

# Nieco różności ...

Cygwin: emulator Linux-a pod Windows [ *zawiera m.in. gcc* ]

# Cygwin

Get that [Linux](#) feeling - on Windows

## This is the home of the Cygwin project

### What...

#### ...is it?

Cygwin is:

- a large collection of GNU and Open Source tools which provide functionality similar to a [Linux distribution](#) on Windows.
- a DLL (cygwin1.dll) which provides substantial POSIX API functionality.

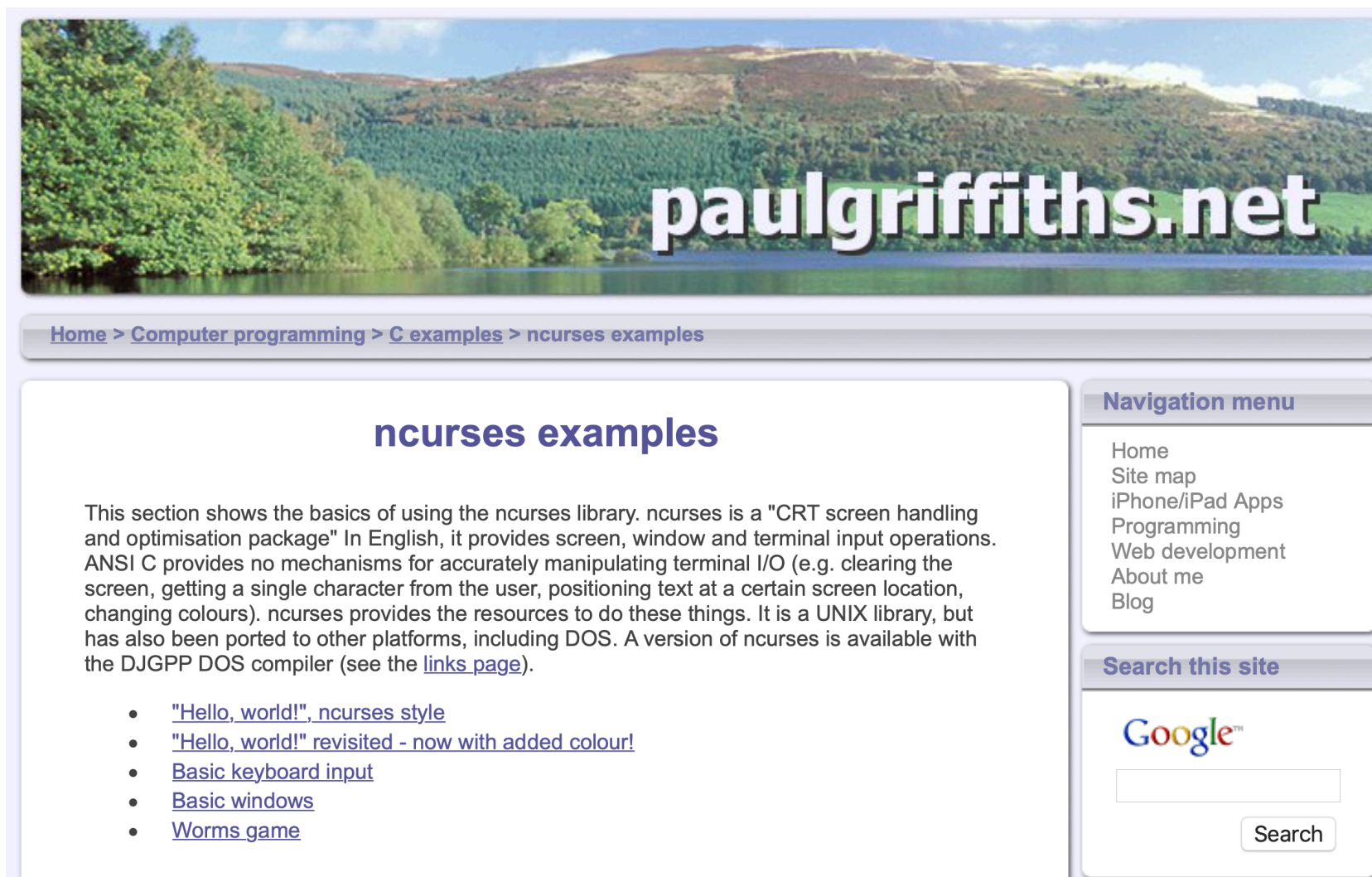
#### ...isn't it?

Cygwin is not:

- a way to run native Linux apps on Windows. You must rebuild your application *from source* if you want it to run on Windows.
- a way to magically make native Windows apps aware of UNIX® functionality like signals, ptys, etc. Again, you need to build your apps *from source* if you want to take advantage of Cygwin functionality.

zob: <https://www.cygwin.com>

# Przykłady zastosowań biblioteki `ncurses` ( i wielu innych ... )



The screenshot shows the website [paulgriffiths.net](http://www.paulgriffiths.net). The header features a landscape image with the text "paulgriffiths.net" overlaid. Below the header is a breadcrumb trail: [Home](#) > [Computer programming](#) > [C examples](#) > [ncurses examples](#). The main content area is titled "ncurses examples" and contains a paragraph explaining the library: "This section shows the basics of using the ncurses library. ncurses is a "CRT screen handling and optimisation package" In English, it provides screen, window and terminal input operations. ANSI C provides no mechanisms for accurately manipulating terminal I/O (e.g. clearing the screen, getting a single character from the user, positioning text at a certain screen location, changing colours). ncurses provides the resources to do these things. It is a UNIX library, but has also been ported to other platforms, including DOS. A version of ncurses is available with the DJGPP DOS compiler (see the [links page](#))." Below the paragraph is a bulleted list of links: 

- ["Hello, world!", ncurses style](#)
- ["Hello, world!" revisited - now with added colour!](#)
- [Basic keyboard input](#)
- [Basic windows](#)
- [Worms game](#)

On the right side, there is a "Navigation menu" with links to Home, Site map, iPhone/iPad Apps, Programming, Web development, About me, and Blog. Below that is a "Search this site" section with a Google search box and a "Search" button.

zob: <http://www.paulgriffiths.net/program/c/curses.php>

Co zrobić, gdy musimy np. rozwiązać duży *układ r. liniowych*?

$$„ \mathbf{Ax} = \mathbf{b} ”$$

==> **Biblioteka GSL** (*GNU Scientific Library*):

*Uniwersalne narzędzie do różnorodnych problemów numerycznych (całkowanie, rozwiązywanie równań różniczkowych, obliczanie wartości funkcji specjalnych, minimalizacja funkcji, ... )*

[ *Zawarta w wielu dystrybucjach Linuxa; także **część Cygwin-a*** ]

zob. <https://www.gnu.org/software/gsl/>

**Inna możliwość:** LAPACK [ ==> dla zainteresowanych: [wyklad11a.pdf](#) ]

# Poprzedni wykład [ 4. 01. 2022 ] :

- **Struktury** (deklaracje, kopiowanie, przypisanie)
- Funkcje operujące na strukturach i *zwracające* struktury
- Wskaźniki na struktury i tablice struktur

# Struktury rekurencyjne

*Na poprzednim wykładzie* dyskutowaliśmy problem wyszukiwania słów kluczowych języka C.

[ *Problem był szczególnie prosty, ponieważ lista 32 takich słów jest **zamknięta**; można było zatem jawnie wypisać w programie wszystkie szukane słowa w porządku alfabetycznym.* ]

**Na ogół jest inaczej** — jeśli np. zliczamy słowa w jakimś *długim* tekście wejściowym, *lista* ( a nawet *liczba!* ) słów nie jest znana wcześniej, pojawiają się też one w losowej kolejności.

=> W takich sytuacjach przydają się bardziej zaawansowane struktury danych, jak np. **drzewa binarne**.

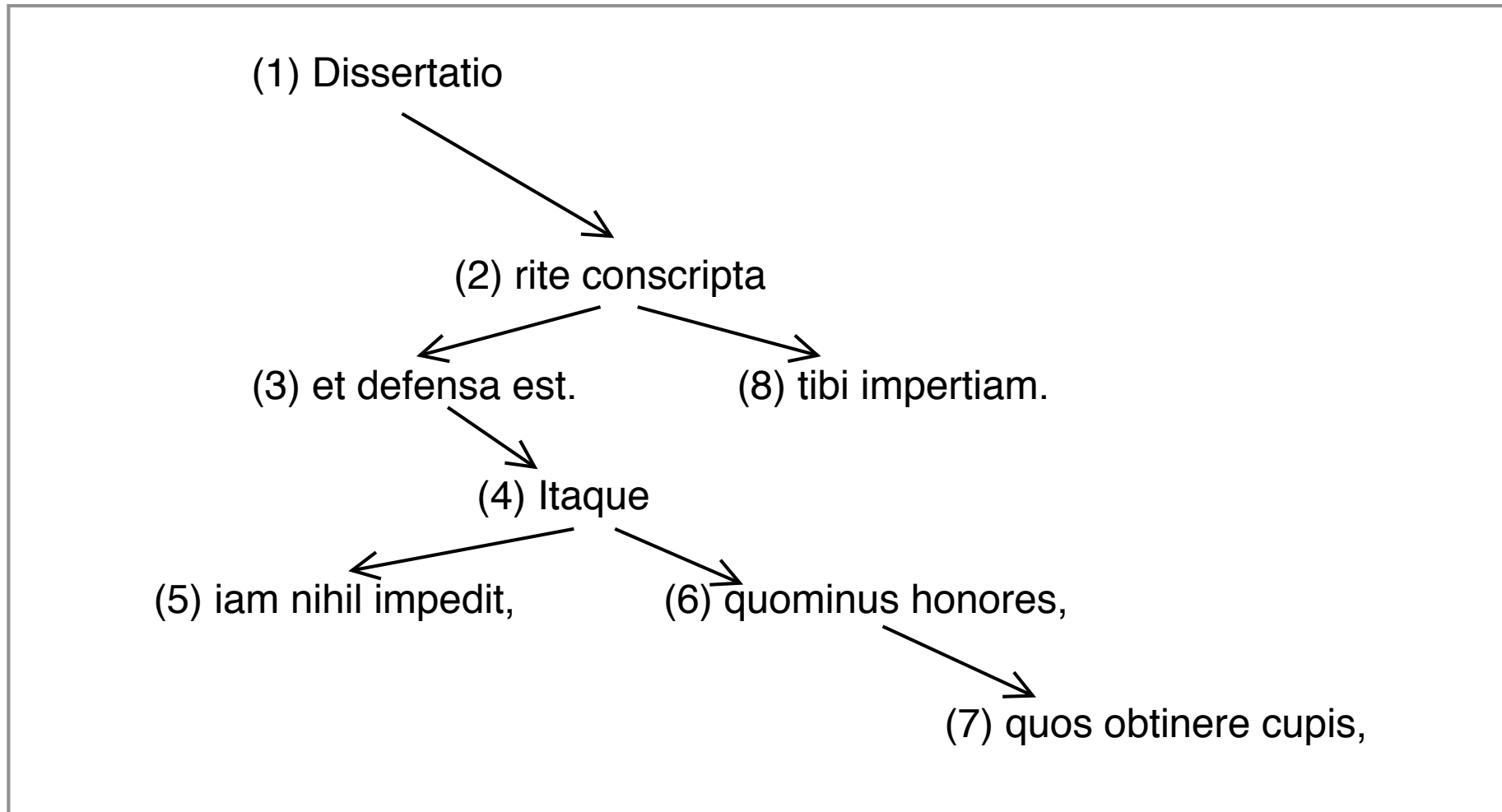
Wrócimy teraz do naszego przykładu z **porządkowaniem alfabetycznym** wierszy tekstu [ zob. [wyklad08.pdf](#) ]

Założymy, że napisy *mogą się powtarzać*, ich liczba nie jest z góry znana, a dodatkowo — *chcemy mieć możliwość wypisania posortowanych napisów w każdej chwili*.

Wiersze dobrze jest zatem przechowywać w strukturze dynamicznej, rozrastającej się w miarę potrzeb, które jednocześnie przechowuje informację o uporządkowaniu, tak aby wypisanie napisów (w kolejności alfabetycznej) było łatwe na każdym etapie powstawania struktury.

==> Taką strukturą jest **drzewo binarne**.

**Drzewo dla tekstu:** „(1) *Dissertatio* (2) *rite conscripta* (3) *et defensa est.* (4) *Itaque* (5) *iam nihil impedit,* (6) *quominus honores,* (7) *quos obtinere cupis,* (8) *tibi impertiam.*” będzie wyglądać tak —



```
/* Sortowanie wierszy: Wersja 3 (z drzewem binarnym) */
/* AR, 2018 + elementy: Kernighan-Ritchie, 1994 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct tnode {    /* wezel drzewa */
    char *line;   /* wiersz tekstu */
    int count;    /* licznik wystapien */
    struct tnode *left;    /* lewy potomek */
    struct tnode *right;   /* prawy potomek */
};

struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int treesize(struct tnode *);
int treedepth(struct tnode *);
```



```
#define MAXLINE 2000 /* dopuszczalna dlugosc wiersza */

int main()
{
    struct tnode *root = NULL; /* korzen drzewa */
    char line[MAXLINE+1];

    while (fgets(line, MAXLINE+1, stdin))
        root = addtree(root, line);
    fprintf(stderr, "=> Wprowadzono %d roznych wierszy. "
        "Glebokosc drzewa: %d <==\n",
        treesize(root), treedepth(root));
    treeprint(root);
    return 0;
}
```

```
/* treeprint: wypisz uporządkowane drzewo */
void treeprint(struct tnode *p)
{
    int i;

    if (p != NULL) {
        treeprint(p->left);
        for (i=0; i < p->count; i++)
            printf("%s\n", p->line); /* Dopisujemy '\n'! */
        treeprint(p->right);
    }
}
```

```
/* treesize:  liczba wezlow drzewa */  
int treesize(struct tnode *p)  
{  
    if (NULL == p)  
        return 0;  
    else  
        return treesize(p->left) + 1 + treesize(p->right);  
}
```

```

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

/* treedepth:  glebokosc drzewa */
int treedepth(struct tnode *p)
{
    int ldep, rdep;

    if (NULL == p)
        return 0;
    else {
        ldep = treedepth(p->left);
        rdep = treedepth(p->right);
        return 1 + max(ldep, rdep);
        /*
            A dlaczego nie napisac tak:  ( Do przemyślenia! )
            >> max(treedepth(p->left), treedepth(p->right)) <<
        */
    }
}

```

```

/* addtree: dodaj wezel dla s; ... */
struct tnode *addtree(struct tnode *p, char *s)
{
    int len, cond;

    if (NULL == p) { /* nowy wiersz */
        len = strlen(s); /* pomijamy '\n' */
        p = (struct tnode *)malloc(sizeof(struct tnode));
        if (!(p) || !(p->line = strdup(s, len-1))) {
            fprintf(stderr, "ERROR: INPUT TOO BIG TO SORT\n");
            exit(1);
        }
        p->count = 1;
        p->left = p->right = NULL;
    }
    else if ((cond = strcmp(s, p->line)) == 0 ) {
        p->count++; /* powtórzony wiersz */
    }
    /* c.d.n. */
}

```

```

/* c.d. – (s) inny niż (p->line) */
/* ==> sprawdzamy pod-drzewa: */

else if (cond < 0) { /* wiersz leksykalnie MNIEJSZY */
    p->left = addtree(p->left, s);
}
else { /* wiersz leksykalnie WIEKSZY */
    p->right = addtree(p->right, s);
}

return p;
}

```

**Porównanie czasu działania** z „klasyczną” wersją programu [ *napisy przechowywane w tablicy wskaźników znakowych, algorytm quicksort* ] dla tekstu nieuporządkowanego (~5000 wierszy):

```
$ time ./napisy2.out <AR_dokto.tex >output2.txt
real 0m0.010s
user 0m0.005s
sys 0m0.003s
```

```
$ time ./napisy3.out <AR_dokto.tex >output3.txt
==> Wprowadzono 5025 roznych wierszy. Glebokosc drzewa: 328 <==
real 0m0.012s
user 0m0.007s
sys 0m0.003s
```

=> ***BRAK istotnej różnicy w wydajności !***

Jeśli jednak na wejście podamy tekst uporządkowany, sytuacja zmienia się **dramatycznie**:

```
$ time ./napisy2.out <output2.txt >output2a.txt
real 0m0.010s
user 0m0.005s
sys 0m0.003s
```

```
$ time ./napisy3.out <output2.txt >output3a.txt
==> Wprowadzono 5025 roznych wierszy. Glebokosc drzewa: 5025 <==
real 0m0.150s
user 0m0.146s
sys 0m0.003s
```

=> **~ 20-krotne spowolnienie wersji z drzewem;** w. klasyczna działa tak samo wydajnie (jak dla tekstu nieuporządkowanego).



# Dlaczego tak się dzieje?

W przypadku tekstu nieuporządkowanego, *głębokość drzewa* jest rzędu  $\sim \log_2 N$ , gdzie  $N$  to liczba porządkowanych elementów, tego samego rzędu będą też liczby rekurencyjnych wywołań funkcji `addtree` i `treeprint`, a zatem algorytm może działać ***równie szybko jak quicksort.***

Jeśli jednak tekst wejściowy jest uporządkowany (*alfabetycznie* lub *antyalfabetycznie*) drzewo rozrasta się tylko w jednym kierunku (odpowiednio: *prawym* lub *lewym*), do głębokości  $N$ .

W takiej sytuacji, drzewo degeneruje się do *jednokierunkowego łańcucha odsyłaczy*, a algorytm przeszukiwania staje się ***równoważny przeszukiwaniu liniowemu.***

Struktury **odwołujące się do samych siebie** niezwykle upraszczają implementację wielu algorytmów, bywają jednak są mało wydajne:

Kolejne *rekordy* zajmują często odległe miejsca w pamięci; w specyficznych przypadkach takie struktury mogą przybierać formy *zdegenerowane*, co zwykle prowadzi do drastycznego spadku wydajności oprogramowania.

[ *A zatem — jeśli to tylko możliwe, używamy **prostszych konstrukcji**, najlepiej dedykowanych do konkretnych problemów ...* ]

# Tablice mieszające (tzw. „*haszmapy*” )

Kolejny przykład ilustrujący własności struktur rekurencyjnych dotyczyć będzie zagadnienia szybkiego przeoglądania tablic, np. *w poszukiwaniu **ustalonego wzorca***.

*W uproszczeniu: **tablice mieszające** są formą przechowywania [ lub raczej: **indeksowania** ] danych o własnościach pośrednich pomiędzy strukturami rekurencyjnymi a klasycznymi tablicami:*

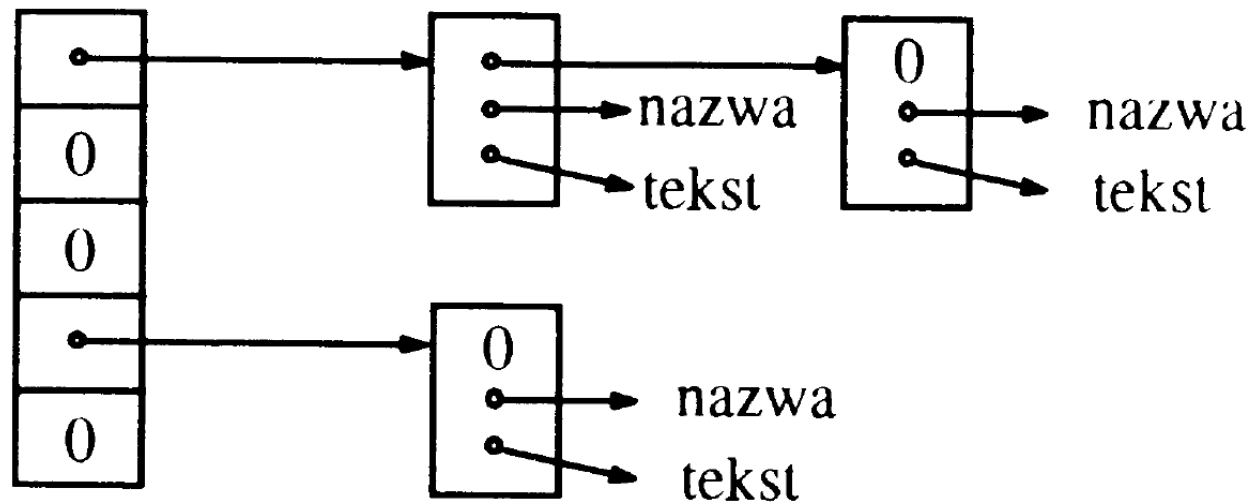
- pozwalają przechowywać ilość danych *ograniczoną jedynie dostępną pamięcią maszyny*
- dostęp do danych jest przyspieszony w ten sposób, że każdej *jednostce danych* przyporządkowana jest pewna **liczba nieujemna**, która definiuje pozycję w tablicy ( $\Rightarrow$  *haszowanie*).

[ *Komórka tablicy haszującej to najczęściej wskaźnik „kotwiczący”  
łańcuch odsyłaczy lub drzewo binarne.* ]

Przestudiujemy teraz zestaw funkcji, pozwalających na zapamiętywanie *nazw* i *zastępujących je tekstów*; podobnie jak robi to preprocesor języka C przetwarzając makrodefinicje:

```
#define tekst1 "Mam dzisiaj dobry dzień.\n"
```

(Podobne narzędzia pojawiają się np. w algorytmach kompresji danych.)



Tablicę obsługiwać będzie para funkcji:

`install(s,t)` — rejestruje nazwę `s` i zastępujący ją tekst `t`; reprezentowane jako ciągi znaków.

`lookup(s)` — przegląda tablicę w poszukiwaniu nazwy `s` i zwraca wskaźnik do miejsca, w którym nazwa jest zapisana, lub `NULL`, jeśli nazwy nie ma w tablicy.

Dodatkowo, funkcja `hash(s)` przyporządkowuje każdej nazwie pewną (*niezbyt dużą*) liczbę nieujemną (tzw. wartość rozproszenia), której używamy do indeksowania tablicy.

Przyjmujemy, że każdy element tablicy to wskaźnik do początku **listy** nazw i zastępujących tekstów, reprezentowanej w formie łańcucha odsyładczy ( NULL oznacza koniec listy ).

**Ogniwo łańcucha** będzie zatem *strukturą* postaci:

```
struct nlist {      /* ogniwo łańcucha */
    struct nlist *next; /* następne ogniwo */
    char *name;      /* nazwa */
    char *defn;     /* tekst zastępujący */
};
```

**Tablica wskaźników** może z kolei wyglądać tak:

```
#define HASHSIZE 101
static struct nlist *hashtab[HASHSIZE];
```

Przykładowa, prosta *funkcja mieszająca*:

```
/* hash: wyznacz wartość rozproszenia dla s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31U * hashval;
    return (hashval % HASHSIZE);
}
```

[ *Arytmetyka liczb całkowitych bez znaku* — unsigned — gwarantuje, że obliczona wartość będzie **nieujemna**. ]

**Wartość rozproszenia** to indeks początkowego wskaźnika w `hashtab` — jeśli poszukiwana nazwa `s` jest gdziekolwiek zapisana, to na pewno jest zapisana w łańcuchu zaczynającym się od wskaźnika: `hashtab[hash(s)]`

**Funkcja przeszukująca** tablicę:

```
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np=hashtab[hash(s)]; np!=NULL; np=np->next)
        if (!strcmp(s, np->name))
            return np; /* ==> znaleziono */
    return NULL;
}
```



Pętla w funkcji `lookup` zawiera często spotykaną w języku C konstrukcję, która służy do *poruszania się wzdłuż jednokierunkowego łańcucha odsyłaczy*:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

Druga funkcja ( `install` ) będzie używać `lookup` do sprawdzania, czy wprowadzana właśnie nazwa już występuje w łańcuchu: jeśli tak, *nowa definicja **zastąpi starą***, jeśli nie — *zostanie utworzone kolejne ogniwo łańcucha*.

```

/* install: umieść (name, defn) w tablicy hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np, *lookup(char *);
    unsigned hashval;

    if (!(np = lookup(name))) { /* nie znaleziono */
        np = (struct nlist *)malloc(sizeof(*np));
        if ( !(np) || !(np->name = strdup(name)) )
            return NULL; /* błąd przydziału pamięci ... */
        hashval = hash(name);
        np->next = hashtab[hashval]; /* wymiana głowy */
        hashtab[hashval] = np;      /* ==> F.I.L.O.(!)* /
    } else /* nazwa już jest - */
        free((void *)np->defn);     /* c.d.n. */
}

```

```
/* c.d. – kopiowanie "defn" */  
if (!(np->defn = strdup(defn)))  
    return NULL;  
  
return np;  
}
```

# Deklaracje typedef

W przypadku skomplikowanych typów danych (por. struktury!), deklaracje zmiennych i funkcji często można znacząco uprościć definiując „nowe” typy. [ *Tak naprawdę — typy deklarowane typedef-em zawsze są **synonimami** już istniejących.* ]

**Składnia deklaracji** jest niemal identyczna jak w przypadku *deklaracji zmiennych*, z tym, że na początku pojawia się dodatkowo słowo `typedef`, a identyfikator — który normalnie oznaczałby nazwę zmiennej — staje się **synonimem typu**.

**Przykładowo:**

```
typedef int integer;
```

tworzy *synonim* typu `int` o nazwie `integer`.

Z kolei:

```
typedef int IntTab[100];
```

tworzy synonim `IntTab` dla ***tablicy 100 liczb całkowitych***.

[ **Deklarowany typ pojawia się zawsze *w miejscu nazwy zmiennej!*** ]

Ostatnia deklaracja pozwala łatwo się przekonać, że **`typedef`** w istocie nie tworzy nowego typu, a jedynie synonim:

*Jeśli spróbujemy napisać funkcję zwracającą „wartość typu `IntTab`”, kompilator zgłosi błąd! `IntTab` nadal **jest tablicą**, nie może zatem występować jako typ-zwracany przez funkcję.*

*( To ograniczenie można w razie potrzeby obejść opakowując tablicę w strukturę ... )*

W pojedynczej deklaracji **typedef** nowa nazwa może pojawiać się tylko raz; deklaracje dla **struktury rekurencyjnej** i **wskaźnika** do niej mogą wyglądać tak:

```
typedef struct tnode *Treenode; /* wskaźnik! */
```

```
typedef struct tnode { /* wezel drzewa */
    char *line; /* wiersz tekstu */
    int count; /* licznik wystapien */
    Treenode left; /* lewy potomek */
    Treenode right; /* prawy potomek */
} Treenode;
```

Funkcja **tworząca nowy węzeł**:

```
Treenode talloc(void)
{ return (Treenode)malloc(sizeof(Treenode)); }
```

## Dalsze uwagi o typedef:

- Deklaracje **typedef** nie są dyrektywami preprocesora (*są kompilowane!*), a zatem możliwości są tutaj dużo większe. Przykładowo, synonim wskaznika do funkcji typu int, której dwa argumenty są typu `char *`, tworzymy tak:

```
typedef int (*PFI)(char *, char *);
```

po czym możemy pisać prototypy: `PFI strcmp; ...`

[ *wygodne, jeśli często zmieniamy zdanie...* ]

- Jeśli zależy nam na przenośności oprogramowania, warto używać synonimów **typedef** dla tych typów, które mogą zależeć od maszyny — dobrym przykładem jest `size_t` z biblioteki standardowej języka C.

# Unie (oryg. union)

*Unia* jest zmienną, która — raz zadeklarowana — może później **przyjmować wartości różnych typów**.

Składnia deklaracji jest b. *podobna do struktur*:

```
union u_tag {  
    int ival;  
    double fval;  
    char *sval;  
} u;
```

Zmienna `u` będzie wystarczająco obszerna, aby pomieścić wartości wszystkich trzech typów składowych.



**Unie inicjujemy** zawsze wartością tego typu, jaki ma *pierwsza składowa* (tutaj: `int` ).

Typ *wartości pobieranej* musi być zawsze taki sam, jak wartości *ostatnio przypisanej*.

Unie można także **przypisywać i kopiować w całości**.

Odwołania do **składowych unii** mają postać identyczną jak w przypadku struktur, tzn:

*nazwa-unii.składowa* albo *wskaźnik-do-unii->składowa*

Podobnie, w przypadku **tablic, tablic wskaźników, oraz struktur zawierających unie**, przenosimy odpowiednie konstrukcje leksykalne stosowane dla struktur.

```
/* https://www.geeksforgeeks.org/union-c/ */  
  
#include <stdio.h>  
  
union test {  
    int x;  
    char y;  
};  
  
int main()  
{  
    union test p1;  
    union test* p2 = &p1; /* p2 is a pointer to p1 */  
    p1.x = 65;  
  
    /* Accessing union members using pointer */  
    printf("%d %c", p2->x, p2->y);  
    return 0;  
}
```

# Pola bitowe

Kiedy chcemy oszczędzać pamięć — język C pozwala na *upakowanie kilku odrębnych zmiennych całkowitych* w jednym **słowie maszynowym**. Przykładowo, deklaracja:

```
struct date
{
    unsigned int d: 5;    /* (d < 32) ==> 5 bits */
    unsigned int m: 4;    /* (m < 16) => 4 bits */
    unsigned int y;
}; /* => https://www.geeksforgeeks.org/bit-fields-c/ */
```

pozwole na przechowywanie *dnia* i *miesiąca* we wspólnym słowie.

Odwołania — jak do **składowych struktur**. ( *x.d*, *x.m*, ... )

[ *Bardzo wiele własności pól bitowych zależy od implementacji ...* ]