Podstawy programowania.

Wykład 7

Funkcje

Programowanie proceduralne

- Pojęcie procedury (funkcji) - programowanie proceduralne
  - realizacja określonego zadania – specyfikacja procedury
  - względna niezależność - dane wejściowe i dane wyjściowe, warunki początkowe i warunki końcowe – kontrakt procedury
  - problem efektów ubocznych
- Zalety użycia procedur
  - redukcja rozmiaru kodu (likwidacja powtarzania się sekwencji rozkazów)
  - umożliwienie ponownego użycia kodu – biblioteki
  - polepszenie czytelności kodu (hierarchiczna struktura kodu)
  - ułatwienie zarządzania kodem (rozbicie jednej złożonej sekwencji instrukcji na szereg mniejszych o względnej autonomii)
- Wniosek: uwaga na efekty uboczne
  - powinny być zapisane w specyfikacji funkcji
- Przykład_1: funkcja wypisz_komunikat_RODO

Programowanie proceduralne

- W C nie używa się pojęcia procedury
  - pojęcie procedury jako wyodrębnionego fragmentu kodu (podprogramu) o własnej nazwie jest bardziej podstawowe niż pojęcie funkcji
  - w innych językach funkcjami nazywane bywają tylko procedury zwracające wynik
  - w pierwotnej wersji C zakładano, że każda funkcja zwraca wartość
    - domyślnie typu `int`
  - w nowszych standardach można definiować funkcje, które nie zwracają wyniku (typ `void` w definicji)
  - zwracany wynik funkcji można pominąć w programie
    - może się tak zdarzyć np. wtedy, gdy funkcja zwraca komunikat o poprawnym lub błędnym wykonaniu ...
      - a programista jest przekonany (?), że wykonanie musi być poprawne
- Wniosek: korzystając z funkcji uwzględniać zwracane wartości

Funkcje w C

- Funkcja jako wyodrębniony fragment kodu o własnej nazwie
  - funkcja (definicja funkcji) zaczyna się od nazwy
  - kod funkcji zawarty jest między nawiasami klamrowymi { ... }
  - wywołanie w dowolnym miejscu programu funkcji (za pomocą jej nazwy) powoduje wykonanie kodu funkcji
  - interakcja funkcji z wywołującym programem (funkcją) polega na:
    - przyjęciu argumentów (jednego, wielu lub żadnego)
    - zwróceniu wyniku (jednego lub żadnego)
  - nazwa funkcji, typy przyjmowanych argumentów i typ zwracanego wyniku stanowią najważniejsze elementy pozwalające poprawnie definiować i wywoływać funkcje (poprawność składniowa kodu)
    - zwyczajowo elementy te tworzą nagłówek funkcji
  - Nagłówek funkcji nie mówi nic o tym co robi kod funkcji
    - informacja ta musi znajdować się w dokumentacji funkcji
      - przydatne jest także stosowanie znaczących nazw funkcji

Programowanie proceduralne

- Przykłady specyfikacji (dokumentacji, opisu) funkcji do umieszczenia w kodzie źródłowym:

```
// oblicz_pierwiastek – funkcja oblicza pierwiastek liczby podwójnej precyzji
double oblicz_pierwiastek( // funkcja zwraca obliczoną wartość pierwiastka
                           // dla liczb nieujemnych, dla liczb ujemnych
                           // funkcja zwraca kod błędu -1
   double liczba // in: zadana liczba
);

// wielomian – funkcja oblicza wartość wielomianu stopnia <= 10
//                brak obsługi błędów – funkcja przerywa działanie
//                                      dla błędnych danych, takich jak: .....
double wielomian( // funkcja zwraca wartość obliczonego wielomianu
   double wsp[],     // in: tablica współczynników wielomianu
   double stopien,  // in: rzeczywisty stopień wielomianu (<= 10 – maksymalny)
   double arg        // in: argument dla którego liczony jest wielomian
);
```

Programowanie proceduralne

- Przykłady specyfikacji funkcji (dokumentacji, opisu) do umieszczenia w kodzie źródłowym:

```
// wczytaj_linie – funkcja wczytuje znaki ze standardowego wejścia, aż do
//                 napotkania EOF lub \n, wynik zapisuje w przesłanej tablicy
//                 dodając \0 po wczytanych znakach
int wczytaj_linie( // funkcja zwraca: kod sukcesu 0 w przypadku powodzenia
                   // (ewentualnie liczbę wczytanych znaków – decyduje kontrakt)
                   //  lub kod -1 w przypadku, gdy zadana tablica jest zbyt krótka
                   // (liczba wczytanych znaków do EOF lub /n jest większa niż
                   // (rozmiar tablicy-1) – tablica zawiera wtedy rozmiar-1 znaków)
   char linia[], // tablica do wpisania wczytanych znaków (+ \0)
   int rozmiar  // rozmiar tablicy – maksymalna liczba wczytanych znaków +1 !
);
```

- Rozmaite konwencje specyfikacji
  - np. przyjęta w podręczniku (*man*) w Unixie/Linuxie

Funkcje

- W momencie kompilacji kodu wywołującego funkcję kompilator musi być w stanie ocenić poprawność wywołania
  - poprawność wywołania funkcji oznacza właściwe:
    - liczbę i typy argumentów
    - typ zwracanego wyniku
  - do oceny poprawności wystarcza znajomość nagłówka
    - formalnie, odpowiednio sformatowany nagłówek stanowi prototyp funkcji
      - prototyp funkcji jest jej deklaracją
    - definicja funkcji także zawiera nagłówek
      - definicja jest jednocześnie deklaracją funkcji
  - w kodzie źródłowym deklaracja funkcji musi poprzedzać wywołanie
    - jako zasadę będziemy przyjmować używanie prototypu funkcji, umieszczonego na początku pliku źródłowego, jako deklaracji
      - dotyczy to także pliku zawierającego definicję funkcji

Funkcje

```
double oblicz_pierwiastek ( double arg ); // deklaracja, prototyp funkcji
                                          // zwyczajowo, mniej ściśle, także: nagłówek funkcji

int main( void ) {
    double liczba = 4.0;
    // wywołanie funkcji – przesłanie argumentu, zwrócenie wyniku
    double pierwiastek = oblicz_pierwiastek( liczba );
    printf("liczba %lf, jej pierwiastek %lf\n", liczba, pierwiastek);
}

// definicja funkcji
double oblicz_pierwiastek ( // funkcja oblicza i zwraca pierwiastek liczby
    double arg // in: liczba do obliczenia pierwiastka
) {
    double pierwiastek;
    // obliczenia pierwiastka, np. za pomocą odpowiedniego ciągu lub szeregu
    return( pierwiastek );
}
```

Programy = algorytmy + struktury danych

- Program w C składa się z funkcji i zmiennych
- Nazwy funkcji mają domyślnie zasięg globalny (cały program)
  - nie ma definicji funkcji wewnątrz innych funkcji
  - standardowo definiowane nazwy funkcji
    - są widzialne (=widoczne) w całym programie
    - nie mogą się powtarzać
- Zmienne można definiować
  - poza funkcjami – ich nazwy mają zasięg globalny lub ograniczony do pliku, w którym są zdefiniowane
    - są widoczne we wszystkich funkcjach programu lub pliku
  - lokalnie wewnątrz funkcji
    - są widoczne tylko w tej funkcji
  - w wewnętrznym bloku w funkcji
    - są widoczne tylko w tym bloku
- Widzialność zmiennych jest odpowiednikiem zasięgu nazw

Widzialność zmiennych

- W bloku instrukcji wewnątrz funkcji widoczne są:
  - zmienne zdefiniowane w tym bloku
  - zmienne zdefiniowane w funkcji...
    - chyba że jakaś zmienna zdefiniowana w bloku zasłoniła zmienną zdefiniowaną w funkcji, poprzez zastosowanie tej samej nazwy
  - zmienne zdefiniowane globalnie...
    - chyba że jakaś zmienna zdefiniowana w bloku lub w funkcji zasłoniła zmienną zdefiniowaną globalnie, poprzez zastosowanie tej samej nazwy
- Wniosek: jeśli nie chcemy, żeby zmienne się zasłaniały stosujmy
  - znaczące nazwy
  - konwencje, np. nazwy zmiennych lokalnych od małej, a globalnych od dużej litery

Czas życia zmiennych

- Z widzialnością zmiennych powiązany jest czas ich życia
  - zmienna może być widoczna tylko i wyłącznie jeśli istnieje
    - wydaje się to oczywiste, ale częstym błędem jest próba wykonania operacji związanej ze zmienną już (jeszcze) nie istniejącą
  - argumenty (formalne) i zmienne lokalne funkcji istnieją tylko w czasie wykonania funkcji
    - zasady funkcjonowania stosu systemowego tłumaczą ich czas życia i widzialność
    - stos systemowy jest elementem pamięci dynamicznej
      - dynamiczny obszar pamięci zmienia swój rozmiar w trakcie wykonania programu
  - zmienne globalne istnieją i są widzialne przez cały czas wykonania programu
    - obszar przechowywania zmiennych istniejących przez cały czas wykonania programu nazywany jest pamięcią statyczną
      - statyczny obszar pamięci jest stały w trakcie wykonania programu

Widzialność zmiennych

- Zmienne zdefiniowane wewnątrz bloku instrukcji są widoczne tylko w tym bloku

```
{
int i;
.....
}
printf("%d\n",i); // - błąd
```

```
int i;
{
.....
}
printf("%d\n",i); // - OK
```

```
for(int i=0; i<N; i++){
.....
}
printf("%d\n",i); // - błąd
```

```
int i;
for(i=0; i<N; i++){
.....
}
printf("%d\n",i); // - OK
```

Widzialność zmiennych

- Język C, definiując zasady zasięgu nazw, dopuszcza zasłanianie jednych zmiennych przez inne o tej samej nazwie

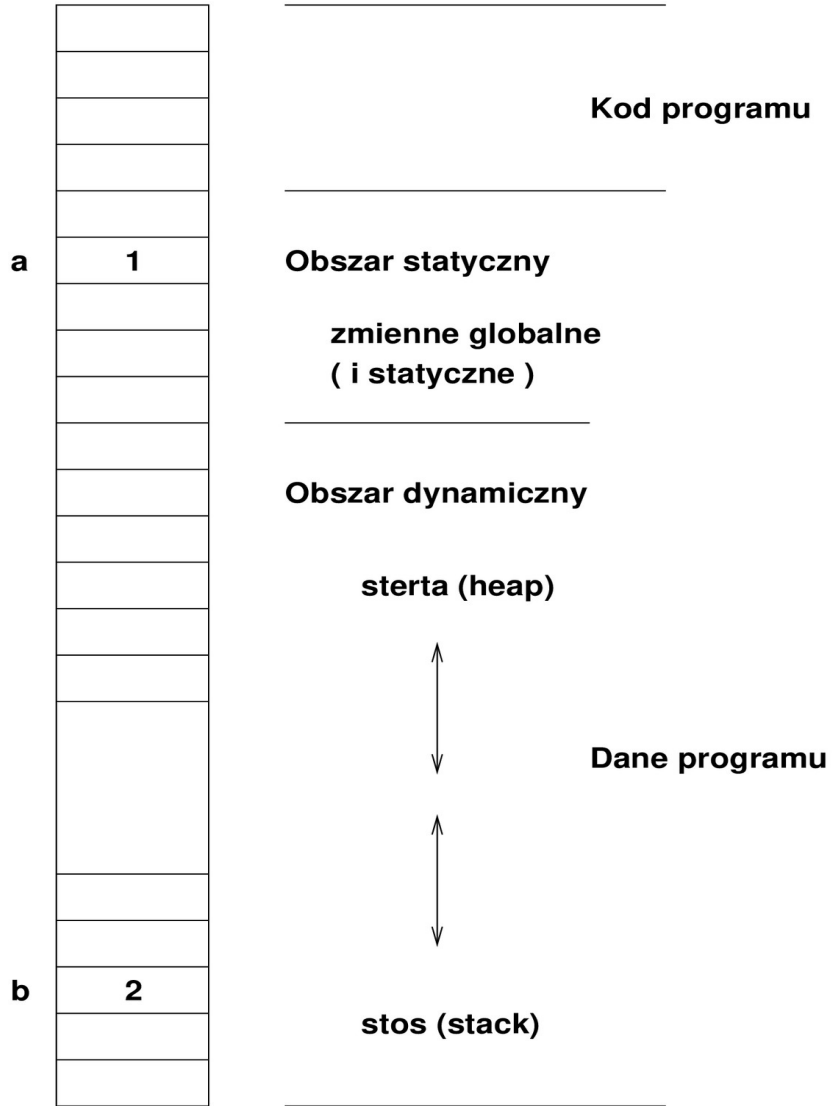
```
int i=3;
// wydruk -> i==3
{
   // wydruk -> i==3
   int i = 5;
   // wydruk -> i==5
}
// wydruk -> i==3
```

- Zasłanianie nazw może utrudniać rozumienie kodu
  - szczególnie używanie tych samych nazw w różnych blokach tej samej funkcji
  - staranne użycie nazw oznacza wykorzystanie znaczących, niepowtarzających się nazw
  - "niestaranne" użycie nazw może nie powodować problemów jeśli dotyczy zmiennych służących tylko jako pomocnicze zmienne wykorzystywane lokalnie
    - wtedy można umieścić ich definicje na początku funkcji

Argumenty funkcji

- Argumenty przesłane do funkcji
  - są traktowane jak zmienne lokalne
  - zmienne odpowiadające argumentom są inicjowane wartościami przesłanymi w wywołaniu funkcji
    - w C przesyłanie argumentów do funkcji ZAWSZE odbywa się przez wartość,
      - czyli przez kopiowanie wartości argumentu (aktualnego) w miejscu wywołania ...
      - ... do zmiennych (argumentów formalnych) w wywoływanej funkcji
- Wygodną ilustracją mechanizmu przesyłania argumentów do funkcji jest stos (systemowy)
  - stos jest stosowany w praktyce przez kompilatory do realizacji wywołania funkcji
    - szczegóły działania mogą różnić się od uproszczonego schematu
    - w rzeczywistości stosuje się różne warianty organizacji stosu, specjalne rejestry itp.

Struktura pamięci w C (umowny schemat)



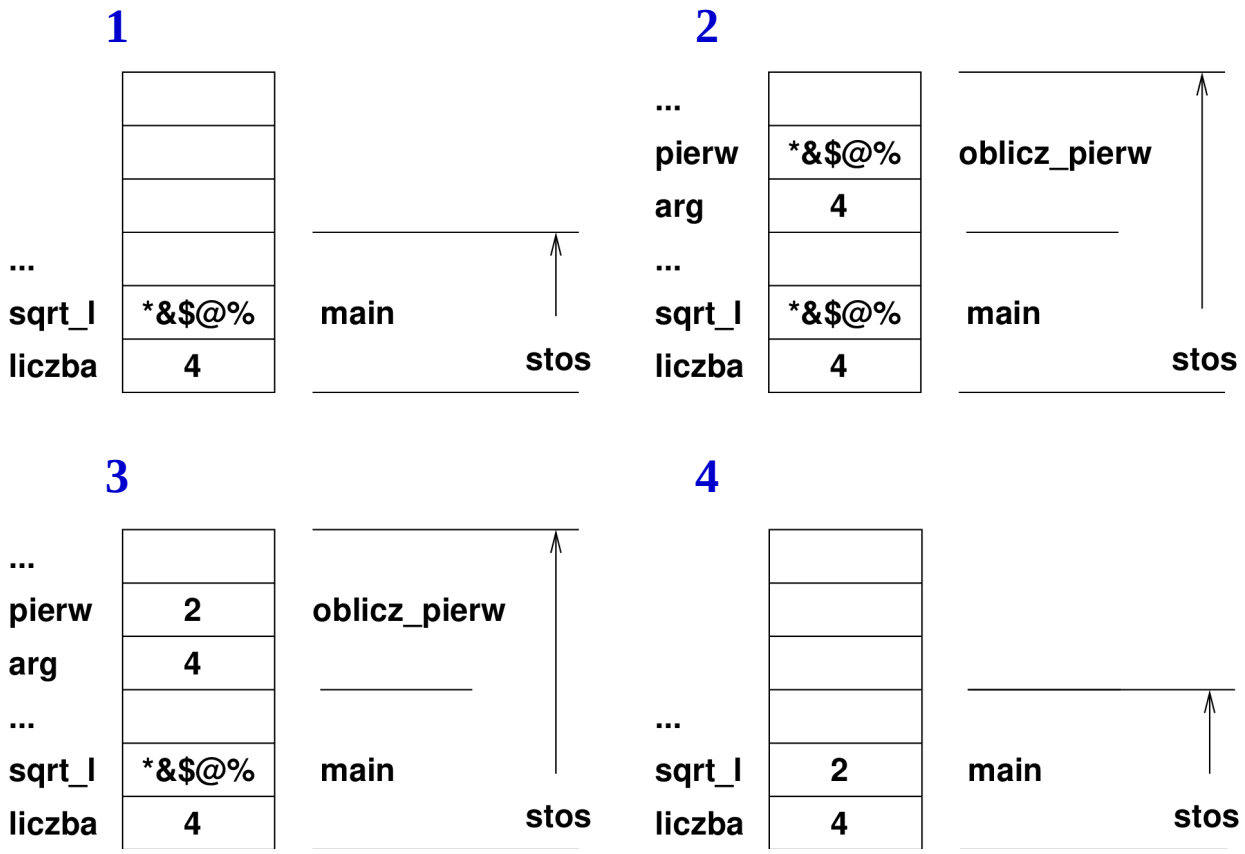

```
int a = 1;
int main(void){
  int b = 2;
  ... // definicje innych zmiennych
  printf("a=%d, b=%d\n", a, b);
  return;
}
```

- Stan pamięci w momencie wywołania funkcji printf
- Zmienne jako nazwane obszary pamięci
- Wartość zmiennej w jednej komórce, niezależnie od typu

Schemat stosu przy wywołaniu funkcji

Stan stosu w czterech kolejnych punktach w trakcie wykonania programu:

- w punktach 1 i 4 nie ma ramki dla funkcji oblicz_pierw




```
double oblicz_pierw( double arg)
{
   double pierw;
   // 2.
   // obliczenia pierwiastka
   pierw = .....
   // 3.
   return( pierw );
}

int main( void ) {
   double liczba = 4.0;
   double sqrt_l;
   // 1.
   sqrt_l = oblicz_pierw(liczba);
   // 4.
}
```

Rekurencja

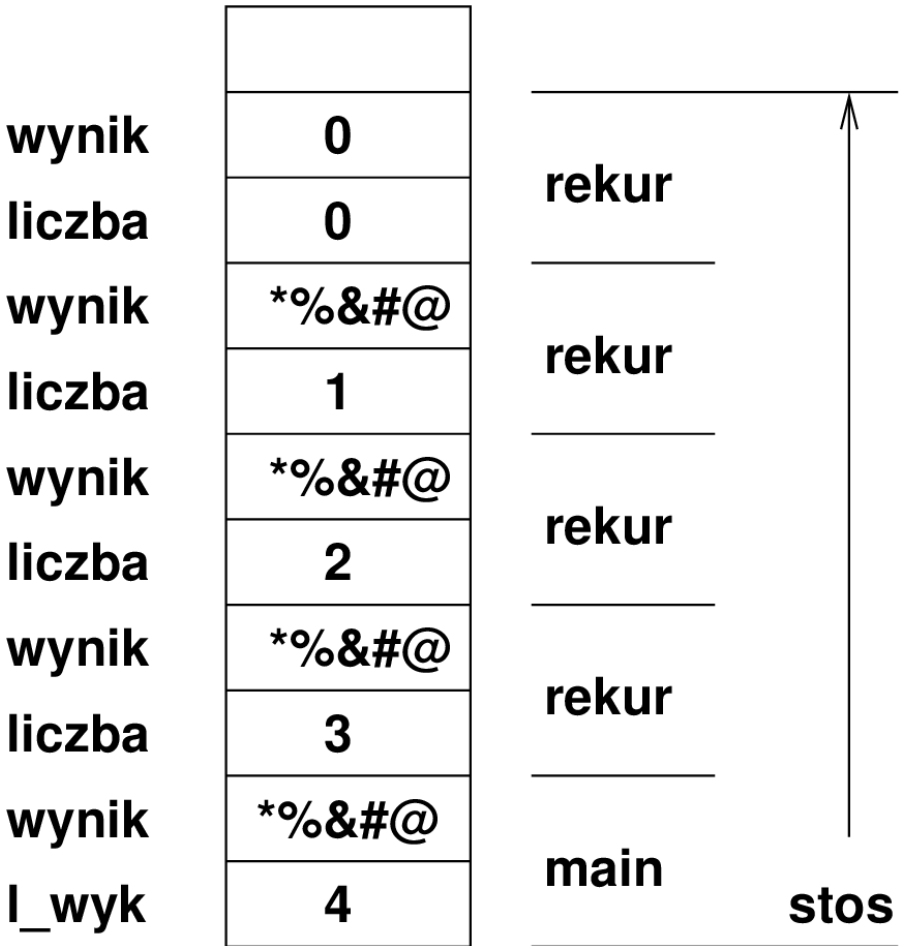
- W języku C funkcja może wywoływać sama siebie

```
int funkcja_rekur(int nr_wyw){
    printf("Wywołanie %d\n",nr_wyw);
    if(nr_wyw>1) return(funkcja_rekur (nr_wyw-1));
}
```

  - wywołanie takie nazywane jest rekurencją (rzadziej rekursją)
  - wywołania nie mogą trwać w nieskończoność
    - kolejne wywołania powinny zmieniać argumenty wywołania
    - mechanizm tych zmian i przetwarzania w funkcji rekurencyjnej powinien gwarantować zakończenie obliczeń po skończonej liczbie wywołań
- Znaczenie takiego wywołania jest dobrze opisywane poprzez mechanizm funkcjonowania stosu
- Wywołania rekurencyjne służą realizacji algorytmów rekurencyjnych

Struktura pamięci w C (umowny schemat)

- Stan pamięci w punkcie testowania przy największym rozmiarze stosu




```
int rekur( int liczba ){
   int wynik;
   liczba--;
   ... // operacja do wykonania
   if( liczba<1 ) wynik = 0;
   else wynik = rekur(liczba);
   // punkt testowania
   return( wynik );
}
int main(void){
   int l_wyk = 4;
   int wynik = rekur( l_wyk );
   if( wynik != 0 ) f_error( ... );
   return;
}
```

Algorytmy rekurencyjne

- Algorytmy rekurencyjne
  - obliczanie ciągu liczb Fibonacciego: $F_n = F_{n-1} + F_{n-2}$, $(F_0 = 0, F_1 = 1)$
  - wyszukiwanie binarne w tablicach posortowanych
    - dopóki tablica ma rozmiar większy od jeden
    - podziel tablicę na pół
    - jeśli środkowy wyraz mniejszy od wyszukiwanego
      - wyszukaj w prawej połowie
    - jeśli środkowy wyraz większy od wyszukiwanego
      - wyszukaj w lewej połowie
    - zwróć właściwy wynik
  - w algorytmach rekurencyjnych istotne są
    - zamiana jednego problemu w jeden lub wiele takich samych problemów dla innych danych wejściowych
    - opracowanie wyniku dla przypadków, kiedy dalsza rekurencja nie jest potrzebna/wskazana/możliwa

Sortowanie rekurencyjne

- Rekurencyjne algorytmy sortowania
  - sortowanie szybkie
    - dopóki rozmiar tablicy jest większy od 1
    - podziel tablicę na dwie części, tak aby wyrazy w lewej części były mniejsze niż wyrazy w prawej części
    - posortuj lewą część
    - posortuj prawą część
  - sortowanie przez scalanie
    - dopóki rozmiar tablicy jest większy od 1
    - podziel tablicę na dwie połowy
    - posortuj lewą połowę
    - posortuj prawą połowę
    - scal obie połowy, zachowując posortowanie elementów
  - rekurencyjne algorytmy sortowania należą do najszybszych
    - sortowanie szybkie posiada implementację *qsort* (*quick sort*) w bibliotece standardowej

Rekurencja

- Często algorytmy (funkcje) rekurencyjne można przekształcić w iteracyjne, zamieniając sekwencję wywołań w sekwencję iteracji
  - funkcje rekurencyjne często ułatwiają zrozumienie istoty implementowanego algorytmu i umożliwiają zwarty, elegancki zapis kodu
  - z punktu widzenia wydajności rekurencja, wymagająca intensywnej obsługi stosu, może prowadzić do nieefektywnych programów
  - przykład kodu obliczania silni:

wersja rekurencyjna:

```
int silnia(int n)
{
   if (n==0) return 1;
   else return n*silnia(n-1);
// return n == 0 ? 1 : n * silnia(n - 1);
}
```

wersja iteracyjna:

```
int silnia(int n)
{
   int s=n;
   while (--n) s*=n;
   return s;
}
```