo duże).

Przykład pokazuje jak to działa.

Program:

```
1  #include <stdio.h>
2  int main()
3  {
4      double y, x = 1.23456789;
5      y = x;
6      int i = 0;
7      while ( i < 10 )
8      {
9          printf("%15g\t␣␣%15g\n", x, y);
10         x = x * 10;
11         y = y / 10;
12         i++;
13     }
14     return 0;
15 }
```

Wynik działania programu.

```
1            1.23457              1.23457
2            12.3457             0.123457
3            123.457            0.0123457
```

[23] https://pl.wikipedia.org/wiki/NaN

```
4          1234.57          0.00123457
5          12345.7         0.000123457
6           123457          1.23457e-05
7      1.23457e+06          1.23457e-06
8      1.23457e+07          1.23457e-07
9      1.23457e+08          1.23457e-08
10     1.23457e+09          1.23457e-09
```

3.12.11. Modyfikatory (kwalifikatory) typów

— **short** — krótki
— **long** — długi

oraz
— **signed** — ze znakiem
— **unsigned** — bez znaku

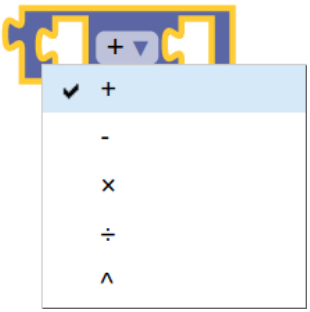
Zatem **int**, **short int**, **long int**, **unsigned int**, unsignet **short int** i **unsigned long int** to zazwyczaj zupełnie różne typy danych (przechowują liczby z różnych zakresów.

Jak się w tym wszystkim połapać?

1. Zapewne nie warto...
2. Ale trzeba wiedzieć, że taki problem jest!
3. Można napisać malutki programik:

```
1       #include <stdio.h>
2
3       int main()
4       {
5       printf("sizeof(char  ) = %d\n", sizeof(char  ));
6       printf("sizeof(short ) = %d\n", sizeof(short ));
7       printf("sizeof(int   ) = %d\n", sizeof(int   ));
8       printf("sizeof(long  ) = %d\n", sizeof(long  ));
9       printf("sizeof(float ) = %d\n", sizeof(float ));
10      printf("sizeof(double) = %d\n", sizeof(double));
11      printf("sizeof(long double) = %d\n", sizeof(long double));
12      return 0;
13      }
```

W naszym laboratorium wyniki tego programu są następujące:



Rysunek 3.8. Operatory arytmetyczne w blockly

```
1       sizeof(char   ) = 1
2       sizeof(short  ) = 2
3       sizeof(int    ) = 4
4       sizeof(long   ) = 8
5       sizeof(float  ) = 4
6       sizeof(double) = 8
7       sizeof(long double) = 16
```

3.12.12. Operatory arytmetczne

— dodawanie: $+$
— odejmowanie: $-$
— mnożenie: $*$
— dzielenie: $/$
— reszta z dzielenie (modulo): % (tylko liczby całkowite!)
— zwiększenie $++$
— zmniejszenie $--$

Blockly zna tylko i wyłącznie najbardziej podstawowe operatory: dodawanie, odejmowanie, mnożenie, dzielenie i podnoszenie do potęgi (rys. 3.8).

Operatory arytmetyczne (cd)
Zwiększenie

1. Występuje w dwu wariantach:
   — przedrostkowym $++x$
   — przyrostkowym $x++$

2. Na pierwszy rzut oka nie ma różnicy. Zarówno $++x$ jak i $x++$ oznacza tyle co $x = x + 1$.
3. W programie może wystąpić tak:

```
int x;
x++;
++x;
```

4. W takim kontekście, po wykonaniu operacji

```
int x, z;
z = 6;
x = 2;
z = z/++x;
```

z przyjmie wartość **2**, a x przyjmie wartość 3. Najpierw zwiększane jest x, a później wykonywane dzielenie.

5. A w takim kontekście, po wykonaniu operacji

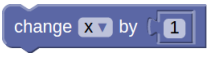
```
int x, z;
z = 6;
x = 2;
z = z/x++;
```

z przyjmie wartość **3**, a x przyjmie wartość 3. Najpierw wykonywane jest dzielenie, a później zwiększane x.

Operatory arytmetyczne (cd)
Zmniejszenie

1. Występuje w dwu wariantach:
   — przedrostkowym $--x$
   — przyrostkowym $x--$
2. Na pierwszy rzut oka nie ma różnicy. Zarówno $--x$ jak i $x--$ oznacza tyle co $x = x - 1$.
3. $--x$ — najpierw zmniejszane jest x i taka wartość bierze udział w kolejnej operacji (jeżeli występuje).
4. $x--$ — aktualna wartość x bierze udział operacji (jeżeli taka występuje), a później zmniejszane jest x.

Blockly ma instrukcję bardzo podobną do instrukcji zwiększania/zmniejszania. Nazywa się change (patrz rys. 3.9) i nie może być używana



Rysunek 3.9. Instrukcja zmiany wartości zmiennej o zadaną wartość

„wewnątrz" innych operacji arytmetycznych. Rezlizuje operację $x = x + \alpha$ (α może przyjmować wartości dodatnie i ujemne).

3.12.13. Operatory przypisania

1. Oprócz najzwyklejszego operatora przypisania (=) używanego w kontekście:
$$a = b$$
co czytamy *zmiennej a przypisz wartośc zmiennej b*
2. Występują operatory „złożone" $+=$ $-=$ $*=$ $/=$ $\%=$ $<<=$ $>>=$ $\&=$ $\hat{}=$ $|=$ stosowane w następujący sposób ($\odot$ oznacza jeden z symboli $+, -, *, / \ldots$)
$$a \odot= b$$
co czyta się
$$a = a \odot b$$

3.12.14. Operatory logiczne

Jednym z większych dziwactw języka ANSI C jest brak typu logicznego. Są instrukcje pozwalające podejmować działania w odpowiedzi na prawdziwość pewnych warunków, ale wszystko załatwiane jest wartościami całkowitymi.

1. == równy
2. ! = nie równy
3. > większy
4. < mniejszy
5. >= większy lub równy
6. <= mniejszy lub równy
7. && logiczne I (AND)
8. || logiczne LUB (OR)
9. ! logiczne NIE (NOT)

wer. 14 z drobnymi modyfikacjami!

Warunek a > b jest, po prostu wyrażeniem arytmetycznym o wartościach całkowitych. Jeżeli istotnie a jest większe od b — wartość tego wyrażenia to 1 (czyli prawda). Gdy a jest mniejsze lub równe b — wartością wyrażenia jest 0 (czyli fałsz).

Zatem można obyć się bez typu logicznego!

1. W języku C **nie ma** typu logicznego!
2. Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest **1** jeżeli jest ono prawdziwe lub **0** jeżeli nie jest prawdziwe.
3. W operatorach logicznych **każda wartość różna od zera** traktowana jest jako prawda.
4. Operatory logiczne mają priorytet niższy od operatorów arytmetycznych; dzięki temu wyrażenie `i < lim-1` jest rozumiane właściwie jako `i < (lim-1)`.

Operatory logiczne
&& i ||

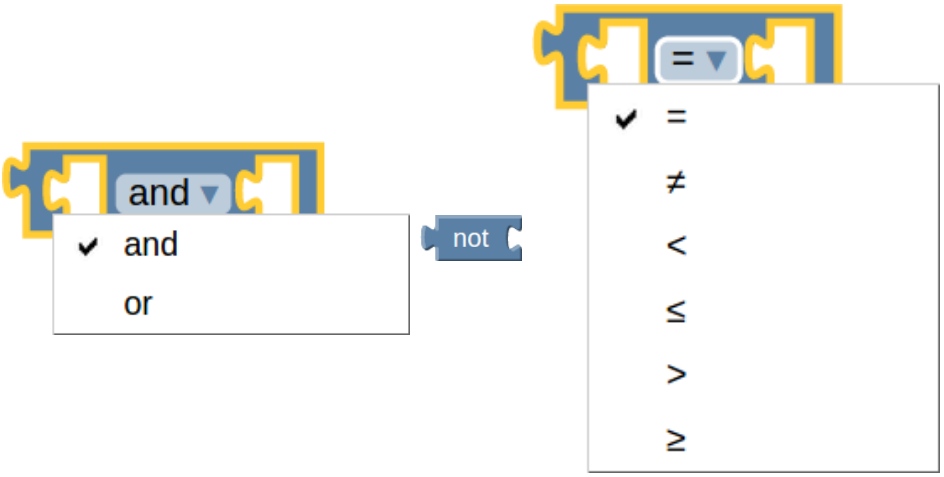
1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.
2. Koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda” lub „fałsz”.
3. Wiele programów korzysta z tego faktu. (*Hakerstwo!*).
4. Priorytet operatora && jest wyższy od priorytetu operatora ||.
5. Priorytety obu operatorów są niższe od priorytetów operatorów relacji i porównania.

Język blockly posiada trzy podstawowe operacje logiczne: AND, OR i NOT (rys. 3.10). Pozwala również na porównywanie liczb.

Operatory logiczne
&& — przykład

```
1           i < lim-1 &&
2           (c = getchar()) != '\n' &&
3           c != EOF
```

1. Najpierw sprawdzamy czy `i < lim-1`.
2. Jeżeli nie — nie zostaną wykonane kolejne operacje.
3. Następnie czytany jest znak (`c = getchar()`)...



Rysunek 3.10. podstawowe operatory logiczne języka blockly

4. . . . a w kolejnym kroku sprawdza się czy nie jest to znak nowej linii `'\n'` (*Hakerstwo!*)
5. W ostatnim kroku sprawdza się czy wcześniej wczytany znak nie jest znakiem końca pliku (EOF)

Przedstawiony fragment kodu dotyczy bardzo typowej sytuacji: program czyta (znak po znaku) informację wprowadzoną z zewnątrz. Nie wie zatem jak długa może ona być. Przygotowany jest pewien bufor służący do przechowania informacji, ale trzeba dbać, żeby nie został przepełniony.

Zmienna i zawiera i liczbę już wprowadzonych znaków. Gdy jest ona mniejsza od pojemności bufora minus 1 ($\text{lim} - 1$) można wczytać jeszcze jeden znak. Zatem skoro pierwszy człon jest prawdą wykonywany jest człon drug wyrażenia logicznego. Składa się on z trzech czynności:

1. czytamy znak (getchar())
2. przeczytany znak zapamiętujemy w zmiennej c (c = getchar()[24]) „Wynikiem" tej instrukcji jest kod przeczytanego znaku. . .
3. który porównujemy ze znakiem '\n' (sprawdzając czy przypadkiem nie jest to znak nowej linii — oznaczający koniec komunikatu.

Kolejny człon złożonej operacji logicznej polega na sprawdzeniu czy przeczytany znak nie niesie przypadkiem informacji o końcu zbioru (EOF[25]).

[24] Zwracam uwagę na nawiasy wokół instrukcji podstawienia! Powinny wystąpić.
[25] *End of File*.

Jeżeli całość wyrażenia jest prawdą — przeczytaliśmy kolejny znak, który powinien być zapisany w buforze. Trzeba będzie czytać kolejny znak. W przeciwnym razie (zapełniony bufor lub przeczytano znak końca wiersza lub informację o końcu danych) — trzeba przeanalizować zawartość bufora.

3.12.15. Operatory inne

1. **sizeof**() wielkość obiektu/typu danych
2. & Adres (operator jednoargumentowy)
3. ∗ Wskaźnik (operator jednoargumentowy)
4. ? Wyrazenie warunkowe
5. : Wyrazenie warunkowe
6. , Operator serii

3.12.16. Kolejność operacji

1. () [] −> .
2. ! ~ ++ −− + − ∗ & sizeof
3. ∗ / %
4. + −
5. << >>
6. < <= >= >
7. == ! =
8. &
9. ^
10. |
11. &&
12. ||
13. ?:
14. = + = − = ∗ = / = % = & = ^= | = <<= >>=
15. ,

3.12.17. Type casting

1. Automatycznie dokonywane są jedynie konwersje nie powodujące straty informacji (np. zamiana liczby całkowitej do float w wyrażeniu f + i).
2. Pewne operacje są niedozwolone (np. indeksowanie tablicy wskaźnikiem typu float) i żadna konwersja nie będzie wykonana.

3. Operacje powodujące utratę informacji (zmiana typu zmiennoprzecinkowego do całkowitego) mogą powodować ostrzeżenie, ale nie są zabronione.
4. Niejawne przekształcenia arytmetyczne działają na ogół zgodnie z oczekiwaniami. Gdy nie korzystamy z typu „unsigned" to w wyrażeniach arytmetycznych:
   — jeżeli jeden z argumentów jest long double, pozostały jest zamieniany do long double,
   — w przeciwnym przypadku jeżeli jeden z argumentów jest double drugi zostanie przekształcony do tego typu,
   — w przeciwnym razie jeżeli jeden jest typu float drugi zostanie przekształcony do tego typu.
   — w przeciwnym przypadku wszystkie argumenty char i short zostaną przekształcone do typu int
   — następnie jeżeli którykolwiek z argumentów ma kwalifikator long drugi zostanie przekształcony do tego typu.
5. Zwracam uwagę, że typ float nie jest automatycznie zmieniany do double
6. W operacji podstawienia typ prawej strony jest przekształcany do typu lewej (co może powodować utratę informacji).
7. Dłuższe (bitowo) liczby przekształcane są do krótszych przez odrzucenie bardziej znaczących bitów.
8. W każdym wyrażeniu można wskazać sposób przekształcenia używając operatora cast (rzut): (nazwa-typu) wyrażenie.

Type casting

— `round()`: round to nearest integer, halfway away from zero
— `rint()`, `nearbyint()`: round according to current floating-point rounding direction
— `ceil()`: smallest integral value not less than argument (round up)
— `floor()`: largest integral value (in double representation) not greater than argument (round down)
— `trunc()`: round towards zero (same as typecasting to an int)

Część I

Instrukcje sterujące

3.13. Ala ma kota

Problem

1. Zadanie polega na tym, żeby opracować algorytm który dla dowolnej liczby całkowitej (być może ograniczonej do zakresu 0—100) wygenerował poprawny (dla języka polskiego) napis: *Ala ma i kot*$\{a|y|ów\}$ gdy i zmienia się w zakresie od 0 do 100.
2. Po co?
   — zapoznanie się z instrukcją **if—then—else**[26],
   — zapoznanie się z ideą rozgałęziania algrytmów,
   — a, ponieważ to pierwszy program, zapoznanie się z podstawowymi konstrukcjami programistycznymi oraz
   — zapoznanie się z instrukcjami dzielenia całkowitoliczbowego...

Najpierw trzeba zastanowić się na czym polega trudność. W tym celu najlepiej powiedzieć sobie głośno jak to będzie z tą swoistą „odmianą przez liczebniki”?

3.13.1. Pakujemy koty po dziesięć

Ala ma kota
Zerowa dziesiątka

— Ala ma 0 kot*ów*.
— Ala ma 1 kot*a*.
— Ala ma 2 kot*y*.
— Ala ma 3 kot*y*.
— Ala ma 4 kot*y*.
— Ala ma 5 kot*ów*.
— Ala ma 6 kot*ów*.
— Ala ma 7 kot*ów*.
— Ala ma 8 kot*ów*.
— Ala ma 9 kot*ów*.

Pierwsza dziesiątka zachowuje się zupełnie niestandardowo! Wszędzie jest „kotów”.

[26] Pamiętamy cały czas, że **then** w języku C nie występuje!

Ala ma kota
Pierwsza dziesiątka

— Ala ma 10 kot*ów*.
— Ala ma 11 kot*ów*.
— Ala ma 12 kot*ów*.
— Ala ma 13 kot*ów*.
— Ala ma 14 kot*ów*.
— Ala ma 15 kot*ów*.
— Ala ma 16 kot*ów*.
— Ala ma 17 kot*ów*.
— Ala ma 18 kot*ów*.
— Ala ma 19 kot*ów*.

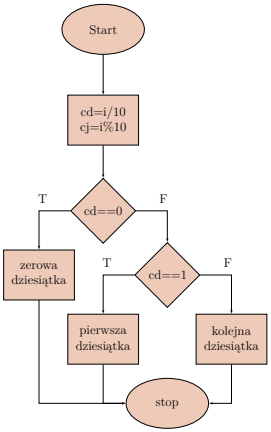
Natomisast każda kolejna dziesiątka zachowuje się już jednakowo (choć inaczej niż „dziesiątka zerowa").

Ala ma kota
Kolejna dziesiątka

— Ala ma 20 kot*ów*.
— Ala ma 21 kot*ów*.
— Ala ma 22 kot*y*.
— Ala ma 23 kot*y*.
— Ala ma 24 kot*y*.
— Ala ma 25 kot*ów*.
— Ala ma 26 kot*ów*.
— Ala ma 27 kot*ów*.
— Ala ma 28 kot*ów*.
— Ala ma 29 kot*ów*.

3.13.2. Idea algorytmu

Idea algorytmu — zmienne pomocnicze
Niech zmienna typy **int**, i oznacza liczbę kotów ($0 \leqslant i \leqslant 100$).
— Wówczas $i/10$ oznacza numer dziesiątki (cd — cyfra dziesiątek),
a
— $i\%10$ oznacza „numer kota w dziesiątce" (cj — cyfra jednostek).



Rysunek 3.11. Schemat blokowy i idea algorytmu „Ala ma kota”

3.13.3. Schemat blokowy

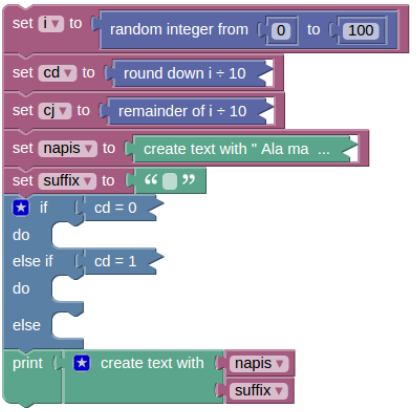
Nawet w przypadku tak prostego algorytmu warto zacząć od naszkicowania schemtau blokowego (przy okazji będzie to schemat instrukcji **if**–**then**–**else**).

Schemat blokowy i ideę algorytmu przedstawia rysunek 3.11.

Na tej podstawie można przygotować zarys programu. W miejscu bardziej szczegółowych instrukcji wstawiam komentarze...

```
1  cd = i / 10;
2  cj = i % 10;
3  printf("Ala␣ma␣%d␣kot", i);
4  if ( cd == 0 )
5  {
6  //    Zerowa dziesiątka
7  }
8  else if ( cd == 1 )
9  {
10 //    Pierwsza dziesiątka
11 }
12 else
13 {
14 //    Kolejna dziesiątka
15 }
```

Ten sam fragmen programu w języku Google blockly przedstawia rysunek 3.12. Z różnych powodów musiałem dokonać pewnych zmian. *i* nie wer. 13 z drobnymi modyfikacjami!



Rysunek 3.12. Fragment programu ala

zmienia się w sposób „ciągły” od zera do stu; po każdym uruchomieni programu generowana jest losowa wartość z tego zakresu.

Inaczej też działa instrukcja wyprowadzania informacji; stąd kolejne drobne korekty.

W kolejnym kroku trzeba zaplanować generowanie przyrostków dla każdej dziesiątki.

Idea algorytmu

Zerowa dziesiątka

Przyrostek dla zerowej dziesiątki może być określony następującym „wzorem matematycznym”:

$$\text{suffix} = \begin{cases} \text{ów} & \text{gdy } cj = 0 \cup 4 < cj < 10 \\ \text{a} & \text{gdy } cj = 1 \\ \text{y} & \text{gdy } 1 < cj < 5 \end{cases}$$

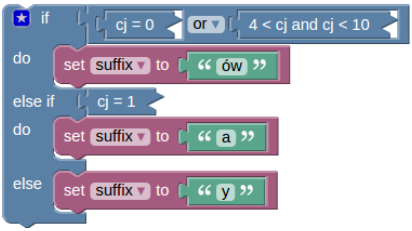
Powyższy „wzór” programujemy również za pomocą „rozgałęzionej instrukcji **if**. Zwracam uwagę, że zapis matematyczny:

$$0 < x \leqslant 5$$

zapisujemy tak: 0 < x && x <= 5.

Powyższe polecenia w języku blockly przedstawia rysunek 3.13.

```
1 //    Zerowa dziesiątka
  2022-02-22 16:14:00 +0100
```



Rysunek 3.13. Instrukcja warunkowa dla „zerowej” dziesiatki

Rysunek 3.14. Pierwsza dziesiątka

```
2 if (cj == 0 || (4 < cj && cj < 10))
3     printf("ow\n");
4 else if (cj == 1)
5     printf("a\n");
6 else
7     printf("y\n");
```

Idea algorytmu

Pierwsza dziesiątka

Na dobrą sprawę nie ma o czym mówić:

```
1 //    Pierwsza dziesiątka
2 printf("ow\n");
```

W blockly również będzie to bardzo proste (patrz rys. 3.14).

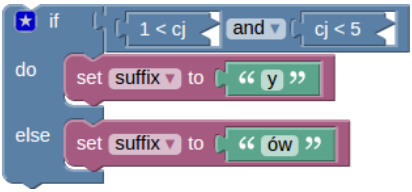
Idea algorytmu

Kolejna dziesiątka

Przyrostek dla każdej następnej dziesiątki może być określony następującym wzorem matematycznym:

$$\text{suffix} = \begin{cases} \text{ów} & \text{gdy } 0 <= cj < 2 \cup 4 < cj < 10 \\ \text{y} & \text{gdy } 1 < cj < 5 \end{cases}$$

wer. 13 z drobnymi modyfikacjami!



Rysunek 3.15. Kolejne dziesiątki w języku blockly

Zwracam uwagę, że żeby uprościć sobie życie, zaprogramuje wyrażenia w odwrotnej kolejności: zacznę od tego prostszego, a ten bardziej skomplikowany warunek będzie załatwiony za pomocą klauzuli **else**.

```
//   Kolejna dziesiątka
if (1 < cj && cj < 5)
   printf("y\n");
else
   printf("ow\n");
```

Kolejna dziesiątka w języku blockly nie różni się specjalnie od zapisu w języku C (rys. 3.15).

Program (prawie kompletny)

```
cd = i / 10;
cj = i % 10;
printf("Ala␣ma␣%d␣kot", i);
if ( cd == 0 )
{
//   Zerowa dziesiątka
   if ( cj == 0 || ( 4 < cj && cj < 10 ) )
      printf("ow\n");
   else if ( cj == 1 )
      printf("a\n");
   else
      printf("y\n");
}
else if ( cd == 1 )
//   Pierwsza dziesiątka
   printf("ow\n");
```

```
17 else
18 {
19 //    Kolejna dziesiątka
20     if ( 1 < cj && cj < 5 )
21         printf("y\n");
22     else
23         printf("ow\n");
24 }
```

W miejsce komentarzy wystarczyło wstawić opracowane wcześniej fragmenty kodu. Zwracam też uwagę, że na samym początku instrukcja printf jest użyta do wydrukowania fragmentu napisu:
printf("Ala␣ma␣%d␣kot", i);
Pozostałe instrukcje służą do dodania przyrostków. (Jak wspominałem drukowanie w blockly zrealizowane jest nieco inaczej. Najpierw w zmiennej pomocniczej „kompletowany' jest napis, który zostaje wydrukowany na samym końcu.)

Kompletny program w języku blockly przedstawia rysunek 3.16. Gotowy program znaleźć można pod adresem `ala_blockly1.xml`.

3.14. Instrukcje

Przyda nam się teraz rozróżnienie między instrukcjami prostymi i złożonymi. Każdy napis zakończony średnikiem traktowany jest jak instrukcja. Ale, oczywiście, musi to być poprawny napis w rozumieniu języka C.

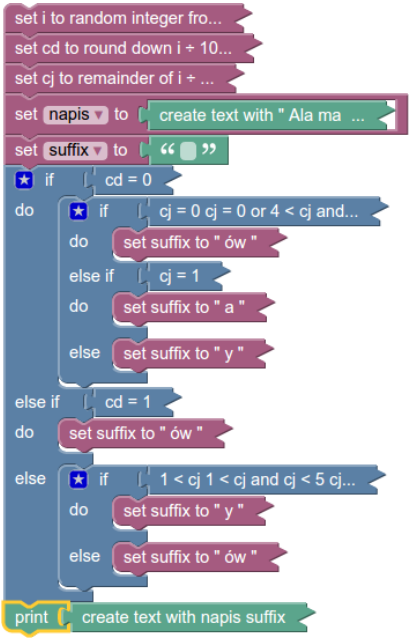
Instrukcje proste

Każde wyrażenie typu `a = b` lub `puts("ala")` staje się instrukcją gdy dodamy na końcu średnik.

Instrukcje złożone

Grupa instrukcji prostych, zamknięta w bloku i traktowana przez kompilator jak jedna instrukcji (w pewnym sensie!).

```
1 {
2   a = b + c;
3   d = e * (f + a);
4 }
```



Rysunek 3.16. Kompletny program w języku blockly; dla oszczędności niektóre bloki zostały „zmniejszone” (*collapsed*)

Uwaga: W ramach każdego bloku można deklarować zmienne **lokalne**. Ich zawartość nie jest dostępna poza blokiem! Natomiast dostępna jest wartość wszystkich zmiennych zadeklarowanych w nadrzędnym bloku (chyba, że „przykryjemy" je lokalną deklaracją).

```
1 {
2   int d = 1;
3   printf("%d\n", d);
4   {
5       int d = 2;
6       printf("%d\n", d);
7   }
8   printf("%d\n", d);
9 }
```

Co pojawi się na wyjściu programu?

3.15. Instrukcje warunkowe

— Wariant „**if**–`then`"

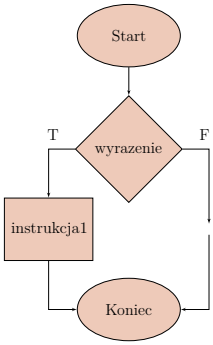
```
1 if (wyrazenie)
2     instrukcja1
```

Schemat blokowy tej instrukcji przedstawia rysunek 3.20. Gdy warunek logiczny jest prawdziwy wykonywana jest instrukcja1; w każdymn przypadku jak następna wykonywana jest kolejna instrukcja po **if**.

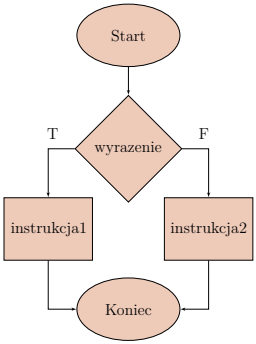
— Wariant „**if**–`then`–`else`"

```
1 if (wyrazenie)
2     instrukcja1
3 else
4     instrukcja2
```

Schemat blokowy przedstawiony na ryskunku 3.18 przedstawia taką właśnie instrukcję. W zależności od wartości wyrażenia warunkowego wykonywana jest albo instrukcja1 albo instrukcja2; później obie ścieżki programu „schodzą się". „instrukcja1" i „instrukcja2" w powyższym to albo jakaś instrukcja prosta (gdy jest tylko jedna) albo instrukcja złożona — gdy



Rysunek 3.17. Podstawowa wersja instrukcji **if**

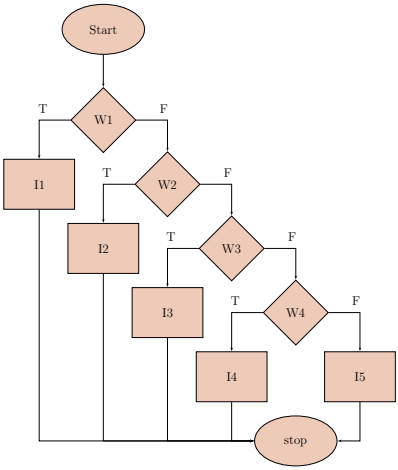


Rysunek 3.18. Schemat blokowy instrukcji **if**–**else**

musimy wykonać dwie albo więcej czynności. W obu schematach blokowych „Koniec” oznacza kolejne instrukcje programu.

— Wariant „**if**–`then`–**else if**” jej schemat blokowy przedstawiony jest na rysunku 3.19.

```
1 if (W1)
2     I1
3 else if (W2)
4     I2
5 else if (W3)
6     I3
7 else if (W4)
8     I4
```



Rysunek 3.19. Schemat blokowy wielokrotnej instrukcji **if**–**else**–**if**

```
 9 else
10     I5
```

I1–I5 to kolejne instrukcje (proste lub złożone) realizujące czynności programu.

Instrukcje warunkowe

Uwagi:

1. Wyrażenie warunkowe **musi** być zapisane w nawiasach okrągłych.
2. Słowo „`then`” nie występuje (w odróżnieniu od innych języków programowania).
3. C (w wersji ANSI) właściwie **nie zna** typu logicznego (w odróżnieniu od innych języków programowania).
4. Instrukcja **if** sprawdza numeryczną wartość wyrażenia; zamiast (*wyrazenie!=0*) piszemy (możemy pisać!) (*wyrazenie*).
5. Wyrażenie `(a > b)` ma wartość 1 gdy istotnie a jest większe od b i 0 w przeciwnym razie.

Zwracam uwagę, że wyrażenie podstawienia również może wystąpić w instrukcji warunkowej:

```
1 int a = 1;
2 int b = 2;
```

wer. 13 z drobnymi modyfikacjami!

```
3 if ( a = b )
4       printf("1: a = %d\n", a);
5 else
6       printf("2: a = %d\n", b);
```

Wykonanie powyższego fragmentu kodu będzie przebiegać w następujący sposób: w linii trzeciej programu zmiennej a zostanie przypisana wartość zmiennej b — w tym wypadku zmienna a przyjmie wartość 2. Równocześnie ta wartość (2) będzie „testowana” i, ponieważ, jest różna od zera — zostanie wykonana instrukcja o numerze 4. Gdyby, w instrukcji 2 zmiennej b była przypisana wartość równa zero, wówczas zmiennej a przypisano by wartość 0, czyli nieprawda, a następnie wykonana zostanie instrukcja w linii 6.

Wynik instrukcji **if** zależy od wartości zmennej b i nie zależy od (pierwotnej) zawartości zmiennej a.

Uwaga: Mimo, że powyższyt fragment skompiluje się, kompilator nie mając pewności jakie były intencje programisty zgłosi komunikat:

```
warning: suggest parentheses around
assignment used as truth value [-Wparentheses]
```

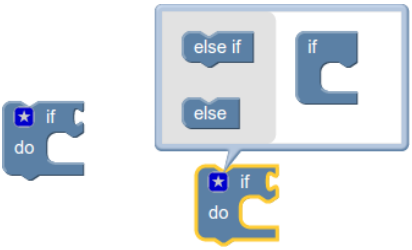
znaczący tyle, że jeżeli intencją programisty było użycie instrukcji podstawienia jako warunku to powinien on być zamknięty w dodatkowych nawiasach!

```
1 ...
2 if ( ( a = b) )
3       printf("1: a = %d\n", a);
4 else
5       printf("2: a = %d\n", b);
```

W języku blockly do konstruowania instrukcji warunkowej służy jeden blok (rys. 3.20), który po kliknięciu w gwiazdkę rozwija się pozwalając konstruować potrzebne rodzaje instrukcji (rys. 3.21 czy rys. 3.22). Wystarczy przeciągnąć albo kilka bloczków elseif i/lub bloczek else we wgłębienie „prototypu” instrukcji pokazane na dymku.

3.16. Wyrażenia warunkowe

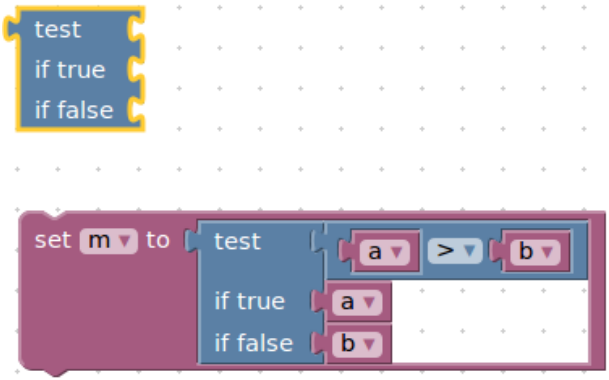
Oprócz instrukcji warunkowej język C zna również wyrażenia warunkowe. Jest ono bardzo wygodne, gdyż może być użyte po prawej stronie znaku równości.



Rysunek 3.20. Podstawowy blok if

Rysunek 3.21. Klasyczna instrukcja if-then-lelse w języku blockly

Rysunek 3.22. Najprostsza instrukcja if-then-else-if-else w języku blockly



Rysunek 3.23. Wyrażenie warunkowe w blockly

Wyrażenia warunkowe

Wyrażenie

```
1 if (a > b)
2     m = a;
3 else
4     m = b;
```

powoduje wstawienie do m większej z liczb a i b.

Powyższe może być zastąpione wyrażeniem:

```
1 m = (a > b) ? a : b;
```

Od niedawna blockly rownież zostało wyposażone w wyrażenia warunkowe (rys. 3.23).

Wyrażenia warunkowe mają różnorodne zastosowania. Jedno z ciekawszych:

```
1 *( ( a ) ? &b : &a ) = z;
```

Zakładając, że wszystkie zmienne są zadeklarowane i są odpowiednich typów, to co jest w zewnętrznych nawiasach może być interpretowane w sposób następujący: „Jeżeli a[27] to wartością tego wyrażenia będzie &b, w przeciwnym razie wartością będzie &a. Natomiast *(&a) lub *(&b) oznacza

[27] To znaczy „jeżeli a jest prawdą” albo „jeżeli a ma wartość różną od zera”.

pod adres zmiennej a lub pod adres zmiennej b wstaw to co jest po prawej stronie znaku równości (czyli z)[28].

Tu kolejne zastosowanie: wyrażenie warunkowe może być wykorzystane w naszym programie o kotach, w instrukcji printf, do drukowania przyrostków:

```
1     int z = 3;
2     printf("%s␣\n", (z == 0)? "zero": "nie␣zero");
```

w zależności od wartości zmiennej z drukowany będzie napis „zero” lub „nie zero”.

3.16.1. Dowcip

Jest taki dowcip (o programistach)
Żona programisty wysyła go do sklepu:
— Idź do sklepu i kup bochenek chleba, jak będą jajka — kup tuzin...
Wraca programista z dwunastoma bochenkami chleba...

Narysuj schemat blokowy działania programisty oraz schemat blokowy (prawdopodobnych) oczekiwań jego małżonki. Jakie są różnice?

A jak już jesteśmy przy schematach blokowych, to proszę rozważyć następujący fragment kodu:

```
1 if (U) if (V) if (W) A else B
```

Zakładamy, że U, V i W to pewne warunki logiczne lub wyrażenia, których wartość będziemy traktowali jako „zmienne logiczne”. Natomiast A i B to jakieś polecenia języka C. Dla uproszczenia można przyjąć, że A to printf("A\n"); natomiast B to printf("B\n");.

Zadanie polega na tym, żeby narysować schemat blokowy, a następnie sprawdzić jego prawdziwość prostym programem, który wygeneruje wszystkie możliwe kombinacje wartości U, V i W[29] i sprawdzi dla nich wyniki działania zaprezentowanego kodu.

Jak już to będzie gotowe — można narysować (i sprawdzić) schemat blokowy następującego kodu:

[28] Zrozumienie tego dowcipu wymaga jednak zapoznania się ze wskaźnikami i adresowaniem pośrednim.

[29] Pamiętamy, że wartość zerowa to fałsz, a wartość różna od zera to prawda. Ponieważ są trzy zmienne, każda może przyjmować dwie wartości — wszystkich kombinacji jest $8 = 2^3$.

```
1 if (U) if (V) {if (W) A} else B
```

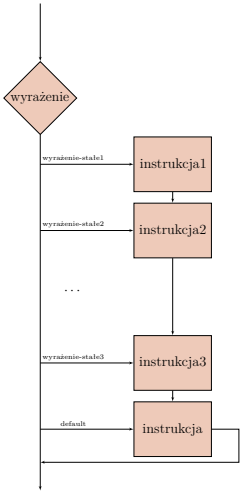
3.17. Rozgałęzienia — instrukcja switch

```
1 switch (wyrażenie){
2    case wyrazenie-stale1: instrukcje1
3    case wyrazenie-stale2: instrukcje2
4    ...
5    case wyrazenie-stale3: instrukcje3
6    default: instrukcje
7 }
```

1. Instrukcja **switch** służy do podejmowania decyzji wielowariantowych.
2. Najpierw wyiczana jest wartość wyrażenia.
3. Następnie sprawdza się czy wartość wyrażenia pasuje do jednej ze wskazanych stałych wartości (wyrażenie-stale*i*).
4. W przypadku zgodnośc z *i*-tym wyrażeniem — wykonywana jest insrukcja*i* i wszystkie występujące po niej polecenia.
5. Wszystkie *wyrażenia-stałe* muszą być różne.
6. Przypadek **default** zostanie wykonany gdy *wyrażenie* nie jest zgodne z żadnym przypadkiem.
7. **default** nie jest obowiązkowy: jeżeli nie występuje, a wyrażenie nie jest zgodne z żadnym przypadkiem — nie podejmuje się żadnej akcji.
8. Klauzula **default** może wystąpić na dowolnym miejscu.

Polecenie **switch** nie występuje we wszystkich językach programowania. W języku C znajduje zastosowane do organizacji rozgałęzień programu sterowanych przez jeden parametr. Polecenie często bywa używane do obsługi komunikacji z użytkownikiem:

1. Zadajemy pytanie (na przykład „Czy kontynuować?”).
2. Czytamy odpowiedź (do zmiennej **char** odp (interesuje nas tylko pierwszy jej znak: „T” lub „t” to odpowiedź twierdząca, „N” lub „n” — przecząca.
3. Używamy polecenia **switch** do zapisania decyzji użytkownika (zmienna kont przyjmie wartość 1 gdy użytkownik chce kontynuacji, 0 gdy nie che i −1 gdy popełni błąd podczas komunikacji z programem):



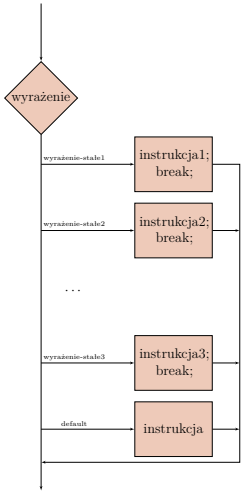
Rysunek 3.24. Schemat bokowy: instrukcja **switch** — wersja bez **break**

```
1  switch (odp)
2    {
3    case 'N':;
4    case 'n':
5      kont = 1;
6      break;
7    case 'T':;
8    case 't':
9      kont = 0;
10     break;
11   default:
12     kont = -1;
13     printf ("Zła odpowiedź!");
14   }
```

case

— Jeżeli nie podoba nam się przedstawione działanie (po Instrukcji 1 Wykonywana jest Instrukcja 2 i tak dalej)...
— Powinniśmy dodać instrukcję **break**!

wer. 13 z drobnymi modyfikacjami!



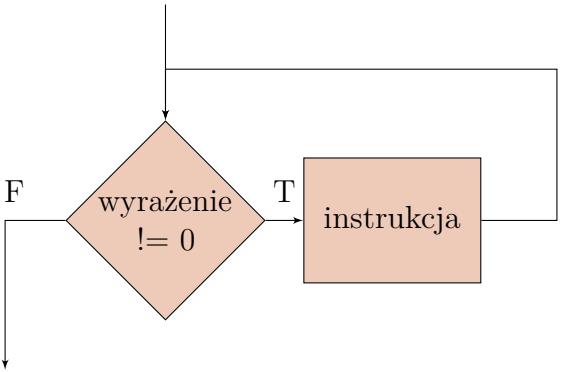
Rysunek 3.25. Schemat bokowy: instrukcja **switch** — wersja z **break**

— Wówczas schemat blokowy będzie nieco inny.

switch — przykład

```
1  char keystroke = getch();
2  switch( keystroke ) {
3      case 'a':
4      case 'b':
5      case 'c':
6      case 'd':
7          KeyABCDPressed();
8          break;
9      case 'e':
10         KeyEPressed();
11         break;
12     default:
13         UnknownKeyPressed();
14         break;
15 }
```

1. Zwracam uwagę na instrukcje **break** po rozpatrzeniu każdego przypadku!



Rysunek 3.26. Pętla **while** — schemat blokowy

2. Gdy nie zostanie ona umieszczone — po rozpatrzeniu jednego z przypadków wykonywane będą kolejne instrukcje (z kolejnych przypadków).
3. Napis **case** (aż do dwukropka) może być traktowany jako etykieta; nie ogranicza wykonywania poleceń.

   W języku blockly nic takiego nie występuje.

3.18. Pętle

3.18.1. Pętla while

```
1 while (wyrażenie)
2     instrukcja
```

1. Najpierw oblicza się wyrażenie.
2. Jeżeli jego wartość jest **różna od zera** wykonuje się instrukcję.
3. Ten cykl powtarza się do chwili, w której wartość wyrażenia stanie się zerem.
4. Gdy tak się stanie sterowanie przekazywane jest do instrukcji następującej po pętli.

Uwaga! Aby pętla się skończyła **coś musi spowodować** zmianę wartości wyrażenia.

wer. 13 z drobnymi modyfikacjami!

Zwracam uwagę, że jest pewna różnica pomiędzy poleceniem **if**, a poleceniem **while**. Mimo pozornego podobieństwa (wszędzie występuje warunek), polecenie:

```
1 if (W)
2   {
3     I;
4   }
```

sprawdza czy warunek W jest spełniony, i jeżeli tak — wykonuje **jednokrotnie** instrukcję I. Natomiast polecenie:

```
1 while (W)
2   {
3     I;
4   }
```

jeżeli warunek W jest spełniony (jest prawdziwy) polecenie I wykonuje wielokrotnie w pętli (tak długo jak ten warunek jest spełniony)!

3.18.2. Pętla for

Pętla

```
1 for(wyr1; wyr2; wyr3)
2     instrukcja
```

jest równoważna rozwinięciu:

```
1 wyr1;
2 while (wyr2){
3     instrukcja
4     wyr3;
5 }
```

1. Wszystkie trzy składniki instrukcji for są wyrażeniami.
2. Najczęściej wyr1 i wyr3 są przypisaniami lub wywołaniami funkcji
3. wyr2 to wyrażenie warunkowe.
4. Każdy ze składników można pominąć — wówczas znika on też z rozwinięcia. Średnik pozostaje!

Pętla for

Przykłady

```
1 for(i = 1; i < n; i++)
2     printf("%d\n", i);
```

```
1 i = 1;
2 while ( i < n ) {
3     printf("%d\n", i);
4     i++;
5 }
```

```
1 for( ; ; )
2     printf("%d\n", i);
```

Zatrzymajmy się chwilkę nad ostatnim przykładem. Jest to jak najbardziej legalna postać instrukcji **for**. Dosyć trudno ją „przetłumaczyć" do wersji **while** gdyż

```
1 while ( )
2     printf("Hello␣world\n");
```

jest błędne. Natomiast pętla **for** z pustymi parametrami jest równoważna następującemu kodowi:

```
1 while ( 1 )
2     printf("Hello␣world\n");
```

czyli będzie wykonywać się w nieskończoność...

Popatrzmy teraz na coś takiego (zakładam, że wszystkie zmienne są zadeklarowane i są właściwego typu):

```
1     for ( ; y; printf("%d␣%d␣\n", x, y) )
2         y = x++ <= 5;
```

Tłumaczy się ona na:

```
1     for (/* nic */ ; y; printf("%d␣%d␣\n", x, y) )
2         y = x++ <= 5;
```

i dalej na

```
1 /*nic*/
2 while (y)
3 {
4     y = x++ <= 5;
5     printf("%d %d \n", x, y);
6 }
```

Pamiętajmy więc, że „drugie polecenie” (w naszym przypadku y) to nie jest górny zakres pętli! To jest warunek (logiczny) który musi być prawdą by pętla powtarzałą się... W naszym przypadku y wyliczane jest w skomplikowanym wyrażeniu y = x++ <= 5; które tłumaczy się tak:

1. Najpierw sprawdamy czy x <= 5 (x mniejsze lub równe pięć), Jeżeli tak, to wynikiem tej operacji logicznej jest warość 1, w przeciwnym razie 0.
2. I ta warość podstawiana jest do zmiennej y.
3. Następnie (bo to przyrostek) x jest zwiększane o 1.

Pętla **for** używana jest najczęściej wtedy, gdy trzeba wygenerować ciąg liczb rosnący lub malejący ze stałym krokiem. Oczywiście m ożliwe są i inne jej zastosowania, ale ta jest najprostsza i najbardziej naturalna

3.18.3. Pętla do—while

1. Konstrukcja używana stosunkowo najrzadziej:

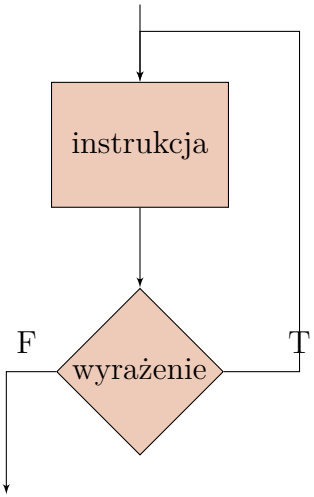
```
1 do
2     instrukcja
3 while (wyrażenie)
```

2. Najpierw wykonuje się instrukcję...
3. ...a następnie wyznacza wartość wyrażenia.
4. Pętla jest powtarzana gdy wyrażenie jest prawdziwe, czyli...
5. ...pętla zostanie zatrzymana gdy wyrażenie okaże się fałszywe.

Niektóre języki programowania nie mają takiej konstrukcji. Warto czasami zastanowić się, w jaki sposób zastąpić pętlą **while** pętlę **do—while**.

3.18.4. Instrukcje break i continue

Obie te instrukcjie mogą być używane **wyłacznie** wenątrz pęli **for**, **while** lub **do–while**. Instrukcja **break** może również pojawić się w instrukcji **switch**.



Rysunek 3.27. Pętla **do**—**while**: schemat blokowy

Instrukcja break

1. Polecenie **break** powoduje (kontrolowane) opuszczenie pętli przed jej **normalnym** zakończeniem.
2. Polecenie może być stosowane w przypadku pętli **while**, **for**, **do** oraz instrukcji **switch**; gdzie indziej jego użycie będzie błędem.
3. W przypadku zagnieżdżonych pętli wyskakujemy tylko jeden poziom wyżej.

```
1    while( x < 100 ) {
2      if( x < 0 )
3        break;
4      printf("␣%d␣\n", x);
5      x++;
6    }
```

W przypadku gdy x jest mniejsze od zera — nie realizujemy pętli.

Instrukcja continue

1. Instrukcja **continue** jest „spokrewniona” z instrukcją **break**.
2. Może być stosowana wyłącznie wewnątrz pętli!

3. Powoduje przerwanie przetwarzania bieżącego kroku pętli i przejście do kroku następnego.

```
for(i = 0; i < n; i++){
    if (a[i] < 0) /* pomiń element ujemny */
        continue;
    ...     /* przetwarzaj element nieujemny */
```

W przypadku gdy element tablicy jest ujemny — pomijamy przetwarzanie.

3.19. Skoki

Instrukcja skoku

1. Język C oferuje instrukcję skoku **goto** (pisane **bez** odstępu!) oraz etykiety pozwalające oznaczyć różne miejsca programu.
2. Instrukcja skoku formalnie **nie** jest potrzebna.
3. W praktyce, **prawie** zawsze można się bez niej obejść.
4. Idea programowania strukturalnego sugeruje, żeby z niej nie korzystać.
5. Czasami zdarzają się sytuacje (awaryjne!), gdzie zastosowanie instrukcji skoku może być bardzo przydatne.

Przykład:

```
...
if (warunek)
  goto error; /* Skok do obsługi błędow */
...
...
error:
  /* Jakis komunikat o bledzie lub próba
     naprawy sytuacji */
```

Natomiast etykieta to dowolny napis (skonstruowany zgodnie z zasadami tworzenia nazw zmiennych w języku C (bez odstępów, zaczyna się od litery, dospuszczalne litery, cyfry i znak podkreślenia) zakończony dwukropkiem. Może występować samotnie w linii, lub można umieścić po nim dowolne wyrażenie języka C. Na przykład:

```
1 petla:
2     goto petla;
```

Oczywiście jest to, tak zwana, pętla nieskończona...

3.20. Przykład algorytmu z użyciem instrukcji goto

Przez wiele lat trwała ożywiona dyskusja między informatykami na temat sensu używania instrukcji **goto** oraz złych nawyków które jej używanie powoduje. Wszystkie „stare" języki programowania zostały wyposażone w takie polecenie. Również język wewnętrzny wszystkich procesorów jest w taką instrukcję wyposażony. Kumulacja tych dyskusji przypadłą na lata siedemdziesiąte ubiegłego wieku.

Po wielu dyskusjach zaprojektowano języki programowania w których **nie ma** polecenia go to. Jednym z takich języków (w którym po raz pierwszy zabrakło mi tego polecenia był język programowania systemu DBase). Python nie ma takiej instrukcji. Występuje ona natomiast na liście słów kluczowych języka Java (z adnotacją „nieużywana"[30]).

Do grona przeciwników tej instrukcji zaliczał się Edsger Dijkstra natomiast inny teoretyk i praktyk informatyki, Donald Knuth optował za jej przydatnością.

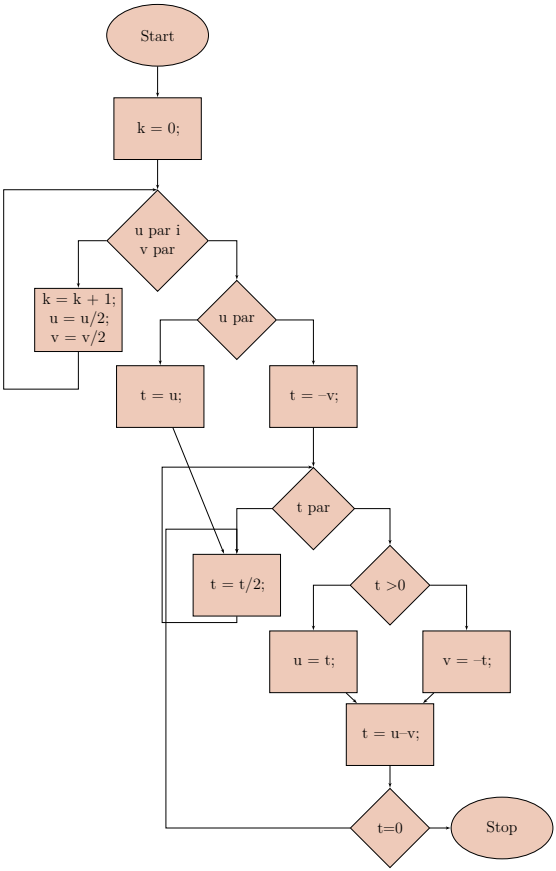
Dalszy porzykład zaczerpnięty został z pracy Knutha [12] i nazywa się „Binarnym algorytmem Euklidesa" albo w skrócie „Algorytmem B".

3.20.1. Algorytm B

Algorytm B

1. Przyjmij $k \leftarrow 0$, a następnie powtarzaj operacje: $k \leftarrow k+1$, $u \leftarrow u/2$, $v \leftarrow v/2$ zero lub więcej razy do chwili gdy przynajmniej jedna z liczb u i v przestanie być parzysta.
2. Jeśli u jest nieparzyste to przyjmij $t \leftarrow -v$ i przejdź do kroku 4. W przeciwnym razie przyjmij $t \leftarrow u$.
3. (W tym miejscu t jest parzyste i różne od zera). Przyjmij $t \leftarrow t/2$.
4. Jeśli t jest parzyste to przejdź do 3.

[30] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html.



Rysunek 3.28. Schemat blokowy Algorytmu B

5. Jeśli $t > 0$, to przyjmij $u \leftarrow t$, w przeciwnym razie przyjmij $v \leftarrow -t$.
6. Przyjmij $t \leftarrow u - v$. Jeśli $t \neq 0$ to wróć do kroku 3. W przeciwnym razie algorytm zatrzymuje się z wynikiem $u \cdot 2^k$.

Strzałka w lewo w przedstawionym zapisie oznacza podstaw.

Aby porządnie zaprogramować ten algorytm trzeba dobrze go zrozumieć. W związku z tym należy przyjąc następujące postępowanie:

1. Czytamy i staramy się zrozumieć algorytm.
2. Wybieramy dwie, niezbyt wielkie, liczby: jedna dwu- druga trzycyfrowa i algorytm realizujemy „na piechotę”, korzystając z kartki i ołówka.
3. Samodzielnie rysujemy schemat blokowy algorytmu.

4. Programujemy go „krok po kroku”, tak, żeby każdy krok był samodzielnym kawałkiem kodu. Wszędzie tam gdzie występuje „przejdź do...” używamy instrukcji **goto**. Taka realizacja algorytmu jest właściwie banalnie prosta.
5. Następnie staramy się wyeliminować[31] wszystkie instrukcje **goto** i zaprogramować go z użyciem wyłącznie instrukcji **while**, **do-while** oraz **if-then-else**.

Schemat blokowy algorytmu jest na rysunku 3.28.
To jest właściwie ostatnie miejsce gdzie pojawia się instrukcja **goto**.
W blokach decyzyjnych „u par”, „t par”,... oznacza zapytanie czy liczba jest parzysta

3.20.2. Kilka uwag o realizacji algorytmu

Algorytm nazywa się binarnym nie bez przyczyny. Wszystkie operacje bardzo łatwo zrealizować pamiętając, że liczby w komputerze zapisywane są w dwójkowym układzie liczenia.

1. Dzielenie przez dwa. Bardzo łatwo i co najważniejsze bardzo szybko realizuje się używając operacji przesunięcia w prawo jeden bit.

```
1 // Dzielnie przez dwa
2 t = t >> 1
```

2. Sprawdzenie czy liczba jest parzysta. Najprostszą metodą sprawdzenia czy liczba jest parzysta jest podzielenie jej przez dwa i zbadanie czy reszta wyszła zero.

```
1 // Sprawdzenie czy liczba parzysta
2 if (t % 2 == 0 )
```

Jednak można to zrealizować w inny sposób: Jak wiadomo, najmniej znacząca cyfra rozwinięcia dwójkowego dla liczb parzystych jest równa zero (w nieparzystych — 1)[32]. Zatem operacja `t & 1` „wycina” najmniej znaczący bit z liczby...

```
// Binarne sprawdzenie czy liczba parzysta
if ( ( t & 1 ) == 0 )
```

[31] Czytaj „eliminujemy”!
[32] Czemu???

3. Podnoszenie liczby dwa do potęgi, można zrealizować na wiele różnych sposobów. Najprostsze (i najmniej efektywne) będzie użycie funkcji **pow**. Można też użyć do tego pętli **for** lub **while**. Ale najprościej będzie zaprogramować to tak:

```
1 // Dwa do potęgi n
2 1 << n
```

   czyli uzywając przesunięcia w lewo o n bitów.
   Wytłumaczyć można to w nasyępujący sposób: liczba w zapisie dwójkowym to zestaw bitów (zer i/lub jedynek). Jej wartość wyliczana jest ze wzoru:

$$w = \sum_{i=0}^{n} c_i 2^i \tag{3.9}$$

   (najmniej znaczący bit to bit z prawej strony, c_i to i-ta cyfra). Liczba 1 ma $c_0 = 1$ i $c_i = 0, i = 1, 2, \ldots, n$. Przesunięcie tej jedynki w lewo o jedno miejsce — wygeneruje liczbę 2 (10 dwójkowo). Odpowiada to mnożeniu przez 10 liczb liczb dziesiętnych. Co więcej u $*$ 1 << k może być zapisane jako u << k (podobnie jak mnożenie liczby przez potęgę dziesiątki polega na dodawaniu zer po prawej stronie).
4. Odejmowanie i negacja — normalnie...

4. Preprocesor języka C: dyrektywy, makrodefinicje

Od kodu do programu wykonywalnego

1. Kod źródłowy
2. Preprocesor (gcc -E)
3. Kompilacja (konwersja do języka wewnętrznego) (gcc -S). Warto zerknąć (jak kto ciekawy na stronę godbolt.org
4. Assembler (tworzy pliki wynikowe) (gcc -c)
5. Konsolidowanie (przeglądanie bibliotek, budowa pliku wykonywalnego).
6. Uruchomienie
7. Debugowanie

4.1. Preprocesor

Jedną z pierwszych instrukcji w każdym (naszym[1]) programie jest dziwnie wyglądające polecenie **#include**<stdio.h>.

Ten wykład poświęcony będzie różnym poleceniom rozpoczynającym się od znaczku #[2]

Wszystkie polecenia rozpoczynające się od tego znaku przetwarzane są przez specjalny program zwany pre-procesorem.

Preprocesor

1. Pierwszym krokiem kompilacji jest przetwarzanie programu za pomocą **preprocesora**.

[1] Mam na myśli programy przygotowywane podczas zajęć laboratoryjnych.

[2] W zagonie informatyków nazywa się on „hasz" (ang. *hash*). Częściej określany jest jako *number sign* i „#1" czytany jest jako *number one*.

2. Najczęściej stosowanymi poleceniami preprocesora są:
   — **#include**
   — **#define**
   — polecenia kompilacji warunkowej (**#if**, **#ifdef**, **#ifndef**, **#endif**, **#else**, **#elif**)

4.2. Wstawianie plików

1. Każdy wiersz programu o postaci:
   — **#include** "*nazwa-pliku*"
   — **#include** <*nazwa-pliku*>
   jest zastępowany zawartością pliku o wskazanej nazwie.
2. Jeśli nazwa pliku ograniczona jest cudzysłowami — poszukiwanie pliku rozpoczyna się tam gdzie znaleziono plik źródłowy.
3. Jeśli nazwa pliku ograniczona jest nawiasami kątowymi <, > plik szukany jest wśród plików „systemowych” (w miejscach zależnych od implementacji).
4. Plik włączany poleceniem **#include** może zawierać kolejne polecenia **#include**.

Standardowe pliki nagłówkowe
ANSI C library header files

Ważne pliki nagłówkowe (globusiki są odsyłaczami do dokumentacji dostępnej w Internecie):

1. <assert.h> Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
2. <complex.h> A set of functions for manipulating complex numbers. (New with C99)
3. <ctype.h> Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
4. <errno.h> *For testing error codes reported by library functions.*
5. <fenv.h> For controlling floating-point environment. (New with C99)

6. <**float**.h> *Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).*
7. <inttypes.h> For precise conversion between integer types. (New with C99)
8. <iso646.h> For programming in ISO 646 variant character sets. (New with NA1)
9. <limits.h> *Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).*
10. <locale.h> *For setlocale() and related constants. This is used to choose an appropriate locale.*
11. <math.h> *For computing common mathematical functions*
12. <setjmp.h> Declares the macros setjmp and longjmp, which are used for non-local exits
13. <signal.h> *For controlling various exceptional conditions*
14. <stdarg.h> *For accessing a varying number of arguments passed to functions.*
15. <stdbool.h> For a boolean data type. (New with C99)
16. <stdint.h> For defining various integer types. (New with C99)
17. <stddef.h> *For defining several useful types and macros.*
18. <stdio.h> *Provides the core input and output capabilities of the C language. This file includes the venerable printf function.*
19. <stdlib.h> *For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.*
20. <string.h> *For manipulating several kinds of strings.*
21. <tgmath.h> For type-generic mathematical functions. (New with C99)
22. <time.h> *For converting between various time and date formats.*
23. <wchar.h> *For manipulating wide streams and several kinds of strings using wide characters — key to supporting a range of languages.* (New with NA1)
24. <wctype.h> *For classifying wide characters.* (New with NA1)

4.3. Makrorozwinięcia

1. Makrorozwinięcia to definicje które pozwalają duży fragment tekstu zastąpić krótkim:

```
1 #define nazwa zastepujacy_tekst
```

2. Każde użycie ciągu znaków `nazwa` zostanie zamienione przez *zastepujacy_tekst*.
3. Definicja może korzystać z poprzednich definicji.
4. Makrorozwinięcia nie obowiązują wewnątrz stałych tekstowych (ograniczonych znakami cudzysłowów).
5. Makrorozwinięcia stosuje się do całych jednostek leksykalnych. Zatem jeżeli zdefiniowane jest coś takiego:

```
1 #define YES "Tak"
```

   to każde wystąpienie YES zostanie zamienione na "Tak", ale nie będzie to dotyczyło

```
1 printf( "YES" );
```

   albo wystąpienia napisu „YESMAN”
   A co będzie w przypadku

```
1 printf( YES );
```

4.3.1. Rzeczywiste przykłady

Z pliku math.h

```
1 # define M_PI           3.14159265358979323846  /* pi */
2 # define M_PI_2         1.57079632679489661923  /* pi/2 */
```

Z pliku stdlib.h

```
1 #define EXIT_FAILURE    1 /* Failing exit status. */
2 #define EXIT_SUCCESS    0 /* Successful exit status. */
```

Z pliku stdin.h

```
1 #define EOF (-1)
```

4.3.2. Makrorozwinięcia z parametrami

1. Istnieje możliwość definiowania makr z argumentami — zastępujący tekst może być różny dla różnych wywołań:

```
1 #define max(A, B) ((A) > (B) ? (A) : (B))
```

   Bardzo przypomina wywołanie funkcji, ale realizowane będzie w sposób następujący:
   — Pojawienie się w tekście programu napisu

```
1 x = max(p+q, r+s);
```

   zostanie rozwinięte jako

```
1 x = ((p+q) > (r+s) ? (p+q) : (r+s))
```

   — Rozwiązanie takie ma wadę: wyrażenia obliczane są dwukrotnie, zatem w

```
1 z = max(i++, j++);
```

   wartości `i`, `j` zostaną zwiększone dwukrotnie!
2. W przypadku gdy makrorozwinięcia używane są do prowadzenia obliczeń, pamiętać należy o odpowiednim stosowaniu nawiasów.

```
1 #define square(x) x * x
```

   tłumaczone jest rozsądnie w najprostszym przypadku `y = square(v);` na

```
1 y = v * v;
```

   ale w przypadku `y = square(v + 1);` na

```
1 y = v + 1 * v + 1;
```

3. Poprawna definicja powinna wyglądać jakoś tak:

```
1 #define square(x) (x) * (x)
```

4. *Uwaga:* pomiędzy nazwą a nawiasem otwierającym nie może być odstępu!
5. Makra rozwijanie są przed kompilacją!

1. Jeżeli nazwę parametru w zastępującym tekście poprzedzimy znakiem # to cała kombinacja (parametr i znak) zostaną rozwinięte w ciąg znaków ograniczonych cudzysłowami.
2. Przyjmijmy, że mamy taką definicję:

```
1 #define drukuj(tekst) printf( #tekst )
```

   to wywołanie

```
1 drukuj(ala ma kota);
```

   jest rozwijane w

```
1 printf( "ala␣ma␣kota" );
```

3. Znaków ## można używać do sklejania parametrów formalnych. Makro

```
1 #define sklej(przod, tyl) przod ## tyl
```

   wywołane

```
1 sklej(a, 1)
```

   zostanie rozwinięte jako a1

Makrorozwinięcia

Prosty przykład

```
1 #ifdef DEBUG
2 # include <stdio.h>
3 # define wypisz(x) do { printf("Zmienna␣" # x \
4                              "␣=␣%d\n", x); } \
5    while (0);
6 #else
7 # define wypisz(x)
8 #endif
```

Użycie

```
1 #define DEBUG
2 ...
3 wypisz( a );
```

Odwoływanie makra

Jeżeli jakaś definicja przestaje być potrzebna można ją zlikwidować za pomocą polecenia

```
#undef
```

Na przykład tak:

```
#define DEBUG 1 /* Wydruki diagnostyczne */
/* ...          */
#ifdef DEBUG
/* ...          */
#endif
/* ...          */
#undef DEBUG
/* ...          */
```

4.4. Zmienne preprocesora

Wiele z tego zależy od implementacji, ale zazwyczaj zdefiniowane są następujące zmienne:

```
__LINE__
__FILE__
__DATE__
__TIME__
__cplusplus
__STDC__
```

1. __LINE__ i __FILE__ oznaczają, odpowiednio, numer linii źródłowej kompilowanego pliku i jego nazwę.
2. Zmienna __DATE__ zawiera datę — precyzyjniej datę kiedy dokonywana jest kompilacja.
3. Zmienna __TIME__ zawiera czas — precyzyjniej czas kiedy dokonywana jest kompilacja.
4. Zmienna __cplusplus zdefiniowana jest tylko wtedy, gdy kompilowany jest program w języku C++.
5. Zmienna __STDC__ zdefiniowana jest wtedy, gdy kompilowany jest program w języku C.

Inne przydatne stałe

— _WIN32 — Windows, wersja 32 bity,

— _WIN64 — Windows, wersja 64 bity,
— ___APPLE___ — Apple,
— ___linux___ — Linux,
— ___unix___ — inny Unix,
— ...

Pełniejsza lista definicji akceptowanych przez większość kompilatorów: https://sourceforge.net/p/predef/wiki/OperatingSystems/ a lista definicji rozpoznawanych przez kompilator gcc tu: http://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html.

4.4.1. Przykład

```
#ifdef DEBUG
# include <stdio.h>
# define D(x) do { printf("%s:%d (%s) ",\
                     __FILE__, __LINE__,\
                     __FUNCTION__); \
                     printf x ;\
                     fputc('\n', stdout); \
                     fflush(stdout); } \
                while (0);
#else
# define D(x)
#endif
```

Użycie makra D
D(("z= %d ", z))
Czemu tak?

4.5. Kompilacja warunkowa

Preprocesor został wyposażony w kilka poleceń pozwalających na sterowanie przetwarzaniem kodu źródłowego. Są to:

```
#if warunek
/*  ...             */
```

```
3 #endif
4
5 #if warunek
6 /* ...          */
7 #else
8 /* ...          */
9 #endif
```

```
1 #if warunek1
2 /* ...          */
3 #elif warunek2
4 /* ...          */
5 #elif warunek3
6 /* ...          */
7 #else
8 /* ...          */
9 #endif
```

Dodatkowo sprawdzać można czy **zostały** zdefiniowane makra poleceniami

```
1 #ifdef makro
2 /* ...          */
3 #endif
```

Oraz czy makro **nie zostało** zdefiniowane

```
1 #ifndef makro
2 /* ...          */
3 #endif
```

4.5.1. Po co?

1. Warunkowe włączanie kodu na potrzeby uruchomienia (DEBUG).
2. Tworzenie kodu przenośnego — w zależności od wersji systemu (kompilatora) standardowy zestaw makrodefinicji zawarty może być w plikach nagłówkowych.
3. „Zakomentowanie” dużego fragmentu kodu.

5. Funkcje

5.1. Funkcje

Mówiąc o funkcjach odwoływał się będę do intuicji matematycznej. Każdy (chyba) słyszał o pojęciach takich jak $\sin(x)$ czy $\log(x)$, czyli takich obiektach (matematycznych), do których po wrzuceniu jakiejś wartości argumentu, otrzymujemy wyliczoną według skomplikowanego algorytmu wartość funkcji.

Właściwie (nawet w matematyce) nikt nie wylicza wartości tych funkcji na podstawie ich definicji czy wzoru. Posługujemy się tylko pojęciem zakładając, że jak przyjdzie do wyliczania ich wartości — skorzystamy z jakichś pomocy (kalkulator, komputer, nomogram,...).

Dokładnie tak samo jest z językami programowania. Funkcja to **samodzielny**, zamknięty algorytm, który dla zadanej wartości argumentów wylicza jakąś wartość. Tyle tylko, że w bardzo wielu przypadkach musimy funkcję sami zaprogramować.

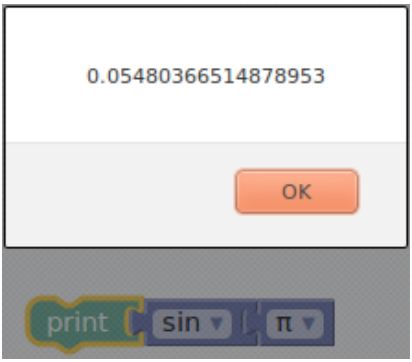
5.2. Przykład

Wyobraźmy sobie, że chcemy zaprogramować jakieś wyrażenie trygonometryczne w Blockly. Piszemy większy program, a uznaliśmy, że przetestowanie pojedynczych wyrażeń w Blockly będzie prostsze. Okazuje się jednak, że Blockly (w odróżnieniu od większości „normalnych" jerzyków programowania zakłada, że argument funkcji trygonometrycznych jest w stopniach, a nie radianach (por. rys. 5.1).

Zatem trzeba napisać jakąś formułę, która skonwertuje radiany na stopnie[1]. Formuła jest dosyć prosta:

$$y = x/\pi * 180. \tag{5.1}$$

[1] Podobny problem będziemy mieli, gdy równania, które wymagają podawania kątów w stopniach zechcemy zaprogramować w C (patrz rozdział 5.9).



Rysunek 5.1. Gdyby argument funkcji sinus był w radianach wynik powinien być równy 0 albo jakiejś bardzo małej wartości

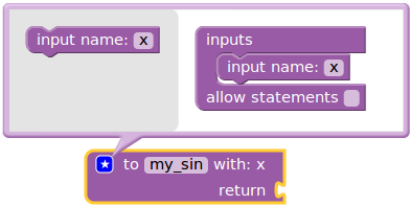
Rysunek 5.2. Formułą $x/\pi * 180$ zaprogramowana w Blockly

Formułą nie jest skomplikowana i została przedstawiona na rysunku 5.2.

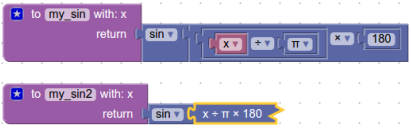
Aby stworzyć własną funkcję sięgamy do menu Funkcje i wybieramy obiekt, który „tworzy funkcję z wynikiem". Nazywamy go my_sin oraz dodajemy zmienną wejściową x i odhaczamy ptaszka w polu *allow statements* (co oznacza, że nasza funkcja będzie dosyć prosta, bez dodatkowych poleceń). Prezentuje to rysunek 5.3.

Działanie naszej funkcji my_sin(x) polegać będzie na tym, że dla każdej wartości x jako swoją wartość zwracać będzie sin($x/\pi * 180$) (patrz rys. 5.4).

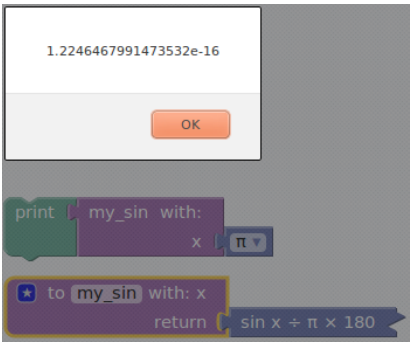
Gotowy program przedstawia rysunek 5.5. Udało się stworzyć narzędzie,



Rysunek 5.3. Tworzenie prostej funkcji w Blockly



Rysunek 5.4. Gotowa funkcja my_sin(). Aby trochę skompresować „skomplikowane wyrażenie arytmetyczne" użyłem polecenia Collapse z menu podręcznego (po kliknięciu prawym klawiszem myszy na wyrażeniu)



Rysunek 5.5. Gotowy program

zastępujące bloczek liczący funkcję sin (gdzie argumentem są stopnie) na własny bloczek podający wynik po odpowiednim przekształceniu).

Zaglądając do zakładki JavaScript, Dart albo Python łatwo przekonać się, że standardowa funkcja sinus zwraca wartości dla argumentu podanego w radianach...

Funkcje

1. Funkcje to sposób na podzielenie dużego programu na mniejsze, łatwiejsze w zarządzaniu fragmenty.
2. Odpowiedni (umiejętny) podział programu na moduły (funkcje) pozwala na powtórne (i wielokrotne) wykorzystanie ich w innych programach.
3. „Ukrycie" pewnych fragmentów pod postacią funkcji pozwala na uproszczenie struktury programu i uczynienie jej bardziej czytelną.
4. *Funkcje to, wreszcie, podstawa programowania strukturalnego.*
5. Praktycznie każdy język programowania wyposażony jest w mechanizmy podziału na moduły oraz tworzenia funkcji (i procedur).

6. W matematyce pod pojęciem funkcji rozumiemy twór, który pobiera pewną liczbę argumentów i zwraca wynik. Jeśli dla przykładu weźmiemy funkcję $\sin(x)$ to x będzie zmienną rzeczywistą, która określa kąt, a w rezultacie otrzymamy inną liczbę rzeczywistą — sinus tego kąta.
7. Warto wiedzić, że pojęcie funkcji zostało do programowania przeniesione z matematyki.

5.3. Budowa funkcji

W języku C każdy obiekt, którego chcemy używać musi zostać „zadeklarowany", to znaczy należy zdefiniować jego:

1. typ przechowywanych/zwracanych wartości (całkowity, zmiennoprzecinkowy, znakowy,...)[2],
2. nazwę,
3. rodzaj (zmienna prosta, zmienna złożona, funkcja,...). Tu pojawiają się dodatkowe argumenty.

Budowa funkcji

Definicja funkcji wygląda w sposób następujący

```
1 typ_powrotu nazwa_funkcji ( deklaracja parametrów )
2 {
3    deklaracje i instrukcje
4 }
```

Zatem „typ_powrotu" jest bardzo istotną informacją, która musi być uwzględniona podczas prowadzenia obliczeń na wartościach funkcji.

Gdy typ powrotu funkcji nie zostanie zadeklarowany, zakłada się, że funkcja zwraca wartości całkowite.

1. Funkcja **musi** być *zadeklarowana* przed pierwszym jej użyciem!

[2] Przypominam, że każda wartość w języku C musi mieć zadeklarowany **typ**. Czymś zupełnie innym jest wartość dwa zapisana jako typ **char** (na ośmiu bitach), **int** (32 bity) czy **double** (64 bity typu float). Warość może i jest taka sama, ale reprezentacja bitowa za każdym razem zupełnie inna.

2. Funkcja zwraca wartość będącą wynikiem jej działania. **Typ** zwracanej wartości zdefinowany jest podczas deklaracji funkcji i oznaczony tu jako **typ powrotu**.
3. Funkcję wywołuje się najczęściej w następujący sposób:

```
a = nazwa_funkcji( parametry funkcji );
```

zwłaszcza gdy zależy nam na zapamiętaniu, lub dalszym przetwarzaniu, wyniku zwracanego przez funkcję. Gdy nie jest on potrzebny (istotny) lub funkcja nie zwraca żadnych wyników można wykonać tak:

```
nazwa_funkcji( parametry funkcji );
```

(W ten sposób najczęściej wywoływana jest funkcja printf)
4. Jeżeli funkcja zwraca jakąś wartość wśród jej instrukcji powinno znaleźć się polecenie

```
return wyrażenie;
```

powoduje ona, że wartość wyrażenia przypisywana jest jako wartość funkcji!

Zakładam, że zdziwienie może budzić fakt, że printf jest funkcją (a nie poleceniem języka C) oraz, że funkcja ta zwraca jakiś wynik. Przetestujmy to:

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int i;
        i=printf("a");
        printf(" i= %d\n", i);
        i=printf("ab");
        printf(" i= %d\n", i);
        i=printf("abc");
        printf(" i= %d\n", i);
        i=printf("abcd");
        printf(" i= %d\n", i);
        return 0;
}
```

Wynik działania tego programu jest następujący:

```
a i= 1
ab i= 2
abc i= 3
abcd i= 4
```

Budowa funkcji

Najprostsza funkcja

```
dummy ()
{
}
```

— Funkcja nie ma parametrów.
— Funkcja nie zwraca żadnej wartości.
— Funkcja „nic nie robi”

Użycie:

```
        dummy();
```

Program z funkcją

```
void dummy(void)
{}

int main()
{
    dummy();
    return ( 0 );
}
```

Program z funkcją

```
void dummy(void)
{
    glupia();
}
```

```
5
6  void glupia(void)
7  {}
8
9  int main()
10 {
11     dummy();
12     return 0;
13 }
```

Źle !!!

Powyższe jest niedobre, ponieważ funkcja musi być zadeklarowana (jako prototyp) lub zdefiniowana przed pierwszym jej użyciem w programie.

Funkcja dummy wywołuję funkcję glupia. Ponieważ kod analizowany jest od góry do dołu — to w tym miejscu kodu (linia 3), w którym wywoływana jest funkcja glupia nie jest ona jeszcze zdefiniowana! Zdefiniowana została w linii 6.

Program z funkcją

```
1  void glupia(void)
2  {}
3
4  void dummy(void)
5  {
6      glupia();
7  }
8
9  int main()
10 {
11     dummy();
12     return   0;
13 }
```

OK !!!

5.3.1. Funkcje zagnieżdżone

Każda funkcja **musi** być zadeklarowana przed pierwszym użyciem. W związku z tym, na ogół, deklaracje i algorytmy funkcji muszą być umieszczone przed funkcją main. Pojawia się pytanie, czy taka konstrukcja jest możliwa:

Funkcje zagnieżdżone

```
1  int main( void )
2  {
3      void dummy( void )
4      {
5          void glupia( void ) { }
6          glupia();
7      }
8      dummy();
9      return 0;
10 }
```

Źle !!!

Standard języka C na to nie pozwala!

Konstrukcja taka (funkcje zagnieżdżone, nested functions) występująca w wielu językach, **nie jest dopuszczona** — jako część standardu — w języku C. Niestety kompilator gcc dopuszcza ją.

Uważam, że konstrukcja taka nie powinna być stosowana (nawet, jeżeli kompilator na to pozwala).

5.4. Argumenty funkcji

Argumenty funkcji

1. Argumenty funkcji służą do przekazania informacji z zewnątrz do jej wnętrza.
2. Według standardu ANSI C typ argumentów musi być zadeklarowany. W definicji funkcji zapisuje się to tak:

```
1 typ identyfikator (typ1 arg1, typ2 arg2, typn argn)
2 {
3   /* instrukcje */
4 }
```

Na przykład:

```
1 int iloczyn ( int x, int y )
2 {
```

```
3   int iloczyn_xy;
4   iloczyn_xy = x * y;
5   return iloczyn_xy;
6 }
```

```
1 int iloczyn ( int x, int y )
2 {
3   return x * y;
4 }
```

3. Argumenty funkcji użyte podczas wywołania funkcji są kopiowane do odpowiednich zmiennych zadeklarowanych w definicji funkcji. Oznacza to, że jakiekolwiek modyfikacje tych argumentów nie mają wpływu na wartości zmiennych (czy wyrażeń) w wywołaniu funkcji. Mówi się, że argumenty są przekazywane przez wartość, czyli wewnątrz funkcji operujemy tylko na ich kopiach.

Argumenty funkcji

1. Funkcja nie musi mieć argumentów.

```
1 int smieszna()
2 {
3    return 7;
4 }
```

```
1 int smieszna(void)
2 {
3    return 7;
4 }
```

2. W takim wypadku wywołanie funkcji ma postać:

```
1 a = smieszna();
```

Nawiasy muszą być nawet jak nie ma argumentów!
2023-04-16 13:56:24 +0200

5.5. Wynik wykonania funkcji

1. Funkcja (na ogół) zwraca jakieś wyniki.
2. Do przekazania wyników na zewnątrz funkcji służy instrukcja **return**.
3. Program wywołujący może zignorować zwrócone wyniki.
4. Gdy funkcja nie zwraca wyników nazywana bywa procedurą.

„Procedury”

Procedury, to specjalny rodzaj funkcji, która nie zwraca żadnej wartości i nie może być użyta w żadnym wyrażeniu matematycznym.

1. Procedurę deklaruje się w następujący sposób:

```
1 void procedurka( int x )
2 {
3     printf("-------\n"\
4             "%d\n"\
5             "-------\n",\
6             x);
7 }
```

2. Procedurę wywołuje się w następujący sposób:

```
1 int main()
2 {
3     int z = 123;
4     procedurka( z + 7);
5     return 1;
6 }
```

Kompletny program będzie wyglądał tak:

```
1 #include<stdio.h>
2
3 void procedurka( int x )
4 {
5     printf("-------\n"\
6             "%d\n"\
7             "-------\n",\
8             x);
```

wer. 16 z drobnymi modyfikacjami!

```
9 }
10
11 int main()
12 {
13     int z = 123;
14     procedurka( z + 7);
15     return 1;
16 }
```

„Procedury” — kilka uwag

1. Każda procedura (jak i funkcja) powinna być zadeklarowana przed pierwszym jej użyciem.
2. Deklaracja to **definicja** albo **prototyp** (a definicja później).
3. Bardzo wiele procedur systemowych deklarowanych jest w plikach nagłówkowych (o rozszerzeniu **.h**).
4. Pliki nagłówkowe powinny być wczytywane na początku.
5. Ogólna struktura programu powinna być zatem taka:
   — wczytanie plików nagłówkowych,
   — definicje wszystkich procedur,
   — program główny.

Deklaracja funkcji (prototyp) wygląda (jakoś) tak: **typ** nazwa (parametry i ich typ);

Program z funkcją i prototypy

```
1 void glupia(void);
2 void dummy(void);
3
4 int main()
5 {
6     dummy();
7     return ( 0 );
8 }
9
10 void dummy(void)
11 {
12     glupia();
```

```
13 }
14
15 void glupia(void)
16 {}
```

Teraz kolejność nie jest już istotna.

5.6. Definicje i deklaracje globalne

1. Każda zmienna musi być zadeklarowana.
2. Zmienna dostępna jest tylko w bloku, w którym została zadeklarowana (i wszystkich blokach w nim zawartych. Są to zmienne lokalne.
3. *Uwaga:* blok to zazwyczaj wszystko co się znajduje wewnątrz nawiasów klamrowych **{ }**
4. Deklaracje w blokach niższych „przysłaniają" deklaracje z bloków wyższego poziomu.
5. Po wyjściu z bloku zmienne lokalne „znikają". Są niedostępne, a ich zawartość jest zapominana.
6. Po powrocie do bloku **nie ma dostępu** do poprzedniej wartości zmiennej!
7. Po powrocie do funkcji (w zasadzie) nie ma dostępu do poprzednich wartości zmiennych.

Definicje i deklaracje globalne

1. Zmienne zadeklarowane na zewnątrz wszystkich modułów (funkcje, procedury, funkcja **main**) nazywane są zmiennymi globalnymi.
2. Zmienne globalne dostępne są we wszystkich blokach...
3. ...chyba, że zostaną przysłonięte przez definicją lokalną.
4. Zmienne globalne mogą być wykorzystanie do przekazywania dodatkowych wyników zwracanych przez funkcję. Nie jest to najlepsze rozwiązanie...

```
1 #include<stdio.h>
2 int v = 100;
3 void procedurka( int x )
4 {
5     int v = 7;
6     printf("———————\n"\
```

wer. 16 z drobnymi modyfikacjami!

```
7                 "%d\n"\
8                 "-------\n",\
9                 x);
10      printf("v = %d\n", v);
11 }
12 int main()
13 {
14      int z = 123;
15      procedurka( z + 7);
16      procedurka( v );
17      return 1;
18 }
```

Czym się różni?

```
1 int i;
2 int main(void)
3 {
4    return 1;
5 }
```

```
1 int main(void)
2 {
3    int i;
4    return 1;
5 }
```

5.7. Funkcja main

1. Każdy program w języku C musi mieć segment główny.
2. Segment główny musi nazywać się **main**...
3. ...i jest funkcją!
4. Wartość, którą zwraca funkcja main przekazywana jest do systemu operacyjnego.
5. Wartość ta zazwyczaj informuje czy program zakończył się z błędami i, czasami, o typie (rodzaju) błędu.

6. Standardowe kody zakończenia programu zdefiniowane są w pliku nagłówkowym **stdlib.h** są to

```
1 #define     EXIT_FAILURE    1     /* Failing exit status
2 #define     EXIT_SUCCESS    0     /* Successful exit sta
```

7. Każdy segment główny powinien się kończyć poleceniem **return**.

Funkcja main

1. To jest właściwie poprawny program w języku C

```
1 void main(void)
2 {
3     ;
4 }
```

   Nic nie robi, nie zwraca żadnej informacji. Kompilator sygnalizuje komunikat „warning: return type of ‘main’ is not ‘int”’
2. Zamiana pierwszego **void** na **int**

```
1 int main(void)
2 {
3     ;
4 }
```

   powoduje komunikat „warning: control reaches end of non-void function” (czyli brakuje polecenia **return**).
3. Poprawny (minimalny) program powinien wyglądać jakoś tak:

```
1 int main(void)
2 {
3     return 0;
4 }
```

5.8. Argumenty funkcji main

1. Skoro main jest funkcją — powstaje zatem pytanie jak przekazać mu (jej?) jakieś parametry.
2. Właściwie jest na to miejsce:

```
1 int main(int x)
2 {
3     printf("x=%d\n", x);
4     return 0;
5 }
```

Ale taki zapis generuje komunikat błędu „warning: ‘main’ takes only zero or two arguments”...

3. Druga sprawa to jak (wykorzystując System Operacyjny) przekazać do programu jakieś parametry (zwłaszcza podczas klikania myszą)?
4. Mimo wszystko uruchomimy go. Najpierw tak:

```
$ ./kody
x=1
```

Później tak:

```
$ ./kody a
x=2
```

Hmmm... Jest jakaś prawidłowość? Patrzymy dalej:

```
$ ./kody a
x=2
$ ./kody a b
x=3
$ ./kody a b c
x=4
$ ./kody a b c d
x=5
```

5. Autorzy języka C (i systemu Unix) obmyślili to tak:
   — Po wpisaniu nazwy uruchamianego programu tworzona jest struktura danych zawierająca kopię linii polecenia oraz informację o liczbie parametrów (a tak na prawdę liczbie „wyrazów” oddzielonych odstępami).
   — Liczba parametrów przekazywana jest jako pierwszy argument funkcji main.
   — Drugim argumentem jest tablica znakowa zawierające w kolejnych komórkach kolejne argumenty.

```
1 #include<stdio.h>
2
3 int main (int cnt, char ** arg)
4 {
5     int i;
6     printf("Liczba argumentow = %d\n", cnt);
7     for (i = 0;i < cnt; i++)
8      printf(" argument %d = %s\n", i, arg[i]);
9     return 0;
10 }
```

```
$ ./kody1 ala ma kota
Liczba argumentow = 4
 argument 0 = ./kody1
 argument 1 = ala
 argument 2 = ma
 argument 3 = kota

$ ./kody1 "ala ma kota"
Liczba argumentow = 2
 argument 0 = ./kody1
 argument 1 = ala ma kota
```

5.9. Prosty przykład

Zacznijmy od bardzo prostego przykładu. Chcemy stworzyć odpowiednik systemowej funkcji sin, która będzie podawała wartość funkcji dla argumentu w stopniach. Zakładam, że funkcja będzie nazywała się my_sin. Zacznijmy od „algorytmu”, który w tym przypadku jest bardzo prosty:

$$\mathsf{my\_sin}(x) = \sin(x/180 * \pi) \tag{5.2}$$

Kolejny etap, do określenie typu wartości przyjmowanych jako argument i jako wynik funkcji. Jest oczywiste, że muszą to być wartości typu **double**.

```
1 double my_sin(double x)
2 {
3         double temp;
```

wer. 16 z drobnymi modyfikacjami!

```
4         temp = x * 180 / 3.14;
5         temp = sin(temp);
6         return temp;
7 }
```

Linia 1 to początek „deklaracji" funkcji. Muszą w nim wystąpić typy wartości (**double** sin i argumentu(ów): **double** x. Gdy je pominiemy (albo pominiemy tylko jeden z nich) kompilator automatycznie przyjmie, że typ funkcji/argumentu jest **int**. To znaczy:

```
1 my_sin (x)
2 {
3 ...
4 }
```

jest najzupełniej poprawną definicją funkcji my_sin ale argument (x) jest typu **int** i wartość zwracana przez funkcję jest typu **int**. Zatem powyższy zapis jest równoważny:

```
1 int my_sin (int x)
2 {
3 ...
4 }
```

Kompilując fragment kodu:

```
1     y = my_sin(185.)
```

kompilator dokładnie sprawdza typ argumentu wstawionego w funkcji (185. jest typu double) i typ zadeklarowanego i gdy zajdzie taka potrzeba wstawia kod dokonujący odpowiedniej konwersji typu (rzutowania) argumentu, wyniku funkcji lub obu tych wartości. Zatem najlepiej od razu, świadomie deklarować wszystkie typy poprawnie, a stałe (180) od razu wpisywać jako stałe odpowiedniego typu... Warto również wpisać stałą π z większą dokładnością. Plik nagłówkowy math.h zawiera stałą M_PI zapisaną z największą możliwą dokładnością. Warto z niej korzystać. Po zmianach program będzie wyglądał tak:

```
1 double my_sin(double x)
2 {
3         double temp;
4         temp = x * 180. / M_PI;
```

```
5         temp = sin(temp);
6         return temp;
7 }
```

Jak się zastanowić to zmienna pomocnicza temp nie jest potrzebna i program można zapisać w zwartej postaci:

```
1 double my_sin(double x)
2 {
3         return sin( x * 180 / M_PI );
4 }
```

5.10. Rekurencja

1. Przypadek gdy funkcja (lub procedura) wywołuje samą siebie nazywamy rekurencją.
2. Nie potrafię powiedzieć, czy rekurencja to dobra czy zła technika programowania.
3. Rekurencja była bardzo pożyteczna podczas tworzenia algorytmów.
4. W realizacjach programowych (zwłaszcza bardzo skomplikowanych problemów) stwarza wiele kłopotów.
5. Problemy wynikają z konieczności przechowania wszystkich argumentów i całej struktury danych używanej przez funkcję gdy wywołuje ona samą siebie.

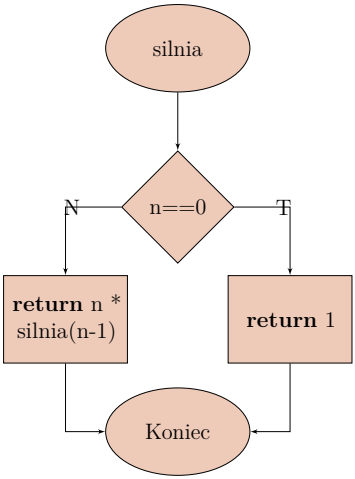
5.10.1. Silnia

Schemat blokowy rekurencyjnej funkcji wyliczającej silnię przedstawia rysunek 5.6.

Program w C funkcji wyliczającej silnię.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float silnia(int n)
5 {
6     if (n == 0)
7     return 1.;
```

wer. 16 z drobnymi modyfikacjami!



Rysunek 5.6. Schemat blokowy rekurencyjnej wersji funkcji silnia

```
8       else
9       return n * silnia(n-1);
10  }
11  int main(int cnt, char ** arg)
12  {
13      int n;
14      n=atol( arg[1] );
15      printf("%d! = %g\n", n, silnia(n));
16      return 0;
17  }
```

Zwracam uwagę, że opisany wcześniej mechanizm przekazywania informacji do programu z linii polecenia wykorzystałem do przekazania wartości liczby z której chcemy wyliczyć silnię.

W dalszej części, nieco bardziej realistyczna wersja programu. Ma mniejszy zakres danych wejściowych, ale liczy więcej cyfr wyniku.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 unsigned long long int silnia(int n)
4 {
5       if ( n == 0 )
6           return 1.;
```

```
7       else
8           return n * silnia(n - 1);
9   }
10
11  int main(int cnt, char ** arg)
12  {
13      int n;
14      n = atol( arg[1] );
15      printf( "%d! = %Lu\n", n, silnia(n) );
16      return 0;
17  }
```

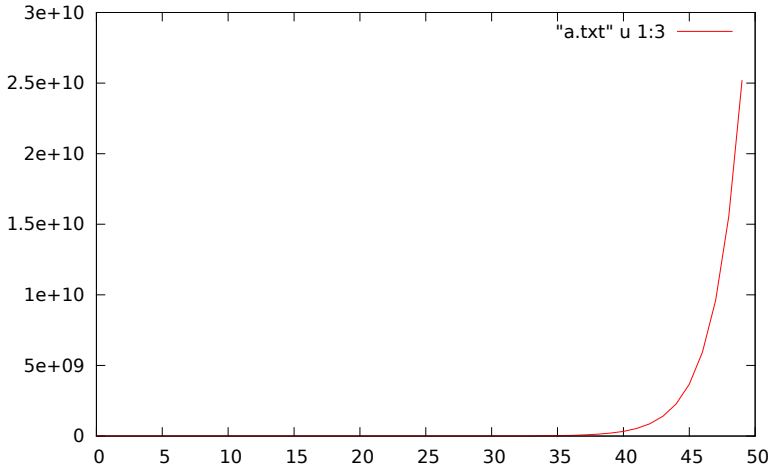
Ciąg Fibonacciego

$$F_n := \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ F_{n-1} + F_{n-2} & \text{dla } n > 1. \end{cases}$$

W dalszej części implementacja tego algorytmu. Wykorzystałem tu celowo zmienną globalną aby wyliczać liczbę wywołań funkcji fib. Jest to sensowny przykład wykorzystania zmiennej globalnej, ale podobny efekt może być uzyskany też inaczej. Jak?

```
1  #include <stdio.h>
2  unsigned long int k;
3  unsigned long int fib(int n)
4  {
5      k++;
6      if ( n == 0 )
7          return 0;
8      else if ( n == 1 )
9          return 1;
10     else
11         return fib(n - 1) + fib(n - 2);
12 }
13
14 int main(int argc, char **argv)
15 {
```

wer. 16 z drobnymi modyfikacjami!



Rysunek 5.7. Liczba wywołań funkcji fib(n) w zależności od n.

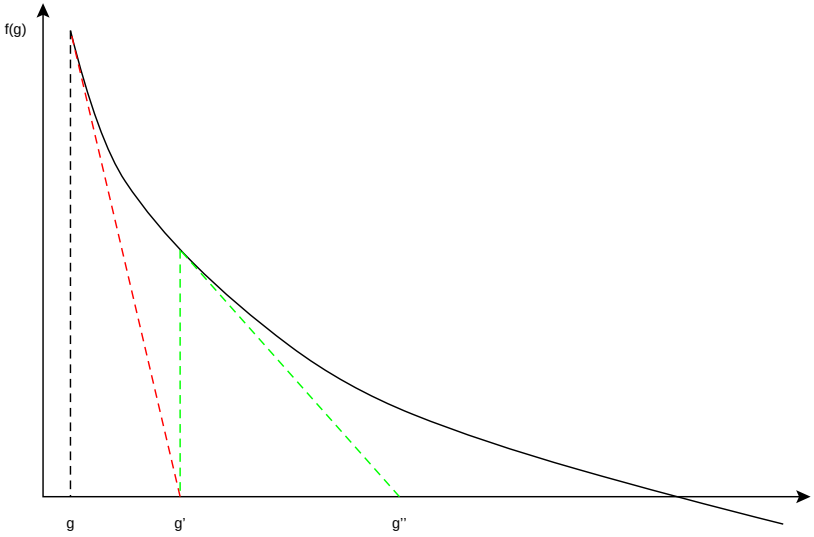
```
16      int n, m;
17      for ( n = 0; n < 100; n++ )
18      {
19          k = 0;
20          m = fib(n);
21          printf("%lu , %lu , %lu\n", n, m, k);
22      }
23      return 0;
24  }
```

Liczba wywołań funkcji fib (podobnie jak i czas wykonania całego programu) rośnie bardzo szybko wraz ze wzrostem parametru funkcji n. Ilustruje to wykres na rysunku 5.7.

5.11. Programowanie strukturalne

5.11.1. Metoda Newtona-Raphsona

Idea programowania strukturalnego
Metoda Newtona-Raphsona: pierwiastek dowolnego stopnia
2023-04-16 13:56:24 +0200



Rysunek 5.8. Pierwsze kroki działania procedury Newtona-Raphsona

Załóżmy, że mamy wyznaczyć pierwiastek stopnia n z liczby w, czyli znaleźć taką liczbę x, że:

$$x^n = w \tag{5.3}$$

lub inaczej:

$$x^n - w = 0 \tag{5.4}$$

Jeżeli oznaczymy $f(x) = x^n - w$ to zadanie to można zapisać ogólniej: należy znaleźć takie x, że $f(x) = 0$.

Jeżeli zadanie dodatkowo uprościmy zakładając:
— funkcja ma dokładnie jedno miejsce zerowe,
— jest różniczkowalna,
— jej pochodna w całym przedziale jest albo dodatnia albo ujemna;
to nasze zadanie można zilustrować rysunkiem 5.8.

Zaczynamy w punkcie g; wartość funkcji w tym punkcie wynosi $f(g)$. Musimy w jakiś sposób zdecydować w którym kierunku należy wykonać następny krok. Zauważmy, że możemy w tym celu wykorzystać pochodną (czerwona, przerywana linia na rysunku 5.8). Jeżeli przybliżymy funkcję za pomocą pochodnej (stycznej do funkcji, przechodzącej przez punkt $(g, f(g))$ to następnym przybliżeniem będzie punkt przecięcia stycznej z osią x.

wer. 16 z drobnymi modyfikacjami!

Z równania prostej mamy:

$$\frac{f(g)-0}{g-g'} = f'(g) \tag{5.5}$$

czyli

$$\frac{f(g)}{f'(g)} = g - g' \tag{5.6}$$

i dalej

$$g' = g - \frac{f(g)}{f'(g)} \tag{5.7}$$

Jeżeli zauważymy, że $f(x) = x^n - w$ oraz, że $f'(x) = nx^{n-1}$ to kolejne przybliżenie wyliczane będzie ze wzoru:

$$g' = g - \frac{g^n - w}{ng^{n-1}} \tag{5.8}$$

albo

$$g' = \frac{ng^n - g^n + w}{ng^{n-1}} = \frac{(n-1)g^n + w}{ng^{n-1}} = \frac{1}{n}\left((n-1)g + \frac{w}{g^{n-1}}\right) \tag{5.9}$$

Gdy $n = 2$, wówczas

$$g' = \frac{1}{2}\left(g + \frac{w}{g}\right). \tag{5.10}$$

Umawiamy się, że program kończy pracę gdy kolejna poprawka g' nie różni się zbytnio od poprzednio wyliczonej wartości g, czyli $|g - g'| < \varepsilon$.

5.11.2. Realizacja programowa

Idea programowania strukturalnego może być realizowana przez nieustanne myślenie jakie całości (mniejsze algorytmy) można wyróżnić w projektowanym algorytmie. I cały czas dekomponować zadanie na mniejsze, ale logicznie zmknięte, zadania.

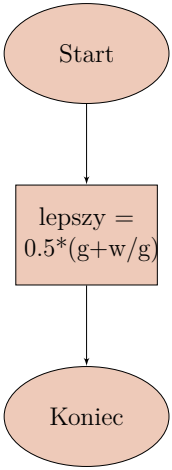
Program będzie się składał z trzech części:

1. `blisko(g, gprim)` — funkcja o wartościach logicznych sprawdzająca czy $|g - g'| \leqslant \varepsilon$,
2. `lepszy(n, w, g)` — funkcja rzeczywista wyliczająca następne, lepsze przybliżenie pierwiastka,

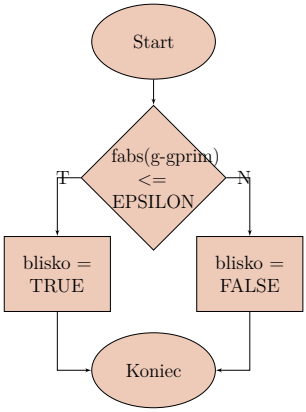
3. `pierwiastek(n, w, g)` — funkcja (rzeczywista) wyliczająca pierwiastek stopnia n z w zaczynając od przybliżenia g.

Uwaga: Dalszy przykład zakłada n = 2

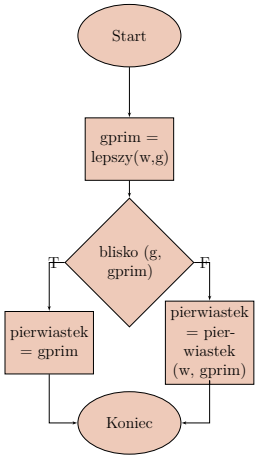
Realizacja programowa
`lepszy(w, g)`



Realizacja programowa
`blisko(g, gprim)`

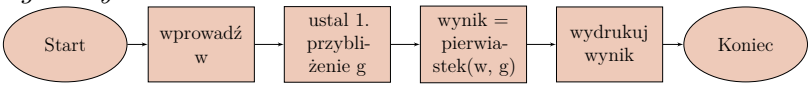


Realizacja programowa
`pierwiastek(w, g)`



Realizacja programowa

Program główny



Metoda Newtona

Realizacja programowa

Program składa się z trzech części:

1. `blisko(g, gprim)` — funkcja o wartościach logicznych sprawdzająca czy $|g - g'| \leqslant \varepsilon$,
2. `lepszy(n, w, g)` — funkcja rzeczywista wyliczająca następne, lepsze przybliżenie pierwiastka,
3. `pierwiastek(n, w, g)` — funkcja (rzeczywista) wyliczająca pierwiastek stopnia n z w zaczynając od przybliżenia g.

Uwaga: Dalszy przykład zakłada n = 2

W poniższych fragmentach kodu zastosowano powszechną konwencję zaznaczania linii kontynuacji za pomocą znaku \ umieszczonej na końcu linii. Zatem

```
1 int blisko(double g,          \
2                double gprim)
3 {
4     return fabs(g - gprim) \
5                  < EPSILON;
```

```
6 }
```

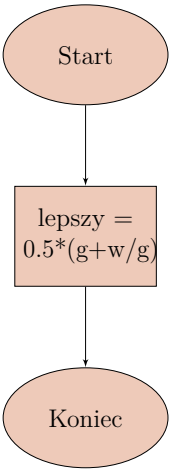
należy interpretować jako

```
1 int blisko(double g, double gprim)
2 {
3     return fabs(g - gprim) < EPSILON;
4 }
```

Każdy kompilator języka C rozumie ten zapis!

Realizacja programowa

lepszy(w, g)

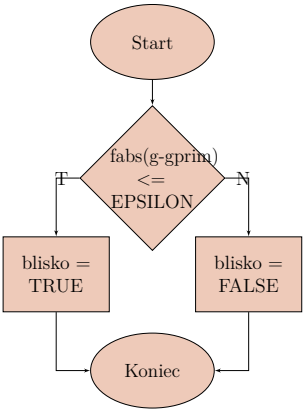



```
1 double lepszy(double w, double g)
2 {
3     return 0.5 * (g + w/g);
4 }
```

Metoda Newtona

blisko(g, gprim)

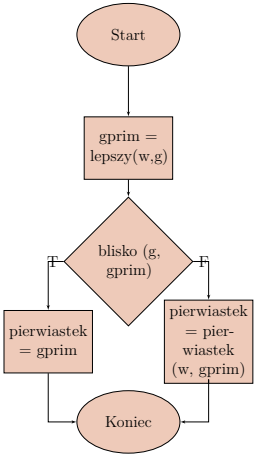
wer. 16 z drobnymi modyfikacjami!




```
1 int blisko(double g,          \
2                double gprim)
3 {
4     return fabs(g - gprim) \
5                  < EPSILON;
6 }
```

Metoda Newtona

pierwiastek(w, g)

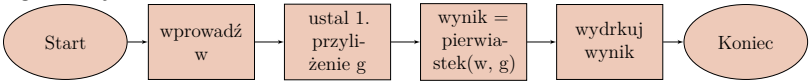



```
1 double pierwiastek(double w, \
2                    double g)
3 {
4     double gprim;
```

```
5       gprim = lepszy(w, g);
6       if ( blisko(g, gprim) )
7       return gprim;
8       else
9       return pierwiastek(w, \
10                         gprim);
11  }
```

Metoda Newtona

Program główny




```
1   int main(void)
2   {
3       double w, g, wynik;
4       w = 2.;
5       g = 1.;
6       wynik = sqrtf(w);
7       printf("%f\n", wynik);
8       wynik = pierwiastek(w, g);
9       printf("Pierwiastek␣kwadratowy␣z␣liczby" \
10              "␣%f␣wynosi␣%f\n", w, wynik);
11      return 0;
12  }
```

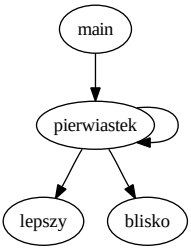
Metoda Newtona-Raphsona

Zadanie domowe

1. Narysować schemat blokowy dla dowolnego n (wszystko to co było to było dla $n = 2$.
2. Napisać program (w C) realizujący ten schemat blokowy.

wer. 16 z drobnymi modyfikacjami!

5.11.3. Call graph



Powyższy diagram przedstawia tak zwany „call graph” (czyli graf przedstawiający „kto kogo wywołuje”) programu newton opisywanego wcześniej.

Tworzony jest on automatycznie na podstawie analizy przebiegu programu. Do jego uzyskania potrzebny są perlowy program egypt oraz program graphviz.

6. Tablice (jedno i wielowymiarowe), łańcuchy znaków

6.1. Zmienne

1. Wszystkie zmienne muszą być zadeklarowane.
2. Nazwa zmiennej składa się z liter i cyfr, a rozpoczyna się literą; znak podkreślenia zalicza się do liter.
3. Nazwy zmiennych nie powinny się zaczynać od znaku podkreślenia (tak nazywają się zmienne systemowe).
4. Deklaracja obowiązuje wewnątrz bloku (i we wszystkich blokach znajdujących się „niżej").
5. W C występują zmienne globalne (zewnętrzne) i lokalne.
6. Deklaracja lokalna przysłania deklarację globalną (jeżeli nawa zmiennej jest taka sama).

W szczególności...
...poniższa konstrukcja

```
1 for(int i = 0; i < 10; i++)
2 {
3     printf("i = %d\n), i);
4 }
```

jest poprawna. Ale poniższa

```
1 for(int i = 0; i < 10; i++)
2 {
3     printf("i = %d\n), i);
4 }
5 printf("i = %d\n), i);
```

już nie... ...gdyz zmienna i jest zmienną lokalną tylko dla pętli!

6.2. Zmienne statyczne i automatyczne

Zmienne zewnętrzne i wewnętrzne

```
1 #include <stdio.h>
2 int a; // <-- Zmienna zewnetrzna
3 int main(void)
4 {
5       int b; // <-- Zmienna wewnetrzna
6       ...
```

Zmienne zewnętrzne nazywane bywają zmiennymi „globalnymi” (czyli dostępnymi dla każdej funkcji programu).

Zmienne statyczne i automatyczne

1. Dodatkowo można zażądać od zmiennej żeby była „statyczna” (co deklaruje się dodając słowo kluczowe **static** przed nazwą typu).

```
1 static int x;
```

2. Zmienna statyczna zewnętrzna pozostaje zdefiniowana **tylko** dla funkcji zdefiniowanych w jednym pliku źródłowym (ukryta jest dla funkcji z innych plików źródłowych).
3. Zmienna statyczna wewnętrzna zachowuje swoją wartość pomiędzy kolejnymi wywołaniami funkcji.
4. Zmienne, które nie są statyczne **nie muszą** zachowywać wartości między wejściami do funkcji (ale mogą) — ale nie można na to liczyć!
5. Dodatkowo zmienne statyczne wewnętrzne inicjowane są na wartość zero (jeżeli programista nie zażąda żeby było inaczej).

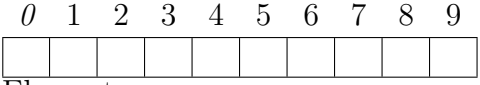
Zmienne statyczne i automatyczne

```
1 #include <stdio.h>
2 void f(void)
3 {
4     static int x ; /* zmienna statyczna */
5     int y = 0;     /* zmienna automatyczna */
6     x++;
7     y++;
8     printf("X=%d, Y=%d\n", x, y);
9 }
```

```
10 int main()
11 {
12     f();
13     f();
14     f();
15     return 0;
16 }
```

6.3. Tablice

1. Gdy potrzebujemy przechować kilka zmiennych tego samego typu (i jakoś powiązanych ze sobą) stosujemy **tablicę**.
2. Tablica to ciąg zmiennych o tej samej nazwie; dostęp do poszczególnych elementów odbywa się przez podanie numeru zmiennej (indeksu/ów).



3. Elementy numerowane są **począwszy od zera**.
4. Deklaracja wygląda tak: **typ** nazwa_tablicy[rozmiar];

Tablice

1. Tablica jest zmienną złożoną (strukturą pewnego rodzaju).
2. Słuzy do przechowywania danych tego samego typu.
3. Jeżeli chcemy nadać elementom tablicy wartości początkowe **int** tablica[3] = {1,2,3};
4. To jest również poprawna deklaracja: **int** tablica[20] = {1,}; (pierwszy element tablicy ma wartość 1, pozostałe mają wartość 0)
5. Nie zawsze trzeba podawać rozmiar tablicy — czasami kompilator może się domyślić sam: **int** tablica[] = {1, 2, 3, 4, 5}; zostanie zadeklarowana tablica o pięciu elementach.

6.3.1. Wielkość tablic

```
1 #include<stdio.h>
2 int main(void)
3 {
4     int t[] = {1, 2, 3, 4, };
5     int i;
```

wer. 8 z drobnymi modyfikacjami!

```
6       for (i = -1; i < 7; i++)
7           printf("t[%d] = %d\n", i, t[i]);
8       return 0;
9   }
```

Ile elementów ma tablica t?

Poniżej wynik kilku kolejnych uruchomieć tego programu. Jak widać zawartości „niezdefiniowanych” komórek różnią się w kolejnych wywołaniach.

Wykonany po raz pierwszy

```
t[-1] = 11131
t[0] = 1
t[1] = 2
t[2] = 3
t[3] = 4
t[4] = -1296194160
t[5] = 32767
t[6] = 0
```

Wykonany po raz drugi

```
t[-1] = 10955
t[0] = 1
t[1] = 2
t[2] = 3
t[3] = 4
t[4] = -868000288
t[5] = 32767
t[6] = 0
```

Wykonany po raz trzeci

```
t[-1] = 11015
t[0] = 1
t[1] = 2
t[2] = 3
t[3] = 4
t[4] = -143761264
t[5] = 32767
```

```
t[6] = 0
```

6.4. Inicjowanie zmiennych (zwłaszcza tablic)

1. W deklaracji obiektu można zawrzeć wartość początkową deklarowanego identyfikatora.
2. Inicjator, który poprzedza się operatorem = jest albo wyrażeniem, albo listą inicjatorów zawartą w nawiasach klamrowych.
3. Lista może kończyć się przecinkiem.
4. Dla obiektów i tablic statycznych wszystkie wyrażenia w inicjatorach muszą być wyrażeniami stałymi.
5. Nie inicjowany jawnie obiekt statyczny jest inicjowany tak, jakby jemu, (lub jego składowym) przypisano wartość zero.
6. Początkowa wartość nie zainicjowanego jawnie obiektu automatycznego jest niezdefiniowana.
7. Inicjatorem dla obiektu arytmetycznego jest pojedyncze wyrażenie (być może ujęte w nawiasy klamrowe).
8. Inicjatorem dla struktury jest albo wyrażenie tego samego typu albo ujęta w nawiasy klamrowe lista inicjatorów dla jej kolejnych składowych.
9. Inicjatorem dla tablicy jest ujęta w klamry lista inicjatorów dla jej kolejnych elementów.
10. Jeśli nie jest znany rozmiar tablicy — to rozmiar ten wylicza się na podstawie liczby inicjatorów.
11. Jeśli tablica ma ustalony rozmiar — liczba inicjatorów nie może przekroczyć liczby elementów tablicy; jeśli lista jest krótsza uzupełniana jest zerami.
12. Specjalnym przypadkiem jest tablica znakowa, która może być inicjowana napisem (kolejne znaki napisu inicjują kolejne elementy tablicy).
13. Jeżeli nie jest znany rozmiar tablicy znakowej jest on wyliczany na podstawie liczby znaków w napisie (włączając w to końcowy znak zerowy).

6.5. Tablice wielowymiarowe

```
1 #include<stdio.h>
2 int main(void)
3 {
```

wer. 8 z drobnymi modyfikacjami!

```
4       int a[4][3] = {
5           {1, 3, 5},
6           {2, 4, 6},
7           {3, 5, 7},
8       };
9       int i, j;
10      for (i = 0; i < 4; i++){
11          for (j = 0; j < 3; j++)
12          printf(" %d |", a[i][j]);
13          printf("\n");
14      }
15      return 0;
16  }
```

Z zapisu wynika, że tablica dwuwymiarowa, może być traktowana jako tablica jednowymiarowa, której elementami są wektory (wierszowe).

Tablice wielowymiarowe

Wynik działania programu

```
 1 | 3 | 5 |
 2 | 4 | 6 |
 3 | 5 | 7 |
 0 | 0 | 0 |
```

Tablice wielowymiarowe

Zadanie domowe

Zmodyfikować tak przykładowy progam, żeby drukował wyniki w następującej postaci:

```
| 1 | 3 | 5 |
| 2 | 4 | 6 |
| 3 | 5 | 7 |
| 0 | 0 | 0 |
```

Tablice wielowymiarowe

Inicjowanie — warianty

```
1       int a[4][3] = {
2           1, 3, 5, 2, 4, 6, 3, 5, 7
```

```
3       };
```

Niestety, powyższe nie do końca jest poprawne: pojawią się ostrzeżenia podczas kompilacji!

```
1 | 3 | 5 |
2 | 4 | 6 |
3 | 5 | 7 |
0 | 0 | 0 |
```

Tablice wielowymiarowe

Inicjowanie — warianty

```
1       int a[4][3] = {
2           { 1 }, { 2 }, { 3 }, { 4 }
3       };
```

```
1 | 0 | 0 |
2 | 0 | 0 |
3 | 0 | 0 |
4 | 0 | 0 |
```

6.5.1. Napisy

1. Stała znakowa (złożona z jednego znaku) zapisywana jest tak `'c'` („c” to dowolny znak lub specjalna stała złożona ze znaku „backslash” (\) i specjalnego symbolu.
2. Stała tekstowa zapisywana jest w cudzysłowach (podwójne apostrofy) `"Ala ma kota"`
3. Sąsiadujące ze sobą napisy łączone są w jeden napis ("Ala" "ma""kota" tworzy napis "Alamakota"; "Ala␣" "ma""␣kota" tworzy "Ala ma kota").
4. Na końcu napisu umieszczany jest znak o kodzie ASCII równym zero ('\x00') pozwalający rozpoznać koniec tekstu.
5. W napisach można używać wszystkich symboli specjalnych dostępnych w stałych znakowych.
6. Typem do przechowywania znaków jest **char**.
7. Napisy trzeba przechowywać w tablicach typu **char**.
8. Polskie znaki — na razie proponuję o tym zapomnieć!

wer. 8 z drobnymi modyfikacjami!

6.5.2. Tablice znakowe

To jest poprawna deklaracja. Tablica będzie miała rozmiar 14 (13 znaków napisu i znak null kończący napis). Można to sprawdzić za pomocą funkcji `sizeof(tekscik)`.

```
1     char tekscik[] = "Ala ma kotala";
```

Poniżej również poprawna deklaracja tablicy (o łącznym rozmiarze 21 znaków). Liczba wierszy wyliczana jest automatycznie podczas kompilacji.

```
1     char teksty[][7] = {
2         {"Ala"},
3         {"ma"},
4         {"kotala"}
5     };
```

To niepoprawna forma deklaracji:

```
1     char teksty[3][] = {
2         {"Ala"},
3         {"ma"},
4         {"kotala"}
5     };
```

Tablice jako argumenty funkcji

1. Trzeba bardzo uważać i myśleć zanim się coś zrobi!
2. Rzecz nie jest prosta (choć może nie aż tak skomplikowana).

Przykład

— Chcemy napisać funkcję, która zeruje tablicę (wypełnia wartością zero wszystkie elementy tablicy).
— Sam algorytm jest bardzo prosty

```
1 int N = 10;
2 double A[N];
3 for(int i =0; i < N; i++)
4    A[i] = 0.;
```

— Musimy go tylko obudować w funkcję

— nie zwraca parametrów
— ma dwa parametry:
1. tablicę
2. jej rozmiar

Przykład c.d.

```
1 void zeruj(int N, double A[])
2 {
3    for(int i = 0; i < N; i++)
4       A[i] = 0.;
5 }
```

A używać jej będziemy tak:

```
1 int main()
2 {
3   double X[1000];
4   int M = 1000;
5 //   ...
6   zeruj(M, X):
7 // ...
8 }
```

6.6. Kilka uwag na temat notacji

Zapis a[m][n] jest wieloznaczny:

1. Gdy jest poprzedzony nazwą typu danych (**char**, **int**, **float**,...) m i n oznaczają **wymiary tablicy**:

```
1 int n = 10, m = 10, a[m][n];
```

Deklaruje tablicę a o 10 wierszach i 10 kolumnach.
2. podobnie deklaracja funkcji:

```
1 double srednia(int m, int n, double a[m][n]);
```

iformuje, że trzecim jej argumentem jest powinien być obiekt typu **int** (∗)[n].

3. W wyrażeniu

```
1 b = a[m][n];
```

oznacza z tablcy a elementu leżącego na przecięciu m-tego wiersza i n-tej kolumny (m i n muszą być mniejsze niż wymiary tablicy).

4. Podobnie w przypadku wywołania funkcji:

```
1 y = f(a[m][n])
```

oznacza przekazanie do funkcji f wartości wskazanego elementu tablicy do funkcji, zaś

```
1 y = g(a)
```

oznacza przekazanie do funkcji g adresu początku tablicy.

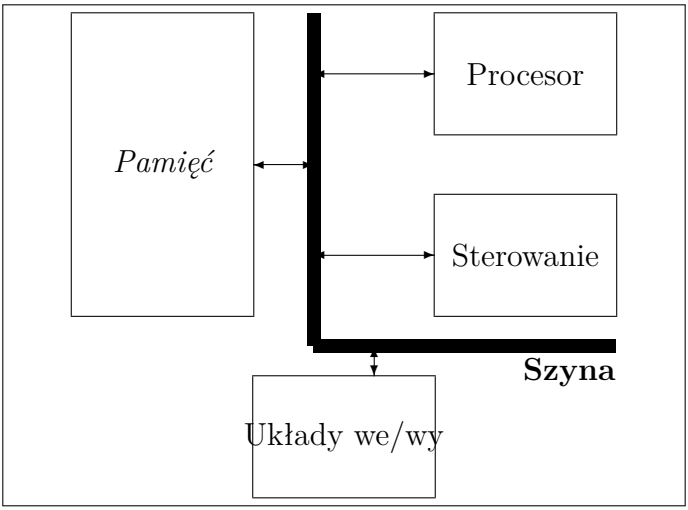
7. Wskaźniki. Pamięć dynamiczna

7.1. Wskaźniki

— Wskaźniki to podstawa C.
— Kto nie umie się nimi posługiwać — ten nie potrafi wykorzystać siły języka C.
— C używa wskaźników bardzo często. Czemu?
  — Bardzo trudno zaprogramować beż nich pewne operacje.
  — Pozwalają na tworzenie zwartego i efektywnego kodu.
  — Są bardzo efektywnym narzędziem.
— Korzystając z:
  — tablic,
  — struktur (będzie później),
  — funkcji
  korzystamy ze wskaźników.
— Wskaźniki to — najprawdopodobniej — najtrudniejszy fragment języka C. Nie przydadzą się tu doświadczenia uzyskane podczas programowania w innych językach.

Wskaźnik
to specjalna zmienna zawierająca „adres” (w pamięci RAM) innej zmiennej.

7.1.1. Pamięć komputera



Pamięć komputera

1. RAM (Random Access Memory — pamięć o dostępie swobodnym).
2. ROM (Red Only Memory — pamięć tylko do odczytu).
3. Pamięć dyskowa.
4. Pamięci „zewnętrzne” (dyski USB, dyski sieciowe, ...)

My (teraz) zajmować się będziemy tylko pamięcią RAM komputera, choć bardzo często pamieć dyskowa jest swojego rodzaju przedłużeniem pamięci RAM („systemowy plik wymiany” czy *swap*).

Pamięć operacyjna

1. Można ją przedstawić w postaci następującej:

adres	0	1	2	3	4	5	6	7	8	···	···
zawartość	·	·	·	·	·	·	·	·	·	·	·

2. Pamięć praktycznie wszystkich komputerów ma organizację **bajtową**. (Najmniejszą adresowalną jednostką pamięci jest bajt.)
3. Zmienne różnych typów zajmują w pamięci różną liczbę bajtów.
4. W chwili deklarowania zmiennych kompilator przydziela im w pamięci miejsce.
5. Każdy program uruchamiany jest w „maszynie wirtualnej” i ma dostęp tylko do przydzielonej mu pamięci.

Zmienne a pamięć operacyjna

1. Zmienne różnych typów zajmują w pamięci różną liczbę bajtów.
2. W chwili deklarowania zmiennych kompilator przydziela im w pamięci miejsce.
3. Używając nazwy zmiennej (w jakiś) sposób wskazujemy miejsce w pamięci operacyjnej

```
1 a = b + 3;
```

   należy czytać „pobierz zawartość pamięci operacyjnej przydzielonej zmiennej b, dodaj do niej 3, a wynik zapisz w miejscu pamięci operacyjnej przydzielonym zmiennej a”.
4. . . . ale (przy takim zapisie) nie mamy żadnego dostępu do adresu zmiennej!
5. Należy natomiast pamiętać, że polecenie:

```
1 a = b;
```

   powoduje **przekopiowanie** zawartości zmiennej b do zmiennej a. a i b to dwa różne obiekty!

Pytanie jest takie: *Czy jest nam potrzebny (i do czego) adres zmiennej?*

Typy danych i zajętość pamięci

```
1 sizeof(char   ) = 1
2 sizeof(short  ) = 2
3 sizeof(int    ) = 4
4 sizeof(long   ) = 8
5 sizeof(float  ) = 4
6 sizeof(double) = 8
7 sizeof(long double) = 16
```

7.1.2. Pamięć automatyczna, statyczna i „ręczna”

Język C dostarcza trzech sposobów zarządzania pamięcią (to ponad dwa razy więcej niż inne języki programowania [11]).

Pamięć automatyczna. To ten rodzaj pamięci, która jest deklarowana wtedy kiedy jest potrzebna i znika gdy tylko opuścimy „kontekst” (czyli blok w którym deklaracja została użyta. Wszystkie zmienne w funkcjach (o ile tylko nie poprzedzone deklaracją **static** są automatyczne.

wer. 13 z drobnymi modyfikacjami!

Pamięć statyczna. Zmienne statyczne w trakcie działania programu zajmują to samo miejsce. Wielkości tablic nie mogą się zmieniać (choć ich zawartość już tak). Wartości są inicjalizowane zanim rozpoczną się obliczenia, więc nie mogą zależeć od innych zmiennych. Zmienne deklarowane na zewnątrz funkcji oraz wewnątrz funkcji, gdy deklaracja poprzedzona jest słowem **static** są statyczne. Dodatkowo, gdy nie dokonamy inicjalizacji zmiennej — przyjmie ona wartość zero.

Pamięć przydzielana „ręcznie”. To ten rodzaj pamięci, który przydzielamy używając funkcji malloc i zwalniamy funkcją free. Ten rodzaj pamięci stwarza bardzo wiele problemów (i studenci płaczą gdy muszą iej używać), ale daje też pewne zalety: możemy zmieniać wielkość tablic w trakcie pracy programu.

	Statyczna	Automatyczna	Ręczna
Zerowana podczas startu programu	√		
Ograniczona do kontekstu	√	√	
Można inicjalizować	√	√	
Można inicjalizować wartościami wyliczanymi		√	
Zachowuje wartość między „wejściami do funkcji”	√		√
Może być globalna	√		√
Można ustalać wielkość tablic podczas pracy programu		√	√
Można zmieniać wielkość tablic			√
Studenci płaczą			√

Niektóre z powyższych właściwości mogą się przydać podczas programowania, na przykład możliwość zmiany wielkości tablicy w trakcie pracy programu, albo inicjalizacja tablicy podczas uruchamiania programu. Zazwyczaj niestety nie można mieć wszystkiego.

Warto pamiętać o następujących rzeczach:

1. Gdy zadeklarujesz dane typu **struct**, **char**, **int**, **double** lub inną zmienną na zewnątrz funkcji lub wewnątrz funkcji, ale poprzedzając deklarację słowem **static** — będzie zmienną statyczną, w przeciwnym razie — automatyczną.
2. Jeżeli zadeklarujesz wskaźnik, ma on swój typ (automatyczny lub statyczny zgodnie z zasadą 1). Ale wskaźnik może wskazywać na dowolny

typ pamięci: statyczny wskaźnik na pamięci utworzoną ręcznie za pomocą funkcji malloc, wskaźnik automatyczny na dane statyczne; każda kombinacja jest możliwa.

Powyższe powoduje, że, z jednej strony, model jest bardzo wygodny, i wszędzie można stosować ten sam zapis, ale z drugiej trzeba pamiętać, że stwierdzenie „W języku C wskaźniki i tablice to to samo" jest, po prostu, nieprawdziwe.

7.1.3. Stos i sterta

Warto w tym miejscu wspomnieć jeszcze o dwu sprawach związanych z przydziałem pamięci w programach napisanych w C. Otóż podczas wykonania funkcji, tworzone jest specjalne „środowisko" w którym ona działa. W szczególności środowisko to zawiera wszystkie zmienne zadeklarowane w funkcji. Gdy funkcja wywołuje inną funkcję — środowisko zostaje odłożone na stos. Pojemność stosu jest ograniczona.

Popatrzmy na taki program:

```
1 #include <stdio.h>
2 int f(int x)
3 {
4         static int y = 0;
5         printf("%d\n", y++);
6         return f(x);
7 }
8 int main(int argc, char **argv)
9 {
10        int x = 0;
11        x = f(x);
12        return 0;
13 }
```

Nie robi on nic specjalnie zmyślnego. Funkcja f wywołuje samą siebie. Wykorzystałam zmienną statyczną[1] y do przechowywania licznika wywołań funkcji — po każdym wywołaniu f zwiększa się on o jeden. Udało mi się wykonać funkcję 261 931 razy zanim program skończył obliczenia z komunikatem

[1] Zmienna statyczna nie „znika" po wyjściu z funkcji i zachowuje swoją wartość.

„Program has been terminated receiving signal 11 (Segmentation fault)", więc widać, że przestrzeń stosu jest ograniczona.

Kolejną konsekwencją tego faktu jest to, że sumaryczna pojemność zmiennych automatycznych jest ograniczona pojemnością stosu, to znaczy w następującym fragmencie

```
1         int a[1000];
```

wielkość tablicy a jest ograniczona! 1000 elementów na pewno się zmieści. Ale ile więcej?

Aby się o tym przekonać można przetestować następujący program:

```
1 #include <stdio.h>
2 int main ()
3 {
4     int i;
5     for ( i = 1;; i = i * 2 )
6     {
7         printf("i=%d\n", i);
8         double a[i];
9         int j;
10        for ( j = 0; j < i; j++ )
11            a[j] = j;
12    }
13    return 0;
14 }
```

U mnie, program kończy pracę po wydrukowaniu: i=1 048 576. Natomiast poniższy program:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main ()
4 {
5         int i;
6         for (i=1;;i=i*2)
7         {
8                 printf("i=%d\n", i);
9                 double *a;
10                a = malloc(8 * i);
```

```
11                  int j;
12                  for (j=0;j<i;j++)
13                          a[j]=j;
14                  free(a);
15          }
16          return 0;
17 }
```

przerywa pracę po wydrukowaniu i=268 435 456. Jedyna różnica to użycie funkcji malloc.

W przypadku używania funkcji malloc pamięć nie jest rezerwowana na stosie, a na tak zwanej stercie (ang. heap). W przybliżeniu można zakładać, że wielkość sterty jest równa wielkości dostępnej pamięci komputera (RAM + plik wymiany – pamięć zajęta przez inne programy).

7.1.4. Pamięć statyczna

Pamięć statyczna przydać się może podczas realizacji funkcji Fibonacciego. Wersja rekurencyjna opisana na stronie 134 ma szereg wad: wykonuje się strasznie długo!

Poniższy program ([11]) wykonuje się błyskawicznie:

```
1 #include <stdio.h>
2 long long int fibonacci(){
3     static long long int first = 0;
4     static long long int second = 1;
5     long long int out = first + second;
6     first = second;
7     second = out;
8     return out;
9 }
10
11 int main(){
12     int i;
13     for ( i = 0; i < 50; i++ )
14         printf( "%lli\n", fibonacci() );
15     return 0;
16 }
```

7.2. Deklaracja wskaźników

Wskaźniki

1. Wskaźnik, w języku C to (w pewnym uproszczeniu) adres zmiennej w pamięci operacyjnej.
2. Wskaźniki, tak jak wszystko, muszą być deklarowane.
3. Deklaracja wskaźnika jest „dziwna” choć nie różni się wiele od deklaracji innych zmiennych:

```
1 int *ip;
2   /* ip jest wskaznikiem do obiektu typu int */
3 double *fp;
4   /* fp jest wskaznikiem do obiektu typu double */
5 char* cp // wskaznik do typu char
6 float*Fp // wskaznik do typu float
```

   mówi jakiego typu jest zmienna na którą wskaźnik wskazuje.
   Inaczej na sprawę patrząc można napis **int** $*$ (albo **double** $*$) traktować jako inny typ danych. Położenie gwiazdki nie jest tak istotne!
4. Podstawowe operacje na wskaźnikach to
   — & pobranie adresu zmiennej na którą ma wskazywać wskaźnik.
   — $*$ pobranie zawartości zmiennej wskazywanej przez wskaźnik (gdy występuje po prawej stronie znaku równości) lub nadanie wartości zmiennej wskazywanej przez wskaźnik (gdy występuje po lewej stronie znaku równości).
   — − Odejmowanie wskaźników. Wynikiem jest liczba calkowita długa.
   — + Dodanie do wskaźnika stałej/zmiennej (całkowitej).
   — − Odjęcie od wskaźnika stałej/zmiennej całkowitej.
   W ostatnich dwu przypadkach wynikiem jest nowy adres.

```
1 int x = 1, y = 2, z[10];
2 int *ip;     /* ip jest wskaznikiem do obiektu typu int */
3
4 ip = &x;     /* teraz ip wskazuje na x */
5 y = *ip;     /* y ma teraz wartosc 1 */
6 *ip = 0;     /* x ma teraz wartosc 0 */
7 ip = &z[0]; /* teraz ip wskazuje na element z[0] */
```

Aliasy

2023-04-12 11:02:16 +0200

Jeżeli pa i pb to wskaźniki tego samego typu

```
1  int b;
2  int *pa, *pb;
3  pb = &b;
```

to polecenie

```
1  pa = pb;
```

tworzy coś w rodzaju aliasu: przez nazwy pa i pb możemy odwoływać się do tej samej zmiennej.

7.2.1. Wskaźniki i tablice

Wskaźniki i tablice

1. Wskaźniki mogą przydawać się w przypadku organizowania dostępu do tablic.
2. Działają w tym przypadku trochę jak indeks tablicy.
3. Jeżeli mamy coś takiego:

```
1 int a[10];
2 int *pa;
3
4 pa = &a[0];
```

w **pa** mamy wskaźnik pokazujący na zerowy element tablicy **a**, czyli

```
1 x = *pa;
```

jest równoważne poleceniu

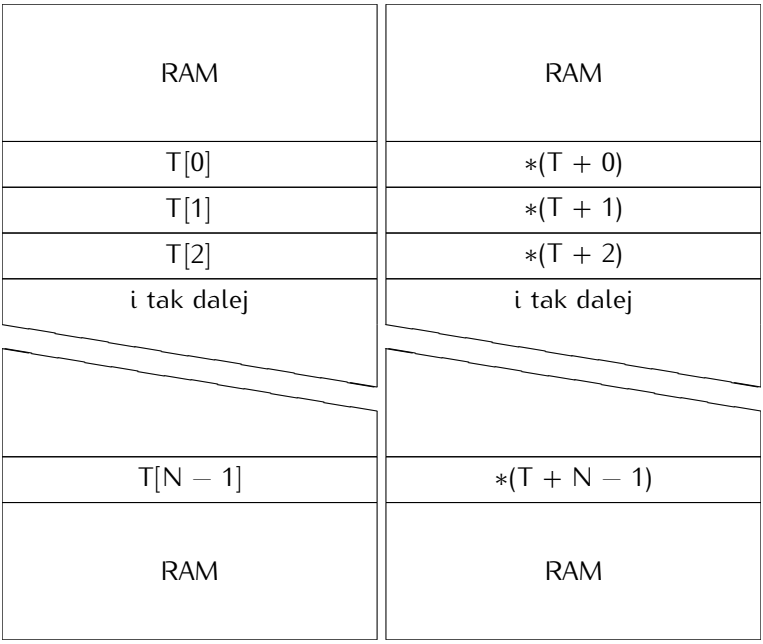
```
1 x = a[0];
```

Natomiast zwiększenie wskaźnika o 1 da nam dostęp do następnej komórki w tablicy **a**; zapisujemy to tak:

```
1 y = *(pa + 1);
```

gdyż jednoargumentowy operator $*$ ma wyższy priorytet niż dodawanie!

wer. 13 z drobnymi modyfikacjami!



Wskaźniki i tablice

1. Wszystko działa poprawnie niezależnie od tego ile miejsca w pamięci zajmuje element tablicy (jakiego jest typu) tak długo, jak poprawnie deklarujemy zmienną wskaźnikową...
2. pa+1 pokazuje „następny obiekt" tablicy
3. pa+i pokazuje element oddalony od pa o i takich obiektów.
4. Nazwa tablicy jest **stałą** typu wskaźnikowego! Zamiast pisać

```
1 pa = &a[0];
```

   można napisać

```
1 pa = a;
```

5. Natomiast odwołanie do a[i] można zapisać jako ∗(a+i). Nazwa tablicy pełni rolę wskaźnika do jej pierwszego elementu.
6. Identyczne są również a+i oraz &a[i].
7. Natomiast wskaźnik jest zmienną i można na nim wykonywać operacje typu pa=a czy pa++. Nazwa tablicy nie jest zmienną (raczej należy ją traktować jak stałą), zatem konstrukcja a=pa czy a++ są niedozwolone!

7.3. Tablice dwuwymiarowe

1. Język C zna pojęcie tablicy dwuwymiarowej.
2. Deklaruje się ją tak:
   <typ> <*nazwa*> [<*liczba_wierszy*>][<*liczba_kolumn*>]
3. **int** tab2 [10][20] // to tablica o 10 wierszach i 20 kolumnach.
4. Dane w tablicy przechowywane są „wierszami": najpierw w pamięci zapisane są dane pierwszego wiersza, później drugiego,...
5. Można też wyobrazić sobie inną konstrukcję: najpierw deklarujemy tablicę jednowymiarową w której zapisujemy wskaźniki do jednowymiarowych tablic (wierszy tablicy).

7.3.1. Wskaźniki i funkcje

Funkcje i parametry raz jeszcze

Podczas wywoływania funkcji wykonywane są następujące czynności:

1. Wyliczana jest wartość wszystkich argumentów funkcji.
2. Dokonywane są konwersje typów.
3. Wyznaczone tak wartości „przekazywane" są do wnętrza funkcji (to znaczy wyznaczone wartości przypisywane są zmiennym wewnątrz funkcji.
4. Nastepnie zostaje wykonana funkcja.
5. Ewentualy wynik przekazywany jest poleceniem **return**.
6. Jakiekolwiek zmiany wartości zmiennych lokalnych lub parametrów **wewnątrz** funkcji nie są przekazywane na zewnątrz.

7.4. Funkcje

Paramatry

Jednym z ważniejszych obowiązków programisty jest zadbanie aby:

1. liczba parametrów w wywołaniu funkcji była identyczna z liczbą parametrów w definicji
2. zgadzał się typ parametrów.

O ile brak zgodności w pierwszym punkcie kończy się niepowodzeniem kompilacji, to w drugim przypadku zazwyczaj powoduje ostrzeżenia. W wer. 13 z drobnymi modyfikacjami!

większości przypadków dokonywane jest rzutowanie, które może powodować utratę informacji...

Uwaga
Szczególnie niebezpieczne nieprzemyślane rzutowanie wskaźników.

Wartość zwracana przez funkcje

1. Może być tylko **jedna**!
2. Typ funkcji **musi** być zgodny z typem wyrażenia pojawiającego się jako argument polecenia `return`.

```
typ f()
{
    ...;
    return cos
}
```

Czyli typy zmiennej `cos` lub wyrażenia arytmetycznego `cos` powinny być identyczne. W pewnym zakresie można liczyć na **automatyczne** rzutowanie.

Gdy nie określimy typów funkcji ani parametrów

```
f(x)
{
    x = x * 2;
    return x;
}
```

system **automatycznie** przyjmie, że typ zmiennej `x` oraz typ wartości zwracanej przez funkcję `f(x)` jest `int`.

Funkcje i parametry

1. Parametrem (formalnym) funkcji może być wskaźnik:

```
1 int funkcja(int *par)
2 {
3     return *par;
4 }
```

2. Podczas wywołania funkcji parametrem „aktualnym” w tym miejscu musi być adres zmiennej (prostej lub złożonej). Na przykład:

```
int a;
...
u = fumkcja(&a);
```

Prosty program

```
#include <stdio.h>
#include <stdlib.h>

int funkcja(int *par)
{
    return *par;
}

int main(void)
{
    int tablica[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    int zmienna_prosta = 123;
    int wynik;
    wynik = funkcja(&zmienna_prosta);
    printf("1. wynik = %d\n", wynik);
    wynik = funkcja(&tablica[3]);
    printf("2. wynik = %d\n", wynik);
    wynik = funkcja(tablica);
    printf("3. wynik = %d\n", wynik);
    return EXIT_SUCCESS;
}
```

Wynik

```
1. wynik = 123
2. wynik = 4
3. wynik = 1
```

Wskaźniki i funkcje

— Efektem ubocznym przekazywania parametrów do funkcji przez wartość jest to, że nie można w poniższy sposób stworzyć funkcji realizującej funkcję $a \leftrightarrow b$ (zamiana miejscami wartości dwu zmiennych; pierwsze, naiwne podejście, ***nie działa***:

```
1 void swap(int a, int b) /* To jest zle!!! */
2 {
3     int temp;
4     temp = a;
5     a = b;
6     b = temp;
7 }
```

gdyż żadne zmiany dokonywane na zmiennych wewnątrz funkcji nie „wydostają" się na zewnątrz.

Wskaźniki i funkcje

— Problem można rozwiązać za pomocą wskaźników:

```
1 void swap(int *pa, int *pb)
2 {
3     int temp;
4
5     temp = *pa;
6     *pa = *pb;
7     *pb = temp;
8 }
```

— Wywołanie tej funkcji będzie takie:

```
1 swap(&x, &y);
```

— Zwracam uwagę, że próba zmiany wartości wskaźników wewnątrz funkcji również nie przenosi się poza funkcję.

Funkcje, tablice, wskaźniki

— Ponieważ nazwa tablicy (w pewnym zakresie) może być traktowana jako wskaźnik, prawidłowa jest poniższa definicja parametru formalnego: **int** x[ ] i jest ona równoważna **int** *x

— Wydaje się, że drugi zapis jest „lepszy" (jako bliższy prawdy?)

— Można do funkcji przekazać fragment tablicy podając jako parametr aktualny wskaźnik do początku (pod)tablicy: f(&a[2])
— Wewnątrz funkcji **f** deklaracja parametru formalnego może mieć postać:
f(**int** arr[ ]) {...}
lub
f(**int** *arr) {...}
— Można też odwoływać się do elementów tablicy wstecz (arr[−1], arr[−2]) jeśli jest rzeczą pewną, że elementy te istnieją

Funkcje, tablice dwuwymiarowe, wskaźniki

— Tabice dwuwymiarowe mogą sprawiwć pewne problemy gdy powinny być parametrem funkcji.
— Gdy w programie głównym zadeklarowaliśmy tablice jako:
int tab2[10][20]
— W funkcji deklaracja parametru może wyglądać albo tak:
int f(**int** a[10][20])
albo tak:
int f(**int** a[ ][20])
albo tak: **int** f(**int** (*a)[20])

— Wywołanie funkcji zaś tak: i = f(tab2);

7.4.1. Wskaźnik jako wynik funkcji

W szczególności wynikiem funkcji może być wskaźnik. Funkcja taka będzie zadeklarowana jakoś tak:

```
1 int * test (...)
2 {
3     int * a;
4     ...
5     ...
6     return a;
7 }
```

Argumentem polecenia **return** musi być stała lub zmienna typu wskaźnikowego.

Przykład

wer. 13 z drobnymi modyfikacjami!

```
1 int * test (...)
2 {
3     int a[10]={1, 2, 3, 4};
4     ...
5     ...
6     return a;
7 }
```

— Powyższy przykład z formalnego punktu widzenia jest OK.
— Zmienna a (tablica) jest automatyczna i w związku z tym po wyjściu z funkcji przestaje istnieć, zatem zwracany wskaźnik wskazuje na coś czego już nie ma!
— Program można zmodyfikować tak żeby było lepiej, zastępując tablicę a tablicą statyczną:

```
1     static int a[10]={...};
```

W szczególności sens mogą mieć następujące funkcje (przydatne zupełnie gdzie indziej)

```
1 char glowa(char *X)
2 {
3     return X[0];
4 }
5 char * ogon{char *X)
6 {
7     return X + 1;
8 }
```

Co one robią?

7.4.2. Argumenty wywołania programu

— Możemy teraz (szybko) wrócić do przekazywania parametrów do funkcji main czyli programu.
— W chwili uruchomienia programu System Operacyjny tworzy strukturę danych zawierającą liczbę parametrów (integer, zazwyczaj nazywany argc — *argument count*) oraz tablicę zawierającą parametry (argv — *argument vector*).

```
1 #include <stdio.h>
2 main(int argc, char *argv[])
3 {
4     int i;
5     for (i = 1; i < argc; i++)
6         printf("%s%s", argv[i], (i < argc - 1)?" ":"");
7     printf("\n");
8     return 0;
9 }
```

Teraz powinno być już jasne w jaki sposób można „dobierać się” do danych (o których liczbie i długości nic nie wiemy w trakcie pisania programu).

— Ale jak się dobrać do pierwszego znaku pierwszego argumentu?
— *argv jest wskaźnikiem pokazującym zerowy element tablicy argumentów (nazwa programu).
— *++argv wskazuje na pierwszy (czyli właściwy) element tekstu.
— Najprawdopodobniej zatem (*++argv)[0] to pierwszy znak.
Uwaga Nawias okrągły jest potrzebny, gdyż operator [ ] (pobierania elementu tablicy) ma wyższy priorytet niż operatory adresu (*) i operator zwiększenia (++).

7.5. Pamięć dynamiczna

— Jedną z funkcji Systemu Operacyjnego jest przydział pamięci.
— Jak jest to realizowane?
  — Po pierwsze — gdy użytkownik uruchamia program, SO przydziela programowi pamięć niezbędną do pracy.
  — Po drugie przydziela pamięć — niejako automatycznie — na potrzeby powstających w trakcie pracy programu zmiennych.
— Co jednak zrobić, gdy podczas pisania programu nie znamy liczby danych (a zatem ilości potrzebnej do ich przechowywania pamięci)?
— Musimy wykorzystać funkcje dynamicznego przydziału pamięci!

Pamięć dynamiczna
malloc, calloc

1. Do dynamicznego przydzielania pamięci służy funkcja malloc (pamiętamy o **#include**<stdlib.h>)
2. Sposób użycia funkcji („prototyp" funkcji): **extern void** ∗malloc(size);
   — **extern** mówi, że funkcja jest „zewnętrzna" czyli zdefiniowana gdzie indziej niż nasze pliki źródłowe
   — **void** ∗malloc mówi, że funkcja zwraca wynik w postaci wskaźnika typu **void** (nieokreślonego typu). wskaźnik ten jest równy NULL[2] gdy System Operacyjny nie może przydzielić pamięci lub wskazuje na początek obszaru przydzielonej pamięci.
   — size to parametr mówiący ile (bajtów) pamięci potrzebujemy.
3. Żeby z funkcji korzystać trzeba użyć dyrektywy **#include** <stdlib.h> gdzieś na początku programu.
4. Przydzielona pamięć może zawierać przypadkowe wartości.
5. Funkcja **extern void** ∗calloc(nelem, elsize); może być wykorzystania do uzyskania „wyzerowanego" obszaru pamięci; pierwszy argument (nelem) określa liczbę żądanych jednostek, drugi (elsize) wielkość w bajtach każdej jednostki.

Pamięć dynamiczna

Przykład

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int number;
5     int *ptr;
6     int i;
7     printf( "How many ints would you like store? " );
8     scanf( "%d", &number );
9     /* allocate memory */
10    ptr = malloc( number * sizeof( int ) );
11    if ( ptr != NULL )
12    {
13        for ( i = 0; i < number; i++ )
14            *( ptr + i ) = i;
15        for ( i = number; i > 0; i— )
```

[2] NULL to stała typu wskaźnikowego o wartości zero.

```
16          /* print out in reverse order */
17          printf( "%d\n", *( ptr + ( i - 1 ) ) );
18       /* free allocated memory */
19       free( ptr );
20       return 0;
21    }
22    else
23    {
24       printf( "\nMemory allocation failed -"
25               " not enough \ memory.\n" );
26       return 1;
27    }
28 }
```

Dynamiczny przydział pamięci
Zwalnianie pamięci

1. Dobry obyczaj każe oddać to co się pożyczyło.
2. W zasadzie, w chwili zakończenia programu cała przydzielona pamięć powinna zostać automatycznie zwrócona...
3. ...ale różnie bywa.
4. Funkcja **extern void** free (**void** *ptr) zwraca pamięć. Jedynym argumentem jest wskaźnik początku obszaru przydzielonej pamięci.
5. Funkcja **extern void** *realloc (**void** *ptr, size) pozwala zmienić (rozszerzyć, zmniejszyć) posiadany obszar pamięci do zadanego obszaru.

Dynamiczny przydział pamięci
Pożytek ze wskaźników

```
1 #include <stdio.h>
2 /* required for the malloc and free functions */
3 #include <stdlib.h>
4 int main() {
5    int number;
6    int *ptr;
7    int i;
8    printf("How many ints would you like store? ");
9    scanf("%d", &number);
```

wer. 13 z drobnymi modyfikacjami!

```
10    /* allocate memory */
11    ptr = malloc( number * sizeof( int ) );
12    if ( ptr != NULL )
13    {
14       for ( i = 0; i < number; i++ )
15 //       *(ptr+i) = i;
16          ptr[i] = i;
17       for ( i = number; i > 0; i— )
18          /* print out in reverse order */
19 //       printf("%d\n", *(ptr+(i-1)));
20 /* print out in reverse order */
21          printf("%d\n", ptr[i - 1]);
22       /* free allocated memory */
23       free(ptr);
24       return 0;
25    }
26    else
27    {
28       printf("\nMemory␣allocation␣failed␣—␣"
29              "not␣enough␣memory.\n");
30       return 1;
31    }
32 }
```

7.5.1. Drobne podsumowanie

Drobne podsumowanie

```
1 int T[10];
2 ...
3 ...
4 for (i = 0 ; i < 10 ; i++)
5     T[i] = i;
6 ...
7 for(i = 0 ; i < 10 ; i++)
8     printf("%d\n", T[i]);
```

```
1 int * T =
2     malloc(10 * sizeof(int));
3 ...
4 for (i = 0; i < 10 ; i++)
5     *(T + i) = i;
6 ...
7 for (i = 0 ; i < 10 ; i++)
8     printf("%d\n", T[i]);
```

7.6. Tablice wielowymiarowe

Tablice wielowymiarowe...

— ...również mogą być deklarowane w sposób dynamiczny,
— ...ale jest to bardziej skomplikowane

Tablica o N wierszach i M kolumnach:

```
1 int ** U;
2 ...
3 U = malloc(N * sizeof(int *))
4 for(int i = 0; i < N; i++)
5     U[i] = malloc(M * sizeof(int));
```

7.7. Pamięć dynamiczna: próba podsumowani

Co już wiemy? (Co powinniśmy wiedzieć)

1. Zmienne **automatyczne** to zmienne deklarowane w sposób następujący: `int a, A[10], B[N][M];`
   — są one deklarowane w pamięci stosu (o ograniczonym rozmiarze);
   — deklaracja obowiązuje jedynie w ramach bloku (`{...}`), w którym została wykonana (i bloków niższych);
   — `A` to **stała** typu „wskaźnik na `int`'' (czyli `int *`) zawierająca adres tablicy `A`;
   — adres zmiennej `a` trzeba wyliczyć za pomocą operatora `&`;

   - — tablice (automatyczne) dwu (i więcej wymiarowe) to konstrukcje *skomplikowane...*
   - — zajmują ciągły obszar pamięci (o rozmiarze `N * M * sizeof(typ)`)
   - — mogą być traktowane jako „tablica tablic" — w przypadku tablicy dwuwymiarowej `B` jest to tablica o `N` wierszach, każdy wiersz tej tablicy to tablica o `M` kolumnach, zatem...
   - — ...`B` to również **stała** typu wskaźnikowego, ale typ jest *dziwaczny*: `int (*)[M]` — tablica o `M` elementach — (zwiększenie `B` o jeden w arytmetyce wskaźników zmienia adres tak, by wskazywał na kolejny wiersz tablicy)
2. Zmienne dynamiczne tworzone są z wykorzystaniem funkcji
   - — `void *malloc(size_t size)`
   - — `void *calloc(size_t nmemb, size_t size)`
     w powyższym `void *` to wskaźnik **bez określonego typu**,
     `size_t` to specjalny typ (zbliżony do `int`, ale 64-bitowy, bez znaku),
     `size` rozmiar pamięci (w bajtach),
     `nmemb` — liczba elementów tablicy,
     `size` rozmiar (w bajtach) jednego elementu tablicy;
   - — funkcja `calloc()` zeruje przydzieloną pamięć;
   - — obie funkcje zwracają adres początku tablicy.
3. Tworzenie tablicy jednowymiarowej `T` o `N` elementach jest proste:
   ```
   typ *T;
   T = malloc(N * sizeof(typ));
   ```
   lub
   ```
   typ *T;
   T = calloc(N, sizeof(typ));
   ```
   zawsze jednak trzeba sprawdzić, czy wartość zwracana przez funkcję nie jest równa zero — w tym przypadku pamięć **nie została** przydzielona. Podczas operacji podstawienia (`T =`) dokonywana jest konwersja (rzutowanie) wskaźnika `void` na wskaźnik odpowiedni dla typu zmiennej `T`.
4. Tworzenie tablicy dwuwymiarowej `U` o `N` wierszach i `M` kolumnach jest bardziej skomplikowane
   - — zaczynamy od zadeklarowania zmiennej `U` typu
     `typ ** U;`
     (***jednowymiarowa*** *tablica, której każdy element jest wskaźnikiem typu* ***typ***)

— przydzielamy jej pamięć używając funkcji `malloc()`:
`U = malloc(N * sizeof(typ));`
(i sprawdzamy, czy zwrócona wartość jest różna od zera)
tworzy się struktura, która będzie użyta do przechowywania adresów początkowych każdego wiersza,
— następnie przydzielamy pamięć dla każdego wiersza tablicy:
```
for (int i = 0; i < N; i++)
    U[i] =  malloc(M * sizeof(typ));
```
a uzyskane adresy wpisujemy do kolejnych komórek tablicy `U`;
(przypominam o konieczności każdorazowego sprawdzenia czy wartość zwracana przez `malloc()` jest różna od zera).

5. W każdym przypadku `B[i]` oraz `U[i]` definiują adres `i`-tego wiersza w pamięci operacyjnej; ale w każdym przypadku wartość ta jest inaczej **wyliczana**:
— w przypadku tablicy `B` jest to adres `B` zwiększony (w arytmetyce wskaźników) o `i`,
— w przypadku tablicy `U`, jest to zawartość `i`-tej komórki tablicy `U`;
kolejna operacja służąca „dobraniu" się do `j`-tego elementu wiersza tablicy `(B[i])[j]` czy `(U[i])[j]` wykonywana jest tak samo: adres zwiększany jest korzystając z arytmetyki wskaźników.

Drobne podsumowanie

Ciąg dalszy

Wyobraźmy sobie, że mamy następującą definicję funkcji:

```
1 int f(int n, double b, int c[ ], double * d,
2                int e[ ][n], double ** g);
```

1. Funkcja ma sześć (6) parametrów.
2. Funkcja zwraca wartości całkowite *(to jest nieistotne)*.
3. Podczas wywołania funkcji:
— pierwszym argumentem może być: stała, zmienna lub wyrażenie typu całkowitego;
— drugim argumentem może być: stała, zmienna lub wyrażenie typu podwójnej precyzji;
W przypadku gdy typ pierwszych dwu argumentów będzie inny niż zadeklarowany — zostanie dokonana odpowiednia konwersja.

wer. 13 z drobnymi modyfikacjami!

4. Trzecim (i czwartym) argumentem powinien być wskaźnik do (adres) tablicy typu **int** (**double**). *Żadne konwersje nie będą wykonywane gdy argumentem będzie wskaźnik innego typu. Należy się spodziewać najgorszych możliwych efektów w takim przypadku.*

   Przykład

```
1 int t[10];
2 int * u = malloc( 40 );
```

   Argumentem może być:
   — t
   — u
   — &t[0] albo nawet &t[1] czy alternatywnie u + 1,

5. W przypadku argumentu piątego (**int** e[ ][n] parametrem wywołania funkcji może być wyłącznie tablica automatyczna lub statyczna (typu **int**) zadeklarowana jako:

```
1 int v[m][n]
```

   gdzie m i n to jakieś stałe. *(Wówczas pierwszy parametr wywołania funkcji (n) powinien mówić o liczbie kolumn!)*
6. Ostatnim argumentem funkcji (podwójny wskaźnik) powinna być tablica dynamiczna typu **double** w, zadeklarowana jakoś tak:

```
1 double ** w;
2 int m, n, i;
3 w = (double **) malloc(n * sizeof( double *));
4 for(i = 0; i < n; i++)
5 w[i] = (double *) malloc(m * sizeof(double));
```

   o n wierszach i m kolumnach.
   (Uwaga: Wszędzie powinno być dodane sprawdzenie, czy funkcja malloc nie zwróciła wartości zero (NULL).

7.8. Wskaźniki do funkcji

Wskaźniki do funkcji

2023-04-12 11:02:16 +0200

```c
#include <stdio.h>

int add( int x, int y );  /* declare function */

int main() {
  int x=6, y=9;
  int (*ptr)(int, int); /* declare pointer to function*/

  ptr = add; /* set pointer to point to "add" function */

  printf( "%d plus %d equals %d.\n", x, y, (*ptr)(x,y) );
  /* call function using pointer */
  return 0;
}

int add( int x, int y ) { /* function definition */
 return x+y;
}
```

Wskaźniki do funkcji

1. Pamiętać należy aby w deklaracji wskaźnik zamknąc w nawiasach.
2. Pamiętać należy aby typ wskaźnika funkcji odpowiadał typowi wartości zwracanych przez funkcję.
3. Po co to?
   — Żeby studentom było trudniej.
   — Żeby móc przekazać do funkcji parametr będący funkcją.
   — ...

Zamiast zakończenia proponuję przeanalizować żarcik zamieszczony na rysunku 7.1. W zasadzie każdy jego fragment ma jakieś znaczenie.

7.9. Przykładowe programiki

Wszystkie zamieszczone dalej przykładowe programy pochodzą z [9].

7.10. Dla dociekliwych

(Za http://orcik.net/programming/pointers-in-assembly-language/)

Rysunek 7.1. Zamiast zakończenia żarcki

```
1  /* Program 1.1 from PTRTUT10.TXT   6/10/97 */
2
3  #include <stdio.h>
4
5  int j, k;
6  int *ptr;
7
8  int main(void)
9  {
10     j = 1;
11     k = 2;
12     ptr = &k;
13     printf("\n");
14     printf("j has the value %d and is stored at %p\n", j,
15             (void *)&j);
16     printf("k has the value %d and is stored at %p\n", k,
17             (void *)&k);
18     printf("ptr has the value %p and is stored at %p\n", ptr,
```

```
19              (void *)&ptr);
20      printf("The value of the integer pointed to by ptr is %d\n",
21              *ptr);
22
23      return 0;
24  }
```

Dla dociekliwych

Indeks tablicy i wskaźnik do elementu

```
1  /* Program 2.1 from PTRTUT10.HTM   6/13/97 */
2
3  #include <stdio.h>
4
5  int my_array[ ] = {1,23,17,4,-5,100};
6  int *ptr;
7
8  int main(void)
9  {
10     int i;
11     ptr = &my_array[0];       /* point our pointer to the first
12                                  element of the array */
13     printf("\n\n");
14     for (i = 0; i < 6; i++)
15     {
16       printf("my_array[%d] = %d   ", i, my_array[i]);    /*<-- A */
17       printf("ptr + %d = %d\n", i, *(ptr + i));          /*<-- B */
18     }
19     return 0;
20 }
```

Dla dociekliwych

Napisy

```
1  /* Program 3.1 from PTRTUT10.HTM   6/13/97 */
2
3  #include <stdio.h>
4
5  char strA[80] = "A string to be used for demonstration purposes";
6  char strB[80];
7
8  int main(void)
9  {
```

```
10
11      char *pA;       /* a pointer to type character */
12      char *pB;       /* another pointer to type character */
13      puts(strA);     /* show string A */
14      pA = strA;      /* point pA at string A */
15      puts(pA);       /* show what pA is pointing to */
16      pB = strB;      /* point pB at string B */
17      putchar('\n');          /* move down one line on the screen */
18      while(*pA != '\0')      /* line A (see text) */
19      {
20          *pB++ = *pA++;      /* line B (see text) */
21      }
22      *pB = '\0';             /* line C (see text) */
23      puts(strB);             /* show strB on screen */
24      return 0;
25  }
```

Dla dociekliwych

Dziwactwa

```
#include <stdio.h>
int main(void)

   char a[20];
   int i;
   a[3] = 'x';
   3[a] = 'x';
   printf("%c   %c\n", a[3], 3[a]);
   return 0;
```

O co chodzi?

— Skoro a[i] jest równoważne ∗(a + i)

— Skoro dodawanie jest przemienne... to jest równoważne ∗(i + a)

— A zatem równoważne i[a]???

Co robi ten program

```
1  /*
2   cc -Aa +O3 zapchaj.c -o zapchaj
3   gcc -O3 zapchaj.c -o zapchaj
4  */
```

2023-04-12 11:02:16 +0200

```
5
6  #include <stdlib.h>
7
8  int main (int cnt, char ** arg)
9  {
10  char *tab;
11
12  long
13    bytes,
14    loops;
15
16  long
17   i,
18   n;
19
20  if (cnt!=3)
21   {
22    printf("usage:\n\t%s bytes loops\n", arg[0]);
23    return(-1);
24   }
25
26  bytes = atol(arg[1]);
27  loops = atol(arg[2]);
28
29  tab = malloc(bytes*sizeof(char));
30
31  if (tab==0)
32   {
33    printf("%s:\n\tcannot allocate %ld bytes\n",arg[0],bytes);
34    return(-2);
35   }
36
37  for (i=1; i<=loops; i++)
38   {
39    printf("%ld\n",i);
40    for (n=0; n<bytes; n++) tab[n]=(char)n;
41   }
42
43  return(0);
44 }
```

8. Struktury danych, unie

8.1. Struktury danych

1. W Pascalu — rekordy.
2. Struktura „podobna” do tablicy, ale pozwalająca na przechowywanie danych różnych typów.
3. Przykłady:
   — Lista płac:
      — Imię
      — Nazwisko
      — PESEL
      — Numer konta bankowego
      — Kwota do wypłaty
      — ...
   — Współrzędne punktu:
      — współrzędna X
      — współrzędna Y
      — (ewentualnie) współrzędna Z

8.1.1. Przykład: ułamki

Zacznijmy od prostego przykładu. Jego celem jest pokazanie, że struktury danych istotnie rozszerzają możliwości języka programowania. Wszystko co można zaprogramować z uzyciem struktur może być również zrealizowane jnaczekjm, ale skorzystanie ze struktur ma istotne zalety: upraszcza programowanie.

Zadanie

— Chcemy (musimy?) napisać program–kalkulator wykonujący obliczenia na ułamkach „zwykłych”.

— Ułamki takie to liczby postaci:

$$\frac{p}{q}$$

gdzie $p, q \in N$ oraz $q \neq 0$.

— Kalkulator będzie wykonywał operacje dodawania, odejmowania, mnożenia i dzielenia.

— Do każdej operacji musimy napisać funkcję.

— Dodatkowo potrzebne będą funkcje upraszczania ułamków i ich drukowania. Chcemy żeby wydruk ułamków wyglądał jak najbardziej naturalnie:

```
     17
-35---
    123
```

Sposób przechowywania danych

1. Pierwszą sprawą wymagającą naszej uwagi jest sposób przechowywania danych.
2. Najwygodniej (czemu?) przechowywać wszystkie dane, na których prowadzimy operacje jako ułamki (niewłaściwe)
3. Wydaje się, że najwygodniej będzie dane przechowywać jako dwie liczby:
   — licznik przechowujący również znak liczby
   — mianownik
4. Po każdej operacji ułamek będzie normalizowany (upraszczany).

Operacje

— dodawanie/odejmowanie

$$\frac{a}{b} \pm \frac{c}{d} = \frac{a * d \pm c * b}{b * d}$$

— mnożenie

$$\frac{a}{b} * \frac{c}{d} = \frac{a * c}{b * d}$$

— dzielenie

$$\frac{a}{b} / \frac{c}{d} = \frac{a * d}{b * c}$$

wer. 8 z drobnymi modyfikacjami!

Uwagi

Ponieważ licznik przechowuje znak — w wyniku dzielenia może okazać się, że mianownik stanie się ujemny. Trzeba to sprawdzić i skorygować.

Do rozstrzygnięcia pozostaje kwestia działań podejmowanych gdy mianownik będzie równy zeru.

Naiwna próba realizacji

— Napiszmy funkcję suma.
— argumenty: wartości a, b, c, d (zmienne typu **int**)
— wynik: nie będzie mógł być przekazany poleceniem **return** — można tak przesłać tylko jedną zmienną czyli:
  — albo piszemy dwie funkcje do wyliczania wartości licznika i mianownika będziemy potrzebowali cztery funkcje do wyliczania liczników dla sumy, różnicy, iloczynu i ilorazu oraz dwie funkcje do wyliczania mianownika: jedną dla sumy, różnicy i iloczynu, oraz drugą dla ilorazu;
  — albo wyniki przekazujemy „przez adres” jako wskaźniki

  Drugie rozwiązanie wydaje się bardziej racjonalne

Suma

```
1 void suma ( int a, int b, int c, int d, int *p, int *q )
2 {
3  *p = a * d + c * b;
4  *q = b * d;
5 }
```

Użycie:

```
1 int a, b, c, d, e, f;
2 ...
3 suma (a, b, c, d, &e, &f);
```

Czy można inaczej?

1. Możliwość posiadania zmiennej, która będzie mogła przechowywać więcej niż jedną wartość ułatwiłaby nasz problem.
2. Taki typ zmiennych istnieje w języku C i nazywa się **strukturą**.
3. Deklarujemy najpierw strukturę danych:

```
1 struct Ulamek
2 {
3  int l;
4  int m;
5 };
```

4. Używamy polecenia **typedef** aby łatwiej używać zmiennych

```
1 typedef struct Ulamek ulamek;
```

i możemy teraz deklarować zmienne w sposób następujący:

```
1 ulamek A, B, C;
```

Zalety i wady takiego postępowania

+ Mamy zmienną mogącą przechowywać więcej niż jedną wartość.
+ Przechowywane wartości mogą być różnego typu.
+ Ponieważ korzystamy z nowego typu — funkcje mogą używać tego typu do we wszelki obliczeniach.
– Nie można wykonywać operacji na takich zmiennych złożonych.
– Dostęp do składowych struktury jest nieco bardziej skomplikowany;
– Nie można (łatwo) nadawać wartości zmiennym tego typu (nie istnieją stałe typu strukturalnego).

Nowe sformułowanie funkcji suma

```
1 struct Ulamek
2 {
3  int l;
4  int m;
5 };
6 typedef struct Ulamek ulamek;
7 ulamek suma ( ulamek A, ulamek B, ulamek Wynik )
8 {
9  Wynik.l = A.l * B.m + B.l * A.m;
10  Wynik.m = A.m * B.m;
11  return Wynik;
12 }
13 ...
```

wer. 8 z drobnymi modyfikacjami!

```
14 ulamek aa, bb, cc;
15 ...
16 cc = suma ( aa, bb );
```

Co dalej?

1. Możemy wykonywać operacje złożone (ulamek aa, bb, cc;):

```
1 cc = iloczyn(suma(aa, bb), roznica(aa, bb))
```

2. Deklarując zmienną, można nadać jej wartość (ale tylko tu!):

```
1 ulamek aa = {1, 2}; bb = {3, 4}, cc;
```

3. Ma sens coś takiego:

```
1 int z = suma(aa, bb).l;
```

8.1.2. Deklaracja

1. Deklaracja struktury, tak na prawdę, to deklaracja nowego typu danych!

```
1 struct point {
2     int x;
3     int y;
4 };
```

2. x i y to **składowe** struktury.
3. Deklaracja zmiennej:

```
1 struct point punkt1, punkt2, punkt3, pt;
```

4. Można też tak:

```
1 struct point3D {
2     int x;
3     int y;
4     int z;
5 } p1, p2, p3;
```

Można też w dwu etapach (jak w ułamku):
a) najpierw definiujemy strukturę,
b) później definiujemy nowy typ używając **typedef**.

8.1.3. Użycie

1. Inicjalizacja:

```
1 struct point maxpt = { 320, 200 };
```

2. W wyrażeniach dostęp uzyskuje się *nazwa_struktury.składowa*
3. Kropka to **operator** składowej struktury.
4. Przykłady:

a)
```
printf("%d,%d", punkt1.x, punkt1.y);
```

b) Odległość między punktami

```
1 double dist, sqrt(double); /* sqrt: pierwiastek */
2 dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.
```

8.1.4. Przykład

— Zagnieżdzona struktura

```
1 struct rect {
2     struct point pt1;
3     struct point pt2;
4 };
```

Gdy zadeklarujemy screen:

```
1 struct rect screen;
2 screen.pt1.x = 100;
```

nadaje wartośc współrzędnej x pierwszego punktu (pt1) struktury ekran
— Inicjacja struktury złożonej

```
1 struc rect ekran = { { 0, 0 }, { 1024, 768 }};
```

8.2. Struktury i funkcje

1. Funkcja może zwracać daną typu strukturalnego.

```
/* makepoint: utworz punkt ze
              wspolrzednych x i y */
struct point makepoint(int x, int y)
{
   struct point temp;
   temp.x = x;
   temp.y = y;
   return temp;
}
```

Użycie punkt2 = makepoint(100, 59);

2. Parametrami funkcji mogą być dane typu strukturalnego

```
/* addpoint: dodaj dwa punkty */
struct point addpoint(struct point p1, \
                      struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

użycie:

```
int main (void)
{
        struct rect ekran = { { 0, 0 }, { 1024, 768 } };
        struct point srodek;
        srodek = makepoint(addpoint(ekran.pt1,
                        ekran.pt2).x/2,\
                        addpoint(ekran.pt1,
                        ekran.pt2).y/2);
        printf("(%d,%d)", srodek.x, srodek.y);
        return 0;
}
```

8.3. Struktury i wskaźniki

1. W sytuacji gdy do funkcji jako daną mamy przekazać bardzo rozbudowaną strukturę (będzie ona **kopiowana** do zmiennej tymczasowej) lepiej jest użyć wskaźnika.

2. Deklaracja jak zwykle:

```
1 struct point origin, *pp;
```

   origin to struktura (o składowych x i y), pp to wskaźnik do struktury typu point; (*pp).x oraz (*pp).y to jej składowe.

3. Użycie

```
1 pp = &origin;
2 printf("punkt␣poczatkowy␣(%d,%d)\n", \
3                          (*pp).x, (*pp).y);
```

4. Ponieważ struktur używa się bardzo często podobnie jak wskaźników do nich, zeby uprościć życie wymyślono specjalną notację. Jeżli p jest wskaźnikiem do struktury to p−>skladowa−struktury. Zatem poniższy zapis jest równoważny poprzedniemu:

```
1 printf("punkt␣poczatkowy␣(%d,%d)\n", \
2                              pp->x, pp->y);
```

5. W przypadku struktur zagnieżdżonych sprawa nieco się komplikuje (to samo można zapisać na kilka różnych sposobów!):

```
1 struct rect r, *rp = &r;
```

   Następujące wyrażenia są równoważne:

```
1 r.pt1.x
2 rp->pt1.x
3 (r.pt1).x
4 (rp->pt1).x
```

6. Operatory strukturowe ., −> oraz nawiasy okrągłe () wywołania funkcji oraz nawiasy kwadratowe [] indeksowania tablicy mają najwyższy priorytet (najsilniej wiążą swoje argmenty).

8.4. Tablice struktur

1. Podobnie jak z danych innych typów, również ze struktur można tworzyć tablice (bazy danych???)

```
21 struct point dane[100];
```

wer. 8 z drobnymi modyfikacjami!

deklaruje tablicę o 100 elementach, z których każdy to dana typu point — struktura złożona z dwu składowych x i y.

8.4.1. Przykład

1. Piszemy program zliczający częstość występowania słów kluczowych języka C.
2. Dane: tablica, każdym jej elementem jest para (słowo kluczowe, liczba wystąpień) — struktura.
3. Schemat blokowy:
   a) Jeżeli koniec pliku — **skończ**.
   b) Wprowadź słowo (z pliku).
   c) Znajdź słowo w danych.
   d) Jezeli wystąpiło — zwiększ odpowiedni licznik.
   e) (Jeżeli nie wystąpiło — przechodzimy dalej.)
   f) Przejdź na początek.

Sam program znajduje się w [10]:

1. Zakładamy, że słowa kluczowe uszeregowane są w kolejności rosnącej — ułatwi to wyszukiwanie.
2. Do wyszukiwania wykorzystamy metody szukania binarnego (podział na pół).
3. Ile miejsca w pamięci (bajtów) zajmuje struktura key?

```
1  struct key {
2      char *word;
3      int count;
4  } keytab[] = {
5      "auto", 0,
6      "break", 0,
7      "case", 0,
8      "char", 0,
9      "const", 0,
10     /* ... */
11     "volatile", 0,
12     "while", 0
13 };
```

8.5. Pola bitowe

1. Pamięć jest tania i zazwyczaj nie ma na potrzeby jej oszczędzać.
2. Czasami chcemy jednak operować an bitach.
3. Definiujemy sobie coś takiego:

```
1 struct {
2     unsigned int is_keyword : 1;
3     unsigned int is_extern  : 1;
4     unsigned int is_static  : 1;
5 } flags;
```

   ta 1 (po dwukropku) do długość pola w bitach.
4. Dostajemy zmienną o nazwie flags zawierającą trzy jednobitowe pola.
5. Bardzo łatwo możemy manipulować bitami (zerować je, ustawiać i sprawdzać ich wartość), na przykład:

```
1 flags.is_extern = flag.is_static = 0;
```

8.6. Unie

1. Na pierwszy rzut oka deklaracja unii jest bardzo podobna do deklaracji struktury:

```
1 union uni {
2     int val;
3     float fval;
4     char sval[4];
5 } zmienna;
```

2. Podobieństwo jest jednak złudne.
3. Unia to obszar pamięci, w którym zapisać można wartości różnych typów; kompilator dba o przydział odpowiednio dużego obszaru pamięci (mieszczącego „największy” z podnych typów).
4. Użycie: zmienna.val = 5 albo x = zmienna.fval
5. Wykorzystanie...

8.7. Delaracja nowego typu

1. Standardowy zestaw typów w języku C jest niezbyt bogaty (**int**, **float**, **double**, **char**, **struct**).
2. Niektóre typy mogą być modyfikowane za pomocą słów **short**, **long**, **unsigned**.
3. Język C pozwala na definiowanie „aliasów” dla struktur istniejących (ułatwiają one konstruowanie programów). Służy do tego instrkcja **typedef**:

```
1 typedef int Lenght;
```

   deklarująca **nowy** typ danych o nazwie **Lenght** (służący do deklarowania wszystkich zmiennych związanych z długością):

```
1 Lenght len, maxlen, minlen;
```

4. Możemy w ten sposób stworzyć sobie typ zespolony:

```
1 typedef struct { double r, theta; } Complex;
```

   albo

```
1 typedef struct { double real, imaginary; }
2                Complex;
```

5. Albo cośą takiego:

```
1 typedef char* string;
```

   (Do czego może się to przydać?)

   Tu nieśmiały przykład zastosowania ([11]):

```
1 #include <stdio.h>
2 typedef char* string;
3 int main(){
4         string list[] = {
5                 "first", "second", "third", NULL
6         };
7         string *p;
8         for (p = list; *p != NULL; p++) {
9                 printf("%s\n", *p);
10        }
11        return 0;
12 }
```

Jak to działa? Jak zmienia się p?

Deklaracje typów

```
1 typedef struct {
2       char forename[20];
3       char surname[20];
4       float age;
5       int childcount;
6       } person;
```

1. Definiujemy nowy typ o nazwie **person**.
2. Jest to struktura danych (rekord?) o zawartości: imię, nazwisko, wiek, liczba potomstwa.
3. Deklaracja poszczegolnych zmiennych staje się bardzo prosta:

```
1 person OsobaI, OsobaII, OsobaIII;
2 int a, b, c;
3 double x, y, z;
```

9. Wejście/wyjście

9.1. Strumienie

1. W czasach przed-uniksowych program wykonujący operacje wejścia/wyjścia musiał „podłączyć” wszystkie urządzenia, z których chciał (musiał) korzystać.
2. Unix wprowadził abstrakcyjne urządzenia wejścia/wyjścia zwane strumieniami.
3. Strumień to uporządkowany ciąg bajtów, które mogą być odczytywane jeden po drugim aż do napotkania znacznika końca.
4. Unix wprowadził również ideę automatycznego uaktywniania tych strumieni. (Wcześniejsze systemy operacyjne wymagały wykonywania pewnych, czasami skomplikowanych, czynności.)
5. Standardowo program ma do dyspozycji trzy strumienie:
   a) Standardowe wejście (*standard input*) — **stdin**,
   b) Standardowe wyjście (*standard output*) — **stdout**,
   c) Standardowy strumień błędów (standard error) — **stderr**.

Idea strumieni została zaakceptowana przez inne języki programowania. W szczególności obecna jest w systemie DOS czy Windows. W tym ostatnim (podobnie jak i we współczesnych uniksach czy linuksach) straciła na znaczeniu ze względu na powszechne używanie graficznych interfejsów użytkownika (GUI).

W dalszej części podaję podstawowe fakty na temat każdego strumienia.

Standardowe wejście

— Pewno powinno się mówić *standardowy strumień wejścia...*
— Standardowo dane pobierane są z terminala, z którego został uruchomiony program.
— Strumień może być „przekierowany” (*redirection*).
— Nie wszystkie programy z niego korzystają.

— Używany do wprowadzania informacji do programu.
— Deskryptor 0.

Standardowe wyjście

— Pewno powinno się mówić *standardowy strumień wyjścia...*
— Standardowo dane wysyłane są na terminal, z którego został uruchomiony program.
— Strumień może być „przekierowany" (*redirection*).
— Nie wszystkie programy z niego korzystają.
— Służy do wyprowadzania informacji z programu.
— Deskryptor 1.

Standardowy strumień błędów

— Standardowo dane wysyłane są na terminal, z którego został uruchomiony program.
— Strumień może być „przekierowany" (*redirection*).
— Dobrze napisane programy kierują tam informacje o błędach i komunikaty diagnostyczne.
— Służy do informowania operatora o błędach.
— Deskryptor 2.

Informacje w standardowym strumieniu błędów pojawiają się w nim zapewne wbrew woli programisty (nikt nie chce, żeby program generował błędy).

9.1.1. Przekierowanie strumieni

— Przekierowania to funkcja środowiska, w którym uruchamiany jest program.
— stdout do pliku (poprzednia zawartość pliku ulega zniszczeniu)

```
1 program >a.txt
```

— stdout do pliku (w trybie dopisywania)

```
1 program >>a.txt
```

— stderr do pliku

```
1 program 2>a.txt
```

— stderr oraz stdin do tego samego pliku

```
1 program >a.txt 2>&1
```

(najpierw stdout łączy z plikiem, następnie stderr łączy z **aktualnym** stdout)

— Z pliku do stdin

```
1 program < b.txt
```

— Z stdout (jednego programu) do stdin (drugiego)

```
1 program1 | program2
```

Jednym z najprostszych programów systemu Unix jest program cat. Może być użyty do łączenia plików, ale również w nieco prostszych zastosowaniach. Uruchomiony z „linii poleceń" program cat czyta iinformacje ze standardowego wejścia i w niezmienionej postaci powtarza je na standardowym wyjściu.

Poniższy przykład pokazuje użycie programu do skopiowania zawartosci z jednego pliku (a.txt) do innego pliku (b.txt).

```
1 cat <a.txt >b.txt
```

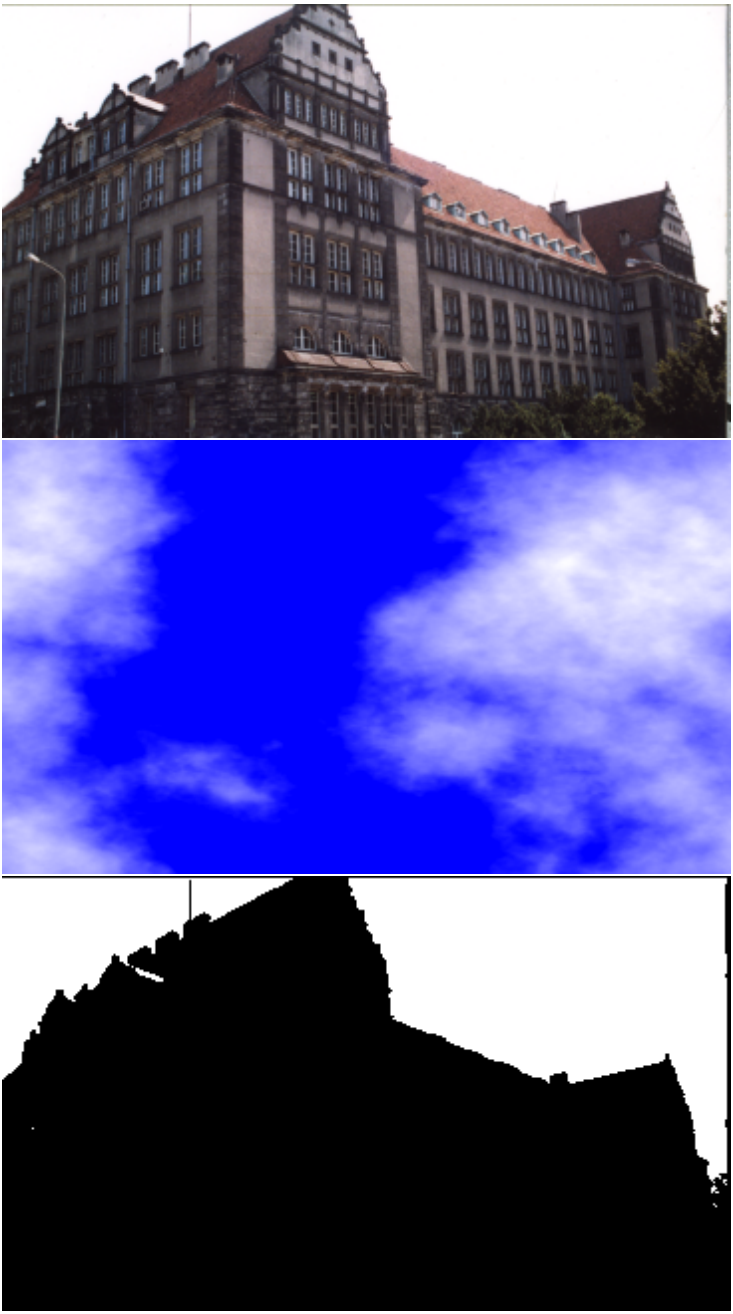
Natomiast użycie tego polecenia w sposób następujacy:

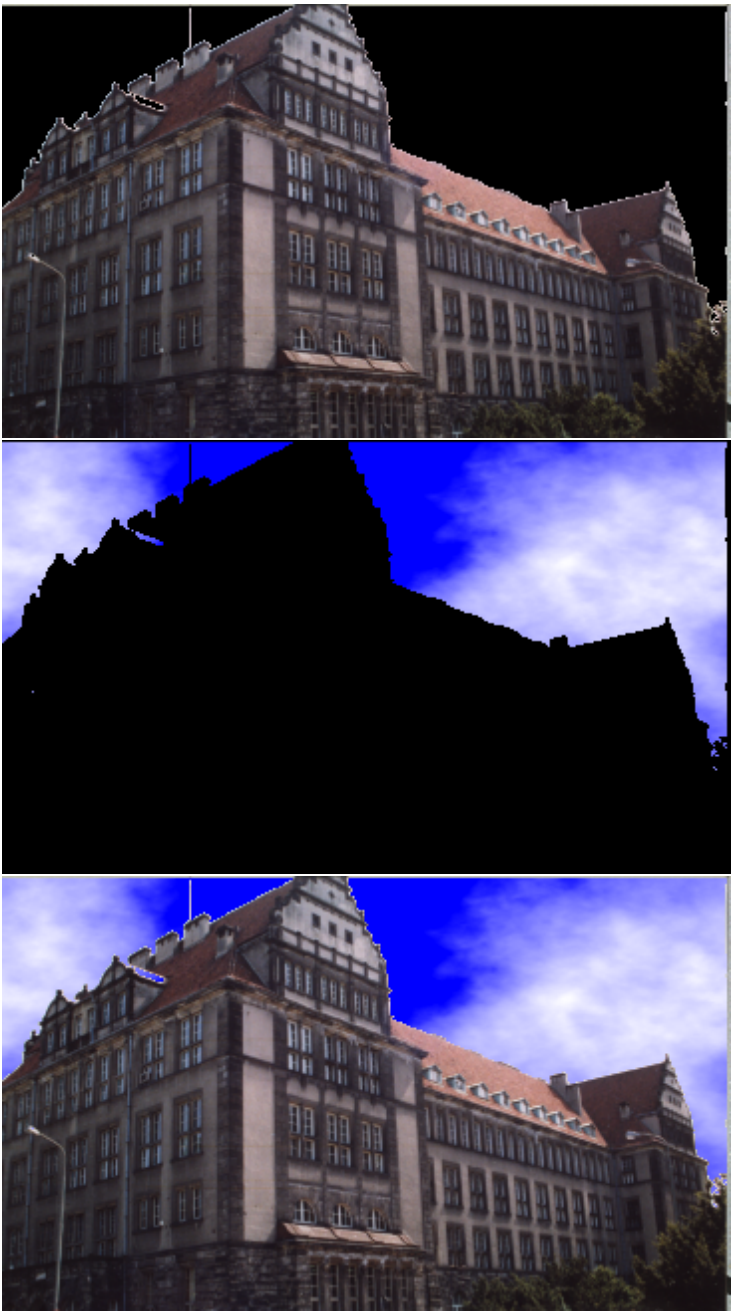
```
1 cat a.txt | sort > b.txt
```

kopiuje plik a.txt na standardowe wejście pliku programu sort, a ułożone w kolejnosci alfabetycznej dane zapisuje do pliku b.txt.

Dziwny przykład

1. Mamy obrazek
2. Chcemy dodać chmurki
3. Tworzymy „maskę"...
4. ...którą nakładamy na obrazek...
5. ...i na chmurki
6. Otrzymane obrazki sumujemy





Jak to jest zrobione?

```
1 echo pbmmask
```

2022-03-01 12:56:37 +0100

```
pngtopnm obj.png >obj.ppm
ppmforge -clouds -width 366 -height 218 >dest.ppm
ppmtopgm obj.ppm | pgmtopbm -threshold -v 0.85| pbmmask\
 |pnminvert> objmask.pbm
pnmarith -multiply dest.ppm objmask.pbm > t1.ppm
pnminvert objmask.pbm | pnmarith -multiply obj.ppm - > t2
pnmarith -add t1.ppm t2.ppm >t3.ppm
pnmcat -lr obj.ppm dest.ppm >a1.ppm
pnmcat -lr objmask.pbm t1.ppm >a2.ppm
pnmcat -lr t2.ppm t3.ppm >a3.ppm
pnmcat -tb a1.ppm a2.ppm a3.ppm >przyklad2.ppm
rm a1.ppm
rm a2.ppm
rm a3.ppm
rm objmask.pbm
rm t1.ppm
rm t2.ppm
rm t3.ppm
rm obj.ppm
rm dest.ppm
pnmtopng przyklad2.ppm >przyklad2.png
```

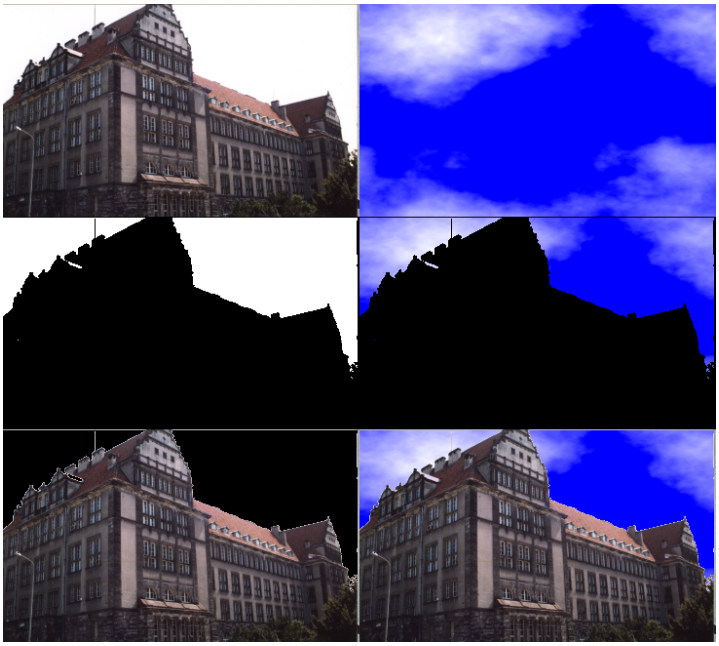
Poszczególne etapy przetwarzania zaprezentowane są na rysunku 9.1. Rysunek ten również został uzyskany za pomocą programów z prezentowanego pakietu (linie począwszy od dziewiątej).

9.2. Podstawowe instrukcje wyjścia

1. **#include**<stdio.h>
2. **printf()**

```
#include <stdio.h>
int main(void)
{
   printf("Hello world!\n");
   return 0;
}
```

wer. 16 z drobnymi modyfikacjami!



Rysunek 9.1. Ilustracja pokazująca poszczególne etapy przetwarzania

Wersja ogólna:

```
1 printf(format, argument1, argument2, ...);
```

Format to napis ujęty w cudzysłowy określający sposób wyświetlania informacji. Format wyświetlany jest tak jak go zapiszemy z wyjątkiem pewnych specjalnych znaków, które są zamieniane na coś innego. Znak % ma znaczenie specjalne (podobnie jak znak \).

```
1 printf("Procent: %% Backslash: \\");
```

Najczęstsze użycie printf():
— printf("%d", i); gdy i jest typu **int**; zamiast %d można też użyć %i,
— printf("%f", i); gdy i jest typu **float** lub **double**,
— printf("%Lf", i); gdy jest typu **long double**,
— printf("%c", i); gdy i jest typu **char** (i chcemy wydrukować znak)
— printf("%s", i); gdy i jest napisem (typu **char***)
Format może być zmienną! *Funkcja zwraca liczbę znaków w tekście (nie licząc znaku \0 kończącego tekst) w przypadku sukcesu lub znak EOF w przypadku błędu.*

3. **puts()**

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   puts("Hello world!");
6   return 0;
7 }
```

Funkcja po prostu kopiuje tekst zawarty w argumencie (może być zmienna!) do standardowego strumienia wyjścia dodając na końcu znak przejścia do nowej linii. *Funkcja zwraca liczbę nieujemną w przypadku sukcesu lub EOF w przypadku błędu.*

4. **putchar()** Funkcja służy do wyprowadzenia pojedynczego znaku do strumienia stdio. *Funkcja zwraca kod znaku traktowany jako unsigned char przekształcony do typu int; w przypadku błędu funkcja zwraca wartość EOF.*

```
1 #include<stdio.h>
2 int main (void)
3 {
4     int i;
5     for (i = 'a'; i <= 'z'; ++i)
6         putchar (i);
7     return 0;
8 }
```

9.3. Podstawowe instrukcje wejścia

1. **#include** <stdio.h>
2. **scanf()**

```
1 #include <stdio.h>
2 int main()
3 {
4     int liczba = 0;
5     printf("Podaj liczbe: ");
```

```
6       scanf("%d", &liczba);
7       printf("%d*%d=%d\n", liczba, liczba,
8               liczba * liczba);
9       return 0;
10  }
```

Zwracam uwagę na znak & przy drugim argumencie funkcji!
Typowe użycie:
— scanf("%i", &liczba); wczytuje liczbę typu int,
— scanf("%f", &liczba); — liczbę typu float,
— scanf("%lf", &liczba); — liczbę typu double,
— scanf("%s", tablica_znakow); ciąg znaków[1].
Zwracam uwagę na brak znaku & w ostatnim przypadku — gdy **nazwa tablicy** pojawia się jako argument funkcji automatycznie przekazywany jest adres. *Funkcja zwraca liczbę poprawnie wczytanych zmiennych lub EOF jeżeli nie ma już danych w strumieniu lub nastąpił błąd.*

```
1 #include <stdio.h>
2 int main(void)
3 {
4       int a, b;
5       while (scanf("%d␣%d", &a, &b) == 2) {
6           printf("%d\n", a + b);
7       }
8       return 0;
9 }
```

Co robi ten program:

```
1 #include <stdio.h>
2 int main(void)
3 {
4       int result, n;
5       do {
6           result = scanf("%d", &n);
7           if (result == 1) {
8               printf("%d\n", n * n * n);
```

[1] Funkcja scanf nie jest najlepszym sposobem wczytywania napisów. Korzystać z niej należy bardzo ostrożnie. Funkcja nie sprawdza czy w tablicy do której chcemy wczytać teks jest wystarczająco dużo miejsca.

```
9           } else if (!result) { /* !result to to
10                                  samo co result == 0 */
11              result = scanf("%*s");
12          }
13      } while (result != EOF);
14      return 0;
15  }
```

Oto wynik działania programu

```
ala ma kota
2
8
ola ma s3
3derft
27
sdsdsd
pi
1234pies
1879080904
```

3. **gets()**
   — Funkcja służy do wczytywania linii tekstu.
   — Nie należy jej używać...
   — ...gdyż funkcja nie sprawdza, czy jest miejsce do zapisu w tablicy

```
1 #include <stdio.h>
2 int main(void)
3 {
4     char napis[50], *n;
5     n = gets(napis);/* jesli pierwsza linia
6                        ma wiecej niz 49
7                        znakow nastapi
8                        przepelnienie bufora */
9     if(n != NULL)
10        printf("%s\n",napis);
11    else
12        printf("blad odczytu");
13    return 0;
14 }
```

wer. 16 z drobnymi modyfikacjami!

4. **getchar()**

```c
#include <stdio.h>
int main(void)
{
  int c;
  while ((c = getchar()) != EOF) {
    if (c == ' ') {
      c = '_';
    }
    putchar(c);
  }
  return 0;
}
```

Bardzo prosta funkcja czytająca (pobierająca) jeden znak z stdio.

— Ze względu na specyfikę komunikacji (istnienie bufora) funkcja nie jest w stanie „zauważyć" pojedynczego naciśnięcia klawisza; zwraca informację gdy naciśnięty zostanie klawisz Enter lub bufor się przepełni.
— Kod naciśniętego klawisza Enter też trafia do bufora!
— Gdy nastąpi błąd lub nie ma już danych funkcja zwraca EOF.
— EOF to zazwyczaj -1
— Funkcja zwraca kod pobranego znaku traktowany jako **unsigned char** przekształcony do typu **int**

```c
#include<stdio.h>
int main()
{
   int i;
   i = getchar();
   printf("przeczytano znak o numerze %d", i);
   return 0;
}
```

9.4. Pliki

1. Właściwie nie ma wielkiej różnicy w (podstawowej) komunikacji ze standardowymi strumieniami wejścia i wyjścia a plikami.
2. Język C zna dwa sposoby komunikowania z plikami:
   — wysokopoziomowy,
   — niskopoziomowy (patrz rozdział 9.6).
3. Nazwy procedur pierwszej grupy zaczynają się na literę „f” (fopen(), fread(), fclose())
4. Identyfikatorem pliku jest wskaźnik do struktury danych deklarowanej jako FILE
5. Niskopoziomowe operacje to read(), open(), write() i close()
6. Identyfikatorem pliku jest liczba całkowita jednoznacznie identyfikująca plik (w systemie Unix — deskryptor pliku)
7. *Nie należy tych funkcji mieszać!*

Wskaźnik FILE

1. Plik stdio.h zawiera definicję (typedef) typu FILE.
2. W pierwszej kolejności musimy zadeklarować zmienną, która będzie przechowywałą adres początkowy odpowiedniej struktury danych:

```
1 FILE * fp;
```

3. Zmienne tego typu przechowują wszystkie informacje na temat pliku.
4. Na ogół nie ma potrzeby odwoływać się do poszczególnych pól tej struktury danych bezpośrednio.
5. Funkcje zdefiniowane w stdio.h powinny w zupełności wystarczyć.

9.4.1. Otwarcie pliku

1. Przed skorzystaniem z pliku należy go „otworzyć” (*open*).
2. Standardowa biblioteka (stdio.h) zawiera trzy funkcje fopen, freopen i fclose.
3. Pierwsze dwie służą do „skojarzenia” (powiązania) pliku z odpowiednią strukturą danych.
4. Ostatnia — powiązanie to likwiduje.
5. Wszystkie czynności związane z otwieraniem i zamykaniem plików realizowane są przez System Operacyjny.

1. Funkcja **fopen()** otwiera plik o podanej nazwie we wskazanym trybie, tworzy strukturę danych opisującą go i zwraca do niej adres. W przypadku błędu — zwraca NULL.

```
fp = fopen("plik.txt", "r")
```

2. Drugi parametr funkcji to tryb otwarcia pliku:
   — "r" — odczyt
   — "w" — zapis
   — "r+" — odczyt i zapis (najpierw czytamy)
   — "w+" — zapis i odczyt (najpierw piszemy)
   — "a" — zapis na końcy pliku (*append*, dodawanie)
   — "a+" — zapis i odczyt (w trybie dopisywania)
3. Funkcja **freopen()** najpierw zamyka otwarty wcześniej plik i otwiera go ponownie we wskazanym trybie, zwracając wskaźnik do struktury danych opisujących strumień. W przypadku błędu — zwraca NULL.

```
fp = freopen("plik.txt", "r+", fp)
```

4. Funkcja **fclose()** zamyka wskazany plik. W przypadku błędu — zwraca NULL.

```
#include <stdio.h>
int fclose(FILE *stream);
```

   stream to wskaźnik do struktury danych opisujących strumień danych.

```
fclose( fp );
```

9.4.2. Funkcje pomocnicze

Poniższe funkcje są nieco mniej istotne (na tym poziomie nauki języka C)

1. **fflush()**

```
#include <stdio.h>
int fflush(FILE *stream);
```

   Funkcja powoduje zapis w pliku (związanym ze strumieniem wskazywanym przez stream) wszystkich zbuforowanych danych.
2. **setvbuf()**

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, \
                                    size_t size);
```

— funkcji można użyć zaraz po otwarciu pliku, ale przed wykonaniem jakiejkolwiek innej operacji na pliku
— zmienna mode określa sposób buforowania danych (_IOFBF — pełne buforowanie, _IOLBF — buforowanie linii, _IONBF — buforowanie wyłączone)
— pozostałe parametry (buf i size) określają — jeżeli buf nie jest NULL — obszar (i jego wielkość) przeznaczony do buforowania danych. Jeżeli buf jest równe NULL funkcja przydziela taki obszar automatycznie. Funkcja zwraca zero gdy zakończy się normalnie i wartość różną od zera w przypadku błędów.

9.4.3. Pozycja w pliku

Z każdą operacją wejścia/wyjścia związany jest tak zwany wskaźnik pozycji w pliku. Mówi on do ktorego miejsca dane zostały „doczytane" (czyli gdzie rozpocznie się kolejna operacja). W przypadku otwarcia pliku do odczytu — początkowo ma on wertość zero. W przypadku otwarcia pliku w trybie do dopisywania — ustawiany jest on za ostatnim bajem pliku.

W przypadku standardowych strumieniów danych (stdin, stdout) położenia wskaźnika nie można zmieniać (niezależnie od tego czy system operacyjny za pomocą poleceń przekierowania związał ten strumień z plikiem czy jest on powiązany standardowo z klawiaturą/ekranem).

W przypadku innych strumieni wejścia (powiązanych z plikiem) można tym wskaźnikiem manipulować w dosyć dowolny sposób.

Funkcje **fgetpos**/**fsetpos** mają bardziej techniczny charakter. Lepiej nie zmieniać wartości zwróconej przez **fgetpos** żeby użyć jej do zmiany położenia wskaźnika w pliku funkcją **fsetpos**. Do łatwego manipulowania lepiej użyć funkcji **fseek**/**ftell**.

1. **fgetpos()** i **fsetpos()**

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

— Funkcja **fgetpos** zapisuje aktualną wartość wskaźnika pozycji związanego ze strumieniem stream w obiekcie wskazywanym przez pos
— Funkcja **fsetpos** ustala aktualną wartość wskaźnika pozycji strumienia strem na wartość z obiektu wskazywanego przez pos. Wartość ta powinna być wcześniej uzyskana za pomocą funkcji fgetpos.

2. **fseek()** i **ftell()**

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, \
                                  int whence);
long int ftell(FILE *stream);
```

— Funkcja **ftell** zwraca aktualną wartość wskaźnika pozycji (dla plików binarnych jest to liczba znaków od początku plików; dla plików tekstowych sprawa jest bardziej skomplikowana) strumienia określonego przez stream.
— Funkcja **fseek** dla plików binarnych ustala aktualną wartość wskaźnika pozycji przez sumowanie zmiennej offset z pozycją wskazywaną przez whence; stdio.h zawiera makra SEEK_SET, SEEK_CUR i SEEK_END określające odpowiednio początek pliku, bieżącą pozycję i koniec pliku. Dla plików tekstowych wartość offset powinna być albo zero albo wartością uzyskana wcześniej przez odwołanie do funkcji **ftell**.

3. **rewind()**

```
#include <stdio.h>
void rewind(FILE *stream);
```

Funkcja ustawia wskaźnik pozycji w pliku na początek pliku.

9.4.4. Czytanie z pliku

1. **fget()**

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Funkcja zwraca następny znak (jako unsigned char zamieniony na int) ze strumienia wskazywango przez stream. Jeżeli odczyt dotrze do końca pliku fukcja zwraca EOF. W przypadku błędu ustawiany jest wskaźnik błędu i funkcja zwraca EOF.

2. **fgets()**

```
1 #include <stdio.h>
2 char *fgets(char *s, int n, FILE *stream);
```

Funkcja odczytuje co najwyżej n-1 znaków z pliku wskazywanego przez stream i zapisuje je w tablicy wskazywanej przez s. Odczyt kończy się z chwilą napotkania znaku nowej linii, dojścia do końca strumienia lub przeczytania n-1 znaków. **Uwaga:** Znak nowej linii to \n dla Unixa, \r\n dla DOS/Windows i \r dla MAC (przed OS X).

3. **getc()** — odpowiednik fgetc, może być zaimplementowana jako makro.
4. **getchar()** — odpowiednik getc ale dla stdin
5. **ungetc()**

```
1 #include <stdio.h>
2 int ungetc(int c, FILE *stream);
```

Funkcja „zwraca" znak c do wskazanego strumienia. Operacja może się nie powieść, jeżeli zwracamy w ten sposób zbyt wiele znaków bez wykonania operacji czytania lub pozycjonowania pliku. Zawsze działa dla jednego znaku.
Plik wejściowy nie ulega zmianie!

9.4.5. Odczyt formatowany

1. Funkcje typu **scanf**

```
1 #include <stdio.h>
2 int fscanf(FILE *stream, const char *format, ...);
3 int scanf(const char *format, ...);
4 int sscanf(const char *s, const char *format, ...);
```

Dane czytane są ze wskazanego strumienia (stream) i interpretowane zgodnie z użytym formatem. W formacie powinna wystąpić wystarczająca liczba specyfikacji aby dostarczyć dane dla wszystkich argumentów.

— fscanf zwraca liczbę przeczytanych wartości lub wartość EOF
— scanf jest odpowiednikiem fscanf, ale dotyczy strumienia stdin
— sscanf jest odpowiednikiem fscanf, z tym, że dane „czytane" są z tablicy znakowej; dotarcie do końca tabeli jest równoważne z dotarciem do końca pliku

9.4.6. Wejście: Specyfikacja formatu

Na specyfikację formatu składają się:
— odstępy
— znaki (różne od % i odstępów). Jeżeli taki znak wystąpi w specyfikacji musi się pojawić w strumieniu wejściowym na odpowiednim miejscu!
— specyfikacje konwersji (rozpoczynające się znakiem %)
Po znaku % wystąpić może:
— nieobowiązkowy znak ∗ (oznaczający, że zinterpretowana wartość ma być zignorowana)
— nieobowiązkowa specyfikacja długości pola (określa maksymalną liczbę znaków pola)
— jeden z nieobowiązkowych znaków **h**, **l** (mała litera „el”) lub **L** mówiących jak ma być interpretowana czytana wielkość (h — short, l — long albo double w przypadku float, L — long double)
— znak określający typ konwersji
Po znaku procent (%) wystąpić może jeden ze znaków
d liczba całkowita
i liczba całkowita
o liczba całkowita kodowana ósemkowo
u liczba całkowita bez znaku (unsigned int)
x liczba całkowita kodowana szesnastkowo
e, f, g liczba typu float
s ciąg znaków (na końcu zostanie dodany znak NULL)
c znak (lub ciąg znaków gdy wyspecyfikowano szerokość pola), nie jest dodawane zero na końcu
n do zmiennej odpowiadającej tej specyfikacji konwersji wpisana zostanie liczba znaków przetworzonych przez fscanf
% znak %
Przykład

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int x, y, n;
5     x = scanf("%*i %i %n", &y, &n);
6     printf("y= %i , n= %i\n", y, n);
7     printf("x= %i\n", x);
```

```
8       return 0;
9   }
```

Dane i wyniki

```
10 20 30 40
y= 20, n= 6
x= 1
```

```
1␣␣␣␣␣2222␣ala␣ma␣kota
y=␣2222,␣n=␣11
x=␣1
```

Kolejny przykład

```
1  #include <stdio.h>
2  int main(void)
3  {
4       int x, y, z, n;
5       x = scanf("%4i␣%5i␣%n", &y, &z, &n);
6       printf("y=␣%i,␣z=␣%i,␣n=␣%i\n", y, z, n);
7       printf("x=␣%i\n", x);
8       return 0;
9  }
```

Dane i wyniki

```
1234567890
y= 1234, z= 56789, n= 9
x= 2
```

I jeszcze jedne przykład

```
1  #include <stdio.h>
2  int main(void)
3  {
4       int x, y, z, n;
5       x = scanf("%4i␣a␣%5i␣%n", &y, &z, &n);
6       printf("y=␣%i,␣z=␣%i,␣n=␣%i\n", y, z, n);
7       printf("x=␣%i\n", x);
8       return 0;
9  }
```

wer. 16 z drobnymi modyfikacjami!

Dane i wyniki

```
123b123b
y=␣123,␣z=␣32767,␣n=␣2023244192
x=␣1

1a2␣3␣4␣5␣6␣7␣8␣9
y=␣1,␣z=␣2,␣n=␣4
x=␣2
```

```c
#include <stdio.h>
int main(void)
{
    int x, y, z, n;
    x = scanf("%4ia%5i%n", &y, &z, &n);
    printf("y=␣%i,␣z=␣%i,␣n=␣%i\n", y, z, n);
    printf("x=␣%i\n", x);
    return 0;
}
```

```
123a123
y= 123, z= 123, n= 7
x= 2
```

9.5. Pisanie do pliku

1. **fputc()**

```c
#include <stdio.h>
int fputc(int c, FILE *stream);
```

   Funkcja zapisuje podany znak do pliku (czy też dodaje go do strumienia wyjściowego). Wersja **putc** dodaje znak do stumienia stdout. **putchar** jest odpowiednikiem **putc**.

2. **fputs()**

```c
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Funkcja dodaje wskazany ciąg znaków do strumienia wyjściowego. **puts** jest odpowiednikiem działającym na strumieniu stdout.

Rodzina polecen fprintf

Wszystkie funkcje realizują formatowane wyprowadzanie informacji.

```
1 #include <stdarg.h>
2 #include <stdio.h>
3 int fprintf(FILE *stream, const char *format, ...);
4 int printf(const char *format, ...);
5 int sprintf(char *s, const char *format, ...);
6 int vfprintf(FILE *stream, const char *format, va_list arg);
7 int vprintf(const char *format, va_list arg);
8 int vsprintf(char *s, const char *format, va_list arg);
```

— **fprintf** to podstawowa wersja funkcji
— **printf** używa stdout jako strumienia wyjściowego.
— **sprintf** przekazuje sformatowane informacje do tablicy znakowej (odpowiedniej długości)
— warianty o nazwie rozpoczynającej się na v (vfprintf, fprintf, vsprintf) używają specyficznej metody przekazywania listy parametrów zmiennej długości — nie będziemy się nimi zajmować.

Podobnie jak w przypadku formatowanego wprowadzania danych, zmienna lub stała tekstowa zawiera informacje o sposobie wyprowadzania danych. Zasadą jest, że znak % rozpoczyna specyfikację konwersji, a pozostałe znaki są wyprowadzane w postaci takiej jak w formacie.

Po znaku % wystąpić może:
— zero lub więcej wskaźników modyfikujących sposób konwersji,
— nieobowiązkową specyfikację minimalnej długości pola wyjściowego; gdy wyprowadzana wartość jest krótsza niż pole zostanie uzupełniona odstępami (z lewej lub prawej strony w zależności od specyfikacji)
— Dokładność (podana jako liczba cyfr) dla specyfikacji **d**, **i**, **o**, **u**, **x** oraz **X**, liczba cyfr po przecinku (dla specyfikacji konwersji **a**, **A**, **e**, **E**, **f** oraz **F**) maksymalna liczba cyfr znaczących (dla specyfikacji **g** i **G**) lub liczba znaków (dla specyfikacji **s**). W przypadku liczb „zmiennoprzecinkowych" precyzja ma postać **.** (kropki) po której jest albo ∗ albo liczba określająca liczbę cyfr po kropce dziesiętnej. Liczby podczas wyprowadzania są

zaokrąglane czyli printf("%1.1f\n", 1.19); spowoduje wyprowadzenie 1.2

Specjalne wskaźniki modyfikujące sposób konwersji to:

– wynik będzie justowany do lewej strony (standardowo do prawej)

\+ Liczby będą zapisywane zawsze ze znakiem

odstęp odstęp będzie zawsze dodawany przed liczbą (chyba, że liczba poprzedzona jest znakiem)

\# Wynik jest konwertowany do postaci „alternatywnej” (zależnej od typu konwersji)

0 Liczby będą poprzedzone wiodącymi zerami

9.6. Wejście/wyjście na niskim poziomie

W rozdziale 9.4 opisałem podstawowe operacje na plikach. Wybrałem tryb „wysokopoziomowy”, gdyż — w pierwszym podejściu — jest on łatwiejszy. Poziom „niskopoziomowy” realizowany jest za pomocą funkcji:

— open,
— close,
— creat,
— unlink.

Są to funkcje „systemowe” nie współpracujące z poleceniami scanf, prntf, fread czy fwrite.

9.6.1. fopen

Pierwsza z funkcji służy do otwarcia pliku[2]. Używamy jej tak:

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4
5 int open(const char *pathname, int flags);
6 int open(const char *pathname, int flags, mode_t mode);
```

[2] W dalszym ciągu powinienem używać określenia „strumień”, zwłaszcza, że polecenia te mogą być użyte do komunikacji z urządzeniem podłączonym, na przykład, przez port szeregowy (RS–232)

Pierwszy argument to stała lub zmienna znakowa zawierająca nazwę pliku. Drugi parametr (flags) określa tryb dostępu do pliku; symboliczne nazwy stałych[3]:
— O_RDONLY — plik otwarty będzie w trybie do odczytu.
— O_WRONLY — otwiera plik w trybie do zapisu.
— O_RDWR — tryb wejścia/wyjścia.
Stałe te są zadeklarowane w odpowiednim pliku nagłówkowym.

Trzeci parametr, gdy potrzebny, (mode) określa dokładniej parametry komunikacji. Bardzo często może być równy zero.

Wywołanie funkcji open tworzy nowy deskryptor strumienia i zwraca tę wartość do programu. Wybierana jest kolejna najmniejsza liczba całkowita będąca indeksem do systemowej tablicy zawierającej informacje o strumieniu. Standardowo strumienie 0, 1 i 2 przypisane są do standardowego wejścia, wyjścia oraz standardowego strumienia błędów. W przypadku gdy plik nie może byc otwarty — funkcja zwraca wartość −1.

Plik powinien istnieć. Jeżeli nie istnieje — należy go utworzyć poleceniem creat.

9.6.2. creat

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4
5 int creat(const char *pathname, mode_t mode);
```

Pierwszy parametr to nazwa pliku. Drugi parametr określa (prawa dostępu) jaki będzie miał utworzony plik. Jeżeli chcemy utworzyć plik, do którego właściciel chce mieć prawa dostępu odczyt i zapis parametr ten powinien mieć wartość S_IRUSR|S_IWUSR. Zwracam uwagę na znak | oznacza on sumę bitową. Stała S_IRUSR ma wartość 00400[4] zaś S_IWUSR 00200. Wynikowa wartość stałej będzie zatem 00600. Jak obejrzeć listing kartoteki (w systemie Linux) wykonany poleceniem ls można zobaczyć coś takiego:

[3] Odpowiednie stałe zdefiniowane są w plikach nagłówkowych. Znacznie wygodniej jest używać nazwy O_RDONLY niż za każdym razem sprawdzać w dokumentacji jaka wartość całkowita jej odpowiada...

[4] Pierwsza cyfra stałej to zero, zatem stała kodowana jest ósemkowo!

```
1 -rw-rw-r-- 1 myszka myszka     179 maj 18 19:42 main.c
```

Pierwszy napis (−rw−rw−r−−) określa prawa dostępu do pliku. Pierwszym znakiem nie będziemy się teraz zajmować. Kolejne trzy (rw−) określają, że właściciel ma prawa do odczytu pliku (r), zapisu (w) i nie ma prawa do wykonania pliku (−).

Szczegóły użycia funkcji open i creat zawiera odpowiednia strona podręcznika (man 2 open — rozdział drugi manuala to funkcje systemowe).

9.6.3. close

```
1 #include <unistd.h>
2
3 int close(int fd);
```

Funkcja zamyka strumień związany z deskryptorem fd.

9.7. unlink

```
1 #include <unistd.h>
2
3 int unlink(const char *pathname);
```

Funkcja kasuje wskazany plik. W przypadku sukcesu zwraca wartość 0, w przeciwnym razie −1.

9.7.1. read

Wejście i wyjście niskiego poziomu realizują funkcje read i write.

```
1 #include <unistd.h>
2
3 ssize_t read(int fd, void *buf, size_t count);
```

Pierwszy parametr to deskryptor strumienia, drugi — tablica (znakowa) w programie użytkownika, a trzeci określa liczbę bajtów do przesłania. Każde wywołanie zwraca liczbę rzeczywiście przesłanych bajtów, zero w przypadku końca pliku lub −1 w przypadku błędu.

9.7.2. write

```
1 #include <unistd.h>
2
3 ssize_t write(int fd, const void *buf, size_t count);
```

Parametry identyczne jak w funkcji write.

Funkcje read i write nie dokonują żadnych konwersji — przepisują jedynie ciąg bajtów między tablicą, a plikiem...

Poniższy program tworzy plik zapisuje do niego informację, zamyka plik, otwiera go ponownie odczytuje informacje i kasuje plik.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

// int creat(const char *pathname, mode_t mode);
int main()
{
    int fd;
    int n;
    char bufor[30];
    fd = creat("AlaMa.Kota", S_IRUSR | S_IWUSR);
    printf("fd=%d\n", fd);
    if (fd <= 0)
        return 1;
    n = write(fd, "Ala ma kota\n", 12);
    printf("n=%d\n", n);
    close(fd);
    fd = open("AlaMa.Kota", O_RDONLY, 0);
    printf("fd=%d\n", fd);
    n = read(fd, bufor, 30);
    printf("n=%d\n", n);
    printf("Przeczytalem:\n \"%s\"", bufor);

    n = unlink("AlaMa.Kota");
    return 0;
```

```
}
```

9.8. Port szeregowy

Poniżej pobieżnie opiszę zasady programowania portu szeregowego. Nie jest to specjalnie trudne, wymaga natomiast poznania podstawowych zasad działania portu i sposobu komunikacj. Podstawową dokumentacją może być [7, 5]. Funkcjonowanie portu szeregowego opisane też jest po polsku w [15], ale ta książka może być trudna do zdobycia.

Opiszę tu najbardziej podstawowe fakty. Będę używał następujących pojęć:

DTE *Data Terminal Equipment* czyli urządzenie końcowe: drukarka, klawiatura, monitor, urządzenie pomiarowe.

DCE *Data Communication Equipment* czyli komputer.

Standardowym złączem używanym do komunikacji jest złącze szufladowe, 25-stykowe. A połączenia realizowane są według schematu przedstawionego na rysunku 9.2.

Do przesyłania danych służą linie 7 (SGND czyli „masa”), 2 (TxD — dane wysyłane) i 3 (RxD — dane odbierane). Pozostałe połączenia służą do sygnalizowania stanu urządzenia i sterowania transmisją. Zatem, alternatywnie połączenie może być zrealizowane tak jak na rysunku 9.3. Dziś powszechnie stosuje się (wszędzie tam, gdzie jeszcze zostały porty szeregowe) złącza szufladowe 9-stykowe[5].

Aby móc korzystać z portu szeregowego, trzeba określić parametry jego pracy. Są to:

1. Szybkość transmisji danych,
2. Sposób sterowania przepływem danych,
3. liczbę bitów w przesyłanym znaku i sposób kontroli poprawności (parzystości) transmisji,
4. parametry sterowania modemem.

Kilka słów wyjaśnienia. Najważniejszym parametrem jest szybkość transmisji. Oba urządzenia nadające i odbierające muszą pracować z taką samą szybkością. Ze względu na parametry lini przesyłowej maksymalna odległość nadajnika i odbiornika syganłu jest jakoś związana z szybkością: może być

[5] http://pl.wikipedia.org/wiki/RS-232#Sygna.C5.82y_w_PC

ona większa gdy odległość jest mniejsza. Szybkość podaje się w bitach na sekundę (zwanych czasami bodami). Przyjmuje wartości:

— 1200
— 1800
— 2400
— 3600
— 4800
— 7200
— 9600
— 14400
— 19200
— 28800
— 38400
— 56000
— 57600
— 115200

Urządzenia końcowe używane do odbierania danych z komputera miały zazwyczaj bardzo ograniczoną inteligencję i bardzo niewielką pamięć na przechowywanie danych. W związku z tym, gdy urządzenie nie może przyjąć już żadnych danych informuje nadawcę o tym fakcie odpowiedni stan linii sterujących. Taki sposób sterowania przepływem danych nazywa się sprzętowym... W przypadku przesyłania danych z wykorzystaniem wyłącznie linii TxD i RxD — używany jest specjalny protokół komunikacyjny: aby wstrzymać transmisję urządzenie wysyła specjalny znak ASCII DC3 (Ctrl–S), gdy może już przyjmować wysyła znak DC1 (Ctrl–Q).

Dodatkowo, port szeregowy przystosowany jest do współpracy z modemem. W związku z tym standard przewiduje istnienie specjalnych lini sygnałowych informujących o nadejściu połączenia (dzwonek) czy o przerwaniu połączenia czy wykryciu nośnej[6]. Nie mają one zastosowania po gdy nie korzysta się z modemu. Taki tryb pracy nosi czasami nazwę trybu lokalnego.

Standardowo port związany z portem szeregowym nazywa się w systemie DOS com1 lub com2 (może też być wyższy numerek gdy portów jest więcej). W systemach linuksowych są to urządzenia związane z /dev/ttyS0/ czy /dev/ttyS1/.

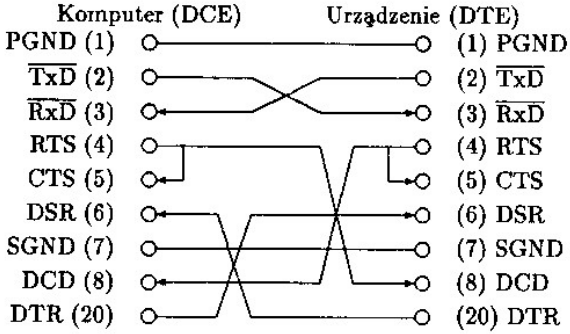
Fragment listingu kartoteki /dev/

[6] http://en.wikipedia.org/wiki/RS-232#Signals.

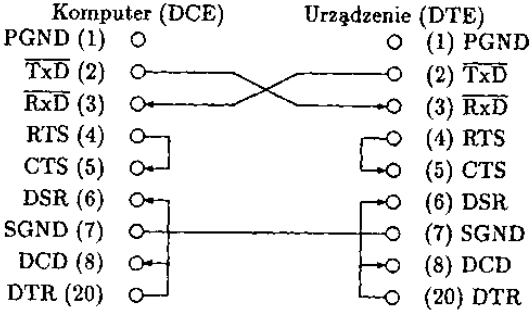
```
1 crw-rw---- 1 root dialout 4, 64 2013-01-06 22:13 ttyS0
2 crw-rw---- 1 root dialout 4, 65 2013-01-06 22:13 ttyS1
3 crw-rw---- 1 root dialout 4, 66 2013-01-06 22:13 ttyS2
4 crw-rw---- 1 root dialout 4, 67 2013-01-06 22:13 ttyS3
```

Plik (urządzenie) ttyS0 to pierwszy port szeregowy (com1).

Szczegóły programowania znaleźć można w cytowanych już dokumentach z serii HowTo oraz w niezłym cyklu zmieszczonym na stronach czasopisma Elektronika Praktyczna [1].



Rysunek 9.2. Przykład typowego połączenia RS 232C ([15])



Rysunek 9.3. Przykład połączenia RS 232C za pomocą kabla trójprzewodowego ([15])

10. Formatowane (tekstowe) wejście/wyjście. Binarne wejście/wyjście.

10.1. Otwarcie pliku

1. Przed skorzystaniem z pliku należy go „otworzyć” (*open*).
2. **stdin**, **stdout**, **stderr** otwarte są w trybie tekstowym.
3. Standardowa biblioteka (stdio.h) zawiera trzy funkcje fopen, freopen i fclose.
4. Pierwsze dwie służą do „skojarzenia” (powiązania) pliku z odpowiednią strukturą danych.
5. Ostatnia — powiązanie to likwiduje.
6. Wszystkie czynności związane z otwieraniem i zamykaniem plików realizowane są przez System Operacyjny.

Operację otwarcia pliku (ponownego otwarcia) realizują funkcje **fopen** i **freopen**:

```
1 #include <stdio.h>
2 FILE *fopen(const char *filename, const char *mode);
3 FILE *freopen(const char *filename, const char *mode,
4                FILE *stream);
```

Użycie:

```
1 FILE *fp;
2 ...
3 fp = fopen("plik.txt", "w");
4 ...
5 fp = freopen("plik.txt", "r", fp);
```

— Pierwszy parametr (filename) to wskaźnik do tablicy tekstowej (na przykład stałej) zawierającej nawę pliku.
— Parametr drugi to ciąg znaków (lub zmienna) zawierająca w sposób symboliczny określenie trybu dostępu:
  — **r** otwórz plik **tekstowy** do odczytu (plik musi istnieć)
  — **w** otwórz plik **tekstowy** do zapisu (niszcząc jego zawartość jeżeli już istnieje)
  — **a** (*append*); otwórz plik **tekstowy** do dopisywania, jeżeli plik nie istnieje — zostanie utworzony.
  — **r+** otwarcie pliku **tekstowego** do zapisu i odczytu (plik powinien już istnieć!)
  — **w+** otwórz istniejący plik (kasując jego zawartość) lub utwórz nowy plik **tekstowy** w trybie do zapisu i odczytu
  — **a+** otwórz plik **tekstowy** na końcu w trybie do zapisu i odczytu

Przykład

```
1 #include <stdio.h>
2 ...
3 FILE *pp;
4 ...
5 pp = fopen("Ala.txt", "r");
6 if (pp == NULL)
7     { /* Błąd! */}
8 ...
9 fscanf(pp, "%d", &i);
```

Przykład pokazuje w jaki sposób zadeklarować zmienną pełniącą rolę „identyfikatora strumienia” wejścia/wyjścia. Plik Ala.txt otwierany jest w trybie do odczytu ("r"). Gdy plik nie może być otwarty zmienna pp

będzie miała wartość zero (stała NULL to właśnie stałą typu **void** ∗ mająca wartość zero; warto z niej korzystać!)
— Trzeci parametr (stream — tylko w przypadku funkcji reopen) zawiera wskaźnik do struktury danych opisujących strumień danych.

1. Funkcja **fopen()** otwiera plik o podanej nazwie we wskazanym trybie, tworzy strukturę danych opisującą go i zwraca do niej adres. W przypadku błędu — zwraca NULL.
2. Funkcja **freopen()** najpierw zamyka otwarty wcześniej plik i otwiera go ponownie we wskazanym trybie, zwracając wskaźnik do struktury danych opisujących strumień. W przypadku błędu — zwraca NULL.
3. Funkcja **fclose()** zamyka wskazany plik. W przypadku błędu — zwraca NULL.

```
1 #include <stdio.h>
2 ...
3 int fclose(FILE *stream);
```

stream to wskaźnik do struktury danych opisujących strumień danych. Przykład:

```
1 ...
2 fclose( fp );
3 ...
```

10.2. Odczyt formatowany

1. Funkcje typu **scanf**

```
1 #include <stdio.h>
2 int fscanf(FILE *stream, const char *format, ...);
3
4 int scanf(const char *format, ...);
5
6 int sscanf(const char *s, const char *format, ...);
```

Dane czytane są ze wskazanego strumienia (stream) i interpretowane zgodnie z użytym formatem. W formacie powinna wystąpić wystarczająca liczba specyfikacji aby dostarczyć dane dla wszystkich argumentów.

— fscanf zwraca liczbę przeczytanych wartości lub wartość EOF
— scanf jest odpowiednikiem fscanf, ale dotyczy strumienia stdin
— sscanf jest odpowiednikiem fscanf, z tym, że dane „czytane" są z tablicy znakowej; dotarcie do końca tabeli jest równoważne z dotarciem do końca pliku

Wejście: Specyfikacja formatu

Na specyfikację formatu składają się:
— odstępy; Wystąpienie w formacie „białego znaku" (*white space*) powoduje, że funkcje z rodziny scanf będą odczytywać i odrzucać znaki, aż do napotkania pierwszego znaku nie będącego białym znakiem.
— znaki (różne od % i odstępów). Jeżeli taki znak wystąpi w specyfikacji musi się pojawić w strumieniu wejściowym na odpowiednim miejscu!
— specyfikacje konwersji (rozpoczynające się znakiem %)

Po znaku % wystąpić może:
— nieobowiązkowy znak ∗ (oznaczający, że zinterpretowana wartość ma być zignorowana)
— nieobowiązkowa specyfikacja długości pola (określa maksymalną liczbę znaków pola)
— jeden z nieobowiązkowych znaków **h**, **l** (mała litera „el") lub **L** mówiących jak ma być interpretowana czytana wielkość (h — **short**, l — **long** albo **double** w przypadku **float**, ll – **long long**, L — **long double**),
— znak określający typ konwersji.

Po znaku procent (%) wystąpić może jeden ze znaków
d liczba całkowita,
i liczba całkowita (można wprowadzać wartości szesnastkowe — jeżeli poprzedzone znakami 0x lub ósemkowe jeżeli poprzedzone 0; 031 czytane z formatem %d to 31 a z formatem %i to 25),
o liczba całkowita kodowana ósemkowo,
u liczba całkowita bez znaku (**unsigned int**),
x liczba całkowita kodowana szesnastkowo,
a, e, f, g liczba typu float; można również wczytać w ten sposób nieskończoność lub wartość Not a Number (NaN),
s ciąg znaków (na końcu zostanie dodany znak zero),
c znak (lub ciąg znaków gdy wyspecyfikowano szerokość pola), nie jest dodawane zero na końcu,

[odczytuje niepusty ciąg znaków, z których każdy musi należeć do określonego zbioru, argument powinien być wskaźnikiem na **char**,

n do zmiennej odpowiadającej tej specyfikacji konwersji wpisana zostanie liczba znaków przetworzonych przez fscanf,

p w zależności od implementacji — służy do wprowadzania wartość wskaźników,

% znak %.

Format „[" — Po otwierającym nawiasie następuje ciąg określający znaki jakie mogą występować w odczytanym napisie i kończy się on nawiasem zamykającym tj.]. Znaki pomiędzy nawiasami (tzw. *scanlist*) określają możliwe znaki, chyba że pierwszym znakiem jest ^ — wówczas w odczytanym ciągu znaków mogą występować znaki nie występujące w scanlist. Jeżeli sekwencja zaczyna się od [] lub [^] to ten pierwszy nawias zamykający nie jest traktowany jako koniec sekwencji tylko jak zwykły znak. Jeżeli wewnątrz sekwencji występuje znak − (minus), który nie jest pierwszym lub drugim jeżeli pierwszym jest ^ ani ostatnim znakiem zachowanie jest zależne od implementacji.

Przykład

```c
#include <stdio.h>
int main(void)
{
    int x, y, n;
    x = scanf("%*d␣%d␣%n", &y, &n);
    printf("y=␣%d,␣n=␣%d\n", y, n);
    printf("x=␣%d\n", x);
    return 0;
}
```

Dane i wyniki

```
10 20 30 40
y= 20, n= 6
x= 1
```

```
1␣␣␣␣␣2222␣ala␣ma␣kota
y=␣2222,␣n=␣11
x=␣1
```

Kolejny przykład

2020-04-01 11:35:22 +0200

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int x, y, z, n;
5     x = scanf("%4d␣%5d␣%n", &y, &z, &n);
6     printf("y=␣%d,␣z=␣%d,␣n=␣%d\n", y, z, n);
7     printf("x=␣%d\n", x);
8     return 0;
9 }
```

Dane i wyniki

```
1234567890
y= 1234, z= 56789, n= 9
x= 2
```

I jeszcze jeden przykład

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int x, y, z, n;
5     x = scanf("%4d␣a␣%5d␣%n", &y, &z, &n);
6     printf("y=␣%d,␣z=␣%d,␣n=␣%d\n", y, z, n);
7     printf("x=␣%d\n", x);
8     return 0;
9 }
```

Dane i wyniki

```
123b123b
y=␣123,␣z=␣32767,␣n=␣2023244192
x=␣1

1a2␣3␣4␣5␣6␣7␣8␣9
y=␣1,␣z=␣2,␣n=␣4
x=␣2
```

...

wer. 16 z drobnymi modyfikacjami!

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int x, y, z, n;
5     x = scanf("%4da%5d%n", &y, &z, &n);
6     printf("y= %d, z= %d, n= %d\n", y, z, n);
7     printf("x= %d\n", x);
8     return 0;
9 }
```

```
123a123
y= 123, z= 123, n= 7
x= 2
```

10.2.1. Podchwytliwe pytania

P: Jak powinna wyglądać specyfikacja wprowadzania pozwalająca poprawnie zinterpretować dane w postaci:

```
123ala456
```

O: Oczywiście tak: %iala%i lub lepiej %dala%d

P: A jak powinna wyglądać specyfikacja pozwalająca przeczytać dane postaci

```
123ala456
```

oraz

```
123ola456
```

Odpowiedź będzie dłuższa...

Popatrzmy na program

```
1 #include <stdio.h>
2 int main(void){
3     int x, y, z, n;
4     char tekst[100];
5     x = scanf("%d%3[a-z]%d%n", &y, tekst, &z, &n);
6     printf("y= %d ", y);
7     printf("tekst= %s ", tekst);
8     printf("z= %d ", z);
```

```
9       printf("n= %d\n", n);
10      printf("x= %d\n", x);
11      return 0;
12  }
```

```
123ala20
y= 123 tekst= ala z= 20 n= 8
x= 3

12ooo3456
y= 12 tekst= ooo z= 3456 n= 9
x= 3

123ALA34
y= 123 tekst=  z= 1250361488 n= 0
x= 1
```

Jeżeli specyfikację zmienimy na: "%d%*3[a–z]%d%n" to dane tekstowe będą ignorowane, odpowiednie polecenie będzie wyglądać tak:

```
1       x = scanf("%d%*3[a-z]%d%n", &y, &z, &n);
```

Co jeszcze może pojawić się wewnątrz nawiasów kwadratowych?
— ciąg znaków — dotyczy wymienionych znaków: [abc]
— znak ^ na pierwszym miejscu — wszystkie znaki za wyjątkiem wymienionych: [^ABC]
— *początek–koniec* — znaki z podanego zakresu: [a–z]
Warto też poczytać o wyrażeniach regularnych czasami nazywanych „regułowymi”[1]

P: W jaki sposób przeczytać dowolny napis?
— Jak wiadomo specyfikacja %s czyta znaki do pierwszego odstępu. Zatem nie można w ten sposób przeczytać napisau „Ala ma kota” (ze względu na odstępy.
— Można wspomóc się wyrażeniami regularnymi: specyfikacja %[aA]s czytać będzie dane tekstowe **zgodne z wzorcem**. Czyli tylko litery a lub A.
— Użycie specyfikacji %[^]s powoduje przeczytanie dowolnych znaków aż do znaku odstępu (w istocie znaczy to samo co poprzednio: wzorcem jest dowolny znak różny od spacji!).
— Co spowoduje zatem specyfikacja %[^\n]s?

[1] http://pl.wikibooks.org/wiki/AutoIt/Wyra%C5%BCenia_regularne.

10.3. Formatowane wyprowadzanie danych

Rodzina poleceń fprintf

Wszystkie funkcje realizują formatowane wyprowadzanie informacji.

```
1 #include <stdarg.h>
2 #include <stdio.h>
3 int fprintf(FILE *stream, const char *format, \
4             ...);
5 int printf(const char *format, ...);
6 int sprintf(char *s, const char *format, ...);
7 int snprintf(char *str, size_t size, const char *format, ...);
8 int vfprintf(FILE *stream, const char *format, \
9               va_list arg);
10 int vprintf(const char *format, va_list arg);
11 int vsprintf(char *s, const char *format, \
12               va_list arg);
```

— **fprintf** to podstawowa wersja funkcji
— **printf** używa stdout jako strumienia wyjściowego.
— **sprintf** przekazuje sformatowane informacje do tablicy znakowej (odpowiedniej długości — musi zadbać programista)
— **snprintf** dodatkowy parametr mówi o długości tablicy znakowej, do której mają być zapisywane dane; dłuższe zostaną przycięte!
— warianty o nazwie rozpoczynającej się na v (vfprintf, fprintf, vsprintf) używają specyficznej metody przekazywania listy parametrów zmiennej długości — nie będziemy się nimi zajmować.

Wyjście: Specyfikacje formatu

Podobnie jak w przypadku formatowanego wprowadzania danych, zmienna lub stała tekstowa zawiera informacje o sposobie wyprowadzania danych. Zasadą jest, że znak % rozpoczyna specyfikację konwersji, a pozostałe znaki są wyprowadzane w postaci takiej jak w formacie.

Specjalne wskaźniki modyfikujące sposób konwersji to:

– wynik będzie justowany do lewej strony (standardowo do prawej)
\+ Liczby będą zapisywane zawsze ze znakiem
odstęp odstęp będzie zawsze dodawany przed liczbą (chyba, że liczba poprzedzona jest znakiem)

\# Wynik jest konwertowany do postaci „alternatywnej” (zależnej od typu konwersji)
   — dla formatu **o** powoduje to zwiększenie precyzji, jeżeli jest to konieczne, aby na początku wyniku było zero;
   — dla formatów **x** i **X** niezerowa liczba poprzedzona jest ciągiem 0x lub 0X;
   — dla formatów **a**, **A**, **e**, **E**, **f**, **F**, **g** i **G** wynik zawsze zawiera kropkę nawet jeżeli nie ma za nią żadnych cyfr;
   — dla formatów **g** i **G** końcowe zera nie są usuwane.

0 Liczby będą poprzedzone wiodącymi zerami (dla formatów **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g** i **G**)

10.3.1. Szerokość pola i precyzja

Minimalna szerokość pola oznacza ile najmniej znaków ma zająć dane pole. Jeżeli wartość po formatowaniu zajmuje mniej miejsca jest ona wyrównywana spacjami z lewej strony (chyba, że podano flagi, które modyfikują to zachowanie). Domyślna wartość tego pola to 0.

Precyzja dla formatów:
— **d**, **i**, **o**, **u**, **x** i **X** określa minimalną liczbę cyfr, które mają być wyświetlone i ma domyślną wartość 1;
— **a**, **A**, **e**, **E**, **f** i **F** — liczbę cyfr, które mają być wyświetlone po kropce i ma domyślną wartość 6;
— **g** i **G** określa liczbę cyfr znaczących i ma domyślną wartość 1;
— dla formatu **s** — maksymalną liczbę znaków, które mają być wypisane.

Szerokość pola może być albo dodatnią liczbą zaczynającą się od cyfry różnej od zera albo gwiazdką. Podobnie precyzja z tą różnicą, że jest jeszcze poprzedzona kropką. Gwiazdka oznacza, że brany jest kolejny z argumentów, który musi być typu **int**. Wartość ujemna przy określeniu szerokości jest traktowana tak jakby podano flagę − (minus).

10.3.2. Rozmiar argumentu

1. Dla formatów **d** i **i** można użyć jednego ze modyfikator rozmiaru:
   — **hh** — oznacza, że format odnosi się do argumentu typu **signed char**,
   — **h** — oznacza, że format odnosi się do argumentu typu **short**,
   — **l** (*el*) — oznacza, że format odnosi się do argumentu typu **long**,

— **ll** (**el el**) — oznacza, że format odnosi się do argumentu typu **long long**,
— **j** — oznacza, że format odnosi się do argumentu typu intmax_t,
— **z** — oznacza, że że format odnosi się do argumentu typu będącego odpowiednikiem typu size_t ze znakiem,
— **t** — oznacza, że że format odnosi się do argumentu typu ptrdiff_t.

2. Dla formatów **o**, **u**, **x** i **X** można użyć takich samych modyfikatorów rozmiaru jak dla formatu **d** i oznaczają one, że format odnosi się do argumentu odpowiedniego typu bez znaku.
3. Dla formatu **n** można użyć takich samych modyfikatorów rozmiaru jak dla formatu **d** i oznaczają one, że format odnosi się do argumentu będącego wskaźnikiem na dany typ.
4. Dla formatów **a**, **A**, **e**, **E**, **f**, **F**, **g** i **G** można użyć modyfikatorów rozmiaru **L**, który oznacza, że format odnosi się do argumentu typu **long double**.
5. Dodatkowo, modyfikator **l** (*el*) dla formatu **c** oznacza, że odnosi się on do argumentu typu wint_t, a dla formatu **s**, że odnosi się on do argumenty typu wskaźnik na wchar_t.

10.3.3. Format

Funkcje z rodziny printf obsługują następujące formaty:

d, i — argument typu **int** jest przedstawiany jako liczba całkowita ze znakiem w postaci [−]ddd.

o, u, x, X — argument typu **unsigned int** jest przedstawiany jako nieujemna liczba całkowita zapisana w systemie oktalnym (**o**), dziesiętnym (**u**) lub heksadecymalnym (**x** i **X**).

f, F — argument typu **double** jest przedstawiany w postaci [−]ddd.ddd.

e, E — argument typu **double** jest reprezentowany w postaci [−]d.ddde+dd, gdzie liczba przed kropką dziesiętną jest różna od zera, jeżeli liczba jest różna od zera, a + oznacza znak wykładnika. Format **E** używa wielkiej litery E zamiast małej.

g, G — argument typu **double** jest reprezentowany w formacie takim jak **f** lub **e** (odpowiednio **F** lub **E**) zależnie od liczby znaczących cyfr w liczbie oraz określonej precyzji.

a, A — argument typu **double** przedstawiany jest w formacie [−]0xh.hhhp+d czyli analogicznie jak dla **e** i **E**, tyle że liczba zapisana jest w systemie heksadecymalnym.

c — argument typu **int** jest konwertowany do **unsigned char** i wynikowy znak jest wypisywany. Jeżeli podano modyfikator rozmiaru **l** argument typu wint_t konwertowany jest do wielobajtowej sekwencji i wypisywany.

s — argument powinien być typu wskaźnik na **char** (lub wchar_t). Wszystkie znaki z podanej tablicy, aż do, i z wyłączeniem znaku null są wypisywane.

p — argument powinien być typu wskaźnik na **void**. Jest to konwertowany na serię drukowalnych znaków w sposób zależny od implementacji.

n — argument powinien być wskaźnikiem na liczbę całkowitą ze znakiem, do którego zapisana jest liczba zapisanych znaków.

10.4. Pliki binarne

10.4.1. Zapis/odczyt danych do/z pliku

„Otwarcie" pliku

```
1 #include <stdio.h>
2 FILE *fopen(const char *filename,
3             const char *mode);
4 FILE *freopen(const char *filename,
5               const char *mode, FILE *stream);
6 ...
7 FILE *fp;
```

— Pierwszy parametr (filename) to wskaźnik do tablicy tekstowej (na przykład stałej) zawierającej nawę pliku.

— Parametr drugi to ciąg znaków (lub zmienna) zawierająca w sposób symboliczny określenie trybu dostępu:

  — **rb** otwórz plik binarny do czytania

  — **wb** utwórz plik do zapisu w trybie binarnym, jeżeli plik istnieje, jego zawartość zostanie skasowana

  — **ab** otwórz plik binarny w trybie do dopisywania (jeżeli plik nie istnieje — zostanie utworzony).

  — **r+b** lub **rb+** zapis i odczyt dla plików binarnych (plik musi istnieć)

  — **w+b** lub **wb+** otwórz w trybie binarnym plik istniejący (kasując jego zawartość) lub utwórz plik nowy w trybie do zapisu i odczytu

  — **a+b** lub **ab+** otwórz plik w trybie binarnym na końcu w trybie do zapisu i odczytu

— Trzeci parametr (stream — tylko w przypadku funkcji reopen) zawiera wskaźnik do struktury danych opisujących strumień danych.

Zapis/Odczyt binarny

1. **fread()**

```
1 #include <stdio.h>
2 size_t fread(void *ptr, size_t size,
3              size_t nmemb, FILE *stream);
```

Funkcja realizująca dostęp bezpośredni do pliku, pozwala odczytać **nmemb** elementów o wielkości **size** każdy do tablicy wskazywanej przez **ptr** ze strumienia **stream**. Funkcja zwraca liczbę przeczytanych elementów, która może być mniejsza od **nmemb** gdy wystąpi błąd lub funkcja dojdzie do końca pliku.

2. **fwrite()**

```
1 #include <stdio.h>
2 size_t fwrite(const void *ptr, size_t size,
3               size_t nmemb, FILE *stream);
```

Funkcja zapisuje binarnie elementy tablicy wskazanej przez **ptr** do strumienia stream. **size** określa wielkość jednego obiektu, a **nmemb** liczbę obiektów.

Zapis tablicy

```
1 #include <stdio.h>
2 int main()
3 {
4     FILE *pp;
5     double t[10] = {
6         0, 1, 2, 3, 4, 5, 6, 7, 8, 9
7     };
8     pp = fopen("Ala.txt", "wb");
9     if ( pp == NULL )
10    {
11        printf("Blad!\n");
12        return 1;
13    }
```

```
14     fwrite(t, sizeof( double ), 10, pp);
15     fclose(pp);
16     return 0;
17 }
```

```
1  /*
2   * Odczyt jednej liczby w trybie binarnym
3   */
4  #include <stdio.h>
5  int main(void)
6  {
7     FILE *fp;
8     unsigned int e;
9     fp = fopen("foo.txt", "rb");
10    if ( fp == NULL )
11    {
12       printf("blad1\n");
13       return 1;
14    }
15    if ( fread(&e, sizeof( e ), 1, fp) == 0 )
16    {
17       printf("blad2\n");
18       return 1;
19    }
20    printf("e = %u\n", e);
21    return 0;
22 }
```

1. Plik foo.txt nie istnieje

```
$ ls
b.c  Debug
$ Debug/binarne
blad1
```

2. Plik foo.txt istnieje, ale jest pusty

```
$ touch foo.txt
$ ls -l
```

wer. 16 z drobnymi modyfikacjami!

```
razem 8
-rw-r--r-- 1 myszka myszka  319 2008-05-05 12:05 b.c
drwxr-xr-x 2 myszka myszka 4096 2008-05-05 12:05 Debug
-rw-r--r-- 1 myszka myszka    0 2008-05-05 12:12 foo.txt
$ Debug/binarne
blad2
```

3. Plik istnieje, ma długość trzech bajtów:

```
$ ls -l
razem 12
-rw-r--r-- 1 myszka myszka  319 2008-05-05 12:05 b.c
drwxr-xr-x 2 myszka myszka 4096 2008-05-05 12:05 Debug
-rw-r--r-- 1 myszka myszka    3 2008-05-05 12:15 foo.txt
$ Debug/binarne
blad2
```

4. Plik istnieje, ma długość czterech bajtów:

```
$ cat foo.txt
abc
$ ls -l
razem 12
-rw-r--r-- 1 myszka myszka  319 2008-05-05 12:05 b.c
drwxr-xr-x 2 myszka myszka 4096 2008-05-05 12:05 Debug
-rw-r--r-- 1 myszka myszka    4 2008-05-05 12:18 foo.txt
$ Debug/binarne
e = 174285409
```

10.4.2. Zapis i odczyt

```
1 #include <stdio.h>
2 int main(void)
3 {
4     FILE *fp;
5     unsigned int i, k;
6     fp = fopen("foo.txt", "wb");
7     if ( fp == NULL )
8     {
```

```
9         printf("blad1\n");
10        return 1;
11    }
12    for ( i = 0; i < 10; i++ )
13    {
14        fseek(fp, i * sizeof( i ), SEEK_SET);
15        fwrite(&i, sizeof( i ), 1, fp);
16    }
17    fclose(fp);
18    fp = fopen("foo.txt", "rb");
19    if ( fp == NULL )
20    {
21        printf("blad1\n");
22        return 1;
23    }
24    for ( i = 0; i < 10; i++ )
25    {
26        fseek(fp, ( 9 - i ) * sizeof( i ), SEEK_SET);
27        fread(&k, sizeof( i ), 1, fp);
28        printf("k= %d\n", k);
29    }
30    fclose(fp);
31    return 0;
32 }
```

Zapis i odczyt (wariant drugi)

```
1 #include <stdio.h>
2 int main(void)
3 {
4     FILE *fp;
5     unsigned int i, k;
6     fp = fopen("foo.txt", "w+b");
7     if ( fp == NULL )
8     {
9         printf("blad1\n");
10        return 1;
11    }
```

wer. 16 z drobnymi modyfikacjami!

```
12    for ( i = 0; i < 10; i++ )
13    {
14        fseek(fp, i * sizeof( i ), SEEK_SET);
15        fwrite(&i, sizeof( i ), 1, fp);
16    }
17    for ( i = 0; i < 10; i++ )
18    {
19        fseek(fp, ( 9 - i ) * sizeof( i ), SEEK_SET);
20        fread(&k, sizeof( i ), 1, fp);
21        printf("k= %d\n", k);
22    }
23    fclose(fp);
24    return 0;
25 }
```

10.5. Czytanie z pliku: najprostsza sytuacja

Program może wygladać tak:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5         char z;
6         int i;
7         while ( (i=scanf("%c", &z)) != -1)
8         {
9                 printf("i= %d, z= %d\n", i, z);
10        }
11        return 0;
12 }
```

Program czyta informacje z klawiatury, znak po znaku używając formatu %c a wyniki zapisuje w zmiennej c. W zmiennej i przechowywana jest informacja zwracana przez funkcję scanf. Cała „magia" znajduje się w linii 7 programu. Wykonywane są tu nastepujące czynnosći:

1. Funkcja scanf czyta jeden znak i zapisuje go do zmiennej z.

2. Wynik zwracany przez funkcję scanf zapisywany jest do zmiennej i, a następnie...
3. ...Wynik ten porównywany jest z wartością -1[2] (oznacza ona, że nie ma więcej danych do przeczytania — wystąpił koniec pliku).

Wadą programu jest to, że czyta z klawiatury. Żeby czytał z pliku musi być uruchomiony w specjalny sposób:

```
./program < ala.txt
```

10.6. Czytanie z zadanego pliku

Program można przerobić, żeby czytał nie z klawiatiry tylko z pliku. Będzie wyglądał tak:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        FILE * plik;
        char z;
        int i;
        plik = fopen("ala.txt", "r");
        if (plik == NULL)
        {
                printf("Nie moge otworzyc pliku!\n");
                return 1;
        }
        while ( (i=fscanf(plik, "%c", &z)) != -1)
        {
                printf("i= %d, z= %d\n", i, z);
        }
        return 0;
}
```

Zwracam uwagę na następujące rzeczy:

[2] Dla wygody programisty zdefiniowana jest stała EOF (***E**nd **O**f **F**ile*).

— linię 5: zadeklarowana jest tam specjalna zmienna plik, które będzie przechowywałą wszystkie informacja na temat pliku.
— w linii 8 dokonuje się operacja „otwarcia” pliku; program mówi systemowi operacyjnemu, że wszystkie operacje bedzie wykonywał na pliku, ktory nazywa się "ala.txt",
— linie 9–13 zawierają sprawdzenie czy udało się otworzyć plik o podanej nazwie. Nie uda się otworzyć, gdy go nie ma, na przykład.
— w linii 14 scanf zostało zastąpione funkcją fscanf — czyta ona z pliku, o którym informacje przechowywane są w zmiennej plik...

Reszta bez zmian.

10.7. Sytuacja poważniejsza: cały plik w pamięci

Gdy nie chcemy operaować na pojedynczych znakach czytanych jeden po drugim, mozemy wczytać cały plik do pamieci. Program wygląda tak:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define TRUE 1
4 int main()
5 {
6         FILE * plik;
7         char z;
8         int i;
9         plik = fopen("ala.txt", "rb");
10        if (plik == NULL)
11        {
12                printf("Nie moge otworzyc pliku!\n");
13                return 1;
14        }
15        char ala[1000];
16        int I;
17        I=fread(ala, 1, 1000, plik);
18        printf("I=%d\n",I);
19        printf("%s\n==========\n", ala);
20
21        return 0;
```

```
22 }
```

Używamy teraz dodatkowej tablicy (zadeklarowana w linii 15) która musi mieć wystarczającą wielkość aby pomieścił sę tam cały plik.

W linii 17 użyta jest funkcja fread która czyta zadaną liczbę bajtów do tablicy. Ponieważ zadeklarowana tablica ma 1000 bajtów (znaków) to czytam tylko 1000 jednostek pamięci (trzeci parametr funkcji fread)[3]. Każda jednostka pamięci zajmuje 1 bajt (drugi parametr wywołania funkcji). Dane trafiają do tablicy ala (pierwszy parametr), a czytamy z pliku otwartego w linii 9 programu (plik). Zwracam uwagę, że teraz drugi parametr unkcji fopen ma wartość "rb" co oznacza otwarcie pliku w trybie binarym. W tym trybie nie używamy funkcji scanf, a raczej fread.

Zawartość pliku trafia do tablicy, któr amoże być traktowana jak długi napis. Są tam wszystkie znaki (bajty z pliku) łącznie ze znakami nowej lini... Linia 19 programu drukuje zawartość pliku traktując ją jak zmienną tekstową...

10.8. Odczyt pliku dowolnej długości

Gdzy chcemy odczytac plik dowolnej długości, można użyć takiego programu:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define TRUE 1
4 int main()
5 {
6         FILE * plik;
7         int i;
8         plik = fopen("ala.txt", "rb");
9         if (plik == NULL)
10        {
11                printf("Nie moge otworzyc pliku!\n");
12                return 1;
13        }
```

[3] Gdy plik jest krótszy, przeczytany będzie do końca, a zmienna I będzie zawierać liczbę przeczytanych bajtów.

```
14          fseek (plik,0,SEEK_END);
15          i=ftell (plik);
16          rewind(plik);
17          printf("dlugosc pliku: %d\n",i);
18          char *ala;
19          ala=malloc(i);
20          int I;
21          I=fread(ala, 1, i, plik);
22          printf("I=%d\n",I);
23          printf("%s\n==========\n", ala);
24
25          return 0;
26  }
```

Magia powyższego programu zaczyna się w linii 14. Po otwarci pliku wydaje polecenie ftell prosząc aby przejśc na koniec pliku (SEEK_END). Dodatkowy (drugi) parametr mówi jak daleko od końca mam być. 0 oznacza, że na samym końcu).

Funkcja ftell to pytanie gdzie jestem (jak daleko — w bajtach — od początku), zatem zmienna i zawiera długość pliku. Kolejne polecenie (rewind) przenosi mnie z powrotem na początek pliku.

Deklaruję wskaźnik ala (linia 18) i proszę przydział i bajtów pamieci korzystając z funkcji malloc.

Reszta jak w programie poprzednim...

Teraz można zamiast na pliku operować na tablicy tekstowej zawierającej dokładnie to samo co plik.

11. Operacje na łańcuchach znaków

11.1. Łańcuch znaków

1. Z łańcuchów znaków korzystamy powszechnie.
2. Najprostszy przykład:

```
1 printf("Ala␣ma␣kota");
```

3. Łańcuchem znaków (ang: *string*) jest każdy napis zawarty w cudzysłowach.
4. Łańcuch znaków jest **tablicą** typu **char**, w której na ostatniej pozycji znajduje się znak null (znak o kodzie zero).

```
1 char *tekst = "Jakis␣tam␣tekst";
2 printf("%c\n", "przyklad"[0]);
3    /* wypisze p - znaki w napisach sa
4       numerowane od zera */
5 printf("%c\n", tekst[2]); /* wypisze k */
```

Chwili zastanowienia wymaga uświadomienie sobie, że wszystkie informacje przechowywane w komputerze, przechowywane są w postaci dwójkowej. Dotyczy to tak wartości liczbowych jak i wszelkich napisów: wszystko pamiętane jest jako ciąg bajtów.

Współczesne komputery mają „bajtową" strukturę pamięci. Oznacza to, że najmniejszą jednostką informacji którą można z pamięci wyjąć (lub włożyć) jest jeden bajt[1].

Dla wygody przyjęto, że każdy znak ASCII[2] zajmuje jeden bajt — pozostawiając niewykorzystany najbardziej znaczący jego bit. I każdy znak może być traktowany jako liczba (zwana kodem ASCII tego znaku).

[1] Pomijam tu kwestię praktycznej realizacji dostępu do pamięci. Przyjęta architektura, tak na prawdę daję łatwy dostęp jedynie do dwu (lub czterech) bajtów o parzystym (podzielnym przez cztery) adresie.

[2] Zapomnijmy na razie o znakach Unicode.

1. Język C nie robi jakichś wielkich rozróżnień między znakami a cyframi:

```
#include <stdio.h>
int main(void)
{
    char test[] = "test";
    int i;
    printf("%ld\n", sizeof(test));
    for (i = 0; i < sizeof(test); i++)
        printf("%d\n", test[i]);
    return 0;
}
```

2. Efekt działania tego programu

```
5
116
101
115
116
0
```

11.2. Deklaracje

Poniższe deklaracje są (w zasadzie równoważne)

```
char *tekst  = "Jakis tam tekst";
 /* Umieszcza napis w obszarze danych
    programu i przypisuje adres */
char tekst[] = "Jakis tam tekst";
 /* Umieszcza napis w tablicy */
char tekst[] = {'J','a','k','i','s',\
    ' ','t','a','m',' ','t','e','k',\
    's','t','\0'};
 /* Tekst to taka tablica jak kazda inna */
```

Jednak w pierwszym przypadku tekst jest odsyłaczem do ***stałej!*** Elementów tablicy nie można zmieniać! I w tym przypadku, o ile poprawne jest odwołanie

```
    a = tekst[2];
```

to niepoprawne jest:

```
1     tekst[2] = 'z';
```

Nie można zmienić wartości stałej!

W pozostałych dwu przypadkach obie powyższe operacje są poprawne i dozwolone.

11.3. Operacje na łańcuchach

Plik nagłówkowy `string.h` zawiera definicje stałych i funkcji związanych z operacjami na łańcuchach tekstów.

```
1  void     *memccpy(void *, const void *, int,\
2           size_t);
3  void     *memchr(const void *, int, size_t);
4  int      memcmp(const void *, const void *,\
5           size_t);
6  void     *memcpy(void *, const void *,\
7           size_t);
8  void     *memmove(void *, const void *,\
9           size_t);
10 void     *memset(void *, int, size_t);
11 char     *strcat(char *, const char *);
12 char     *strchr(const char *, int);
13 int      strcmp(const char *, const char *);
14 int      strcoll(const char *, const char *);
15 char     *strcpy(char *, const char *);
16 size_t   strcspn(const char *, const char *);
17 char     *strdup(const char *);
```

```
1  char     *strerror(int);
2  size_t    strlen(const char *);
3  char     *strncat(char *, const char *,\
4           size_t);
5  int      strncmp(const char *, const char *,\
6           size_t);
7  char     *strncpy(char *, const char *,\
```

```
8           size_t);
9  char     *strpbrk(const char *, const char *);
10 char     *strrchr(const char *, int);
11 size_t   strspn(const char *, const char *);
12 char     *strstr(const char *, const char *);
13 char     *strtok(char *, const char *);
14 char     *strtok_r(char *, const char *,\
15          char **);
16 size_t   strxfrm(char *, const char *,\
17          size_t);
```

11.4. Porównywanie ciągów znaków

1. Porównywanie pojedynczych znaków — na podstawie ich kodów ASCII
2. Dla ciągów znaków "aaa" > "aa" > "a" > ""; "ab" > "aa"; "a" > "A"
3. Funkcja *strcmp* służy do porównywania dwu ciągów znaków; ma dwa parametry i zwraca -1 gdy pierwszy ciąg znaków jest mniejszy od drugiego, 0 gdy są równe i $+1$, gdy pierwszy ciąg jest większy od drugiego.
4. Funkcja *strncmp* (o trzech parametrach, trzeci parametr wskazuje ile znaków ma być porównanych) może być użyta, gdy nie chcemy porównywać całych ciągów znaków.

11.5. Kopiowanie znaków

1. Do kopiowania służy funkcja *strcpy* o dwu parametrach; ciąg znakach zawarty w drugim parametrze kopiowany jest do pierwszego parametru. *Do obowiązku programisty należy zapewnienie aby tablica będąca pierwszym parametrem miała wystarczającą ilość miejsca na pomieszczenie całego zapisu!*
2. Drugi wariant funkcji *strncpy*; dodatkowy, trzeci parametr mówi ile znaków ma być skopiowanych — (ale zawsze trzeba pamiętać o miejscu na znak null znajdującym się na końcu napisu). *Znaki z drugiego parametry kopiowane są do końca łańcucha, chyba że tekst jest dłuższy niż liczba znaków do skopiowania; w tym przypadku programista musi dodać znak NULL „ręcznie" na końcu skopiowanego tekstu.*

11.6. Łączenie napisów

1. Do łączenia napisów służy funkcja *strcat* (nazwa pochodzi od STRing conCATenate).
2. Podobnie jak w poprzednich przypadkach jest również drugi wariant *strncat*
3. W wersji standardowej Zawartość drugiego parametru funkcji dopisywana jest na końcu zawartości pierwszego.

```
1 char tekst[80] = "";
2 strcat(tekst, "ala");
3 strcat(tekst, " ma ");
4 strcat(tekst, "kota");
5 puts(tekst);
```

4. Pierwszy parametr funkcji musi mieć dostateczną ilość miejsca na pomieszczenie połączonych ciągów znaków!
5. W drugim wariancie występuje trzeci parametr mówiący ile znaków ma być dołączonych.

11.7. Długość ciągu znaków

1. Funkcja *strlen* podaje długość ciągu znaków (bez znaku NULL)

11.8. Wyszukiwanie

1. Funkcja *strchr* (gdzie pierwszym parametrem jest ciąg znaków a drugim pojedynczy znak) zwraca numer znaku zgodnego z poszukiwanym.
2. Funkcja *strrchr* poszukuje od prawej do lewej.
3. Funkcja *strstr* (o dwu parametrach) wyszukuje pierwszego wystąpienia ciągu znaków będącego drugim parametrem w pierwszym.
4. Funkcja *strtok* może służyć do podziału pierwszego parametru na ciąg żetonów; zakładamy, że żetony (tokeny) oddzielone są zadanym znakiem (drugi parametr).

1. *atol*, *strtol* — zamienia łańcuch na liczbę całkowitą typu **long**
2. *atoi* — zamienia łańcuch na liczbę całkowitą typu **int**

3. *atoll*, *strtoll* — zamienia łańcuch na liczbę całkowitą typu **long long** (64 bity); dodatkowo istnieje przestarzała funkcja *atoq* będąca rozszerzeniem GNU,
4. *atof*, *strtod* — przekształca łańcuch na liczbę typu double

Funkcje ato* nie wykrywają błędów konwersji

Funkcje zdefiniowane są w pliku `stdlib.h`

Plik nagłówkowy `ctype.h` zawiera definicje funkcji (i różnych stałych) związanych z rozpoznawaniem typów informacji

```
1 int    isalnum(int);
2 int    isalpha(int);
3 int    isascii(int);
4 int    isblank(int);
5 int    iscntrl(int);
6 int    isdigit(int);
7 int    isgraph(int);
8 int    islower(int);
9 int    isprint(int);
```

Operacja realizowana przez poszczególne funkcje odpowiada znaczeniu jej nazwy: isalnum sprawdza czy testowany znak należy do klasy znaków alfanumerycznych, isdigit — czy jest cyfrą, islower — czy litera jest „mała”, czy wielka...

```
1 int    ispunct(int);
2 int    isspace(int);
3 int    isupper(int);
4 int    isxdigit(int);
5 int    toascii(int);
6 int    tolower(int);
7 int    toupper(int);
8 int    _toupper(int);
9 int    _tolower(int);
```

isspace — jest prawdą, gdy znak należy do klasy „white space” znaków, które generują odstęp (spacja, wysuw strony ('\f'), nowa linia ('\n'), powrót karetki ('\r'), tabulator ('\t'), wysuw papieru ('\v')).

2020-05-20 08:40:00 +0200

11.9. Konwersje

```
1 int atoi( const char * string );
2 long atol( const char *s );
3 double atof( const char* string );
```

Parametrem funkcji jest ciąg znaków zawierający napis, który ma być skonwertowany

Funkcje z rodziny strto*

```
1 long int strtol(const char *str, char **endptr, int base);
2 unsigned long int strtoul(const char *str, char **endptr, int base);
3 double strtod(const char *str, char **endptr);
```

Pierwszym parametrem funkcji jest napis podlegający konwersji, drugim jest znak ograniczający napis (może to być znak NULL). Trzecim parametrem w przypadku funkcji konwersji do postaci całkowitej jest podstawa zapisu liczb (zawarta pomiędzy 2 a 36). Specjalna wartość 0 pozwala automatycznie rozpoznawać liczby postaci 0nnnnn (gdzie n to cyfra od zera do 7) jako ósemkowe a liczby postaci 0xuuuu jako szesnastkowe (u to cyfry od zera do 9 oraz a — f lub A — F)

11.10. Unicode

1. Jeżeli ograniczyć się do kodowania jednobajtowego (255 znaków) nie ma właściwie żadnych problemów, poza tym, że pogram musi być uruchamiany w odpowiednim środowisku. Dostępne kodowania znaków to ISO-8859-n (n od 1 do 9), cp-12xx, i tak dalej, i tak dalej...
2. Standardowe funkcje porównywania ciągów znaków nie będą działały!
3. Uniwersalny system kodowania znaków Unicode został dopuszczony do użytku w standardzie C99
4. Z Unicode też nie jest łatwo, mamy kilka rodzajów. Zestaw znaków Unicode obejmuje ponad 900 tys. symboli. Zapisać to można na 3 bajtach.
5. Ze względów technicznych przyjęto 4 bajty jako podstawowa wielkość znaku (bo łatwo jedną instrukcją) wybrać z pamięci.
6. W praktyce to za dużo, stąd warianty
   a) UTF-8 — od 1 do 6 bajtów (dla znaków poniżej 65536 do 3 bajtów) na znak; wszystkie standardowe znaki ASCII na jednym bajcie.
   b) UTF-16 — 2 lub 4 bajty (UTF-32) na znak.

   c) UCS-2 — 2 bajty na znak przez co znaki z numerami powyżej 65 535 nie są uwzględnione
7. Domyślne kodowanie dla C to kodowanie zleżne od systemu; linux: *czterobajtowe!*.

— powinniśmy korzystać z typu **wchar_t** (ang. „wide character"), jednak jeśli chcemy udostępniać kod źródłowy programu do kompilacji na innych platformach, powinniśmy ustawić odpowiednie parametry dla kompilatorów, by rozmiar był identyczny niezależnie od platformy.
— korzystamy z odpowiedników funkcji operujących na typie char pracujących na **wchar_t** (z reguły składnia jest identyczna z tą różnicą, że w nazwach funkcji zastępujemy „str" na „wcs" np. *strcpy — wcscpy*; *strcmp — wcscmp*)
— jeśli przyzwyczajeni jesteśmy do korzystania z klasy **string**, powinniśmy zamiast niej korzystać z **wstring**, która posiada zbliżoną składnię, ale pracuje na typie **wchar_t**.

```
1 #include <stdio.h>
2 #include <string.h>
3 int main(void)
4 {
5     char napis1[30] = "Ala ma kota";
6     char napis2[30] = "Ala ma kotę";
7     printf( "napis1: %d\n", (int) strlen(napis1) );
8     printf( "napis2: %d\n", (int) strlen(napis2) );
9     return 0;
10 }
```

Wyniki:

```
napis1: 11
napis2: 12
```

11.11. Polskie literki

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <wchar.h>
```

```
4  int main(void)
5  {
6      char napis1[30] = "Ala␣ma␣kota";
7      char napis2[30] = "Ala␣ma␣kotę";
8      wchar_t napis3[12] = L"Ala␣ma␣kotę";
9      printf( "napis1:␣%d\n", (int) strlen(napis1) );
10     printf( "napis2:␣%d\n", (int) strlen(napis2) );
11     printf( "napis3:␣%d,␣sizeof␣napis3␣%d\n",
12             (int) wcslen(napis3),
13             (int) sizeof( napis3 ) );
14     return 0;
15 }
```

Wyniki

```
napis1: 11
napis2: 12
napis3: 11, sizeof napis3 48
```

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <wchar.h>
4  int main(void)
5  {
6      int i;
7      wchar_t m[2] = L"A";
8      union ccc{
9          char n[8];
10         wchar_t N[2];
11     } c;
12     c.N[0] = m[0];
13     c.N[1] = m[1];
14     for (i=0; i<8; i++)
15     printf("%d␣-␣%x\n", i, c.n[i]);
16     return 0;
17 }
```

Wynik działania programu

```
0 - 41
1 - 0
2 - 0
```

```
3 - 0
4 - 0
5 - 0
6 - 0
7 - 0
```

```c
#include <stdio.h>
#include <string.h>
#include <wchar.h>
int main(void)
{
    int i;
    wchar_t m[2] = L"Ą";
    union ccc{
        char n[8];
        wchar_t N[2];
    } c;
    c.N[0] = m[0];
    c.N[1] = m[1];
    for (i=0; i<8; i++)
    printf("%d - %x\n", i, c.n[i]);
    return 0;
}
```

Wynik działania programu

```
0 - 4
1 - 1
2 - 0
3 - 0
4 - 0
5 - 0
6 - 0
7 - 0
```

11.12. Locale

1. *Locale* to zestaw definicji określających specyficzny dla różnych języków (i krajów) sposób prezentacji różnych informacji.
2. Z jakichś dziwnych powodów problem nie ma zbyt bogatej „literatury”.

2020-05-20 08:40:00 +0200

3. Jeżeli ktoś chce się zapoznać z pewną realizacją uniksową — mogę polecić bardzo stary tekst System Operacyjny HP-UX a sprawa polska.

Zachowanie systemu po wyborze określonego języka zmienia się. Za pomocą kilku zmiennych środowiska można regulować zakres w jakim podporządkowujemy się specyficznym regułom. I tak:

LANG określa używany język,
LC_CTYPE definiuje charakter (litery, cyfry, znaki przestankowe, znaki drukowalne) poszczególnych znaków,
LC_COLLATE określa sposób sortowania,
LC_MONETARY opisuje zaznaczania jednostek monetarnych (symbol waluty, gdzie jest on umieszczany...),
LC_NUMERIC sposób zapisu liczb, znak oddzielający grypy cyfr, znak oddzielający część całkowitą od ułamkowej,
LC_TIME postać podawania daty i czasu,
LC_MESSAGES język używany w wyświetlanych komunikatach,
LC_ALL wszystkie LC_*.

1. Sprawa nie jest prosta!
2. Można wszystko zrobić na wiele sposobów.
3. Bardzo często komunikaty wyprowadzanie przez (różne) programy powtarzają się.
4. Powstaje myśl stworzenia „bazy danych" zawierających różne komunikaty oraz ich tłumaczenia na różne języki.
5. Są też gotowe biblioteki zapewniające obsługę takiej bazy danych (oraz jej tworzenie).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <wchar.h>
5 #include <locale.h>
6 int main(void)
7 {
8 /*     printf("%s\n", getenv("LANG")); */
9       setlocale(LC_ALL, getenv("LANG"));
10      printf("%d␣\n", wcscoll(L"żółć", L"żółw"));
11      return 0;
12 }
```

Wyniki

```
$ LANG=C ./stringi
1
$ LANG=pl_PL.utf8 ./stringi
-278
```

11.13. Operacje na łańcuchach znaków – najważniejsze fakty

1. Łańcuchem znaków (ang: *string*) jest każdy napis zawarty w cudzysłowach.
2. Łańcuch znaków jest **tablicą** typu **char**, w której na ostatniej pozycji znajduje się znak null (znak o kodzie zero).
3. Język C nie robi jakichś wielkich rozróżnień między znakami a cyframi.
4. Plik nagłówkowy `string.h` zawiera definicje stałych i funkcji związanych z operacjami na łańcuchach tekstów.
5. „Ułomność" podstawowych funkcji manipulacji znakami, na przykład:
   — „*Do obowiązku programisty należy zapewnienie aby tablica będąca pierwszym parametrem miała wystarczającą ilość miejsca na pomieszczenie całego zapisu!*"
   — „*Znaki z drugiego parametry kopiowane są do końca łańcucha, chyba że tekst jest dłuższy niż liczba znaków do skopiowania; w tym przypadku programista musi dodać znak NULL „ręcznie" na końcu skopiowanego tekstu.*"
6. Jeżeli ograniczyć się do kodowania jednobajtowego (255 znaków) nie ma właściwie żadnych problemów (ze znakami narodowymi), poza tym, że pogram musi być uruchamiany w odpowiednim środowisku. Dostępne kodowania znaków to ISO-8859-n (n od 1 do 9), cp-12xx, i tak dalej, i tak dalej...
7. Standardowe funkcje porównywania ciągów znaków nie będą działały!

12. Programy pomocnicze: diff, make, systemy rcs i cvs, debugger. Zarządzanie wersjami.

12.1. Co jest potrzebne programiście?

1. Umiejętność logicznego myślenia.
2. Ukończone kursy dokształcające.
3. Motywacja do pracy.
4. Komputer.
5. Zadanie.
6. Kompilator (program, który kod źródłowy przetłumaczy na na kod maszynowy).
7. Jakiś edytor (program pozwalający na wygodne wpisywanie kodu źródłowego).

12.2. Jak to robiono kiedyś?

1. Czas komputera był drogi, a dostęp do niego limitowany.
2. Dostęp on-line był wyjątkiem.
3. Gdy algorytm był gotowy — rozpoczynało się jego kodowanie (na papierze).
4. Następnie przenoszono kod na nośnik maszynowy (taśma papierowa, karty perforowane, taśma magnetyczna).
5. Nośnik z kodem źródłowym (w centrum obliczeniowym) był czytany przez kompilator, który tłumaczył na postać maszynową; w razie błędów były one zaznaczane na wydruku.
6. W przypadku błędów formalnych — koder poprawiał je i zaczynał cykl od początku.

7. Gdy błędów nie było — następowało uruchomienie programu (na dostarczonych danych).
8. Wyniki otrzymywał programista i sprawdzał czy są zgodne z oczekiwaniami. Jeżeli nie — rozpoczynała się żmudna analiza algorytmu. Program był uruchamiany po raz kolejny na danych testowych. Żeby ułatwić sobie pracę dodawano „wydruki kontrolne".
9. I tak do skutku.

12.3. Praca w środowisku interaktywnym

1. Najpierw odpadły „nośniki maszynowe" (choć w okresie przejściowym znacznie taniej było wpisać kod na prymitywnym urządzeniu na kartach lub tasiemce papierowej niż blokować dostęp do deficytowego terminala komputera i linii telekomunikacyjnej.
2. Pojawiły się środowiska ułatwiające uruchamianie programów w trybie interaktywnym.
3. Pojawiły się języki konwersacyjne.

12.4. Jak jest dziś?

Ci z Państwa, którzy programują — wiedzą doskonale. Inni...

12.5. Jak jest tworzony duży program?

1. Podzielony jest na kawałki (moduły obejmujące pewne dobrze zdefiniowane „całości").
2. Grupy modułów tworzące pewne całości — grupowane są w „biblioteki".
3. Wspólne definicje grupowane są w „plikach nagłówkowych".

12.6. Kompilacja

Tworzenie programu jest procesem wieloetapowym.

1. Fragmenty programu mogą być pisane w jakimś **metajęzyku**, który tłumaczony jest na kod źródłowy.

2. Program w kodzie źródłowym tłumaczony jest na postać wynikową (zazwyczaj nazywa się to *object*).
3. Niektóre moduły wynikowe grupowane są w biblioteki.
4. Moduły wynikowe i biblioteki używane są do budowy programu wykonywalnego.

12.7. Schemat

```
metajęzyk1 --> kod źródłowy1 --> object1 +
             + kod źródłowy2 --> object2 +--> biblioteka1 +
             | kod źródłowy3 --> object3 +                ---->exe
             + kod źródłowy4 --> object4 -------------->  +
pliki nagł---+
```

12.8. Program make

Jakakolwiek zmiana w plikach z kodem metajęzyka, plikach nagłówkowych lub z kodem źródłowym wymaga przekompilowania części lub wszystkich plików.

Gdy projekt jest bardzo duży — problem jest bardzo poważny.

Wymyślono program make pozwalający ułatwiający programiście życie.

Program korzysta ze specjalnego pliku (tradycyjnie nazywa się on Makefile) który opisuje strukturę projektu.

12.9. Makefile

Plik Makefile definiuje czynności jakie należy wykonać aby z pliku jednego typu uzyskać plik wynikowy oraz zależności pomiędzy plikami.


```
metajęzyk1 --> kod źródłowy1 --> object1 +
             + kod źródłowy2 --> object2 +--> biblioteka1 +
             | kod źródłowy3 --> object3 +                ---->exe
             + kod źródłowy4 --> object4 -------------->  +
pliki nagł---+
```


W naszym przypadku exe zależy od biblioteka1 i object4. biblioteka1 zależy od object1, object2, object3.

object1 zależy od kod źródłowy1.

kod źródłowy1 zależy od metajęzyk1.

object2 zależy od kod źródłowy2

object3 zależy od kod źródłowy3
kod źródłowy2 zależy plik nagł

12.10. Przykładowy plik Makefile

```
exe:     biblioteka1, object4
         link -o exe object4 -L biblioteka1

biblioteka1:     object1, object2, object3
                 ar -o biblioteka1 object1 object2 object3

object1:     kod źródłowy1
             kompiluj kod źródłowy1

object2:     kod źródłowy2, plik nagł
             kompiluj kod źródłowy2

object3:     kod źródłowy3, plik nagł
             kompiluj kod źródłowy3

object4:     kod źródłowy4, plik nagł
             kompiluj kod źródłowy4

kod źródłowy1:   metajęzyk1
                 metakompiluj metajęzyk1 -o kod źródłowy1

all:     exe

clean:
         kasuj object* biblioteka1 exe plik źródłowy1
```

12.11. Metajęzyki

1. Metajęzyki to języki jeszcze wyższego poziomu niż języki typu C, Pascal, Fortan.

2. Pozwalają one za pomocą stosunkowo prostych zależności (i bez przejmowania się szczegółami) stworzyć kod źródłowy programu.
3. Przykład. Chcemy napisać program, który będzie rozróżniał *liczby* od *słów*.
   — Najpierw precyzyjnie musimy zdefiniować co to jest **liczba**. Liczba to jedna lub więcej cyfr z zakresu od 0 do 9. W kategoriach wyrażeń regularnych można to zapisać tak: [0123456789]+ lub [0-9]+ (plus oznacza tu **jedno** lub więcej powtórzeń).
   — Natomiast **słowo** to jeden lub więcej znaków, z których pierwszy jest literą i który nie zawiera odstępów i innych znaków specjalnych. Zapisujemy to tak: [a-zA-Z][a-zA-Z0-9]* (gwiazdka oznacza tu **zero** lub więcej powtórzeń.

12.11.1. flex

Idea programu jest następująca:

```
1     %%
2     [0123456789]+              printf("NUMBER\n");
3     [a-zA-Z][a-zA-Z0-9]*       printf("WORD\n");
4     %%
```

i sprowadza się do następującego: „jak zobaczysz liczbę — wypisz **liczba**, jak zobaczysz wyraz — wypisz **wyraz**”

A sam program niewiele bardziej skomplikowany:

```
1 %{
2 #include <stdio.h>
3 %}
4
5 %%
6 [0123456789]+              printf("NUMBER\n");
7 [a-zA-Z][a-zA-Z0-9]*       printf("WORD\n");
8 %%
```

Kompilacja:

```
1 flex -o test.c test.l
2 ~/c$ ls -l test.c
3 -rw-r--r-- 1 myszka myszka 41749 2008-05-26 15:50 test.c
4 gcc test.c -o test -ll
```

Uruchomienie

```
~/c$ ./test
ala ma 3 koty
WORD
 WORD
 NUMBER
 WORD
```

12.12. Bardziej skomplikowany przykład: kalkulator

1. Chodzi mi o pokazanie jednie pewnej idei, a nie wnikanie w głębokie szczegóły.
2. Kalkulator składa się z dwu części:
   a) wprowadzanie danych
   b) przetwarzanie danych
3. Do zbudowania analizatora danych wykorzystamy program **lex** (lub jego darmowy odpowiednik **flex**)
4. Do zbudowania części przetwarzającej dane wykorzystamy program **yacc** (lub jego darmowy odpowiednik **bison**)

12.12.1. Wprowadzanie danych

```
1      /* calculator #1 */
2  %{
3      #include "y.tab.h"
4      #include <stdlib.h>
5      void yyerror(char *);
6  %}
7
8  %%
9
10 [0-9]+       {
11                  yylval = atoi(yytext);
12                  return INTEGER;
13              }
14
```

```
15 [-+\n]       { return *yytext; }
16
17 [ \t]        ;       /* skip whitespace */
18
19 .            yyerror("Unknown character");
20
21 %%
22
23 int yywrap(void) {
24     return 1;
25 }
```

12.12.2. Przetwarzania

```
1  %{
2      #include <stdio.h>
3      int yylex(void);
4      void yyerror(char *);
5  %}
6
7  %token INTEGER
8
9  %%
10
11 program:
12         program expr '\n'          { printf("%d\n", $2); }
13         |
14         ;
15
16 expr:
17         INTEGER
18         | expr '+' expr            { $$ = $1 + $3; }
19         | expr '-' expr            { $$ = $1 - $3; }
20         ;
21
22 %%
23
```

wer. 16 z drobnymi modyfikacjami!

```
24 void yyerror(char *s) {
25     fprintf(stderr, "%s\n", s);
26 }
27
28 int main(void) {
29     yyparse();
30     return 0;
31 }
```

12.13. Czterodziałaniowy kalkulator z nawiasami

12.13.1. Dane

```
1 %{
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include "calc1.h"
5 void yyerror(char*);
6 extern int yylval;
7 %}
8 %%
9 [ \t]+       ;
10 [0-9]+      {yylval = atoi(yytext);
11              return INTEGER;}
12 [-+*/]      {return *yytext;}
13 "("         {return *yytext;}
14 ")"         {return *yytext;}
15 \n          {return *yytext;}
16 .           {char msg[25];
17              sprintf(msg,"%s␣<%s>",
18                   "invalid␣character",yytext);
19              yyerror(msg);}
```

12.13.2. Przetwarzanie

2023-04-16 13:56:42 +0200

```
%{
#include <stdlib.h>
#include <stdio.h>
int yylex(void);
#include "calc1.h"
%}
%token INTEGER
%%
program:
        line program
        | line

line:
        expr '\n' { printf("%d\n",$1); }
        | '\n'

expr:
        expr '+' mulex { $$ = $1 + $3; }
        | expr '-' mulex { $$ = $1 - $3; }
        | mulex { $$ = $1; }

mulex:
        mulex '*' term { $$ = $1 * $3; }
        | mulex '/' term { $$ = $1 / $3; }
        | term { $$ = $1; }

term:
        '(' expr ')' { $$ = $2; }
        | INTEGER { $$ = $1; }
%%
void yyerror(char *s)
{
 fprintf(stderr,"%s\n",s);
 return;
}
int main(void)
{
```

```
38  /*yydebug=1;*/
39  yyparse();
40  return 0;
41 }
```

12.14. Narzedzia pomocnicze

1. **diff** program pozwalający porównywać dwa pliki źródłowe
2. **patch** program pozwalający na podstawie różnic (raportowanych przez diff) wprowadzić poprawki do plików źródłowych. Pozwala to rozpowszechniać znacznie mniejsze pliki.
3. **git**, **svn**, **bazaar**, **rcs**, **cvs**,... — systemy zarządzania wersjami.
4. **debuger** — program pozwalający uruchamiać programy krok po kroku, sprawdzać zawartość zmiennych,...

12.15. configure

— Pliki dla programu make mogą być przygotowywane ręcznie.
— Gdy tworzymy wersję programu która powinna być zbudowana w (nieomal) dowolnym środowisku, może się okazać, że budowa programu wymaga spełnienia pewnych dodatkowych zależności (które nie są spełnione standardowo).
— W takim przypadku warto skorzystać z dodatkowych narzędzi pomocniczych. Jednym z nich jest **Autotools**.
— Narzędzie to składa się z szeregu skryptów, które po uruchomieniu sprawdzają w jakim środowisku budowane jest oprogramowanie, starają się zidentyfikować zależności od dodatkowych pakietów oprogramowania (bibliotek) i tworzą skrypt (program), który uruchomiony na systemie docelowych sprawdza czy wszystkie zależności są spełnione.

13. Język C — próba podsumowania

13.1. Ogólne

1. Bardzo już stary — powstał w latach 1969–1973.
2. Język „wysokiego poziomu".
3. Zaliczany do grupy języków „proceduralnych" a czasami „imperatywnych strukturalnych języków programowania".
4. Bardzo popularny: ciągle opierają się na nim wszystkie dystrybucje uniksopodobne (w tym i Linux).
5. Microsoft preferuje język C++.

13.2. Słowa kluczowe

1. Słowa kluczowe są **zastrzeżone**.
2. Jest ich stosunkowo niewiele:
   - ***auto*** — **break** — **case** — **char**
   - **const** — **continue** — **default** — **do**
   - **double** — **else** — ***enum*** — ***extern***
   - **float** — **for** — **goto** — **if**
   - **int** — **long** — ***register*** — **return**
   - **short** — **signed** — **sizeof** — **static**
   - **struct** — **switch** — **typedef** — **union**
   - **unsigned** — **void** — ***volatile*** — **while**
3. Tymi zaznaczonymi kursywą się nie zajmowaliśmy!
4. Komentarz */* dowolny napis */* czasami też *// napis*

13.3. Podstawowe typy danych

char	1 bajt	
unsigned char	1 bajt	
signed char	1 bajt	
int	2 lub 4 bajty	
unsigned int	2 lub 4 bajty	
short int	2 bajty	
unsigned short int	2 bajty	
long int	4 bajty	
unsigned long int	4 bajty	
long long int	8 bajtów	tylko w nowych wersjach
unsigned long long int	8 bajtów	tylko w nowych wersjach
float	4 bajty	
double	8 bajtów	
long double	8, 10 lub 12, a nawet 16 bajtów	
void		

13.4. Operacje

Operacje

Wszystkie operacje (operatory) dostępne w C można podzielić na następujące grupy:

1. arytmetyczne
2. przypisania
3. logiczne
4. bitowe (działające na bitach)
5. pozostałe

Operatory arytmetyczne

Operatory arytmetyczne

— dodawanie +
— odejmowanie −
— mnożenie ∗

— dzielenie /
— reszta z dzielenie % (modulo) (*Tylko dla liczb typu całkowitego!*)
— zwiększenie ++
— zmniejszenie −−

Operatory przypisania

1. Oprócz najzwyklejszego operatora przypisania (=) używanego w kontekście:
$$a = b$$
co czytamy *zmiennej a przypisz wartośc zmiennej b*, czyli *od prawej do lewej!*
2. Występują operatory „złożone” $+=$ $-=$ $*=$ $/=$ $\%=$ $<<=$ $>>=$ $\&=$ $\hat{}=$ $|=$ stosowane w następujący sposób ($\odot$ oznacza jeden z symboli $+, -, *, /\ldots$)
$$a \odot= b$$
co czyta się
$$a = a \odot b$$

Operatory logiczne

Operatory logiczne

1. == równy
2. != nie równy
3. > większy
4. < mniejszy
5. >= większy lub równy
6. <= mniejszy lub równy
7. && logiczne I (AND)
8. || logiczne LUB (OR)
9. ! logiczne NIE (NOT)

Operatory logiczne

Uwagi

1. W języku C **nie ma** typu logicznego!

2. Zatem operator może i jest typu **logicznego**, ale zwraca wartości **arytmetyczne**!
3. Z definicji numeryczną wartością wyrażenia logicznego lub relacyjnego jest **1** jeżeli jest ono prawdziwe lub **0** jeżeli nie jest prawdziwe.
4. W operatorach logicznych **każda wartość różna od zera** traktowana jest jako prawda; zero to fałsz.
5. Operatory logiczne mają priorytet niższy od operatorów arytmetycznych; dzięki temu wyrażenie `i < lim-1` jest rozumiane właściwie jako `i < (lim-1)`.

Operatory logiczne
&& *i* ||

1. Wyrażenia połączone tymi operatorami oblicza się od strony lewej do prawej.
2. Koniec obliczania następuje natychmiast po określeniu wyniku jako „prawda” lub „fałsz”.
3. Wiele programów korzysta z tego faktu. (*Hakerstwo!*).
4. Priorytet operatora && jest wyższy od priorytetu operatora ||.
5. Priorytety obu operatorów są niższe od priorytetów operatorów relacji i porównania.

Operatory inne

Operatory inne

1. `sizeof()` wielkość obiektu/typu danych
2. & Adres (operator jednoargumentowy)
3. ∗ Wskaźnik; operator „wyłuskania” (jednoargumentowy)
4. ? Wyrazenie warunkowe
5. : Wyrazenie warunkowe
6. , Operator serii

13.5. Zmienne i typy

1. Deklaracja zmiennej — bardzo prosta:

```
1 typ nazwa;
```

2. Nazwa musi zaczynać się od litery, może zawierać również cyfry i znak podkreślenia (który traktowany jest jak litera).
3. Typy pochodne:
   a) typ wyliczeniowy:

```
1 enum nazwa { jeden, dwa };
```

   Może pojawić się pytanie w jaki sposób przedstawiane są zmienne typu wyliczeniowego (tao znaczy jak w powyższym przykładzie pamiętana jest wartość „jeden” a jak „dwa”)? Standardowo pierwsza nazwa ma liście otrzymuje wartość 0, druga 1 i tak dalej. Można to zmienić:

```
1 enum miesiac {
2     STY =  1, LUT =  2, MAR =  3, KWI =  4,
3     MAJ =  5, CZE =  6, LIP =  7, SIE =  8,
4     WRZ =  9, PAZ = 10, LIS = 11, GRU = 12
5 }
```

   b) struktury:

```
1 struct nazwa {
2     typ1 nazwa1;
3     typ2 nazwa2;
4 };
```

   c) Unie

```
1 union nazwa {
2     typ1 nazwa1;
3     typ2 nazwa2;
4 };
```

   d) Pola bitowe

```
1 typ [identyfikator] : dlugosc;
```

   e) Tablice

```
1 typ nazwa[liczba];
```

   f) Wskaźniki

```
1 typ *nazwa;
2 typ **nazwa;
```

```
3 typ_zwracany (*nazwa_wsk_do_funkcji)\
4                          (typ nazwa_param1,\
5                           typ nazwa_param2,...);
```

Dla każdego typu może wystąpić wskaźnike „tego typu". Mimo, że za każdym razem są to adresy pamieci — różnią się między sobą ze względu na zasady arytmetyki.

Wskaźnik **typ** *nazwa może być czasami używany do obsługi tablic (jednowymiarowych), ale nie można go utożsamiać z tablicą. Deklaracja:

```
1 int A[30];
```

czyni zmienną A stałą wskaźnikową i nadaje jej taką wartość, że wskazuje na obszar w pamieci operacyjnej przeznaczony do przechowania tablicy A. Natomiast deklaracja:

```
1 int *a;
```

deklaruje zmienną wskaźnikową a, ale nie nadaje jej żadnej konkretnej wartości, i zmienna żeby mogłą być traktowana jako tablica powinna wskazywać na obszar pamięci, który może służyć do przechowywania danych. Można to uzyskać albo prosząc system operacyjny o przydzielenie pewnej pamięci (funkcja malloc) albo nadając zmiennej wartość równą adresowi jakiejś tablicy:

```
1 a = A;
```

Po wykonaniu powyższej operacji a będzie „aliasem"[1] do tablicy A.

13.5.1. Zajętość pamięci

Bardzo przydatne polecenie języka C pozwala zapytać o wielkość obiektu:

— **sizeof**(<*zmienna*>)
— **sizeof**(<*typ*>)
— **sizeof**(<*stała*>)

[1] Inną nazwą tablicy.

13.5.2. Logiczne

1. Typ logiczny nie istnieje (w podstawowej wersji języka C — ANSI)!
2. Istnieją operatory, które normalnie powinny dawać wynik logiczny: `&&` i `||`
3. Istnieją polecenia, które gdzie indziej korzystają ze zmiennych i wyrażeń typu logicznego.
4. Prawda (true)
   — na „wejściu” (argument!) każda wartość różna od zera
   — na „wyjściu” (wynik) jeden (zazwyczaj – nie zawsze!)
5. Fałsz (false)
   — na wejściu i wyjściu — zero

Zmienne i typy – różne takie

1. Podstawowy typ zmiennoprzecinkowy: **double**
2. To jest stała typu **double**: 3.141592365
3. To jest stała typu **float**: $3.141592365F$
4. To jest stała typu **int**: 1234
5. To jest stałą typu **unsigned**: $1234U$
6. To jest stała typu **long**: $1234L$
7. To jest stała tycpu **char**: "Ala ma kota”
8. To jest stałą typu wchar_t: L"Ala ma małpę"

13.6. Konwersje typów

1. Niejawne (czyli niezadeklarowane):
   — Typ „mniejszy” jest **promowany** do „większego” w wyrażeniach dwuargumentowych (gdy argumenty mają różny typ!).
   — Typem wyniku jest typ „większy” (ale obliczenia wykonywane są — jeżeli nie ma istotnej potrzeby — bez dokonywania konwersji).
   — Typ float nie jest automatycznie przekształcany do double.
   — Kłopoty dla typów unsigned (omijam!).
   — Obiekt znakowy zmienia się w liczbę całkowitą (co ze znakiem??).
   — Dłuższe liczby całkowite są przekształcane do krótszych przed odcięcie „wystających” **znaczących** bitów.
2. Jawne (zadeklarowane):
   — wygląda tak (*nazwa typu*)*wyrażenie*

— Nazywa się to rzutem (ang: *cast*)

13.7. Instrukcje sterujące

1. Instrukcja **if**

```
if (warunek1) {
    instrukcje;
}
else if(warunek2){
    instrukcje;
}
else {
    instrukcje;
}
```

Instrukcja if to podstawowa instrukcja warunkowa w C — gdy warunek1 jest spełniony (**zwraca wartość niezerową**), wykonany zostanie kod zawarty w bloku ograniczonym klamrami. Instrukcje **else if** i **else** są opcjonalne, sprawdzane są wyłącznie, gdy podstawowy warunek nie jest spełniony.
Wykorzystywana jest idea „drogi na skróty". Gdy wyrażenie logiczne wygląda tak:

```
A && B && C && D
```

obliczenia prowadzone są tak długo, żeby móc jednoznacznie określić wynik.

2. Pętla **while**

```
while (wyrazenie) {
 instrukcje;
}
```

Pętla while — instrukcja wykonuje kod zawarty w bloku ograniczonym klamrami tak długo, dopóki jej warunek jest spełniony (ma wartość różną od zera). Instrukcja sprawdza warunek przed wykonaniem ciała pętli. Pętla **while** może wykonywać się nieskończoną ilość razy, gdy wyrażenie nigdy nie przyjmie wartości 0, może także nie wykonać się nigdy, gdy wartość przed pierwszym przebiegiem będzie zerowa.

3. Pętla do...while

```
do {
instrukcje;
}
while(warunek);
```

Pętla **do**...**while** jest podobna do pętli **while** z tą różnicą, że warunek sprawdzany jest po każdym wykonaniu pętli, a więc instrukcje w pętli zawsze wykonają się co najmniej raz.

4. Pętla for

```
for (wyr1; wyr2; wyr3) {
  instrukcje;
}
```

Pętla **for** jest rozwinięciem pętli **while** o instrukcję wykonywaną przed pierwszym obiegiem oraz dodatkową instrukcję wykonywaną po każdym przebiegu — najczęściej służącą jako licznik obiegów. Często zmienną liczącą kolejne wykonania ciała pętli nazywa się iteratorem.
Powyższa instrukcja jest równoważna rozwinięciu:

```
wyr1;
while (wyr2){
   instrukcja
   wyr3;
}
```

5. Instrukcja switch

```
switch (wyrazenie) {
      case wartosc1 :
         instrukcje;
         [break;]
      case wartosc2 :
         instrukcje;
         [break;]
      default :
         instrukcje;
         [break;]
}
```

Instrukcją decyzyjną **switch** zastąpić można wielokrotne wywoływanie instrukcji warunkowej **if** np. dla różnych wartości tej samej zmiennej — przykładowo, gdy zmienna może przyjąć 10 różnych wartości, a dla każdej z nich należy podjąć inne działanie. Należy pamiętać o **break**.

13.8. Funkcje

1. Definicja

```
[klasa_pamięci] [typ] nazwa([lista_argumentów])
{
  instrukcje;
  [return wartość;]
}
```

Klasa pamięci, określenie zwracanego typu oraz lista argumentów są opcjonalne. Jeżeli nie podano typu, domyślnie jest to typ liczbowy int, a instrukcję return kończącą funkcję i zwracającą wartość do funkcji nadrzędnej można pominąć. Listę argumentów tworzą wszystkie zmienne (zarówno przekazywane przez wartość jak i wskaźniki) wraz z określeniem ich typu. Dozwolona jest rekurencja, nie ma natomiast możliwości przeciążania funkcji.

2. Funkcja musi być zdefiniowana przed pierwszym jej użyciem (prototyp!).

Obfuscated C code

Co to jest?

```
_(__,____,_____){____/__<=1?_(__,____+1,_____):!(____%__)?_(__,____+1,0):___%__==___/
__&&!____?(printf("%d\t",____/__),_(__,____+1,0)):___%__>1&&___%__<___/__?_(__,1+
___,_____+!(____/__%(___%__))):___<__*__?_(__,____+1,_____):0;}main(){_(100,0,0);}
```

13.9. Kolejność (priorytet) operatorów

13.9.1. Podstawowe

1. () [] -> .
2. ! ~ ++ -- + - * & **sizeof**
3. * / %
4. + -

2022-05-27 07:25:47 +0200

5. << >>
6. < <= >= >
7. == !=
8. &
9. ^
10. |
11. &&
12. ||
13. ?:
14. = += -= *= /= %= &= ^= |= <<= >>=
15. ,

13.9.2. Operatory „przynależności”

1. Operator funkcji () (w05)
2. Tablica [] (w06)
3. Wskaźnik do struktury -> (w07, w08)
4. Element struktury . (w08)

13.9.3. Operatory jednoargumentowe (unarne)

1. ! Logiczne NIE
2. ~ Uzupełnienie „do jednego” (bitowa zamiana 0 na 1 a 1 na 0)
3. ++ Zwiększ (Uwaga: przyrostek i przedrostek!)
4. -- Zmniejsz (Uwaga: przyrostek i przedrostek!)
5. + Po prostu: x = +2
6. - Zmiana znaku liczby x = −2
7. * Wskaźnik do zmiennej
8. & Pobranie adresu
9. **sizeof** Operator zwracający ilość miejsca (w bajtach) zajmowaną przez zmienną albo typ danych

13.9.4. Operatory dwuargumentowe (binarne)

1. * Mnożenie (Co znaczy 2*+3)
2. / Dzielenie (Co znaczy 2/−3)
3. % Modulo

Uwaga: Dzielenie nie wyprowadza poza typ! (w wyniku dzielenia dwu liczb całkowitych (**int**) zawsze dostaniemy wartość całkowitą).

13.9.5. Operatory dwuargumentowe (binarne)

1. + Suma (3++3 — ŹLE!! 3+ +3)
2. − Różnica (3+−3)

13.9.6. Operatory bitowe

Po lewej stronie operatora co przesuwamy, po prawej o ile

1. << Bitowe przesunięcie w lewo
2. >> Bitowe przesunięcie w prawo

Przesunięcie w lewo o jeden równoważne jest pomnożeniu przez dwa. Przesunięcie w prawo o jeden równoważne jest podzieleniu przez dwa.

Uwaga: najlepiej działa z liczbami typu **unsigned int**.

−3 >> 1 w wyniku daje −2

+3 >> 1 w wyniku daje 1

13.9.7. Operatory relacji

1. <
2. <=
3. >=
4. >

1. == (Równe)
2. != (Różne)

Do przemyślenia
Jaki będzie wynik 3>5==5>7

13.9.8. Operatory bitowe

Operatory mają różne priorytety, wymienione w kolejności od najważniejszego do najmniej waznego

1. & (1 & 2 daje 0)
2. ^ (1 ^ 2 daje 3)
3. | (1 | 2 daje 3)

13.9.9. Operatory logiczne

Operatory mają różne priorytety, wymienione w kolejności od najważniejszego do najmniej waznego

1. && Iloczyn logiczny 4 && −4 daje 1
2. || Suma logiczna −4 || 0 daje 1

Kolejność operatorów

if then else

1. ?:

```
1 if ( x == 1 )
2     y = 10;
3   else
4     y = 20;
```

jest równoważne

```
1 y = (x == 1) ? 10 : 20;
```

```
1 if ( x == 1 )
2    puts("take car");
3  else
4    puts("take bike");
```

jest równoważne

```
1 (x == 1) ? puts("take car") :\
2            puts("take bike");
```

albo

```
1 puts( (x == 1) ? "take car" :\
2                  "take bike");
```

13.9.10. Operatory podstawienia

1. = operatr przypisania (podstawienia) a = b
2. += to znaczy a = a + b czyli a += b
3. −=
4. ∗=
5. /=
6. %=

7. &=
8. ^=
9. |=
10. <<=
11. >>=

13.9.11. Separartor

1. , (przecinek) — oddziela parametry funkcji albo dane

Wskaźnik do wskaźnika albo podwójny wskaźnik

1. Używany, na przykład wtedy, gdy chcemy aby funkcja zwracała jako jeden z parametrów wskaźnik
2. Przykład

```
1  #include     /* malloc */
2
3  void Func(char **DoublePtr);
4
5  main()
6  {
7       char *Ptr;
8       Func(&Ptr);
9  }
10
11 void Func(char **DoublePtr)
12 {
13      *DoublePtr = malloc(50);
14 }
```

14. Wskaźniki — próba podsumowania

14.1. Zmienne

1. Zmienna, to po prostu zmienna: miejsce w pamięci komputera do przechowywania jakiejś jednej wartości.
2. Każda zmienna ma jakiś adres, ale tymi adresami interesujemy się dosyć rzadko.

14.2. Tablice

1. Tablica to pojemnik do przechowywania wielu danych tego samego typu.
2. W komputerze — adres początku, typ elementu i numer elementu to wszystko co jest potrzebne aby dobrać się do konkretnego elementu.
3. adres(t[i]) = adres(t) + i $*$ długość elementu
4. Z tego powodu w C tablice indeksowane są od 0 (zera) — bo pierwsza (o numerze zerowym) wartość w tablicy umieszczana jest na samym jej początku).

Niezbędne jest drobne komentarze dotyczące tablic. Deklarując tablicę (dowolnego typu):

```
1 int A[N]
```

rezerwowany jest obszar pamięci odpowiedni do rozmiaru tablicy, zmiennej A przypisywany jest (na stałe) adres początku tego obszaru. Zatem A jest wskaźnikiem typu **int**.

Gdy gdziekolwiek w kodzie programu pojawi się napis typu $\alpha[\beta]$ należy rozumieć go (niezależnie od kolejności argumentów) jako $*(\alpha+\beta)$. Oczywiście arytmetyka (operacja dodawania) jest to arytmetyka wskaźników.

14.3. Adresy

1. Adres — wartość jak każda inna.
2. Właściwie jest to liczba całkowita (**long int**), ale raczej nie należy (bezmyślnie) dokonywać podstawień typu **long int** → **int** ∗ lub odwrotnie!.
3. Inaczej działą arytmetyka na liczbach **long int**, a inaczej na wskaźnikach!
4. Do przechowywania adresów są potrzebne specjalne zmienne zwane „wskaźnikami" (*pointer*).
5. Do pobrania adresu obiektu służy operator jednoargumentowy & (*apersand*).
6. W języku C czym innym jest adres zmiennej int, czym innym adres zmiennej char, czym innym adres zmiennej double...
7. Do deklaracji wskaźników używamy specjalnego znaczka „∗" (przed nazwą).
8. Żeby utrudnić wszystkim życie wymyślono też wskaźniki bez podania typu (**void**).
9. Na adresach (i zmiennych zawierających adresy) czyli wskaźnikach można wykonywać proste operacje: dodawanie i odejmowanie stałej oraz odejmowanie adresów tego samego typu.

14.4. Użycie wskaźników

1. Pobranie wartości (występuje zazwyczaj po prawej stronie znaku równości): ... = ∗ip.
2. Podstawienie wartości: ∗ip = ...

14.4.1. Adresowanie pośrednie

1. Adresowanie pośrednie czyli wpisanie jakieś wartości do miejsca pamięci wskazywanego przez adres (lub zmienną przechowującą adres):
   ∗ip = 7;

14.5. Funkcje

1. Funkcje nie mają wiele wspólnego ze wskaźnikami.

Natomiast wspomieć należy, że każda funkcja musi być zapisana (jej binarny obraz) po kompilacji gdzieś w pamięci. I warto wiedzieć, że nazwa funkcji (podobnie jak nazwa tablicy) to wskaźnik.

2. Gdy argumentem funkcji jest zmienna, do wnętrza funkcji przekazywana jest jej wartość.
3. Gdy argumentem funkcji jest wyrażenie — do funkcji przekazywana jest jego wartość.
4. Co jest drugim argumentem funkcji scanf("%d", &i) (Nie zajmujmy się funkcją scanf, jest zbyt skomplikowana!) Co jest argumentem funkcji moja_funkcja(&i)?
5. Argumentem jest wyrażenie polegające na pobraniu adresu zmiennej i. Adres zmiennej i jest przekazywany do wnętrza funkcji!
6. Deklaracja funkcji powinna wyglądać jakoś tak: moja_funkcja(int *ip);
7. Co funkcja może zrobić z przekazanym jej adresem? Niezbyt wiele, ale zawsze może użyć go w celu adresowania pośredniego, czyli zrobić coś takiego:
   $*ip = 127;$
   albo
   $*ip = (*ip) + 1;$
8. Parametrem funkcji może być element tablicy:
   inna_moja_funkcja(tablica[7])
   do wnętrza funkcji przekazywana jest wartość wyrażenia polegającego na pobraniu z tablicy jej siódmego elementu.
9. Argumentem funkcji może być nazwa tablicy:
   inna_funkcja(tablica)
   Żeby to miało sens, deklaracja funkcji musi wyglądać jakoś tak:
   inna_funkcja(**int** t[ ]);
   Jeżeli dodatkowo funkcja znać będzie jeszcze długość tablicy — będzie mogła wykonywać na niej dowolne operacje...
   Załóżmy, że chcemy napisać funkcję, która będzie zerowała (wypełniała zerami) zadaną tablicę. Do funkcji przekażemy nazwę tablicy (czyli adres jej początku) oraz jej długość.
   Funkcję można zaprogramować na kilka różnych sposbów. W pierwszym przypadku nie użyjemy (bezpośrednio) wskaźników.

```
1 void zerowanie(int t[ ], int n)
2 {
3     int i;
```

wer. 16 z drobnymi modyfikacjami!

```
4       for(i = 0; i < n; i++)
5           t[i] = 0;
6   }
```

W drugim przypadku, wykorzystamy wskaźniki, w deklaracji parametrów funkcji użyjemy zmiennej **int** ∗ it.

```
1   void zerowanie(int *it, int n)
2   {
3       int i;
4       for(i = 0; i < n; i++)
5           *(it + i) = 0;
6   }
```

Trzeci przypadek jest „mieszany”. Deklarujemy zmienną it jako adres (wskaźnik) ale do elementów tablicy odwołujemy się w sposób „klasyczny” — korzystamy z zapisu z indeksem.

```
1   void zerowanie(int *it, int n)
2   {
3       int i;
4       for(i = 0; i < n; i++)
5           it[i] = 0;
6   }
```

W pewnym sensie wszystkie zapisy są sobie równoważne.

Tablice po raz drugi

1. Tablice muszą być deklarowane.
2. W deklaracji trzeba podać ich długość.
3. Co prawda taki kod jest poprawny:

```
1   int n;
2   n = 10;
3   int tablica[n]
```

ale jego działanie jest ograniczone do niezbyt wielkich n! Mi udało się, zadeklarować tablicę o 2 milionach elementów, ale już nie o 2 100 000 elementów.
4. Znacznie lepsze rozwiązanie jest takie:

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int n, i;
    printf("Podaj n ");
    scanf("%d", &n);
    int *tablica;
    tablica = malloc(n * sizeof(int));
    if (tablica != NULL) {
        for (i = 0; i < n; i++)
            tablica[i] = i;
/*  Tu reszta programu */
        free(tablica);
        return 0;
    } else {
        printf("Pomocy! Nie dostalem " \
               "pamieci!\n");
        return 1;
    }
}
```

14.6. Parę pokręconych przykładów

W dalszej części będzie kilka przykładów z dosyć szczegółowym opisem tego co (i czemu) robią. Osobna kwestia *po co to robią* nie będzie rozpatrywana.

14.6.1. Przykład pierwszy

```c
int * f1(int N)
{
        int tab[N];
        return tab;
}
```

```
int main(int argc, char **argv)
{
        int * T;
        int i;
        T = f1(1000);
        printf("%p\n", T);
        for(i=0;i < 1000; i++)
                T[i] = -1;
        return 0;
}
```

Idea jest taka: chcemy stworzyć funkcję, która korzystając ze standardowych deklaracji tablic w języku C stworzy tablicę o zadanym rozmiarze, a jej adres początkowy zwróci do funkcji wywołującej.

Ze względu na lokalność zmiennych automatycznych — nie ma szans to działać. W funkcji f1 zostanie utworzona tablica, natomiast po „wyjściu" z funkcji przestanie ona istnieć. Zatem wartość zwracana przez funkcję wskazuje na **nic**. Pokazuje to kolejny przykład.

Przykład pierwszy prim

```
void g1(int N)
{
        int T[N];
        printf("g1: %p\n", T);
}

void g2(int M)
{
        float T[M];
        printf("g2: %p\n", T);
}
```

```
int main(int argc, char **argv)
{
        g1(1000);
        g2(1000);
        return 0;
}
```

```
g1: 0x7ffcdd5f1d90
g2: 0x7ffcdd5f1d90
```

Jak widać, pamięć raz przydzielona w funkcji g1() jest zwalniana i ponownie przydzielana w funkcji g2(). Świadczy o ym ta sama wartość drukowanego adresu tablicy (wskaźnika)[1].

W zależności od wersjoi użytego kompilatora Zachowanie programu może byc różne. Wersja „stara" (obecna w chwli pisania tych słów w laboratorium 604 B1) podczas kompilacji zgłąsza ostrzeżenie brzmiące tak:
tab1.c:32:9: warning: function returns address of local variable
ale funkcja zwraca adres wskaźnika.

Wersja kompilatora współcześniejsza[2] zgłasza podobny komunikat, natomiast sama funkcja zwraca wartość (nil) (0 — zero). Skutecznie uniemożliwia to wykorzystanie takiego wskaźnika.

14.6.2. Przykład drugi

```
1  int * f3()
2  {
3          static int tab[1000];
4          return tab;
5  }
```

```
1  int main(int argc, char **argv)
2  {
3          int * T;
4          int i;
5          T = f3();
6          printf("%p\n", T);
7          for(i=0;i < 1000; i++)
8                  T[i] = -1;
9          return 0;
10 }
```

Powyższe rozwiązanie jest poprawne — ale ze względu, że tablica tab jest tablicą statyczną — nie moża zmieniać jej długości. Praktyczne znaczenie powyższego rozwiązania jest wątpliwe.

[1] Specyfikacja formatu %p służy do drukowania wartości wskaźników. Wyprowadzane są one w formie szesnastkowej.

[2] 6.2.0 20161005 (Ubuntu 6.2.0-5ubuntu12)

14.6.3. Przykład trzeci

```
int * f2(int N)
{
        int * tab =
         malloc(N * sizeof(int));
        return tab;
}
```

```
int main(int argc, char **argv)
{
        int * T;
        int i;
        T = f2(1000);
        printf("%p\n", T);
        for(i=0;i < 1000; i++)
                T[i] = -1;
        return 0;
}
```

Jedynie te rozwiązanie jest poprawne i przydatne — wykorzystuje funkcję malloc i tworzy tablicę o zadanej długości oraz zwraca jej adres.

14.6.4. Przykład czwarty

```
char * f4()
{
        char * tekst=
                "Ala␣ma␣kota";
        return tekst;
}
```

```
int main(int argc, char **argv)
{
        char * N;
        N = f4();
        printf("%p\n", N);
        printf("%s\n", N);
        N[1]='Z';
        return 0;
```

```
9 }
```

Mimo, że tekst „Ala ma kota” nie jest oznaczony jako **static**, funkcja f4() zadziała poprawnie zwracając adres do napisu. Po wyjściu z tej funkcji wskaźnik poprawnie wskazuje na ten napis. Co więcej napis „Ala ma kota” jest traktowany jako stała[3] i zapisywany w obszarze pamięci chronionej przed zmianami. Tak więc próba wykonania polecenia N[1] = ‘Z’; zakończy się błędem wykonania (*Segmentation fault*).

14.7. Małe podsumowanie

Zwracam uwagę, że identyczne z wyglądu zapisy miewają różne znaczenia.

14.8. Tablice znakowe

Tablice znakowe mogą sprawiać różne kłopoty. Warto pamiętać, że sposób w jaki obsługiwane są tablice znakowe jest identyczny, jak sposób obsługi tablic „numerycznych”.

Zapewne bardzo wygodnym byłby zapis:

```
1 int a[10];
2 int b[10];
3 for( i = 0; i < 10; i++)
4         a[i] = i * i;
5 b = a;
```

Chciałoby się oczekiwać, że linia 5 znaczy tyle co „zawartość tablicy a skopiuj do tablicy b. Niestety tak nie jest. Trzeba napisać sobie funkcję realizującą taką operację:

```
1 void copy_tab(int N, int * A, int * B)
2 {
3         int i;
4         for(i = 0; i < N; i++)
5                 *(B + i) = *(A + I);
6 }
```

[3] I jeżeli gdziekolwiek w programie użyjemy takiej stałej tekstowej — będzie to **ta sama** stała.

Tablice znakowe: problemy

Bardzo naturalny zapis:

```
1 char * a = "Ala ma kota";
2 char * b;
```

Oznacza on tyle, że deklarujemy dwie zmienne wskaźnikowe: a oraz b. Pierwszej z nich przypisujemy adres stałej znakowej zawierającej napis ‚Ala ma kota". Czy można napisać

```
1 b = a;
```

Tak. Zapis oznacza tyle, że adres (wskaźnik) zawarty w zmiennej a zostaje przepisany do zmiennej b. Oba wskaźniki będą wskazywały na te same miejsce.

Czy można napisać tak:

```
1 char c = a[5];
```

Tak. Piąty znak z napisu „Ala ma kota" („a" z wyrazu „ma") zostanie przypisany do zmiennej c.

A czy można napisać tak:

```
1 a[5] = 'm';
```

żeby wyraz „ma" zamienić na „mm"?

Nie. Co prawda a to „tablica", ale jej zawartość jest stałą. A stałych nie można zmieniać!

Tablice znakowe: problemy (cd)

Inny naturalny zapis:

```
1 char a[] = "Ala ma kota";
2 char b[100];
```

Oznacza on tyle, że deklarujemy dwie zmienne wskaźnikowe: a oraz b. Pierwszej z nich przypisujemy adres stałej znakowej zawierającej napis ‚Ala ma kota". Czy można napisać

```
1 b = a;
```

Nie. Zapis oznacza tyle, że adres (wskaźnik) związany z nazwą a chcemy przypisać do b. Ale a oraz b to stałe typu **char** ∗.

Czy można napisać tak:

```
1 char c = a[5];
```

Tak. Piąty znak z napisu „Ala ma kota” („a” z wyrazu „ma”) zostanie przypisany do zmiennej c.

A czy można napisać tak:

```
1 a[5] = 'm';
```

żeby wyraz „ma” zamienić na „mm”?

Oczywiście. 'm' nadpisze zawartą w tablicy literę 'a'.

Poza tym oba zapisy są właściwie równoważne. Ale, zwracam uwagę, że problemy zazwyczaj związane są z subtelnościami.

14.9. Problemy z sizeof

Na jednym z zajęć projektowych/laboratoryjnych zaproponowałem następujący trik pozwalający łatwo wyznaczyć rozmiar tablicy:

```
1 double x[] = {
2     0., 1., 2., 3., 4., 5.,
3 };
4 int n = sizeof ( x ) / sizeof ( double );
```

Pozwala to łatwo zwiększyć/zmniejszyć rozmiar tablicy :

```
1 double x[] = {
2     0., 1., 2., // 3., 4., 5.,
3 };
4 int n = sizeof ( x ) / sizeof ( double );
```

Dodany komentarz „ukrywa” część wartości i tablica ulega automatycznemu skróceniu. Łatwo dopisywać kolejne wartości. Bardzo przydatne zwłaszcza podczas testowania programów.

Dziwne zastosowania

Wiele osób uznało to za „uniwersalną” metodę wyznaczania rozmiaru tablic. I stosuje ją w bardzo różnych sytuacjach.

1. Do wyznaczania długości tekstów:

```
1 char a[] = "Ala␣ma␣kota";
2 int n = sizeof( a ) / sizeof( char );
```

a czasami nawet sprytniej:

```
1 int n = sizeof( a ) / sizeof( char ) - 1;
```

Lepiej użyć funkcji strlen!
A taki prosty programik robi to samo:

```
1 n = 0;
2 while (a[n]) n++;
```

2. Wewnątrz funkcji. Czyli jakoś tak:
W funkcji main mamy:

```
1 double a[] = {1., 2., 3., 4.,};
2 double avg = srednia( a );
```

A w funkcji:

```
1 double srednia (double x[])
2 {
3         double srednia = 0.;
4         int n = sizeof( x ) / sizeof( double );
5         int i;
6         for(i = 0; i < n; i++)
7                 srednia += x[i];
8         return srednia / n;
9 }
```

I cały problem polega na tym, że mierzymy nie to o czym myślimy. Porządny kompilator zgłosi taki komunikat:

```
warning: ‘sizeof’ on array function parameter ‘x’
will return size of ‘double *’ [-Wsizeof-array-argument]
  int n = sizeof( x ) / sizeof( double );
                  ^
```

Gdyż to, co przed chwilą, było tablicą — w funkcji jest tylko i wyłącznie **adresem** początku tablicy. Resztę załatwia magia wskaźników.
Jako dobrą zasadę możemy przyjąć, że jeżeli tworzymy funkcję operującą na tablicach — jednym z parametrów **musi być** rozmiar tablicy.

14.10. Zmiana przydziału pamięci

Nie jest to może najlepsze określenie, ale na razie nie znajduję nic lepszego. Chodzi o to, że w miarę potrzeby pamięcią przydzieloną z wykorzystaniem funkcji malloc można „gospodarować" — zmniejszając lub zwiększając jej ilość. W dalszej części podam prosty dosyć przykład takiego zastosowania funkcji realloc.

Zadanie jest takie:

— Wczytać mamy dane nieznanej długości.
  — Może to być linia tekstu.
  — Może to być strumień danych (nieznanej objętości), który należy wczytać w całości zanim będzie mógł być przetworzony.

W instrukcjach laboratoryjnych rozważane są różne pomysły jak problem rozwiązać, tu opiszę dokładniej jedno z rozwiązań.

Funkcja czytanie

```
1 /*
2  * napis.c
3  * Copyright 2016 wojciech myszka <myszka@norka.eu.org>
4  */
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 char * czytaj(void)
9 {
```

dN (linia 10) zawiera informację o „kwancie" przydzielanych bajtów. Może to być praktycznie dowolna wartość: gdy pamięć zostanie zapełniona, przydzielony zostanie następny kwant. Gdy odczyt danych się zakończy, niezapełniona danymi pamięć będzie zwolniona.

```
10 #define dN 10
11     int N = dN; // początkowy przydział
12     int i;
13 /*
14  * Przydzielamy funkcją malloc pamięć na dN znaków
15  */
16     char * bufor;
```

wer. 16 z drobnymi modyfikacjami!

```
17      bufor = (char *) malloc(N);
```

Po każdym wykonaniu funkcji malloc/calloc/realloc należy sprawdzić czy zostałą ona wykonana poprawnie. Jakikolwiek błąd powoduje, że funkcja zwraca wartość 0. W języku C zdefiniowana jest stałą NULL typu wskaźnikowego o wartości zero. Przyjęło się, że obszar pamięci ozaczynający się od adresu 0 jest zarezerwowany i niedostępny do programów.

Gdy wystąpi błąd (co raczej jest mało prawdopodobne) — funkcja napis zwraca wartość NULL co jest informacją dla programu wywołującego, że coś poszło źle.

```
18      if ( bufor == NULL )
19          return bufor;
```

Odczyt znaków wykonujemy w nieskończonej pętli (linia 21). Pętla się kończy gdy strumień danych wejściowych się skończy (warunek EOF w linii 24) lub napotkamy znak końca wiersza.

Rozpoznawanie końca wiersza ma sens gdy czytamy informację tekstową, poszczególne „rekordy" oddzielane są znakami \\n (naciśnięcia klawisza Enter). Program będzie działał gdy odrzucimy drugą część warunku w linii 24.

```
20      i = 0; // Liczba przeczytanych znaków
21      while ( 1 )
22      {
23          bufor[i] = getchar();
24          if ( bufor[i] == EOF || bufor[i] == '\n' )
```

Gdy uznajemy, że nie będzie już żadnych dodatkowych informacji — program najpierw „kończy" tekst znakiem o kodzie ASCII 0 (linia 26), a później zwalnia niewykorzystną pamięć używając funkcji realloc. Argumentem te funkcji jest liczba wykorzystanych bajtów pamięci (i przeczytanych i dodane zero, zatem i + 1).

```
25          {
26              bufor[i] = 0; // koniec tekstu
27              bufor = realloc(bufor, i + 1); // zwalniamy
28                                             // nadmiarową pamięć
29              return bufor;
30          }
31          i++;
```

Przeczytany znak został zapisany w pamięci, zatem zwiększamy i (linia 31) oczekując na kolejny znak. Gdy okaże się że grozi przekroczenie pamięci (linia 32) — będziemy musieli przydzielić kolejny kwant pamięci. Zmienna N (linia 36) gromadzi informacje o sumarycznym rozmiarze bufora i jest odpowiednio ukatualniana.

```
32          if ( i >= N ) // Czy skonczyla sie przydzielona
33                                                  // pamiec?
34          {
35              bufor = realloc(bufor, i + dN);
36              N += dN;
37              printf("%p %d\n", bufor, N);
38          }
39      }
40  }
```

Wydruk (polecenie printf w linii 37) ma charakter diagnostyczny. Aby uprościć kod nie sprawdzam, czy polecenie realloc wykonało się poprawnie (i funkcja zwróciła wartość różną od zera), ale w prawdziwych programach powinno to się robić.

```
41  int main(int argc, char **argv)
42  {
43      char * tekst;
44      char bufor[10];
45      tekst = czytaj();
46      if ( tekst != NULL ) // Sprawdzamy czy coś przeczytan
47          printf("przeczytalem: %d znakow\n \"%s\"\n",
48                  (int) strlen(tekst), tekst);
49      free(tekst); // Zwalniamy przydzieloną pamięć
50      return 0;
51  }
```

14.11. Wskaźniki do funkcji

Metoda połowienia

1. Zadanie jest proste. Mamy funkcję $f(x)$ ciągłą i taką, że na końcach pewnego przedziału $[A, B]$ $f(A)f(B) < 0$. Zatem, funkcja ta zmienia znak w przedziale $[A, B]$ (co najmniej raz) ma zatem (co najmniej jedno) miejsce zerowe w tym przedziale.
2. Przedział $[A, B]$ dzielimy na pół (wyznaczając odpowiednio punkt C).
3. Odrzucamy ten z przedziałów $[A, C]$, $[C, B]$ w którym funkcja nie zmienia znaku (to znaczy ma ten sam znak na końcach przedziału).
4. Postępowanie prowadzimy tak długo, aż długość przedziału $[A, B]$ będzie mniejsza od zadanej liczby ε.

Uwaga: Obliczenia najprościej wykonać dla funkcji sin wybierając $0 < A < 3$ i $3{,}5 < B < 6$.

Powyższe zadanie można również zaprogramować korzystając z rekurencji!

Realizacja

```
1  double polowienie(double A, double B)
2  {
3     double C;
4  pocz:
5     C = ( A + B ) / 2.;
6     if ( f(A) * f(C) < 0 )
7        B = C;
8     else if ( f(A) * f(C) > 0 )
9        A = C;
10    else
11       return C;
12    if ( fabs(A - B) > 0.001 )
13       goto pocz;
14    return C;
15 }
```

Tu jest użyta instrukcja **goto** ale można inaczej.

Aby pozbyć się polecenia **goto** wyrzucamy etykietę w linii 4 i wstawiamy tam **do** { oraz linie 12 i 13 zastępujemy zamknięciem polecenie **do**: **while** (fabs(A − B) > 0.00000001);, otrzymując:

Inna realizacja

```
1  double polowienie(double A, double B)
2  {
3     double C;
```

```
4     do
5     {
6        C = ( A + B ) / 2.;
7        if ( f(A) * f(C) < 0 )
8           B = C;
9        else if ( f(A) * f(C) > 0 )
10          A = C;
11       else
12          return C;
13    }
14    while ( fabs(A - B) > 0.00000001 );
15    return C;
16 }
```

Z pętlą **do**.

„Zapętlenie” można zrealizować również wywołując ponownie funkcję polowienie i realizując rekurencję. Ponownie wyrzucamy linie 4 i 5 oraz zastępując linie 14 (**while**) i 15 ponownie instrukcją **if**:

```
1     if ( fabs(A - B) >  0.00000001 )
2        return polowienie(A, B);
3     else
4        return C;
```

oytrzymując:

Jeszcze inna realizacja

Zastąpimy pętlę **do** rekurencją.

```
1  double polowienie(double A, double B)
2  {
3     double C;
4        C = ( A + B ) / 2.;
5        if ( f(A) * f(C) < 0 )
6           B = C;
7        else if ( f(A) * f(C) > 0 )
8           A = C;
9        else
10          return C;
11    if ( fabs(A - B) >  0.00000001 )
12       return polowienie(A, B);
13    else
14       return C;
15 }
```

wer. 16 z drobnymi modyfikacjami!

Jak z tego skorzystać?

Aby z tego programu skorzystać, trzeba jego kod dołączyć do naszego oraz napisać funkcję pomocniczą f

```
1 double f(double x)
2 {
3         return sin(x);
4 }
```

oraz wywołać:

```
1 u = polowienie(2., 4.);
```

Niestety, w kodzie programu na stałę jest zapisana nazwa funkcji której miejsca zerowego szukamy. Nie jest to zbyt wygodne. Najwygodniej byłoby uczynić z funkcji jeszcze jedną zmienną. Do tego mogą przydać się wskaźniki.

Wskaźnik do funkcji

1. Deklarazja zwykłej zmiennej wygląda tak:

```
1 typ nazwa;
```

2. A wskaźnik deklaruje się tak:

```
1 typ * inna_nazwa;
```

3. Funkcję deklarujemy tak (mam na myśli prototyp):

```
1 typ_funkcji nazwa_funkcji( typ_argumentu );
```

4. A wskaźnik? Przez analogię:

```
1 typ_funkcji * nazwa_funkcji( typ_argumentu );
```

Połowienie jeszcze inaczej

```
1 double polowienieR( double A, double B, double f(double) )
2 {
3     double C;
4     C = ( A + B ) / 2.;
5     if ( f(A) * f(C) < 0 )
6         B = C;
7     else if ( f(A) * f(C) > 0 )
8         A = C;
```

```
9     else
10        return C;
11    if ( fabs(A - B) > 0.00000001 )
12        return polowienieR(A, B, f);
13    else
14        return C;
15 }
```

Połowienie jeszcze inaczej(cd)

```
1 double polowienieR( double A, double B, double f(double) )
```

Zwracam uwagę na trzeci argument w definicji funkcji polowienieR — jest to funkcja. Zatem z polowienieR można korzystać w sposób następujący:

```
1 double f(double x)
2 {
3     return sin(x);
4 }
5 ...
6     double ( *g )(double);
7     g = f;
8 ...
9     y = polowienieR(A, B, g);
```

albo

```
1     y = polowienieR(A, B, f);
```

albo

```
1     y = polowienieR(A, B, sin);
```

Dodatkowo po wykonaniu podstawienia g = f (albo g = sin) symbol (**zmienna**) g staje się „aliasem” (przezwiskiem) podstawionej funkcji.

.1. Jak powinien wyglądać program w C

.1.1. Wstęp

Pisząc program, głównym celem piszącego jest to, żeby program „działał”. Nawet korzystając z jakiegoś środowiska zintegrowanego dokonuje się bardzo wielu zmian, poprawek, kopiuje lub przenosi fragmenty kodu. Po kilku chwilach intensywnej pracy program wygląda bardzo niechlujnie.

Oto przykład programu, który miał realizować Algorytm E.

Przykład

```
1 #include <stdio.h>
2 int main()
3 {
4             int m=100 ;
5             int n=60 ;
6             int r;
7             {
8            while (r=m%n)
9            m=n;
10           n=r;
11          printf("nwd=%d\n",n);
12            }
13  getchar();
14  return 0;
15 }
```

Każda linijka kodu w innej odległości od lewego marginesu, jakieś nawiasy... Nie wiadomo o co chodzi!

Kod ten można zapisać inaczej. Zgodnie z wypracowanymi przez lata zasadami. Główną ideą tych zasad jest to, żeby każdy podrzędny blok kodu był bardziej „wcięty” od bloku nadrzędnego.

```
1 #include <stdio.h>
2 int main()
3 {
4     int m = 100;
5     int n = 60;
```

wer. 13 z drobnymi modyfikacjami!

```
6       int r;
7       {
8           while ( r = m % n )
9               m = n;
10          n = r;
11          printf("nwd=%d\n", n);
12      }
13      getchar();
14      return 0;
15  }
```

Widać wyraźnie, że linia kodu o numerze 9 jest **jedyną** linią podrzędną w stosunku do linii 8 (**while**. I od razu można się domyśleć, że kod jest niepoprawny. Widać też wyraźnie, że nawiasy (linie 7 i 12) są nadmiarowe i zupełnie niepotrzebne. Nie są błędem, ale są niepotrzebne...

Jak tworzyć taki kod?

.1.2. Narzędzia

W sumie wystarczy uważać podczas pisania programu. I „wcinać" bloki zgodnie z ideą algorytmu. Każdy programista rozumie algorytm, który programuje, potrafi wskazać co jest instrukcją nadrzędną, a które instrukcje są podrzędne.

— IDE

1. Geany
   Najprostsze środowisko nie posiada żadnych specjalnych narzędzi aby przeformatować źle wpisany kod. Natomiast jeżeli pisać kod poprawnie i z namysłem — będzie on formatowany poprawnie. Aby sformatować gotowy kod, można użyć programu zewnętrznego.
2. Anjuta.
   Zaznaczamy fragment kodu, który ma być przeformatowany i naciskamy klawisz Ctrl-I, albo wybieramy z menu Edycja → Automatyczne wcięcia.
3. Code::Blocks.
   Zaznaczamy fragment kodu, który ma być przeformatowany i z menu wybieramy Plugins → Source Code Formatter (AStyle).
4. Programy samodzielne

Jeżeli chodzi o programy „samodzielne” to najprostsza sytuacja jest w środowisku linuksowym. Wszystkie te programy są łatwo dostępne do zainstalowania jako „pakiety”. Instalujemy i mamy.

a) bcpp (jest źródlo, które trzeba skompilować w środowisku Windows),
Chyba najprostszy program. Pakiet nazywa się bcpp, ale zazwyczaj[4] główny program nazywa się indent. Program musimy uruchomić w konsoli, jako parametr podając nazwę pliku źródłowego (poprzedzoną całą ścieżką dostępu, gdy nie jesteśmy w kartotece, w której jest plik źródłowy). Czyli jakoś tak:
indent main.c
Program ma wiele dodatkowych parametrów, które zmieniają wygląd sformatowanego kodu, ale układ standardowy jest wystarczający.

b) Artistic Style (astyle) (jest również wersja windows/mac).
Równie prosty jak indent program.
astyle main.c
oferuje sensowny, choć (w moim przekonaniu) gorszy niż indent, kod wynikowy.

c) uncrustify (jest również wersja win32),
Osobiście uważam, za najlepszy program do formatowania, ale...
Chyba najmniej wygodny. Nie zlecam początkującym użytkownikom.

.1.3. Jak pisać program?

W pewnym sensie to zupełnie nieistotne. Z drugiej strony łatwiej czyta się programy poprawnie sformatowane, ale niepoprawny program, ale logicznie sformatowany, będzie wprowadzał czytelnika w błąd.

Sprawdzający prace studenckie są w trudnej sytuacji — czytanie czegoś takiego:

```
1 /*
2  * Maszyna.c
3  *
4  * Copyright 2013 stanislaw dac <206539@piwo002>
5  *
6  * This program is free software; you can redistribute it and/or modify
7  * it under the terms of the GNU General Public License as published by
8  * the Free Software Foundation; either version 2 of the License, or
9  * (at your option) any later version.
```

[4] Choć nie zawsze!

```
10  *
11  * This program is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program; if not, write to the Free Software
18  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
19  * MA 02110-1301, USA.
20  *
21  *
22  */
23 #include <stdio.h>
24 #include <string.h>
25 int sprawdzenie(char t){
26     if ( t == ' ' )
27         return 1;          //printf("spacja\n");
28     else if ( t == '+' )
29         return 2;           //printf("plus\n");
30     else if ( t == '-' )
31         return 3;           //printf("minus\n");
32     else if ( '0' <= t && t <= '9' )
33         return 4;           //printf("liczba: %c\n", t);
34 }
35
36 int maszyna(){
37     char t[] = "   +123 ";
38     int w;
39     int d = strlen(t);
40     printf("dlugosc ciagu: %d\n", d);
41     int i = 0;
42     while ( i < d )
43     {
44         if ( sprawdzenie(t[i]) == 1 )
45             printf("spacja\n");
46         else if ( sprawdzenie(t[i]) == 2 )
47             printf("plus\n");
48         else if ( sprawdzenie(t[i]) == 3 )
49             printf("minus\n");
50         else if ( sprawdzenie(t[i]) == 4 )
51         {
52             printf("liczba: %c\n", t[i]);
53             w = 4;
54         }
55         i++;
56     }
57     if ( w == 4 )
58         printf("\nW danym ciagu jest liczba!\n");
59     else
60         printf("\nW danym ciagu nie ma liczby!\n");
61 }
62
63 int main(int argc, char **argv)
64 {
65     printf("--Analiza ciagu--\n");
66     maszyna();
67     return 0;
68 }
69
70 /*
71    strlen(t)
72
73    '0'<=t[i] && t[i] <= '9'
74
75
76    ' ' == t[i] analogiczne sprawdzanie dla + i -
77  *  t[i]==\0
78  */
```

jest, po prostu niewygodne!

Kod po przeformatowaniu wyglądać może tak (astyle po lewej, indent po prawej):

```
#include <stdio.h>

int main(int argc, char **argv)
{    int u,v,t,k;
     u=300;
     v=100;
     k=0;
     while ((u%2==0) && (v%2==0)) {
          k=k+1;
          u=u/2;
          v=v/2;
     }
     if (u%2!=0)
     {    t=-v;
          goto etykieta3;
     }
     else
     {    t=u;
          goto etykieta2;
     }
etykieta2:
     t=t/2;
etykieta3:
     if (t%2==0) {
          goto etykieta2;
     }
     else {
          goto etykieta4;
     }
etykieta4:
     if (t>0) {
          u=t;
     }
     else {
          v=-t;
     }
     t=u-v;
     if (t!=0) {
          goto etykieta2;
     }
     else {
          printf ("%d",u*(1<<k));
     }
     return 0;
}
```

```
#include <stdio.h>

int
main (int argc, char **argv)
{
  int u, v, t, k;
  u = 300;
  v = 100;
  k = 0;
  while ((u % 2 == 0) && (v % 2 == 0))
    {
      k = k + 1;
      u = u / 2;
      v = v / 2;
    }
  if (u % 2 != 0)
    {
      t = -v;
      goto etykieta3;
    }
  else
    {
      t = u;
      goto etykieta2;
    }
etykieta2:
  t = t / 2;
etykieta3:
  if (t % 2 == 0)
    {
      goto etykieta2;
    }
  else
    {
      goto etykieta4;
    }
etykieta4:
  if (t > 0)
    {
      u = t;
    }
  else
    {
      v = -t;
    }
  t = u - v;
  if (t != 0)
    {
      goto etykieta2;
    }
  else
    {
      printf ("%d", u * (1 << k));
    }
  return 0;
}
```

wer. 13 z drobnymi modyfikacjami!

Przeformatowanie odpowiednio skonfigurowanym programem uncrustify może dać taki efekt:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int u, v, t, k;
    u = 300;
    v = 100;
    k = 0;
    while ( ( u % 2 == 0 ) && ( v % 2 == 0 ) )
    {
        k = k + 1;
        u = u / 2;
        v = v / 2;
    }
    if ( u % 2 != 0 )
    {
        t = -v;
        goto etykieta3;
    }
    else
    {
        t = u;
        goto etykieta2;
    }
etykieta2:
    t = t / 2;
etykieta3:
    if ( t % 2 == 0 )
        goto etykieta2;
    else
        goto etykieta4;
etykieta4:
    if ( t > 0 )
        u = t;
    else
        v = -t;
    t = u - v;
    if ( t != 0 )
        goto etykieta2;
    else
        printf ( "%d", u * ( 1 << k ) );
    return 0;
}
```

1. Najprościej skorzystać ze środowiska zintegrowanego.
2. Zazwyczaj pozwalają one na uzyskanie przyzwoitego wyglądu programu z właściwymi „wcięciami” pod warunkiem, że program wpisuje się własnoręcznie.
3. Środowiska zintegrowane posiadają czasem funkcję pozwalającą wskazany blok kodu przeformatować (w Anjuta, na przykład, zaznaczamy fragment kodu i naciskami klawisz **Ctrl-I**; w Code::Blocks zaznaczamy fragment kodu i wybieramy z menu Plugins → Source Code Formatter (AStyle)).
4. Gdy mamy gotowy (i nieładny) plik źródłowy można go szybko naprawić korzystając z programu indent (wchodzi w skład pakietu **bcpp**).

Oczywiście, czasami programiści starają się, aby ich programy wygadały śmiesznie. Poniżej przykład takiego programu zaczerpnięty z *21st International Obfuscated C Code Contest (2012)* http://www.ioccc.org/years.html#2012.

```c
                              #define   \
                           D(s)"<<"#s">>"
                         #define  q(s)p(#s)
                        #define S "␣endobj␣"
                       #define Y "endstream"S
                       #include     <stdio.h>
                       #define  o(s) b[s]=_;\
                        p("%u␣␣␣␣0␣␣obj",s);
#define E for            (c=d;c  < 123;c++)
  #define DANCE            "trailer␣␣␣"D\
     (/Root 3 0              R /Size %d)              "\nstartxref␣%u␣%%%EOF\n*/"
       #define                p(s, ...)               _+=printf(s,  ##__VA_ARGS__)
        #define C            "<</Type/Page␣␣␣␣␣␣␣␣␣␣/Parent␣%d␣%d␣R␣/Resources␣<<\
␣␣␣␣␣␣␣␣␣/ProcSet[␣␣␣␣␣␣␣␣/PDF/Text]/Font␣␣␣␣␣␣␣␣<</U"D(/Subtype/Type1/BaseFont\
            /Courier) "␣␣/T<</Subtype/Type3␣␣␣/FontBBox[0␣0␣10␣10]/FontMatrix[\
␣␣␣␣␣␣␣␣␣␣␣␣%f␣0␣0␣%f␣0␣0]/FirstChar␣%d/LastChar␣%d/Encoding<</Differences[%d"
               typedef int N;typedef char*Nyan;typedef char A;N a,b[64],d=65,_,v
                 [32]={84,0,64,282,90,74,330,85,93,173,167,176,80,208,81,13,7,87
                   ,160,346,32,128,170,218,16,26}; Nyan w[]={"+*-(,&-&","+*,&-&"
                     ,"+*.&/&","+*/*/+","+*())'('"   ,"+**&)&","+*(&'&","+*'*'+",
                       "","+,./","+,-,./","+--,+*"   ,"","+,(/","+,),(/","+-),+*"
                        ,"10␣0␣d0␣","8␣7␣2␣2␣re␣"    ,"+*+.'`'@'mi␣+/+/(mi"};Nyan
                          nyan(Nyan _,N y,A n){
                           N g=v[~-y%32];Nyan
                            s=w[g>>n&3|n*2];
                            for(a=0;*_=*s++;
                            a++,_++,*_++=32)
                            {*_+=*_-32?10:0;
                            if(a%2&&*_/16==3
                            ){if(g>>8)*_=105-*_;
                           _++;*_++=32;*_=~-a&&a-13
                           ?108:109;}}return n?n-9?nyan
                          (_,y,n-2):_:nyan(_+=~y&'␣'?sprintf
                         (_,17[w]):0,y,9);}N main(N c){A e[256];
                        p("/*%%PDF-1.3%%*/")-2;q(#include<stdio.h>\n);
                       q(#define o *_++&& *_-41\n#define);p("␣endstream␣");q
                      (main(){for(;*_++;       *_-40?:putchar(o?*_:o?10:41));\n)
                     ;q(#define  endobj            return 0;}\n);q(typedef int ET;/)
                    ;q(*);o(1)*b=~(p(D                  (/Length 2 0 R)"stream\n"))
                   ;for(p("BT␣12␣818"                     "␣Td/%c␣12␣Tf␣12␣TL%%%c/"
                  "static␣char*_=\""                        "␣\\\n",7[v],*v/2);c=
                 getchar(),~c;c-10?                          p("/%c␣12␣Tf(\\%o)"
                "Tj",v[~-c%'␣'<25[                           v]&&!~-(~-c/'@')?0
               :7],c):(p("()'")))                           ;p("%%\";\nET␣");*
              b-=~_;p(Y"/*");E{a                          =nyan(e+sprintf(e,
             16[w]),c,6)-e;o(c-                          59)p(D(/Length %d)
            "stream\n%s"Y,a,e)                          ;}o(2)p("␣%u"S,*b);
           o(3)p(D(/Pages 4 0                          R)S);o(4)p(D(/Count
          1/MediaBox[0 0 595                          842]/Kids[5 0 R])S)
         ;o(5)p(C,4,0,6e-2,                          6e-2f,d,122,d);E p(
        "/%c",~-c/6+~14?c:                          d);q(]>>/CharProcs<<)
       ;E if(~-c/6+~14)p(                           "/%c␣%d␣0␣R",c,c+~58);
      for(q(>>/Widths[),                            c=59;--c;p("␣10"));a=p
     ("]>>>>>>/Contents␣1␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣0␣R>>"S);for(p("xref\
␣0␣%d␣",--d);c<d;p("%010u\
␣%05d␣n␣␣",*(c+++b),NULL));
  return!(p(DANCE,d,a));}
```

Program hamano (http://www.ioccc.org/2012/hamano/hint.html służy do zamiany napisu w plik PDF z tym napisem zakodowany w dziwnym „alfabecie ludzikowym” (inspirowanym opowiadaniem Arthura Conan Doyla „Tańczące sylwetki”).

A. Dokumentacja pod Linuksem

A.1. Wstęp

Ponieważ język programowania C jest „częścią" systemu Linux, jego dokumentacja jest również częścią systemu dokumentacji linuksa. Dokumentacja linuksa obsługiwana jest przez system man (od manual).

A.2. Dokumentacja HTML

Najprostszą formą dokumentacji jest uruchomienie przeglądarki www i wpisanie w pasku adresu: `file:///usr/share/doc/c-cpp-reference/index.html`. Później wybieramy język programowania (C) i jedną z pozycji:
— Master Index
— Keywords
— Functions

A.3. Dokumentacja man

Sprawa jest prosta, choć mało klikalna.

1. Otwieramy terminal (najprościej nacisnąć równocześnie klawisze Ctrl-Alt-T).
   — w terminalu wpisujemy man <*temat*> (gdzie <*temat*> to to co nas interesuje), na przykład man strlen (żeby dowiedzieć się czegoś na temat funkcji strlen) albo man string żeby dowiedzieć się czegoś na temat funkcji operujących na napisach w języku C.
   — Alternatywnie możemy napisać gman co otworzy prymitywną przeglądarkę wszystkich stron manuala i w polu wyszukiwania można wpisać coś co nas interesuje.

— Gdy nie znajdujemy nic ciekawego można użyć polecenia apropos <*temat*>. Wyświetli ono wszystkie strony jakoś związane z tematem. Wynik działania polecenia apropos string wygląda jakoś tak:

```
...
strcat (3)            - concatenate two strings
strchr (3)            - locate character in string
strchrnul (3)         - locate character in string
strcmp (3)            - compare two strings
strcoll (3)           - compare two strings using the curren
...
```

I po lewej mamy nazwę funkcji, później w nawiasach jest numer rozdziału podręcznika (3 to właśnie funkcje) i na koniec, krótki opis.

2. Do graficznej przeglądarki tematów można też dobrać się naciskając klawisze Alt-F2 i wpisując w polu wyszukiwania gman. (Kolejne naciśnięcia klawiszy Alt-F2 będą wyświetlały ostatnio wyszukiwane w ten sposób programy.

B. Ćwiczenia

Poniższe ćwiczenia mają za główne zadanie zatrzymać się nad bardzo prostym (na ogół) kodem i dokładnie zastanowić się co robią poszczególne instrukcje, oraz próbować odgadnąć co robi cały program (lub całą funkcja).

Ćwiczenie 1

```
1 int p1(char str[])
2 {
3         int i = 0;
4         while (str[i] != '\0')
5         {
6                 i++;
7         }
8         return i;
9 }
```

Ćwiczenie 2

```
1 int p2(char* str)
2 {
3         int i = 0;
4         while (*str != '\0')
5         {
6                 i++;
7                 str++;
8         }
9         return i;
10 }
```

Ćwiczenie 3

```
1  void p3(char dest[], char src[])
2  {
3          int i = 0;
4          while (src[i] != '\0')
5          {
6                  dest[i] = src[i];
7                  i++;
8          }
9          dest[i] = '\0'
10 }
```

Ćwiczenie 4

```
1  void p4(char dest[], char src[])
2  {
3          int i = 0;
4          while (( dest[i] = src[i] ) != '\0')
5          {
6                  i++;
7          }
8  }
```

Ćwiczenie 5

```
1  void p5(char* dest, char* src)
2  {
3      while (( *dest = *src ) != '\0')
4      {
5          dest++;
6          src++;
7      }
8  }
```

Ćwiczenie 6

```
1  void p6(char* dest, char* src)
2  {
```

```
3     while (( *dest++ = *src++ ) != '\0');
4 }
```

Ćwiczenie 7

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int findA(char[]);
4 int main(int argc, char* argv[]){
5     char oldString[] = {'H', 'e', 'l', 'l', 'o', '\n'};
6     printf("Calling␣findNull,␣will␣seg␣fault␣now\n");
7     findA(oldString);
8     printf("Program␣complete\n");
9     return EXIT_SUCCESS;
10 }
11 int findA(char newArray[]){
12     int i = 0;
13     while(newArray[i] != 'A' || newArray[i + 1] != 'A') {
14         i++;
15     }
16     printf("Found␣at:␣%d\n", i);
17     return i;
18 }
```

W tym ćwiczeniu szczególnie należy zwrócić uwagę na linię 13. Jaki jest jej sens. Co ona oznacza

Ćwiczenie 8

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #define SIZE 1000000000
4 #define TRUE 1
5 int main() {
6     int j;
7     int *data;
8     int numMallocs = 0;
9     while (TRUE) {
```

```
10          data = (int*)malloc(SIZE * sizeof( int ));
11          if (data == NULL) {
12              printf("memory allocation error: data\n");
13              return EXIT_FAILURE;
14          }
15          printf("%d\n", ++numMallocs);
16          for (j = 0; j < SIZE; j++) {
17              data[j] = j;
18          }
19      }
20      free(data);
21      return EXIT_SUCCESS;
22  }
```

Ten program jest źle zaprogramowany. Na czym to „złe zaprogramowanie" polega?

Ćwiczenie 9

```
1  int p9(char *s, char *t)
2  {
3    int Result = 0;
4    int s_length = 0;
5    int t_length = 0;
6    s_length = strlen(s);
7    t_length = strlen(t);
8    if(t_length <= s_length)
9    {
10     s += s_length - t_length;
11     if(0 == strcmp(s, t))
12     {
13       Result = 1;
14     }
15   }
16   return Result;
17 }
```

Ćwiczenie 10

wer. 16 z drobnymi modyfikacjami!

```
double p10(int n, double x, double a[])
{
    double r = 0;
    int i;
    for (i = n; i >= 0; i--)
        r = r * x + a[i];
    return r;
}
```

Zaproponować sposób użycia.

Ćwiczenie 11

```
#include <stdio.h>
main()
{
    char *text_pointer = "Good morning!";
    for(; *text_pointer != '\0'; ++text_pointer)
        printf("%c", *text_pointer);
}
```

Ćwiczenie 12

```
#include <stdio.h>
main()
{
    static char *days[] = {
        "Sunday", "Monday", "Tuesday", "Wednesday", \
        "Thursday", "Friday", "Saturday"
    };
    int i;
    for ( i = 0; i < 6; ++i )
        printf( "%s\n", days[i]);
}
```

O zmiennej typu **static** można poczytać w rozdziale 7.1.2.

Dziwne algorytmy sortowania

Poniżej przedstawiam dwa, bardzo proste algorytmy. Należy zaprogramować je jako funkcje i napisać program z nich korzystający. Oba algorytmy realizują sortowanie tablic liczb.

Zadanie 1

— Dana jest liczba całkowita, dodatnia N.
— Dana jest tablica (dowolnego typu) K o N elementach.

Należy zaprogramować poniższy algorytm:

1. Wykonaj kroki 2 i 3 dla $j = N, N-1, \ldots, 2$.
2. Wśród liczb $K[j], K[j-1], \ldots, K[1]$ znajdź największa. Niech to będzie $K[i]$, **gdzie i jest możliwie największe**.
3. Zamień $K[i]$ z $K[j]$.

Dla wybranego N należy wypełnić tablicę K losowymi danymi, przetestować algorytm.

Zadanie 2

— Dana jest liczba całkowita, dodatnia N.
— Dana jest tablica (dowolnego typu) K o N elementach.
— Potrzebna będzie tablica robocza C (takiego samego typu jak K) o N elementach.

Należy zaprogramować poniższy algorytm:

1. Wyzeruj $C[1]$ do $C[N]$.
2. Wykonaj krok 3 dla $i = N, N-1, \ldots, 2$.
3. Wykonaj krok 4 dla $j = i-1, i-2, \ldots, 1$.
4. Jeżeli $K[i] < K[j]$ to zwiększ $C[j]$ o jeden; w przeciwnym przypadku zwiększ $C[i]$ o jeden.

Dla wybranego N należy wypełnić tablicę K losowymi danymi, przetestować algorytm.

Do czego służy tablica C? Jakie znaczenie mają poszczególne jej elementy? Jak można je wykorzystać?

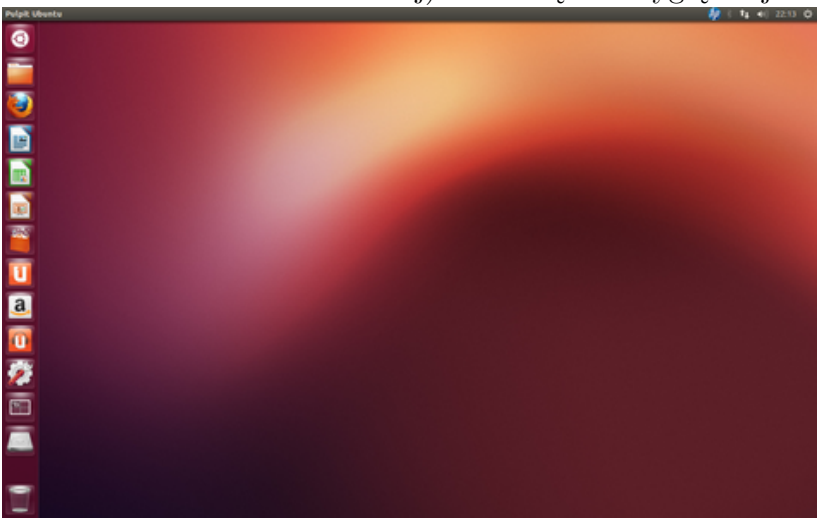
C. Laboratoria

C.1. Środowisko pracy

Na komputerach w laboratorium zainstalowany jest system Ubuntu[1]. Różni się ono nieco od środowiska Windows. W związku z tym pozwoliłem sobie poświęcić mu kilka zdań.

C.1.1. Logowanie do systemu

Aby zalogować się, należy na stronie wybrać „Zaloguj”, następnie wpisać swój numer indeksu i, następnie, hasło. Po zalogowaniu się (pierwsze logowanie może trwać nieco dłużej) ekran będzie wyglądał jakoś tak:



Jest to widok „pulpitu”. Po lewej stronie znajduje się „pasek uruchamiania” (*launcher*). Na pasku umieszczone są ikony standardowych aplikacji. Kliknięcie w ikonę uruchamia aplikację. Uwaga: Klikamy tylko taz! Podwójne kliknięcie może powodować poważne kłopoty!

[1] W chwlili pisania tych słów jest to 12.04.5 LTS.

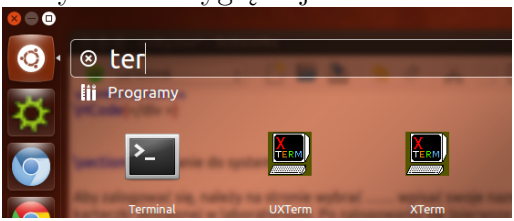
W razie kłopotów można się zalogować jako „Gość”. Zwracam uwagę, że konto Gość jest kontem **wirtualnym**, po wyłączeniu komputera wszystkie pliki przepadają! Poza tym jest najzupełniej funkcjonalne, pliki można nagrywać na nośnikach zewnętrznych (podłączonych przez USB).

C.1.2. Uruchamianie aplikacji

Zwracam uwagę na ikonę kliknięcie w nią pozwala wybierać aplikacje do uruchomienia. Trochę podobne działanie ma równoczesne naciśnięcie klawiszy „windows” i klawisza „A”. Jednak w tym drugim przypadku dostajemy dostęp do znacznie szerszego zestawu zainstalowanych aplikacji. Tak, czy inaczej — wyszukiwanie aplikacji po ikonce jest dosyć niewygodne. Najlepiej zacząć pisać jej nazwę...

Załóżmy, że chcemy uruchomić aplikację terminal (zaraz będzie nam potrzebna). Po wpisaniu kilku liter mamy kilka aplikacji o nazwie z nimi zgodnej. Możemy wybrać jedną z nich za pomocą myszy lub pierwszą naciskając terminal. Dostęp do funkcji związanych z tą ikoną można uzyskać również po naciśnięciu klawisza „windows”[2]. Zatem aby uruchomić terminal wystarczy nacisnąć klawisz napisać „ter” i nacisnąć klawisz Enter. Albo jeszcze prościej naciskając równocześnie trzy klawisze Ctrl–Alt–T...

Wszystko to wygląda jakoś tak:



C.1.3. Zmiana hasła

Aby zmienić hasło należy wykonać następujące czynności:

1. Otworzyć terminal (jak to zrobić opisano powyżej).
2. W terminalu wydać polecenie `yppasswd` i odpowiedzieć na dwa pytania:

```
Changing NIS account information for xxxxx on xeon4.immt.pwr.wr
Please enter old password:
Changing NIS password for xxxxx on xeon4.immt.pwr.wroc.pl.
```

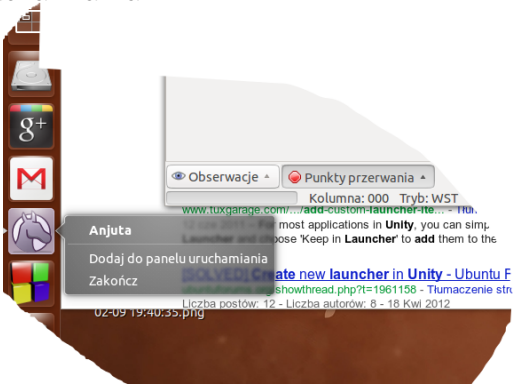
[2] W terminologii linuksowej nazywa się on *Super*.

```
Please enter new password:
Please retype new password:
```

czyli najpierw podać stare hasło, a następnie **dwa razy** podać hasło nowe.
Uwaga: Wpisywanych informacji nie będzie widać!

C.1.4. Dodawanie/usuwanie aplikacji do/z paska uruchamiania

Aplikacje, z których korzystamy najczęściej można dodać do paska uruchamiania. Pierwszy raz aplikację uruchamiamy standardowo (patrz rozdział C.1.2). Ikona aplikacji pojawi się na pasku uruchamiania. Klikamy w nią prawym klawiszem myszy i z menu wybieramy „Dodaj do panelu uruchamiania”:



W podobny sposób możemy aplikację usunąć z paska (pozostanie w systemie i będzie można ją uruchamiać):

Z listy wybieramy „Usuń z panelu uruchamiania”.

D. Jak zmierzyć czas obliczeń?

Wyobraźmy sobie, że chcielibyśmy zmierzyc czas wykonania jakiegoś fragmetu programu. Na laboratorium, ćwiczeniach lub tylko na listach zadań poznaliśmy funkcję time():

```
1 #include <time.h>
2 ...
3 time_t czas;
4 czas = time( NULL )
```

Funkcja time zwraca (gdy nie było błędu) aktualny czas jako liczbę sekund od początku „Epoki”[1]. Kiedyś (i być może w dalszym ciągu na niektórych starych systemach) czas był przechowywany w zmiennej 32-bitowej (co powodowało, że ten system liczenia czasu przestanie działać w 2038 roku). Dziś przechowywany jest w zmiennej typu time_t, który jest zdefinowany w sposób pozwalający efektywnie przechowywać czas (może to być zmienna typu **long int**).

Funkcja ta może być więc wykorzystana do pomiaru odcinków czasu w następujący sposób:

```
1 time_t Start, Stop;
2 ...
3
4 Start = time(NULL);
5 // Tu coś robimy, co zajmuje trochę czasu
6 Stop = time(NULL);
```

Wartość różnicy stop − start zawiera czas obliczeń. Niestety, rozdzielczość pomiaru czasu wynosi tylko jedną sekundę. Zatem nie zawsze można z tego skorzystać. Na przykład

[1] Za początek Epoki uważa się godzinę 00:00 (według czasu UTC) 1 stycznia 1970 roku.

```
    Start = time(NULL);
    for(i=0; i < N; i++)
    sin(0.333333);
    Stop = time(NULL);
```

Dla N równego 10000000 (dziesięć milionów) czas obliczeń jest równy 0. Dla N równego 100 milionów — 1 sekunda.

Zatem potrzebny jest dokładniejszy zegar...

Funkcja gettimeofday podaje czas od początku Epoki w mikrosekundach. Jej definicja jest następująca:

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Pierwszy argument funkcji (wskaźnik do struktury timeval)

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

Zwraca czas (od początku Epoki) jako pełną liczbę sekund (składowa tv_sec) i liczbę mikrosekund w rozpoczętej sekundzie (składowa tv_usec). I teraz

```
struct timeval start, stop;
unsigned long long Start, Stop;
// ......
gettimeofday(&start, NULL);
Start = ((unsigned long long) start.tv_sec * 1000000)
        + start.tv_usec;
for (i = 0; i < N  ; i++)
{
    sin(0.33333);
}
gettimeofday(&stop, NULL);
Stop = ((unsigned long long) stop.tv_sec * 1000000)
       + stop.tv_usec;
printf("Czas: %ld\n", Stop - Start);
```

Pozwala wyznaczyć czas operacji nawet dla N = 100000 (na moim laptopie czas ten wynosi około 500 mikrosekund). Ciągle oblczenie czasu pojedynczej operacji wyliczenia nie jest możliwe, ale...

D.1. Minus jeden do potęgi n

W jednym z zadań na kolokwium pojawił się problem wyliczenia następującego wyrażenia:

$$(-1)^n$$

Większość studentów zaproponowała, żeby wyliczyć wartość tego wyrażenia używając funkcji pow: pow(−1, n). Sprawdźmy zatem ile trwa wykonanie funkcji pow:

```
1  gettimeofday(&start, NULL);
2  Start = ((unsigned long long) start.tv_sec * 1000000)
3          + start.tv_usec;
4  k = -1
5  for (i = 0; i < N  ; i++)
6  {
7      pow(k, i);
8  }
9  gettimeofday(&stop, NULL);
10 Stop = ((unsigned long long) stop.tv_sec * 1000000)
11         + stop.tv_usec;
12 printf("Czas: %ld\n", Stop - Start);
```

Czas obliczeń dla N = 1000000 (milion) wynosi około 120–130 tysięcy mikrosekund. Obliczenia można jednak uprościć. Zamiast używać pow(k, i) można użyć k=k∗(−1) i wówczas czas obliczeń spada do około 5500 mikrosekund (czyli przyśpieszenie około 22 razy)!

Tak na marginesie zamina k=k∗(−1) na k = −k nie przynosi już żadnego istotnego przyśpiesznia.

Możemy poprosić kompilator, żeby w trakcie kompilacji dokonał opymalizacji kodu. W tym przypadku czas spadnie do ciut mniej niż 100 tysięcy mikrosekund w pierwszym przypadku (pow) i 1200 mikrosekund w drugim, więc przyśpieszenie będzie około 80 razy!

E. Kody ASCII

Na ilustracji E.1 przedstawiono kody ASCII wszystkich znaków, które mogą być zapisane w zmiennych typu **char**. Tablicę kodów można także obejrzeć po wpisaniu polecenia `man ascii` w terminalu.

ASCII CONTROL CODE CHART

b7 b6 b5 / BITS	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4 b3 b2 b1	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0	16 DLE 20	32 SP 40	48 0 60	64 @ 100	80 P 120	96 ‘ 140	112 p 160
0 0 0 1	1 SOH 1	17 DC1 21	33 ! 41	49 1 61	65 A 101	81 Q 121	97 a 141	113 q 161
0 0 1 0	2 STX 2	18 DC2 22	34 ” 42	50 2 62	66 B 102	82 R 122	98 b 142	114 r 162
0 0 1 1	3 ETX 3	19 DC3 23	35 # 43	51 3 63	67 C 103	83 S 123	99 c 143	115 s 163
0 1 0 0	4 EOT 4	20 DC4 24	36 $ 44	52 4 64	68 D 104	84 T 124	100 d 144	116 t 164
0 1 0 1	5 ENQ 5	21 NAK 25	37 % 45	53 5 65	69 E 105	85 U 125	101 e 145	117 u 165
0 1 1 0	6 ACK 6	22 SYN 26	38 & 46	54 6 66	70 F 106	86 V 126	102 f 146	118 v 166
0 1 1 1	7 BEL 7	23 ETB 27	39 ’ 47	55 7 67	71 G 107	87 W 127	103 g 147	119 w 167
1 0 0 0	8 BS 10	24 CAN 30	40 (50	56 8 70	72 H 110	88 X 130	104 h 150	120 x 170
1 0 0 1	9 HT 11	25 EM 31	41) 51	57 9 71	73 I 111	89 Y 131	105 i 151	121 y 171
1 0 1 0	10 LF 12	26 SUB 32	42 * 52	58 : 72	74 J 112	90 Z 132	106 j 152	122 z 172
1 0 1 1	11 VT 13	27 ESC 33	43 + 53	59 ; 73	75 K 113	91 [133	107 k 153	123 { 173
1 1 0 0	12 FF 14	28 FS 34	44 , 54	60 < 74	76 L 114	92 \ 134	108 l 154	124 | 174
1 1 0 1	13 CR 15	29 GS 35	45 – 55	61 = 75	77 M 115	93] 135	109 m 155	125 } 175
1 1 1 0	14 SO 16	30 RS 36	46 . 56	62 > 76	78 N 116	94 ^ 136	110 n 156	126 ~ 176
1 1 1 1	15 SI 17	31 US 37	47 / 57	63 ? 77	79 O 117	95 _ 137	111 o 157	127 DEL 177

LEGEND: dec / CHAR / hex oct

Victor Eijkhout
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37996, USA

Rysunek E.1. Kody ASCII

wer. 16 z drobnymi modyfikacjami!

Bibliografia

[1] Arkadiusz Antoniak. Programowanie portu szeregowego w systemach operacyjnych Linux i Windows. *Elektronika Praktyczna*, (3), 2006. Dziewięcioczęściowy artykuł, który ukazał się w numerach 3/2006 do 7/2007 czasopisma Elektronika raktyczna. Dostępny ze strony czasopisma pod adresem http://ep.com.pl/files/3151.pdf.

[2] Programowanie w języku C, 2007. Wersja elektroniczna dostępna pod adresem: http://pl.wikibooks.org/wiki/Programowanie:C.

[3] Edsger W. Dijkstra. Notes on structured programming. Raport instytutowy T.H.-Report 70-WSK-03, Technological University Eidhoven, 1970. http://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF.

[4] Commie Pinko Dirtbag. Now, here's a program for calculating the date of easter. - democratic underground. http://www.democraticunderground.com/discuss/duboard.php?az=view_all&address=105x6110716, Styczeń 2007.

[5] Gary Frerking, Peter Baumann. Serial programming HOWTO. http://www.tldp.org/HOWTO/Serial-Programming-HOWTO/, Sierpień 2001.

[6] David Goldberg. What every computer scientist should know about Floating-Point arithmetic. *Numerical Computation Guide*. Sun Microsystems, Palo Alto, 2000. http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[7] Greg Hankins, David S. Lawyer. Serial HOWTO. http://www.tldp.org/HOWTO/Serial-HOWTO.html, Luty 2011.

[8] Richard Heathfield. The C programming language answers to exercises. http://users.powernet.co.uk/eton/kandr2/, 2002.

[9] Ted Jensen. A tutorial on pointers and arrays in C, Sept. 2003.

[10] B. W. Kernighan, D. M. Ritchie. *Język ANSI C*. WNT, Warszawa, 2007.

[11] Ben Klemens. *21st Century C*. O'Reilly Media, 2012. http://shop.oreilly.com/product/0636920025108.do.

[12] Donald E. Knuth. *The Art of Computer Programming, Volumes 1-4A Boxed Set*. Addison-Wesley Professional, wydanie 1, Marzec 2011.

[13] Donald Ervin Knuth. *Sztuka programowania*. Klasyka Informatyki. Wydawnictwa Naukowo-Techniczne, Warszawa, 2002. Tomy od 1 do 4A.

[14] Russ Miles, Kim Hamilton. *UML 2.0 wprowadzenie*. Helion, Gliwice, 2007.

[15] Wojciech Myszka, redaktor. *Komputerowy System Obsługi Eksperymentu*. Wydawnictwa Naukowo-Techniczne, Warszawa, 1991.

[16] Ciaran O'Riordan. *Learning GNU C*. Ciaran O'Riordan, 2002. http://www.faqs.org/docs/learnc/.

[17] Richard Reese. *Wskaźniki w języku C: przewodnik*. Helion, Gliwice, 2014. Dostęp po zalogowaniu w bazie NASBI. http://biblioteka.pwr.wroc.pl/NASBI_Naukowa_Akademicka_Sieciowa_Biblioteka_Internetowa,161.dhtml.

[18] Richard M Reese. *Understanding and Using C Pointers*. O'Reilly Media, 2013.

[19] Piotr Stańczyk. *Algorytmika praktyczna: Nie tylko dla mistrzów*. Wydawnictwo Naukowe PWN, Warszawa, 2009.

[20] Steve Summit, V. S. Carpenter, Sunil Rao. comp.lang.c answers (abridged) to frequently asked questions (FAQ). http://www.faqs.org/faqs/C-faq/, 2008.

[21] Clovis L. Tondo, Scott E. Gimpel, Paweł Koronkiewicz. *Język ANSI C: programowanie, ćwiczenia*. Helion, Gliwice, wydanie wyd. 2, 2010.

[22] John Voelcker. NHTSA has no software engineers or EEs to analyze toyotas. http://www.thecarconnection.com/marty-blog/1042836_nhtsa-has-no-software-engineers-or-ees-to-analyze-toyotas, Luty 2010.

[23] Tomasz Wawrzyczek. Z pamiętnika "Młodego Technika" – marzec 1984 – czyli o tym kiedy wypada Wielkanoc. http://www.spidersweb.pl/2013/03/z-pamietnika-mlodego-technika-marzec-1984.html, Marzec 2013.

[24] Peter Wayner. 12 predictions for the future of programming. http://www.infoworld.com/d/application-development/12-predictions-the-future-of-programming-235292, Luty 2014.

Kolofon

Bryk powstał na podstawie slajdów będących materiałem pomocniczym do wykładu prowadzonego przez mnie dla kierunku Mechatronika i Automatyka i Robotyka na Wydziale Mechanicznym Politechniki Wrocławskiej.

Slajdy zostały przygotowane z użyciem pakietu beamer systemu LaTeX 2_ε. Bryk stworzono przetwarzając kod źródłowy slajdów z użyciem pakietu beamerarticle... System LaTeX pozwolił na łatwe wygenerowanie slajdów oraz pliku PDF zawierającego bryk złożony w formacie A4 oraz A5 (na urządzenia o mniejszym ekranie).

Na okładce wykorzystałem ilustrację „good-code" pochodzącą ze strony https://xkcd.com prowadzonej przez Randalla Munroe.