ISO/ANSI C – obsługa błędów

Standardowy strumień dla komunikatów o błędach: **stderr**

Tam zwykle wypisujemy komunikaty o błędach, np.:

```
fprintf( stderr, "%s", "Błąd otwarcia pliku");
```

W przypadku wystąpienia błędów przy wywołaniu funkcji bibliotecznych, zręczniej jest wysłać systemowy komunikat o błędzie do strumienia `stderr` za pomocą funkcji `perror`:

```
void perror( const char *string );          <stdio.h>
```

Zmienna *string* zawiera komunikat użytkownika. Zaraz za nim zostanie wypisany komunikat systemowy o błędzie.

Wywołanie:

```
perror( "Błąd otwarcia pliku" );
```

wypisze w oknie konsoli komunikat będący złożeniem dwóch – komunikatu użytkownika i komunikatu systemowego :

```
Błąd otwarcia pliku: No such file or directory
```


ISO/ANSI C – obsługa błędów

W trakcie działania na pomyślnie otwartym pliku (zapisywania lub odczytywania) mogą również pojawić się błędy.

Aby sprawdzić flagę błędu:

```
int ferror( FILE *stream );
```

z biblioteki **<stdio.h>**

Przykład:

```
FILE * pFile;
pFile=fopen("myfile.txt",„wb");
if (pFile==NULL) perror ("Error opening file");
else {
    fwrite (buffer,sizeof(buffer),1,pFile);
  if (ferror (pFile))
    perror ("Error writing to myfile.txt");
```


ISO/ANSI C – obsługa błędów

Aby oczyścić flagi błędu i flagę końca pliku dla otwartego pliku:

```
void clearerr( FILE *stream );
```

z biblioteki **<stdio.h>**

Flagi błędu i końca pliku nie są automatycznie czyszczone, ale pozostają, póki nie zostanie wywołana funkcja **clearerr**, **fseek**, **fsetpos**, lub **rewind**.

W przeciwnym razie każda kolejna próba działania na takim pliku będzie ciągle zwracała raz ustawiony kod błędu.

ISO/ANSI C – obsługa błędów

Wyszukiwanie błędów w nowym kodzie

Oprócz błędów wynikających z zewnętrznych warunków w których pracuje program są jeszcze błędy, których źródłem jest sam autor kodu, tj. błędy związane z niepoprawnym zakodowaniem algorytmu

Typowy błąd: dopuszczenie do sytuacji, w której program nadał zmiennej wartość spoza zakresu przewidzianego przez programistę, tj. dla której nie ma właściwej obsługi i stąd program próbował:

- dzielić przez zero
- obliczyć pierwiastek z liczby ujemnej
- robić coś równie nierozsądnego

ISO/ANSI C – obsługa błędów

Intuicyjne rozwiązanie: dodanie sprawdzenia, czy zmienna ma wartość należąca do dozwolonego zakresu:

Przykład:

```
void printd(int n) {
if (n<=0) {/* n może mieć tylko wartości dodatnie */
    printf("uwaga: n<=0!\n");
    return; }
…
```

To zwiększa liczbę linii kodu, komplikuje kod i spowalnia działanie programu.

Takie sprawdzenia stają się zbędne, kiedy już program jest wytestowany i wiadomo, że funkcja NIGDY nie zostanie wywołana z argumentem o wartości 0 lub mniejszej.

ISO/ANSI C – asercje

Potrzebny jest prosty mechanizm, który będzie sprawdzał poprawność warunku logicznego, sygnalizował, kiedy jest nie spełniony, oraz znikał, kiedy tworzona jest finalna postać programu.

```
void assert( int expression );    <assert.h>
```

Sprawdza wartość wyrażenia logicznego i jeżeli jest spełnione, nie robi nic. W przeciwnym przypadku przerywa działanie programu i wyrzuca komunikat do standardowego strumienia wyjściowego. Komunikat zawiera: treść warunku, nazwę pliku źródłowego i numer linii.

ISO/ANSI C – asercje

Przykład:

```
void printd(int n) {
    assert(n>0);
    …
```

Jeżeli zmienna n ma wartość większą od zera, nie dzieje się nic.

W przeciwnym przypadku dostaniemy w oknie konsoli komunikat:

```
Assertion failed: n>0, file main5.c, line 34

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

Ten komunikat zawiera wszystkie niezbędne informacje, aby ułatwić znalezienie błędu w swoim programie.

174

ISO/ANSI C – asercje

Czy przerwanie działania programu to nie jest zbyt drastyczne rozwiązanie?

W praktyce znacznie gorszym pomysłem byłoby dopuszczenie do dalszego działania programu w sytuacji, kiedy nie są spełnione podstawowe założenia projektanta i program wymaga poprawek.

175

ISO/ANSI C – asercje

Użycie

```
assert(n>0);
```

jest prostsze i wymaga mniej pisania niż:

```
if (n<=0) {
  printf(„uwaga: n<=0!”);
  exit;
}
```

Ponadto *znika* w finalnej wersji programu. ☺

176

ISO/ANSI C – asercje

- Aby **assert** zadziałał, kompilacja kodu musi być wykonywana w trybie generującym dodatkowe informacje dla debuggera. Ten tryb pozwala np. na krokowe wykonanie kodu i obserwowanie wartości zmiennych w poszczególnych krokach wykonania, co jest przydatne podczas tworzenia i poprawiania kodu.
- Finalna wersja kodu (tzw. *release*) jest generowana zawsze przy *wyłączonym* trybie generowania dodatkowych informacji dla debugera (wtedy program jest mniejszy i działa szybciej).
- W tym trybie wszystkie wywołania **assert** są ignorowane przez kompilator dzięki odpowiednim dyrektywom preprocesora w kodzie - żaden kod dla wywołań tej instrukcji nie jest generowany.

177

ISO/ANSI C – asercje

Uwaga!

Ponieważ po wyłączeniu trybu debugera asercje znikają z programu, nie wolno umieszczać w nich instrukcji dokonujących zmiany wartości zmiennych w programie, lub dokonujących jakichkolwiek innych działań na danych, np.:

```
func(void)
{
  int c;
  assert((c = getchar()) != EOF);
  putchar(c);
}
```


178

ISO/ANSI C – asercje

Tryb kompilacji można ustawić poprzez napisanie odpowiedniej instrukcji w pliku programu

Ta instrukcja to utworzenie określonej stałej. Tworzymy ją używając dyrektywy preprocesora **`#define`**

Użycie `#define` - przykłady:

```
#define MyName Krzysztof
#define TrybCichy
```

W drugim przypadku stała **`TrybCichy`** ma wartość pustą, ale istnieje, dlatego można teraz sprawdzać jej zdefiniowania pisząc:

```
#ifdef TrybCichy
…
#endif
```


179

ISO/ANSI C – asercje

Aby wyłączyć sprawdzanie asercji można przed odwołaniem się do biblioteki `assert.h` utworzyć stałą **NDEBUG**.

Przykład:

```
#define NDEBUG
#include <assert.h>
…
```

Trzeba jednak być świadomym ewentualnych skutków ubocznych takiego postępowania. Jeżeli nie wiesz, jakie skutki może powodować w kodzie użycie dyrektyw wpływających na działanie kompilatora, to jest pierwszy powód, dla którego nie powinieneś ich używać.

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.

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?

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. ☹

ISO/ANSI C - biblioteki

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.

ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych

Deklaracja:

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

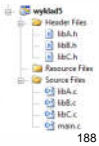
Definicja:

```
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;
};
```


ISO/ANSI C - biblioteki

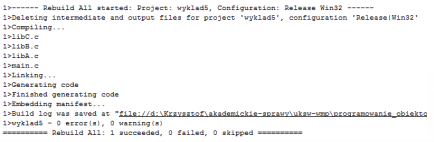
Dzielenie kodu na kilka plików źródłowych

- 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.



ISO/ANSI C - biblioteki

Dzielenie kodu na kilka plików źródłowych

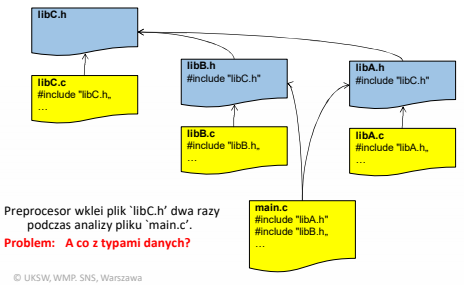


Proces kompilacji jest dwuetapowy:

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

ISO/ANSI C - biblioteki

Przykład:



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

Problem: A co z typami danych?

ISO/ANSI C - biblioteki

Budowa pliku nagłówkowego *.h

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

Wykorzystać dyrektywy preprocesora:

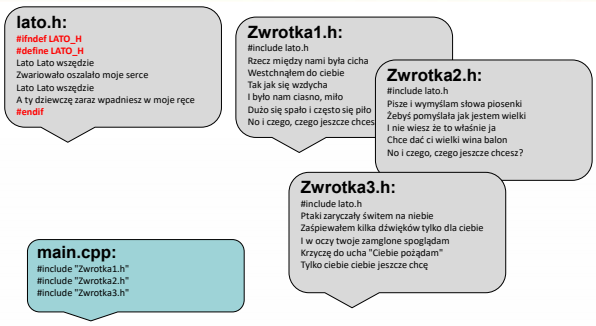
```
#ifndef nazwaPliku_h
#define nazwaPliku_h
/*
   tutaj deklaracje funkcji
*/
#endif
```

#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

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

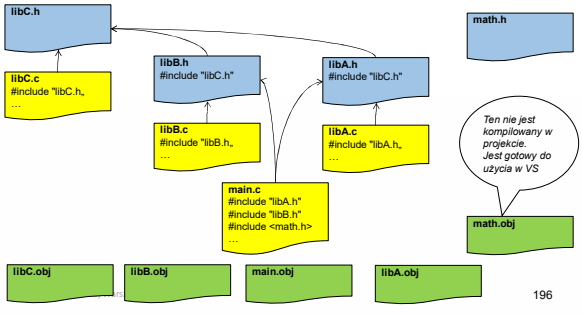
```
#pragma once
```

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

195

ISO/ANSI C - biblioteki

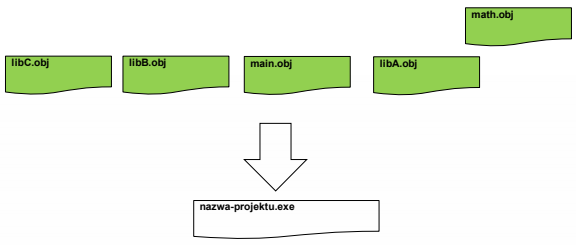
Kompilacja plików z projektu:



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

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 );
```


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.