Kurs C

Radosław Łukasik

Wykład 3

Wskaźniki na funkcje

Wskaźniki mogą również wskazywać na funkcję. Dzięki temu możemy wykonywać różne czynności na tych samych danych podając tylko funkcję, którą chcemy wywołać. Utworzenie tego wskaźnika, przypisanie mu jakiejś funkcji oraz wywołanie wygląda następująco:

```
typ_funkcji (*nazwa_wsk_funkcji) (typ1 arg1, ... ,typn argn);
nazwa_wsk_funkcji=nazwa_funkcji;
zmienna=nazwa_wsk_funkcji(arg1, ... ,argn);
```

Taki wskaźnik na funkcję można również umieścić bezpośrednio w innej funkcji.

Przykład

```
int dodawanie(int a, int b){
    return a + b;
}

int mnozenie(int a, int b){
    return a * b;
}

int oblicz(int a, int b, int( *pDzialanie )(int, int)){
    return pDzialanie(a, b);
}

int main(){
    printf("Wynik mnozenia  = %d\n",oblicz(3, 4, mnozenie));
    printf("Wynik dodawania = %d\n",oblicz(3, 4, dodawanie));
    return 0;
}
```

Przy okazji wskaźników na wskaźnik mówiliśmy, że można je wykorzystać gdy chcemy zmienić nie zawartość tablicy a jej położenie czy rozmiar.

Przykład

```
void funkcja(int **pTab,int *n){
    if(*n<2048){
        int *tmp=(int*)realloc(*pTab,2048*sizeof(int));
        if(tmp!=0){
            *pTab=tmp;// zmieniamy wskaźnik tab z main()
            *n=2048;
            // jajkieś operacje na tablicy, o której już
        }   // wiemy, że ma rozmiar >=2048
    }
}

int main(){
    int n=1024;// rozmiar tablicy
    int *tab=(int*)malloc(n*sizeof(int));// początkowo 1024 el.
    if(tab!=0){
        funkcja(&tab,&n);
    }
    printf("Rozmiar %d\n",n);
    return 0;
}
```

Rekurencja funkcji

Przez rekurencję rozumiemy wywołanie funkcji wewnątrz niej samej (bezpośrednio lub poprzez inną funkcję). Używając jej należy się upewnić, że w pewnym momencie funkcja już nie będzie wywoływana oraz że ilość wywołań nie zapełni nam stosu (poprzez tworzenie zmiennych lokalnych i odkładanie parametrów funkcji na stosie).

Przykład

Rozpatrzmy funkcję obliczającą nam wyrazy ciągu Fibonacciego.

```
// deklaracja
int fib(int n);
:
:
// definicja
int fib(int n){//złożoność wykładnicza
    if(n<3){
        return 1;
    }else{
        return (fib(n-2)+fib(n-1));
    }
}
```

W niektórych przypadkach da się uniknąć rekurencji (zmniejszając złożoność pamięciową, a czasem nawet zmniejszając złożoność czasową) stosując inne rozwiązania.

Przykład

Dla ciągu Fibonacciego możemy zapamiętywać ostatnie dwa obliczone wyrazy ciągu zamiast liczyć je kilka razy. Da nam to złożoność liniową.

```
int fib(int n){//złożoność liniowa
    if(n<3){
        return 1;
    }else{
        int a=1;
        int b=1;
        int temp;
        for(int c=3;c<=n;c++){
            temp=b+a;
            a=b;
            b=temp;
        }
        return b;
    }
}
```

Kolejny przykład rekurencji, którą rozwiązaliśmy wcześniej uzyskując taką samą czasową złożoność liniową. W poniższym rozwiązaniu złożoność pamięciowa (stos) wynosi $\mathcal{O}(n)$.

Obliczanie silni

```
unsigned long long SILNIA(unsigned n){
    if(n<2)
        return 1;
    return n*SILNIA(n-1);
}
```

Funkcje o zmiennej liczbie argumentów

Uwaga w C++ występują funkcje o parametrach domyślnych. Standard C tego nie zawiera.
Mamy za to możliwość używania funkcji o zmiennej liczbie parametrów. Potrzebna jest do tego biblioteka **stdarg.h**. W samej funkcji za wszystkimi argumentami, które występują na stałe we funkcji dajemy 3 kropki.

```
typ nazwa(argumenty_stale, int ostatni_arg_staly, ... ){
    // treść funkcji
}
```

Żeby móc używać zmiennych wymienionych za ostatnim stałym parametrze najpierw musimy utworzyć listę i powiązać ją z naszymi dodatkowymi zmiennymi funkcji (występujące po ostatnim argumencie stałym)

```
va_list nazwa_listy;
va_start( nazwa_listy, ostatni_arg_staly );
```

Dostęp do kolejnych parametrów dostarcza nam funkcja zwracająca zmienną typu, który podamy

```
va_arg( nazwa_listy, typ_kolejnego_parametru );
```

Na sam koniec musimy po sobie posprzątać używając

```
va_end ( nazwa_listy );
```

Przykład

```c
#include <stdio.h>
#include <stdarg.h>

void PrintFloats(int n, ...){
    double val;
    printf("Wyswietlanie liczb:");
    va_list vl;
    va_start(vl,n);
    for (int i=0;i<n;i++){
        val=va_arg(vl,double);
        printf(" [%.2f]",val);
    }
    va_end(vl);
    printf ("\n");
}

int main (){
    PrintFloats(3,3.14,2,1.41421);
    return 0;
}
```

Funkcje inline

Funkcja **inline** (dostępna od C99) jest funkcją, której kod wstawiany jest w miejsce wywołania zamiast wywoływania tej funkcji. Jeżeli funkcja jest krótka i wywołana tylko kilka razy, to czasem lepiej wstawić jej kod w miejsce wywołania niż wywoływać ją jako funkcję. Nowoczesne kompilatory mimo użycia dyrektywy **inline** same decydują o tym czy funkcję traktować jako **inline** czy nie. Czasami mogą występować błędy podczas linkowania dla funkcji inline. Wystarczy wtedy taką funkcję uczynić statyczną (**static** przed **inline**).

Przykład

```
// deklaracja
inline float kwadrat(float);
:
:
// definicja
inline float kwadrat(float x){
    return (x*x);
}
:
:
float y=2;
float z=kwadrat(x); // ta linia zostaje przetłumaczona na kod
                    // float z=x*x;
:
:
```

Argumenty funkcji main

Funkcja **main** może mieć argumenty. Służą one jako parametry wywołania programu. Mają one postać:

```
int main(int argc, char *argv[]){
    ⋮
}
```

- Zmienna **argc** mówi o rozmiarze tablicy **argv** (o jeden więcej), pierwszy element wskazuje na nazwę programu (wraz ze ścieżką), ostatni element tej tablicy jest wskaźnikiem zerowym.
- Jeżeli są jakieś parametry wywołania programu, to **argc**$\geqslant 2$.

Przykład - wypisanie parametrów wywołania

```
#include <stdio.h>

int main(int argc, char *argv[]){
    for(int k = 0; k<argc; k++)
        printf("%s\n",argv[k]); // wypisujemy kolejne parametry
    printf("\nWcisnij enter, aby zamknac program.");
    char c;
    scanf("%c",&c);
    return 0;
}
```

Wyszukiwanie binarne

Można je stosować tylko do posortowanych tablic. Polega na porównywaniu poszukiwanego elementu z elementem w środku tablicy, a następnie zawężaniu przeszukiwanej tablicy do połówki w której może występować ten element.

```
int WyszukiwanieBinarne(int *tablica, unsigned n, int elem){
    unsigned pocz=0, koniec=n-1;
    while(pocz<=koniec){
        unsigned srodek=(pocz+koniec)>>1;
        if(elem<tabl[srodek]){
            koniec=srodek-1;
        }else if(elem>tabl[srodek]){
            pocz=srodek+1;
        }else return srodek; // zwraca pozycje lub
    }
    return n; // zwraca n gdy nie znaleźliśmy
}
```

Złożoność tego wyszukiwania wynosi $\mathcal{O}(\log_2 n)$.

Szybkie sortowanie - kod w C

```
void sortowanie_szybkie(int tab[], unsigned lewy, unsigned prawy){
    int i=(lewy+prawy)>>1;
    int j=lewy;
    int temp;
    temp=tab[i];tab[i]=tab[j];tab[j]=temp;
    for(i=lewy+1;i<=prawy;i++)
        if(tab[i]<tab[lewy])
            temp=tab[i];tab[i]=tab[++j];tab[j]=temp;
    temp=tab[lewy];tab[lewy]=tab[j];tab[j]=temp;
    if(lewy+1<j) sortowanie_szybkie(tab,lewy,j-1);
    if(j+1<prawy) sortowanie_szybkie(tab,j+1,prawy);
}
```

Powyższy algorytm wywołujemy z prawym końcem równym rozmiarowi -1, a lewym równym 0.

Istnieją pewne optymalizacje szybkiego sortowania, które zapobiegają złożoności czasowej $\mathcal{O}(n^2)$ i złożoności $\mathcal{O}(n)$ na stosie. Przykładem jest tu sortowanie introspektywne będące połączeniem sortowań szybkiego, przez kopcowanie i wstawianie. Ma ono złożoność czasową $\mathcal{O}(n \log n)$ oraz pamięciową $\mathcal{O}(\log n)$.

Sortowanie przez scalanie - kod w C

```
void sortowanie_scalanie(int tab[],int pom[],int i_p, int i_k){
    int i_s,i1,i2,i;    // wywołane z i_p =0, i_k =n
    i_s=(i_p + i_k)>>1;
    if(i_s-i_p>1)
        sortowanie_scalanie(tab,pom,i_p,i_s);
    if(i_k-i_s>1)
        sortowanie_scalanie(tab,pom,i_s,i_k);
    i1=i_p;
    i2=i_s;
    for(i=i_p;i<i_k;i++)
        pom[i] = ((i1 == i_s) || ((i2 < i_k) && (tab[i1]>tab[i2]))) ?
                    tab[i2++] : tab[i1++];
    for(i=i_p;i<i_k;i++)
        tab[i]=pom[i];
}
```

Istnieją wersje tego algorytmu niewymagające rekurencji (i przez to nieco szybsze). By przyspieszyć działanie, dla małych długości podtablic można wewnątrz wywołać sortowanie przez wstawianie zamiast sortowania przez scalanie. Można sprawdzić, że szybkie sortowanie jest nieco szybsze dla losowych danych nie zawierających zbyt wielu powtórzeń elementów, gdy wystąpień tego samego elementu jest dość dużo, to quicksort staje się dość powolny i może powodować przepełnienie stosu (rozmiar tablicy rzędu 1MB, liczby od 0 do 255 lub mniej).

Sortowanie przez kopcowanie - kod w C

```
void sortowanie_kopcowanie(int tab[],int n){
    int i,j,k,m,x,temp;
    while(i<n){// Budujemy kopiec
        k=i>>1; j=i; x = tab[i++];
        while((k>0) && (tab[k]<x)){
            tab[j] = tab[k]; j=k; k>>=1;
        }
        if(tab[0]<x){
            tab[j]=tab[0]; j=0;
        }
        tab[j] = x;
    }
    for(i=n-1;i>0;i--){// Rozbieramy kopiec
        temp=tab[0];tab[0]=tab[i];tab[i]=temp;
        j = 0; k = 1;
        while(k<i){
            m=k;
            if((k+1<i) && (tab[k+1]>tab[k])) m++;
            if(tab[m]<=tab[j]) break;
            temp=tab[m];tab[m]=tab[j];tab[j]=temp;
            j=m; k=m<<1;
        }
    }
}
```

Sortowanie kubełkowe - kod w C

```
void sortowanie_kubelkowe(int tab[],int n,int vmin,int vmax){
    int m=vmax-vmin+1;
    int *L=(int*)malloc(m*sizeof(int));
    if(L==0)
        printf("Pamiec pelna - ");
    else{
        memset(L,0,m*sizeof(int));//zerowanie pamięci
        int i=n,j=0;
        while(i>0)
            (L[tab[--i]-vmin])++;
        for(;i<m;i++)
            while(L[i]>0){
                tab[j++]= i+vmin;
                (L[i])--;
            }
    }
    free(L);
}
```

Wadą tego algorytmu jest zapotrzebowanie na pamięć, dla małych zakresów nie jest to problem, ale sortowanie to na pewno nie nadaje się do tablic o zakresie długości większej od 2^{29} ze względu na funkcję alokującą pamięć. Złożoność pamięciowa wynosi $\mathcal{O}(m)$, a dokładniej tworzymy tablicę o rozmiarze m.
Złożoność czasowa jest łatwa do policzenia - najpierw zerujemy tablicę liczników, co jest liniowo zależne od m. Następnie przebiegamy całą tablicę by zwiększać liczniki - robimy to w n krokach. Na samym końcu mamy dwie połączone pętle - zewnętrzna wykona się m razy, przy czym tylko dla n przypadków będziemy przypisywać wartość do tabeli.
Pamiętajmy, że powyższy algorytm działa tylko i wyłącznie dla liczb całkowitych (poprzednie algorytmy prosto przerabia się na liczby zmiennoprzecinkowe). Istnieją pewne modyfikacje, które pozwalają działać sortowaniu kubełkowemu również na liczbach zmiennoprzecinkowych (czy też pełnym zakresie **int**), ale jest to już bardziej skomplikowane choć pozwala na osiągniecie złożoności pamięciowej $\mathcal{O}(n)$ przy zachowaniu złożoności czasowej $\mathcal{O}(n)$. W rozwiązaniu tym kubełki są przedziałami (jest ich około n). Tworzymy również listę wszystkich wartości i wskaźników na najmniejszy element w kubełku (każda ma więc po n elementów). W sumie dla typu **int** potrzebujemy więc $3 * n*$**sizeof**(**int**) wolnej pamięci na stercie.

Sortowanie kubełkowe rozszerzone - kod w C I

```
typedef struct{// pomocnicza struktura
    int nastepnik;// indeks większego elementu
    int dane;// wartość
} lista;

void sortowanie_kubelkowe_ext(int *tab,int n,int vmin,int vmax){
    unsigned char k=0;
    long long zakres=vmax;zakres-=vmin;zakres++;// zakres zmiennych
    while(n<(zakres>>k)) k++;// szerokość kubełka to 2^k
    int *K=(int*)malloc(n*sizeof(int));
    lista *L=(lista*)malloc((n*sizeof(lista)));
    if(L==0 || K==0){
        printf("Pamiec pelna - ");
    }else{
        int i=0,j=0,ikb;
        memset(K,-1,n*sizeof(int));// kubełki puste
        for(;i<n;i++){// przypisujemy elementy do kubełków
            int we=tab[i];
            L[i].dane=we;
            ikb=(((unsigned)(we-vmin))>>k);// nr. kubełka
            int ip=-1;
            int ib=K[ikb];// ib = -1 lub indeks najmniejszego
```

Sortowanie kubełkowe rozszerzone - kod w C II

```
            while((ib>=0)&&(L[ib].dane<we)){
                ip = ib;
                ib = L[ib].nastepnik;
            }
            L[i].nastepnik=ib;
            if(ip==-1)
                K[ikb]=i;
            else
                L[ip].nastepnik=i;
        }
        for(ikb=0;ikb<n;ikb++){// sortowanie
            i=K[ikb];// i=-1 lub indeks najmniejszego elementu
            while(i>=0){
                tab[j++] = L[i].dane;
                i=L[i].nastepnik;// i=-1 lub kolejny element
            }
        }
    }
    free(L);free(K);
}
```

Podsumowanie sortowań

Można się pytać oczywiście jakie sortowanie mamy stosować znając rozmiar n tablicy i zakres m wartości z tablicy. Poniższa tabela została opracowana na podstawie wyników z mojego komputera i na innym sprzęcie może wyglądać inaczej (tabela typu **int**, dla **double** jest podobna, ale bez zwykłego sortowania kubełkowego).

n	m	wolna pamięć na stercie	sortowanie
$\leqslant 128$	-	-	wstawianie
> 128	$< \alpha_n n$	$\geqslant m$	kubełkowe
		$\in [3n, m)$	kubełkowe2
		$< 3n < m$	introspektywne
	$\geqslant \alpha_n n$	$\geqslant 3n$	kubełkowe2
		$< 3n$	introspektywne

Występujące tutaj α_n rośnie wraz z n od wartości 5 (poniżej 256tyś.) aż do 22 (powyżej 1mln). Wynika to z istnienia pamięci podręcznej procesora i jej wielkości jak i działania w tym samym czasie innych programów.

Sortowanie.cpp Sortowanie2.cpp

Funkcje matematyczne

Używając biblioteki **math.h** mamy dostęp do pewnych funkcji matematycznych. Większość z nich występuje dla wszystkich trzech typów liczb zmiennoprzecinkowych (**float**, **double**, **long double**), ale przedstawiając je tutaj będziemy w uproszczeniu mówić tylko o **double**.
Gdy w poniższych funkcjach występują kąty, to są one zawsze w radianach.

Funkcje trygonometryczne:

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);// x w [-1,+1]
double acos(double x);// x w [-1,+1]
double atan(double x);// wartość w [-pi/2,+pi/2]
double atan2(double y,double x);// atan(y/x), wartość w [-pi,+pi]
```

Funkcje hiperboliczne:

```
double sinh(double x);
double cosh(double x);// wartości w (1,+∞)
double tanh(double x);// wartości w (-1,1)
```

Funkcje wykładnicze i logarytmiczne:

```
double exp(double x);// e^x
double ldexp(double x,int exp);// zwraca x*2^exp
double frexp(double x,int* exp);// zwraca mantysę z [0,1) i wykładnik
                                // dla x=mantysa*2^exp
double log(double x);// ln x
double log10(double x);// log_10 x
```

Funkcje potęgowe:

```
double pow(double base,double exponent);// base^exponent
double sqrt(double x);// pierwiastek kwadratowy
```

Przybliżanie i reszty:

```
double floor(double x);// podłoga - cecha
double ceil(double x);// sufit
double fmod(double numer,double denom);// reszta z dzielenia obciętego
        // zwraca numer - trunc(numer/denom)*denom
```

wartość	(int)	floor	ceil
2.3	2.0	2.0	3.0
3.8	3.0	3.0	4.0
5.5	5.0	5.0	6.0
-2.3	-2.0	-3.0	-2.0
-3.8	-3.0	-4.0	-3.0
-5.5	-5.0	-6.0	-5.0

Inne:

```
double fabs(double x);// wartość bezwzględna
```

Stałe:

```
HUGE_VAL // maksymalna dodatnia wartość dla double
```

Jeżeli jest to nam potrzebne, to możemy sobie pomocniczo zdefiniować potrzebne nam ważne stałe matematyczne (można je otrzymać wykorzystując wcześniej omówione funkcje):

```
#define M_E   2.71828182845904523536
#define M_PI  3.14159265358979323846
```

Biblioteka complex.h

Biblioteka ta pozwala operować na liczbach zespolonych. Aby utworzyć taką liczbę zespoloną należy dodać do naszego typu (całkowitego lub rzeczywistego) dopisać typ zespolony (przed lub za tym typem). Funkcje **scanf** i **printf** niestety nie obsługują formatu zespolonego więc musimy sami zadbać o wczytanie i wyświetlanie liczb zespolonych, wykorzystując to, że liczba zespolona jest zapisywana jako tablica dwuelementowa.

```
double complex z;
complex int s=2+3*I;// I jest zdefiniowaną stałą w bibliotece
double * wsk=(double*)&z;
scanf("%lf %lf",wsk,wsk+1);
```

Wszystkie funkcje występujące w bibliotece mogą występować w trzech typach: operujące na **double** (domyślne), operujące na **float** (nazwa funkcji ma przyrostek "f") oraz operujące na **long double** (nazwa funkcji ma przyrostek "l").

Podstawowe operatory:

```
double creal(x);// zwraca część rzeczywistą
double cimag(x);// zwraca część zespoloną
double cabs(x);// zwraca moduł
double carg(x);// zwraca argument główny (w radianach)
double complex conj(x);// zwraca sprzężenie liczby zespolonej
```

Pozostałe funkcje zespolone:

```
double complex  cacos (x);
double complex  casin (x);
double complex  catan (x);
double complex  ccos (x);
double complex  csin (x);
double complex  ctan (x);
double complex  cacosh (x);
double complex  casinh (x);
double complex  catanh (x);
double complex  ccosh (x);
double complex  csinh (x);
double complex  ctanh (x);
double complex  cexp (x);
double complex  clog (x);
double complex  cpow (x, y);// x^y
double complex  csqrt (x);// cz. rzeczywista >=0
double complex  cproj (x);// rzutowanie na sferę Riemanna
```

Przykład

```
double complex x=-4;
double *wsk=(double*)&x;
printf("%f %+f i\n",creal(x),cimag(x));// -4.0 + 0.0 i
x=csqrt(x);
printf("%f %+f i\n",*wsk,*(wsk+1));// 0.0 + 2.0 i
```

Biblioteka ctype.h

W bibliotece tej mamy funkcję operującą na pojedynczych znakach.

Sprawdzanie rodzaju znaku:

```
int isdigit(int c);// czy znak jest cyfrą
int isxdigit(int c);// czy znak jest cyfrą w systemie szesnastkowym
int isalnum(int c);// czy znak jest cyfrą lub literą
int isalpha(int c);// czy znak jest literą
int islower(int c);// czy znak jest małą literą
int isupper(int c);// czy znak jest dużą literą
int isspace(int c);// czy znak jest białym znakiem
int isprint(int c);// czy znak jest znakiem drukowalnym
int isgraph(int c);// czy znak jest drukowalny i nie jest spacją
int iscntrl(int c);// czy znak jest znakiem kontrolnym
int ispunct(int c);// czy znak jest drukowalny ale nie alfanumerycznym
                   // i nie spacją
```

Wszystkie powyższe funkcje zwracają wartości logiczne ($0 =$ **false**).

Konwersja znaku:

```
int tolower(int c);// zmienia literę na małą
int toupper(int c);// zmienia literę na dużą
```

Funkcje te zwracają przekształconą literę.

Przykład

```
#include <ctype.h>
#include <stdbool.h>
int main(void){
    char znak;
    bool bznakprzed=true;
    puts("Podaj zdanie: ");
    do{
        znak=getchar();
        if(znak==EOF) break;
        if(isspace(znak)){
            if(!bznakprzed)
                putchar(' ');
            bznakprzed=true;
        } else if(isalpha(znak)){
            if(bznakprzed)
                znak=toupper(znak);
            putchar(znak);
            bznakprzed=false;
        } else if(!iscntrl(znak)){
            bznakprzed=false;
            putchar(znak);
        }
    }while(znak!='\n');
    return 0;
}
```

Biblioteka stdlib.h

Wcześniej wspominaliśmy o pewnych funkcjach dostępnych w bibliotece standardowej, teraz co nieco dopowiemy o innych funkcjach tam dostępnych.

Konwersja tekstu na liczbę:

```
double strtod (const char* str, char** endptr);
```

zwraca liczbę typu double zapisaną w łańcuchu znaków **str**, w **endptr** (o ile nie jest zerowy) zostaje zapisany wskaźnik na pierwszy znak z łańcucha, który nie był związany z konwertowaną liczbą. Łańcuch powinien zawierać cyfry (przed nimi może być znak), między którymi może wystąpić jedna kropka a także może zawierać litery "e" lub "E" oznaczające wykładnik (który również może zawierać znak). Ewentualnie liczba ta może być zapisana szesnastkowo z przedrostkiem "0x" lub "0X" (z kropką i wykładnikiem jak powyżej). W przypadku błędu konwersji funkcja zwraca 0, a w przypadku przekroczenia zakresu najbliższą + lub - maksymalną wartość dla **double** (**HUGE_VAL**).

```
float strtof (const char* str, char** endptr);
long double strtold (const char* str, char** endptr);
```

podobnie jak wcześniej tylko zwracają liczbę typu **float** lub **long double**.

```
long int strtol(const char* str, char** endptr, int base);
unsigned long int strtoul(const char* str, char** endptr, int base);
long long int strtoll(const char* str, char** endptr, int base);
unsigned long long int strtoull(const char* str, char** endptr, int
    base);
```

przekształcają liczbę zapisaną w tekście na odpowiedni typ całkowity (l - **long**, ll - **long long**, u - **unsigned**), **base** oznacza bazę systemu (od 2 do 26), w której zapisana jest liczba, przy czym wartość 0 oznacza, że przedrostek będzie mówił o tym jaki jest system (ósemkowy lub szesnastkowy). Liczba może zwierać cyfry lub litery (w zależności od systemu) a także prefiks "0x" lub "0X" w systemie szesnastkowym albo "0" dla ósemkowego. Funkcje te zwracają 0 w przypadku błędu lub najbliższą maksymalną/minimalną możliwą wartość danego typu, gdy przekroczony był zakres.
Wszystkie wymienione wcześniej funkcje usuwają z wejścia spacje poprzedzające zapisaną tekstowo liczbę.

Przykład

```
char szOrbity[] = "365.24 29.53";
char* pKoniec;
double d1, d2;
d1 = strtod(szOrbity, &pKoniec);
d2 = strtod(pEnd, NULL);
printf("Księżyc w ciągu roku okrąża Ziemię %.2f razy.\n", d1/d2);
```

Przykład

```
char szLiczby[] = "2001 60c0c0 -11011101001101001000000 0x6fffff";
char* pKoniec;
long int li1, li2, li3, li4;
li1 = strtol(szLiczby,&pKoniec,10);
li2 = strtol(pKoniec,&pKoniec,16);
li3 = strtol(pKoniec,&pKoniec,2);
li4 = strtol(pKoniec,NULL,0);
```

Oprócz wymienionych wcześniej funkcji konwertujących łańcuch na tekst mamy również pewne ich uproszczenia:

```
double atof(const char* str);
```

dla liczb rzeczywistych oraz

```
int atoi(const char* str);
long int atol(const char* str);
long long int atoll(const char* str);
```

dla liczb całkowitych o podanym typie.
Niestety w przypadku nieprawidłowych znaków wartość zwracana jest trudna do przewidzenia, więc lepiej je stosować gdy mamy pewność, że łańcuch znaków jest poprawny.
Żeby przekonwertować liczbę na ciąg znaków możemy wykorzystać **sprintf**:

```
char szLiczba[50];
float liczba = 23.34;
sprintf(szLiczba, "%f", liczba);
printf("Liczba na tekst: %s", szLiczba);
```

W bibliotece **stdlib.h** mamy jeszcze dwie ważne funkcje:

```
void qsort(void* tablica, size_t rozm_tab, size_t rozm_el,
           int (*f_por)(const void*,const void*));
```

pozwala na posortowanie (zmodyfikowany algorytm quicksort) tablicy elementów dowolnego typu. Należy podać wskaźnik na tablicę (**tablica**), rozmiar tablicy (**rozm_tab**) oraz rozmiar pojedynczego elementu (**rozm_el**). Ponadto musimy podać wskaźnik na funkcję (**f_por**), która porównuje dwa elementy naszej tablicy i zwraca 0 gdy są równe, wartość <0 gdy pierwszy parametr jest mniejszy niż drugi lub wartość >0 gdy pierwszy parametr jest większy niż drugi.
Druga funkcja, to:

```
void* bsearch (const void* klucz, const void* tablica,
               size_t rozm_tab, size_t rozm_el,
               int (*f_por)(const void*,const void*));
```

która wyszukuje binarnie w posortowanej tablicy dowolnego typu podanego przez nas elementu (**klucz**). Również tutaj wymagana jest funkcja porównująca dwa elementy z tablicy. Funkcja ta zwraca element pasujący do wyszukiwania lub 0 gdy nie udało się go odnaleźć.

Przykład

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char tablica[3][16] = {"Kowalski", "Nowak", "Chrobry"};
int f_por(const char* a, const char* b){
    return strcmp(a,b);
}

int main(void){
    char nazwisko[16];
    char* wsk;
    qsort(tablica,sizeof(tablica)/sizeof(tablica[0]),
          sizeof(tablica[0]),f_por);
    for(int i=0;i<3;i++)
        printf("%s\n",tablica[i]);
    printf("Podaj nazwisko do wyszukania: ");
    scanf("%16s",nazwisko);
    wsk= bsearch(nazwisko,tablica,sizeof(tablica)/sizeof(tablica[0]),
                 sizeof(tablica[0]),f_por);
    if(wsk!=0)
        printf("Znaleziono element %s\n",wsk);
    else
        printf("Nie znaleziono.\n");
    return 0;
}
```

W bibliotece **stdlib.h** jak i w innych (**stddef.h stdio.h string.h**, **time.h**, **wchar.h**) można napotkać typ całkowity bez znaku **size_t**. Jest on używany w różnych funkcjach (np. związanych z alokacją pamięci czy odczytem/zapisem do pliku) do przechowywania rozmiarów czy liczników. W zależności od architektury może on zajmować od 2 do 8 B. Jego maksymalną wartość można oczytać ze stałej **SIZE_MAX**. Wyświetlanie lub wczytywanie tego typu za pomocą **printf** i **scanf** jest możliwe dzięki "z" występującym przed jednym ze specyfikatorów **d, i, u, o, x, X** wyświetlających liczby całkowite.

Biblioteka stdint.h

Może się zdarzyć, że chcemy kompilować nasz kod na różnych architekturach (np. 32 lub 64 bitowych). Niektóre typy całkowite mają zmienną wielkość zależną właśnie od architektury. Aby mieć pewność, że nasze typy danych mają dokładnie tyle bitów ile chcemy możemy użyć biblioteki **stdint.h**. Znajdziemy w niej następujące typy całkowite:

ze znakiem	bez znaku	opis
int8_t	**uint8_t**	8 bitów
int16_t	**uint16_t**	16 bitów
int32_t	**uint32_t**	32 bity
int64_t	**uint64_t**	64 bity
intmax_t	**uintmax_t**	maksymalna dostępna ($\geqslant$ 64 bity)

W niektórych dziwnych konfiguracjach sprzętowych typy o ustalonej ilości bitów mogą nie istnieć. Z tego względu mamy do dyspozycji typy, które zamiast **int** zawierają **int_least** lub **int_fast**. Mają one co najmniej tyle bitów ile jest podanych, przy czym wersja "fast" ma wielkość dopasowaną do jak najszybszych działań na tych typach, a wersja "least" jak najmniejszy rozmiar.

Biblioteka inttypes.h

Wyświetlanie czy pobieranie typów z **stdint.h** może powodować czasem pewne problemy. Z tego względu w bibliotece **inttypes.h** mamy zdefiniowane pewne wartości, które służą do obsługi tych typów całkowitych w funkcjach **printf** i **scanf**:

przedrostek	typ	przyrostek	bity
PRI	d, i	brak	8, 16, 32, 64
	u	LEAST	
SCN	o	FAST	
	x	MAX	brak

Przykład

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
intmax_t duza;
uint32_t liczba;
scanf("%" SCNdMAX,&duza);
printf("duza = %" PRIdMAX ", liczba 32 bit = %" PRIu32, duza, liczba);
```

Operacje na plikach

Jeżeli chcemy wykonywać operacje na plikach, to możemy wykorzystać bibliotekę **stdio.h**. Do obsługi pliku potrzebny jest nam uchwyt (czyli wskaźnik) do pliku

```
FILE * nazwa_uchwytu;
```

Domyślnie mamy zdefiniowane 3 takie uchwyty: **stdin** - oznaczające standardowe wejście, czyli klawiaturę; **stdout** - oznaczające standardowe wyjście, czyli konsolę; **stderr** - standardowe wyjście błędów, które domyślnie też jest powiązane z wyświetlaniem w konsoli.
Zanim przejdziemy do otwierania plików, to powiedzmy sobie najpierw o innych możliwych operacjach. Plik możemy usunąć lub zmienić jego nazwę za pomocą funkcji:

```
int remove(const char* nazwa_pliku);
int rename(const char* stara_nazwa, const char* nowa_nazwa);
```

Obie funkcje zwracają 0, gdy operacja się powiedzie.

Aby otworzyć plik używamy funkcji

```
FILE* fopen(const char *nazwa_pliku, const char *tryb);
```

przy czym tryb mówi w jaki sposób ma być otwarty plik i może być równy:

"r"	odczyt z pliku, plik musi istnieć
"w"	zapis do pliku (nadpisywanie)
"a"	dopisywanie do pliku (na końcu)

Za powyższymi znakami mogą występować również (mogą być wymieszane w dowolnej kolejności)

"+"	oznacza, że plik jest otwarty do aktualizacji (odczyt i zapis)
"b"	oznacza tryb binarny otwarcia, czyli nietekstowy
"x"	od C11 tylko razem z "w", oznacza, że plik musi istnieć

W trybie dopisywania (czyli zawierającym "a") nie działają funkcje, które zmieniają aktualną pozycję w pliku. Jeżeli uda się otworzyć plik, to zwracany wskaźnik jest niezerowy.
Każdy otwarty plik należy zamknąć, gdy już nie będziemy z niego korzystać. Aby to zrobić należy wywołać funkcję:

```
void fclose(FILE* nazwa_pliku);
```

Jeśli tego nie zrobimy, to system operacyjny będzie uważał plik za otwarty co może uniemożliwić innym programom jego modyfikację.
Zamknięty uchwyt do pliku może być ponownie wykorzystany do otwarcia innego pliku.
Standardowe strumienie **stdin**, **stdout**, **stderr** nie zamykamy (o ile chcemy ich jeszcze użyć), chyba, że wcześniej zmieniliśmy je, tak by wskazywały na pliki.
Taką zmianę możemy zrobić za pomocą

```
FILE*freopen(const char* nazwa_pliku,const char* tryb,FILE *strumien);
```

gdzie tryb przyjmuje wartości jak dla **fopen**.

Przykład

```
#include <stdio.h>
int main (){
  freopen ("wyjscie.txt","w",stdout);
  printf("Zapisujemy do pliku a nie wypisujemy w konsoli.");
  fclose(stdout);
  printf("Teraz już wypisujemy w konsoli.");// nic się nie wyświetli
  return 0;
}
```

W trybie tekstowym do pliku możemy zapisywać za pomocą funkcji podobnych do znanych **printf** i **scanf**:

f		zapis do pliku
s		zapis do łańcucha znaków
v	printf	wyświetlenie zmiennej ilości danych
vf		zapis zmiennej ilości danych do pliku
vs		zapis zmiennej ilości danych do łańcucha znaków

Analogiczne funkcję występują dla **scanf**. Mają one składnię:

```
int fprintf(FILE* wsk_do_pliku, const char* format, ... );
int fscanf(FILE* wsk_do_pliku, const char* format, ... );
int vfprintf(FILE* wsk_do_pliku, const char* format, va_list arg );
int vfscanf(FILE *wsk_do_pliku, const char* format, va_list arg );
int sprintf(char* lancuch, const char* format, ... );
int sscanf(const char* lancuch, const char* format, ...);
int vsprintf(char* lancuch, const char* format, va_list arg );
int vsscanf(const char* lancuch, const char* format, va_list arg );
int vprintf(const char* format, va_list arg );
int vscanf(const char* format, va_list arg );
```

Funkcje wypisujące/zapisujące zwracają ilość znaków pomyślnie wypisanych/zapisanych. Funkcję wczytujące zwracają ilość zmiennych pomyślnie wypełnionych.

Przykład

```c
#include <stdio.h>
int main(){
    int len=0;
    char bufor[256];
    FILE *plik;
    plik=fopen("wyjscie.txt","r+");
    if(plik){
        printf("Otwarto plik. Pierwsze 26 znaki beda zmienione:\n");
        fscanf(plik,"%[26]s",bufor);
        bufor[26]=0;
        while(len<26){
            bufor[len]='A'+len;
            len++;
        }
        len=fprintf(plik,"%s",bufor);
        printf("Zapisano znakow %d\n",len);
    }else{
        printf("Nie udalo sie otworzyc pliku.\n");
    }
    fclose(plik);
    return 0;
}
```