Kurs C

Radosław Łukasik

Wykład 2

Etykiety i skoki bezwarunkowe

Etykietą nazywamy napis składający się z liter alfabetu angielskiego, cyfr, oraz podkreślnika (nie może on zaczynać się od cyfry), zakończony dwukropkiem, wskazujący miejsce w kodzie. Polecenie

```
goto etykieta;
```

jest instrukcją skoku bezwarunkowego do miejsca wskazanego przez etykietę.

Przykład

```
    int x=5;
poczatek:
    printf("Jestesmy na poczatku\n");
    x++;
srodek:
    printf("Jestesmy w srodku\n");
    x++;
    if(x<8)
        goto poczatek;
    if(x<12)
        goto srodek;
    printf("Koniec\n");
```

Chociaż skokami bezwarunkowymi oraz **if**...**else** można symulować każdą pętle, to należy ich ostrożnie używać (niektórzy nawołują do całkowitego zrezygnowania z nich) ponieważ źle napisany kod z ich użyciem może zawiesić program lub spowodować błąd w jego działaniu. Przeważnie wszystkie zastosowania tej instrukcji można zastąpić bardziej czytelnymi poleceniami **if**...**else**, instrukcją wyboru **switch** oraz pętlami.

```
for(int i=0;i<20;i++){
    if(i%5 != 0)
        printf("%d\n",i);
}
```

```
int i=0;
poczatek:
    if(i>=20)
        goto koniec;
    if(i%5 != 0)
        printf("%d\n",i);
    i++;
    goto poczatek;
koniec:
```

Stos, sterta i inne segmenty danych

Stos jest to liniowa struktura danych w pamięci przydzielona do konkretnego programu, w którym dane są dokładane na jej wierzchołku i później z niego zdejmowane (bufor typu LIFO). Na stosie przechowywane są wszystkie niestatyczne zmienne lokalne oraz parametry funkcji, wartości zwracane przez funkcje i adresy instrukcji występujących po wywołaniu funkcji. Stos ma z góry ustalony rozmiar zależny od kompilatora (można go zmienić) i jest rzędu 1MB. Nadmierna rekurencja funkcji lub tworzenie dużej ilości zmiennych lokalnych może powodować do jego przepełnienia i zakończenia działania programu.
Sterta jest to obszar pamięci dostępny dla wszystkich programów, w którym można tworzyć dynamicznie zmienne.
Należy tutaj pamiętać, że zmienne globalne lub lokalne statyczne, które są zainicjowane są przechowywane w segmencie danych a te niezainicjowane w segmencie BSS. Może się też zdarzyć, że dane są ukryte w segmencie kodu (wartości przy operacjach arytmetycznych, czy też przyrównaniach, a także pewne łańcuchy znaków), który to jest tylko do odczytu. Przykładowo poniższy kod powoduje błąd naruszenia pamięci:

```
char *tekst="Tekst w segmencie kodu";
tekst[0]='a';// to powduje błąd
```

Tablice (statyczne)

Tablica jest ciągiem zmiennych tego samego typu. Nazwa tablicy jest zarazem wskaźnikiem na jej początek (czyli pierwszy element numerowany indeksem 0).
Tablice mogą być jedno lub wielowymiarowe. W przypadku tablic wielowymiarowych o wymiarach n_1 aż do n_k pozycja $[i_1][i_2]...[i_k]$ jest oddalona od początku tablicy o (w rozmiarze typu tablicy)

$$i_k + n_{k-1} * (i_{k-1} + n_{k-2} * (i_{k-3} + ... + n_1 * i_1)...).$$

Łańcuchem znaków nazywamy tablicę jednowymiarową typu **char** (ewentualnie **wchar_t**). Znak '\0' jest zarezerwowany do oznaczania końca łańcucha (który nie musi być końcem tablicy).
Deklarując tablicę (w sposób statyczny, na stosie) należy podać jej wszystkie wymiary (jako stałe liczby). W przypadku gdy tablica jest zainicjowana pewnymi wartościami możemy pominąć podawanie pierwszego wymiaru. Lista wartości początkowych tablicy musi zgadzać się z rozmiarem tablicy.
Od C99 można tworzyć tablicę o długości zdefiniowanej poprzez zmienną, przy czym takiej tablicy nie można zainicjować.
Również od C99 można inicjować tylko wybrane pola tablicy, reszta zostanie uzupełniona zerami.

```
typ nazwa[rozmiar] = { [wspolrzedna] = wartosc };
```

Rozmiar tablicy można uzyskać za pomocą **sizeof**, a ilość elementów poprzez podzielenie rozmiaru tablicy przez rozmiar typu tablicy (lub przez rozmiar pierwszego elementu).

Przykład

```
int tablica1[40];
double liczby[4]={ 1.0, 2.0, 3.0, 4.0 };
int power[]={ 1, 2, 4, 8, 16 };
int tablica2[3][2]={{1,2},{2,3},{3,4}};
    // można też tak   ={1,2,2,3,3,4};
    // gdy są podane wszystkie wymiary
char znaki[][10]={"zero","jeden","dwa"};
int tablica3[5]={[2]=16,[4]=11};// ={0,0,16,0,11}
```

Przykład

```
int tablica1[40];
for(int i=0;i<40;i++)
    tablica1[i]=2*i;

int tablica2[3][3];
for(int i=0;i<3;i++)
    for(int j=0;j<3;j++)
        tablica2[i][j]=i+j;
```

Wskaźniki

Wskaźnikiem nazywamy zmienną przechowującą adres pamięci, pod którym znajduje się zmienna pewnego typu (zmienna wskazywana). Deklarujemy go w następujący sposób:

```
typ_wskazywanej_zmiennej *nazwa_wskaznika;
```

Do wskaźnika możemy przypisać adres zmiennej używając operatora przypisania & lub przypisać do niego inny wskaźnik tego samego typu lub przypisać wskaźnik pusty (wartość 0 lub **NULL**). Do zmiennej możemy przypisać wartość wskazywaną przez wskaźnik (tego samego typu co zmienna) używając operatora wyłuskania *. Należy dbać o to by wskaźnik nie miał losowej wartości, bo wtedy jego użycie może spowodować błąd programu. Używanie pustego wskaźnika nie powoduje błędów, więc gdy już nie będziemy korzystać ze wskaźnika warto go wyzerować.

Przykład

```
int x=5;
int *pX =0; // pusty wskaźnik
pX=&x;      // pX wskazuje na zmienną x, czyli zawiera jej adres
int y=*pX;  // y=x
*pX+=5;     // x=10
```

Uwaga

Używając operatora wyłuskania i działań inkrementacji lub dekrementacji lepiej jest napisać operator wyłuskania wraz ze wskaźnikiem w nawiasach, bo kompilator może zrozumieć, że działamy na wskaźniku a nie na pamięci przez niego wskazywanej (co może powodować błędy).

```
int x=5;
int *pX =&x; // pX zawiera adres x
(*pX)++;     // zwiększamy x o 1
*pX++;  // zwiększamy wskaźnik o rozmiar typu wskaźnika
        // (rozmiar int = 4), pX nie wskazuje już na x
```

Każdy wskaźnik ma taki sam rozmiar. Jeżeli nie wiadomo na jaki typ ma dany wskaźnik wskazywać można go zrobić wskaźnikiem typu **void** (tak samo, gdy ma on obejmować wiele różnych typów), a następnie go rzutować na wskaźnik typu jaki potrzebujemy. Istnieje również wskaźnik na wskaźnik o następującej definicji:

```
typ **nazwa;
```

Może się on przydać np. w przypadku tablic wskaźników lub przekazywaniu wskaźników do funkcji, gdzie mogą być one zmienione.

Przykład

```
char a; //znak
char* b; //wskaźnik na znak
char** c; //wskaźnik na wskaźnik na znak
char d[10]; //tablica znaków
char* e[2]; //tablica wskaźnikow na znaki
char* (*f)[]; // wskaźnik na tablicę wskaźnikow na znak

//2 napisy po 10 znaków każdy
char tab[2][10]={"123456789","ABCDEFGHI"};
e[0]=tab[0];
e[1]=tab[1];
char** tabptr = e;

a=**tabptr;          //pierwszy znak pierwszego napisu
printf("%c\n",a);// 1
a=**(tabptr+1);      //pierwszy znak drugiego napisu
printf("%c\n",a);// A
a=*(*tabptr+1);      //drugi znak pierwszego napisu
printf("%c\n",a);// 2
a=*(*(tabptr+1)+1); //drugi znak drugiego napisu
printf("%c\n",a);// B
```

Alokacja pamięci na stercie

Aby tworzyć duże obiekty na stercie należy zarezerwować na nie pamięć. Służą do tego funkcje dostępne w bibliotece **stdlib.h**. Zwracają one wskaźnik na dowolny typ (wskaźnik ten wskazuje początek zarezerwowanego bloku pamięci), który musimy rzutować na typ, którego chcemy używać.

```
void* malloc (size_t size);//alokuje podaną ilość pamięci zwracając
                           //wskaźnik do niej (lub NULL gdy błąd)
void* calloc (size_t num, size_t size);//alokuje i zeruje num*size
     // bajtów pamięci zwracając wskaźnik do niej (lub NULL)
void* realloc (void* ptr, size_t size);//zwraca wskaźnik na pamięć
     // powstałą z ptr po zmianie rozmiaru na size w przypadku sukcesu
     // kopiuje* i zwalnia* pamięć pod starym wskaźnikiem
void free (void* ptr);// zwalnia pamięć pod adresem ptr
```

Przykład

```
int *wsk=(int*)malloc(1024*sizeof(int));// rzutowanie na int*
if(wsk!=0){
    /* jakieś operacje na pamięci */
}
free(wsk);
```

Należy pamiętać o tym by to co było alokowane za pomocą **malloc** (**calloc** czy też **realllloc**) było usuwane za pomocą **free**.

Należy też ostrożnie używać realokacji pamięci by nie doprowadzić do wycieku pamięci. Z tego względu dobrze jest użyć drugiego wskaźnika.

Przykład

```
int *wsk=(int*)malloc(1024*sizeof(int));// rzutowanie na int*
if(wsk!=0){
    /*
    jakieś operacje na pamięci
    */
    // chcemy zmienić rozmiar zarezerwowanej pamięci
    int *tmp=(int*)realloc(wsk,2048*sizeof(int));// dodatkowy wskaźnik
    if(tmp!=0){
        wsk=tmp;
        /*
        dalsze operacje na powiększonej pamięci
        */
    }else{
        // nie udało się zmienić rozmiaru, ale dalej możemy coś robić
        // na pamięci wskazywanej przez wsk
    }
}
free(wsk);
```

Liczby pseudolosowe

Często by testować pewne operacje na liczbach przydałoby się mieć je wypełnione pewnymi losowymi wartościami.
W bibliotece **stdlib.h** znajdują się funkcje związane z pseudolosowością.

```
srand(unsigned int ziarno);
```

funkcja ta inicjuje generator liczb pseudolosowych. Jeżeli **ziarno** jest stałe, to po każdym uruchomieniu będziemy mieć stały ciąg liczb pseudolosowych. Dlatego jako ziarno najlepiej stosować np. obecny czas w sekundach uzyskany za pomocą **time(0)** z biblioteki **time.h**. Inicjację generatora liczb losowych wystarczy wywołać raz na początku programu.

```
int rand();
```

funkcja ta zwraca wartość będącą liczbą pseudolosową w zakresie od 0 do **RAND_MAX** (ma to być co najmniej 15 losowych bitów, w Windows jest to $2^{15} - 1 = 32767$, a w Linux $2^{31} - 1 = 2147483647$).
Jeżeli chcemy mieć liczby losowe należące do jakiegoś przedziału $[m, n]$ ($n - m <$ RAND_MAX), to możemy to zrobić kodem:

```
liczba=m+(rand()%(n+1-m));
```

Przykład

```
int n=1024*1024;
int *tabl=(int*) malloc(n*sizeof(int));
if(tabl==0)
    printf("Brak pamieci\n");
else{
    long long srednia=0;
    for(int i=0;i<n;i++){
        tabl[i]=rand()%100;
        srednia+=tabl[i];
    }
    printf("srednia rand: %f\n",((double) srednia/n));
}
free(tabl);
```

Kopiowanie i porównywanie bloków pamięci i łańcuchów znaków

Gdy mamy dwie tablice i chcemy skopiować zawartość jednej do drugiej, to możemy do tego celu użyć pętli. Niestety nie jest to najefektywniejsze podejście. Lepiej jest skorzystać z pewnych zoptymalizowanych funkcji znajdujących się w bibliotece **string.h**. W tej bibliotece mamy również inne ciekawe funkcje, które tutaj przedstawimy.

```
memcpy(void *cel, const void *zrodlo, size_t ilosc_bajtow);
```

kopiuje z pamięci wskazanej przez **zrodlo** podaną **ilość bajtów** (**size_t** -typ całkowity bez znaku) do pamięci wskazanej przez **cel**.

```
memccpy(void *cel, const void *zrodlo, int znak, size_t ilosc_bajtow);
```

kopiuje z pamięci wskazanej przez **zrodlo** do pamięci wskazanej przez **cel** aż do napotkania **znaku** (**unsigned char**). Jeżeli go nie napotka, to kopiuje podaną **ilość bajtów**.

```
memmove(void *cel, const void *zrodlo, size_t ilosc_bajtow);
```

kopiuje z pamięci wskazanej przez **zrodlo** do pamięci wskazanej przez **cel** podaną **ilość bajtów**, przy czym pamięci te mogą na siebie zachodzić.

```
memset(void *cel, int wartosc, size_t ilosc_bajtow);
```

wypełnia podaną **ilość bajtów** pamięci wskazanej przez **cel** wskazaną **wartością** (**unsigned char**).

```
memchr(void *zrodlo, int wartosc, size_t ilosc_bajtow);
```

szuka pierwszego wystąpienia **wartości** (**unsigned char**) w pierwszej **ilość bajtów** pamięci wskazanej przez **źródło**.

```
int memcmp(const void *zrodlo1, const void *zrodlo2, size_t dlugosc);
```

porównuje zadaną **długość** bajtów w dwóch **źródłach** pamięci. Zwraca 0 gdy są równe, liczbę dodatnią gdy pierwszy bajt nie będący taki sam w obu blokach był większy w **pierwszym źródle**, wartość ujemna w przeciwnym wypadku.

Przykład

```
int n=1024;
char *pTab1=(int*) malloc(n);
char *pTab2=(int*) malloc(n);
if(pTab1==0||pTab2==0)
    printf("Za malo pamieci\n");
else{
    memset(pTab1,0,n);// zerujemy tablicę 1
    memcpy(pTab1,pTab2,n);// kopiujemy ją do tablicy 2
}
free(pTab1);
free(pTab2);
```

PetleTablic.c

Można oczywiście stosować wcześniejsze funkcje dla łańcuchów znaków zakończonych zerem, ale są dla nich dostępne specjalne funkcje.

```
strcpy(char *cel, const char *zrodlo);
```

kopiuje tekst ze **źródła** do **celu**.

```
strncpy(char *cel, const char *zrodlo, size_t dlugosc);
```

kopiuje podaną **ilość znaków** ze **źródła** do **celu** (gdy **ilość znaków** $<$ długość **źródła**, to trzeba dodać kod **cel[n]=0;**).

```
int strcmp(const char *lancuch1, const char *lancuch2);
```

porównuje dwa łańcuchy znaków ze sobą, zwraca 0 gdy są równe, liczbę dodatnią gdy pierwszy znak nie będący taki sam w obu łańcuchach był większy w **pierwszym łańcuchu**, wartość ujemna w przeciwnym wypadku.

```
int stricmp(const char *lancuch1, const char *lancuch2);
```

działa tak samo jak **strcmp** tylko ignoruje wielkość liter.
Powyższe dwie funkcje nie radzą sobie z polskimi znakami. Jeżeli chcemy porównywać z uwzględnieniem polskich liter, to musimy wcześniej zdeklarować, że używamy języka polskiego (używając polecenia **setlocale(LC_ALL,"Polish");** z biblioteki **locale.h**) a następnie skorzystać z odpowiedniej funkcji

```
int strcoll(const char *lancuch1, const char *lancuch2);
int stricoll(const char *lancuch1, const char *lancuch2);
```

będących kolejno odpowiednikiem **strcmp** i **stricmp**.

Przykład

```
char szA[32]={"Nowak Jan"};
char szB[32]={"Kowalski Piotr"};
char szTmp[32];
if(strcmp(szA,szB)>0){// sortowanie
    strcpy(szTmp,szA);
    strcpy(szA,szB);
    strcpy(szB,szTmp);
}
```

```
size_t strlen(const char *lancuch);
```

zwraca długość **łańcucha** znaków.

```
char * strchr(const char *lancuch, int znak);
```

szuka w **łańcuchu** pierwszego wystąpienia **znaku** (w ASCII). Zwraca wskaźnik na to wystąpienie lub 0, gdy nie ma takiego znaku.

```
char * strrchr(const char *lancuch, int znak);
```

szuka w **łańcuchu** ostatniego wystąpienia **znaku** (w ASCII). Zwraca wskaźnik na to wystąpienie lub 0, gdy nie ma takiego znaku.

```
size_t strspn(const char *lancuch1, const char *lancuch2);
```

zwraca indeks pierwszego znaku z **pierwszego łańcucha**, który nie występuje w **drugim łańcuchu** znaków. Jeżeli wszystkie znaki występowały, to zwraca długość **pierwszego łańcucha**.

```
size_t strcspn(const char *lancuch1, const char *lancuch2);
```

zwraca indeks pierwszego znaku z **pierwszego łańcucha**, który występuje w **drugim łańcuchu** znaków. Jeżeli żaden ze znaków nie występował, to zwraca długość **pierwszego łańcucha**.

```
const char * strstr(const char *lancuch1, const char *lancuch2);
```

szuka w **pierwszym łańcuchu** pierwszego wystąpienia **drugiego łańcucha**. Zwraca wskaźnik na to wystąpienie lub 0, gdy nie ma takiego znaku.

```
strcat(char *lancuch1, const char *lancuch2);
```

dopisuje **drugi łańcuch** na końcu **pierwszego łańcucha**.

```
strncat(char *lancuch1, const char *lancuch2, size_t  liczba_znakow);
```

dopisuje co najwyżej podaną **liczbę znaków** z **drugiego łańcucha** na końcu **pierwszego łańcucha**.

Większość z wymienionych funkcji z **string.h** ma swoje odpowiedniki dla typu **w_char_t** zamiast **char** i są one dostępne w bibliotece **wchar.h**.

Przykład

```
char napis[]="Numer1234wtekscie";
char cyfry[]="0123456789";
char lancuch[256];
lancuch[0]=0;
int pocz=strcspn(napis,cyfry); // pocz=5
if(pocz<strlen(napis)){
    int kon=strspn(napis+pocz,cyfry); // kon=4
    printf("Poczatek numeru: %d, dlugosc: %d\n",pocz,kon);
    strncpy(lancuch,(napis+pocz),kon); // lancuch="1234"
    lancuch[kon]=0;
}else
    printf("W tekscie nie ma numeru.\n");
strncat(lancuch,napis,5); // lancuch="1234Numer";
printf("%s,lancuch);
```

Przykład

```
char napis[]="Jakies krotkie zdanie. I kolejne.";
char *pocz,*kon;
char lancuch[256];
kon=strchr(napis,'.');// szukamy kropki
if(kon!=0){
    *kon=0;
    pocz=strrchr(napis,' ');// szukamy spacji
    if(pocz!=0)
        strncpy(lancuch,pocz+1,kon-pocz);
    else
        strncpy(lancuch,napis,kon-napis);
    printf("%s",lancuch);// "zdanie"
}else
    printf("Nie ma kropki.");
```

Funkcje

- Funkcja jest podprogramem, który wykonuje pewne konkretne zadanie.
- Funkcja może pobierać pewne argumenty, może także zwracać pewne wartości.
- Jeżeli w programie mamy kod, który często się powtarza, to warto zastąpić go funkcją.
- Poszczególne argumenty funkcji są oddzielone przecinkami.
- Rozróżniamy deklarację i definicję funkcji. Definicja polega na podaniu typu zwracanego przez funkcję (**void** gdy nic nie zwraca), argumentów funkcji wraz z ich typami, a także kodu samej funkcji. Deklaracja natomiast zawiera typ zwracany oraz typy argumentów (można podać również nazwy zmiennych). Jeśli funkcja jest wywoływana w kodzie po swojej definicji, to nie wymaga deklaracji.
- Funkcja zwraca wartość za pomocą **return** (wartość lub wyrażenie);
  Jeżeli funkcja nic nie zwraca, to albo używamy **return**; bez wartości zwracanej albo nie używamy go wcale.
- Wywołanie funkcji polega na podaniu wartości argumentów funkcji i przypisaniu wartości funkcji do zmiennej czy też wyświetleniu jej.
- Nazwy argumentów funkcji mogą być takie same jak zmiennych globalnych (przesłanianie).

```
typ nazwa(typ1, ...,typn);// deklaracja - sam prototyp funkcji
:
typ nazwa(typ1 arg1, ...,typn argn){// definicja
    /*                                // = prototyp + treść funkcji
        instrukcje
    */
    return (wyrażenie);
}
```

Przykład - n^n

```
#include <stdio.h>
void oblicz(unsigned);// deklaracja
void oblicz(unsigned n){// definicja
   int m=1;// złożoność czasowa O(n)
   for(int j=0;j<n;j++)// n iteracji
       m*=n;
   printf("%u ^ %u = %u\n",n,n,m);
}
int main(){
    unsigned n;
    printf("Podaj liczbe");
    scanf(" %u",&n);
    oblicz(n);// wywołanie funkcji
    return 0;
}
```

Przekazywanie parametrów poprzez wartości

- Przy wywołaniu funkcji tworzone sa lokalne kopie argumentów wejściowych (na stosie). W obliczeniach modyfikowane sa wartości tych kopii a nie samych zmiennych.
- Argumentami przekazywanymi podczas wywołania funkcji mogą być stałe, zmienne lub wyrażenia.
- Tablica jest wskaźnikiem, więc można ją modyfikować wewnątrz funkcji.
- W przypadku tablic wielowymiarowych konieczne jest podanie wszystkich jej wymiarów w argumencie funkcji.

Przykład

```
// double potega(double, int); - prototyp
int n=5;
potega(1.2,n); // stałe i zmienne
potega(1.2+2.3,n+4); // wyrażenia
```

Przekazywanie wartości poprzez wskaźniki

Uwaga W C++ występuje przekazywanie poprzez referencje ale nie występuje ono w C (wskaźniki i referencje to prawie to samo, referencje są łatwiejsze w użytkowaniu).

- Do funkcji przesyłane sa wskaźniki do argumentów
- zmienna w definicji funkcji jest wskaźnikiem i trzeba użyć operatora wyłuskania * aby zmodyfikować zmienną wewnątrz funkcji oraz operatora referencji & by przekazać funkcji adres zmiennej.
- Modyfikacje zmiennych wskazywanych wewnątrz funkcji sa widoczne po wykonaniu funkcji.
- Argumentami przekazywanymi do funkcji jako wskaźniki mogą być wyłącznie zmienne.

Przykład - przekazywanie poprzez wskaźniki

```
short tab[]={12,14,17,102};
short minimum; // najmniejszy element tablicy
int zaDuze; // liczba elemtów większych od 100
int funkcja(short *pminimum,short tablica[],int n){
    int j=0;
    (*pminimum) = tablica[0];
    for(int i=0;i<n;i++){ // n-rozmiar tablicy
        if(tab[i]>100){
            tab[i]=100; // zmniejszamy za duże elementy
            j++;        // i zliczamy je
        }else
            if(tablica[i]<*pminimum)
                *pminimum=tablica[i];
    }
    return j;
}
:
:
zaDuze=funkcja(&minimum,tab,4); // minimum=12, zaDuze=1
                                // tab={12,14,17,100}
```

Wskaźniki **const** w funkcjach

Wśród wskaźników mamy specjalny ich rodzaj - wskaźniki **const**. Służą do zabezpieczania zmiennych przed przypadkową zmianą. Mamy ich trzy rodzaje:

```
const typ_zmiennej *nazwa_wskaznika;
typ_zmiennej * const nazwa_wskaznika = wskaznik;
const typ_zmiennej * const nazwa_wskaznika = wskaznik;
```

W pierwszym przypadku wskaźnik wskazuje na stałą wartość (a więc nie można zmienić tej wartości). W drugim przypadku wskaźnik jest stały i nie można przypisać mu innego miejsca w pamięci (można za to zmieniać zmienną na którą wskazuje). Trzeci przypadek jest połączeniem dwóch wcześniejszych (nie możemy zmieniać ani zmiennej na którą wskazuje ani zmienić samego wskaźnika).

Przykład

```
int A=1,B=7,C=2,D=3;
void funkcja(int *const pA, const int *pB, const int *const pC){
    (*pA)+=(*pB)+(*pC); // możemy zmieniać wartość *pA, ale nie pA
    pB=&D; // możemy zmieniać pB ale nie wartość którą wskazuje
           // pC ani *pC nie może być zmienione
}
.
.
.
funkcja(&A,&B,&C); // A=10 reszta bez zmian
```

Będąc przy wskaźnikach występujących jako parametry funkcji warto wspomnieć, że oprócz kwalifikatora **const** mamy również kwalifikator **restrict** informujący kompilator, że pozostałe wskaźniki odwołują się do innego adresu pamięci. Dzięki temu kompilator może wygenerować możliwie najszybszy kod.