Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 18.03.2020

4. Funkcje

Funkcje 1

- Funkcja to ciąg instrukcji, któremu nadaje się pewną nazwę i za pośrednictwem tej nazwy można się do niego odwołać i (wielokrotnie) wykonać
- Termin pochodzi z matematyki: funkcja odwzorowuje zbiór (iloczyn kartezjański zbiorów) w zbiór…
- W języku C:
  - Funkcja nie musi mieć argumentów
  - Funkcja nie musi mieć wartości
  - Funkcja może wywoływać efekty uboczne: np. obliczając wartości pewnej funkcji `int f(int x)` równocześnie zliczamy, ile razy `f()` była użyta

Funkcje 2

- Każda funkcja jest samodzielnym niewielkim programem z własnymi deklaracjami i instrukcjami
- Dzięki funkcjom można:
  - Unikać duplikowania kodu
  - Zwiększyć czytelność
  - Logicznie podzielić program na wykonywalne bloki o różnym poziomie abstrakcji
- W programie można użyć funkcji napisanych przez kogoś innego (ang. *reuse*)

Przykład: logowanie

```
int main()
{
    char login[256];
    char passwd[256];
    int i=0;
    int passwdOk=0;

    for(i=0;i<3;i++){
        printf("Enter login:");
        scanf("%s",login);
        printf("Enter passwd:");
        scanf("%s",passwd);
        // sprawdź czy wprowadzono poprawne dane
        if(!strcmp(login,"admin") && !strcmp(passwd,"admin")){
            passwdOk=1;
            break;
        }
    }

    if(passwdOk)printf("Welcome. Logged in\n");
    else printf("Account has been blocked");
    return 0;
}
```

`scanf` odczytuje dane ze standardowego wejścia i umieszcza we wskazanym miejscu

`strcmp` porównuje teksty (zwraca 0, jeżeli są identyczne)

Logowanie - checkPassword

```
int checkPassword()
{
    char login[256];
    char passwd[256];
    int passwdOk=0;
    int i;

    for(i=0;i<3;i++){
        printf("Enter login:");
        scanf("%s",login);
        printf("Enter passwd:");
        scanf("%s",passwd);
        // sprawdź czy wprowadzono poprawne dane
        if(!strcmp(login,"admin") && !strcmp(passwd,"admin")){
            passwdOk=1;
            break;
        }
    }
    return passwdOk;
}
```

`return passwdOk`
zwraca wyznaczoną wartość

Logowanie – wywołanie checkPassword

```
int main()
{
    int passwdOk=0;

    passwdOk = checkPassword();

    if(passwdOk)printf("Welcome. Logged in\n");
    else printf("Account has been blocked");

    return 0;
}
```

```
int checkPassword()
{
    // ...
    return 1;
}
```

- Ciąg instrukcji zostaje zastąpiony **wywołaniem** funkcji
- Deklaracje zmiennych `char login[256], char passwd[256], int i=0` mogą zostać przesunięte do wnętrza funkcji
- Ciąg instrukcji przekształcony w funkcję może zostać wykonany wielokrotnie, zamiast kopiować ciąg instrukcji i powtarzać deklaracje zmiennych – używa się nazwy funkcji
- Użytkownika funkcji nie interesuje, jak funkcja jest zaimplementowana, ważne jest, że spełnia założoną specyfikację.

Definicja funkcji 1

```
return_type function_name( parameters)
{
      declarations
      statements
}
```

Blok wewnątrz nawiasów klamrowych, to tzw. ciało funkcji (ang. *function body*)

- `return_type` – typ zwracanych wartości; jeżeli funkcja nie zwraca wartości – należy użyć słowa kluczowego `void`
  - Funkcje nie mogą zwracać tablic (ale mogą zwracać adresy tablic)
  - W standardzie ANSI C89 można ominąć typ zwracanej wartości (domyślnym typem jest `int`). W C99 typ jest wymagany
- `function_name` – nazwa funkcji (identyfikator)
- `parameters` – lista parametrów:

  `type1 param1, type2 param2,…`

Definicja funkcji 2

- Definicja funkcji pociąga za sobą:
  - umieszczenie jej kodu w segmencie kodu tzw. *text w* wynikowym pliku *object* (**.obj, *.o*) powstałym w wyniku kompilacji
  - Nazwa funkcji, typ zwracanej wartości i lista typów parametrów jest (zazwyczaj) umieszczona w tablicy symboli i jest widoczna dla konsolidatora
- Nie należy definiować funkcji wewnątrz funkcji (nawet jeżeli taki kod może zostać skompilowany i uruchomiony). To na ogół nie ma sensu. W językach C++/Java jest to niezgodne ze składnią.

```
int b(){
    void a(){
        printf("Function a() called");
    }
    a();
    return 0;
}

int main()
{
    b();
    return 0;
}
```

```
Function a() called
```

Definicja funkcji – przykłady 1

```
double odsetki(double kwota, int liczbaDni, double oproc)
{
        return kwota * (double) liczbaDni * oproc /36500;
}
```

- Zwracana wartość podawana jest po instrukcji `return;`
- Wartość powinna być zgodna z typem, ale też może zostać automatycznie przekonwertowana
- Zazwyczaj kompilator ostrzega, jeżeli funkcja nie jest typu `void` i przypadkowo nie zwraca wartości

```
int pi()
{
    return 3.14; // zamiast 3.14 zwróci 3
}

int foo(){
{
    // return 0;
}
```

Definicja funkcji – przykłady 2

```
void message(int errorCode)
{
        if(errorCode!=0){
                printf(”Error %d\n”, errorCode);
                return;
        }
        printf(”No error\n”)
}
```

- W przypadku funkcji niezwracających wartości wyjście z funkcji następuje po wykonaniu ostatniej instrukcji lub instrukcji `return`.
- Dodanie instrukcji `return` na końcu funkcji niezwracających wartości jest zbędne (ale poprawne).

```
void print(int x)
{
    printf("x=%d\n",x);
    return;
}
```

Definicja funkcji – przykłady 3

- Jeżeli funkcja nie ma parametrów, w języku C należy (można) zamiast listy parametrów podać `void`. Pozwala to na przeprowadzenie kontroli zgodności argumentów wywołania z definicją funkcji.

```
void about(void)
{
    printf("Program hello\n");
    printf("(c) Jan Kowalski\n");
}
```

```
int main() {
    about();
    return 0;
}
```

```
Program hello
(c) Jan Kowalski
```

```
int main(){
    about(1,2,3,4);
    return 0;
}
```

```
main.c:56: error: too
many arguments to
function `about'
```

Definicja funkcji – przykłady 4

- Pozostawienie pustej listy argumentów oznacza, że niczego nie wiadomo o parametrach funkcji. Każde wywołanie jest dobre!
- W C++ nie jest to konieczne!

```
void about(/* brak void */)
{
    printf("Program hello\n");
    printf("(c) Jan Kowalski\n");
}
```

```
int main() {
    about();
    return 0;
}
```

```
Program hello
(c) Jan Kowalski
```

```
int main(){
    about(1,"Ala ma kota",3.2,'c');
    return 0;
}
```

```
Program hello
(c) Jan Kowalski
```

Wywołanie funkcji 1

- Wywołanie funkcji ma postać nazwy funkcji, po której w nawiasach podawana jest lista argumentów oddzielona przecinkami.
- Zamiast argumentów można użyć wyrażeń odpowiedniego typu.
- Jeżeli lista formalnych parametrów funkcji jest pusta, lista argumentów powinna być również pusta.
- Wywołanie funkcji musi być umieszczone wewnątrz innej funkcji.

```
double odsetki(double kwota, int liczbaDni, double oproc)
{
        return kwota * (double) liczbaDni * oproc /36500;
}

int main()
{
    double o = odsetki(1000,30,5);
    printf("%f",o);
    return 0;
}
```

Wywołanie funkcji 2

- Jeżeli funkcja nie zwraca wartości, wówczas niemal zawsze wywołanie funkcji stanowi pojedynczą instrukcję.

```
void o_lokacie(double kwota, int liczbaDni, double oproc)
{
    printf("Lokata %.2f, oproc.: %.2f",kwota, oproc);
}

int main()
{
    o_lokacie(1000, 30, 5.2);
    return 0;
}
```

Wywołanie funkcji 3

- Jeżeli funkcja zwraca wartość, jej rezultat może zostać użyty do konstrukcji wyrażeń.

```
double odsetki(double kwota, int liczbaDni, double oproc)
{
        return kwota * (double) liczbaDni * oproc /36500;
}

int main()
{
    int i=0;
    double k = 1000;
    for(i=1;i<=12;i++){
        k=k+odsetki(k,30,5);
        printf("Miesiac %d %.2f\n",i,k);
    }
    return 0;
}
```

```
Miesiac 1 1004.11
Miesiac 2 1008.24
Miesiac 3 1012.38
Miesiac 4 1016.54
Miesiac 5 1020.72
Miesiac 6 1024.91
Miesiac 7 1029.12
Miesiac 8 1033.35
Miesiac 9 1037.60
Miesiac 10 1041.86
Miesiac 11 1046.15
Miesiac 12 1050.45
```

Funkcja główna 1

- W programach w języku C/C++ zawsze występuje funkcja główna, od której rozpoczyna się wykonanie programu.
- W zależności od platformy funkcja ta nazywa się: `main`, `wmain`, `WinMain`
- W szczególnych przypadkach funkcja główna może być umieszczona w zewnętrznej bibliotece. W takim przypadku programista nie musi jej implementować.

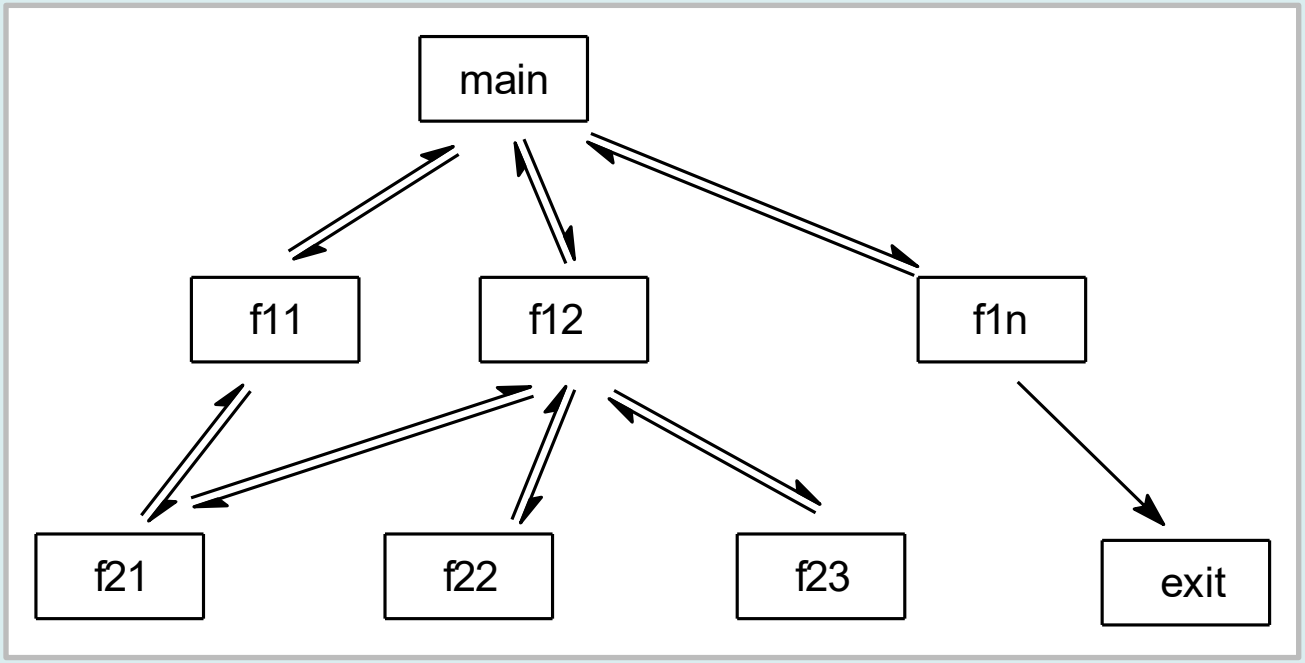
Funkcja główna 1

- W programach w języku C/C++ zawsze występuje funkcja główna, od której rozpoczyna się wykonanie programu.
- W zależności od platformy funkcja ta nazywa się: `main`, `wmain`, `WinMain`
- W szczególnych przypadkach funkcja główna może być umieszczona w zewnętrznej bibliotece. W takim przypadku programista nie musi jej implementować.
- Zazwyczaj biblioteka zawierająca funkcję główną oferuje specyficzne rozwiązanie (np. interfejs użytkownika). Nazywana jest wtedy szkieletem/platformą (ang. *framework*)

Funkcja główna 2

- Program kończy działanie po
  - ostatniej instrukcji funkcji `main`
  - lub po instrukcji `return`.
- Wyjątkiem jest wykonanie funkcji bibliotecznych `exit` lub `abort`, które powodują przerwanie wykonania programu w momencie ich wywołania.
- Wykonanie programu ma polega na poruszaniu się w głąb drzewa funkcji. Z funkcji `main` wołane są funkcje pierwszego poziomu, z nich kolejne funkcje drugiego poziomu, itd.

Drzewo wywołań funkcji




```
void f11()
{
        f21();
}
...
void f1n(int x)
{
        if(x<0)exit();
}
```

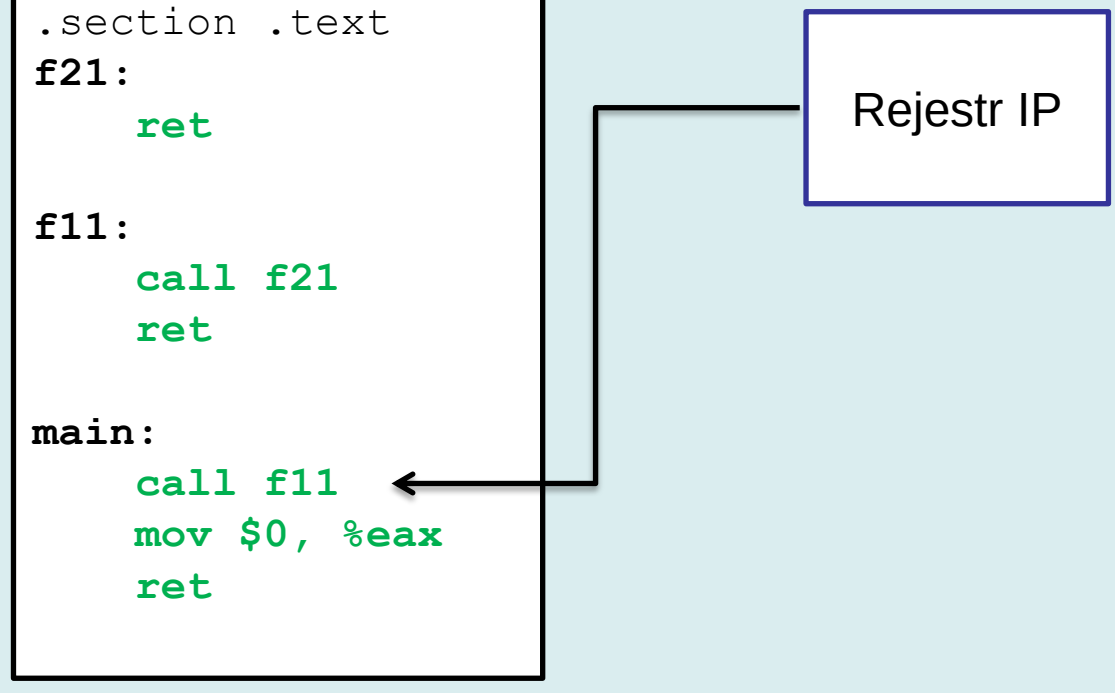
```
void main()
{
        f11();
        f12();
        f1n(2);
}
```

Stos wywołań funkcji 1

- Jak przebiega wywołanie `main` → `f11` → `f21`

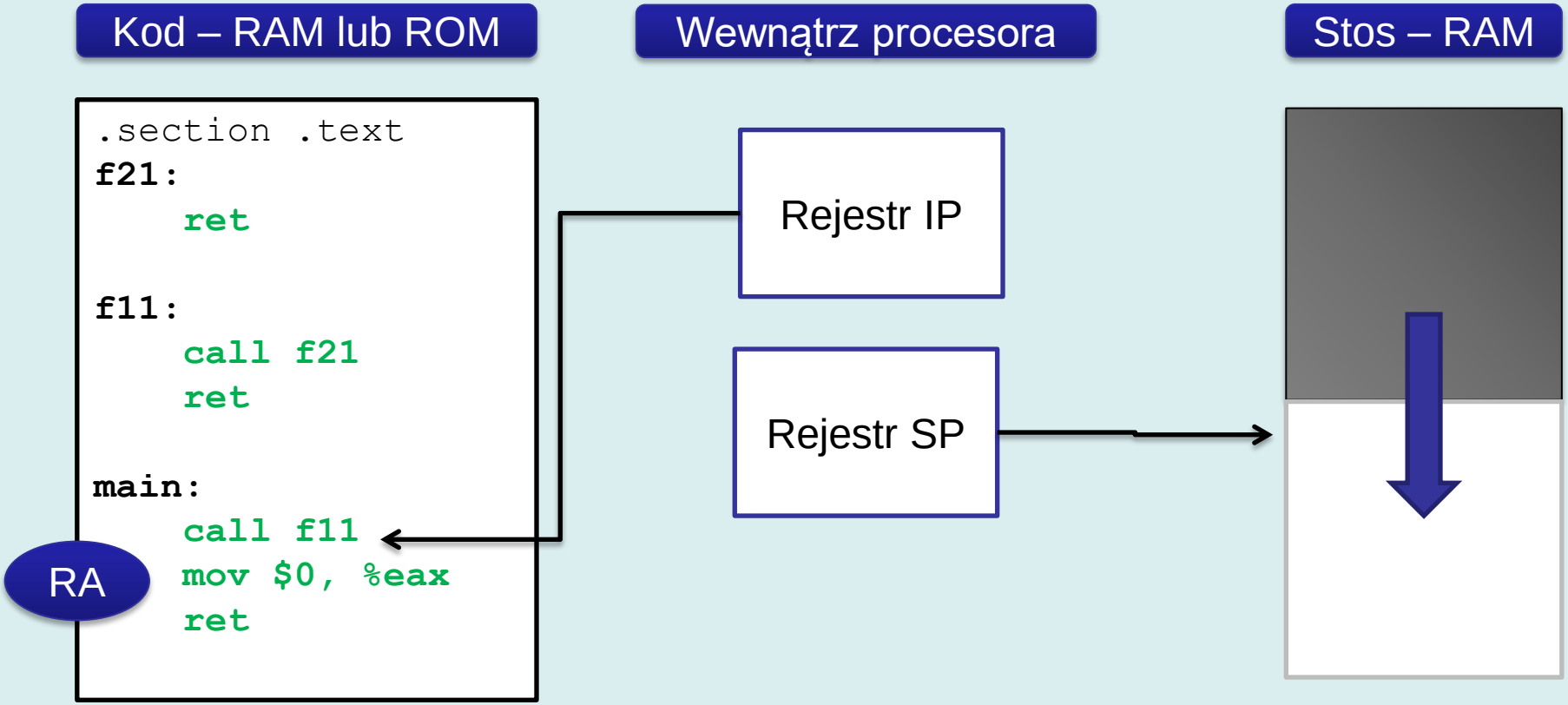
```
void f21()
{
}

void f11()
{
        f21();
}

int main()
{
        f11();
        return 0;
}
```

```
.section .text
f21:
    ret

f11:
    call f21
    ret

main:
    call f11
    mov $0, %eax
    ret
```



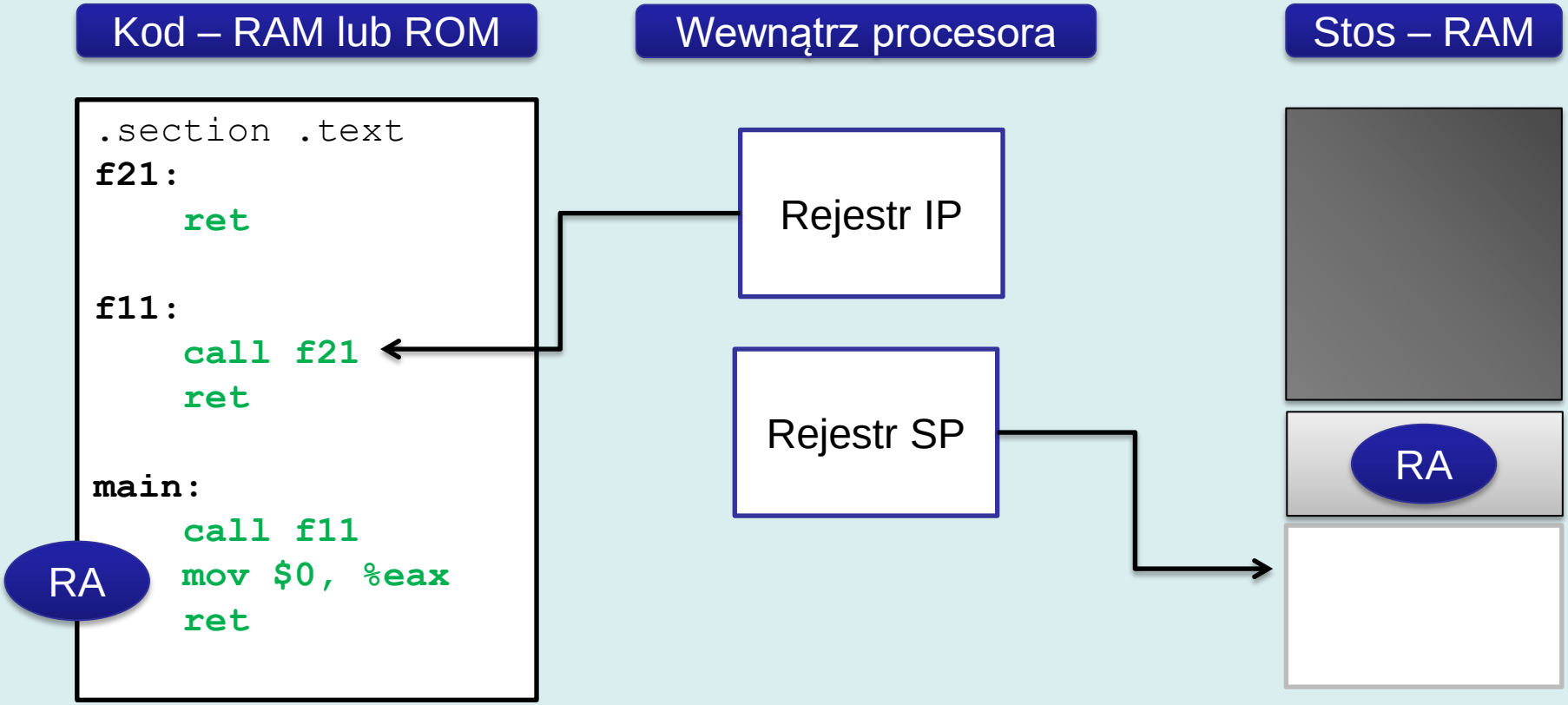
- Kod programu załadowany jest do pamięci RAM. Może też być na stałe umieszczony w pamięci ROM.
- Rejestr IP (ang. *instruction pointer*) zawiera adres instrukcji do wykonania

Stos wywołań funkcji 2



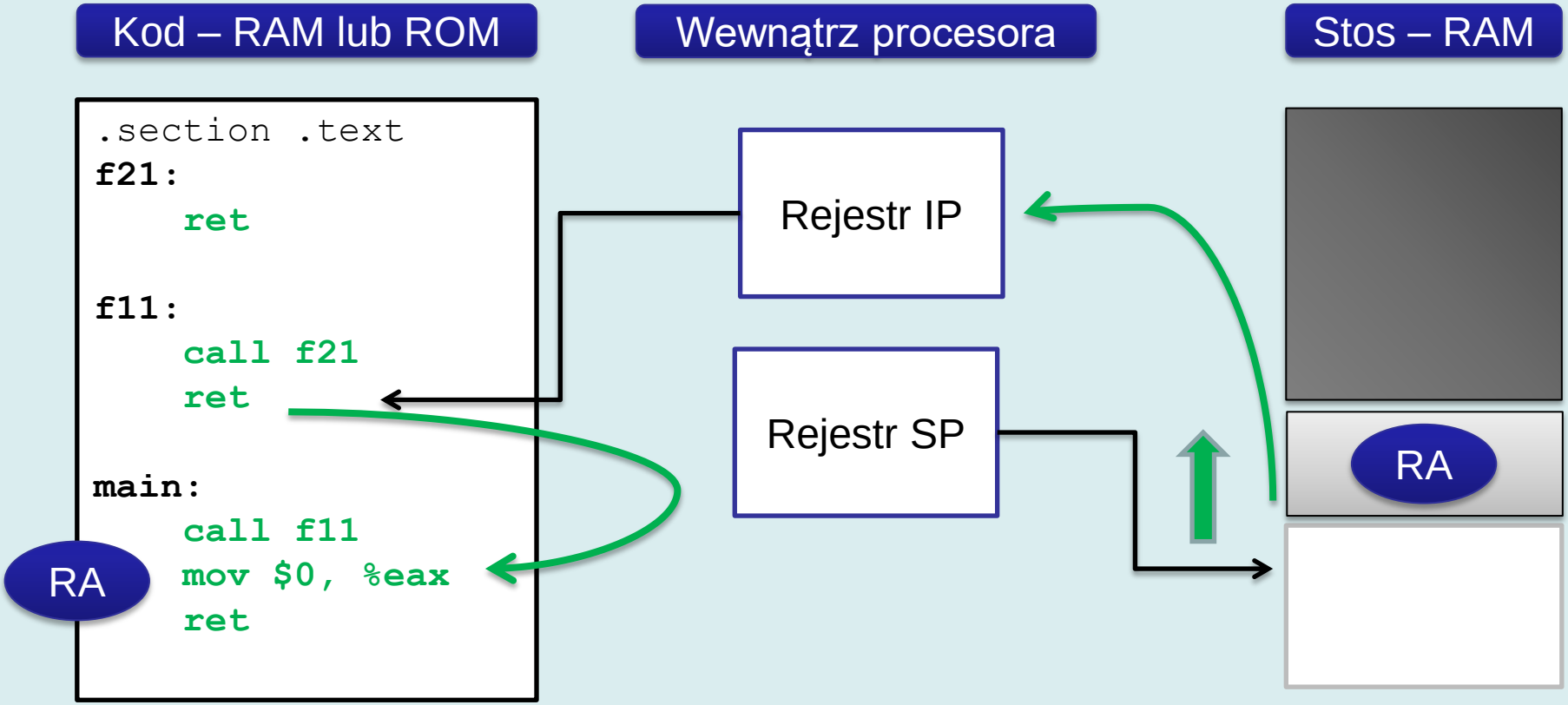
- **Przed wykonaniem rozkazu `call f11` rejestr IP zawiera jego adres.**
- **Rejestr SP wskazuje wolne miejsce na stosie (dodawanie elementów na stosie zmniejsza wartość rejestru)**

Stos wywołań funkcji 3



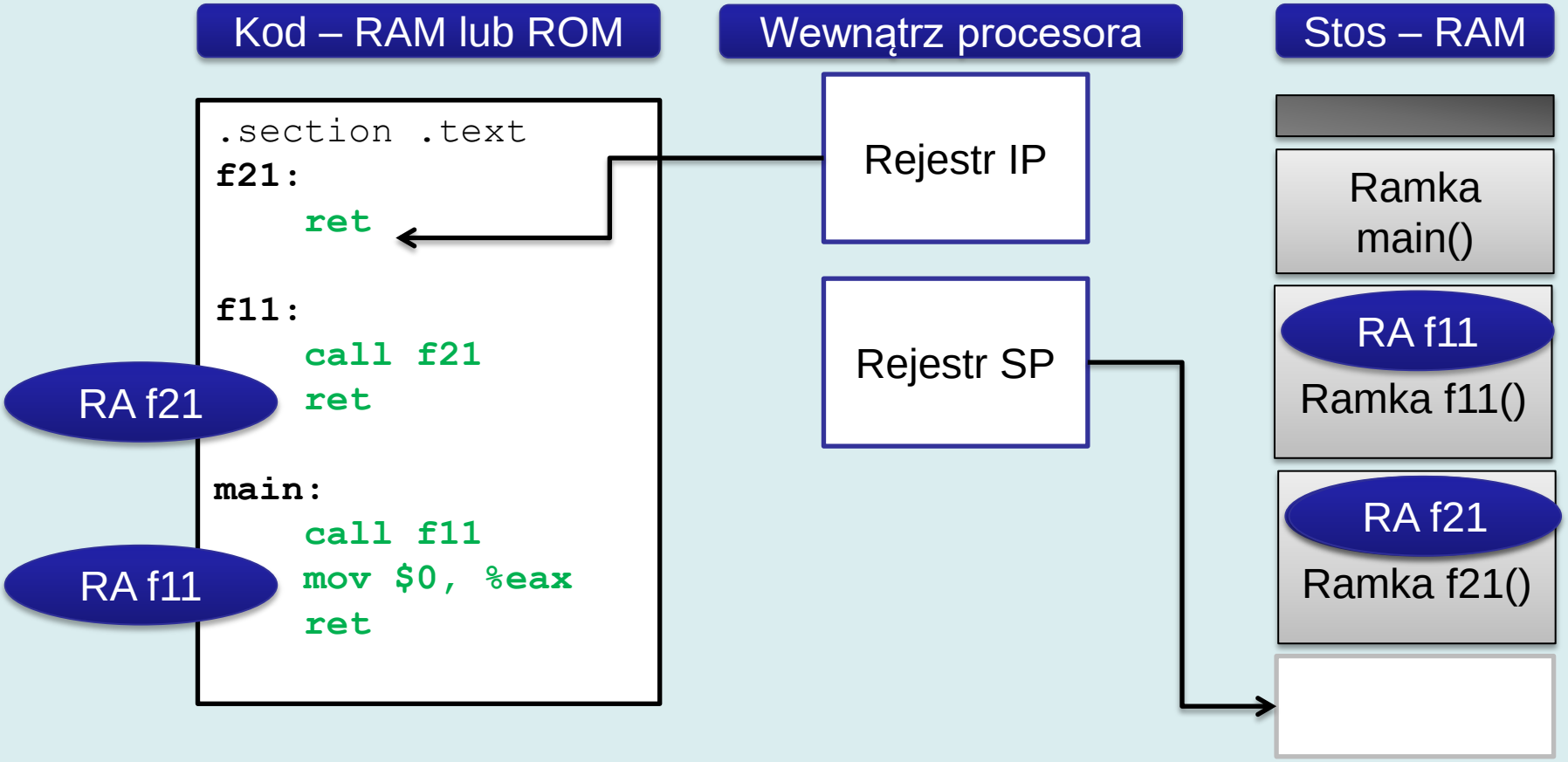
- Po wykonaniu rozkazu `call f11` rejestr IP zawiera adres pierwszego rozkazu funkcji f11.
- Na stosie został umieszczony adres powrotu RA

Stos wywołań funkcji 4



- W wyniku wykonania rozkazu `ret` :
- Adres powrotu RA zostanie pobrany ze stosu i załadowany do rejestru IP – będzie on wskazywał następną instrukcję po wywołaniu funkcji
- Wskaźnik stosu zostanie przesunięty w górę

Stos wywołań funkcji 5



- Podczas wywołania i wykonania funkcji na stosie odkładane są dane: adres powrotu, użyte argumenty wywołania, zmienne deklarowane lokalnie, także informacje dla debuggera. Jest to tzw. ramka funkcji.
- Ciąg ramek funkcji na stosie reprezentuje drogę w drzewie wywołań funkcji – od korzenia `main()` poprzez `f11()` do aktualnie wykonywanej funkcji `f21()`
- Funkcje `abort()` i `exit()` cofając SP przeskakują wszystkie ramki

Funkcje z parametrami 1

- **Parametry** funkcji pojawiają się w definicji funkcji. Są to zmienne, np.: **`double kwota, int liczbaDni`** – *formalne parametry funkcji*
- **Argumenty** to wyrażenia, które pojawiają się w wywołaniach funkcji – *argumenty wywołania*

```
double odsetki( double kwota, int liczbaDni, double oproc  )
{
      return kwota * (double) liczbaDni * oproc /36500;
}

int main()
{
    double k = 1000;
    int m=3;
    k=k+odsetki(k, m*30 ,5);
    return 0;
}
```

Funkcje z parametrami 2

- Argumenty do funkcji przekazywane są przez wartość:
- Obliczane są wartości wszystkich wymaganych argumentów, zazwyczaj od lewej do prawej
- Jeżeli w celu obliczenia wartości argumentu należy wywołać inną funkcję – będzie ona wywołana wcześniej

```
double pole(double a,double b)
{
    return a*b;
}

void prostopadloscian(double a, double b, double c)
{
    printf("Pole: %f Objętosc: %f \n",
            2*pole(a,b)+2*pole(b,c)+2*pole(a,c),
            pole(a,b)*c);
}
```

Funkcje z parametrami 3

- W przypadku zagnieżdżonych wywołań – wpierw będą wywołane umieszczone najgłębiej

```
double odsetki(double kwota, int liczbaDni, double oproc)
{
        return kwota * (double) liczbaDni * oproc /36500;
}

int main()
{
    double k = 1000;
  3 k=k+odsetki(
          2 k+odsetki(
              1 k+odsetki(k,30,5)
            ,30,5)
        ,30,5);
    printf("Miesiac %d %.2f\n",3,k);

    return 0;
}
```

Dlaczego wynik jest zły?
Powinno być:
`Miesiac 3 1012.38`

```
Miesiac 3 1004.13
```

Funkcje z parametrami 4

- Poprawiona wersja…

```
double kapitalizacja(double kwota, int liczbaDni, double oproc)
{
        return kwota + kwota * (double) liczbaDni * oproc /36500;
}


int main()
{
    double k = 1000;
    k=kapitalizacja(
            kapitalizacja(
                kapitalizacja(k,30,5)
            ,30,5)
        ,30,5);
    printf("Miesiac %d %.2f\n",3,k);

    return 0;
}
```

Dlaczego tym razem wynik jest poprawny?

```
Miesiac 3 1012.38
```

Funkcje z parametrami 5

- W jaki sposób argumenty wywołania są przekazywane do funkcji?
  - Formalne parametry funkcji są zmiennymi. Pamięć dla nich jest przydzielana na **stosie**
  - Zmienne te nie są nigdy jawnie inicjalizowane wartościami (jawnie nie nadaje im się wartości początkowych)
  - Za prawidłową inicjalizację odpowiada mechanizm wywołania funkcji – w momencie, kiedy rozpoczyna się wykonanie funkcji (rejestr IP wskazuje pierwszy rozkaz) zmienne mają wartości będącymi argumentami wywołania.

Funkcje z parametrami 5

- W wynikowym kodzie maszynowym dodawany jest niewidoczny kod: przed wywołaniem **prolog**, a po powrocie **epilog.**
- W prologu – na stosie umieszczane są argumenty wywołania
- W epilogu są one usuwane

```
double odsetki( double kwota, int liczbaDni, double oproc  )
{
       return kwota * (double) liczbaDni * oproc /36500;
}

int main()
{
    double k = 1000;
    int m=3;

    prolog
    k=k+odsetki(k, m*30 ,5);
    epilog
    return 0;
}
```

Funkcje z parametrami 6

```
int main()
{
    double k = 1000;
    int m=3;

    prolog

    k=k+odsetki(k, m*30 ,5);

    epilog

    return 0;
}
```

Prolog:

1. Opcjonalnie – zarezerwuj na stosie miejsce na zwracaną wartość.
2. Umieść na stosie 5
3. Oblicz m*3 i umieść wartość na stosie
4. Umieść wartość k na stosie

Epilog:

1. Usuń ze stosu 3 wartości : 8-bajtową (double), 4-bajtową (int) i 8-bajtową.
2. Skopiuj do zmiennej k wartość zwracaną przez stos lub rejestr procesora
3. Jeżeli na stosie było zarezerwowane tymczasowe miejsce na zwracaną wartość – zwolnij je

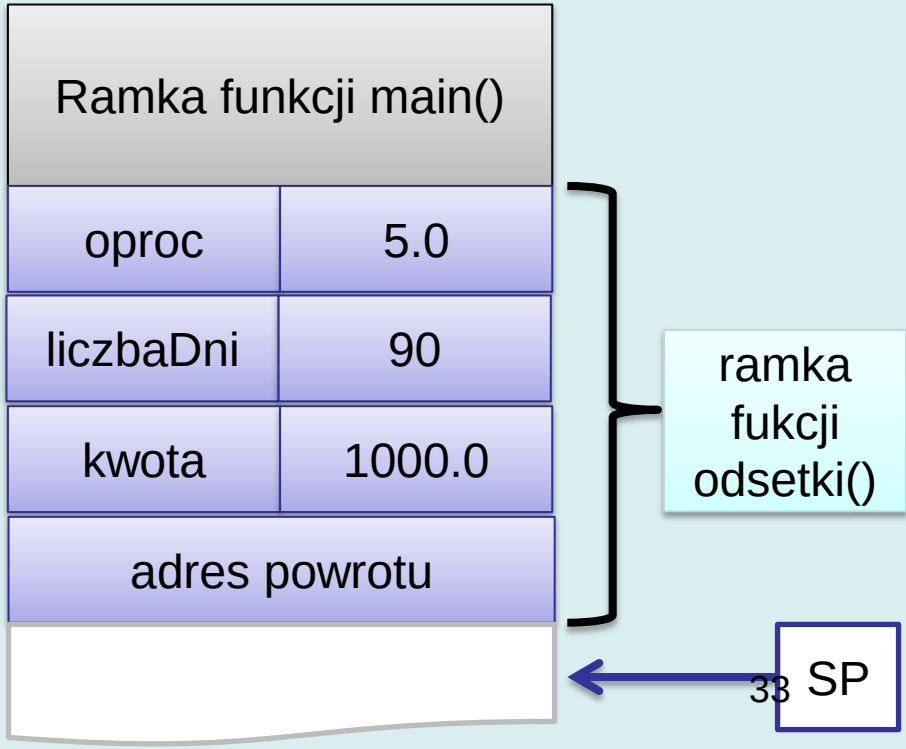
Uwaga istnieją dwa typy wywołań:

- cdecl – usuwanie argumentów ze stosu następuje po powrocie z funkcji
- stdcall – funkcja odpowiada za usuwanie argumentów

Funkcje z parametrami 7

```
double odsetki( double kwota, int liczbaDni, double oproc  )
{
        return kwota * (double) liczbaDni * oproc /36500;
}
int main(){...
k=k+odsetki(k, m*30 ,5);
...}
```

- Wewnątrz funkcji `odsetki()` pamięć na stosie jest widziana jako zmienne o wartościach nadanych podczas wywołania funkcji.
- Z punktu widzenia kodu zawierającego wywołania funkcji – nazwy formalnych parametrów są nieistotne.



Deklaracje funkcji

- Jeżeli kompilator odpowiada za umieszczenie w kodzie wynikowym prologu i epilogu wokół wykonania funkcji, *powinien* mieć informacje:
  - Jaki jest **typ parametrów** (jakie rozmiary mają dane, które mają zostać umieszczone na stosie)
  - Jaka jest **liczba** parametrów
  - W jakiej **kolejności** należy je przekazać
  - Jaki jest **typ zwracanej wartości**
- Jeżeli definicja funkcji jest umieszczona przed wywołaniem – te informacje są znane.
- Jeżeli umieszczona jest po wywołaniu, w innym module lub bibliotece – wszelkich informacji o funkcji dostarcza jej **deklaracja**

Składnia deklaracji

```
return_type function_name( parameters );
```

Tak, jak w przypadku definicji:

- `return_type` - jest typem zwracanej wartości (także `void`)
- `function_name` - jest nazwą funkcji (identyfikatorem)
- `parameters` – oddzielona przecinkami lista parametrów
  - Lista może być pusta – brak informacji o parametrach
  - Lista może zawierać wyłącznie słowo kluczowe `void`
  - Lista może zawierać wyłącznie typy
  - Lista może zawierać typy i nazwy

Przykłady deklaracji

```
void wypisz(double a);

int wczytaj();

int wczytaj(void);

double odsetki( double kwota, int liczbaDni, double oproc  );

double odsetki( double k, int ld, double op  );

double odsetki( double, int, double);
```

- Deklaracja funkcji nazywana jest także prototypem. Prototyp definiuje interfejs funkcjonalny, ale nie podaje implementacji.
- Nazwy zmiennych (formalnych parametrów) w deklaracji nie muszą pokrywać się z nazwami w definicji.
- Nazwy te nie mają żadnego znaczenia dla kompilatora, raczej są narzędziem dokumentowania kodu.
- Można je pominąć…

Pliki nagłówkowe

- Deklaracje (prototypy) funkcji umieszcza się w plikach nagłówkowych (rozszerzenie .h)
- Pliki nagłówkowe użytkownika włącza się do jednostki kompilacji za pomocą dyrektywy
  `#include "myfunctions.h"`
- W podanej nazwie może wystąpić dowolne wyrażenie będące akceptowaną w systemie operacyjnym ścieżką pliku:
  `#include "c:/c-programs/myfunctions.h"`
- Pliki nagłówkowe dostarczane razem z bibliotekami zazwyczaj umieszczone są w znanym katalogu identyfikowanym przez zmienną środowiskową `INCLUDE`. Podajemy je stosując ostre nawiasy.
  `#include <stdio.h>`
-

Generacja kodu wywołania

Kompilator generując kod wywołania funkcji postępuje w zależności od sytuacji:

- Jeżeli definicja lub prototyp funkcji **były widoczne**, wartości argumentów są automatycznie konwertowane do wymaganego typu.
  Na przykład, jeżeli argument typu `int` był przekazany w miejsce parametru typu `double`, nastąpi automatyczna konwersja do postaci zmiennoprzecinkowej.
- Jeżeli deklaracja (definicja) **nie była widoczna**, stosowana jest tzw. **promocja typu**:
  - wartości `float` konwertowane są do `double`
  - Wartości `char`, `short` konwertowane są do `int`

Efekty uboczne 1

- Efekt uboczny funkcji to potencjalna modyfikacja obiektów (zmiennych) zdefiniowanych poza funkcją.
- Mechanizm kopiowania wartości wywołania na stos nie powoduje efektów ubocznych w postaci modyfikacji wartości zmiennych pojawiających się w wywołaniu.

```
int foo(int a){
    printf("a w foo = %d\n",a);
    a++;
    printf("a w foo = %d\n",a);
    return a;
}
int main()
{
    int a=1;
    printf("a w main = %d\n",a);
    printf("Rezultat foo = %d\n",foo(a));
    printf("a w main = %d\n",a);
    return 0;
}
```

```
a w main = 1
a w foo = 1
a w foo = 2
Rezultat foo = 2
a w main = 1
```

Efekty uboczne 3

- Dobrze zaprojektowany program powinien minimalizować liczbę zmiennych globalnych.
- Jeżeli piszemy niewielki program (prototyp) wprowadzenie zmiennej globalnej (np.: dwuwymiarowej tablicy) i napisanie szeregu funkcji do odczytu, zapisu, obliczeń jest uzasadnione (czas kodowania).
- Tam gdzie to możliwe – należy jednak unikać funkcji działających na zewnętrznych obiektach, ponieważ może to powodować błędy. Śledzenie zależności w kodzie może być trudne.
- Nigdy nie należy używać globalnych zmiennych, do iteracji.

Efekty uboczne 4

Ten kod działa poprawnie

```
Liczba pierwsza (0): 2
Liczba pierwsza (1): 3
Liczba pierwsza (2): 5
Liczba pierwsza (3): 7
Liczba pierwsza (4): 11
Liczba pierwsza (5): 13
Liczba pierwsza (6): 17
Liczba pierwsza (7): 19
Liczba pierwsza (8): 23
Liczba pierwsza (9): 29
```

```
int i,n;

int isPrime(int number)
{
    for(i=2;i*i<=number;i++)
    {
        if(number%i==0)return 0;
    }
    return 1;
}

int main(int argc, char** argv) {
    int number=2;

    for(n=0;n<10;){
        if(isPrime(number)){
            printf("Liczba pierwsza (%d): %d\n",n,number);
            n++;
        }
        number++;
    }
    return 0;
}
```

Efekty uboczne 5

Ten nie w pełni poprawnie

```
int i;

int isPrime(int number)
{
    for(i=2;i*i<=number;i++)
    {
        if(number%i==0)return 0;
    }
    return 1;
}

int main(int argc, char** argv) {
    int number=2;

    for(i=0;i<10;){
        if(isPrime(number)){
            printf("Liczba pierwsza (%d): %d\n",i,number);
            i++;
        }
        number++;
    }
    return 0;
}
```

```
Liczba pierwsza (2): 2
Liczba pierwsza (2): 3
Liczba pierwsza (3): 5
Liczba pierwsza (3): 7
Liczba pierwsza (4): 11
Liczba pierwsza (4): 13
Liczba pierwsza (5): 17
Liczba pierwsza (5): 19
Liczba pierwsza (5): 23
Liczba pierwsza (6): 29
Liczba pierwsza (6): 31
Liczba pierwsza (7): 37
Liczba pierwsza (7): 41
Liczba pierwsza (7): 43
Liczba pierwsza (7): 47
Liczba pierwsza (8): 53
Liczba pierwsza (8): 59
Liczba pierwsza (8): 61
Liczba pierwsza (9): 67
```

Efekty uboczne 5

```
int main(int argc, char** argv) {
    int tab[1000][1000];
    printf("I am running...");
    return 0;
}
```

```
RUN FAILED (exit value 1, total time: 65ms)
```

Wyczerpanie stosu uniemożliwia uruchomienie programu.

```
int tab[1000][1000];

void fill(int val)
{
    int i,j;

    for(i=0;i<1000;i++)
        for(j=0;j<1000;j++)
            tab[i][j]=val;
}

int main(int argc, char** argv) {
    printf("I am running...");
    fill(0);
    return 0;
}
```

Użyteczność funkcji `fill()` polega na wykorzystaniu efektu ubocznego (działanie na globalnej tablicy)

```
I am running...
RUN SUCCESSFUL (total time: 48ms)
```

Funkcje rekurencyjne

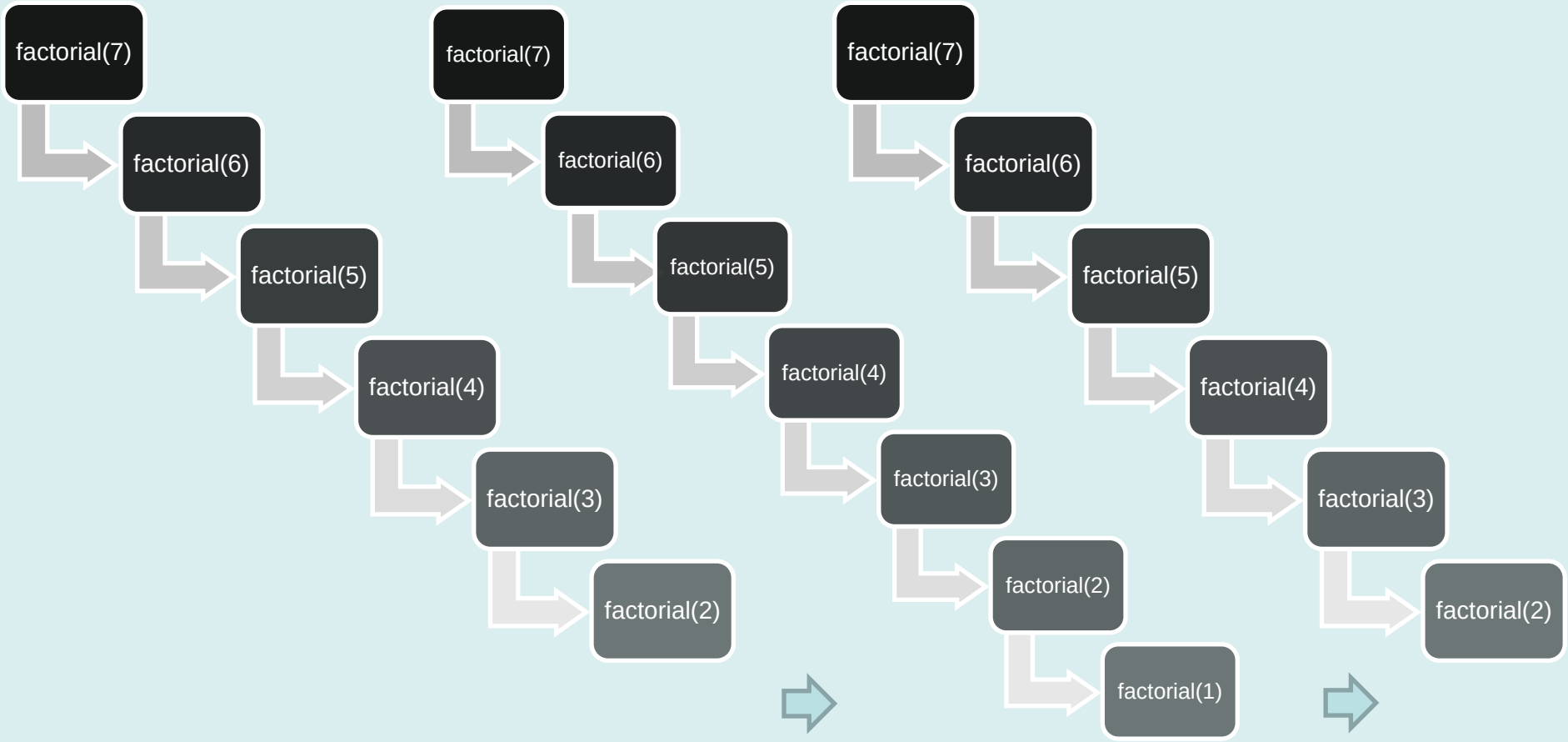
- Funkcja rekurencyjna to funkcja, która wywołuje sama siebie (bezpośrednio lub pośrednio).
- Każde wywołanie jest realizowane dla odrębnego zbioru argumentów
- Poprzednie wywołania czekają na powrót z wywołań potomnych…
- Przez ten czas muszą być przechowywane argumenty wywołań oraz zadeklarowane zmienne

Przykład - silnia

```
int factorial(int n)
{
    if(n<2)return 1;
    return n*factorial(n-1);
}

int main(int argc, char** argv) {
    printf("%d! = %d\n",7,factorial(7));
    return 0;
}
```

```
7! = 5040
```

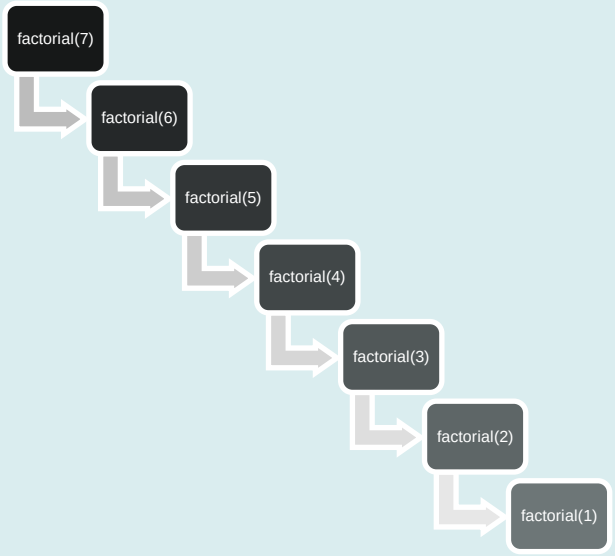
Przykład - silnia



Po wywołaniu funkcji następuje zejście niżej (funkcja jest wołana z nowymi wartościami argumentów) lub powrót.
Ponieważ nie wiadomo z góry, kiedy zakończy się schodzenie w głąb – konieczna jest struktura danych pozwalająca na przechowywanie „nieskończonej” liczby elementów.

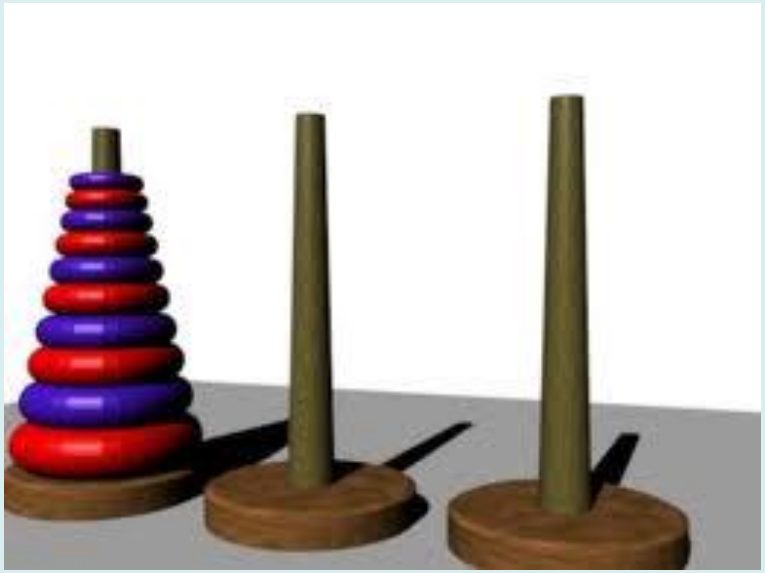
Rekurencja

- W wielu sytuacjach nie musi być stosowana: obliczanie silni może zostać rozwiązane za pomocą prostej iteracji.
- Zamiast ścieżki wywołań funkcji – pętla
- Rekurecja jest celowa w *trudniejszych* przypadkach




```
int factorial(int n)
{
    if(n<2)return 1;
    return n*factorial(n-1);
}

int factorial_i(int n)
{
    int i;
    int f=1;
    for(i=2;i<=n;i++)f=f*i;
    return f;
}
```

Przykład – wieże Hanoi 1

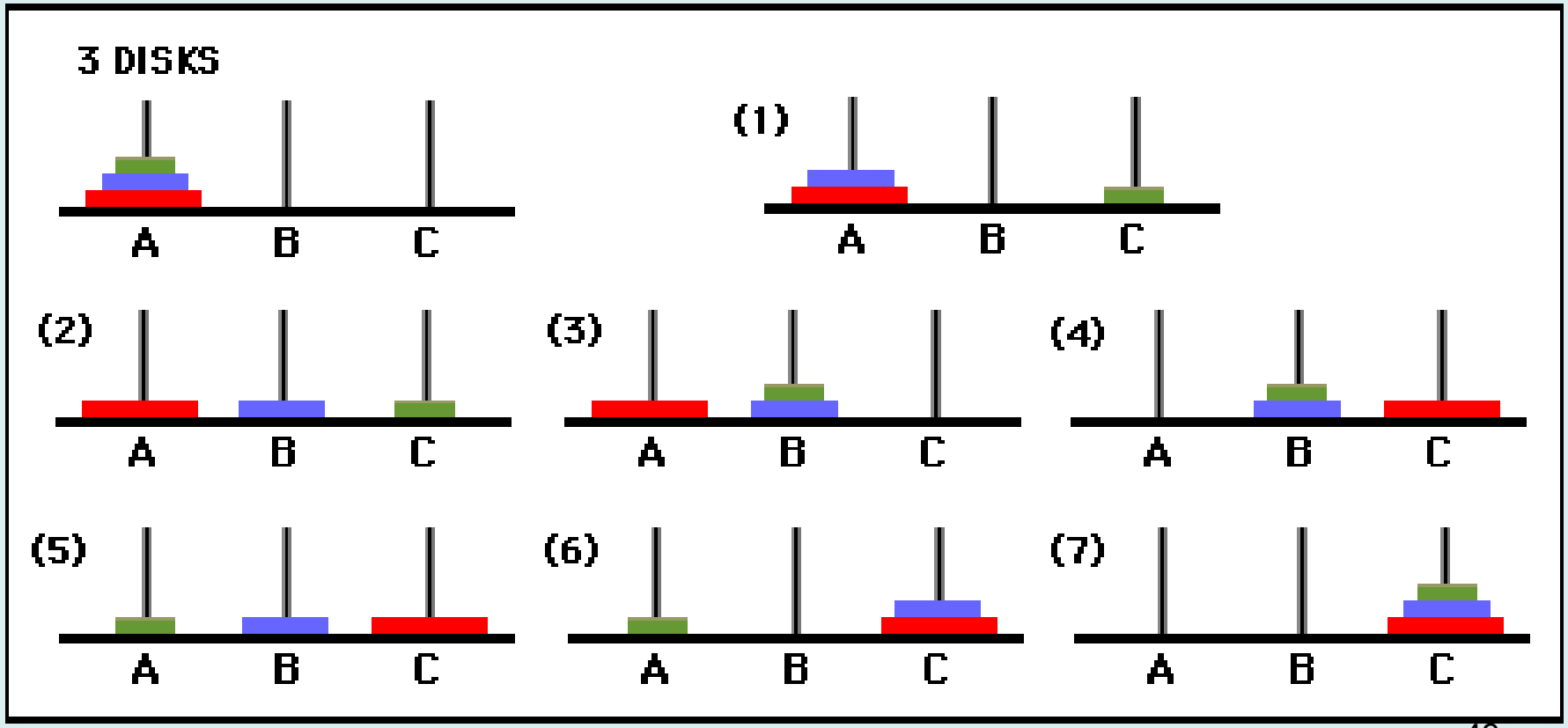


http://mathforum.org/mathimages/index.php/Image:10_Ring_Hanoi.jpg

- Należy przenieść stos krążków z pierwszego pręta na ostatni
- Można przenosić tylko jeden krążek na raz
- Nie można umieszczać większych krążków na mniejszych

Przykład – wieże Hanoi 2

- Rozwiązanie dla 3 krążków
- Dla n krążków - liczba ruchów $2^n - 1$

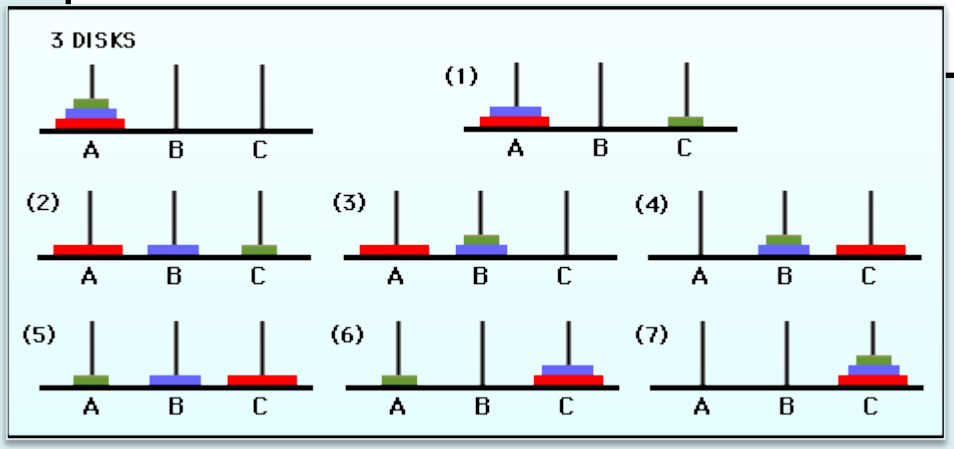


http://samztamang.blogspot.com/2012/07/tower-of-hanoi-solution.html

Przykład – wieże Hanoi 3

```
void move (int n, int src, int dest, int tmp) {
    if(n==1){
        printf ("Move one disk from %d to %d\n", src, dest);
    }else{
        move (n-1, src, tmp, dest);
        printf ("Move one disk from %d to %d\n", src, dest);
        move (n-1, tmp, dest, src);
    }
}

int main() {
    move (3, 1, 3, 2);
    return 0;
}
```

```
Move one disk from 1 to 3
Move one disk from 1 to 2
Move one disk from 3 to 2
Move one disk from 1 to 3
Move one disk from 2 to 1
Move one disk from 2 to 3
Move one disk from 1 to 3
```



Implementacja funkcji rekurencyjnych 1

1. Pamięć dla argumentów wywołania, wartości zwracanych przez funkcję oraz zmiennych automatycznych jest przydzielana **na stosie**.
   - Przydział następuje w momencie wywołania funkcji (dodatkowy kod wygenerowany przez kompilator: operacje PUSH umieszczające wartości na stosie)
   - Dodatkowo, przed wywołaniem funkcji mogą być zapisywane na stosie wartości rejestrów.
2. Wywołanie funkcji
3. Zwolnienie po powrocie z funkcji (operacje POP usuwające dane ze stosu)

Implementacja funkcji rekurencyjnych 2

```
int factorial(int n)
{
     int f_n_1;
     register unsigned esp __asm__ ("esp");

     printf("\nAdress n %u \n",&n);
     printf("factorial( %d ) -- stack: %u \n",n,esp);

     if(n<2)return 1;
     f_n_1 = factorial(n-1);
     return  n*f_n_1;
}

int main()
{
    printf("%d! = %d\n",7,factorial(7));
    return 0;
}
```

Odczyt wartości rejestru SP

Implementacja funkcji rekurencyjnych 3

```
Adress n 2686752
factorial( 7 ) -- stack: 2686720

Adress n 2686720
factorial( 6 ) -- stack: 2686688

Adress n 2686688
factorial( 5 ) -- stack: 2686656

Adress n 2686656
factorial( 4 ) -- stack: 2686624

Adress n 2686624
factorial( 3 ) -- stack: 2686592

Adress n 2686592
factorial( 2 ) -- stack: 2686560

Adress n 2686560
factorial( 1 ) -- stack: 2686528
7! = 5040
```

Program wypisuje informacje o siedmiu ramkach funkcji odkładanych na stosie

Adres zmiennej n (formalnego parametru funkcji) odpowiada wskaźnikowi stosu poprzedniej ramki

Implementacja funkcji rekurencyjnych

Wersja bardziej estetyczna

```
int factorial(int n, int indent)
{
     int f_n_1;
     register unsigned esp __asm__ ("esp");
     int i;

     for(i=0;i<indent;i++)putchar(' ');
     printf("entering factorial( %d ) -- stack: %u \n",n,esp);

     if(n<2)return 1;

     f_n_1 = factorial(n-1,indent+1);

     for(i=0;i<indent;i++)putchar(' ');
     printf("leaving factorial( %d ) -- stack: %u \n",n,esp);
     return  n*f_n_1;
}

int main()
{
    printf("%d! = %d\n",7,factorial(7,0));
    return 0;
}
```

Wcięcie

n – maleje, wcięcie rośnie

Na początku wcięcie wynosi 0

Implementacja funkcji rekurencyjnych 5

Wynik wywołania…

```
entering factorial( 7 ) -- stack: 2686720
 entering factorial( 6 ) -- stack: 2686688
  entering factorial( 5 ) -- stack: 2686656
   entering factorial( 4 ) -- stack: 2686624
    entering factorial( 3 ) -- stack: 2686592
     entering factorial( 2 ) -- stack: 2686560
      entering factorial( 1 ) -- stack: 2686528
     leaving factorial( 2 ) -- stack: 2686560
    leaving factorial( 3 ) -- stack: 2686592
   leaving factorial( 4 ) -- stack: 2686624
  leaving factorial( 5 ) -- stack: 2686656
 leaving factorial( 6 ) -- stack: 2686688
leaving factorial( 7 ) -- stack: 2686720
7! = 5040
```

Pułapki rekurencji 1

- Brak kryterium zatrzymującego ruch w dół stosu

```
int factorial(int n)
{
    return n*factorial(n-1);
}
```

- Nadmierna komplikacja prostego algorytmu…

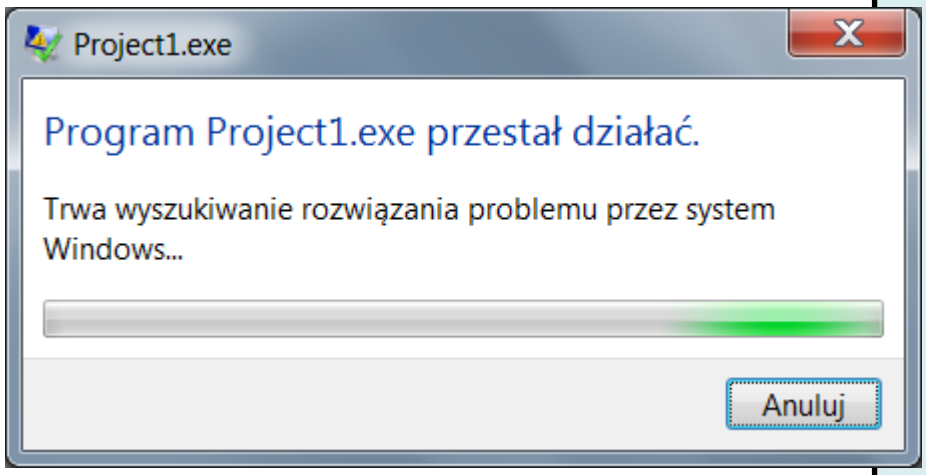
```
int sum(int tab[],int size) {
    if(size==0)return 0;
    return tab[0]+sum(tab+1,size-1);
}

int main() {
    int tab[]={4,3,2,1};
    printf("sum = %d\n",sum(tab,4));
    return 0;
}
```

`tab+1` to podtablica – rozpoczynająca się od następnego elementu…

Pułapki rekurencji 2

- Brak kontroli wykorzystania pamięci. Na stosie nie ma miejsca na 1 000 000 ramek funkcji (ale jednak mieści się około 70 000).
- Wywołanie funkcji trwa dłużej niż pętla iteracji
- Nad iteracją mamy pełną kontrolę

```
int sum(int tab[],int size)
{
    if(size==0)return 0;
    return tab[0]+sum(tab+1,size-1);
}




int tab[10000000];

int main()
{
    printf("sum = %d\n",sum(tab,1000000));
    return 0;
}
```



Do zapamiętania

- Definicja funkcji - składnia
- Deklaracja funkcji (prototypy) – składnia
- Formalne parametry i argumenty
- Użycie stosu do przekazywania wartości argumentów
- Przydział pamięci dla zmiennych automatycznych (deklarowanych w bloku instrukcji wewnątrz funkcji)
- Rola stosu w realizacji funkcji rekurencyjnych