Header assert.h

Header	Opis zawartych funkcji	Funkcja makro
assert.h	Zawiera informacje dotyczące dodawania diagnostyki, które ułatwiają debugowanie programu.	assert()

Assert - Wikipedia:

> Asercja (ang. *assertion*) – predykat (forma zdaniowa w danym języku, która zwraca prawdę lub fałsz), umieszczony w pewnym miejscu w kodzie. Asercja wskazuje, że programista zakłada, że ów predykat jest w danym miejscu prawdziwy. W przypadku gdy predykat jest fałszywy (czyli niespełnione są warunki postawione przez programistę) asercja powoduje przerwanie wykonania programu.

Aby natychmiast przybliżyć ideę ***assert()***, podaję poniższe trzy punkty przed późniejszą głęboką ich analizą:

- Aby zastosować ***assert()*** musimy zdawać sobie sprawę, jakie 'niechciane' przypadki mogą wystąpić i w zapisie **assert(***wyrażenie***)** w miejscu '*wyrażenie*' piszemy to, co unika 'niechnianego' przypadku, np. przy dzieleniu `a/b` piszemy **`assert( b != 0 )`**, wtedy przy `b==0` makro ***assert()*** w sposób kontrolowany przerwie działanie programu.
- Gdy 'niechcianych' przypadków jednego zapisu jest kilka, to powtarzamy zapis ***assert()*** dla każdego z nich osobno - poniżej jest tego przykład dla szeregu warunków wprowadzania hasła.
- ***assert()*** dla 'niechcianych' warunków, zawsze przerywa wykonanie programu, ponieważ znakomicie się nadaje do diagnostyki testowanego kodu.

#include<assert.h>

- **assert(**wyrażenie**)** - sprawdza wartość wyrażenia;
  - jeżeli jest fałszem to wypisuje komunikat diagnostyczny i przerywa działanie programu; komunikat diagnostyczny zawiera wyrażenie, które zakończyło się niepowodzeniem, nazwę pliku źródłowego i numer wiersza, w którym asercji się nie powiodło
  - jeżeli jest prawdą to nie jest podejmowana żadna akcja.

Definicja w pliku nagłówkowym **assert.h**:

```
#define assert(p) ((p) ? (void)0 : (void) __assertfail( \
                    "Assertion failed: %s, file %s, line %d" _ENDL, \
                    #p, __FILE__, __LINE__ ) )
```

Cały ten zapis to 'wyrażenie warunkowe', gdzie:

```
'p'                 <-- to 'wyrażenie' z definicji assert()
(p) = (void)0       <-- nie rób nic jak 'wyrażenie' jest PRAWDĄ
(p) = (void) __assertfail( \
                    "Assertion failed: %s, file %s, line %d" _ENDL, \
                    #p, __FILE__, __LINE__ ) )
                    <-- gdy wyrażenie jest FAŁSZEM wyświetla:
                      - zwrot "Assertion failed: ",
                      - samo 'wyrażenie' (tu: p) w łańcuchu znakowym (%s),
                      - nazwę programu jako łańcuch znakowy (%s),
                      - numer linii w programie (int, %d), gdzie assert() zastał 'błąd'.
```

Przypominam, że `\` tuż przed `[Enter]` jest pomijany przez kompilator i służy do łamania linii długiego zapisu w celu wyłącznie praktycznym.

Ten zapis może się różnić w zależności od wersji 'C' (jak też w wielu innych językach programowania) ale idea ***assert()*** jest niezmienna.

Opisy sytuacyjne zastosowania assert()

Tak więc makro ***assert()*** może diagnozować błędy programu. Jest ono zdefiniowane w funkcji ASSERT.H, a jego prototyp to

```
void assert(int wyrażenie);
```

Argument '*wyrażenie*' może być dowolny, co tylko chce się przetestować - zmienna lub dowolne wyrażenie języka C. Jeśli wyrażenie ma wartość PRAWDA, ***assert()*** nic nie robi. Jeśli wyrażenie ma wartość FAŁSZ, ***assert()*** wyświetla komunikat o błędzie na *stderr* (czyli na ekran monitora) i przerywa wykonywanie programu.

Jak używać ***assert()***? Jest on najczęściej używany do śledzenia błędów programu (które różnią się od błędów kompilacji). Na przykład program do analizy finansowej, który ktoś pisze, może czasami udzielać nieprawidłowych odpowiedzi. Można podejrzewać, że problem jest spowodowany tym, że zmienna *stopa_procentowa* przyjmuje wartość ujemną, co nigdy nie powinno mieć miejsca. Aby to sprawdzić, warto umieścić instrukcję

```
assert(stopa_procentowa >= 0);
```

w programie, tuż przed użyciem zmiennej *stopa_procentowa* w szczególności, gdy jej wartość poddawana jest operacji matematycznej. Jeśli ta zmienna w tym miejscu stanie się ujemna, makro ***assert()*** ostrzeże użytkownika.

Przykład 1 - najbardziej prosty scenariusz (tu: unikanie przypadku, gdy x=0)

```c
/* Makro assert() */
#include <stdio.h>
#include <assert.h>             // ten temat
#include <locale.h>             // dla 'setlocale()'

main()
{  setlocale(LC_CTYPE, "Polish"); //polskie znaki
   int x;
   printf("\nPodaj dowolną liczbę całkowitą: ");
   scanf("%d", &x);
   assert(x);   // x==0 to fałsz, dlatego na tej wartości x, assert przerwie program
   printf("Wprowadziłeś %d.", x);
}
```

Jeśli wprowadzimy wartość różną od zera, program wyświetli tę wartość i zakończy działanie w normalny sposób. Jeśli wprowadzimy zero, makro ***assert()*** wymusi kontrolowane wstrzymanie działania programu.

Wyświetlany komunikat o błędzie podany jest w drugim przykładzie poniżej i obejmuje:

```
Assertion failed!
Program: ścieżka do programu egzekucyjnego (z rozszerzeniem .exe)
File: ścieżka do programu języka C (z rozszerzeniem .cpp)
Line numer linii w programie z rozszerzeniem .cpp, w której wystąpił 'błąd'
Expression wyrażenie (u nas 'x', ale tak naprawdę to 'x!=0')
```

Zauważ, że fraza: *Wprowadziłeś...* już się nie pojawi. Porównaj to z przykładem nr 3 poniżej.

Dla liczby różnej od zera:

```
Podaj dowolną liczbę całkowitą: 10
Wprowadziłeś 10.
--------------------------------
Process exited after 1.92 seconds with return value 0
Press any key to continue . . . _
```

Dla zera:

```
Podaj dowolną liczbę całkowitą: 0
Assertion failed!

Program: C:\Users\Artur\Desktop\assert1.exe
File: C:\Users\Artur\Desktop\assert1.cpp, Line 11

Expression: x

--------------------------------
Process exited after 2.052 seconds with return value 3
Press any key to continue . . . _
```

(zauważ kod powrotu == 3)

Działanie ***assert()*** zależy od innego makra o nazwie **NDEBUG** (od "**N**o **DEBUG**ging"). Jeśli makro **NDEBUG** nie jest zdefiniowane (a jest to stan domyślny), funkcja ***assert()*** jest aktywna. Jeśli **NDEBUG** jest zdefiniowane, ***assert()*** jest wyłączone i nie ma żadnego efektu.

(Na podstawie: Peter Aitken & Bradley Jones: Teach Yourself C in 21 Days, strona 524.)

Przykład 2 - seria 'niechcianych' przypadków

W tym programie funkcja używa makra **sprawdzanie** do testowania ***assert()*** kilku warunków związanych z wprowadzeniem łańcucha znakowego. Pierwszy 'niechciany' przypadek spowoduje wyświetlenie komunikatu wskazującego powód 'błędu' i przerwanie działania programu.

```c
// Obwarowane warunkami wprowadzanie hasła

#include <stdio.h>
#include <assert.h>                // ten temat
#include <string.h>
#include <locale.h>                // dla 'setlocale()'

void sprawdzanie( char *haslo );   // prototyp funkcji

int main( void )
{  setlocale(LC_CTYPE, "Polish");  //polskie znaki
  // do test3[] = "" kompilator wrzuca '\0'
  // podobnie jest z test1[] = "Artur", który automatycznie dostaje na końcu znak '\0'
   char test1[] = "Artur", test2[] = "Al", *test3 = NULL, test4[] = "",
        test5[] = "Grzegorz Brzenczyszczykiewicz";

   printf ( "Wpisane hasło: '%s'\n", test1 ); sprawdzanie( test1 );
   printf ( "Wpisane hasło: '%s'\n", test2 ); sprawdzanie( test2 );
   printf ( "Wpisane hasło: '%s'\n", test3 ); sprawdzanie( test3 );
   printf ( "Wpisane hasło: '%s'\n", test4 ); sprawdzanie( test4 );
```

```
   printf ( "Wpisane hasło: '%s'\n", test5 ); sprawdzanie( test5 );
}

// testowanie hasła na kilka sposobów. Pierwszy niespełniony test zamyka program
void sprawdzanie( char *haslo )
{
   assert( haslo != NULL );          // zmienna 'haslo' nie może nie istnieć
   assert( *haslo != '\0' );         // 'haslo' nie może być puste
   assert( strlen( haslo ) > 3 );    // 'haslo' musi mieć więcej niż 3 znaki
   assert( strlen( haslo ) <= 20 );  // długość 'haslo' nie może być większa niż 20
                                     //    znaków
}
```

Oczywiście w realnym scenariuszu wartość zmiennej **haslo** byłaby przyjmowana przez **scanf()** i przechodziłaby przez wszystkie cztery przypadki **assert** wyszczególnine przez funkcję **sprawdzanie** chyba, że zatrzymana byłaby przez pierwsze niespełnienie jednego z czterech warunków.

Wynik:

```
Wpisane hasło: 'Artur'
Wpisane hasło: 'Al'
Assertion failed!

Program: C:\Users\Artur\Desktop\assert2.exe
File: C:\Users\Artur\Desktop\assert2.cpp, Line 29

Expression: strlen( haslo ) > 3

--------------------------------
Process exited after 3.091 seconds with return value 3
Press any key to continue . . . _
```

Zmień kolejność test1, test2 itd. aby sprawdzić wszystkie cztery opcje kontrolowanego wstrzymania działania programu.

Przykład 3 - tworzenie własnego ***assert()*** z pominięciem tego z pliku nagłówkowego **assert.h**

Jedną z najważniejszych cech makra ***assert()*** jest to, że preprocessor nie dba o jego kod jeśli DEBUG nie jest zdefiniowany. Jest to bardzo pomocne podczas programowania, a przy otrzymaniu wyniku końcowego nie ma spadku wydajności programu ani zwiększenia rozmiaru wykonywalnej (z rozszerzeniem .exe) wersji programu.

Zamiast polegać na dostarczonym firmowo ***assert()***, można napisać własne makro *assert()*.
W dodatku nasz własny *assert()* może, ale nie musi powodować przerwania działania programu.

```
/* Nasz prywatny assert - napisany po polsku */
#define DEBUG
#include <stdio.h>

#ifndef DEBUG
   #define ASSERT(x)
#else
   #define ASSERT(x) \
       if ( !(x) )   \
       {  \
```

```
          printf("   ERROR! Nieudana asercja dla: %s \n", #x); \
          printf("   w linii numer %d \n", __LINE__ ); \
          printf("   programu %s \n", __FILE__ ); \
       }
#endif

int main()
{
    int x = 5;

    printf("Pierwsza asercja: \n");
    ASSERT(x==5);
    printf("   Tu asercja przebiega poprawnie. \n");

    printf("\nDruga asercja: \n");
    ASSERT(x != 5);
    printf("\nZrobione. Koniec programu.\n");
    return 0;
}
```

W linii drugiej (2) od góry powyższego kodu zdefiniowany jest termin DEBUG.

W liniach 8-14 zdefiniowane jest makro *assert()*.

Zazwyczaj kod ten byłby w pliku nagłówkowym, a ten plik (np. ASSERT.HPP) zostałby uwzględniony we wszystkich wersjach plików egzekucyjnych (tu starałem się tak dobrać słowa, aby nie było w nim typowych polskich znaków, mając na uwadze przerzucenie tego kodu do dowolnego programu języka "C").

Sam ***assert()*** jest jedną długą instrukcją, przynajmniej dla prekompilatora, tu podzieloną na siedem linii kodu.

W linii 9 testowana jest wartość przekazana jako parametr (może tu być przecież złożone wyrażenie); jeśli zostanie zwrócona wartość FALSE, uruchamiane są instrukcje w liniach 11-13, wypisując komunikat o błędzie. Jeśli przekazana wartość ma wartość TRUE, nie jest podejmowana żadna akcja (tu wypisałem zdanie: "Tu asercja przebiega poprawnie" aby zaznaczyć, że nic tu złego się nie dzieje).

Debugging* z użyciem assert()

Naturą błędów jest, że to, o czym sądzi się, że jest prawdą, może nie być prawdą w pewnych warunkach. Na przykład wiadomo, że wskaźnik (*pointer*) jest prawidłowo ustawiony, ale program się 'wywala' **. ***assert()*** może pomóc w znalezieniu tego typu błędów, ale tylko wtedy, gdy regularną praktyką jest dla programisty lub testera bezbłędne używanie instrukcji ***assert()*** w kodzie. Za każdym razem, gdy przypisuje on wartość do zmiennej lub przekazuje wskaźnik jako parametr lub wartość zwracaną przez funkcję, powinien upewnić się, że dla funkcji ***assert()*** wskaźnik ten jest ustawiony prawidłowo. Za każdym razem, gdy kod zależy od określonej wartości znajdującej się pod zmienną, upewnij się, że jest to prawda.

Właśnie słowo **assert** w nazwie jej funkcji należałoby przeczytać: ***sprawdź***(*czy ten warunek jest prawdą*) lub ***upewnij_się***(*że ten warunek jest prawdą*).

Nie ma kary za częste używanie instrukcji ***assert()***; przypominam, że jej działanie jest usunięte z kodu - każdy ***assert()*** przestaje być aktywny - po cofnięciu definicji debugowania (makrem **NDEBUG**).

- - - - - -

* Dosłownie 'odrobaczywianie'. Termin powstały po odkryciu powodu zwarcia instalacji dużego komputera (*mainframe*) - był nim robak, którego dołączono do raportu o przyczynie zwarcia.

** Po angielsku: *dump*, z danymi z miejsca przerwania działania programu do miejsca zwanego *log*'iem, na pierwszy rzut oka tak trudnymi do odczytu, że przypominającymi wysypisko (*dump*) śmieci.

Efekt błędu programisty, zwanego tu efektem ubocznym, którego to błędu assert() nie jest w stanie wykryć

'Najciemnej jest pod latarnią' - jeżeli błąd wystąpi w samym zapisie makra ***assert()***, to sam ***assert()*** tego błędu nie wykryje.

Nierzadko zdarza się, że błąd pojawia się dopiero po usunięciu instrukcji ***assert()***. Jest to prawie zawsze spowodowane tym, że program nieumyślnie podlega efektom ubocznym procesów wykonywanych w instrukcji ***assert()*** i innych kodach tylko do debugowania. Na przykład, jeśli napisze się

`ASSERT (x = 5)` zamiast `ASSERT (x == 5)`

kiedy chce się sprawdzić, czy x = 5, stworzy się szczególnie paskudny błąd.

Powiedzmy, że tuż przed tym *assert()* wywołuje się funkcję, która ustawiła x na równe 0. Za pomocą tego *assert()* można myśleć, że sprawdzane jest, czy x jest równe 5. W rzeczywistości ustawiane jest x na równe 5. Test zwraca wartość prawdy (TRUE), ponieważ x = 5 nie tylko ustawia wartość x na 5, ale też zwraca wartość 5, a ponieważ 5 jest różna od zera, test zwraca wartość TRUE.

Kiedy *assert()* przeszedł pomyślnie test, x naprawdę jest równe 5 (po prostu to tester ustawia!), program działa dobrze i jest gotowy do wysłania (czytaj: implementacji), więc tester wyłącza debugowanie. Teraz *assert()* znika i nie ustawia się już x na 5. Ponieważ x zostało ustawione na 0 tuż przed tym testem, pozostaje na 0, a program nieoczekiwanie ... się 'wywala'.

Sfrustrowany programista/tester ponownie włącza debugowanie, a tu *hokus pokus*! Błąd zniknął. To jak zabawa w 'kotka i myszkę' (po angielsku: *Mickey Mouse behaviour* - nieprzewidywalne zachowanie), ale nie zabawne w życiu programisty, więc bądź bardzo uczulony na skutki uboczne w debugowaniu kodu. Jeśli widzisz błąd, który pojawia się tylko wtedy, gdy debugowanie jest wyłączone, przyjrzyj się kodowi debugowania, zwracając uwagę na jego nieprzyjemne skutki uboczne.

Wynik:

```
Pierwsza asercja:
   Tu asercja przebiega poprawnie.

Druga asercja:
   ERROR! Nieudana asercja dla: x != 5
   w linii numer 24
   programu C:\Users\Artur\Desktop\assert3.cpp

Zrobione. Koniec programu.
--------------------------------
Process exited after 2.564 seconds with return value 0
Press any key to continue . . . _
```

(Na podstawie: Jesse Liberty, Teach Yourself C++ in 24 hours, strona 339)