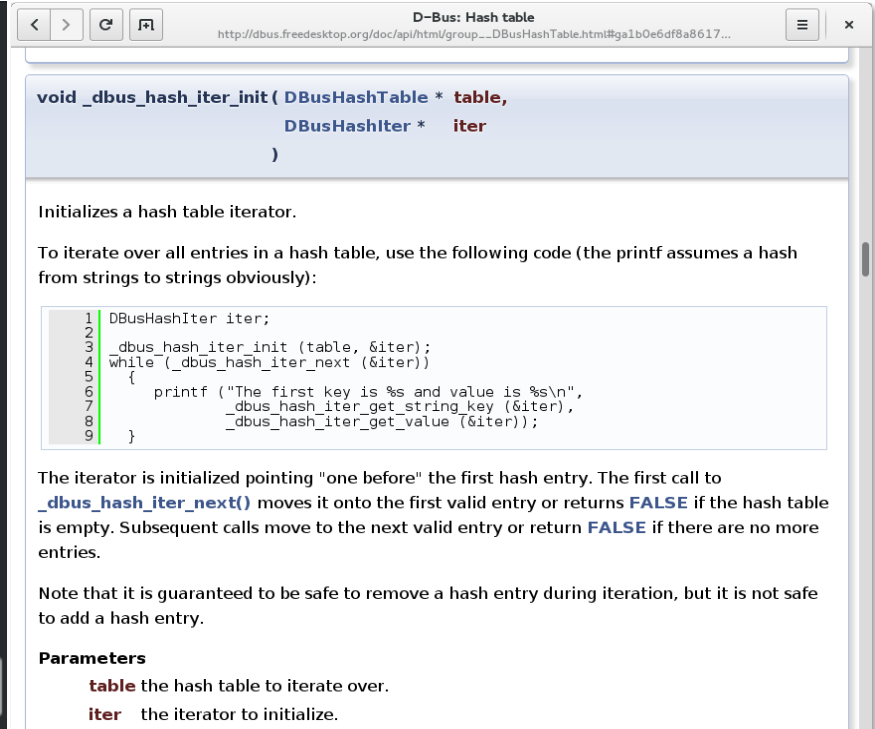


Základy programování

```
0.000078 strcpy(0x564344c364e0, "SSH_ASKPASS=/usr/libexec/openssh"
...) = 0x564344c364e0
0.000117 strlen("HOME=/root") = 10
0.000112 malloc(11) = 0x564344c34f70
0.000078 strcpy(0x564344c34f70, "HOME=/root") = 0x564344c34f70
0.000095 strlen("SHLV=2") = 7
0.000108 malloc(8) = 0x564344c34f90
0.000078 strcpy(0x564344c34f90, "SHLV=2") = 0x564344c34f90
0.000092 strlen("LOGNAME=root") = 12
0.000117 malloc(13) = 0x564344c34990
0.000079 strcpy(0x564344c34990, "LOGNAME=root") = 0x564344c34990
0.000095 strlen("CVS_RSH=ssh") = 11
0.000115 malloc(12) = 0x564344c349b0
0.000079 strcpy(0x564344c349b0, "CVS_RSH=ssh") = 0x564344c349b0
0.000093 strlen("QTLIB=/usr/lib64/qt-3.3/lib") = 27
0.000157 malloc(28) = 0x564344c349d0
0.000082 strcpy(0x564344c349d0, "QTLIB=/usr/lib64/qt-3.3/lib") = 0
x564344c349d0
0.000117 strlen("MODULESHOME=/usr/share/Modules") = 30
0.000166 malloc(31) = 0x564344c34a00
0.000078 strcpy(0x564344c34a00, "MODULESHOME=/usr/share/Modules")
= 0x564344c34a00
0.000109 strlen("LESSOPEN=|/usr/bin/lesspipe.sh %"...) = 33
0.000173 malloc(34) = 0x564344c35140
```



The screenshot shows a web browser window with the title "D-Bus: Hash table" and the URL "http://dbus.freedesktop.org/doc/api/html/group___DBusHashTable.html#ga1b0e6df8a8617...". The page content includes the function signature for `_dbus_hash_iter_init`, a description of its purpose, a code example for iterating over a hash table, and a list of parameters.

```
void _dbus_hash_iter_init (DBusHashTable * table,
                          DBusHashIter * iter
                          )
```

Initializes a hash table iterator.

To iterate over all entries in a hash table, use the following code (the `printf` assumes a hash from strings to strings obviously):

```
1 DBusHashIter iter;
2
3 _dbus_hash_iter_init (table, &iter);
4 while (!_dbus_hash_iter_next (&iter))
5 {
6     printf ("The first key is %s and value is %s\n",
7            _dbus_hash_iter_get_string_key (&iter),
8            _dbus_hash_iter_get_value (&iter));
9 }
```

The iterator is initialized pointing "one before" the first hash entry. The first call to `_dbus_hash_iter_next()` moves it onto the first valid entry or returns `FALSE` if the hash table is empty. Subsequent calls move to the next valid entry or return `FALSE` if there are no more entries.

Note that it is guaranteed to be safe to remove a hash entry during iteration, but it is not safe to add a hash entry.

Parameters

- `table` the hash table to iterate over.
- `iter` the iterator to initialize.

Ladění a dokumentace programů

12. přednáška 2021

Aleš Smrčka, Jitka Kreslíková



Osnova přednášky

1. Ladění programů
2. Zpracování chyb
3. Ověřování správnosti programů
4. Dokumentace zdrojových kódů

Ladění programů : Rekapitulace

Proces ladění odpovídá konceptu:

1. Reprodukce (bug report)
2. Diagnóza
3. Oprava (bug fix)
4. Intergrace (reflect, commit)

Ladění programů : Rekapitulace

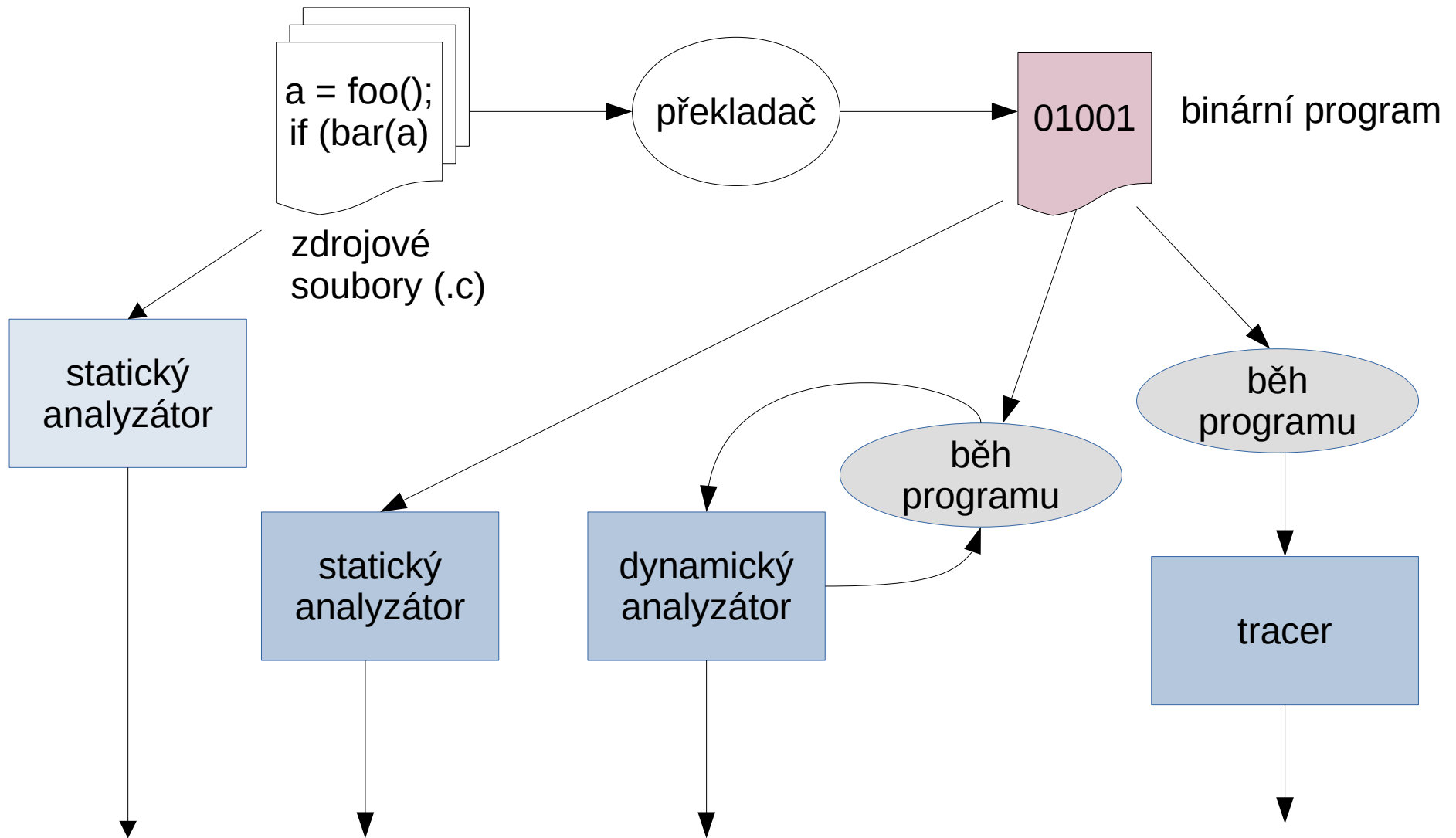
Základní techniky ladění programů:

- ladění jako statická analýza (code review),
- ladění jako dynamická analýza:
 - ladící výpisy (log),
makra, instrumentace
 - interaktivní ladící nástroje (debugger),
krokování programu, sledování proměnných
- další ladící nástroje...

Další ladící nástroje

- Statické analyzátory zdrojových souborů
 - CBMC, Frama-C, Infer, [produkty výzkumné skupiny VeriFIT](#)
 - Nad rámec IZP
- Statické analyzátory binárních souborů
 - slouží pro [statickou kontrolu binárních souborů](#): přeložené instrukce funkcí, práce s proměnnými, datovými strukturami,
 - inspektory = jednoduché statické analyzátory.
 - Příklad: nm, ldd, objdump, (antiviry, ...)
- Dynamické analyzátory/monitory běhu programů
 - [sledují určitý aspekt/vlastnost](#) (např. práce s pamětí, I/O přístupy, synchronizace procesů, otevřené soubory) a podávají hlášení vývojáři,
 - mohou ovlivnit chod programu, resp. jeho prostředí.
 - Příklad: valgrind, lsof, ([Testos@FIT](#), [AnaConDA@FIT](#), ...)
- Programy pro stopování programů
 - angl. *tracers*, slouží pro [výpis činností](#) (např. volaná knihovní volání, stav zásobníku včetně stack-frame).
 - Příklad: strace, ltrace, pstack

Další ladící nástroje



Inspektory programů - nm

nm - výpis tabulky symbolů v binárním souboru

- Binární soubor se skládá z několika sekcí (typicky 8-30), např. sekce pro kód (.text) a sekce pro data (.data, .rodata, .bss).
- Každá sekce bude po spuštění programu nahrána do paměti. Má předem dané místo = adresu (čti: hodnotu ukazatele).
- Každý symbol, např. globální proměnná, funkce nebo statická konstanta, má svoji adresu v dané sekci. Umístění symbolů je uvedeno v tzv. [tabulce symbolů](#).
- Tabulka symbolů je v různých binárních souborech:
 - binární program,
 - dynamická knihovna,
 - objektový soubor (.o)

Inspektory programů - nm příklad

```
$ nm scluster
```

000000000000400ba4	T	append_cluster	↖ deklarované objekty
00000000000060209c	B	__bss_start	
000000000000400ae7	T	clear_cluster	
0000000000004014d0	R	CLUSTER_CHUNK	↖ přidáno překladačem
000000000000400df6	T	cluster_distance	
0000000000006020a8	b	completed.6934	↖
000000000000400a20	t	__do_global_dtors_aux	
	U	fclose@@GLIBC_2.2.5	
000000000000400ed6	T	find_neighbours	↖ externí funkce
0000000000004014b4	T	_fini	
0000000000004010ec	T	load_clusters	
0000000000004012f3	T	main	
	U	malloc@@GLIBC_2.2.5	
000000000000400c2f	T	merge_clusters	
000000000000400d8e	T	obj_distance	

Inspektory programů - ldd

ldd - výpis závislostí binárních programů na dynamických knihovnách

- Programy pro svoji činnost využívají knihovní volání. Nejtypičtější knihovnou je *standardní knihovna jazyka C (stdc, libstdc)*.
- Knihovní funkce mohou být uloženy v tzv. dynamické knihovně*.
- Program tedy závisí na knihovnách, tedy externích binárních souborech. Při spuštění programu je potřeba knihovny načíst do paměti.

* více o knihovnách na přednášce o modulárním programování

Inspektory programů - ldd příklad

```
$ ldd scluster
linux-vdso.so.1 (0x00007ffc3ecf7000)
libm.so.6 => /lib64/libm.so.6 (0x00007f52085b9000)
libc.so.6 => /lib64/libc.so.6 (0x00007f52081f8000)
/lib64/ld-linux-x86-64.so.2 (0x0000559fd00b0000)

$ ldd /usr/lib64/firefox/firefox
linux-vdso.so.1 (0x00007ffd789e7000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f30666a4000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f30664a0000)
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f306611d000)
libm.so.6 => /lib64/libm.so.6 (0x00007f3065e1b000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f3065c04000)
libc.so.6 => /lib64/libc.so.6 (0x00007f3065842000)
/lib64/ld-linux-x86-64.so.2 (0x000056496127c000)

$ ldd /bin/true
linux-gate.so.1 (0xb76f0000)
libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6
(0xb753b000)
/lib/ld-linux.so.2 (0xb76f3000)
```

32 bitová aplikace

Inspektory programů - objdump

objdump - detailní a ucelený výpis (angl. dump) binárních dat:

- sekce programu,
- tabulka symbolů,
- obsah sekcí:
 - obsah datové části včetně výpisu v hexadecimálním formátu (tzv. hexdump),
 - obsah kódových částí v podobě instrukcí procesoru (více viz předmět ISU - Programování na strojové úrovni).

Inspektory programů - objdump příklad

```
$ objdump -D scluster
scluster:      file format elf64-x86-64
```

Disassembly of section .text:

```
...
0000000000400ff3 <sort_cluster>:
400ff3: 55                push    %rbp
400ff4: 48 89 e5          mov     %rsp,%rbp
400ff7: 48 83 ec 10       sub     $0x10,%rsp
400ffb: 48 89 7d f8       mov     %rdi,-0x8(%rbp)
400fff: 48 8b 45 f8       mov     -0x8(%rbp),%rax
401003: 8b 00            mov     (%rax),%eax
401005: 48 63 f0         movslq  %eax,%rsi
401008: 48 8b 45 f8       mov     -0x8(%rbp),%rax
40100c: 48 8b 40 08       mov     0x8(%rax),%rax
401010: b9 a2 0f 40 00   mov     $0x400fa2,%ecx
401015: ba 0c 00 00 00   mov     $0xc,%edx
40101a: 48 89 c7         mov     %rax,%rdi
40101d: e8 7e f8 ff ff   callq   4008a0 <qsort@plt>
401022: 90              nop
401023: c9              leaveq  %rsp,%rbp
401024: c3              retq
```

```
0000000000401025 <print_cluster>:
401025: 55                push    %rbp
401026: 48 89 e5          mov     %rsp,%rbp
401029: 48 83 ec 20       sub     $0x20,%rsp
40102d: 48 89 7d e8       mov     %rdi,-0x18(%rbp)
401031: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
401038: e9 93 00 00 00   jmpq    4010d0 <print_cluster+0x100>
```

```
void sort_cluster(struct cluster_t *c)
{
    qsort(c->obj,
          c->size,
          sizeof(struct obj_t),
          &obj_sort_compar);
}
```

Monitory aktuálního stavu procesů

- Monitory aktuálního stavu procesů reportují o alokovaných prostředcích:
 - alokovaná paměť,
 - otevřené soubory,
 - aktuální využití sítě nebo I/O operací.
- Data získávají přímo od jádra operačního systému (více viz předmět Operační systémy IOS).
- Příklady monitorů:
 - lsof - výpis aktuálně otevřených souborů.
 - /proc - dynamicky měnící se adresářová struktura s daty o běžících procesech.
 - Microsoft Process Explorer - nejen-GUI aplikace s ucelenými daty o běžících procesech.

Dynamické analyzátory - valgrind

- **Valgrind** = framework (sada nástrojů) pro dynamickou analýzu programů.
- Analýza na základě **instrumentace** = před spuštěním kódu je kód mírně obohacen o řídicí prvky (tzv. sondy).
 - Program je instrumentován.
 - Program se spustí normálním způsobem.
 - Spouštěné sondy informují monitor o aktuálních událostech (např. přístupů do paměti).
 - Po skončení programu valgrind informuje souhrnem událostí.
- Nejznámější (původní) částí je **memcheck** = kontrola paměťových přístupů.

Valgrind – příklad

```
$ valgrind ./scluster_valid
```

výstup analyzovaného programu

```
...  
==17885== Command: ./scluster  
usage: proj3 objects.txt [number-of-clusters]  
==17885==  
==17885== HEAP SUMMARY:  
==17885==      in use at exit: 0 bytes in 0 blocks  
==17885==    total heap usage: 0 allocs, 0 frees, 0 bytes allocated  
==17885==  
==17885== All heap blocks were freed -- no leaks are possible  
...
```

```
$ valgrind ./scluster_invalid objekty
```

hlášení valgrind

```
...  
==18072==  
==18072== HEAP SUMMARY:  
==18072==      in use at exit: 240 bytes in 1 blocks  
==18072==    total heap usage: 41 allocs, 40 frees, 2,528 bytes allocated  
==18072== LEAK SUMMARY:  
==18072==      definitely lost: 240 bytes in 1 blocks  
==18072==      indirectly lost: 0 bytes in 0 blocks  
==18072==      possibly lost: 0 bytes in 0 blocks  
==18072==      still reachable: 0 bytes in 0 blocks  
==18072==      suppressed: 0 bytes in 0 blocks  
==18072== Rerun with --leak-check=full to see details of leaked memory
```

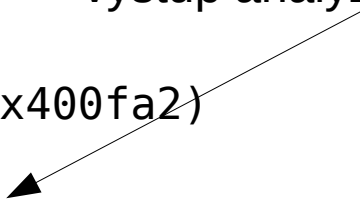
Stopování procesů - ltrace

- Výpis činností běžícího procesu/programu.
- ltrace = trasování knihovních volání (např. volání std. knihovny jazyka C).
- Trasované události musí být známy předem
 - např. prototypy volaných funkcí.

ltrace – příklad

```
$ ltrace ./scluster obj 2
__libc_start_main([ "./proj3", "obj", "2" ] <unfinished ...>
fopen("obj", "r") = 0x1a90010
__isoc99_fscanf(0x1a90010, 0x4014e0, 0x7ffd3620c454, 0x7f212555a940) = 1
malloc(48) = 0x1a90240
__isoc99_fscanf(0x1a90010, 0x4014e9, 0x7ffd3620c440, 0x7ffd3620c444) = 3
malloc(12) = 0x1a90280
__isoc99_fscanf(0x1a90010, 0x4014e9, 0x7ffd3620c440, 0x7ffd3620c444) = 3
malloc(12) = 0x1a902a0
__isoc99_fscanf(0x1a90010, 0x4014e9, 0x7ffd3620c440, 0x7ffd3620c444) = 3
malloc(12) = 0x1a902c0
fclose(0x1a90010) = 0
strtol(0x7ffd3620e0c7, 0x7ffd3620c4a8, 10, 0x7ffd3620c4a8) = 2
sqrtf(512546.000000) = 715.923157
sqrtf(135701.000000) = 368.376160
sqrtf(214669.000000) = 463.323853
realloc(0x1a90280, 24) = 0x1a90280
qsort(0x1a90280, 2, 12, 0x400fa2) = <void>
free(0x1a902c0) = <void>
puts("Clusters:Clusters:
) = 10
printf("cluster %d: ", 0) = 11
printf("%d[%g,%g]", 40, 86.000000, 663.000000) = 10
```

výstup analyzovaného programu



Osnova přednášky

1. Ladění programů, pokračování.
2. Zpracování chyb
3. Ověřování správnosti programů.
4. Dokumentace zdrojových kódů.

Zpracování chyb

- Programy/systémy jsou pod tlakem, aby nepracovaly v normálním režimu:
 - bezpečnostní útoky,
 - chybné nebo škodlivé vstupy,
 - hardwarové nebo softwarové chyby,
 - neočekávané chování uživatele,
 - neočekávané změny prostředí.
- Takové systémy musí zajistit nezbytnou funkcionalitu v rámci:
 - spolehlivosti (safety),
 - bezpečnosti (security) a
 - výkonnosti (performance).

Terminologie

- **Fault tolerant** (system) = schopnost pokračovat v normálním režimu i přes přítomnost chyby.¹
- **Fail safe** = v přítomnosti chyby se automaticky přepíná do bezpečného režimu.¹
- **Fail soft** = v přítomnosti některých chyb pokračuje v částečně operačním režimu.¹
- **Fail secure** = systém, který neprodukuje žádná selhání.¹
- **Fail hard** = v přítomnosti chyby ukončuje svoji činnost.

¹ IEEE Std ISO-24765-2017 Systems and software engineering - Vocabulary.

Detekce chyb a zotavení

- Zpracování chybových stavů rozdělujeme na dvě části:
 1. detekce chyby (**detection**) = odhalení vady nebo selhání, neočekávaného stavu.
 2. zotavení po chybě (**recovery**) = proces obsluhy chybového stavu.
- Zpracování chyb se liší od obvyklého toku řízení programu.
- Změna toku řízení způsobuje problém s (de)alokací prostředků:
 - dynamické paměti (např. malloc),
 - otevřených souborů (např. fopen),
 - zámků pro výlučný přístup (např. pthread_mutex_lock), apod.
- Vyčerpání těchto prostředků může vést ke špatné:
 - výkonnosti (např. timeout, lossy channel, low throughput),
 - bezpečnosti (např. DoS = Denial-Of-Service),
 - spolehlivosti (např. NULL-pointer dereference, data-inconsistency, deadlock).

Způsoby zpracování chyb

- Způsoby zpracování chyb:
 - **Prevence** = návrh programu tak, aby k chybám nedošlo.
 - **Propagace** = program/část programu počítá s možností chyby, zotavení nechává na volajícím.
 - **Terminace** = bez zotavení, při chybě ihned ukončuje činnost.
- Příklad:
 - `prevence` = Program, který nealokuje dynamickou paměť, nemůže mít problém s alokací.
 - `propagace` = Pokud selže alokace, dáme o tom vědět volajícímu, necht' ten se postará o zpracování chyby,
 - `terminace` = Pokud selže alokace paměti, ukončíme celý program (`main-return`, `exit`, `abort`).
- Priority: `prevence` > `propagace` > `terminace`

Propagace chybového stavu

- **Globální chybový indikátor** – globální/statická proměnná pro identifikaci chybového stavu:

- `#include <errno.h> → int errno;`
- `errno` je při startu programu nastaven na 0.

EPERM 1 Operation not permitted
ENOENT 2 No such file or directory
...
EIO 5 Input/output error

- Knihovní operace nastavují `errno` pouze při chybě.

- **Lokální chybové indikátory** – stejný princip jako globální, ale vázané na konkrétní objekt/strukturu/činnost:

- např. `int ferror(FILE*); int feof(FILE*);`
- `unsigned long strtoul(char *nptr, char **endptr, int base);`

- **Návratová hodnota z funkce:**

- celá návratová hodnota vymezena pro indikátor chyby,
- část oboru hodnot vyhrazena pro chybový stav.

Propagace návratovou hodnotou

- unsigned → 0, void* → NULL
- int → -1, 1, != 0
- int → normal: >0, abnormal: -(error_id), viz getchar + EOF

```
void do_something1(void) {  
    object_t *obj;  
    obj = malloc(sizeof(object_t));  
    obj->attr = 42;  
    ...  
}
```

```
int do_something2(void) {  
    object_t *obj;  
    obj = malloc(sizeof(object_t));  
    if (obj == NULL)  
        return 1;  
    obj->attr = 42;  
    ...  
    return 0;  
}
```


Propagace návratovou hodnotou



Zotavení po chybě

- Specifické pro doménu programu. Typicky opakované předávání stavu volanému podprogramu, potom ošetření výjimečného stavu.
- Výjimky (exceptions) jsou nad rámec této přednášky.

C++/Java/...

```
void foo() {  
    throw ExceptionName1("some err");  
}  
...  
try {  
    foo();  
}  
catch (ExceptionName1 e) { ... }  
catch (ExceptionName2 e) { ... }
```

Python:

```
def foo():  
    raise SomeError()  
...  
try:  
    foo  
except SomeError:  
    ...
```

Zotavení po chybě – goto chain

Příklad: Podprogram otevře dva soubory pro čtení a zápis, alokuje objekt v paměti a provede nějaké operace:

```
void do_something() {  
    FILE *fin, *fout;  
    obj_t *obj;  
    fin = fopen("fin.txt", "r");  
    fout= fopen("fou.txt", "w");  
    obj = malloc(sizeof(obj_t));  
  
    // do some work...  
    free(obj);  
    fclose(fout);  
    fclose(fin);  
}
```

Zotavení po chybě – goto chain

```
void do_something() {  
    FILE *fin, *fout;  
    obj_t *obj;  
  
    fin = fopen("fin.txt", "r");  
  
  
  
  
  
  
  
  
  
    fout = fopen("fou.txt", "w");  
  
  
  
  
  
  
  
  
  
}
```

```
obj = malloc(sizeof(obj_t));  
  
  
  
  
  
  
  
  
  
// do some work...  
  
free(obj);  
fclose(fout);  
fclose(fin);
```

```
}
```

Zotavení po chybě – goto chain

```
int do_something() {  
    FILE *fin, *fout;  
    obj_t *obj;  
    int ecode = 0;  
    fin = fopen("fin.txt", "r");  
    if (fin == NULL) {  
        ecode = errno;  
        goto err_fin;  
    }  
  
    fout = fopen("fou.txt", "w");  
    if (fout == NULL) {  
        ecode = errno;  
        goto err_fout;  
    }  
}
```

inicializace a úklid:

10 vs. 27 řádků kódu, tj. 270% nárůst!
Relativně k celku cca 30-40% nárůst.

```
obj = malloc(sizeof(obj_t));  
if (obj == NULL) {  
    ecode = errno;  
    goto err_obj;  
}  
  
// do some work...  
  
free(obj);  
err_obj:  
    fclose(fout);  
err_fout:  
    fclose(fin);  
err_fin:  
    return ecode;  
}
```

Pozor na goto

- goto je (programátorem čtenářem) neočekávaná změna toku → špatně se orientuje v kódu.
- goto by nemělo skákat do bloku s překrytými identifikátory (čitelnost).
- goto by nemělo skákat “nahoru”, hrozí cyklus.
- Skoky mimo oblast funkce se řeší pomocí setjmp+longjmp.
- Zhodnocení: pokud to jde s ohledem na čitelnost, **vyhnout se skokům**. Pokud to nejde, “vyskákat” pouze dolů a pojmenovat návěští podle výjimečnosti (prefix err, fail, apod.)

Osnova přednášky

1. Ladění programů, pokračování.
2. Zpracování chyb
3. Ověřování správnosti programů.
4. Dokumentace zdrojových kódů.

Ověřování správnosti - motivace

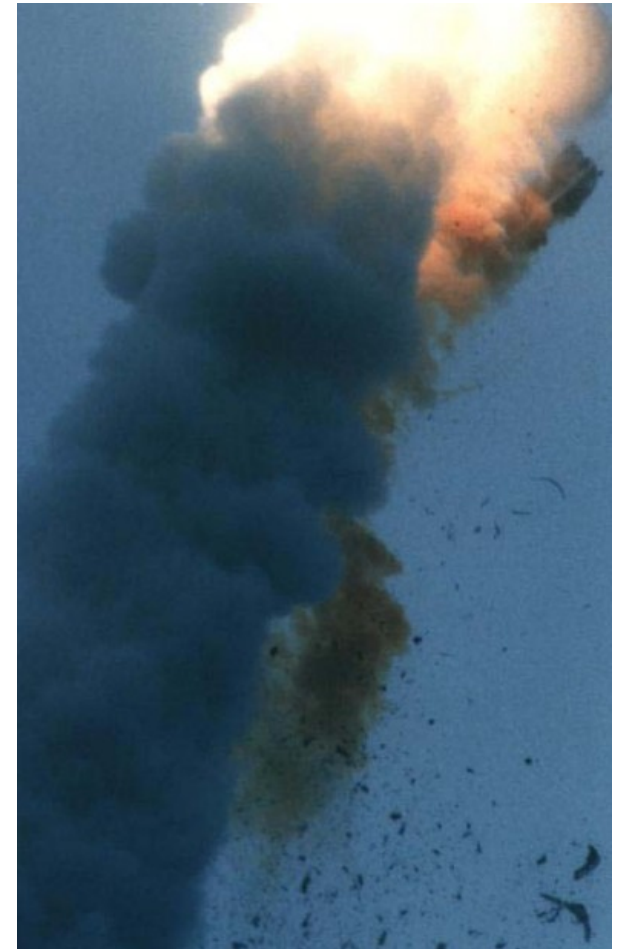
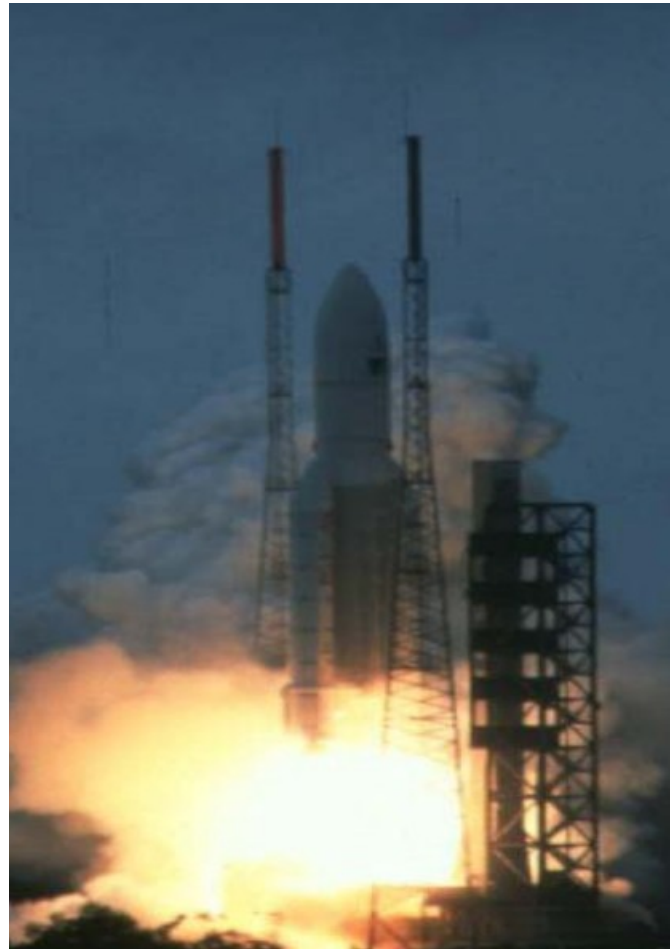
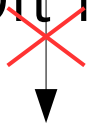
- Proč se zabývat správností programů?
 - V programech jsou chyby. Mnoho chyb se projeví v kritickém okamžiku, některé se projevují průběžně, jiné se však neprojeví nikdy.
 - Tato globálně zatím neodstranitelná skutečnost způsobuje obrovské škody (od fatálních havárií po ekonomické ztráty).
 - Cílem je vypracovat metodiku, která by verifikovala správnost programu proti jeho specifikaci.

Proč se zabývat správností programů

- 4.6.1996 – raketa Ariane5 explodovala 40 s po startu

64bit float

16bit int



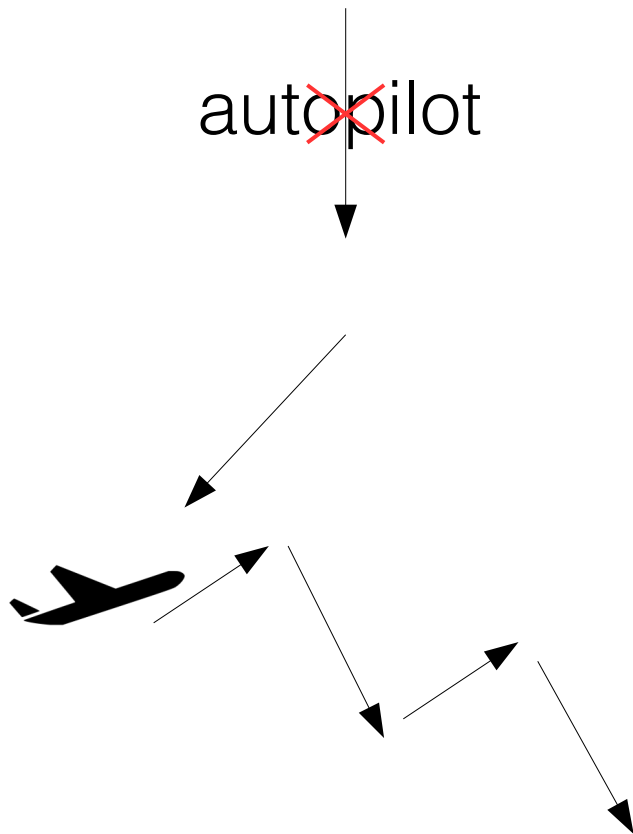
Proč se zabývat správností programů

- 14.8.2003 – část východního pobřeží USA přišla o elektrickou energii (SW chyba v řídicím středisku rozvodné sítě)



Proč se zabývat správností programů

- Srpen 2005 – SW Boeingu 777 poskytoval špatné údaje o rychlosti a zrychlení "air data inertial reference unit" (ADIRU)



Ověřování správnosti - definice

- O algoritmu je možné tvrdit, že je **správný**, když lze dokázat, že je správný vzhledem k jeho **specifikaci**.
- Rozlišujeme validaci a verifikaci programů/systémů:
- **Validate** = cílem je posoudit, zda tvořený systém je ten původně zamýšlený:
 - “Děláme správný systém?”
(Are we building the right system?)
- **Verifikace** = cílem je posoudit správnost programu:
 - “Děláme systém správně?”
(Are we building the system right?)

Ověřování správnosti - definice

Verifikaci dělíme podle základních principů:

- **Testování** = systematické spouštění programů s cílem získat povědomí o jejich kvalitě.
- **Formální analýza** = proces formálního dokazování toho, že počítačový program dělá přesně to, co je uvedeno ve specifikaci programu.
 - Formální analýza programu se zabývá podáním formálního důkazu správnosti algoritmu.
 - Specifikace musí být formální.

Testování vs. formální analýza

- Testováním programu lze pouze **zjistit přítomnost chyby**.
- Testováním **nelze potvrdit nepřítomnost** chyby:
 - Opakovaným spouštěním programu můžeme potvrdit, jak se program choval při daných argumentech.
 - Nemůžeme potvrdit, že se bude chovat stejně pro podobné, ale jiné vstupní hodnoty.
 - Dokonce nemůžeme potvrdit, že se program bude chovat stejně při opakovaném spuštění se stejnými vstupními hodnotami (např. programy založené na náhodných nebo neovlivnitelných datech).

Testování vs. formální analýza

- Ke každému správnému programu lze teoreticky sestavit důkaz.
- Čím složitější konstrukce programu, tím násobně (někdy dokonce exponenciálně) složitější důkaz.
- Formální analýza je efektivní pro:
 - malé programy nebo
 - dokazování pouze vybraných vlastností = např. celočíselné operace, přístupy do paměti, dosažitelnost úseku kódu.
 - více o formálních analýzách v předmětu FAV (magisterský obor).

Testovatelné programy

Zásady pro psaní jednoduše testovatelných programů:

- malé a jednoduché funkce,
- dobře zdokumentované části:
 - dokumentovat funkce, datové typy, celé zdrojové soubory,
- defenzivní programování - dobře jištěné programové konstrukce, testování i na první pohled neobvyklých stavů,
- žádné speciální triky,
- bez povrchové optimalizace:
 - povrchová optimalizace = mírné snížení náročnosti (ušetření proměnných nebo počtu příkazů),
 - hluboká optimalizace zahrnuje výměnu algoritmu.

Testovatelné, odolné programy

- Proto - **testovat, co se dá**:
 - vstupní data (neexistuje tak velký nesmysl, aby ho některý uživatel nebo jiný program na vstupu nedodal),
 - návratové hodnoty funkcí,
 - validitu argumentů funkcí,
 - číselné hodnoty před aritmetickou operací (dělitelnost nulou, přetečení, podtečení, zaokrouhlovací chyby).
- nebo **připravit systém na zotavení** po selhání (tzv. fault-tolerant systémy):
 - detekce selhání - např. využití výjimek (vyšší programovací jazyky), signálů (prostřednictvím operačního systému) nebo monitorů sebe sama (např. kontrola “kanárků”),
 - úprava vstupů nebo vlastního chování a restart problematické akce.

Základní princip testu jednotky

- Základní princip testu **jednotky**:
 - jednotka = jedna funkce, jedna třída, jeden soubor
 - jednotka = unit => unit test (jednotkový test nebo test jednotky)
 - testu říkáme **testovací případ** a zahrnuje vstupy a očekávané výstupy.
- Obecný proces testovacího případu jednotky:
 1. Příprava jednotky (setup)
 2. Spuštění jednotky (exercise)
 3. Ověření výstupů (verify)
 4. Vyčištění po testu (teardown)

Příklad testu jednotky

TEST1: ověření stability řadící metody ve funkci `sort_list`

```
// 1. Setup
ItemData TEST1DATA[] = {{1,'x'}, {42,'y'}, {3,'z'}, {42,'p'}, {242, 'z'}};
List list;
init_list(&list);
append_data_from_array(&list, TEST1DATA, 5);

// 2. Exercise
sort_list(&list, BY_ID);

// 3. Verify
Item *i = find_by_id(&list, 42);
assert(i != NULL);
assert(i->next != NULL);
assert(i->data.second == 'y');
assert(i->next->data.second == 'p');

// 4. Teardown
delete_list(&list);
```

Testovací vstupy: data pro přípravu `list`, hodnoty argumentů pro `sort_list`: `&list`, `BY_ID`

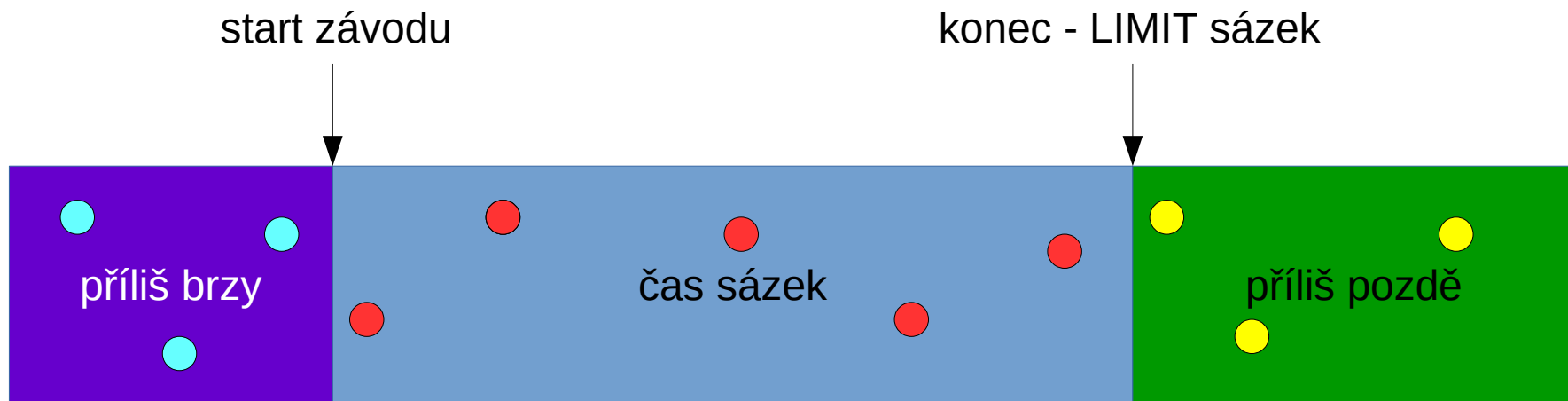
Testovací výstupy: pořadí (42,y)->(42,p), vyčištěná paměť

Generování testovacích dat

- Existují různé techniky testování.
- Každá technika cílí na jiný aspekt programu, ve kterém může být chyba:
 - každý řádek programu (line-coverage),
 - každé větvení programu (branch-coverage),
 - každá neotestovaná funkce (function-coverage),
 - každá konstanta, cesta v řízení programu,
 - více viz předmět Testování a dynamická analýza (ITS)
- Technika určuje, jakým způsobem se volí testovací data.
- Dvě z intuitivních technik jsou:
 - Equivalence partitioning (rozklad ekvivalenčních tříd),
 - Boundary-value analysis BVA (analýza mezí),

Generování testovacích vstupů

- Technika rozkladu ekvivalenčních tříd:
 - Vstupní data se rozdělí na intervaly. Všechny hodnoty v daném intervalu jsou z pohledu funkcionality/správnosti ekvivalentní.



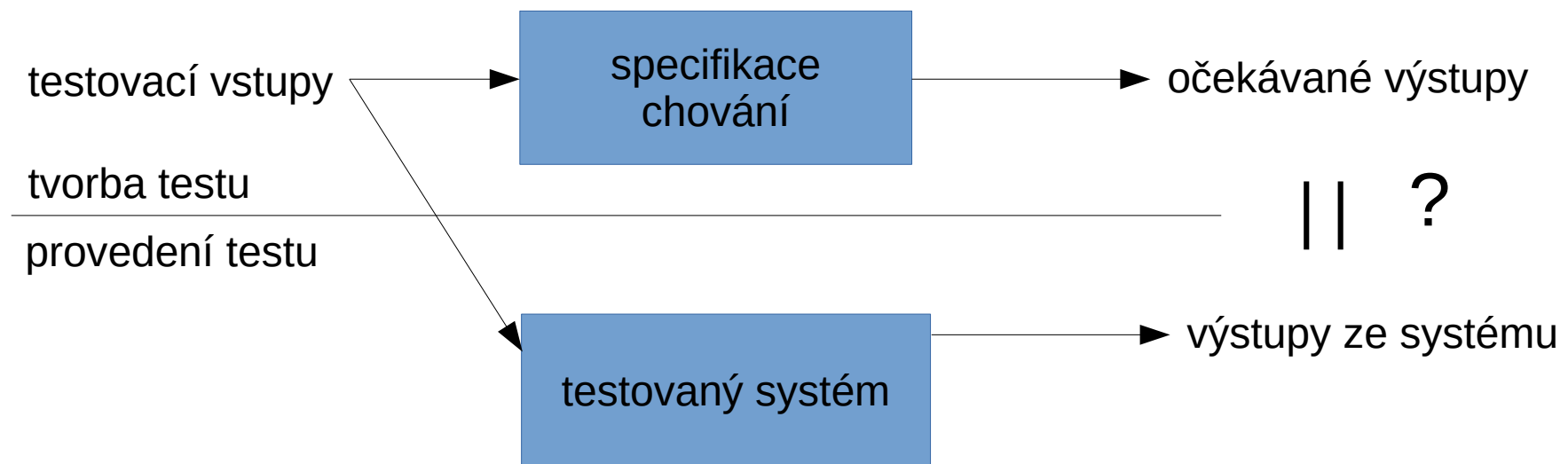
Generování testovacích vstupů

- Technika BVA - analýza mezí:
 - Ve struktuře programu (pokud je známá) nebo ze známých datových typů jsou vybrány mezní hodnoty. Pro každou mez M jsou testovací vstupy vybrány 3: $a = M-1$, $b = M$, $c = M+1$.
 - Hodnota ± 1 může reprezentovat libovolnou jednotku datového typu M , např:
 - minimální přesnost u čísel s plovoucí řád. čárkou,
 - jednotka délky řetězce nebo pole,
 - jedna úroveň zanoření adresáře, pokud M je řetězec cesty k souboru.



Generování testovacích výstupů

- Vstupy + specifikace chování \Rightarrow očekávané výstupy.
- Vstupy + testovaný systém \Rightarrow skutečné výstupy.
- Pokud výstupy neodpovídají, pak test tzv. selže \Rightarrow důkaz, že systém není správný.



Příklad testovacích dat

```
int dom_detect(int dom)
{
    if (dom < 0)
        return NOT_DATE;
    else if (dom > 31)
        return NOT_DATE;
    else
        return DAY_OF_MONTH;
}
```

} testovaná funkce

Specifikace:

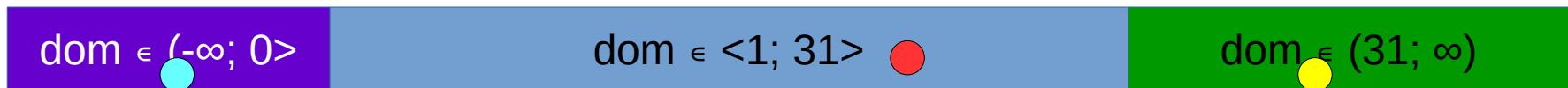
Popis: Funkce dom_detect zjišťuje, zda číslo dom odpovídá dni v libovolném měsíci.

Návratová hodnota: NOT_DATE, pokud číslo dni neodpovídá, DAY_OF_MONTH, pokud dom odpovídá číslu dne v měsíci.

Příklad Equivalence partitioning

```
int dom_detect(int dom)
{
    if (dom < 0)
        return NOT_DATE;
    else if (dom > 31)
        return NOT_DATE;
    else
        return DAY_OF_MONTH;
}
```

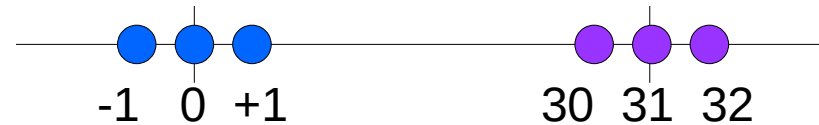
Třídy ze specifikace:



test	třída	hodnota dom	očekávaný výsledek
T1 ✓	dom $\in <-\infty; 0)$	-5	NOT_DATE
T2 ✓	dom $\in <0; 31)$	20	DAY_OF_MONTH
T3 ✓	dom $\in <31; \infty)$	42	NOT_DATE

Příklad Boundary Value Analysis

```
int dom_detect(int dom)
{
    if (dom < 0)
        return NOT_DATE;
    else if (dom > 31)
        return NOT_DATE;
    else
        return DAY_OF_MONTH;
}
```



test	mez	hodnota dom	očekávaný výsledek
T1 ✓	dom: 0	-1	NOT_DATE
T2 ✗	dom: 0	0	NOT_DATE
T3 ✓	dom: 0	+1	DAY_OF_MONTH
T4 ✓	dom: 31	30	DAY_OF_MONTH
T5 ✓	dom: 31	31	DAY_OF_MONTH
T6 ✓	dom: 31	32	NOT_DATE

Osnova přednášky

1. Ladění programů, pokračování
2. Zpracování chyb
3. Ověřování správnosti programů
4. Dokumentace zdrojových kódů

Dokumentování programů

- Existují různé typy dokumentací podle jejich účelu.
- Programátor se nejčastěji setkává (píše nebo čte) se 3 nejčastějšími:
 - Uživatelská dokumentace.
 - Architektonický návrh.
 - Technická dokumentace.

Uživatelská dokumentace

- Cílí na uživatele. Uživatelem se myslí aktér, který používá program a nechce/nepotřebuje znát jeho interní detaily.
- Popisuje, jak se program používá.
- Může mít různou formu:
 - Tutoriál = provádí uživatele vlastnostmi programu spirálovitým způsobem - od jednoduchého příkladu ke složitějším vlastnostem.
 - Popis samostatných vlastností/činností = dokumentaci lze procházet skokově. Typické pro popis řešení problémů (troubleshooting).
 - Referenční příručka = hypertextové odkazy na terminologii používanou v programu.

Architektonický návrh

- Architektonický návrh je určen pro uživatele vyžadující znát interní struktury (angl. internals), tj. systémové inženýry nebo budoucí vývojáře.
- Na vysoké úrovni popisuje strukturu nebo chování systému.
- Často dokumentace obsahuje diagramy (bloková schemata, vývojové diagramy, tabulky).
- Na diagramy je vhodné použít jazyk UML.
- Architektonický návrh je také standardizován: [IEEE 1016-2009 Software Design Description](#)

Technická dokumentace

- Technickou dokumentací se myslí dokumentace zdrojových kódů (samostatných nebo celého projektu).
- Pro vývojáře projektu nebo pro uživatele/vývojáře projekt využívající.
- Různé formy dokumentací:
 - Dokumentační řetězce segmentu kódu (Docstring).
 - Dokumentace aplikačního rozhraní (Javadoc).
 - README soubory.
 - Manuálové stránky (manpages).
- Většinou se jedná o dobře strukturovaná data => existence nástrojů pro tvorbu, čtení a hledání dokumentace.

Technická dokumentace - Docstring

- Dokumentační řetězce (Docstring) jsou řetězcové literály uprostřed kódu.
- Syntakticky (a většinou i sémanticky) akceptované programovacím jazykem.
- Blíží se k tzv. samodokumentovanému kódu, tj. “Nepotřebuji ten kód dokumentovat, je dokumentován dostatečně sám.”
- Jazyky používající docstring: Python, Lisp
- Příklad:

```
$ python
>>> def foo(bar):
...     "This is a foo function. It needs an
argument."
...     pass
...
>>> print foo.__doc__
This is a foo function. It needs an argument.
```


Technická dokumentace - Javadoc

- Javadoc je nástroj pro automatické generování dokumentací zdrojových kódů (původně pro Java od Sun Microsystems).
- Nyní převzato jako formát dokumentací.
- **Javadoc** = dokumentace v komentářích programu. Komentáře mají **danou strukturu** a používají **vlastní klíčová slova (značky, tags, commands)**.
- Nástroje pro automatické generování dokumentací prochází zdrojové kódy a k syntaktickým úsekům kódu přidávají dokumentaci převzatou z komentářů.

Technická dokumentace - Javadoc

- Typická syntaxe:

```
/**  
 * ...A long description of the function.  
 * ...  
 *  
 * @param table the hash table.  
 * @param key the hash key.  
 * @param cinf if #TRUE, create the entry if it didn't exist.  
 * @param iter the iterator to initialize.  
 * @returns #TRUE if the hash entry now exists.  
 */  
dbus_bool_t _dbus_hash_iter_lookup (DBusHashTable *table,  
                                     void          *key,  
                                     dbus_bool_t    cinf,  
                                     DBusHashIter   *iter)  
  
{ ...
```

začátek Javadoc

klíčová slova Javadoc

Technická dokumentace - Javadoc

- Doxygen - jeden z nástrojů pro automatickou tvorbu hypertextových dokumentací.
- Využívá konfigurační soubor (Doxyfile) pro specifikaci:
 - ve kterých souborech se má hledat,
 - které části se mají ignorovat,
 - jaký má být formát výstupu a na jakém umístění, apod.
- Dokumentační komentáře mohou obsahovat celou řadu klíčových slov, viz [oficiální stránka doxygen](#) / příkazy
- Existují další zásuvné moduly (např. [plantUML](#)), které automaticky z textu vykreslí např. graf závislostí, diagram tříd, use-case diagramy.

Technická dokumentace - Doxygen

- Příklad:

```
$ cd path/to/project
$ doxygen -g
$ doxygen
```

• • •

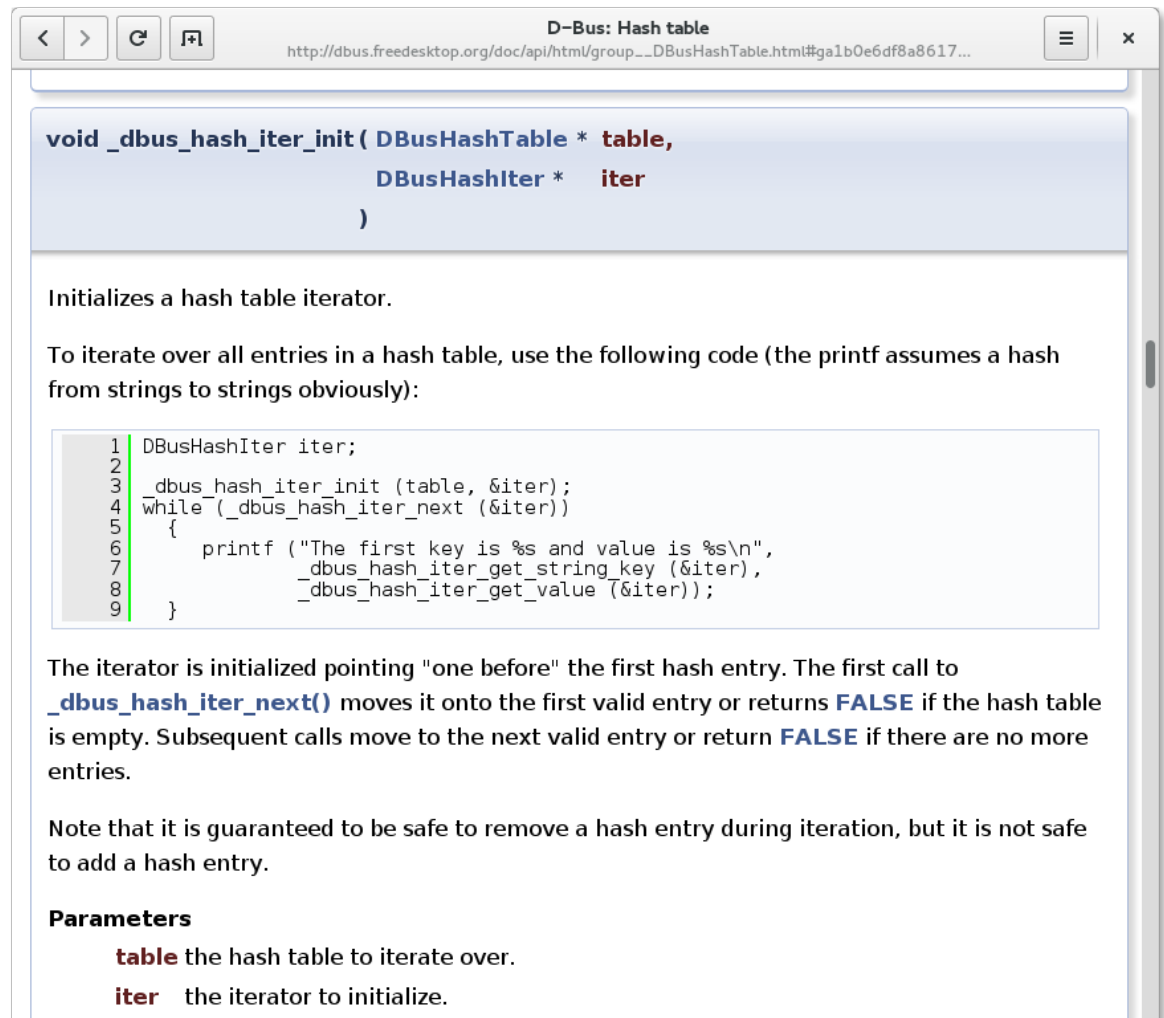
• • •

• • •

• • •

• • •

• • •



The screenshot shows a web browser window with the title "D-Bus: Hash table" and a URL. The main content area displays the function signature for `_dbus_hash_iter_init` in a light blue box. Below the signature, there is a description of the function's purpose, followed by a code example for iterating over a hash table. The code example is enclosed in a light blue box with line numbers. Below the code, there is a detailed explanation of the iterator's behavior, including a note about safety during iteration. The "Parameters" section is at the bottom, listing the `table` and `iter` arguments.

```
void _dbus_hash_iter_init( DBusHashTable * table,
                          DBusHashIter * iter
                          )
```

Initializes a hash table iterator.

To iterate over all entries in a hash table, use the following code (the printf assumes a hash from strings to strings obviously):

```
1 DBusHashIter iter;
2
3 _dbus_hash_iter_init (table, &iter);
4 while (_dbus_hash_iter_next (&iter))
5 {
6     printf ("The first key is %s and value is %s\n",
7             _dbus_hash_iter_get_string_key (&iter),
8             _dbus_hash_iter_get_value (&iter));
9 }
```

The iterator is initialized pointing "one before" the first hash entry. The first call to `_dbus_hash_iter_next()` moves it onto the first valid entry or returns **FALSE** if the hash table is empty. Subsequent calls move to the next valid entry or return **FALSE** if there are no more entries.

Note that it is guaranteed to be safe to remove a hash entry during iteration, but it is not safe to add a hash entry.

Parameters

- table** the hash table to iterate over.
- iter** the iterator to initialize.

Technická dokumentace - README

- Soubor README je textový soubor obsahující dokumentaci k projektu nebo k aktuálnímu adresáři.
- Název souboru vznikl kvůli tomu, aby neznalí uživatelé věnovali pozornost tomuto souboru jako prvnímu. Velká písmena jsou zvolena kvůli řazení na začátek seznamu souborů (podle ASCII).
- Soubor README typicky obsahuje následující kapitoly:
 - Kontaktní osobu, autora.
 - Návod k instalaci, návod k odinstalaci.
 - Seznam souborů s jejich popisem.
 - Odkaz na licenci projektu/programu.
 - Známé chyby, popisy řešení problémů.
- Může mít různé formáty. Nejčastější je holý text (soubor README nebo README.txt) nebo tzv. [Markdown](#) (soubor README.md).

README.md - formát Markdown

- [Markdown](#) je formát textového souboru obsahující ASCII znaky představující strukturované formátování (podtržený nadpis, odrážky pomocí hvězdičky apod.).
- Možnost automatické konverze do jiných formátů (html, pdf).
- Příklad:

```
# Welcome to MkDocs
```

```
For full documentation visit [mkdocs.org](http://mkdocs.org).
```

```
## Commands
```

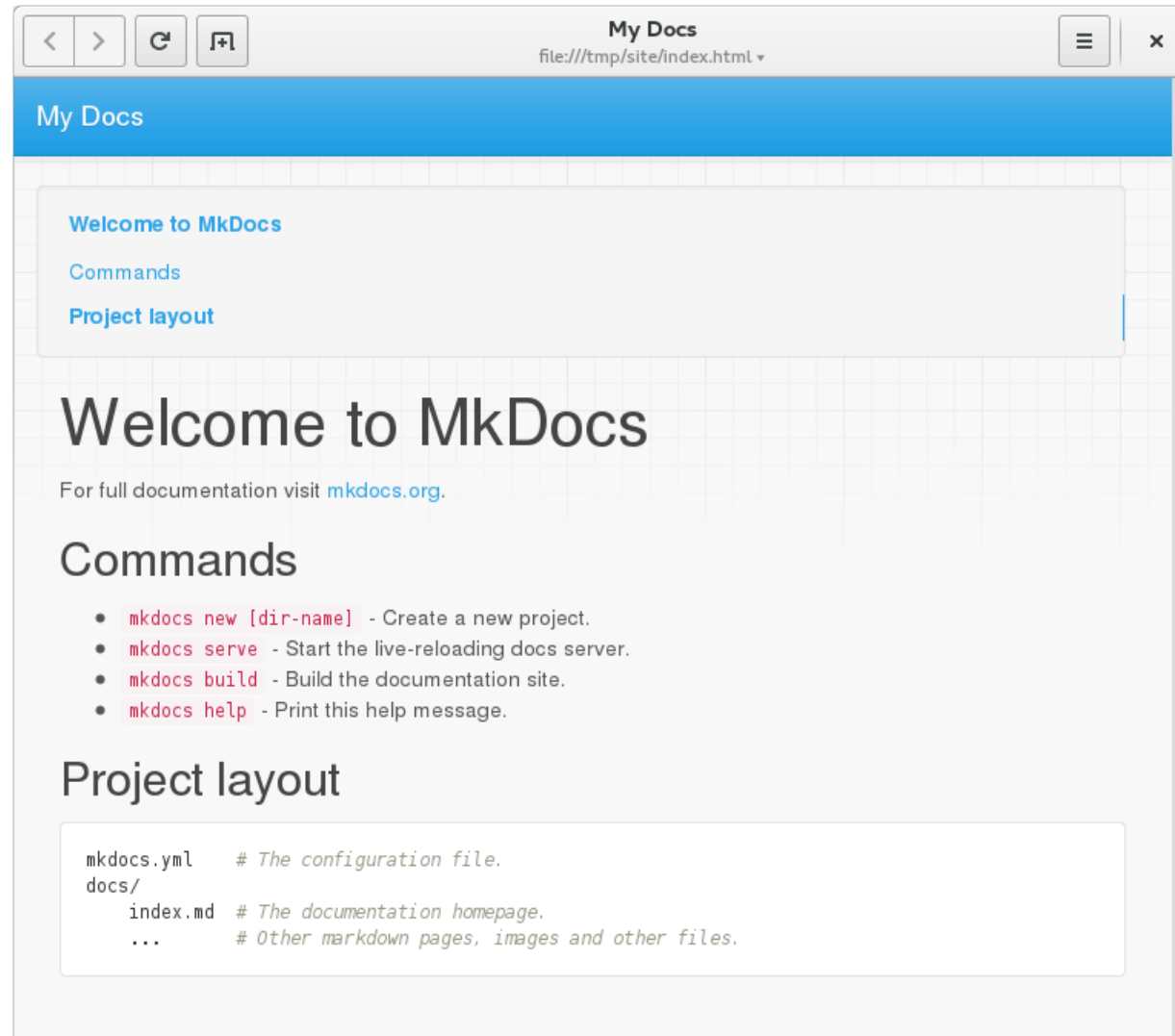
```
* `mkdocs new [dir-name]` - Create a new project.  
* `mkdocs serve` - Start the live-reloading docs server.  
* `mkdocs build` - Build the documentation site.  
* `mkdocs help` - Print this help message.
```

```
## Project layout
```

```
mkdocs.yml    # The configuration file.  
docs/  
  index.md    # The documentation homepage.  
  ...         # Other markdown pages, images and other files.
```

README.md - formát Markdown

- Příklad pokr.: `$ mkdocs build`



Manuálové stránky

- Strukturovaná stránka popisující jeden příkaz/volání. Původ v UNIXových systémech.
- Vyvolává se pomocí příkazu man.
- Typicky obsahuje kapitoly:
 - NAME - dlouhý název
 - SYNOPSIS - syntaxe použití s parametry
 - DESCRIPTION - podrobný popis chování
 - RETURN VALUE - návratová hodnota
 - ERRORS - možné chybové případy
 - CONFORMING TO - odkaz na standardy, ke kterým náleží
 - SEE ALSO - odkaz na další zpřízněné manuálové stránky

Další dokumentační zdroje

- Podobně jako manuálové stránky existují detailnější [info stránky](#). Unixový příkaz `info`.
- Další hypertextové dokumenty - lokální nebo hostované na serveru:
 - [devdocs.io](#) - HTML5 aplikace, po stažení dokumentace lze používat offline.
 - [readthedocs.org](#) - projekt pro hostování různých dokumentací (od API po uživatelské tutoriály).
 - [sphinx](#), [mkdocs](#) - nástroje pro tvorbu hypertextových dokumentací z textových souborů typu markdown.

```
$ mkdocs new proj
$ cd proj
$ mkdocs serve
... navigate to http://127.0.0.1:8000
^C
$ mkdocs build
```



How the customer explained it



How the Project leader understood it



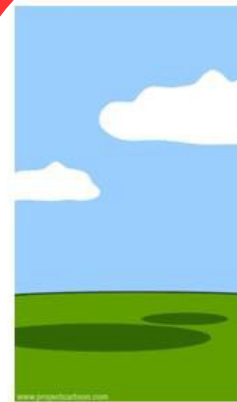
How the Business Consultant described it



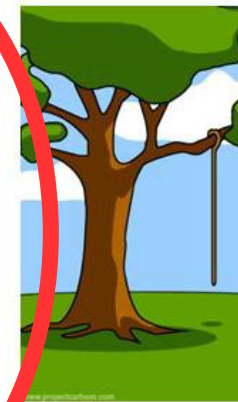
How the Analyst designed it



How the Programmer wrote it



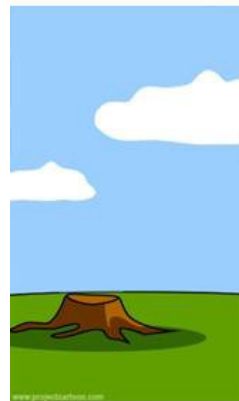
How the project was documented



What Operations installed



How it performed under load



How it was supported



What marketing advertised



How the customer was billed



What the customer really needed

"How the project was documented"

Otázky

- Jaký je rozdíl mezi statickou a dynamickou analýzou programů?
- Jaký je účel inspektorů, monitorů a stopování programů?
- Jaké jsou možnosti předávání chybového stavu?
- Jaké jsou možnosti vyrovnání se s chybovým stavem?
- Můžeme testováním nebo formální analýzou potvrdit či vyvrátit přítomnost chyby v programu?
- Jaké jsou základní druhy dokumentací programového vybavení?
- Popište rozdíl mezi dokumentací docstring a javadoc.
- Jaké typické informace je možné získat ze souboru README?