ISO/ANSI C

Podział kodu programu na pliki

181

ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych

- Duży kod warto dzielić na fragmenty o wspólnej funkcjonalności (np. funkcje numeryczne, funkcje we/wy, funkcje dostępu do plików, itp.)
- Fragmenty należy umieszczać w oddzielnych plikach.
- Pliki dołączamy do naszego pliku za pomocą dyrektywy **#include**, np.:

```
#include "sortowanie.c"
```

- Połączenia dokonuje preprocesor kodu.
- Preprocesor kodu składa postać pośrednią pliku do kompilacji zgodnie z dyrektywami.
- Dyrektywa zaczyna się od znaku # i nie kończy się średnikiem.
- Każda dyrektywa występuje w osobnej linii.

182

ISO/ANSI C - biblioteki

Łączenie – przykład:

Kod z pliku wskazanego przez **#include** jest łączony z kodem naszego pliku tworząc w momencie kompilacji postać pośrednią, tj. jedną dużą całość ułożoną sekwencyjnie wg kolejności dołączeń:

Plik A.txt:

```
Był skrzypek rodem z Prabutów,
#include "B.txt"
od skrzypiec zamiast butów.
```

Plik B.txt:

```
miał nogi za duże do butów.
    Wszystkie go uwierały,
    więc nosił futerały
```

Jaki tekst wygeneruje preprocesor kodu przetwarzając plik A.txt?

183

ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych

- Przy wielopoziomowych włączeniach kodu pojawia się problem.

Przykład:

A korzysta z B i C. B korzysta z D, a także C korzysta z D.
Wtedy D zostanie dołączony dwa razy.

W finalnej postaci kodu całego programu przygotowanej przez preprocesor funkcje z D pojawią się dwa razy – wystąpi błąd kompilacji. ☹

184

ISO/ANSI C - biblioteki

185

ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych

- Przy wielopoziomowych włączeniach kodu pojawia się problem.

Przykład:

A korzysta z B i C. B korzysta z D i C korzysta z D.
Wtedy D zostanie dołączony dwa razy.

- Rozwiązanie: należy włączać same deklaracje, a nie definicje funkcji. Natomiast definicje podać tylko raz, na końcu kodu programu.

186

ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych

Deklaracja:

```
char flip(char , struct klucz );
```

Definicja:

```
char flip(char c, struct klucz k) {
   int i;
   for(i=0;i<24; i++)
       if (c==k.mapa[i]) return k.mapa[(i+k.skok)%10];
   return c;
};
```


187

ISO/ANSI C - biblioteki

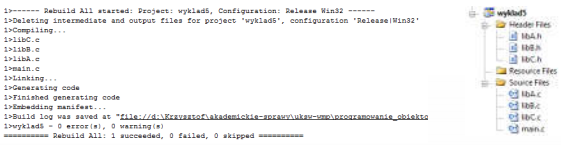
Dzielenie kodu na kilka plików źródłowych

- Tworząc biblioteki, dla każdego zestawu funkcji tworzymy dwa pliki: z deklaracjami i z definicjami. Np. biblioteka z funkcjami do sortowania mogłaby mieć pliki: sortowanie.h i sortowanie.c
- w pliku z funkcją `main` dołączamy tylko nagłówki, np.: **`#include "sortowanie.h"`**
- Jak i kiedy dołączamy plik z definicjami „.c"?
- Do tego służy „projekt" w środowisku programistycznym. W ramach zakładanego projektu wskazujemy wszystkie pliki „.c" oraz wszystkie pliki „.h", które zawierają niezbędny kod naszego programu.

188

ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych



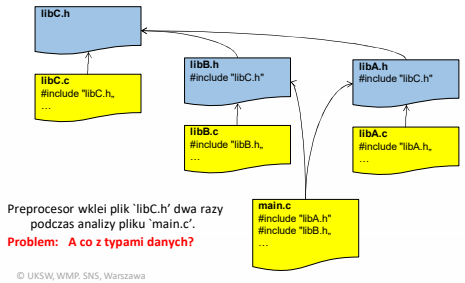
Proces kompilacji jest dwuetapowy:

1. Poszczególne pliki *.c kompilowane są kolejno; kody funkcji bibliotecznych nie są jeszcze konieczne (wystarczą same nagłówki).
2. Pliki wynikowe kompilacji są łączone (linkowane) w jeden plik *.exe

189

ISO/ANSI C - biblioteki

Przykład:



Preprocesor wklei plik `libC.h' dwa razy podczas analizy pliku `main.c'.

Problem: A co z typami danych?

190

ISO/ANSI C - biblioteki

Budowa pliku nagłówkowego *.h

Co zrobić, żeby plik `libC.h' jednak nie został wklejony dwa razy?

Wykorzystać dyrektywy preprocesora:

```
#ifndef nazwaPliku_h
#define nazwaPliku_h
/*
   tutaj deklaracje funkcji
*/
#endif
```

#define – definiuje (tworzy) nową stałą „nazwaPliku_h"

#ifndef – sprawdza, czy nie jest zdefiniowana stała „nazwaPliku_h". Jeżeli nie, włącza kod znajdujący się poniżej tej dyrektywy, do kodu wyjściowego.

#endif – znacznik końca tekstu objętego funkcją #ifndef

191

ISO/ANSI C - biblioteki

192

ISO/ANSI C - biblioteki

Budowa pliku nagłówkowego *.h

Podsumowując:
użycie dyrektyw **`#ifndef`**, **`#define`**, **`#endif`** gwarantuje, że niezależnie ile razy pojawi się dyrektywa

```
#include "libC.h"
```

treść pliku zostanie dołączona w tylko jednym egzemplarzu.

- Nazwy stałych, które definiujemy za pomocą preprocesora muszą być unikatowe podczas kompilacji projektu dla każdego używanego pliku.
- Najpopularniejszą metodą zapewnienia sobie unikatowych nazw stałych, jest korzystanie z nazwy pliku.

193

ISO/ANSI C - biblioteki

Budowa pliku źródłowego *.c

```
#include "nazwaPliku.h"

/*
tutaj definicje funkcji
*/
```

Visual Studio

194

ISO/ANSI C - biblioteki

Alternatywą dla:

```
#ifndef nazwaPliku_h
#define nazwaPliku_h
/*
  tutaj deklaracje funkcji
*/
#endif
```

w środowisku Visual Studio jest

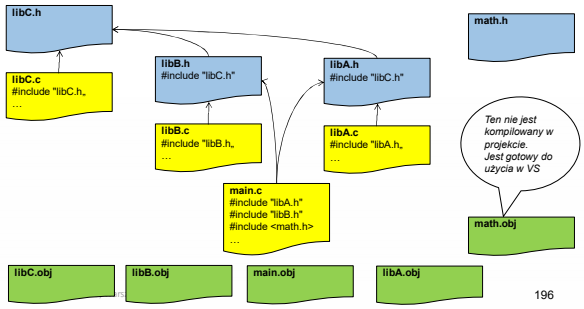
```
#pragma once
```

Zapobiega wielokrotnemu załączeniu treści całego pliku.
Ale nie należy do standardu..

195

ISO/ANSI C - biblioteki

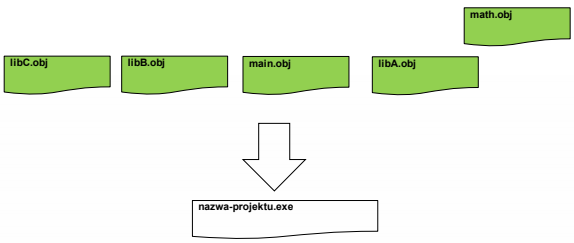
Kompilacja plików z projektu:



196

ISO/ANSI C - biblioteki

Linkowanie wyników kompilacji:



197

ISO/ANSI C - biblioteki

Zmienne zewnętrzne

Zmienne są „widziane” w kodzie znajdującym się poniżej ich deklaracji w obrębie bloku danych a najlepszym razie w obrębie pliku

Zmienne zadeklarowane w pliku poza ciałem funkcji nazywane są zmiennymi globalnymi

Aby zmienna globalna z jednego pliku była „widziana” w drugim, musi zostać zadeklarowana z modyfikatorem **extern**

Zadeklarowanie:

```
extern double x;
```

stanowi informację dla kompilatora, że zmienna ta jest lub będzie zdefiniowana w innym pliku/module

198

ISO/ANSI C - biblioteki

Zmienne zewnętrzne

Deklaracja zmiennej z modyfikatorem **`extern`** :

- deklaracja zmiennej nie jest związana z jej definicją,
- żadna zmienna nie jest tworzona, tzn. pamięć nie jest przydzielana,
- ta sama zmienna może być zadeklarowana jako **`extern`** wiele razy w wielu plikach, ale zdefiniowana może być tylko raz,
- zmiennej deklarowanej nie wolno inicjować:

```
extern double x = 0;  /* Nie! */
```

*Po takiej inicjalizacji kompilator zignoruje słowo kluczowe '**`extern`**' i potraktuje powyższa deklarację jak definicję. Kompilator po znalezieniu właściwej definicji nie zwróci komunikatu o błędzie, ponieważ będzie ona znajdowała się w innym pliku, a więc będzie traktowana jako definicja innej zmiennej, co prawda o tej samej nazwie, ale innej – bo w innym pliku.*

199

ISO/ANSI C

zakończenie

200

ISO/ANSI C – zakończenie

Odczyt i zapis bieżącej daty i czasu odbywa się za pomocą funkcji zdefiniowanych w bibliotece <time.h>

Zdefiniowane są tam następujące typy i stałe:

CLOCKS_PER_SEC – liczba 'tyknięć' na sekundę

clock_t, time_t – typy arytmetyczne do reprezentacji czasu

```
struct tm {
    int tm_sec      /* liczba sekund [0-61] (61 pozwala na 2 sekundy przestępne)*/
    int tm_min      /* liczba minut [0-59] */
    int tm_hour     /* liczba godzin po północy [0-23] */
    int tm_mday     /* dzień miesiąca [1-31] */
    int tm_mon      /* miesiąc w roku [0-11] */
    int tm_year     /* bieżący rok-1900 */
    int tm_wday     /* nr dnia licząc od niedzieli [0-6] */
    int tm_yday     /* nr dnia licząc od pierwszego stycznia [0-365] */
    int tm_isdst    /* znacznik uwzględniania czasu zimowego i letniego */
}
```

Struktura używana do reprezentacji czasu kalendarzowego

201

ISO/ANSI C – zakończenie

Sekunda przestępna, nazywana też **sekundą skokową**

dodatkowa sekunda dodawana czasem (zwykle w czerwcu lub w grudniu) w celu zsynchronizowania uniwersalnego czasu koordynowanego ze średnim czasem słonecznym.

https://www.gum.gov.pl/pl/wiadomosci/informacje-i-komunikaty/sekunda-przestepna/

202

ISO/ANSI C – zakończenie

clock_t clock(void);

Zwraca liczbę „tyknięć" od chwili uruchomienia danego procesu. Aby dostać liczbę sekund należy podzielić tą wartość przez **CLOCKS_PER_SEC**

double difftime(time_t *timer1*, **time_t** *timer0* **);**

Różnica w sekundach pomiędzy dwoma wskazaniami czasu

time_t time(time_t ***timer **);

Aktualny czas systemowy

*Uwaga: zmienna typu **time_t** reprezentuje wskazania czasu od północy, 1 stycznia 1970 do 3:14:07, 19 stycznia 2038*

203

ISO/ANSI C – zakończenie

Przykład:

```
time_t   start, finish;
long loop;
double   result, elapsed_time;

time( &start );
for( loop = 0; loop < 500000000; loop++ )
  result = 3.63 * 5.27;
time( &finish );
elapsed_time = difftime( finish, start );
```


204

ISO/ANSI C – zakończenie

time_t mktime(struct tm *_timeptr_);
Konwertuje dane zapisane w strukturze reprezentującej typ kalendarzowy do postaci wskazania czasu typu time_t

char *asctime(const struct tm *_timeptr_);
Konwertuje dane zapisane w strukturze reprezentującej typ kalendarzowy do postaci tekstowej, np.:
Sun Feb 03 11:38:58 2002

struct tm *localtime(const time_t *_timer_);
Konwertuje wskazanie czasu na typ kalendarzowy wg czasu lokalnego

struct tm *gmtime(const time_t *_timer_);
Konwertuje wskazanie czasu typu time_t do postaci struktury tm

ISO/ANSI C – zakończenie

Przykład:

```
struct tm *newtime;
time_t aclock;

time( &aclock );  /* odczytaj czas */
newtime = localtime( &aclock );  /* Konwertuj do postaci struct tm */

/* Wypisz czas lokalny w oknie konsoli */
printf( „Bieżąca data i czas: %s", asctime( newtime ) );
```


ISO/ANSI C – zakończenie

size_t strftime(char *_strDest_, size_t _maxsize_, const char *_format_, const struct tm *_timeptr_);
Formatuje zapis tekstowy daty i czasu

%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%d	Day of month as decimal number (01 – 31)
%H	Hour in 24-hour format (00 – 23)
%I	Hour in 12-hour format (01 – 12)
%j	Day of year as decimal number (001 – 366)
%m	Month as decimal number (01 – 12)
%M	Minute as decimal number (00 – 59)
%p	Current locale's A.M./P.M. indicator for 12-hour clock
%S	Second as decimal number (00 – 59)
%U	Week of year as decimal number, with Sunday as first day of week (00 – 53)
%w	Weekday as decimal number (0 – 6; Sunday is 0)
%W	Week of year as decimal number, with Monday as first day of week (00 – 53)
%x	Date representation for current locale
%X	Time representation for current locale
%y	Year without century, as decimal number (00 – 99)
%Y	Year with century, as decimal number
%z, %Z	Either the time-zone name or time-zone abbreviation, depending on registry settings; no chars if time zone is unknown
%%	Percent sign

ISO/ANSI C – zakończenie

Przykład:

```
time_t rawtime;
struct tm * timeinfo;
char buffer [80];

time ( &rawtime );
timeinfo = localtime ( &rawtime );

strftime (buffer,80, "Teraz jest %I:%M%p.",
                                        timeinfo);
puts (buffer);
```

Na wyjściu:
Teraz jest 03:21PM.

ISO/ANSI C – zakończenie

Zmienne systemowe (PATH, LIB, itd.) można odczytać używając funkcji `getenv`

```
char *getenv( const char *varname );
```

Przykład:

```
char *pathvar;
pathvar = getenv( "PATH" );
if( pathvar != NULL )
  printf( "PATH variable is: %s\n", pathvar );
```


ISO/ANSI C – zakończenie

Polecenie systemowe można wywołać używając funkcji `system`

```
int system( const char *command );
```

Przykład:

```
system( "type crt.txt" );
/* wypisuje w oknie konsoli zawartość pliku
   tekstowego crt.txt */
```

ISO/ANSI C – zakończenie

Aby zakończyć działanie programu musi być wykonana instrukcja: `exit`, `abort`, lub `return` w funkcji `main`

`void abort( void );`

Przerywa działanie programu, nie zwraca sterowania do procesu nadrzędnego ale wyświetla komunikat:

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

ISO/ANSI C – zakończenie

`void exit( int` *`status`* **`);`**

Funkcja kończy działanie programu w trybie prawidłowego zakończenia

Standardowo przyjmuje się dwie wartości status:
0 (**EXIT_SUCCESS**) – program zakończony prawidłowo;
1 (**EXIT_FAILURE**) - program zakończony niepoprawnie

Przykład:

```
exit( 0 );
```


ISO/ANSI C – zakończenie

Funkcja `exit` przez wyjściem z programu wywołuje wszystkie funkcje wskazane wywołaniem funkcji `atexit`. Może być ich do 32.

Przykład:

```
void fn1(void) {
  printf( "do jutra.\n" );
}
void fn2(void) {
  printf( "Do widzenia " );
}
int main( void ) {
  atexit( fn1 );
  atexit( fn2 );
  exit( 0 );
}                          /* Co pojawi się w oknie konsoli? */
```

Rozszerzenia składni C w środowisku C++

1

Rozszerzenia składni C *vs.* C++

- Nowe symbole komentarzy:
  `// …. tu jest komentarz`
- Możliwość mieszania instrukcji i deklaracji zmiennych w jednym bloku kodu
- Możliwość deklarowania zmiennych np. w instrukcji sterującej pętli **`for`**, np.:
  ```
  for (int i=0; i<100; i++)
  ```
- Nowy typ podstawowy **`bool`**. Zmienne **`bool`** przyjmują dwie wartości **`true`** i **`false`**. Ze względu na wsteczną zgodność jednak pozostawiono domyślną konwersję typu **`bool`** na **`int`**.
  Pisząc w C można jednak było sobie samemu zrobić typ **`bool`** *:*
  ```
  typedef enum {TRUE = 1, FALSE = 0} bool;
  ```
- Deklaracja zmiennych strukturalnych bez słowa **`struct`**, np.:
  ```
  struct Struktura { int a; char c; };
  Struktura X;
  ```


2

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentów *przez odniesienie (referencję)*:

`void fun(int &k)`

Taki zapis oznacza, że funkcji zostanie przekazane odniesienie do zmiennej. Kopia, która zostanie utworzona, będzie korzystała z podanego adresu zewnętrznej zmiennej.

Przykład deklaracji:

```
void fun(int &k){
  k += 2;
};
```

Jak to działa?

3

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentów *przez odniesienie (referencję)*:

`void fun(int &k)`

- W czasie wykonania funkcji nazwa parametru ‘k’ będzie tylko inną nazwą zmiennej podanej w argumencie wywołania
- Wywołanie:

```
int m = 1;
fun(m);
```

spowoduje, że zmienna ‘m’ po zakończeniu działania funkcji będzie miała wartość 3.

4

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentów *przez odniesienie (referencję)*:

Uwaga:

Wywołanie obydwu poniższych funkcji:

```
void fun(int &k)
void fun(int k)
```

wygląda identycznie:

```
fun(m);
```

Natomiast takie wywołanie:

```
fun(m+n);
```

jest dozwolone tylko w przypadku funkcji: **`void fun(int k)`**

5

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentu przez wartość, odniesienie i wskaźnik

```
void fun(int x, int& y, int *z) {
    x = 2*x;
    y = 3*y;
    *z = 4* (*z);
}
```

`x` – przekazywany przez wartość – pracuje na kopii
`y` – przekazywany przez odniesienie – pracuje na oryginale
`z` – przekazywany przez wskaźnik – pracuje na adresie oryginału

Wywołanie:

```
int a=1,b=2,c=3;
fun(a,b,&c);
```


6

Rozszerzenia składni C *vs.* C++

Przekazywanie argumentu przez wartość, odniesienie i wskaźnik

```
void fun(int x, int& y1, int &y2) {
    x = 2*x;
    y1 = 3*y1;
    y1 = y2;
    y1 = 2*y1;
}
```

Raz zainicjalizowana zmienna referencyjna reprezentuje tę samą zmienną aż do końca swego istnienia,

tzn. nie można sprawić, żeby zmienna referencyjna zaczęła od pewnego momentu działania kodu reprezentować inną zmienna.

W szczególności, instrukcja **`y1 = y2`** nie spowoduje, że zmienne referencyjne **`y1`** i **`y2`** zaczną reprezentować tę samą zmienną.

Rozszerzenia składni C *vs.* C++

Przekazywanie tablic w argumencie wywołania funkcji przez odniesienie

```
void funtab(double t[])      // nieznany rozmiar
void funref(double (&t)[6]) // znany rozmiar
```

Wywołanie:

```
double tab[] = { 0 }; /* tablica zer */
funtab(tab);
funref(tab);
```

`funtab` – **`t`** jest typu `double*`; rozmiar tablicy nie jest znany (jest dowolny)

`funref` – **`t`** jest typu *„odniesienie do sześcioelementowej tablicy elementów typu* `double`*”*

Rozszerzenia składni C *vs.* C++

Przekazywanie tablic w argumencie wywołania funkcji przez odniesienie

```
void funref(double (&t)[6])
          /* Nawias: (&t) jest konieczny. */
```

```
double (&t)[6]          double &t[6]
```

Bez nawiasu jest to tablica odniesień do zmiennych typu double – coś takiego nie istnieje w C (!).

- Przy używaniu operatora odniesienia podawanie wymiaru tablicy jest konieczne. *Tablica czteroelementowa to nie to samo co tablica pięcioelementowa.*
- Wywołując **`funtab`** możemy podać tablicę dowolnego rozmiaru. W **`funref`** musi być to tablica dokładnie 6-elementowa.

Rozszerzenia składni C *vs.* C++

Funkcje statyczne

funkcje zadeklarowane jako **`static`**, np.:

```
static int inc(int &x) {
        return ++x;
}
```

Funkcje te są widoczne tylko w obrębie pliku, w którym zostały zadeklarowane, w przeciwieństwie do pozostałych funkcji, które są widoczne we wszystkich plikach należących do projektu (domyślnie: funkcji globalnych).

Rozszerzenia składni C *vs.* C++

Funkcje rozwijane

zadeklarowane z wykorzystaniem słowa kluczowego **`inline`**, np.:

```
inline int pow2(int x) {
        return x*x;
}
```

W miejscach wywołania funkcji kompilator umieszcza kod funkcji zamiast jej wywołania. W ten sposób powstaje szybszy kod (jeżeli funkcja zawiera niewielki kod, to przyrost objętości programu jest minimalny, za to otrzymujemy zysk w postaci większej szybkości)

Rozszerzenia składni C *vs.* C++

Przeciążanie funkcji

zadeklarowanie kilku funkcji o tej samej nazwie, ale różnym zestawie argumentów wywołania , np.:

```
int fun(int a);
int fun(double a);
int fun(char a);
```

Aby funkcję były uważane za różne, muszą różnić się **sygnaturą**. W skład sygnatury wchodzą: nazwa funkcji i lista typów argumentów.

Typ zwracany nie należy do sygnatury.

Rozszerzenia składni C *vs.* C++

Przeciążanie funkcji

Różnienie się sygnaturą nie jest warunkiem wystarczającym, np.:

```
void fun(int k);
void fun(int &k);
```

różnią się sygnaturą, ale to nie wystarcza, aby jednoznacznie rozstrzygnąć, którą funkcję wywołać w przypadku:

```
int k = 0;
fun(k);
```


Rozszerzenia składni C *vs.* C++

Dynamiczny przydział pamięci:

Funkcje biblioteczne malloc, calloc, realloc i free zostają zastąpione operatorami C++:
new i delete

Ogólna postać wyrażenia alokującego:

new *typ*;

gdzie *typ* jest typem zmiennej, która ma zostać zaalokowana na stercie, np.:

```
new int;          new char[100];
```

Wyrażenie new zwraca wskaźnik do nowo utworzonego obiektu dokładnie takiego typu, o jakiego przydzielenie został poproszony, np.:

```
char *buffer = new char[100];
```


Rozszerzenia składni C *vs.* C++

Dynamiczny przydział pamięci:

Ogólna postać wyrażenia zwalniającego:

delete *wskaźnik*;

gdzie *wskaźnik* przechowuje adres do zmiennej dynamicznej i jest dokładnie takiego typu, jak ten zaalokowany na stercie, np.:

```
delete x;   delete []buffer;
```

Uwaga: jeżeli była alokowana tablica, to należy ją zwalniać używając składni właściwej dla tablic