Podstawy programowania.
Wykład 12
Preprocesor i modularna struktura programów

Programy: funkcje i zmienne

- Większość programów w C stanowią rozbudowane kody, definiujące wiele funkcji i jednocześnie korzystające z wielu funkcji zawartych w bibliotekach
- Wygodną formą przechowywania kodu źródłowego takich programów jest użycie wielu plików, często w odrębnych katalogach systemu plików
- Stosowanie wielu plików wymusza wprowadzenie specjalnych konstrukcji w kodzie, ze względu na fakt, że jednostką kompilacji w C jest pojedynczy plik
  - kompilator, dokonujący tłumaczenia pojedynczego pliku źródłowego na plik pośredni, musi posiadać wszelkie informację potrzebne do poprawnej kompilacji oraz późniejszego poprawnego połączenia przez konsolidator plików pośrednich w plik wykonywalny (który zazwyczaj jest jeden dla całego programu)

Definicje i deklaracje

- Jednym z elementów koniecznych do uwzględnienia przy pracy z wieloma plikami jest przekazanie informacji o funkcjach i zmiennych globalnych definiowanych w jednym pliku funkcjom definiowanym w innym pliku
- Sposobem przekazania tej informacji jest użycie deklaracji nie będących definicjami
- Definicje funkcji i zmiennych globalnych:
  - każda funkcja i zmienna globalna definiowana jest tylko raz
  - definicja funkcji zawiera treść funkcji (wykonywane operacje)
  - definicja zmiennej globalnej to miejsce w kodzie, gdzie kompilator otrzymuje informacje o konieczności zarezerwowania odpowiedniej przestrzeni w pamięci
    - definicja jest standardową informacją o typie zmiennej, bez żadnych określeń
    - definicja zmiennej może być połączona z jej inicjowaniem

Definicje i deklaracje

- Deklaracje funkcji i zmiennych globalnych
  - nazwa funkcji lub zmiennej globalnej staje się znana kompilatorowi od momentu jej deklaracji
  - deklaracja funkcji to informacja o nazwie funkcji, typach argumentów oraz typie zwracanego wyniku
    - deklaracją może być **definicja** lub **prototyp** funkcji
  - deklaracja zmiennej globalnej informuje o jej typie
    - deklaracja nie będąca definicją zawiera określenie extern
  - określenie extern informuje, że funkcja lub zmienna są globalne i są zdefiniowane w innym miejscu
    - wszystkie standardowo definiowane (bez określeń) funkcje są globalne
      - określenie extern może zostać pominięte w prototypie, czyli deklaracji funkcji nie będącej definicją
    - określenie extern jest konieczne w deklaracji zmiennych globalnych

Definicje i deklaracje

- Deklaracje (nie będące definicjami)
  - standardowo odnoszą się do funkcji i zmiennych globalnych
  - ich położenie w pliku, wewnątrz lub na zewnątrz funkcji, jest bez znaczenia
  - zwyczajowo deklaracje umieszcza się na początku pliku źródłowego, przed definicjami funkcji
- Kompilator umieszcza informacje o napotkanych zadeklarowanych (ale nie zdefiniowanych) funkcjach i zmiennych globalnych w plikach pośrednich
- Konsolidator będzie starał się znaleźć definicje funkcji i zmiennych globalnych dla wszystkich napotkanych w plikach pośrednich zadeklarowanych funkcji i zmiennych globalnych
  - brak definicji funkcji lub zmiennej globalnej zadeklarowanej i wykorzystywanej w dowolnym pliku źródłowym powoduje zgłoszenie błędu konsolidacji (linkowania)

Definicje i deklaracje

- Definicje funkcji i zmiennej globalnej (w jednym z plików):

```
double global = 0.0;
int jest_wieksza_niz_global( double liczba){
    if( liczba > global ) return 1;
    else return 0; }
```

- Deklaracje funkcji i zmiennej globalnej
  - w każdym pliku, w którym są wykorzystywane (mogą być także w pliku zawierającym definicje)
  - powinny zawierać określenie *extern* (można je pominąć dla funkcji)

```
extern double global; // informacja: gdzieś została zdefiniowana
                      // zmienna globalna global
int jest_wieksza_niz_global (double ); // informacja: gdzieś została
                      // zdefiniowana funkcja jest_wieksza_niz_global
// możliwe użycie:
if ( jest_większa_niż_global( 3.14 ) ) { global = 3.14; } else { ..... }
```

Sposoby przechowywania zmiennych

- Poza standardowo definiowanymi zmiennymi: lokalnymi (definiowanymi w konkretnym bloku programu) i globalnymi, istnieją jeszcze zmienne definiowane z określeniem static
  - takie zmienne definiowane poza funkcjami są widoczne we wszystkich funkcjach definiowanych w danym pliku
    - czas życia jest równy czasowi wykonania programu
  - jeśli zmienna definiowana jest z określeniem static wewnątrz funkcji (bloku), jej widzialność pozostaje określona przez miejsce definicji, ale zmienia się czas życia na równy czasowi wykonania programu
    - jej wartość zostaje zachowana pomiędzy wywołaniami funkcji

```
static int zm_statyczna_glob = 0; // jednokrotne inicjowanie
void funkcja(void){
    static int zm_statyczna_lok = 0; // jednokrotne inicjowanie
    zm_statyczna_glob++;
    zm_statyczna_lok++;
}
```

Sposób linkowania (linkage)

- Sposób linkowania funkcji i obiektów (zmiennych) określa czy dane dwie deklaracje odnoszą się do tego samego obiektu
  - **sposób linkowania statyczny** (*static linkage*) **– jeden obiekt w pliku**
    - nielokalne obiekty posiadające sposób przechowywania *static*
  - **sposób linkowania zewnętrzny** (*external linkage*) **– jeden obiekt w programie**
    - obiekty posiadające sposób przechowywania *extern* (z pewnymi wyjątkami)
    - np. wszystkie nie-statyczne funkcje
  - **sposób linkowania nieokreślony** (*none*) **- każda definicja odnosi się do odrębnego obiektu**
    - takie obiekty w kodzie posiadają wyłącznie definicje – deklaracje (nie będące definicjami) są niepotrzebne
    - np. obiekty na stosie (argumenty i standardowe zmienne lokalne procedur (bez określeń `static` i `extern`))

Czas życia obiektów

- Obiekty (m.in. zmienne) mają jeden z czterech określonych czasów życia (okresów przechowywania, ***storage duration***)
  - statyczny (***static***) - czas życia równy czasowi wykonywania programu
    - jednokrotne inicjowanie na początku wykonania programu
      - jawne lub niejawne za pomocą zer
    - zmienne statyczne (*static*) i globalne (*extern*)
  - automatyczny (***automatic***) - czas życia od momentu definicji do końca bloku (funkcja jest także blokiem), w którym obiekt jest zdefiniowany
    - każdorazowe wejście do bloku rozpoczyna "nowe życie" obiektu
      - bez domyślnego inicjowania - jawne inicjowanie jest przeprowadzane przy każdym napotkaniu definicji z inicjowaniem
  - "dynamiczny", zaalokowany *(**allocated***) - czas życia wynikający z użycia funkcji dynamicznego alokowania i zwalniania pamięci
  - ***thread*** - czas życia związany z istnieniem odrębnego wątku wykonującego określony fragment programu

Preprocesor

- Każdy plik z kodem źródłowym, przed translacją dokonywaną przez kompilator, jest przetwarzany przez preprocesor
  - preprocesor w języku C znajduje w kodzie źródłowym dyrektywy zawarte w linijkach zaczynających się od znaku #
  - podstawowymi dyrektywami preprocesora C są
    - #include
      - do włączania dowolnych plików do pliku z kodem źródłowym
        - » w miejsce linijki #include plik wstawiana jest zawartość pliku
      - włączane pliki także mogą włączać kolejne pliki poprzez ich własne dyrektywy #include
    - #define
      - do definiowania symboli, a także dokonywania bardziej złożonych podstawień w kodzie źródłowym
    - #if, #if defined, #ifdef, #elif, #else, #endif
      - do dokonywania kompilacji warunkowej

Pliki nagłówkowe

- Deklaracje (a także inne elementy, np. definicje nowych typów zmiennych, definicje stałych nazwanych) można przenieść do innego pliku, który zostanie włączony (wklejony) do pliku źródłowego za pomocą dyrektywy preprocesora #include
  - taki plik nazywany jest plikiem nagłówkowym (*header file*)
  - przykładowa zawartość (z przykładu listy powiązanej):

```
typedef struct element_listy* wsk_el_list;     // wszystkie elementy
typedef struct element_listy{                  // powinny posiadać
    char* nazwa_wezla;                         // odpowiednie komentarze
    wsk_el_list nastepny_wezel;                // umożliwiające
} el_list;                                     // ich poprawne użycie, np.:
int wstaw_na_poczatek(  // funkcja zwraca kod sukcesu lub błędu
  el_list** Glowa_wsk,    // lista - identyfikowana przez wskaźnik
  el_list* Element );     // wskaźnik do elementu wstawianego na listę
// itd. itp.
```

Pliki nagłówkowe

- Dyrektywa #include włącza do kompilowanego pliku źródłowego pliki
  - ze znanych kompilatorowi, standardowych lokalizacji

    ```
    #include <stdio.h> // w Linuxie zazwyczaj w /usr/include
    ```
  - z dowolnych lokalizacji

    ```
    #include "moja_biblioteka.h"
    ```
    - katalog (pełna ścieżka do katalogu), w którym znajduje się włączany plik jest przekazywany w ramach opcji kompilatora

      ```
      -Iścieżka_do_katalogu // np. -I/home/student/program/include
      ```
    - domyślnie preprocesor szuka włączanego pliku w katalogu zawierającym plik włączający
- Włączane pliki stają się treścią pliku do którego zostały włączone
  - pliki włączone wraz z plikiem włączającym stają się pojedynczym plikiem, jednostką translacji dla kompilatora

Preprocesor

- Dyrektywa #define ma postać

  #define nazwa tekst

  - działanie preprocesora polega na znalezieniu w pliku wszystkich wystąpień nazwa (od miejsca dyrektywy) i zamienieniu ich na tekst
    - nazwa jest dowolną dopuszczalną nazwą (tak jak nazwy zmiennych)
    - kompilacji poddawany jest plik z podmienionymi wystąpieniami nazwa
      - symbol nazwa znika z treści kompilowanego pliku
- Dyrektywa #define służy najczęściej do definiowania tzw. stałych nazwanych
  - #define nazwa_stałej stała
    - stała jest dowolną dopuszczalną stałą języka C (znakową, tekstową, liczbową – różnych typów)
    - użycie nazwanej stałej zamiast jej wartości ma na celu uczynienie kodu bardziej zrozumiałym
      - eliminacja tzw. liczb magicznych (*magic numbers*)

Preprocesor

- Dyrektywa #define może służyć do definicji symboli wykorzystywanych w kompilacji warunkowej

  #define symbol

- Symbol wykorzystywany w kompilacji warunkowej może także zostać przesłany jako opcja kompilatora
  - najczęściej w postaci -Dsymbol
- Dla zapewnienia, że symbol nie jest zdefiniowany można użyć dyrektywy #undef:

  #undef symbol

- Standardowa kompilacja warunkowa zapisywana jest w kodzie źródłowym w postaci:

```
#ifdef symbol // ewentualnie #if defined(symbol)
... // kod kompilowany tylko wtedy kiedy symbol jest  zdefiniowany
#endif
```

Kompilacja warunkowa

- Kompilacja warunkowa może korzystać także z dyrektywy wymagającej, aby jakaś zmienna nie była zdefiniowana

```
#ifndef SYMBOL // równoważne: if !defined(SYMBOL)
....
#endif
```

- Przykładem zastosowania kompilacji warunkowej jest np.
  - zabezpieczanie plików (np. nagłówkowych) przed wielokrotnym włączaniem do jednego pliku (np. poprzez zagnieżdżone dyrektywy #include)
    - każdy plik określa swój własny powiązany symbol
      - np. _lista_powiazana_ dla pliku *lista_powiazana.h*
    - następnie na początku i na końcu umieszcza linijki

```
#ifndef _lista_powiazana_
#define _lista_powiazana_
... // zawartość pliku
#endif
```

  - wariant programowania generycznego dostępny w C

Preprocesor

- Kompilację warunkową można wykorzystać w kodzie do wykonywania operacji nie wymaganych do poprawnego realizacji algorytmu, ale przydatnych np. do:
  - uzyskiwania parametrów wykonania, np. czasu realizacji konkretnych operacji
  - poszukiwania błędów w programie (debugowania)
    - fragmenty zawierające szczegółowe, czasochłonne sprawdzenie poprawności wykonywanych operacji mogą być kompilowane tylko po jawnym zdefiniowaniu określonego symbolu

```
#ifdef MY_DEBUG
... // szczegółowe sprawdzanie poprawności działania
#endif
```

      - symbol DEBUG bywa definiowany automatycznie w pewnych sytuacjach przez rozmaite środowiska tworzenia oprogramowania
    - dla pełnej kontroli można definiować własne symbole

Preprocesor

- Bardziej złożone postacie kompilacji warunkowej wykorzystują dyrektywy #if, #elif, #else, #endif
- Kompilację warunkową można wykorzystać w kodzie do dostosowania się do konkretnego środowiska wykonania programu:

```
#if SYSTEM == SYSV        // if (obliczane wyrażenie) {
   #define HDR "sysv.h"          // definicje dla systemu SYS V
#elif SYSTEM == BSD       // } else if (obliczane wyrażenie){
   #define HDR "bsd.h"           // definicje dla systemu BSD
#elif SYSTEM == MSDOS     // } else if (obliczane wyrażenie){
   #define HDR "msdos.h"         // definicje dla systemu MSDOS
#else                     // } else {
   #define HDR "default.h"       // definicje domyślne
#endif                    // }
#include HDR
```

Makrodefinicje

- Dyrektywa #define umożliwia tworzenie makr (makrodefinicji)
  - makro oznacza podmianę tekstu w pliku źródłowym połączoną z podstawieniem argumentów
    - makro może być stosowane zamiast funkcji, choć posiada zupełnie inny mechanizm działania, polegający na manipulacji tekstem, zamiast definiowania kodu nazwanej funkcji, przesyłania argumentów o określonym typie przez stos itp.
- Najczęściej przytaczanym przykładem makra jest

  `#define max(A, B)  ((A)>(B) ? (A) : (B))`
  - oznacza ono, że tekst w kodzie:

    `x = max(p+q, r+s);` // uwaga na modyfikujące się argumenty `max(i++, j++)` - BŁĄD!

    zostanie zamieniony na:

    `x = ((p+q) > (r+s) ? (p+q) : (r+s));`
    - istotną rolę odgrywa poprawne użycie nawiasów w makrodefinicji
  - makro `max(A, B)` może być stosowane dla dowolnych typów zmiennych
    - odpowiadającą mu znaczeniowo funkcję powinno się zdefiniować osobno dla różnych typów zmiennych

Narzędzie *make*

- Narzędzie *make* służy do realizacji wybranych operacji, dla zestawu plików, z których część ulega modyfikacjom, a pozostałe muszą się do tych modyfikacji dostosować
  - fakt konieczności dostosowania jednego pliku do modyfikacji w innym pliku jest określany jako zależność
  - przykładem zależności między plikami jest zależność pliku binarnego programu od pliku pośredniego programu, który jest z kolei zależny od pliku źródłowego oraz dołączanych do pliku źródłowego plików pośrednich
- Najczęstszym zastosowaniem narzędzia *make* jest wykorzystanie do kompilacji i budowania programów, zawartych w wielu plikach źródłowych, korzystających z wielu plików nagłówkowych
  - *make* umożliwia optymalizację procesu budowania kodu binarnego przez dokonywanie kompilacji tylko wtedy kiedy jest potrzebna, czyli kiedy pliki, od których dany plik jest zależny zostały zmienione

Narzędzie *make*

- *make* realizuje komendy zawarte w pliku sterującym, zwyczajowo nazywanym *Makefile* (lub *makefile*)
- komendy zgrupowane są w zestawy, każdy zestaw posiada nazwę, tzw. cel (*target*)
  - w klasycznym przypadku kompilacji i linkowania nazwa celu jest zwyczajowo nazwą uzyskiwanego pliku binarnego lub pośredniego
  - wymagana składnia *Makefile* określa, że po nazwie celu następuje zestaw plików, od których cel jest zależny, po czym w kolejnych linijkach, **zaczynających się od znaku tabulacji**, zawarte są komendy
  - najprostszy przykład zastosowania do kompilacji:

```
hello: hello.c
	gcc hello.c  -o hello
```

  - *make* uruchomiony dla powyższej zawartości *Makefile* sprawdzi czasy modyfikacji pliku binarnego *hello* i pliku źródłowego *hello.c*, i jeśli ten ostatni zmodyfikowany został później wykona operacje zapisane poniżej linijki z celem

Makefile

- Pliki *Makefile* zawierają zazwyczaj wiele celów i wiele zestawów komend dla różnych celów
  - różne zestawy mogą zawierać te same komendy (np. wywołanie kompilatora lub konsolidatora)
  - dla ułatwienia zamiany realizowanej komendy dla wszystkich celów, *Makefile* posługuje się symbolami, najczęściej definiowanymi na początku pliku, np:

```
# kompilator C (często symbol CC)
CCOMP = gcc

hello: hello.c
	$(CCOMP) hello.c  -o hello
```

    - w powyższym przykładzie $(CCOMP) zostanie zamienione na gcc, a zamiana kompilatora na dowolny inny polegać będzie tylko na zmianie definicji symbolu
  - przykład pokazuje także użycie komentarzy w plikach *Makefile*, zawartych w linijkach zaczynających się od znaku #

Kompilacja i linkowanie

- Standardowo proces budowania kodu rozbity jest na dwa etapy: kompilacji i konsolidacji (linkowania)
  - na etapie kompilacji do plików źródłowych dołączane są pliki nagłówkowe
    - jeśli pliki nagłówkowe nie znajdują się w lokalizacjach znanych kompilatorowi (odpowiednie katalogi systemowe lub katalog bieżący) ich położenie można przekazać za pomocą opcji -I
    - ścieżkę do położenia plików nagłówkowych przekazywaną kompilatorowi można zapisać w pliku *Makefile* w postaci symbolu, np.:

      ```
      # pliki naglowkowe
      INC = -I../include -I/home/user/program/include
      ```

      - pierwsza podana ścieżka jest względna w stosunku do katalogu bieżącego
      - druga wersja korzysta z ścieżki bezwzględnej

Kompilacja i linkowanie

- Standardowo proces budowania kodu rozbity jest na dwa etapy: kompilacji i konsolidacji (linkowania)
  - na etapie linkowania następuje tworzenie pliku wykonywalnego na podstawie plików kodu pośredniego (*object code*) i bibliotek (statycznych, biblioteki dynamiczne dołączane są w momencie uruchomienia kodu binarnego)
    - jeśli pliki bibliotek nie znajdują się w lokalizacjach znanych kompilatorowi (odpowiednie katalogi systemowe lub katalog bieżący) ich położenie można przekazać za pomocą opcji -L
    - ścieżkę do położenia plików bibiotek przekazywaną kompilatorowi można zapisać w pliku *Makefile* w postaci kolejnego symbolu
    - w tym samym symbolu można także umieścić nazwę biblioteki lub standardową opcję dołączenia biblioteki -l

```
# biblioteki
LIB = -L../lib -L/home/user/program/lib -lmoja_biblioteka -lm
```

Makefile

- Kolejnym często stosowanym symbolem jest zestaw użytych optymalizacji:

```
# opcje kompilacji (często symbol CFLAGS):
# wersja do debugowania
OPT = -g -DMY_DEBUG
# wersja zoptymalizowana
# OPT = -O3
```

- Ostatecznie przepis na kompilację i linkowanie programu z pojedynczego pliku źródłowego może wyglądać następująco:

```
moj_program: moj_program.o
		$(CCOMP) $(OPT) moj_program.o -o moj_program $
		(LIB)
moj_program.o: moj_program.c moj_program.h
		$(CCOMP) -c $(OPT) moj_program.c $(INC)
```

- Plik *Makefile* powinien zaczynać się od linijki określającej właściwą powłokę do interpretacji komend:

```
SHELL = /bin/sh
```

Makefile

- Oprócz celów będących plikami, dla których *make* sprawdza zależności i czasy modyfikacji, *Makefile* może także zawierać tzw. sztuczne cele (*phony targets)*
  - sztuczne cele nie będące plikami, nie zawierają wskazówek kiedy związane z nimi komendy mają być wykonane
    - sztuczne cele wymagają jawnego polecenia realizacji
- Plik *Makefile* może zawierać wiele celów
  - uruchomienie *make* dla konkretnego celu ma postać

    ```
    make cel
    ```
  - uruchomienie *make* bez wskazania celu sprawia, że sprawdza on zależności i wykonuje konieczne do aktualizacji komendy dla pierwszego napotkanego celu
    - chcąc domyślnie realizować kilka celów można zdefiniować sztuczny cel , który będzie od nich zależny i umieszczony jako pierwszy, np.

      ```
      all: program_1, program_2
      ```

Makefile

- Często wykorzystywanym sztucznym celem jest clean , służące do czyszczenia katalogów z poprzednio utworzonych plików (np. plików pośrednich, dla pełnej rekompilacji kodu)

```
clean:
        rm -f moj_program *.o
```

- Program *make* jest złożonym programem, zawierającym wiele innych możliwości sterowania jego wykonaniem poprzez odpowiednią konfigurację plików *Makefile*, m.in.:
  - *make* może pewne operacje realizować automatycznie, np.
    - wyszukiwać zależności między plikami (na podstawie #include)
    - kompilować plik źródłowy, kiedy wiadomo, że potrzebny jest odpowiadający mu plik pośredni
  - *make* może korzystać z predefiniowanych własnych symboli
  - *make* może w jednym procesie budowania kodu korzystać z wielu plików *Makefile* w wielu katalogach
  - *make* może służyć do tworzenia i aktualizowania bibliotek