157

ISO/ANSI C – typy pochodne

Najbardziej nawet złożone struktury mają swój typ, który można zapisać za pomocą istniejących podstawowych typów danych

Przykład:

```
int tab[] = {1,2,3};
int *y[3] = { tab, tab, tab };
int (*z)[3] = &tab;
```

Takim złożonym typom można nadać *aliasy* (nazwy) za pomocą deklaracji **typedef**

Przykład:

```
typedef int T[];
typedef int *T3Wsk[3];
typedef int (*WskT3)[3];

T tab = {1,2,3};
T3Wsk tab1 = {tab, tab, tab };
WskT3 tab2 = &tab;
```


158

159

ISO/ANSI C – obsługa błędów

Poprawa obsługi błędów to jedna z najpewniejszych metod tworzenia solidnie działającego kodu

Źródła błędów w działających programach – sytuacje niezgodne z oczekiwaniami autora programu, ale potencjalnie możliwe do wystąpienia

Przyczyna:

fałszywe oczekiwania autora programu wynikające z lenistwa, albo z „wishful thinking" czyli pobożnych życzeń autora, że wszystkie warunki prawidłowej pracy zostaną spełnione przez użytkownika, zanim uruchomi on program, albo..

160

ISO/ANSI C – obsługa błędów

Obsługa błędów w C polega na kontrolowaniu rezultatów wykonania wywoływanych funkcji i powiązaniu z tymi rezultatami kodu obsługującego błąd.

Przykład:

Otwieranie pliku (*może się udać, lub nie*).

```
FILE* stream;
char s[100];
…
stream  = fopen( nazwa_pliku, "r" ));
fscanf(stream, "%s", s);
```

Czy ten kod jest bezpieczny?

161

ISO/ANSI C – obsługa błędów

Przykład:

```
FILE* stream;
char s[100], file_name[100];
…
stream  = fopen( nazwa_pliku, "r" ));
if (stream == NULL) /* sprawdzamy, czy otwieranie się powiodło */
        ...
```

Co powinno się znaleźć w miejscu kropeczek?

1. Przerwanie pracy całego programu
2. Wyświetlenie komunikatu o problemie i powrót sterowania do początku działania programu
3. Powrót sterowania do miejsca, w którym pobierana jest nazwa pliku

162

ISO/ANSI C – obsługa błędów

Podstawowa strategia obsługi błędów – sprawdzanie wartości zwracanej przez funkcję biblioteczną, której wywołaniu mogło towarzyszyć pojawienie się błędu

Jeżeli informacje zwracane przez funkcję biblioteczną są zbyt proste, dodatkowo wykorzystywana jest zmienna globalna przechowująca kod błędu: **`errno`**

Zmienna **`errno`** ma ustawiany kod błędu w momencie wywołania żądania systemowego, takiego jak np. próba otwarcia pliku.

Należy ja od razu odczytać, bo następne wywołanie może spowodować kolejną zmianę jej wartości.

Mając kod błędu możemy zażądać tekstu odpowiadającego temu kodowi i w ten sposób dowiedzieć się czegoś więcej

ISO/ANSI C – obsługa błędów

Constant	System error message	Value
E2BIG	Argument list too long	7
EACCES	Permission denied	13
EAGAIN	No more processes or not enough memory or maximum nesting level reached	11
EBADF	Bad file number	9
ECHILD	No spawned processes	10
EDEADLOCK	Resource deadlock would occur	36
EDOM	Math argument	33
EEXIST	File exists	17
EINVAL	Invalid argument	22
EMFILE	Too many open files	24
ENOENT	No such file or directory	2
ENOEXEC	Exec format error	8
ENOMEM	Not enough memory	12
ENOSPC	No space left on device	28
ERANGE	Result too large	34
EXDEV	Cross-device link	18

ISO/ANSI C – obsługa błędów

Jeżeli chcemy wyświetlić użytkownikowi komunikat z tekstem odpowiadającym kodowi błędu:

char *strerror(int *errnum* **);**

Funkcja zwraca tekst, który odpowiada opisowi kodu błędu.

```
file = fopen(fname, "r");
if (file == NULL) {
  printf("Error while trying to open '%s': %s\n",
                                  fname, strerror(errno));
  ….
```


ISO/ANSI C – obsługa błędów

Zamiast:

```
file = fopen(fname, "r");
if (file == NULL) {
  printf("Error while trying to open '%s': %s\n",
                                  fname, strerror(errno));
  ….
```

Można też napisać:

```
file = fopen(fname, "r");
if (file == NULL) {
  printf("Error while trying to open '%s': %s",
                                  fname, strerror(NULL));
  ….
```

Wtedy zostanie wypisany komunikat właściwy dla ostatniego wywołania, które wygenerowało błąd, zakończony znakiem nowej linii.

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 );
```

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 );
```

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 );
```

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 – obsługa błędów

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 – obsługa błędów

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.

ISO/ANSI C – obsługa błędów

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.

ISO/ANSI C – obsługa błędów

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. ☺

ISO/ANSI C – obsługa błędów

- 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.

ISO/ANSI C – obsługa błędów

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);
}
```


ISO/ANSI C – obsługa błędów

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
```


ISO/ANSI C – obsługa błędów

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.