

# Termin Egzaminu (Język C):

>> **ŚRODA, 29 STYCZNIA 2025** <<  
**GODZ. 11.00**

[ **POPRAWKA: ŚRODA, 19 LUTEGO, 11.00** ]

**!!!**

**Forma zaliczenia kursu:** Egzamin pisemny – test wyboru \*) \*\*)

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

\*\*) **Ocena 5.0 (bdb)** z ćwiczeń *zwalnia z egzaminu*

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

# Poprzedni wykład [ *14. 01. 2025* ] :

- Uwagi o metodach numerycznych i proste przykłady
- Przekształcanie liczb zmiennopozycyjnych do całkowitych (i odwrotnie) jako składowych unii [= > *fast inv. sq. root*]
- Przykład algorytmu kompresji: ***kodowanie Huffmana***

## ***PRZYPOMNIENIE: Narzędzia WE/WY w bibliotece standardowej C***

Operacje czytanie/pisanie do strumienia `f` możemy zrealizować na kilka sposobów:

- 1) *WE/WY znakowe* realizują *funckje makra* `getc(f)` oraz `putc(c, f)` [które możemy zastąpić ***funkcjami***: `fgetc(f)`, `fputc(c, f)` — *jeśli chcemy mieć pewność, że argumenty zostaną obliczone przed przekazaniem sterowania*].
- 2) *WE/WY formatowane*: funkcje `fscanf(f, ...)` oraz `fprintf(f, ...)`; drugim argumentem jest zawsze napis, który może zawierać ***specyfikator(y) przekształcenia*** [patrz: [wyklad04](#)]. Liczba argumentów jest zmienna; zob. [wyklad07.pdf#page=36](#).

3) WE/WY wierszowe realizują funkcje `fgets(s, N, f)` oraz `fputs(s, f)` gdzie `s` jest wskaźnikiem znakowym zaś `n` — liczbą znaków do wczytania. [patrz: [wyklad8](#)]

4) WE/WY niskopoziomowe jest możliwe z użyciem funkcji `fread`, `fwrite` [zob. [wyklad8](#)], które realizują “hurtowy” transfer bloku bajtów ze strumienia do pamięci (i odwrotnie).

Ostatnia możliwość dotyczy zasadniczo plików binarnych (choć transfer odpowiednich *tablic znakowych* może zadziałać poprawnie dla plików tekstowych); pkt 1. (`getc`, `putc`) opisuje narzędzia uniwersalne, dla plików binarnych trzeba jedynie pamiętać o rzutowaniu znaku do `unsigned char`; narzędzia z pkt. 2 i 3. są z kolei dedykowane dla pl. tekstowych (*znak końca wiersza*: `'\n'`).

# Środowisko systemu UNIX

W systemach operacyjnych zgodnych ze standardem **POSIX** (*Portable Operating System Interface*), takich jak Unix, Linux, MacOSX (i kilku innych...) implementacje C (oraz C++) zaopatrzone są w nagłówek standardowy `<unistd.h>` zawierający deklaracje szeregu funkcji rezydujących wewnątrz systemu operacyjnego (inaczej: **odwołań systemowych**).

**Odwołania systemowe** często możemy zastąpić funkcjami z biblioteki standardowej (tak jest np. z omawianymi wcześniej funkcjami plikowymi, zob. [wyklad07](#)); przydają się jednak jeśli chcemy osiągnąć maksymalną wydajność programu, kontrolować buforowanie zapisu/odczytu danych, itp.

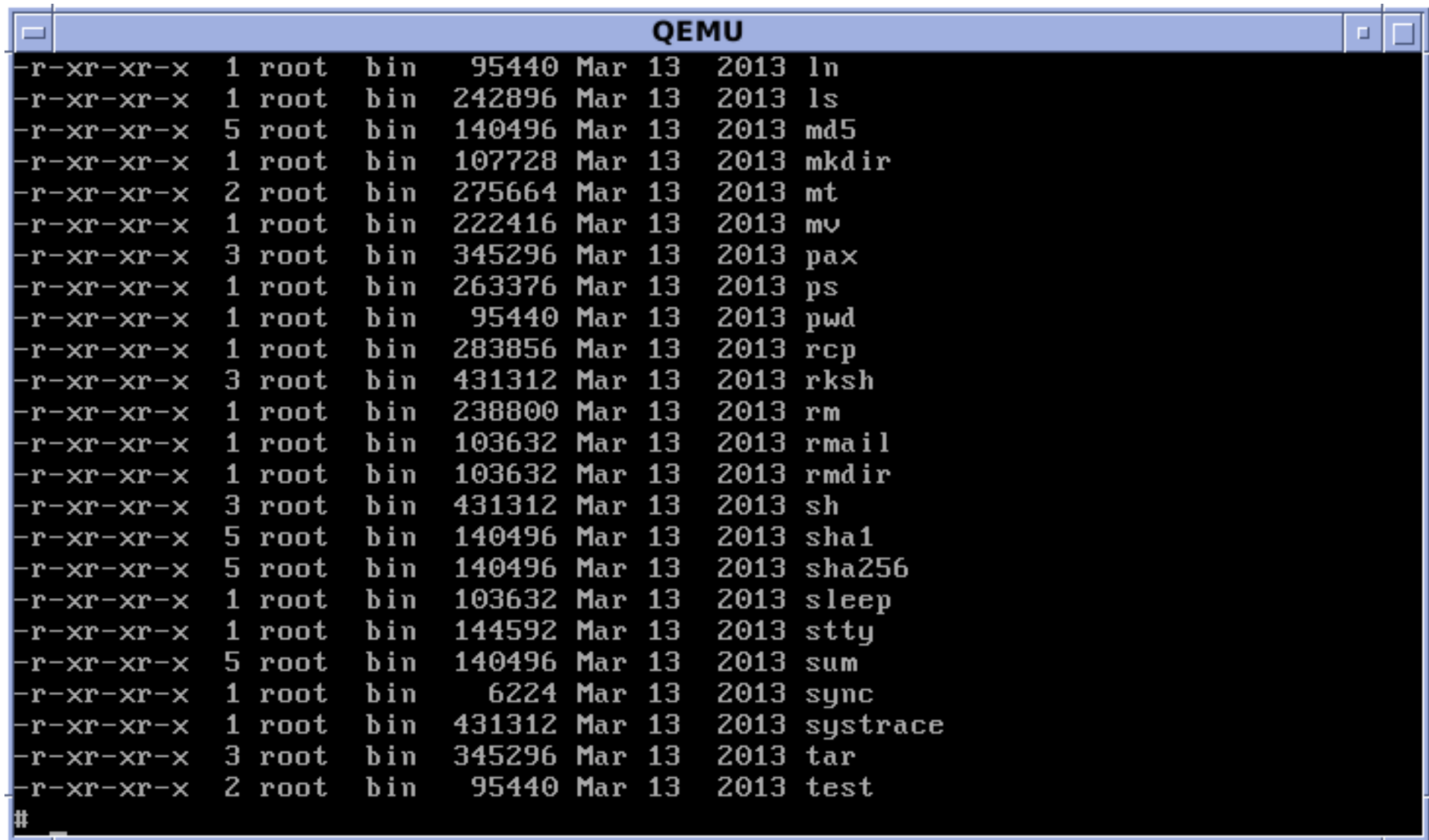
# Operacje wejścia/wyjścia

W systemach typu Unix, operacje wejścia/wyjścia wyrażają się poprzez **czytanie z plików** lub **pisanie do plików**. Każde urządzenie zewnętrzne (także *klawiatura i ekran!*) jest powiązane z pewnym plikiem wchodzącym w skład systemu plików.

*A zatem, całą komunikację programu z urządzeniami zewnętrznymi obsługuje pewien wspólny, jednorodny aparat.*

**Otwieranie plików.** W ogólnym przypadku, aby czytać z pliku lub do niego pisać, musimy najpierw poinformować system o naszym zamiarze ( => otworzyć plik ). Jeśli chcemy pisać do pliku, może być konieczne jego utworzenie lub skasowanie dotychczasowej zawartości; system musi zatem sprawdzić, czy mamy do tego prawo [ => *prawa dostępu w systemie Unix* ].

```
$ ls -l
```



```
QEMU
-r-xr-xr-x 1 root bin 95440 Mar 13 2013 ln
-r-xr-xr-x 1 root bin 242896 Mar 13 2013 ls
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 md5
-r-xr-xr-x 1 root bin 107728 Mar 13 2013 mkdir
-r-xr-xr-x 2 root bin 275664 Mar 13 2013 mt
-r-xr-xr-x 1 root bin 222416 Mar 13 2013 mv
-r-xr-xr-x 3 root bin 345296 Mar 13 2013 pax
-r-xr-xr-x 1 root bin 263376 Mar 13 2013 ps
-r-xr-xr-x 1 root bin 95440 Mar 13 2013 pwd
-r-xr-xr-x 1 root bin 283856 Mar 13 2013 rcp
-r-xr-xr-x 3 root bin 431312 Mar 13 2013 rksh
-r-xr-xr-x 1 root bin 238800 Mar 13 2013 rm
-r-xr-xr-x 1 root bin 103632 Mar 13 2013 rmail
-r-xr-xr-x 1 root bin 103632 Mar 13 2013 rmdir
-r-xr-xr-x 3 root bin 431312 Mar 13 2013 sh
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 sha1
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 sha256
-r-xr-xr-x 1 root bin 103632 Mar 13 2013 sleep
-r-xr-xr-x 1 root bin 144592 Mar 13 2013 stty
-r-xr-xr-x 5 root bin 140496 Mar 13 2013 sum
-r-xr-xr-x 1 root bin 6224 Mar 13 2013 sync
-r-xr-xr-x 1 root bin 431312 Mar 13 2013 systrace
-r-xr-xr-x 3 root bin 345296 Mar 13 2013 tar
-r-xr-xr-x 2 root bin 95440 Mar 13 2013 test
#
```

# Prawa dostępu

W systemie Unix z każdym plikiem związana jest 9 bitowa liczba całkowita, przechowująca informacje o uprawnieniach do czytania (*read*), pisania (*write*), i wykonywania pliku (*execute*), które zdefiniowane są osobno dla *właściciela* pliku, *zespołu* do którego należy, oraz *pozostałych użytkowników* systemu.

Przykładowo, sekwencja znaków: **rw-rw-rw-** w lewej kolumnie po wpisaniu komendy `ls -l` oznacza, że wszyscy mają prawo *czytać i pisać* do pliku, nikt natomiast nie może go *wykonać*.

W zapisie ósemkowym, zwyczajowo stosowanym do kodowania praw dostępu w programach, **rw-rw-rw-** odpowiada **0666**.

[ ***Dalej***, prawa dostępu pojawią się przy omawianiu funkcji **open** i **creat** ]



## Otwieranie plików (c.d.)

Jeśli operacja otwarcia się powiedzie, system przekazuje do programu pewną nieujemną liczbę całkowitą nazywaną **deskryptorem pliku** (lub `-1`, jeśli wystąpi błąd).

Wcześniej [ **zob.** [wykład07](#) ], działając na poziomie biblioteki standardowej, posługiwaliśmy się **wskaźnikami plikowymi** (np. `FILE *f`). Technicznie, struktura wskazywana przez `f` zawiera **między innymi** deskryptor pliku.

**W praktyce**, wiele odwołań dotyczy *klawiatury i ekranu*, stworzono zatem specjalne mechanizmy upraszczające te formy komunikacji.

Unixowy **interpretator poleceń** ( => powłoka; ang. *shell* ) uruchamiając program otwiera trzy pliki, o deskryptorach 0, 1, i 2, tożsame ze standardowym wejściem, wyjściem, i wyjściem błędów.

Czytając z pliku o deskrytorze 0, oraz pisząc do plików o deskrytorach 1 lub 2, nasz program może zatem wprowadzać dane i wypisywać wyniki **bez otwierania żadnych plików**.

Dalej, *standardowe wejście/wyjście* możemy przekierować do „prawdziwego” pliku dzięki mechanizmowi potoków:

```
$ ./a.out < input-file > output-file
```

Powłoka ( => *shell* ) zmieni wówczas domyślnie powiązania deskryptorów 0 oraz 1 i połączy je ze wskazanymi plikami.

[ *Zwykle deskryptor 2 pozostaje trwale związany z ekranem, aby tam pojawiały się komunikaty o błędach.* ]

W mechanizmie potoków, powiązania plików są zmieniane przez powłokę a nie przez program; program używa zatem deskryptorów (0, 1, 2) nie wiedząc, skąd dane są pobierane i dokąd wysyłane.

## Wejście/wyjście niskiego poziomu (`read` i `write`)

Funkcje `read` i `write` z nagłówka `<unistd.h>` udostępniają odwołania systemowe (o identycznych nazwach) umożliwiające elementarne operacje wejścia i wyjścia.

```
int read(int fd, char *buff, int n);  
int write(int fd, char *buff, int n);
```

**Pierwszym argumentem** każdej z tych funkcji jest deskryptor pliku, **drugim** — wskaźnik znakowy do miejsca w pamięci, w którym przychodzące dane mają przechowane (lub z którego mają być wysłane do pliku), **trzecim** — liczba bajtów do przesłania.

Każda z funkcji ( `read/write` ) zwraca liczbę faktycznie przesłanych bajtów. W szczególności, przy czytaniu liczba bajtów może być mniejsza niż żądana; 0 oznacza koniec pliku, a -1 błąd.

Najczęściej pojawiające się wartości  $n$  to 1 (wymusza „niebuforowane” przesyłanie danych po 1 bajcie) lub potęgi dwójki, w rodzaju 1024, 4096 itp, odpowiadające rozmiarowi jednostki alokacji (*bloku pamięci*) w danym systemie plików.

[ *Przesyłanie danych w większych paczkach na raz jest na ogół bardziej efektywne, gdyż wymaga mniejszej liczby odwołań do systemu.* ]

Prosty program **kopiujący dane ze standardowego wejścia** ( $\text{fd}=0$ ) **na standardowe wyjście** ( $\text{fd}=1$ ) może wyglądać np. tak [ *zob. też K & R, rozdz. 8* ]: ...

```
#include <unistd.h>
#define BUFFSIZE 1024

main()
{
    char buff[BUFFSIZE];
    int n;

    while ((n = read(0, buff, BUFFSIZE))>0)
        write(1, buff, n);

    return 0;
}
```

Za pomocą funkcji **read** i **write** możemy łatwo zbudować własne wersje funkcji wyższego poziomu (takie jak **getchar** i **putchar**).

Pokażemy teraz [ [za K&R](#) ], jak napisać funkcję **getchar** tak, aby czytała dane wejściowe dużymi porcjami (zdefiniowanymi wartością stałej `BUFSIZE`), ale wyprowadzała zawsze po jednym znaku (lub wartość `EOF`, w przypadku napotkania znacznika końca pliku).

[ *Stała **EOF** — **End Of File** — jest zdefiniowana w nagłówku `<stdio.h>`; zwykle jej wartość to `-1`, nie należy jednak zakładać, że tak jest zawsze. ]*

Wymagane zachowanie można łatwo osiągnąć używając zmiennych automatycznych o klasie pamięci **static** (!)

```

#include <stdio.h>
#include <unistd.h>
#define BUFFSIZE 1024

#undef getchar

int getchar(void)
{
    static char buff[BUFFSIZE];
    static char *bufp = buff;
    static int n = 0;

    if (n == 0) { /* pusty bufor */
        n = read(0, buff, sizeof buff);
        bufp = buff;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}

```

Dyrektywa `#undef getchar` „zakrywa” deklarację funkcji `getchar()` zawartą w pliku nagłówkowym `<stdio.h>`

## Kilka dalszych (nieoczywistych!) szczegółów:

Tablica `buf` służąca do przechowywania (*buforowania*) wczytanych znaków musi być typu `char`, ponieważ funkcja `read` akceptuje jedynie *wskazniki znakowe*.

[ *Z drugiej strony: wartość zwracana przez `getchar()` jest typu `int`, aby „zmieścić” EOF ... ]*

Przed zwróceniem wartości wczytanego znaku (*jeśli nie zwracamy EOF-a*) wykonujemy zatem rzutowanie na typ: `unsigned char` aby uniknąć możliwych problemów z bitem znaku.



# Funkcje: `open`, `creat`, `close`, `unlink`

Pliki inne niż *standardowe wejście, wyjście, czy też wyjście błędów* muszą zostać **jawnie otwarte** zanim wykonamy na nich jakiegokolwiek operacje.

W *bibliotece standardowej* [zob. **wyklad07**] służy do tego funkcja `fopen(...)` zwracająca wskaźnik plikowy.

Działając na poziomie **odwołań systemowych**, mamy do dyspozycji dwie funkcje: `open` i `creat` („*You left out the »E«*”...)

```
int open(char *name, int flags, int perms);  
int creat(char *name, int perms);
```

Funkcja `open` jest (*nieco...*) podobna `fopen`; zamiast wskaźnika plikowego zwraca deskryptor pliku.

Jeśli gdzieś w programie zadeklarujemy zmienną: `int fd;` możemy wywołać instrukcje postaci:

```
fd = open(name, flags, perms);
```

gdzie **name** wskaźnikiem do ciągu znaków zawierającego nazwę pliku; **flags** jest liczbą typu `int` określającą sposób otwarcia wskazanego pliku. Najważniejsze wartości (nagłówek: `<fcntl.h>`)

<b>O_RDONLY</b>	<i>otwarcie tylko do czytania</i>
<b>O_WRONLY</b>	<i>otwarcie tylko do pisania</i>
<b>O_RDWR</b>	<i>otwarcie do czytania i pisania</i>

Jeżeli np. chcemy otworzyć istniejący plik do czytania, piszemy:

```
fd = open(name, O_RDONLY, 0);
```

argument `perms` równy `0` oznacza, że nie zmieniamy praw dostępu.

Próba otwarcia *pliku, który nie istnieje*, na ogół będzie błędem (choć w nagłówku `<fcntl.h>` mamy zdefiniowaną wartość `O_CREAT` argumentu `flags` do tworzenia nowego pliku).

Zwykle używamy do tego drugiej funkcji ( `creat` ), której drugi argument (`perms`) określa prawa dostępu dla tworzonego pliku.

Funkcja `creat` zwraca deskryptor do utworzonego pliku.

Jeśli funkcja `creat` nie zdoła utworzyć pliku (*większość systemów ogranicza liczbę jednocześnie otwartych plików dla jednego procesu...*) zwraca `-1`.

Jeśli tworzony plik już istnieje, zostanie **wyzerowany** (tj. skrócony do zerowej długości, poprzednia zawartość ulegnie skasowaniu).

Wspomniane wyżej ograniczenia **liczby jednocześnie otwartych** plików sprawiają, że program musi być przystosowany do wielokrotnego używania tych samych deskryptorów plików.

Kiedy plik staje się niepotrzebny (tj. *nie musi dłużej pozostawać otwarty*) możemy go zamknąć, tzn. **przerwać połączenie pliku z jego deskryptorem**. Służy do tego funkcja: `close(int fd);`

[ *Funkcja `close` zasadniczo jest odpowiednikiem `fclose` z biblioteki standardowej, nie ma jednak żadnych buforów do opróżnienia.* ]

Z kolei funkcja `unlink(char *name)` usuwa plik o nazwie `name` z systemu plików. [ *Jest odpowiednikiem `remove` z biblioteki standardowej.* ]

*Podobnie, funkcja **fseek** z biblioteki standardowej (=> wyklad08) ma swój odpowiednik powiązany z odwołaniem systemowym.*

Jest nim funkcja jest **lseek** umożliwiająca **dostęp swobodny** do wybranego miejsca w otwartym pliku (a zarazem — *poruszanie się po pliku bez czytania/pisania* danych):

```
long lseek(int fd, long offset, int origin);
```

Funkcja **lseek** zmienia pozycję w pliku o deskrytorze fd na pozycję zadaną przez przesunięcie ( offset ) liczone w bajtach, od wybranego miejsca ( origin ).

**Historycznie**, wartości origin równe 0, 1, 2 oznaczają, że przesunięcie będzie liczone od początku, bieżącej pozycji, lub końca pliku.

W praktyce, zadając origin zwykle odwołujemy się do stałych zdefiniowanych w nagłówku `<unistd.h>` (których nazwy w większości przypadków są samoopisujące):

**SEEK\_SET** (od początku), **SEEK\_CUR** (...), **SEEK\_END** (od końca), **SEEK\_HOLE** (od początku *największej dziury* o rozmiarze większym lub równym wartości offset) ...

Ostatni przykład ( **SEEK\_HOLE** ) daje pojęcie o *zaletach odwołań systemowych* [ => **przenośność programu .VS. łatwość pisania ...** ]

Obszar pliku wypełniony zerami może (*ale nie musi...*) być oznaczony przez system jako „dziura” (*hole*); odwołanie `lseek` może wyprowadzać poza pozycje EOF (wówczas, zapis w takim miejscu utworzy „dziurę”).

**Informacje o pliku** (np. o nazwie name) można uzyskać za pomocą funkcji **stat** (nagłówek: `<sys/stat.h>`)

```
int stat(const char *name, struct stat *buf);
```

Dane trafiają do *struktury*, która zawiera pola:

```
struct stat {
    mode_t      st_mode;    /* atrybuty pliku */
    ino_t       st_ino;     /* numer węzła */
    dev_t       st_dev;     /* urządzenie powiązane z węzłem */
    dev_t       st_rdev;    /* info. dla plików specjalnych */
    nlink_t     st_nlink;   /* liczba linków do pliku */
    uid_t       st_uid;     /* identyfikator właściciela */
    gid_t       st_gid;     /* identyfikator zespołu */
    off_t       st_size;    /* rozmiar (w bajtach) */
    struct timespec st_atim; /* data/czas ostatniego dostępu */
    struct timespec st_mtim; /* data/czas ostat. modyfikacji */
    struct timespec st_ctim; /* [...] ostat. zmiany w węźle */
    ...
};
```

# Przydział pamięci

**Odwołania systemowe** umożliwiają także przydzielania/zwalnianie bloków pamięci dla programu. Służy do tego funkcja **sbrk(n)**, która zwraca wskaźnik (`void*`) do nowej porcji `n` bajtów pamięci, lub wartość `-1` (**a nie NULL!**) jeśli wolny blok o pożądanym rozmiarze nie został znaleziony.

*Przyjmuje się założenie, że wskaźniki zwracane przez **sbrk** mogą być **porównywane** (tak jak wskaźniki powiązane z tą samą tablicą!).*

Możemy w szczególności napisać własną wersję funkcji bibliotecznej **malloc** lub **calloc** [ **zob. Kernighan & Ritchie, ...** ]



# Nagłówek <stdint.h>

W niektórych zastosowaniach wygodnie jest **dokładnie ustalić** szerokość (tj. *liczbę bitów*) zmiennej całkowitej (*standard języka* — poza typem `char` — określa jedynie **minimalną** l.bitów)

Odpowiednie makra również należą do **POSIX**, a są to:

`int8_t`

`int16_t`

`int32_t`

`uint8_t`

`uint16_t`

`uint32_t`

Ponadto, w implementacjach mogą (*ale nie muszą*) pojawiać się:

`int64_t`

`uint64_t`

Odpowiednie makra są zdefiniowane w nagłówku <stdint.h>

# Odwołania systemowe: *Podsumowanie*

Standard **POSIX** zapewnia identyczność (z punktu widzenia programisty...) licznych funkcji zdefiniowanych w nagłówku `<unistd.h>` a udostępniających **odwołania systemowe**.

**Odwołania systemowe** umożliwiają w szczególności:

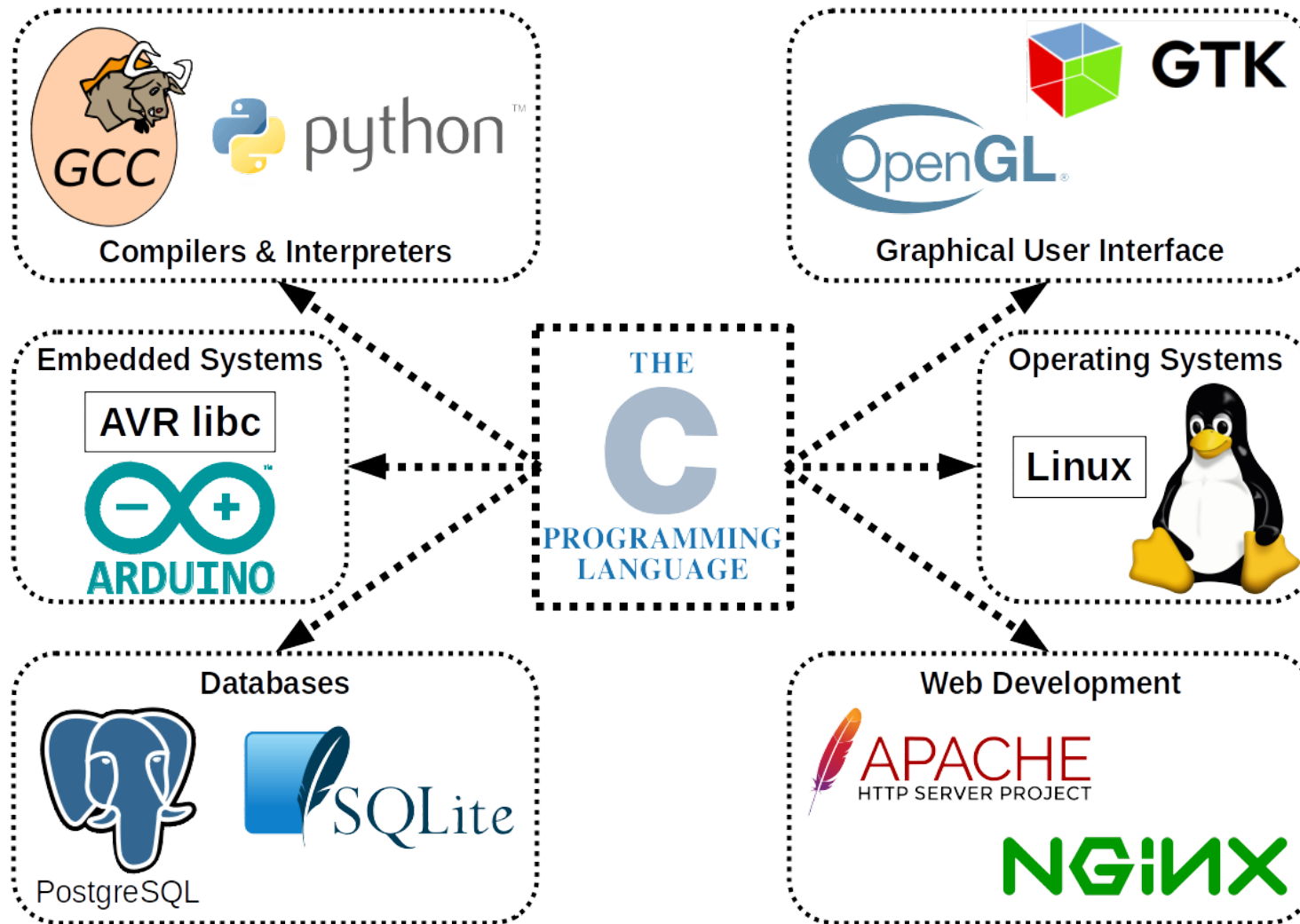
- niskopoziomowe operacje wejścia/wyjścia
- działania w systemie plików
- zarządzanie pamięcią

W wielu przypadkach, istnieją **funkcje biblioteki standardowej** zastępujące odwołania. Zwykle są prostsze w użyciu (kosztem wydajności ...) i powiązane ze standardem języka a nie systemu.

# Najważniejsze zmiany w standardzie C99

- **Usunięto prototypy domyślne(!)**
- Możliwe **mieszanie deklaracji z instrukcjami** oraz deklaracje zmiennych w części inicjującej pętli *for* (a także *while*) [por. C++]
- Dodano **tablice o uzmiennionym rozmiarze** (ang. Variable Size Arrays, VLA) [*cecha została usunięta w C11...*]
- Ostatnim składnikiem struktury może być tablica o niezdefiniowanym rozmiarze (ang. flexible array member)
- Dozwolone komentarze zaczynające się od `//` (por. C++, Java)
- **Nowe typy danych:** `long long int` (co najmniej 64 bity), a także: `bool`, `complex` (zdefiniowane w plikach: `<stdbool.h>`, `<complex.h>`)
- Makra o zmiennej liczbie argumentów

# Współczesne zastosowania C



[ Źródło: [https://commons.wikimedia.org/wiki/File:The\\_C\\_Programming\\_Language.png](https://commons.wikimedia.org/wiki/File:The_C_Programming_Language.png) ]