Prostriedky pre tvorbu korektných programov

Ošetrenie argumentov

Makro assert

Chyby systémových volaní

Statické a dynamické knižnice

Ošetrenie argumentov programu

- Sú dve kategórie argumentov: voľby (options) a ostatné argumenty
- Voľby:

  krátke (-h) -jeden charakter

  dlhé: (--help) dva --skupina charakterov (meno)

Použitie funkcie *getopt_long*

- Ošetruje krátke aj dlhé voľby. Vyžaduje inkludovať <getopt.h>

Príklad

Short Form	**Long Form**	**Purpose**
-h	--help	Display usage summary and exit
-o filename	--output filename	Specify output filename
-v	--verbose	Print verbose messages

getopt_long

- Dostáva argumenty *main* funkcie argc a argv, reťazce charakterov krátkych volieb, štruktúru dlhých volieb a NULL
- Špecifikácia krátkych volieb:

  ho:v po voľbe ktorá si vyžaduje argument musí nasledovať ":"
- Špecifikácia dlhých volieb:

```
const struct option long_options[ ] = {
  { "help",       0, NULL, 'h' },
  { "output",   1, NULL, 'o' },
  { "verbose", 0, NULL, 'v' },
  { NULL,       0, NULL, 0  }
};
```

1 je po tej voľbe, ktorá si vyžaduje argument

getopt_long spracovanie argumentov

- Vždy keď sa funkcia volá vracia písmeno krátkej voľby alebo –1
- v prípade nesprávnej voľby vracia ?
- pre voľbu, ktorá má argument je adresa reťazca v globálnej premenej *optarg*
- po parsovaní všetkých volieb je v globálnej premennej *optind* index (z argv) prvého argumentu

Príklad

```
const char* program_name;    /* The name of this program. */
void print_usage (FILE* stream, int exit_code)
{
  fprintf (stream, “Usage: %s options [ inputfile ... ]\n”,
   program_name);
  fprintf (stream,
    “ -h --help Display this usage information.\n”
    “ -o --output filename Write output to file.\n”
    “ -v --verbose Print verbose messages.\n”);
  exit (exit_code);
}
```

Príklad

```
/* Main program entry point. ARGC contains number of
   argument list elements; ARGV is an array of pointers to
   them. */
int main (int argc, char* argv[]){
  int next_option;
  /* A string listing valid short options letters. */
  const char* const short_options = “ho:v”;
  /* An array describing valid long options. */
  const struct option long_options[] = {
    { “help”, 0, NULL, ‘h’ },
    { “output”, 1, NULL, ‘o’ },
    { “verbose”, 0, NULL, ‘v’ },
    { NULL, 0, NULL, 0 }        /* Required at end of array. */
  };
```

```
 /* The name of the file to receive program output,
    or NULL for standard output. */
const char* output_filename = NULL;
 /* Whether to display verbose messages. */
 int verbose = 0;
 /* Remember the name of the program, to
    incorporate in messages. The name is stored in
    argv[0]. */
 program_name = argv[0];
```

```
do {
   next_option = getopt_long (argc, argv, short_options,
  long_options, NULL);
 switch (next_option)
 {
  case ‘h’:                    /* -h or --help */
  /* User has requested usage information. Print it to
   standard output, and exit with exit code zero (normal
   termination). */
  print_usage (stdout, 0);
```

```
case ‘o’:                          /* -o or --output */
/* This option takes an argument, the name of the
  output file. */
 output_filename = optarg;
 break;
 case ‘v’:                         /* -v or --verbose */
 verbose = 1;
 break;
```

```
  case ‘?’:                 /* The user specified an invalid option. */
  /* Print usage information to standard error, and exit with
     exit code one (indicating abnormal termination). */
  print_usage (stderr, 1);
  case -1:                   /* Done with options. */
  break;
  default:                  /* Something else: unexpected. */
  abort ();
 }
}
while (next_option != -1);
```

```
  if (verbose) {
    int i;
    for (i = optind; i < argc; ++i)
      printf ("Argument: %s\n", argv[i]);
  }
  /* The main program goes here. */
  return 0;
}
```

Písanie spoľahlivého kódu

- Napísať program, ktorý sa sa normálnych okolností správa korektne je ťažké ale napísať program, ktorý sa správa korektne v prípade chybových situácií je ešte ťažšie.
- Nečakané chyby je potrebné nájsť ešte počas vývoja resp. testovania aplikačného programu. Keď sa objavia až u používateľa aplikácie je už neskoro.

Použitie makra *assert*

- Argumentom makra je boolovský výraz. Program je ukončený ak je výraz vyhodnotený ako FALSE, pritom vypíše chybovú správu obsahujúcu zdrojový súbor, číslo riadku a text výrazu.
- Makro sa používa napr. na zistenie platnosti argumentov funkcie, testovanie pre a post podmienok volania funkcie, testovanie neočakávaných návratových hodnôt, atď.
- Informuje čitateľa programu, že podmienka v makre musí byť vždy TRUE ak nie, je v programe chyba.

Použitie makra *assert*

- Vo výraze makra sa nemá používať:

  1. volania funkcií – spomalenie rýchlosti výpočtu.

  Napríklad nie

```
for (i = 0; i < 100; ++i)
assert (do_something () == 0);
```

  ale

```
for (i = 0; i < 100; ++i) {
int status = do_something ();
assert (status == 0);}
```

Použitie makra *assert*

2. priradenie hodnoty premenným
3. modifikačné operátory (++, --)
4. testovanie užívateľského vstupu

Použitie makra *assert*

- Vhodné použitia makra sú:
- testovanie smerníka na NULL

  {assert (pointer != NULL)},

  Chybové hlásenie:

  Assertion ‘pointer != ((void *)0)’ failed je viac informatívne ako

  Segmentation fault (core dumped), ktoré vznikne ak v programe dereferencujete nulový smerník
- testovanie argumentov funkcie na nejakú hodnotu

  assert (foo > 0);

  zvyšuje čitatelnosť kódu, uvádza sa na začiatku funkcie

Chyby systémových volaní

- Systém sa pokúša o získanie zdrojov mimo prípustných hraníc (alokácia príliš veľkej pamäte, otvorenie veľkého počtu súborov, zápis veľkého množsta údajov na disk)
- Program sa pokúša vykonať operáciu, na ktorú nemá oprávnenie (pokus o zápis do read-only súbora, prístup do pamäti iného procesu, zrušenie iného programu)
- Argumenty systémových volaní môžu byť chybné (neplatná adresa pamäte alebo deskriptora súboru, otvorenie adresára ako obyčajného súboru, atď.)

Chyby systémových volaní

- Systémové volania môžu zlyhať z dôvodu, ktorý je vzhľadom k programu externý (chybný hardvér, chýbajúci floppy disk alebo CD nosič)
- Systémové volanie je prerušené externou udalosťou ako je príchod signálu. Ak je to potrebné volajúci program musí reštartovať systémové volanie.

Chybový kód systémových volaní

- Mnoho systémových volaní používa na indikáciu chyby špeciálnu premennú errno
- Pretože všetky systémové volania používajú tú istú premennú je potrebné ihneď po vzniku chyby ju skopírovať do inej premennej
- Hodnoty premennej sú možné hodnoty makier predprocesora konvenčne uvádzané veľkými písmenami začínajúcimi s E. Treba inkludovať <errno.h>

Chybový kód systémových volaní

- GNU/Linux poskytuje funkciu *strerror*, korá vracia reťazec popisujúci chybový kód vhodný pre chybovú správu. Treba inkludovať <string.h>
- Funkcia *perror* vypíše popis chyby priamo do streamu stderr (chybový výstup). Inkludovať <stdio.h>

```
fd = open (“inputfile.txt”, O_RDONLY);
if (fd == -1) {
/* The open failed. Print an error message and exit. */
fprintf (stderr, “error opening file: %s\n”, strerror (errno));
exit (1);
}
```

Chybový kód systémových volaní

- Jedným z možných chybových kódov pre I/O funkcie je EINTR. Niektoré program blokujúce funkcie select, sleep, read po príchode signálu sú prerušené bez skončenia operácie, pričom errno má hodnotu EINTR. Obyčajne je treba v tomto prípade opakovať systémové volania.
- V prípade vzniku chyby je treba v závislosti od účelu syst. volania a hodnoty errno vykonať:

  vytlačiť chybovú správu

  zrušiť operáciu

  abortovať program

  zopakovať operáciu alebo ignorovať chybu

Príklad

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  /* Save errno because it's clobbered by the next system call. */
  int error_code = errno;
  /* The operation didn't succeed; chown should return -1 on error. */
  assert (rval == -1);
  /* Check the value of errno, and take appropriate action. */
  switch (error_code) {
    case EPERM: /* Permission denied. */
    case EROFS: /* PATH is on a read-only file system. */
    case ENAMETOOLONG: /* PATH is too long. */
    case ENOENT: /* PATH does not exit. */
    case ENOTDIR: /* A component of PATH is not a directory. */
    case EACCES: /* A component of PATH is not accessible. */
```

Príklad

```
    /* Something's wrong with the file. Print an error message. */
    fprintf (stderr, "error changing ownership of %s: %s\n", path, strerror(error_code));
    /* Don't end the program; perhaps give the user a chance to choose another file... */
    break;
  case EFAULT:
    /* PATH contains an invalid memory address. This is probably a bug. */
    abort ();
  case ENOMEM:
    /* Ran out of kernel memory. */
    fprintf (stderr, "%s\n", strerror (error_code));
    exit (1);
  default:
    /* Some other, unexpected, error code. We've tried to handle all
    possible error codes; if we've missed one, that's a bug! */
    abort ();
  };
}
```

Chyby a alokácie zdrojov

- Niekedy po vzniku chyby nie je nutné ukončiť program, stačí vrátiť sa z funkcie s indikáciou chyby
- Pred návratom z funkcie musia byť všetky úspešne alokované zdroje dealokované.

  Alokované zdroje môžu byť: pamäť, deskriptory súborov, smerníky na súbory, synchronizačné objekty, atď.

Alokovanie zdrojov- príklad

- Uvažujme funkciu, ktorá číta zo súboru do bufra.
- Obsahuje tieto kroky:
  1. Alokácia bufra
  2. Otvorenie súboru
  3. Čítanie zo súboru do bufra
  4. Zatvorenie súboru
  5. Vrátenie bufra
- Ak je chybný krok 2 je treba dealokovať bufer alokovaný v kroku 1.
- Ak je chybný krok 3 je treba okrem toho aj zatvoriť súbor otvorený v kroku 2.

Príklad

Freeing Resources During Abnormal Conditions

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  ssize_t bytes_read;
```

Príklad

```
  /* Allocate the buffer. */
  buffer = (char*) malloc (length);
  if (buffer == NULL)
  return NULL;
  /* Open the file. */
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
     /* open failed. Deallocate buffer before returning. */
    free (buffer);
    return NULL;
}
```

Príklad

```
  /* Read the data. */
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    /* read failed. Deallocate buffer and close fd before returning. */
    free (buffer);
    close (fd);
    return NULL;
  }
  /* Everything's fine. Close the file and return the buffer. */
  close (fd);
  return buffer;
}
```

Tvorba a používanie knižníc

- Statické a dynamické linkovanie knižníc
- statické – väčší kód ale lepšia organizácia programu
- dynamické – menší kód, horšia organizácia programu
- empirické pravidlá - ktoré linkovanie použiť pre program

Archív – statická knižnica

- Je statická knižnica objektových súborov pamätaných v jednom súbore
- Tvorba:

  % ar cr libtest.a test1.o test2.o // vytvorí sa archív libtest.a

  voľba cr príkazu ar znamená vytvorenie archívu
- Linker hľadá v knižnici definície všetkých symbolov programu. Objektové súbory archívu, v ktorých sa definície symbolov nachádzajú sú následne vybrané a inklúdované do vykonávateľného kódu programu.

Dynamická knižnica

- Podobne ako archív obsahuje objektové súbory
- Podstatný rozdiel od statickej je pri linkovaní

  Výkonný kód obsahuje len referencie do dynamickej knižnice, preto sa niekedy nazýva aj zdieľaná knižnica
- Tvorba

  Preklad: % gcc -c -fPIC test1.c

  Voľba –fPIC sa použije pri tvorbe objektového súboru pre dynamickú knižnicu. PIC (Position Independent Code)

  Vloženie objektového súboru do knižnice:

  % gcc -shared -fPIC -o libtest.so test1.o test2.o

Dynamická knižnica

- Linkovanie:

  % gcc -o app app.o -L. –ltest

  Ak existujú obe knižnice libtest.a aj libtest.so linker vyberie dynamickú inak vyberie tú ktorá existuje
- Ak chceme linkovať so statickou knižnicou treba použiť voľbu static

  % gcc -static -o app app.o -L. –ltest
- Ak chceme linkovať s vlastnou knižnicou

  % gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib

Závery

- Dynamická knižnica šetrí pamäťový priestor pre inštaláciu programu
- Stačí upgradovať len knižnicu miesto všetkých programov, ktoré od nej závisia
- Niekedy je ale táto posledná výhoda nevýhodou, vtedy treba použiť statické linkovanie