Przekazywanie różnych opcji jako jeden parametr (opt1 | opt2) (sheadovas/poradniki/howto/przekazywanie-opcji-jako-jeden-parametr/)

Mar 23, 2017 / howTo (sheadovas/category/poradniki/howto/)

Kilka słów o przekazywaniu opcji typu: *std::fstream.open(file, std::ios::out | std::ios::bin | std::ios::trunc)*.

Hej, w dzisiejszym wpisie chciałbym zająć się dość prostym zagadnieniem, które niekoniecznie jest oczywiste szczególnie dla osób zaczynających zabawę z programowaniem.

Wstęp

Mianowicie chodzi mi konstrukcje pokazane w poniższych kodach:

Listing C++

```
// Example 1
std::fstream file;
file.open(path_to_file,
          std::ios::out | std::ios::bin | std::ios::trunc);


// Example 2
sf::RenderWindow window(/* some params */,
                        sf::Style::Titlebar | sf::Style::Resize);
```

Pierwszy przykład powinien być powszechnie znany – ustawiamy w nim plik w trybie do zapisu (binarnie) wraz z usunięciem zawartości przy jego otworzeniu.

Drugi przykład pochodzi z SFML i tworzy okno z nagłówkiem (titlebar) oraz możliwością zmiany jego rozmiaru.

W obu przypadkach wszelkie opcje ustawiane są za pomocą 1 parametru przy użyciu magicznej pałki. Przyznam szczerze, że w czasach mojej nauki C++ ten mechanizm pozostawał dla mnie zagadką.

Dzisiaj chciałbym Wam przedstawić, że ten mechanizm jest banalnie prosty, zarówno do stworzenia jak i późniejszej obsługi.

Jak to działa

Aby użyć systemu tego typu należy pamiętać o jednej rzeczy: liczby zapisywane są binarnie. Co za tym idzie, to każdą zmienną możemy traktować jak kontener na flagi różnej długości (zależnie od typu), gdzie każdy element może przyjmować wartości 0 lub 1.

Sama zasada działania na kontenerach tego typu wymaga użycia operatorów logicznych, szczególnie przydatne okażą się:

- | – *or*, suma logiczna;
- & – *and*, iloczyn logiczny
- << – *shift-left*, przesunięcie bitów w lewo

Warto zdać sobie sprawę jak one działają, poniżej przedstawiam krótkie wyjaśnienie.

OR (|)

```
# Po prostu OR każdego bitu z każdym, wg tabeli prawdy

0 0 | 0
0 1 | 1
1 0 | 1
1 1 | 1

Np.
1000 1110
0111 0001
---------
1111 1111 <- wynik
```

AND (&)

```C++
Iloczyn logiczny każdego bitu z każdym, stosowany do tzw "masek bitowych".
Uzupełniany wg tabeli prawdy:

0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1

Np.
1001 0001
1111 0000
---------
1001 0000
```

SHL (<<)

```C++
Przesunięcie bitów (w lewo), u nas sprawa bo przy zapisie '1<<k' rozumiemy:
'przesuń "1" na "k-tą" pozycję"

Np.

mamy liczbę: 0000 0000
wykonujemy przesunięcie: 1 << 0
czyli przesuwamy 1 o "0" pozycji, a więc mamy: 0000 0001
```

Pierwsze podejście

Jak pewnie część z Was wie do sprawdzenia parzystości liczby nie musimy wykonywać operacji modulo, wystarczy że sprawdzimy wartość najmłodszego bitu liczby – można go traktować jako flagę nieparzystości:

```C++
if(value & 1)
  printf("liczba nieparzysta!\n");
else
  printf("liczba parzysta!\n");
```

Jak widzimy prostą sztuczką zaoszczędziliśmy całkiem sporo w stosunku do operacji modulo. Nic nie stoi na przeszkodzie aby pójść dalej i sprawić aby każdy z bitów liczby mógł mieć jakieś inne znaczenie, np. mówił czy plik jest w trybie do odczytu, zapisu, itd. Jak wspomniałem: liczbę można traktować jako tablicę wartości bool'owskich.

Poniżej przedstawiam prosty przykład pokazujący tworzenie „tablicy 8-elementowej" oraz zapalenie 3 flagi, a później jej odczytanie:

```C++
```

```
/* stworzenie "tablicy", wszystkie bity ma wygaszone */
uint8_t flags = 0; // 0000 0000

/* podniesienie flagi o indeksie = 3 */
flags = (1 << 3); // 0000 1000

/* sprawdzenie czy trzeci bit jest zapalony */
if(flags & (1<<3)) // [0000 1000] & [0000 1000]
  puts("true");
else
  puts("false");
```

Jeżeli dodatkowo chcielibyśmy podnieść jeszcze 5 bit, to musimy zastosować or'a:

C++

```
/*
  0000 1000 flags
+ 0010 0000 (1<<3)
-----------
  0010 1000
*/
flags |= (1 << 5); // 0010 1000
```

W ten właśnie sposób działają łączone opcje! Wystarczy, że każda z opcji przyjmie inne przesunięcie „1", po czym zostanie połączona operatorem „|". W następnym paragrafie pokazuję nieco większy przykład, który całą ideę demonstruje w bardziej uporządkowany sposób.

Duży przykład

Załóżmy, że piszemy bibliotekę oferującą obsługę na plikach. Przy otwarciu pliku możemy posłużyć się zestawem różnych opcji, wszystkie domyślnie są wyłączone. A są nimi:

- tryb do odczytu;
- tryb do zapisu (można otworzyć plik jednocześnie do zapisu i odczytu);
- tryb binarny lub tekstowy;
- otwarcie pliku z weryfikacją CRC.

Oprócz tego zakładamy, że mogą się pojawić jeszcze inne opcje i z różnych względów chcemy zostawić sobie 4 pola zarezerwowane.

C

```c
#include <stdio.h>
#include <inttypes.h>

/* file mode */
#define READ  (1 << 0)
#define WRITE (1 << 1)

/* modes: text=0, binary=1 */
#define MBIN  (0 << 2)
#define MTEXT (1 << 2)

/* calc crc */
#define CRC   (1 << 3)

/* other options */
#define OPT1  (1 << 4)
#define OPT2  (1 << 5)
#define OPT3  (1 << 6)
#define OPT4  (1 << 7)

/* mixed */
#define RW    (READ | WRITE)
```

Powyżej zdefiniowaliśmy sobie dla wygody wszystkie pola, w gruncie rzeczy można by wrzucić wszystkie wartości do *enum*, ale my trzymamy się **konwencji** C, więc skorzystamy z *define*,ów ;)

> Uwaga! Wiem, że *enumy* są w C, tutaj zastosowałem konwencję stosowaną np. do tworzenia quirków

W ostatnie linii dodatkowo zdefiniowaliśmy sobie flagę, która jest sumą dwóch wcześniejszych, zrobiliśmy to dla wygody użytkownika – założyliśmy że te opcje mogą pojawiać się razem dość często, więc czemu by nie pomóc użytkownikowi.

Zauważamy też, że mamy tylko 1 bit odpowiadający za tryb tekstowy lub binary, wynika to właśnie z tego że plik zawsze musimy uruchomić w jakimś trybie (jednym z tych dwóch) i może być tylko „1”. Bez sensu byłoby komplikować sobie życie używając kolejnego bitu, skoro tak nie musimy sprawdzać czy jakiś bit jest podniesiony, a nie czy np. obydwa.

C

```c
typedef uint8_t fileopts_t;

#define IS_SET(opt, reg) ((opt) & (reg) ? 1 : 0)
```

Kolejnym krokiem jest stworzenie *typedef*,a na uinta, oraz stworzenie prostego makra zwracającego 1 lub 0, w celu poinformowaniu nas o tym czy flaga jest podniesiona czy nie.

C

```c
void fileopts_check_options(fileopts_t opt)
{
  printf("Register status:\n"
    "READ:  %d\n"
    "WRITE: %d\n"
    "MODE:  %d\n"
    "CRC:   %d\n"
    "\n",
    IS_SET(READ, opt), IS_SET(WRITE, opt), IS_SET(MTEXT, opt),
    IS_SET(CRC, opt)
    );
}
```

Powyżej widzimy wydrukowanie zawartości „rejestrów”, z racji że rejestry OPT nas mało interesują toje sobie darujemy.

C

```c
void fileopts_clear_options(fileopts_t* opt)
{
  *opt = 0;
}
```

Tutaj po prostu zerujemy wartość (dzięki @krzaq za zwrócenie uwagi odnośnie niemającego na nic wpływu xor'a).

```c
int main()
{
  fileopts_t reg;
  fileopts_clear_options(&reg);
  fileopts_check_options(reg);

  fileopts_check_options(READ | WRITE | MBIN);
  fileopts_check_options(READ | MTEXT | OPT1 | OPT2);
  fileopts_check_options(RW | CRC);

  return 0;
}
```

W ostatnim listingu widzimy proste demo pokazujące sposób działania kodu, jak widzimy poniżej ustawione flagi są zgodne z tym co ustawiliśmy w kodzie.

```
Register status:
READ:  0
WRITE: 0
MODE:  0
CRC:   0

Register status:
READ:  1
WRITE: 1
MODE:  0
CRC:   0

Register status:
READ:  1
WRITE: 0
MODE:  1
CRC:   0

Register status:
READ:  1
WRITE: 1
MODE:  0
CRC:   1
```

Koniec

Mam nadzieję, że w dość jasny sposób przedstawiłem tę dość prostą ideę. W razie pytań, uwag, chęci wyrażenia swojej opinii to oczywiście zapraszam do systemu komentarzy.

Code ON!