Kurs C

Radosław Łukasik

Wykład 4

Znaki odczytywać lub zapisywać możemy również za pomocą innych funkcji:

```
int fgetc(FILE* wsk_do_pliku);// odczyt znaku
char* fgets(char *lancuch, int ilosc, FILE* wsk_do_pliku);// odczyt
                                                          // łańcucha
int fputc(int znak, FILE* wsk_do_pliku);// zapis znaku
int fputs(const char* lancuch, FILE* wsk_do_pliku);// zapis łańcucha
```

Funkcja **fgetc** zwraca aktualny znak, przesuwa pozycję w pliku o 1. W przypadku gdy nie można odczytać znaku (np. koniec pliku) zwraca wartość **EOF**, przy czym dla końca pliku ustawiony jest odpowiedni znacznik, który można zbadać za pomocą funkcji

```
int feof(FILE* wsk_do_pliku);// zwraca wartość !=0 gdy osiągnięto
                             // koniec pliku
```

lub w przypadku błędów odczytu (dla zapisu też działa ta funkcja)

```
int ferror(FILE* wsk_do_pliku);// zwraca wartość !=0 gdy
                               // wystąpił błąd
```

Czasem się zdarza, że pomimo wystąpienia błędu chcemy dalej kontynuować pewne operacje na pliku (np. wystąpił błąd zapisu, ale chcemy coś z pliku jeszcze odczytać). Wówczas musimy wyzerować znacznik błędu za pomocą

```
void clearerr(FILE* wsk_do_pliku);
```

Funkcja **fgets** odczytuje co najwyżej **ilosc -1** znaków, chyba, że wcześniej napotka znak końca linii (który nie jest odczytywany) lub dotrze do końca pliku. Zwraca wskaźnik na łańcuch znaków do którego zapisano znaki lub 0 gdy wystąpi błąd odczytu (włączając w to dojście do końca pliku).
Funkcja **fputc** zapisuje jeden znak do pliku. Zwraca zapisany znak lub w przypadku błędu **EOF** i ustawia znacznik błędu.
Funkcja **fputs** zapisuje ciąg znaków zakończony zerem (nie zapisuje znaku '\0'). Zwraca wartość niezerową w przypadku powodzenia lub 0 gdy pojawi się błąd (oraz ustawia znacznik błędu).
Jeżeli chcemy wiedzieć jaki dokładnie wystąpił błąd podczas otwierania pliku, odczytu lub zapisu do niego, to możemy to uzyskać korzystając z zdefiniowanego strumienia **stderror** oraz funkcji:

```
void perror(const char* tekst);
```

która wyświetla komunikat błędu poprzedzany podanym przez nas tekstem (może on być wskaźnikiem zerowym).

Przykład

```c
#include <stdio.h>
int main(){
    char znak;
    FILE *plik;
    plik=fopen("wyjscie.txt","r");
    if(plik){
        while((znak=fgetc(plik))!=EOF)
            putchar(znak);
        if(feof(plik))
            printf("\nOdczytano caly plik.\n");
        if(ferror(plik))
            printf("\nWystąpil blad odczytu.\n");
    }else
        perror("Nie udalo sie otworzyc pliku: ");
    fclose(plik);
    return 0;
}
```

Jeżeli chcemy odczytać lub zapisać coś do pliku binarnego, to możemy użyć

```
size_t fread(void* bufor, size_t rozmiar, size_t ilosc,
             FILE* wsk_do_pliku);
```

aby zapisać do bufora (tablicy) podaną **ilość** elementów, przy czym każdy element ma podany przez nas **rozmiar**. Podobnie dla zapisu mamy

```
size_t fwrite(const void* bufor, size_t rozmiar, size_t ilosct,
              FILE* wsk_do_pliku);
```

Funkcje te zwracają ilość prawidłowo odczytanych/zapisanych elementów. Jeżeli jest to ilość różna od podanej przez nas, to zostaje ustawiony znacznik błędu.

Przykład

```
#include <stdio.h>
int tablica[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int main(){
    FILE *plik; plik=fopen("dane.dat","wb");
    if(plik){
        if(fwrite(tablica,sizeof(int),16,plik)!=16)
            perror("Wystąpil blad zapisu: ");
    }else
        perror("Nie udalo sie otworzyc pliku: ");
    fclose(plik);
    return 0;
}
```

Wcześniejsze odczyty lub zapisy były zawsze od momentu w którym ostatnio skończyliśmy. Nie zawsze musi tak być. Możemy odczytywać lub przesuwać aktualną pozycję w pliku za pomocą:

```
int fseek(FILE* wsk_do_pliku, long int przesuniecie, int p_wyjsciowy);
```

gdzie przesuniecie jest liczbą całkowitą i oznacza pozycję względem punktu wyjściowego, który może być równy:

SEEK_SET	początek pliku
SEEK_CUR	aktualna pozycja w pliku
SEEK_END	koniec pliku

Funkcja ta zwraca 0 w przypadku powodzenia. W przypadku błędu zostaje ustawiony znacznik błędu. Może ona nie działać w plikach otwartych jako tekstowe.
Aby odczytać aktualną pozycję możemy użyć funkcji (może nie działać w plikach tekstowych):

```
long int ftell(FILE* wsk_do_pliku);
```

Mamy również funkcję, która pozwala nam skoczyć zawsze na początek pliku

```
void rewind(FILE* wsk_do_pliku);
```

Pomocne jest to szczególnie dla plików otwartych jednocześnie do odczytu i zapisu do przełączania się pomiędzy zapisem i odczytem.

Przykład

```c
#include <stdio.h>
char napis[]="\nKoniec pliku";
int main(){
    int len;
    FILE *plik;
    plik=fopen("wyjscie.txt","rb+");
    if(plik){
        fseek(plik,0,SEEK_END);
        len=ftell(plik);
        printf("Otwarto plik. Rozmiar: %d.\n",len);
        fwrite(napis,1,sizeof(napis)-1,plik);
        rewind(plik);
        fread(napis,1,sizeof(napis)-1,plik);
        printf("%s",napis);
    }else{
        perror("Blad: ");
    }
    fclose(plik);
    return 0;
}
```

W plikach tekstowych funkcje **fseek** i **ftell** mogą zwracać nieprawidłowe wartości z tego względu, że znaki mogą zajmować więcej niż jeden bajt w zależności od kodowania. Dla plików tekstowych mamy więc specjalne funkcje służące do odczytu i zapisu bieżącej pozycji w pliku:

```
int fgetpos(FILE* wsk_do_pliku, fpos_t* pos);// odczyt pozycji
int fsetpos(FILE* wsk_do_pliku, const fpos_t* pos );// ustawienie
    pozycji
```

Funkcje te zwracają 0 w przypadku powodzenia. Musimy w nich podać wskaźnik typu **fpos_t** skąd zostanie pobrana lub gdzie zostanie zapisana aktualna pozycja. Oczywiście zawartość tej zmiennej nie może nam powiedzieć gdzie dokładnie jesteśmy w pliku, ale możemy zapamiętywać i odtwarzać tą pozycję.

Przykład

```c
#include <stdio.h>
int main(){
    char znak;
    fpos_t pos;
    FILE *plik;
    plik=fopen("wyjscie.txt","r");
    if(plik){
        fgetc(plik);
        fgetpos(plik,&pos);
        znak=fgetc(plik);
        printf("2 znak: %c.\n",znak);
        fgetc(plik);
        fsetpos(plik,&pos);
        znak=fgetc(plik);
        printf("2 znak: %c.\n",znak);
    }else
        perror("Blad: ");
    fclose(plik);
    return 0;
}
```

Mamy również możliwość tworzenia plików tymczasowych. Pliki takie mają tworzoną pewną "losową" nazwę, tak by nie powtarzała się z nazwami innych plików oraz po zamknięciu są automatycznie usuwane. Zastosowanie tych plików to chwilowe zastąpienie pamięci RAM, gdy pośrednie wyniki zajmują więcej miejsca niż jest dostępne na stercie.
Aby utworzyć plik tymczasowy należy użyć funkcji:

```
FILE* tmpfile(void);
```

która tworzy i otwiera plik tymczasowy, a także zwraca wskaźnik do niego (zerowy wskaźnik oznacza niepowodzenie utworzenia pliku tymczasowego). Plik jest otwarty w trybie "wb+" i żeby go usunąć wystarczy go zamknąć funkcją **fclose**.
Możemy również tworzyć tymczasowe nazwy plików (nie będą się one powtarzać z istniejącymi już plikami):

```
char * tmpnam(char* str);
```

W funkcji tej podajemy wskaźnik na łańcuch znaków o długości takiej jak stała **L_tmpnam**. Jeżeli podamy wskaźnik zerowy, to zostaniem nam zwrócony wskaźnik na pewien bufor z nazwą tymczasową pliku (zawartość tego bufora może się zmieniać). Jeżeli podaliśmy wskaźnik na łańcuch znaków, to zostanie zwrócony ten wskaźnik, ewentualnie 0 w przypadku niepowodzenia.
Liczba gwarantowanych unikalnych tymczasowych nazw jest określona za pomocą stałej **TMP_MAX**.

Typy złożone

Zdarza się, że zamiast wartości zmiennych wolimy używać pewnych nazw z nimi związanych. Do tworzenia typu wyliczeniowego - czyli tablic z nazwami stałych typu **unsigned int** - służy polecenie **enum**:

```
enum nazwa_typu_wyliczenia{
    nazwa_wyliczenia0 = 7,     // wyliczenia oddzielamy przecinkiem
    nazwa_wyliczenia1 = 3,
    :
    nazwa_wyliczeniaN = 5     // po ostatnim wyliczeniu bez przecinka
};
```

Jeżeli przy pierwszym wyliczeniu nie ma przypisanej wartości, to przypisane zostaje do niej 0. Jeżeli przy którejś niżej ma miejsce brak przypisania, to przypisana zostaje wartość wcześniejszego wyliczenia powiększona o 1.
By przypisać zmiennej całkowitej wartość z naszego typu wyliczeniowego należy użyć kodu

```
zmienna=nazwa_wyliczenia;
```

Typ wyliczeniowy może służyć do definiowania zmiennych, przy czym zmienna ta może przyjmować nie tylko wartości będące wyliczeniami (w C++ jest inaczej).

Przykład

```
enum color{
    red =4,
    blue,     // blue =5
    white =8
};
:
int n=red; // przypisujemy do n wartość red, czyli 4
enum color zmienna;     // tworzymy zmienną typu color
zmienna = blue;    // zmienna = 5
```

Czasem zdarza się, że mamy zmienną składającą się z długiego słowa bądź z kilku słów. Możemy wówczas wprowadzić dla niej swoją krótszą nazwę używając polecenia **typedef**:

```
typedef typ_zmiennej nazwa;
```

Przykład

```
typedef unsigned int uint;
:
uint x=5;
```

Struktury

Czasem chcemy opisać jakieś rzeczy kilkoma parametrami. Przydałby się nam więc pewien obiekt w którym możemy mieć kilka zmiennych i to nawet różnych typów. To tego służy właśnie **struct**, przy czym mamy kilka możliwości:

```
typedef struct dluga_nazwa_struktury{// definiujemy nowy typ
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    :
    typ nazwa_elementuN;
}krotka_nazwa;
struct dluga_nazwa_struktury nazwa_zmiennej;
krotka_nazwa nazwa_zmiennej;
```

przy czym długa nazwa może być pominięta jeśli chcemy używać tylko krótkiej nazwy.

```
struct nazwa_struktury{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    :
    typ nazwa_elementuN;
};
struct nazwa_struktury nazwa_zmiennej;
```

```
struct nazwa_struktury{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    typ nazwa_elementuN;
}nazwa zmiennej;
```

Żeby odwoływać się do składowych struktury używamy kropki między nazwą zmiennej a nazwą elementu

```
nazwa_struktury zmienna; //tworzymy nową zmienną
zmienna.nazwa_elementu=wartosc;
```

Możemy również zainicjować wartości początkowe naraz używając

```
nazwa_struktury zmienna={wartosc1,wartosc2,...,wartoscN};
```

gdzie kolejne wartości odpowiadają kolejnym elementom struktury.
Od C99 można inicjować również wybrane pola struktury oddzielone przecinkami.
Niewymienione pola są wypełnione zerami.

```
nazwa_struktury zmienna={.nazwa_elementu=wartosc};
```

Jeśli mamy wskaźnik na strukturę, to oprócz domyślnego dla wskaźników odwoływania się do elementów możemy również odwoływać się używając operatora wskazywania ->

```
nazwa_struktury *wskaznik;
(*wskaznik).nazwa_elementu=wartosc; // domyślny sposób
wskaznik->nazwa_elementu=wartosc;   // też działa
```

Mamy również inną przydatną rzecz - możemy skopiować zawartość jednej struktury do innej (tego samego typu) używając po prostu =

```
nazwa_struktury zmienna1,zmienna2;
:
:
zmienna2=zmienna1;     // kopiujemy wszystkie elementy
```

Należy tutaj pamiętać, że jeśli elementem struktury jest wskaźnik, to zostaje skopiowany wskaźniki i nie jest tworzona kopia zmiennej na którą on wskazuje, przez co należy przy zwalnianiu zadbać o wyzerowanie również wskaźnika ze skopiowanej struktury (lub utworzeniu kopii elementu wskazywanego).
Jeżeli chcemy wiedzieć ile pamięci zajmuje struktura, to możemy posłużyć się

```
sizeof(nazwa_struktury)
```

Gdy mamy tablicę złożoną ze struktur, to użycie inkrementacji lub dekrementacji na wskaźniku na element tej tablicy powoduje odpowiednio dodanie lub odjęcie od wskaźnika rozmiaru struktury. Dzięki temu możemy przesunąć się w tablicy na następny lub poprzedni jej element.

Struktura może nie mieć nazwy, o ile jest elementem innej struktury (unii lub klasy). Dzięki temu możemy odwoływać się do jej elementów jakby to były elementy jej struktury nadrzędnej (odpowiedni przykład będzie przy uniach). Jeżeli mielibyśmy dwie struktury, i każda z nich odwołuje się wewnątrz do tej drugiej, to należy wcześniej zadeklarować przynajmniej jedną z nich (tą późniejszą w kodzie), bo bez tego otrzymamy błąd kompilacji. Aby to zrobić wystarczy po prostu napisać

```
struct nazwa_dalszej_struktury;
```

W pierwszej strukturze musimy za to użyć tej nazwy struktury poprzedzonej słowem **struct**. Podobnie ma to miejsce gdy chcemy użyć wskaźnika na strukturę wewnątrz jej samej. Nie trzeba jej deklarować wcześniej, ale trzeba użyć **struct** z długą nazwą naszej struktury.
I na koniec najważniejsze - struktury mogą być zwracane poprzez funkcję. Możemy używać również struktur do przypisywania pewnych ustalonych wartości czy też zwracania pewnej ustalonej struktury poprzez wymienienie jej pól.

Przykład

```
typedef struct kolejka{// długa nazwa jest nam potrzebna
    char imie[32];
    char nazwisko[32];
    struct kolejka *nastepny;// bo się do niej tu odwołamy
}Kolejka;
```

Przykład

```
struct samochod;// bez tego błąd kompilacji
typedef struct {// definicja kierowcy
    char imie[32];
    char nazwisko[32];
    struct samochod *woz;
}kierowca;
typedef struct samochod{// definicja samochodu
    char marka[32];
    char model[32];
    kierowca *szofer;
}samochod;
```

Przykład

```
typedef struct {
    double x, y;
} punkt_t;

punkt_t suma(punkt_t A, punkt_t B){
    return (punkt_t){A.x+B.x, A.y+B.y};
}
```

Przykład

```
typedef struct {
    char imie[32];
    char nazwisko[32];
    short wiek;
    short wzrost;
}osoba;
:
:
osoba *radek=(osoba*)malloc(sizeof(osoba));
osoba klon={"Bezimienny","Klon",0,0};// inicjacja bezpośrednia
printf("Podaj imie i nazwisko: ");
scanf("%s %s",radek->imie,radek->nazwisko);
printf("Podaj wiek i wzrost: ");
scanf("%hd %hd",&(radek->wiek),&(radek->wzrost));
klon=*radek;
printf("%s %s ma ",klon.imie,klon.nazwisko);
printf("%hd lat i %hd cm wzrostu\n",klon.wiek,klon.wzrost);
free(radek);
```

Pola bitowe

Tworząc struktury możemy sprawić by zmienne całkowite zawierały tylko liczbę bitów, którą im ustalimy. Może to być przydatne np. w celu optymalizacji pamięci lub pewnych operacji na bitach, ale ma też swoje wady, bo działania wykonywane w ten sposób mogą być wolniejsze.
Pole bitowe tworzymy dodając za nazwą zmiennej dwukropek i liczbę bitów. Należy pamiętać, że typ liczby musi być wyłącznie całkowity (od **char** do **long long**) ze znakiem lub bez. Ten typ decyduje na starcie ile miejsca jest rezerwowanego dla tego pola bitowego i kolejnych pól bitowych tego samego typu (jeśli występują). Jeżeli kolejne pole bitowe przekroczyłoby wstępnie zarezerwowany rozmiar, to zostaje one przeniesione na początek nowej zmiennej tego typu. Jeżeli chcemy wymusić wyrównanie bitów do rozmiaru zmiennej podanego typu, to wystarczy utworzyć pole bitowe bez nazwy zmiennej i z :0 po nazwie typu.
Wyświetlanie bezpośrednie pól bitowych działa natomiast nie da się do nich wczytać bezpośrednio danych od użytkownika - trzeba to zrobić za pomocą pomocniczych zmiennych.

Przykład

```
typedef struct {// rok liczony od 1900
    unsigned short dzien:5;
    unsigned short miesiac:4; // 5+4=9<16
    unsigned short rok:7;     // 9+7=16
} pola_bitowe;// ta struktura zajmuje dwa bajty
pola_bitowe x;
x.rok=120;
x.dzien=15;
x.miesiac=8;
// scanf(" %hd",&(x.rok)); powoduje błąd
short temp;
scanf(" %hd",temp);
x.rok=temp;
printf("Rok:  %hd\n",x.rok);
printf("Miesiac:  %hd\n",x.miesiac);
printf("Dzien: %hd\n",x.dzien);
```

```
struct {
    char dzien:5;
    char miesiac:4;// 5+4=9>8
    char rok:7;// 4+7=11>8
} pola_bitowe;// ta struktura zajmuje trzy bajty
```

Unie

Unie są obiektami podobnymi do struktur ale z jedną zasadniczą różnicą: jej elementy nie są osobnymi zmiennymi tylko nachodzą na siebie (zaczynają się w tym samym miejscu w pamięci). Dzięki temu modyfikując jeden element zmieniamy jednocześnie wszystkie. Jest to pomocne gdy zmienna, którą chcemy przechować może przyjmować różne typy. Rozmiar unii jest maksymalnym rozmiarem jej elementów.

```
union nazwa_unii{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    typ nazwa_elementuN;
};
```

```
typedef union dluga_nazwa_unii{
    typ nazwa_elementu1;
    typ nazwa_elementu2;
    :
    typ nazwa_elementuN;
}krotka_nazwa_unii;
```

Przykład

```
typedef union {
    unsigned DWORD;
    struct{// struktura bez nazwy
        short LOWORD;
        short HIWORD;
    };
} liczba;
liczba x;
x.DWORD=0x12345678;
printf("LICZBA: %#x\n",x.DWORD);
printf("HIWORD: %#x\n",x.HIWORD);
printf("LOWORD: %#x\n",x.LOWORD);
```

Biblioteka time.h

W bibliotece **time.h** mamy dostępne funkcje związane z pomiarem czasu i datą. Potrzebne nam będą następujące typy i struktury:

- **time_t** - służy do przechowywania czasu w sekundach, przeważnie liczonego od godziny 00:00 UTC, 1 stycznia 1970 roku;
- **clock_t** (=**long**) - przechowuje licznik cykli zegara, ilość cykli tego zegara na sekundę jest przechowywana w stałej **CLOCKS_PER_SEC**. Wynik może być zależny od systemu operacyjnego, w windows 7 zegar w przybliżeniu przyjmuje wielokrotności 16ms mimo, że ta stała wynosi 1000;
- struktura **tm** - służy do odczytywania daty i aktualnego czasu, ma następującą postać:

```
struct tm{
    int tm_sec;     // liczba sekund 0-59* (ewentualnie 61)
    int tm_min;     // liczba minut 0-59
    int tm_hour;    // liczba godzin 0-23
    int tm_mday;    // dzień miesiąca 1-31
    int tm_mon;     // miesiąc (od stycznia) 0-11
    int tm_year;    // rok od 1900
    int tm_wday;    // dzień tygodnia (0-niedziela ... 6-sobota)
    int tm_yday;    // dzień roku 0-364 lub 365
    int tm_isdst;   // >0 czas letni
};
```

W tej bibliotece mamy dostępne następujące funkcje:

```
time_t time(time_t * czas);
```

zwraca czas lokalny, jeżeli ***czas** jest różne od zera, to zapisuje tą wartość również do zmiennej **czas**.

```
double difftime(time_t czas2, time_t czas1);
```

zwraca różnicę w sekundach między **drugim czasem** a **pierwszym czasem**.

```
tm * gmtime(const time_t *czas);
```

zwraca wskaźnik na strukturę **tm**, której dane wyrażone są w czasie UTC.

```
tm * localtime(const time_t *czas);
```

zwraca wskaźnik na strukturę **tm**, której dane wyrażone są w czasie lokalnym.

```
time_t mktime(tm *dane_czasu);
```

przekształca lokalne **dane czasu** na czas typu **time_t**. Zwraca -1 gdy nie można przekształcić z powodu niepoprawnych danych. Ignoruje zawartość **tm_wday** i **tm_yday**.

```
clock_t clock();
```

zwraca liczbę cykli, które upłynęły od uruchomienia programu. Przydatna przy mierzeniu wydajności algorytmów.

Przykład

```
char dzien[7][16]={"niedziela","poniedzialek","wtorek","sroda",
                   "czwartek","piatek","sobota"};
struct tm * aCzas;
time_t czas=time(0);     // pobieramy czas
aCzas=localtime(&czas); // konwertujemy na czas lokalny
printf("Aktualny czas %d:%d:%d\n",(*aCzas).tm_hour,(*aCzas).tm_min,
       (*aCzas).tm_sec);
printf("Aktualna data %d/%d/%d\n",aCzas->tm_mday,aCzas->tm_mon,
       1900+aCzas->tm_year);
printf("%s, %d dzien roku\n", dzien[aCzas->tm_wday],aCzas->tm_yday);
```

Przykład

```
clock_t zegar1,zegar2;
zegar1=clock();
// kod algorytmu, jeśli zbyt szybki, to należy powtórzyć go pętlą
// tak, by dało się zmierzyć jego czas trwania
:
:
zegar2=clock();
printf("Czas trwania %fs\n",(double)(zegar2-zegar1)/CLOCKS_PER_SEC);
```

Jeżeli chcemy dokładniej odmierzać czas (lub cykle procesora), to należy użyć funkcji **___rdtsc** lub **___rdtscp** związanych z odczytem TSC (Time Stamp Counter), który przechowuje ilość instrukcji wykonanych od uruchomienia procesora. Instrukcje te, tak jak i inne instrukcje asemblerowe procesora, są dostępne po dodaniu odpowiedniej biblioteki zależnej od kompilatora: dla MSVC **intrin.h**, dla GCC **x86intrin.h**.

```
uint64_t __rdtsc();
uint64_t  __rdtscp(unsigned *IA32_TSC_AUX);// pod linuxem do
// IA32_TSC_AUX zapisuje numer wątku na którym wykonano instrukcję
```

Musimy pamiętać o dwóch ważnych rzeczach. Po pierwsze procesor swój czas przeznacza nie tylko na nasz program, stąd wyniki mogą być nieco zawyżone. Z drugiej strony tak działa system, więc jak tworzymy aplikację, to musimy to uwzględnić, więc nie ma co się tym przejmować. Po drugie mierzenie ilości cykli procesora nie musi zależeć od aktualnej częstotliwości procesora tylko może być wartością stałą związaną z bazową częstotliwością procesora. Informacje te da się wydobyć dla GCC instrukcją **___get_cpuid** dostępną w bibliotece **cpuid.h** lub dla MSVC instrukcją **___cpuid** z **intrin.h**. Można również użyć funkcji **clock()** wraz z **___rdtsc** by wyznaczyć przybliżoną częstotliwość procesora*.

Przykład

tsc.c

```c
#include <stdio.h>
#include <stdint.h>
#ifdef _MSC_VER // w zależności od kompilatora
# include <intrin.h> // dołączamy odpowiednią bibliotekę
#else
# include <x86intrin.h>
#endif
uint64_t readTSC();// nowsze wersje gcc wymagają deklaracji
uint64_t readTSCp();// funkcji inline lub poprzedzenia ją static
// prostsza wersja bez podkreślników z ewentualnymi poprawkami
inline uint64_t readTSC(){
//  _mm_mfence();// wymusza czekanie na wykonanie instrukcji przed
    uint64_t tsc = __rdtsc();
//  _mm_mfence();// wymusza poczekanie na wykonanie istrukcji rdtsc
    return tsc;
}
// jak wyżej, czeka na wykonanie instruckcji przed
inline uint64_t readTSCp(){
    unsigned dummy;// nie używamy, w nim zapiszemy IA32_TSC_AUX
    return __rdtscp(&dummy);
}
int main(){
    printf("Wartosc tsc: %llu\n",readTSC());
    printf("Wersja 2: %llu\n",readTSCp());
}
```

Biblioteki **intrin.h** lub **x86intrin.h** zawierają mnóstwo innych funkcji będących odpowiednikami instrukcji procesora. Można tam znaleźć m.in. pozwalające na używanie MMX, SSE, AVX czy też obrotów bitowych czy wyszukiwań bitowych. Instrukcje te mogą się przydać przy optymalizacji programu, lecz trzeba pamiętać, że są one zależne od sprzętu, więc musimy się upewnić, że platforma na której będzie uruchomiony program będzie miał te instrukcje procesora.
Trzeba dodać, że tutaj rpdzaj kompilatra decyduje czasem o tym gdzie ta funkcja jest i jaką ma składnie (MSVC i GCC mają tutaj czasem całkiem różne podejście).Przykładem mogą być tutaj obroty bitowe, które są dla MSVC są dostępne (w zależności od typy zmiennej całkowitej) jako **_rotl8**, **_rotl16**, **_rotl**, **_rotl64** (dla obrotów w prawo mamy **_rotr8**, **_rotr16**, **_rotr**, **_rotr64**), a w GCC jako **__rolb**, **__rolw**, **__rold**, **__rolq** (dla obrotów w prawo mamy **__rorb**, **__rorw**, **_rord**, **_rorq**).

Przydatnymi funkcjami mogą być wyszukiwania ustawianych bitów (z góry lub z dołu).

- Wyszukanie pierwszego ustawionego bitu w zmiennej **Mask** i zapisanie jego numeru do **Index**. Funkcje dla MSVC zwracają wartość 0 gdy brak ustawionego bitu, w pozostałych przypadkach zwracają wartość 1.

```
unsigned char _BitScanForward( unsigned long * Index, unsigned
    long Mask);// MSVC 32 bit
unsigned char _BitScanForward64( unsigned long * Index, unsigned
    long long Mask);// MSVC 64 bit
int __bsfd(int Mask);// GCC 32 bit
int __bsfq(long long Mask);// GCC 64 bit
```

- Wyszukanie ostatniego ustawionego bitu w zmiennej **Mask** i zapisanie jego numeru do **Index**. Funkcje dla MSVC zwracają wartość 0 gdy brak ustawionego bitu, w pozostałych przypadkach zwracają wartość 1.

```
unsigned char _BitScanReverse( unsigned long * Index, unsigned
    long Mask);// MSVC 32 bit
unsigned char _BitScanReverse64( unsigned long * Index, unsigned
    long Mask);// MSVC 64 bit
int __bsrd(int Mask);// GCC 32 bit
int __bsrq(long long Mask);// GCC 64 bit
```

Innymi przydatnymi funkcjami są:

- Kopiowanie **n** elementów tablicy **src** do tablicy **dst**. Rozmiar elementu tablicy to 1, 2, 4 lub 8 bajtów (przy czym 8 dla procesorów 64 bitowych), odpowiednio dla rozmiaru funkce te nazywają się **__movsb**, **__movsw**, **__movsd**, **__movsq**. Nie będziemy tu bardziej wchodzić w szczegóły tych funkcji, bo **memcpy** działa równie szybko jak one, a nawet szybciej (zależnie od kompilatora, czasem dopiero po włączeniu optymalizacji).
- Wstawianie podanej wartości **data** do **n** elementów tablicy **dst**. Rozmiar elementu tablicy to 1, 2, 4 lub 8 bajtów (przy czym 8 dla procesorów 64 bitowych). Pamiętajmy, że **memset** wypełnia tylko "jednobajtowo", co nie zawsze może nam odpowiadać. Poniższe funkcje działają równie szybko jak **memset**.

```
// dla MSVC
void __stosb(unsigned char* dst, unsigned char data, size_t n);
void __stosw(unsigned short* dst, unsigned short data, size_t n);
void __stosd(unsigned long* dst, unsigned long data, size_t n);
void __stosq(unsigned long long* dst, unsigned long long data,
    size_t n);
```

  Dla GCC niestety nie mamy dostępnych tych funkcji, musimy je sami zdefiniować używając wstawek asemblerowych.

```
static inline void __stosb(int8_t *Dest, int8_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosb" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}

static inline void __stosw(int16_t *Dest, int16_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosw" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}

static inline void __stosd(int32_t *Dest, int32_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosl" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}

static inline void __stosq(int64_t *Dest, int64_t Data, size_t Count){
    __asm__ __volatile__ ( "cld\n\t" "rep stosq" : : "D" (Dest),
                           "c" (Count), "a" (Data) : "memory");
}
```

bits.c

rep.c

Wyrównanie danych (C11)

Dane przetrzymywane w pamięci mogą zaczynać się w pewnych nieoptymalnych dla procesora miejscach, przez co dostęp do nich może być wolniejszy. Domyślnie dane są wyrównane do potęg liczby 2 odpowiadających rozmiarowi naszej zmiennej. W przypadku tablic jest to wyrównanie do wyrównania pojedynczego elementu tablicy. W przypadku struktur do wyrównania największego wyrównania spośród typów zmiennych w strukturze. Pamięć alokowana na stercie również jest domyślnie wyrównywana przynajmniej do największego wyrównania standardowych typów zmiennych (typ ten nazywa się **max_align_t** i jest dostępny w bibliotece **stddef.h**).
Wyrównanie danych można zmieniać jak i sprawdzać za pomocą funkcji z biblioteki **stdalign.h**.

```
alignof(typ_danych)
```

zwraca wyrównanie (jako typ **size_t**) podanego typu danych.

```
alignas(wyrównanie) typ_danych nazwa_zmiennej;
```

ustawia wyrównanie dla danej zmiennej.

Wyrównanie danych

align.c

```c
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
struct test{
    char var1;
    char var2[8];
} test;

struct testalign{
    char var1;
    alignas(8) char var2[8];
} testalign;

int main(void){
    printf("Wyrownanie max_align_t = %zu\n", alignof(max_align_t));
    printf("Wyrownanie int[13] = %zu\n", alignof(int[13]));
    printf("Wyrownanie test = %zu\n", alignof(test));
    printf("Rozmiar test = %zu\n", sizeof(test));
    printf("Wyrownanie test = %zu\n", alignof(testalign));
    printf("Rozmiar testalign = %zu\n", sizeof(testalign));
}// u mnie kolejno: 16, 4, 1, 9, 8, 16
```

Typy generyczne (C11)

W zależności od pewnych rzeczy, pod jedną nazwą chcielibyśmy mieć funkcje operujące na różnych typach zmiennych. Żeby to zrobić można użyć właśnie typów generycznych. Składnia wygląda następująco:

```
_Generic (nazwa, lista_powiazan);
```

gdzie elementy listy powiązań składa się z dwóch elementów

```
typ_danych: wyrazenie
```

Poszczególne elementy listy są oddzielone przecinkami. Typ danych może też być domyślny (**default**), który oznacza tutaj wszystkie pozostałe mogące się pojawić typy.

Przykład typu generycznego

generic.c

```c
#include <stdio.h>
#include <math.h>
#define cbrt(X) _Generic((X), \
    long double: cbrtl(X), \
    default: cbrt(X),  \
    float: cbrtf(X),  \
    int: llround(cbrt(X)) \
)

int main(void){
    double x = 8.0;
    int y=9;
    printf("cbrt(8.0) = %f\n", cbrt(x)); // 2.000
    printf("cbrt(9) = %lld\n", cbrt(y)); // 2
    return 0;
}
```

Funkcje bez powrotu i długie skoki (C11)

Istnieją funkcje, które nigdy nie powracają do miejsca ich wywołania. Takie funkcje tworzy się poprzez dodanie słowa **_Noreturn** lub makra **noreturn** (z biblioteki **stdnoreturn.h**) przed zwracany typ funkcji (który to oczywiście musi być **void**). W funkcji takiej oczywiście nie używamy **return** i kod musi albo zakończyć działanie programu (funkcja **exit**) albo wykonać długi skok do innego miejsca w programie. Długie skoki mają na celu obsługę nieoczekiwanych błędów i są analogiem obsługi wyjątków w innych językach programowania. Potrzebna nam będzie biblioteka **setjmp.h**. W niej mamy typ **jmp_buf**, a zmienna tego typu przechowuje nam adres do kodu wykonywanego przez procesor. Adres taki można zapisać do zmiennej używając funkcji

```
int setjmp(jmp_buf env);
```

która to zwraca 0 gdy nie jest wywołana w kodzie przez długi skok lub w przypadku użycia długiego skoku jest to wartość, która poprzez niego jest przekazywana (nigdy wtedy nie jest zerem).

Sam długi skok jest realizowany za pomocą

```
_Noreturn void longjmp( jmp_buf env, int status );
```

gdzie **env** jest zapisanym adresem powrotu do kodu, a **status** jest wartością do zwrócenia w funkcji **setjmp** (gdy jest ona równa 0, to zostaje poprawiona na wartość 1).

Przykład

```
#include <stdio.h>
#include <setjmp.h>
#include <stdnoreturn.h>

jmp_buf jump_buffer;

noreturn void a(int count){
    printf("a(%d) called\n", count);
    longjmp(jump_buffer, count+1); // setjmp zwróci count+1
}

int main(void){
    volatile int count = 0; // volatile ze względu na setjmp
    if (setjmp(jump_buffer) != 9)
        a(count++);
    return 0;
}// wyświetli się "a(0) called" do "a(8) called"
```

Przykład

longjmp.c

```c
#include <stdio.h>
#include <setjmp.h>
#define DIVIDE_EXCP 1
#define NEG_EXCP 2
double divide(const double x, const double y, jmp_buf env){
    if(y == 0){
        longjmp(env, DIVIDE_EXCP);
    }else if(y < 0){
        longjmp(env, NEG_EXCP);
    }
    return x/y;
}
void oldway(const double x, const double y){
    jmp_buf env;
    int catch_excp = setjmp(env);
    if(catch_excp == 0){
        printf("x/y = %f\n",divide(x, y, env));
    }else if(catch_excp == DIVIDE_EXCP){
        printf("Dzielenie przez zero jest niedozwolone\n");
    }else{
        printf("W sumie bez bledu, y<0\n");
    }
}
int main(int argc, char **argv){
    double x, y;
    printf("Podaj dwie liczby rzeczywiste: ");
    scanf(" %lf %lf",&x,&y);
    oldway(x, y);
    return 0;
}
```