Ladění

Tato kapitola bude věnována chybám programu. Ukážeme jaké chyby se v programech mohou vyskytovat, a také, jakým způsobem chyby odhalovat.

Chyby

Než si řekneme, jakým způsobem chyby dělíme, zopakujeme pojem sémantiky a syntaxe jazyka. *Syntaxe jazyka* popisuje pravidla pro zápis programu. *Sémantika* popisuje chování programu, tedy to, co program dělá. Obecně lze chyby rozdělit na *chyby syntaktické* a *sémantické chyby*.

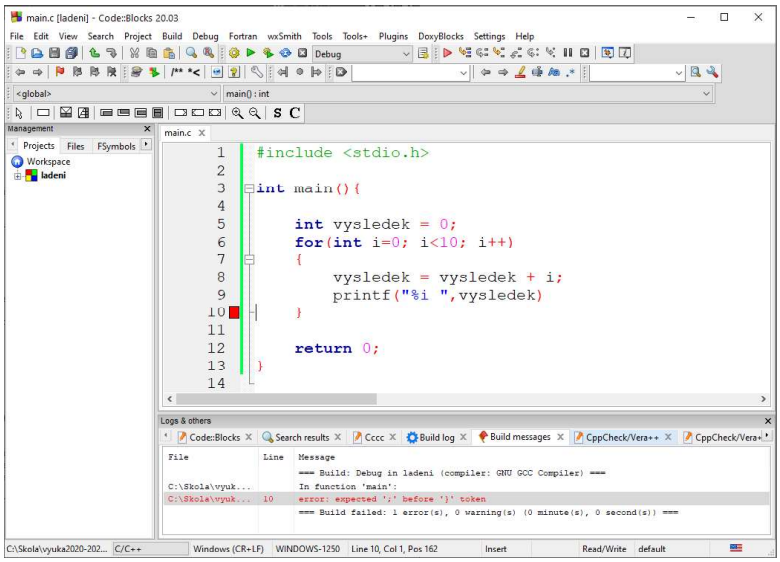
Syntaktické chyby

Syntaktické chyby jsou způsobeny nesprávným zápisem programu. Například častou syntaktickou chybou je chybějící středník na konci řádku. Tyto chyby jsou nalezeny překladačem už při překladu a při jejich výskytu překladač není schopný sestavit spustitelný soubor a vypíše seznam chyb. Většinou je popsána chyba a řádek, na kterém tato chyba vznikla. Syntaktické chyby je snadné nalézt a opravit.
Kromě již zmíněného chybějícího středníku jsou to často překlepy v názvech identifikátorů či chybějící nebo přebývající závorka.
Na obrázku 9 vidíme, že se ve vývojovém prostředí při překladu vyskytla chyba. Ta je vypsaná v konzoli ve spodní části obrazovky. Vidíme, že na řádku 10 je očekáván znak `;` před znakem `}`.[130]

[130] Na řádku 9 nebyl příkaz `printf()` ukončen `;`.

Sémantické chyby

Záludnější chyby jsou chyby sémantické. Ty je možné odhalit až za běhu programu (někdy se přímo označují jako chyby za běhu programu, nebo run-time chyby).
Sémantické chyby ještě můžeme dále dělit na ty, které jsou způsobeny *nesprávným použitím jazyka* a na ty, co jsou způsobené *špatným zdrojovým kódem*.



Obrázek 9: Syntaktická chyba.

První skupina chyb vzniká například při dělení nulou či špatné indexaci polí (přístup k prvkům mimo rozsah pole) nebo při práci se špatnými typy (například výsledek dělení dvou celých čísel je celé číslo a my očekáváme desetinné). Také často vznikají při špatné práci s pamětí, jako je neuvolňování paměti (program používá více paměti, než potřebuje), přístup ke špatné části paměti a tak dále.
Pokud chyba není způsobena tím, jak je program zapsán, ale přesto program nedělá to, co bychom chtěli, tak se bavíme o druhé skupině sémantických chyb. Na vině je algoritmus, který je špatně zapsán (špatně převeden do zdrojového kódu) nebo je chyba v samotném algoritmu.
Hlavní je v tomto případě identifikovat místo, kde chyba vzniká. Je nutné zjistit, co se za běhu programu děje. K tomu potřebujeme najít poslední příkaz, který se vykonal (kde se v programu nacházíme), a také hodnoty jednotlivých proměnných.
Jedním z přístupů, jak sémantickou chybu nalézt je pomocí *analýzy kódu*. Jinak řečeno, opakovaně čteme kód a přemýšlíme nad tím, jak tento kód pracuje (co se stane při vykonání tohoto příkazu, jak se změní hodnoty proměnných a podobně).
Dalším způsobem je pozorování programu za běhu. Nejjednodušší způsob je přidat do kódu (na vhodná místa) výpis hodnot proměnných, které nás zajímají a zkontrolovat, zda mají tyto proměnné hodnoty, které očekáváme.

```c
#include <stdio.h>

int main(){
    int a, b, r;
    printf("Zadejte cisla a a b: ");
    scanf("%i %i",&a, &b);
    printf("Vypocet gcd pro %i a %i\n", a, b);
    while(b != 0){
        r = a % b;
        a = b;
        b = r;
        printf("a: %i, b: %i\n", a, b);
    }
    printf("Nejvetsi spolecny delitel je %i", a);
    return 0;
}
```

Například v předchozím kódu vypisujeme hodnoty proměnných `a` a `b` v každé iteraci cyklu `while`.
Případně můžeme využít nástrojů IDE. Konkrétně se jedná o tzv. *debugger*.

Ladění pomocí debuggeru

Zejména díky konstrukcím, které mění tok programu (jako jsou cykly a větvení), není vždy kód zcela přehledný. Procesu hledání chyb v programu se říká *ladění*.
Při ladění sice program běží od začátku do konce tak, jak jsme zvyklí, ale instrukce jsou vykonávány postupně (je krokován) a v každém kroku je možné sledovat hodnoty všech proměnných.
Debugger dokáže zastavit běh programu na programátorem zvoleném místě (tomuto místu se říká *breakpoint*). V této době (když je program zastaven) je možné si prohlédnout hodnoty jednotlivých proměnných (a také seznam všech neukončených volání funkcí). Dále je možné postupovat po jednotlivých krocích (v každém kroku se provede další příkaz) nebo pokračovat k následujícímu breakpointu.

Další nástroje pro ladění

V některých případech není použití debuggerů dostačující. Například při ladění dynamických procesů se chyba nemusí projevit při každém spuštění programu. Proto je vhodné kromě debuggerů přidávat do kódu i další ladící prostředky, jako jsou například ladící výstupy na `stderr`, využití podmíněného překladu, *self-test* moduly nebo funkce,[131] využití knihovny `<assert.h>` a jiné.

[131] Tento pojem vysvětlíme dále.

Ladící výpisy na `stderr`

Již dříve jsme si představili standardní výstup `stderr`, který se používá pro výpis chyb. Do tohoto proudu můžeme zapisovat například pomocí příkazu `fprintf(stderr, text, ...)`. Ve standardním nastavení se vypíše `text` do konzole. Tento proud můžeme zavřít pomocí `fclose(strderr)` a tím potlačit veškeré jeho výpisy.[132] Toto není úplně čisté řešení. Lepší je výpis přesměrovat do ladícího souboru pomocí funkce `freopen()`. Viz následující kód.

[132] Výpisy ponecháme jen v době, kdy ladíme program.

```
fw = freopen("ladeni.log", "w", stderr);
```

Oba tyto přístupy nám umožňují ovlivnit, co vše se bude vypisovat na obrazovku. Výpisy však v kódu zůstávají a zpomalují program.

Využití podmíněného překladu

Podmíněný překlad nám dává možnost zahrnout do kódu části, které můžeme díky preprocesoru vynechat, když je nepotřebujeme. Přebytečné části kódu zůstávají ve zdrojovém souboru, ale nejsou ve výstupu programu.
Nejjednodušším způsobem je použití podmíněného překladu využívajícího existence symbolické konstanty následujícím způsobem.

```
#ifdef DEBUG /* zacatek podmineneho prekladu */

/* ladici cast */

#endif /* konec podmineneho prekladu */
```

Pokud není symbolická konstanta `DEBUG` definovaná,[133] pak je ladící část kódu preprocesorem odstraněna.
Kromě tohoto přístupu, můžeme podmíněný překlad použít i tak, že nás bude zajímat hodnota konstanty `DEBUG` a podle ní můžeme ovlivnit, jak bude výstup vypadat. Viz následující příklad.

[133] Pro připomenutí symbolickou konstantu definujeme následujícím způsobem.
`#define DEBUG`
Na hodnotě této konstanty v tomto případě nezáleží. Jen na tom, zda je, nebo není definovaná.

```
#if defined(DEBUG) && DEBUG == 1 /* zacatek podmineneho
    prekladu */

/* ladici cast 1 */

#elif defined(DEBUG) && DEBUG == 2

/* ladici cast 2 */

...
#endif /* konec podmineneho prekladu */
```

Nevýhodou těchto přístupů je to, že je potřeba změnit kód,[134] pokud chceme ladící část vynechat nebo do kódu zahrnout a zdrojový kód znovu přeložit.

[134] Definovat symbolickou konstantu, nebo její definici odstranit.

Self-testy modulů nebo funkcí

Využití tzv. self-testů (samotestovací mechanismus) využíváme zejména při odděleném překladu. Tyto self-testy obsahují jednotlivé moduly a je tedy možné je otestovat nezávisle na ostatních modulech. Jejich zapínání a vypínání je realizované pomocí podmíněného překladu.
V každém modulu je funkce `main()` s parametry, aby se dala spouštět z příkazové řádky s testovacími hodnotami. V následujícím příkladu máme pomocný modul, ve kterém definujeme funkci s funkčním prototypem `int funkceA(int a)`. Funkce `main()` je do kódu vložena jen v případě, že je definovaná symbolická konstanta `SELF`.

```c
/* funkce.c
 * Modul obsahujici definici funkce funkceA
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#ifdef SELF

int funkceA(int a);

int main(int argc, char *argv[]){
    if(argc == 1){
        fprintf(stderr, "Nepredany zadny argument.");
        return 1;
    }
    printf("Vysledek = %d", funkceA(atoi(argv[1])));
    return 0;
}

#endif /* konec SELF */

int funkceA(int a){

    /* Definice funkce*/

    return vysledek;
}
```

Využití knihovny `assert.h`

Jak již víme, v programu se mohou objevit chyby, které se projeví už při překladu, a také chyby, které se objeví až za běhu programu. Takové chyby se ale nemusí vyskytnout při každém spuštění programu, ale objeví se jen někdy. Jejich hledání a odstranění je velmi časově náročné. Dají se hledat za použití podmíněného překladu testovacích částí. Knihovna `assert.h` obsahuje nástroje velmi podobné podmíněnému překladu. Tyto nástroje nás upozorní na možnou chybu před jejím výskytem.[135] Zejména se využívá při práci s poli, řetězci a ukazateli, kde vznikají chyby typu špatného přístupu do paměti.[136]

Například příkaz `pole[-2] = 10;` je zřejmě chybný, ale překladači nevadí. Program obsahující tento příkaz se bude na různých systémech chovat různě. Někdy spadne, někdy „jen“ přepíše jinou hodnotu.

Princip je ten, že během výpočtu se vyhodnocuje nějaký logický výraz, o kterém se předpokládá, že je pravdivý. Pokud tomu tak je, tak se nic neděje a program běží dál. Pokud se výraz vyhodnotí na nepravdu, provede se chybový výpis a ukončí se běh programu. Použití je ukázáno v následujícím příkladu.

```
#include <stdio.h>
#include <assert.h>

#define VELIKOST 10

int main(){
    int pole[VELIKOST];
    int cislo;
    int index;

    /* Nejaky kod   */

    assert(index >=0 && index <= VELIKOST);
    pole[index] = cislo;

    return 0;

}
```

V příkladu je zřejmé, že v programu může dojít k tomu, že proměnná `index` nabude hodnot mimo meze pole `pole`. Překladač by na tuto chybu neupozornil, ale `assert()` ano.

Není přesně definováno, kdy by se měla konstrukce `assert()` použít, ale obecně platí, že by se měla vložit do míst, kde víme, že některé hodnoty jsou chybné, ale velmi nepravděpodobné a proto není úplně vhodné v těchto částech kódu kvůli jeho přehlednosti

[135] Před tím, než program zhavaruje.

[136] U polí jsou to například chyby zápisu do paměti za polem (index o jedna větší). Můžeme se setkat s pojmem *off-by-one-error*.

používat testování hodnoty pomocí `if`.
Testy `assert()` jsou založené na podmíněném překladu a je možné je pomocí preprocesoru z kódu odstranit a tím program zrychlit. K jejich odstranění stačí definovat symbolickou konstantu `NDEBUG`.[137] Po její definici nebudou do kódu zahrnuty žádné příkazy `assert()`. Zrušením definice konstanty `NDEBUG` je opět do kódu zahrneme.

[137] `#define NDEBUG`

Cvičení

Úkol 91
Určete, zda se v následujících kódech vyskytuje chyba a pokud ano, jakého je typu, případně, jak bychom jí opravili.

1.

```c
#include <stdio.h>

int main(){
    int x = 10;
    int y = 20;

    printf("(%i, %i)", x, y)
    return 0;
}
```

2. Následující kód vypíše 10. Proč?

```c
#include <stdio.h>

int main(){
    int i;

    for(i = 0; i < 10; i++);
    {
        printf("%i ", i);
    }
    return 0;
}
```

3.

```c
#include <stdio.h>

int main(){
    int i;

    for(i=0, i<10, i++){
        printf("%i ", i);
    }
    return 0;
}
```

4.

```c
#include <stdio.h>

int main()
{
    int pole[3]={1, 2, 3};

    printf("%i ", pole[3]);
    return 0;

}
```

5.

```c
#include <stdio.h>

int main(){
    int b;

    while(b > 0){
        printf("*");
        b--;
    }
    return 0;
}
```

6.

```c
#include <stdio.h>

int main(){
    int n = 10;
    int m = 0;

    m = n / 0;

    printf("%i ", m);
    return 0;
}
```

7.

```c
#include <stdio.h>

int main()
{
    int a=0;

    if(a=0){
        printf("a je 0");
    }
    else{
        printf("a neni 0");
    }
    return 0;
}
```

8.

```c
#include <stdio.h>

int main(){
    int a = 10, b = 20, c;
    a + b = c;
    return 0;
}
```

9.

```c
#include <stdio.h>
int i;

void vykresli_trouhelnik(int n){
    for(i = 1; i <= n; i++){
        for(int j = 0; j < i; j++){
            printf("*");
        }
        printf("\n");
    }
}

int main(){
    int n = 5;
    for(i = 1; i <= n; i++){
       vykresli_trouhelnik(i);
    }
    return 0;
}
```

10.

```c
#include <stdio.h>

typedef struct{
    int x;
    int y;
    int pole[2];
} struktura;

int main(){
    struktura m;
    m = {10, 2, {10, 2}};
    return 0;
}
```

Úkol 92
Následující kód skončí chybou při překladu. Proč tomu tak je?

```c
#include <stdio.h>

int main(){
    char *karty = "JQK";
    char karta = karty[2];

    karty[2] = karty[1];
    karty[1] = karty[0];
    karty[0] = karty[2];
    karty[2] = karty[1];
    karty[1] = karta;

    printf("%s", karty);
    return 0;
}
```

Úkol 93
Určete, které z následujících kódů se podaří zkompilovat. V případě neúspěchu jaké se v kódu objevují chyby? V případě úspěchu určete, jaký bude výstup a zkuste odhadnout, zda se program chová tak, jak by měl.

1.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1)
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    else
        printf("Eso");

    return 0;
}
```

2.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1){
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    else
        printf("Eso");
    }

    return 0;
}
```

3.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1){
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    }else
        printf("Eso");

    return 0;
}
```

4.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1){
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    else
        printf("Eso");

    return 0;
}
```

Úkol 94
Vyzkoušejte si práci s `assert()`.