czech

Misra-C

Céčko bezpečně

Marcel Baláš

2012

Outline

-

Outline

-

MISRA?

- Motor Industry Software Reliability Association
- Standard pro vývoj SW v jazyce C zaměřený na bezpečnost, spolehlivost a snadnou přenositelnost zejména v oblasti embedded systémů.
- MISRA C 1998 obsahuje 127 pravidel, z toho 93 ”required” a 34 ”advisory”.
- MISRAC 2004 obsahuje 142 pravidel, z toho 122 ”required” a 20 ”advisory”, rozdělených do 21 kategorií (”Prostředí”, ”Runtime chyby”, ...).
- MISRAC 2012 vkládá i podporu pro C99, obsahuje 143 pravidel a 16 ”direktiv”, vše klasifikováno jako ”mandatory”, ”required” nebo ”advisory”.
- Placené!

Outline

-

K čemu to?

Po aplikování pravidel uvedených v MISRAC by měl být kód:

- jednoduchý,
- přehledný,
- čitelný,
- snadno udržovatelný,
- snadno debugovatelný.

Čímž se vposledku naplní ony tři základní cíle standardu bezpečnost, spolehlivost, snadná přenositelnost.

Kategorie pravidel

- *Advisory* pravidla nemusí být splněna, ale mělo by to být alespoň zmíněno v dokumentaci.
- *Mandatory* pravidla musí být všechna splněna.
- *Required* pravidla by měla být splněna, nicméně není nutné bezhlavě aplikovat všechna, aby kód splňoval nálepku ”MISRAC compliant”. Odchylky od pravidel ale musí být zdokumentovány, buď přímo v kódu nebo ve speciálním souboru. Komentář musí obsahovat tyto dva body:
  - Pravidlo, které je porušeno.
  - Důvod pro porušení vysvětlení, proč je pravidlo ignorováno/porušeno, a ujištění, že zvolený způsob nebude mít negativní dopad na základní cíle standardu.

Outline

-

Typový systém

Rule 13 (advisory)

The basic types of `char`, `int`, `short`, `long`, `float`, and `double` should not be used, but specificlength equivalents should be typedef'd for the specific compiler, and these type names used in the code.

Důvod je zřejmý v různých prostředích a s různými kompilátory mohou mít tyto typy různou velikost, což jistě nevede ke snadné přenositelnosti kódu, a koneckonců to příliš neprospívá ani bezpečnosti.
Rozumné tudíž je, používat typy s explicitní velikostí a znaménkem:
`int16_t`, `uint32_t`, `real32_t`, ...

Outline

-

Jeden výstupní bod

Rule 82 (advisory)

A function should have a single point of exit.

- Kdo jednou okusil, už nechce jinak.
- Nutí přemýšlet nad funkcí jako *komplexní jednotkou*, žádný adhoc styl.
- Bezpečnější pro synchronizaci (”omutexované” funkce).
- Výhodnější pro debugování.
- Obvykle nadeklarujeme proměnnou návratového typu na začátku funkce, nainicializujeme na defaultní hodnotu, provedeme výpočet funkce a na konci tuto proměnnou předáme do returnu.
- Často nutí kodéry převracet význam podmínek uvnitř funkce namísto odbavení nevyhovujících vstupních podmínek s následným returnem chybové hodnoty, kontrolujeme správnost vstupních hodnot a vnořujeme se až do hlavního bloku funkce.

Jeden výstupní bod

Rule 82 (advisory)

A function should have a single point of exit.

Špatně:

```
bool check(const Object * me, int32 value)
{
  if ((me == NULL) || (value <= 0))
  {
    return false;
  }
  // ... main code
  return true;
}
```

Jeden výstupní bod

Rule 82 (advisory)

A function should have a single point of exit.

Správně:

```
bool check(const Object * me, int32 value)
{
  bool result = false;
  if ((me != NULL) && (value > 0))
  {
    // ... main code
  }
  return result;
}
```

Jeden výstupní bod

Kritika

- Ne vždy je důsledné trvání na tomto pravidle ku prospěchu.
- Často je však na vině jen programátorova neschopnost / neochota / nedostatek invence.

Spousta programátorů si stěžuje kupříkladu na vnořené podmínky. Ty lze ovšem častokrát vyřešit poměrně elegantně a existuje hned několik způsobů, každý vhodný pro jinou situaci:

Jeden výstupní bod

Vnořené podmínky

i) S využitím pomocné proměnné:

```
int strcmp(
  char * s1, int len1,
  char * s2, int len2)
{
  int result = 0;
  bool isDone = true;

  if (len1 == 0)
  {
    if (len2 == 0)
    {
      result = 0;
    }
    else
    {
      result = 1;
    }
    else
    {
      isDone = false;
    }
  }
  if (!isDone)
  {
    /* the very string
    comparison */
```

Jeden výstupní bod

Vnořené podmínky

ii) S využitím ternárního operátoru potřebujeme zavolat v sekvenci n funkcí, přičemž $(k+1)$ se volá jen tehdy, když $k$tá nevrátila chybu, $k = 0..n1$. Potom můžete psát:

```
static Error checkError()
{
  Error res = NO_ERROR;

  res = (res == NO_ERROR) ? checkSth1() : res;
  res = (res == NO_ERROR) ? checkSth2() : res;
  res = (res == NO_ERROR) ? checkSth3() : res;

  return result;
}
```

Pozn.: Tato varianta *není* pomalejší, při zapnutí optimalizací vyplivne kompilátor kód shodný jako u vnořených ifů.

Jeden výstupní bod

Vnořené podmínky

iii) Analogicky jako sub ii), avšak s booleanovskými funkcemi ještě jednodušší:

```
static bool boolCheck()
{
  return (
    checkSth1() && checkSth2() &&
    checkSth3() && checkSth4()
  );
}
```

Závorkuji, závorkuješ, závorkujeme

Rule 59 (required)

The statement forming the body of an `if`, `else if`, `else`, `while`, `do ... while`, or `for` statement shall always be enclosed in braces.

Špatně:

```
if (a == 0)
{
  b = 1;
  c = 2;
}
else
  b = 2;
  c = 1;
```

Závorkuji, závorkuješ, závorkujeme

Rule 59 (required)

The statement forming the body of an `if`, `else if`, `else`, `while`, `do ... while`, or `for` statement shall always be enclosed in braces.

Správně:

```
if (a == 0)
{
  b = 1;
  c = 2;
}
else
{
  b = 2;
  c = 1;
}
```

Závorkuji, závorkuješ, závorkujeme (2)

Rule 34 (required)

The operands of a logical && or || shall be primary expressions.

Obě strany výrazu s binárním logickým operátorem musí být buď konstantami, jednoduchými proměnnými, nebo výrazem *v závorkách*.

Špatně:

```
d = a && b || c;
e = a || b && c;
f = a == 3 || b > 5;
```

Závorkuji, závorkuješ, závorkujeme (2)

Rule 34 (required)

The operands of a logical && or || shall be primary expressions.

Správně:

```
x = a && b && c; // Allowed exception:when using the same
y = a || b || c; // logical operator.
z = (a == 3) || (b > 5);
```

Boole neznal bity!

Rule 36 (advisory)

Logical operators should not be confused with bitwise operators.

Bitové operatory by neměly být použity v kontextu logického výrazu, stejně tak logické operátory by neměly být použity v kontextu nebooleanovského výrazu.

Špatně:

```
d = (c & a) && b;
d = a && b << c;
if (a & 1) { ... }
```

Boole neznal bity!

Rule 36 (advisory)

Logical operators should not be confused with bitwise operators.

Správně:

```
d = a && b ? a : c;
d = ~a & b;
if ((a & 1) == 0) { ... }
```

Jeden break a dost!

Rule 58 (required)

The break statement shall not be used (except to terminate the cases of a switch statement).

- V originální verzi zakázáno jakékoliv použití příkazu `break` mimo `switch`. Každý cyklus tedy mohl skončit pouze na základě kontrolované podmínky za příkazem pro cyklus.
- MISRAC 2004 povoluje použít (nejvýše však jeden jeho výskyt) i v cyklu.

Jeden break a dost!

Rule 58 (required)

The break statement shall not be used (except to terminate the cases of a switch statement).

Špatně (orig. Misra):

```
int32 i;
int32 n = sizeof(a)/sizeof(a[0]);
for (i = 0; i < n; i++)
{
  if (a[i] == searchedItem)
  {
    result = i;
    break;
  }
}
```

Jeden break a dost!

Rule 58 (required)

The break statement shall not be used (except to terminate the cases of a switch. statement).

Správně (orig. Misra):

```
int32 i;
int32 n = sizeof(a)/sizeof(a[0]);
bool done = false;
for (i = 0; (i < n) && !done; i++)
{
  if (a[i] == searchedItem)
  {
    result = i;
    done = true;
  }
}
```

No more side effects

Rule 33 (required)

The righthand operand of an && or || operator shall not contain side effects.

Špatně:

```
if ((a == b) || (*p++ == c))
{
  /* do something */
}
```

No more side effects

Rule 33 (required)

The righthand operand of an && or || operator shall not contain side effects.

Správně:

```
bool doSomething = false;
if (a == b) {
  doSomething = true;
} else if (*p++ == c) {
  doSomething = true;
} else { }

if (doSomething) {
  /* do something */
}
```

Kontrola kompatibility s MISRAC

- Kompilátory
  - Green Hills Software
  - IAR Systems
  - Tasking
- Tooly
  - PCLint by Gimpel Software
  - SonarQube by SonarSource (open source)
  - Coverity Static Analysis
  - ECLAIR by BUGSENG
  - ...