Struktura programu C

```
#include <stdio.h> /*instrukcje preprocesora */
#define TRUE 1

int global_var = 10; /* zmienne globalne */
int global_array[1000000];
void func();

int main(void) { /* glowna funkcja */
  int local_var = 10;
  if(TRUE)
    printf("%d", global_var + 10);
  return 0;
}
void func() {printf("Funkcja func wywolana!");}
```

Preprocesor

Kompilacja — zamiana kodu programu napisanego w jednym języku na program w innym języku (np. zamiana kodu C na kod maszynowy).

Preprocesor

Kompilacja — zamiana kodu programu napisanego w jednym języku na program w innym języku (np. zamiana kodu C na kod maszynowy).

Preprocesor — program uruchomiany na początku kompilacji, przetwarza kod za pomocą *dyrektyw preprocesora*.

Preprocesor

Kompilacja — zamiana kodu programu napisanego w jednym języku na program w innym języku (np. zamiana kodu C na kod maszynowy).

Preprocesor — program uruchomiany na początku kompilacji, przetwarza kod za pomocą *dyrektyw preprocesora*.

Przykładowe dyrektywy:

- `#include`
- `#define`
- `#ifdef/#endif`
- `#ifndef/#endif`

Preprocesor — przykład

```
#define PI 3.1415
int main(){
  double pi_sq = PI*PI;
  double pi_2 = PI/2;
}
```

Preprocesor — przykład

```
#define PI 3.1415
int main(){
  double pi_sq = PI*PI;
  double pi_2 = PI/2;
}
```

Po przetworzeniu przez preprocesor:

```
int main(){
  double pi_sq = 3.1415*3.1415;
  double pi_2 = 3.1415/2;
}
```

Preprocesor — przykład

```
#define PI 3.1415
int main(){
  double pi_sq = PI*PI;
  double pi_2 = PI/2;
}
```

Po przetworzeniu przez preprocesor:

```
int main(){
  double pi_sq = 3.1415*3.1415;
  double pi_2 = 3.1415/2;
}
```

`#define` wstawia dane wyrażenie w dane miejsca w kodzie

Preprocesor — przykład

```
#define PETLA10(X) for(int i=0; i<10;++i){ X; }
int main(){
  PETLA10(printf("Petla!\n"))
}
```

Preprocesor — przykład

```
#define PETLA10(X) for(int i=0; i<10;++i){ X; }
int main(){
  PETLA10(printf("Petla!\n"))
}
```

Po przetworzeniu przez preprocesor:

```
#include <stdio.h>
int main(){
  for(int i=0; i<10;++i){printf("Petla!\n"); }
}
```

Preprocesor — include

plik 'wypisz.h':

```
void wypisz_liczbe(int x) {printf("%d", x);}
```

plik 'main.c':

```
#include "wypisz.h"
int main(){
  wypisz_liczbe(20);
}
```

Preprocesor — include

plik 'wypisz.h':

```
void wypisz_liczbe(int x) {printf("%d", x);}
```

plik 'main.c':

```
#include "wypisz.h"
int main(){
  wypisz_liczbe(20);
}
```

Po przetworzeniu przez preprocesor:

```
void wypisz_liczbe(int x) {printf("%d", x);}
int main(){
  wypisz_liczbe(20);
}
```

#include — po co

Po co używać #include?

#include — po co

Po co używać #include?

- użycie zewnętrznych bibliotek

#include — po co

Po co używać #include?

- użycie zewnętrznych bibliotek
- ładniejsza struktura kodu — ciężko czytać kod jeśli cały jest w jednym pliku

#include — po co

Po co używać #include?

- użycie zewnętrznych bibliotek
- ładniejsza struktura kodu — ciężko czytać kod jeśli cały jest w jednym pliku
- nie musimy przekompilowywać całego programu — program make (o tym na późniejszych zajęciach)

Definicja w osobnym pliku

Cały program to 3 pliki:
plik 'wypisz.h':

```
void wypisz_liczbe(int x); //deklaracja funkcji
```

plik 'wypisz.c':

```
#include <stdio.h>
#include "wypisz.h" //zalaczamy wypisz.h


void wypisz_liczbe(int x) { //def. funkcji
printf("%d\n", x);
}
```

plik 'main.c':

```
#include "wypisz.h"
int main(){
  wypisz_liczbe(20);
}
```

Kompilacja programów z wielu plików *.c

W Code::Blocks wystarczy dodać pliki $*.c$ do projektu.

Kompilacja programów z wielu plików *.c

W Code::Blocks wystarczy dodać pliki $*.c$ do projektu.
Kompilacja z konsoli:

```
gcc plik1.c plik2.c ... itd.
```

Konwencje

- Definicje krótkich funkcji możemy umieścić w pliku $*.h$
- Dłuższe funkcje umieszczamy w osobnym pliku $*.c$

Konwencje

- Definicje krótkich funkcji możemy umieścić w pliku $*.h$
- Dłuższe funkcje umieszczamy w osobnym pliku $*.c$

Dlaczego?

Konwencje

- Definicje krótkich funkcji możemy umieścić w pliku $*.h$
- Dłuższe funkcje umieszczamy w osobnym pliku $*.c$

Dlaczego?

- Preprocesor podstawia pliki $*.h$ do plików $*.c$
- Kompilator kompiluje pliki $*.c$ osobno
- Możemy z użyciem *make* żądać by tylko plik zaw. def zmienionej funkcji i pliki które są dotknięte tą zmianą były na nowo kompilowane — więcej na kolejnych zajęciach.

Strażnicy — Include Guards

plik 'wypisz.h':

```
void wypisz_liczbe(int x) {printf("%d", x);}
```

plik 'main.c':

```
#include "wypisz.h"
#include "wypisz.h"
int main(){
  wypisz_liczbe(20);
}
```

Strażnicy — Include Guards

plik 'wypisz.h':

```
void wypisz_liczbe(int x) {printf("%d", x);}
```

plik 'main.c':

```
#include "wypisz.h"
#include "wypisz.h"
int main(){
  wypisz_liczbe(20);
}
```

Po przetworzeniu przez preprocesor:

```
void wypisz_liczbe(int x) {printf("%d", x);}
void wypisz_liczbe(int x) {printf("%d", x);} \\redef.!!!
int main(){
wypisz_liczbe(20);
}
```

Strażnicy — Include Guards

plik 'wypisz.h':

```
void wypisz_liczbe(int x) {printf("%d", x);}
```

plik 'uzywa_wypisz.h':

```
#include "wypisz.h"
void f(...) {... wypisz_liczbe(5); }
```

plik 'main.c':

```
#include "wypisz.h"
#include "uzywa\_wypisz.h"
int main(){
wypisz_liczbe(20);
f(10);
}
```

Strażnicy — Include Guards

plik 'wypisz.h':

```
void wypisz_liczbe(int x) {printf("%d", x);}
```

plik 'uzywa_wypisz.h':

```
#include "wypisz.h"
void f(...) {... wypisz_liczbe(5); }
```

plik 'main.c':

```
#include "wypisz.h"
#include "uzywa\_wypisz.h"
int main(){
wypisz_liczbe(20);
f(10);
}
```

Nie wystarczy pilnować żeby dokładnie raz w każdym pliku załączać nagłówek!

Strażnicy — Include Guards

plik 'wypisz.h':

```
#ifndef WYPISZ_H
#define WYPISZ_H
void wypisz_liczbe(int x) {printf("%d", x);}
#endif
```

plik 'main.c':

```
#include "wypisz.h"
#include "wypisz.h"
int main(){
wypisz_liczbe(20);
}
```

Stos programu

Jaki będzie wynik tego programu?

```
#include <stdio.h>
#define size 9000000;

int main() {

int array[size];
for (int i=0; i<size; ++i)
  array[i] = 10;

return 0;
}
```

Stos programu

Jaki będzie wynik tego programu?

```
#include <stdio.h>
#define size 9000000;

int main() {

int array[size];
for (int i=0; i<size; ++i)
  array[i] = 10;

return 0;
}
```

Segmentation fault. (najprawdopodobniej)

Stos programu

Jaki będzie wynik tego programu?

```
#include <stdio.h>
#define size 9000000;

int array[size];
int main() {

for (int i=0; i<size; ++i)
  array[i] = 10;

return 0;
}
```

Stos programu

Jaki będzie wynik tego programu?

```
#include <stdio.h>
#define size 9000000;

int array[size];
int main() {

for (int i=0; i<size; ++i)
  array[i] = 10;

return 0;
}
```

Zakończy się i zwróci 0

Pamięć programu jest podzielona na segmenty:

- text data
- initialized data
- uninitialized data (bss) — zmienne globalne
- stack — stos
- heap — sterta

Segmenty pamięci

Pamięć programu jest podzielona na segmenty:

- text data
- initialized data
- uninitialized data (bss) — zmienne globalne
- **stack (stos)** — dzisiaj
- **heap (sterta)** — za tydzień

Stos

Na stosie są przechowywane zmienne lokalne.

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.
Rozmiar stosu można sprawdzić (i modyfikować) poleceniem:
`ulimit -s` (Linux).

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.
Rozmiar stosu można sprawdzić (i modyfikować) poleceniem:
`ulimit -s` (Linux).
Stos jest mały bo:

- Przechowywane są tam zmienne lokalne, duże tablice powinny być alokowane na stercie

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.
Rozmiar stosu można sprawdzić (i modyfikować) poleceniem:
`ulimit -s` (Linux).
Stos jest mały bo:

- Przechowywane są tam zmienne lokalne, duże tablice powinny być alokowane na stercie
- Musi być spójnym obszarem pamięci

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.
Rozmiar stosu można sprawdzić (i modyfikować) poleceniem:
`ulimit -s` (Linux).
Stos jest mały bo:

- Przechowywane są tam zmienne lokalne, duże tablice powinny być alokowane na stercie
- Musi być spójnym obszarem pamięci
- Ma szybszy dostęp niż sterta

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.
Rozmiar stosu można sprawdzić (i modyfikować) poleceniem:
`ulimit -s` (Linux).
Stos jest mały bo:

- Przechowywane są tam zmienne lokalne, duże tablice powinny być alokowane na stercie
- Musi być spójnym obszarem pamięci
- Ma szybszy dostęp niż sterta
- Ogranicza poziom rekursji (następny slajd)

Stos

Na stosie są przechowywane zmienne lokalne.

```
int main(){
  int zmienna; //stos
  int tablica[100];//stos
}
```

Standardowy rozmiar stosu jest dość mały: kilka MB.
Rozmiar stosu można sprawdzić (i modyfikować) poleceniem:
`ulimit -s` (Linux).
Stos jest mały bo:

- Przechowywane są tam zmienne lokalne, duże tablice powinny być alokowane na stercie
- Musi być spójnym obszarem pamięci
- Ma szybszy dostęp niż sterta
- Ogranicza poziom rekursji (następny slajd)

Funkcje rekurencyjne

Jaki będzie wynik?

```
#include <stdio.h>

int silnia(int x){
  int wynik = 1;
  if (x)
    wynik = silnia(x-1)*x % 1009;
    return wynik;
}

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

Funkcje rekurencyjne

Jaki będzie wynik?

```
#include <stdio.h>

int silnia(int x){
  int wynik = 1;
  if (x)
    wynik = silnia(x-1)*x % 1009;
    return wynik;
}

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

Segmentation fault.

Funkcje rekurencyjne

- Podczas każdego wywołania funkcji na stosie tworzone jest nowe lokalne środowisko dla funkcji.

Funkcje rekurencyjne

- Podczas każdego wywołania funkcji na stosie tworzone jest nowe lokalne środowisko dla funkcji.
- Na stosie przechowywane są lokalne zmienne utworzene w funkcji i adres powrotu, po wywołaniu są one usuwane ze stosu.

Funkcje rekurencyjne

- Podczas każdego wywołania funkcji na stosie tworzone jest nowe lokalne środowisko dla funkcji.
- Na stosie przechowywane są lokalne zmienne utworzene w funkcji i adres powrotu, po wywołaniu są one usuwane ze stosu.

Przykład z silnią — rysunek na tablicy

Funkcje rekurencyjne

- Jeśli popełnimy błąd w kodzie i napiszemy niesk. rekurencję to program będzie działał do momentu przepełnienia stosu, a nie do wyczerpania całej dostępnej pamięci
- Przepełnienie stosu — **stack overflow**

Inny przykład

```
#include <stdio.h>

int f(int n, int k){

  if(n == k || k == 0) return 1;

  int f1 = f(n-1, k);
  int f2 = f(n-1, k-1);
  return f1 + f2;
}

int main() {
  printf("%d\n", f(8, 4));
  return 0;
}
```

Rekurencja wzajemna

```
#include <stdio.h>
int czy_parzysta(int n) {
  if (n == 0)
    return 1;
  else
    return czy_nieparzysta(n - 1);
}
int czy_nieparzysta(int n) {
  if (n == 0)
    return 0;
  else
    return czy_parzysta(n - 1);
}
int main() {
  printf("%d\n", czy_parzysta(4));
  return 0;
}
```

Inny przykład

Co jest nie tak z tą funkcją?

```
void mergesort(int n, int tab[]){
  if(n < 2) return;

  int s1 = n/2, s2 = n/2 + (n % 2);
  int tab1[s1];
  int tab2[s2];
  //kopiuje el. z tab do tab1 i tab2
  podziel(tab, tab1, tab2, s1, s2);
  mergesort(s1, tab1);
  mergesort(s2, tab2);
  //kopiuje el. z tab1 i tab2 do tab
  scal(tab, tab1, tab2, s1, s2);
}
```

Stos - podsumowanie

- nie deklarujemy dynamicznie dużych tablic (przynajmniej nie przez `int tab[...]`)
- pamiętamy w rekurencji o limicie na stos