Programování v jazyce C

10 Vazba na OS
Pomocné funkce

- Měření času a kalendářní funkce
- Interakce programu s OS
- Ladicí a diagnostický aparát
- Zpracování chybových stavů
- Vysokoúrovňové funkce
- Funkce s proměnným počtem argumentů

Interakce programu s prostředím
Obecné skutečnosti

- Interakcí programu s prostředím (tj. zejm. s OS) se rozumí především „násilné” **ukončování běhu programu**, volání **příkazového procesoru** OS, práce se systémovým **datem a časem**, práce s **proměnnými prostředí** a práce se **signály**.

- Sice dovolují provádět některé vysokoúrovňové operace, ale při jejich použití může poměrně snadno dojít k tomu, že **nebude možné program bez úprav spouštět na jiné platformě**.

→ týká se zejména např. funkce `system(·)`, která dovoluje vyvolat příkaz operačního systému, ve kterém je program spuštěn.

- Funkce se nachází v několika různých knihovnách – budou zmíněny postupně.

Datum a čas
Obecné skutečnosti

- Funkce a datové typy pro práci s (kalendářním) datem, systémovým časem a časem běžícího procesu poskytuje knihovna **time** → připojit hlavičkový soubor příkazem preprocesoru: `#include <time.h>`

- Jazyk C využívá buď tzv. ***integrální reprezentaci času***, kdy je čas vyjádřen jako **počet sekund od počátku epochy** (většinou od 00:00:00 UTC, 1. ledna 1970), nebo složkovou reprezentaci, kdy jsou složky `struct tm {…}` naplněny hodnotami dne, měsíce, roku, atd.

- Pro měření tzv. ***procesorového času*** (tj. času, po který CPU vykonává běžící proces) se používají ***tiky*** (***Ticks***) – abstraktní jednotky, které je třeba na časový údaj správně **převést**.

Datum a čas
Měření doby běhu procesu

```
clock_t clock();
```

- Funkce vrací (přibližný) čas, po který CPU vykonává volající proces. (nepřesnost v důsledku režie při přepínání kontextu)
- Čas je vrácen v obecných jednotkách – tikách (*Ticks*/*Clocks*), které se mohou lišit na různých platformách (často jsou to mili- nebo µsekundy).
- K převodu tiků na (SI) čas slouží makro **`CLOCKS_PER_SEC`**, které definuje počet tiků za 1 sekundu.
- Není-li čas běhu procesu k dispozici (z důvodů specifických pro daný OS nebo CPU), vrací funkce **`clock(·)`** hodnotu -1.

Pozor na přetečení! Je-li **`clock_t`** např. 32-bitový **`long`** (což je obvyklé) a **`CLOCKS_PER_SEC`** = 1000000, přeteče čas po každých zhruba 36 minutách…

Datum a čas
Měření doby běhu procesu – ukázka

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

int main() {
  clock_t t1, t2;

  t1 = clock();
  function_to_time();
  t2 = clock();

  printf("Function took %f secs.\n",
    ((double) (t2 - t1)) / CLOCKS_PER_SEC);

  return 0;
}
```

Měřit čas běhu funkcí/výpočtů jinak, než pomocí **`clock(·)`** či **profileru**, nemá smysl. Funkce kalendářního času nemají dostatečnou přesnost.

Přetypování je zde nutné, aby nedošlo k celočíselnému dělení (a ztrátě přesnosti).

Datum a čas
Integrální kalendářní čas (a datum)

```
time_t time(time_t *timer);
```

- Funkce vrací kalendářní čas kódovaný jako celé číslo (obvykle je **`time_t`** celočíselný typ **`long`**) – počet sekund od počátku epochy (tj. většinou od 1. 1. 1970).
- Je-li argument **`*timer`** různý od **`NULL`**, uloží se návratová hodnota také na předanou adresu.
- Nelze-li čas zjistit (funkci nepodporuje OS, resp. BIOS), vrací hodnotu -1.

- Vrácená hodnota se typicky předává funkci **`ctime(·)`** nebo **`asctime(·)`**, které převádí integrální reprezentaci času na čitelnou podobu.
- Výpočet časového intervalu mezi dvěma kalendářními časy provádí funkce **`difftime(·)`**.

Datum a čas
Kalendářní čas (a datum) jako struktura

```
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
```

- Obě funkce převádí integrální reprezentaci času do strukturované podoby, **`struct tm { … }`**.
- **`gmtime(·)`** převádí časovou značku na čas UTC (Universal Time – Co-ordinated), tj. čas v časové zóně GMT (Greenwich Mean Time).
- **`localtime(·)`** převádí časovou značku na místní čas, podle časového pásma aktivního locale (a příp. letního času).
- Při chybě vrací hodnotu **`NULL`**.

```
time_t mktime(struct tm *timeptr);
```

- Převádí strukturovanou podobu časové značky na integrální hodnotu typu **`time_t`** (nelze-li převést, vrací -1).

Datum a čas
Kalendářní čas (a datum) jako struktura

- Podoba struktury **`tm`** je **definována normou** ANSI C → programátor se může spolehnout, že se složky jmenují na všech platformách a implementacích knihovny stejně:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

- sekundy (rozsah 0 – **61**)
- minuty (rozsah 0 – 59)
- hodiny (rozsah 0 – 23)
- den v měsíci (rozsah **1** – 31)
- měsíc v roce (rozsah 0 – 11)
- rok od 1900 (≥ 0)
- den v týdnu (rozsah 0 – 6)
- den v roce (rozsah 0 – 365)
- příznak letního času (*Daylight Saving Time*)
  1 ≡ letní čas, 0 ≡ zimní čas, -1 ≡ nezn.

Datum a čas
Převod data/času na řetězec

```
char *ctime(const time_t *timestamp);
char *asctime(const struct tm *timestruct);
```

- Obě funkce vrací ukazatel na řetězec, který představuje čas a datum ve srozumitelné (tisknutelné) podobě, obvykle např. `"Tue Sep 06 08:56:43 2016\n"`.

- **`ctime(·)`** má jako argument ukazatel na integrální časovou značku (tj. hodnotu vrácenou funkcí **`time(·)`**).
- **`asctime(·)`** přijímá ukazatel na strukturu **`tm`**, kterou vrací např. funkce **`localtime(·)`** či **`gmtime(·)`**.

  → `ctime(time(&ts));` ≡ `asctime(localtime(&ts));`
- Obvykle vrací ukazatel na **statickou oblast** → je třeba řetězec buď okamžitě vypsat nebo zkopírovat do vlastní oblasti pomocí **`strcpy(·)`** předtím, než se funkce zavolá znovu.

Datum a čas
Interval mezi časovými značkami

```
double difftime(time_t time1, time_t time2);
```

- Vrací interval mezi dvěma časovými okamžiky, danými hodnotami časových značek **`time1`** a **`time2`** (získanými např. voláním funkce **`time(·)`**).
- Návratová hodnota je v **sekundách**.

- Norma ANSI C nepředepisuje, v jakých jednotkách je uložen integrální čas v proměnných typu **`time_t`** (mohou to být sekundy, ale také nějaké blíže nespecifikované „tiky”) → použít **`difftime(·)`**, aby byla veličina zaručena.

Datum a čas
Ukázka měření kalendářního času

```
#include <stdio.h>
#include <time.h>
⋮
struct tm dat0 = {0};
time_t now, then;

dat0.tm_year = 116;
dat0.tm_mon = 0;
dat0.tm_mday = 1;

then = mktime(&dat0);
now = time(NULL);

if (then != (time_t) -1) {
  printf("Today is %s\n", ctime(&now));
  printf("%f seconds since 1.1.2016\n",
    difftime(now, then));
}
```

Datum a čas
Formátování výpisu kalendářního času

```
size_t strftime(char *str, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

- Formátuje časový údaj předaný ukazatelem na strukturu **`timeptr`** podle pravidel v řídicím řetězci **`format`** a výsledný řetězec ukládá na adresu **`str`**, což je oblast alokované paměti o velikosti **`maxsize`**.

```
time_t now;
struct tm *tinf;
char s[80] = {0};

time(&now);
tinf = localtime(&now);

strftime(s, 80, "%x - %I:%M%p", tinf);
printf("Time: %s\n", s);
```

```
Time: 09/08/16 - 11:47AM
```

Datum a čas
Formátování výpisu kalendářního času

- V řídicím řetězci funkce **`strftime(·)`** lze použít mj.:

`%a`	zkratka dne v týdnu (”Mon”)
`%A`	celé jméno dne v týdnu (”Monday”)
`%b`	zkratka jména měsíce (”Jan”)
`%B`	celé jméno měsíce (”January”)
`%c`	datum a čas dle locale
`%d`	den v měsíci (01 – 31)
`%H`	hodina ve 24-hod. formátu (00 – 23)
`%I`	hodina ve 12-hod. formátu (01 – 12)
`%j`	pořadí dne v roce (001 – 366)
`%m`	měsíc číslem (01 – 12)
`%M`	minuty (00 – 59)
`%S`	sekundy (00 – 61)
`%x`	datum dle locale
`%X`	čas dle locale
`%y`	rok bez stol. (00 – 99)
`%Y`	rok vč. stol. (např. 2016)
`%p`	určení AM/PM
`%U`	číslo týdne (00 – 53)*
`%w`	kód dne v týdnu (0 – 6)
`%W`	číslo týdne (00 – 53)*
`%z`	ozn. časového pásma
`%%`	znak %

*) pro počítání týdnů od neděle (U) nebo od pondělí (W)

Místní nastavení prostředí – locale
Obecné skutečnosti

- Funkce a datové typy pro práci s **místním nastavením prostředí** (nebo též **národním prostředím**) poskytuje knihovna **locale** → připojit hlavičkový soubor příkazem preprocesoru: `#include <locale.h>`

- **Locale** specifikuje podobu údajů, které jsou závislé na místním nastavení, jako např. **formát datumu**, **symbol místní měny**, oddělovače řádů v reálných číslech, pořadí znaků pro lexikografické porovnávání, apod.

- Norma ANSI C definuje pouze locale `"C"`, které odpovídá původní definici a nastavení jazyka C.

- Některé kombinace OS, překladačů a verzí knihovny **nemusí na nastavení locale vůbec reagovat!** (např. Win 8 + gcc)

Místní nastavení prostředí – locale
Změna/zjištění místního nastavení

```
char *setlocale(int cat, const char *locale);
```

- Je-li argument **`locale`** = **`NULL`**, vrací řetězec s hodnotou aktuálního místního nastavení. Když nelze zjistit, vrací **`NULL`**.
- Má-li argument **`locale`** hodnotu platného pojmenování místního nastavení dle norem ISO 639.1/639.2 a ISO 3166 (např. **`en_US`**, **`en_GB`**, **`de_DE`**, **`de_AT`**, **`cs_CZ`**, apod.), je toto locale nastaveno. Nelze-li nastavit, vrací **`NULL`**.

```
printf("%s\n", setlocale(LC_ALL, "en_GB"));
strftime(buffer, 80, "%c", timer);
printf("Date: %s\n", buffer);

printf("%s\n", setlocale(LC_ALL, "de_DE"));
strftime(buffer, 80,
printf("Date: %s\n",
```

```
en_GB
Date: Sun 01 May 2016 08:15:42 UTC
de_DE
Date: So 01 Mai 2016 08:15:42 UTC
```

Místní nastavení prostředí – locale
Změna/zjištění místního nastavení

- Argument **`int cat`** funkce **`setlocale(·)`** má význam kategorie locale, na kterou se dotazujeme (tj. např. nastavení oddělovačů řádů reálných čísel) a může nabývat hodnot:

`LC_ALL`	všechny níže uvedené kategorie
`LC_COLLATE`	lexikografické porovnávání řetězců
`LC_CTYPE`	klasifikace a převod znaků
`LC_MONETARY`	formátování měnových údajů
`LC_NUMERIC`	oddělovače řádů reálných čísel
`LC_TIME`	datum a čas
`LC_MESSAGES`	systémové zprávy

```
⋮
printf("Locale: %s\n", setlocale(LC_ALL, NULL));
setlocale(LC_MONETARY, "cs_CZ");
⋮
```

Místní nastavení prostředí – locale
Konkrétní podoba místního nastavení

```
struct lconv *localeconv(void);
```

- Funkce vrací ukazatel na strukturu **`lconv`**, jejíž složky udržují hodnoty konkrétních položek místního nastavení:

```
struct lconv *lc;

setlocale(LC_MONETARY, "cs_CZ");
lc = localeconv();
printf("%s\n",lc->currency_symbol);
printf("%s\n",lc->int_curr_symbol);

setlocale(LC_MONETARY, "en_US");
lc = localeconv();
printf("%s\n",lc->currency_symbol);
printf("%s\n",lc->int_curr_symbol);

printf("Decimal = %s\n", lc->decimal_point);
```

```
Kč
CZK
$
USD
Decimal = .
```

Místní nastavení prostředí – locale

Stavová struktura místního nastavení `lconv`

```
typedef struct {
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
} lconv;
```

Struktura **`lconv`**, na níž vrací ukazatel funkce **`localeconv(·)`** je **read-only!**

→ **Nelze nastavit chování aktivního locale zápisem do některé ze složek stavové struktury!**

Interakce s operačním systémem
Obecné skutečnosti

- Funkce pro spolupráci programu s operačním systémem se nachází v knihovně **stdlib** → připojit hlavičkový soubor příkazem preprocesoru: `#include <stdlib.h>`

Pozor na přenositelnost! Těsná interakce programu v jazyce C s operačním prostředím s sebou nese riziko snížení nebo úplné znemožnění přenosu na jiné platformy…

- **Funkce samotné jsou** pochopitelně na všech platformách, které podporují ANSI C, **k dispozici** (to zaručuje norma ANSI C), ale jejich **argumenty mohou mít zcela různý** (nebo také žádný) **význam**.

```
⋮
st = system("ls -la");
⋮
```

Co se asi stane při spuštění v OS Windows, které žádný příkaz `ls` nemají?

Interakce s operačním systémem
Příkazový procesor

```
int system(const char *command);
```

- Funkce předává svůj argument **command** příkazovému procesoru operačního systému, ten jej vykoná (pokud je to možné) a předá zpět návratový kód, který funkce vrací.
- V případě chyby (nikoliv chybového návratového kódu vykonávaného příkazu) vrací -1.
- Funkci lze také využít ke zjištění, zda je příkazový procesor k dispozici – při předání argumentu **command** = **NULL**.

```
if (system(NULL)) {
  #ifdef WIN32
  st = system("dir *.exe");
  #else
  st = system("find -type f -perm +111");
  #endif
}
```

Pokud už je nutné použít, pak vždy s podmíněným překladem...

Interakce s operačním systémem
Proměnné prostředí

```
char *getenv(const char *name);
```

- Funkce prohledává **proměnné prostředí** a hledá takovou, jejíž název se shoduje s argumentem **`name`** (case-sensitive, pokud je case-sensitive systém, tj. v UNIXu ano, ve Win ne). Hodnotu proměnné pak vrací jako ukazatel na **statický** řetězec znaků (tj. **neuvolňovat** po použití).
- Pokud taková proměnná neexistuje, vrací **`NULL`**.

```
printf("PATH = %s\n", getenv("PATH"));
printf("JAVA_HOME = %s\n",
  getenv("JAVA_HOME"));
```

- V POSIXu je také funkce **`setenv(·)`**, která dovoluje proměnné prostředí nastavovat, ta ale **není součástí normy ANSI C!**

Interakce s operačním systémem
Ukončení běžícího procesu

```
void abort(void);
```

- Okamžité, „špinavé” ukončení programu v místě volání. Neprovádí žádné sanační procedury.

```
void exit(int status);
```

- Ukončí proces, ale předtím vykoná ukončovací procedury:
  (1) zavolají se všechny funkce registrované via **`atexit(·)`**;
  (2) vyprázdní se vyrovnávací paměti otevřených proudů, proudy se uzavřou;
  (3) zruší se případné dočasné soubory;
  (4) předá se řízení nadřízenému procesu s informací o stavu ukončení (lze využít konstanty **`EXIT_SUCCESS`** = 0 a **`EXIT_FAILURE`** = 1, nebo předat vlastní celočíselný kód).

Interakce s operačním systémem
Registrace „úklidové” funkce

```
int atexit(void (*func)(void));
```

- Argumentem je ukazatel na funkci, která se automaticky vykoná před ukončením programu voláním funkce **`exit(·)`** nebo návratem z funkce **`main(·)`**.
- Lze zaregistrovat min. 32 funkcí (norma ANSI C), ukládají se do zásobníku → při ukončení programu se volají v opačném pořadí, než byly zaregistrovány.
- Prototyp volané funkce je **`void fname(void)`**, tj. bez argumentu, bez návratové hodnoty.
- Je-li funkce zaregistrovaná **vícekrát**, pak se také vícekrát zavolá při ukončení programu.
- Registrované funkce nesmějí pracovat s referencemi na lokální proměnné definované jinde, než v dané funkci.

Interakce s operačním systémem
Registrace „úklidové” funkce

```
...
char *str = NULL;

void cleanup() {
  if (str) free(str);
}

void main() {
  if (atexit(cleanup)) {
    printf("Error registering cleanup func!\n");
    exit(EXIT_FAILURE);
  }

  str = malloc(32);
  strcpy(str, "Hello!\n");
  printf("%s", str);
}
```

Pokud se nezdaří „úklidovou” funkci zaregistrovat, program se včas ukončí (s chybou)...

Zpracování chybových stavů
Obecné skutečnosti

- Globální proměnnou **`errno`** a konstanty chybových stavů poskytuje knihovna **errno** → připojit hlavičkový soubor příkazem preprocesoru: **`#include <errno.h>`**

- Funkce pro převod chybového kódu na řetězec, pro výpis chybového hlášení do konzole a některé další jsou definovány v knihovnách **string**, **stdlib** a **stdio**. Všechny ale využívají globální proměnnou **`errno`**, ve které je uložen číselný kód **poslední nastalé chyby**. Mnoho funkcí z knihovny ANSI C tuto proměnnou modifikuje a ukládá do ní kód případné chyby → **každá další chyba „přepisuje" tu předchozí**.

- **Chybový stav je třeba testovat okamžitě** po provedení akce, která mohla skončit chybou.

Zpracování chybových stavů
Proměnná **errno**, konstanty chybových stavů

- V knihovně **errno** je pouze deklarace (externí) proměnné **int errno** a pak definice konstant chybových stavů. Samotná realizace globální proměnné, udržující kód poslední nastalé chyby může být různá, např. v **gcc** i v **MSVC**:

```
_CRTIMP extern int *__cdecl _errno(void);
#define errno (*_errno())
```

```
#define EPERM 1
#define ENOENT 2
#define ENOFILE ENOENT
#define ESRCH 3
#define EINTR 4
⋮
#define ERANGE 34
⋮
```

- **errno** = 0 znamená, že při poslední operaci **nenastala** žádná **chyba** → před každou kritickou operací (kterou je třeba testovat) je nutné proměnnou **errno** vynulovat!

Zpracování chybových stavů
Výpis chybového hlášení o nastalé chybě

```
void perror(const char *str);
```

- Posílá do proudu **strerr** posloupnost **str** (kam může programátor uložit prefix chybové zprávy), sekvenci dvojtečka, mezera (':␣') a pak **čitelné chybové hlášení** podle kódu chyby v proměnné **errno**, a nakonec znak konce řádky ('**\n**').

```
FILE *fin = fopen("nonexistent_file", "r");
if (fin == NULL) {
  perror("Error");
  return EXIT_FAILURE;
}
```

```
Error: No such file or directory
```

POZOR: Funkce je definovaná v knihovně **stdio**, nikoliv **errno** (z té pouze využívá proměnnou **errno** ke zjištění kódu chyby).

Zpracování chybových stavů
Převod kódu chybového stavu na řetězec

```
char *strerror(int errnum);
```

- Vrací ukazatel na statickou oblast (tj. neuvolňovat!) naplněnou řetězcem s chybovým hlášením odpovídajícím kódu předanému argumentem **`int errnum`**.
- Je-li nutné s řetězcem pracovat později, je třeba zkopírovat ho do vlastní oblasti paměti – ta, na níž ukazuje vrácený ukazatel, bude přepsána při příštím volání funkce.

```
FILE *fin = fopen("nonexistent_file", "r");
if (fin == NULL) {
  printf("Error: %s.\n", strerror(errno));
  return EXIT_FAILURE;
}
```

POZOR: Funkce je definovaná v knihovně **string**, nikoliv **errno** (z té pouze využívá proměnnou **`errno`** ke zjištění kódu chyby).

Zachycení chybových stavů – signály
Obecné skutečnosti

- **Jazyk ANSI C nemá výjimkový aparát** (ten se objevil až s vývojem C++), ale poskytuje mechanismus zachycení a reakce na vznik potenciálně asynchronních událostí, tzv. ***signálů***.

- Signály jsou označeny číselnými hodnotami (na které jsou namapované konstanty s prefixem **`SIG`**-, např.: **`SIGINT`**, **`SIGSEGV`**, **`SIGABRT`**, …) a objevují se v důsledku **chyb** (dělení nulou), programátorského **záměru** (vznik signálu vyvoláním funkce **`raise(·)`**) nebo **externích událostí** (stisk spec. klávesy, např. Ctrl+Break).

- Funkce, makra a datové typy pro práci se signály se nachází v knihovně **signal** → připojit hlavičkový soubor příkazem preprocesoru: **`#include <signal.h>`**

Zachycení chybových stavů – signály
Signály, které lze ošetřit v ANSI C

- V knihovně **signal** jsou definovány tyto signály:

`SIGABRT`	abnormální ukončení programu
`SIGFPE`	chyba při počítání s reálnými čísly (÷ 0)
`SIGILL`	neplatná instrukce
`SIGINT`	požadavek na přerušení programu
`SIGSEGV`	narušení chráněné oblasti paměti
`SIGTERM`	požadavek na ukončení programu (Ctrl+C)

- Programátor může ke každému signálu definovat tzv. ***ovladač***, což je běžná funkce v jazyce C, která se automaticky vyvolá, jakmile se signál objeví.

Knihovna **signal** je součástí definice normy ANSI C!
→ na všech platformách, které podporují ANSI C, tedy musí jít definovat ovladače těchto asynchronních událostí…

Zachycení chybových stavů – signály
Instalace ovladače signálu

```
typedef void (*psigfn_t)(int);
psigfn_t signal(int signum, psigfn_t func);
```

- Instaluje **ovladač signálu** – funkci, jejíž adresa je předána v argumentu **`psigfn_t func`**. Tato funkce bude automaticky vyvolána, jakmile bude zachycen signál identifikovaný číslem **`int`** `signum`.
- Prototyp ovladače má tvar:

```
void __cdecl sig_handler(int signum);
```

- Funkce **`signal(·)`** vrací v případě úspěšné instalace ukazatel na původní ovladač daného signálu (aby ho bylo lze např. uložit a později nainstalovat zpět), **v případě selhání vrací hodnotu `SIG_ERR`** (-1).

Zachycení chybových stavů – signály
Ovladač signálu – ukázka

```
void sig_handler(int sig) {
  printf("Signal %d caught! Exiting...\n", sig);
  exit(EXIT_FAILURE);
}

int main() {
  int i = 1, j = 0, k;

  if (signal(SIGFPE, sig_handler) == SIG_ERR) {
    printf("Failed to install sig handler!\n");
    return EXIT_FAILURE;
  }

  k = i / j;

  return EXIT_SUCCESS;
}
```

Tento úsek kódu neskončí RTE, ale vyvoláním „našeho“ ovladače.

Zachycení chybových stavů – signály
Programové vyvolání signálu

```
int raise(int sig);
```

- Volání funkce způsobuje vyvolání signálu **`int sig`**.
- Bylo-li vyvolání signálu úspěšné, vrací funkce 0. V případě neúspěchu vrací nenulovou hodnotu.

- Z důvodů nepřítomnosti výjimkového aparátu v ANSI C nelze vzniku RTE aktivně předcházet, ale prostřednictvím mechanismu signálů (a také možnosti registrace ukončovacích funkcí) lze program napsat kulturně a „vybavit“ ho funkcemi pro kultivované ukončení, uložení mezivýsledků, apod.

Diagnostika a debugging
Obecné skutečnosti

- Jazyk ANSI C nenabízí příliš komfortní prostředky podpory ladění a diagnostiky programů, avšak dostupné nástroje jsou spolehlivé a účinné.

- Funkce a makra podpory ladění a diagnostiky jsou ve dvou malých knihovnách **assert** a **stddef** → připojit hlavičkový soubor příkazem preprocesoru: `#include <assert.h>`, případně `#include <stddef.h>`

- Systematické používání makra `assert(·)` výrazně zvyšuje bezpečnost výsledného kódu, navíc celkem účinně brání známým programátorským průšvihům, kdy v kódu po odladění zůstanou „diagnostické“ výpisy s různým nevhodným obsahem.

Diagnostika a debugging
Ladící makro **`assert(·)`**

```
void assert(int expression);
```

- Vyhodnocuje předaný výraz **`int expression`**. Pokud je tento výraz nenulový (tj. jeho „logická hodnota“ je **true**), **nedělá nic**.
- Pokud je hodnota výrazu po vyhodnocení rovna 0, zapíše chybové hlášení do proudu **`stderr`** a **ukončí vykonávání** programu (chybové hlášení obsahuje číslo řádky kódu, kde k ukončení došlo).

```
void main() {
  int i = 1, j = 0, k;

  assert(j != 0);
  k = i / j;

  return EXIT_SUCCESS;
}
```

```
File: assert01.c, Line 8

Expression: j != 0
```

Diagnostika a debugging
Ladící makro `assert(·)`

- Po odladění kódu **není nutné** hledat všechny výskyty makra `assert(·)` a **odstraňovat je!** → stačí nadefinovat symbol **`NDEBUG`**, např. přepínačem překladače na příkazové řádce:

```
Y:\Work\CTests>gcc assert_test.c -o assert_test -DNDEBUG
```

- Je-li definován symbol **`NDEBUG`**, makro **`assert(·)`** se rozvine na výraz, který překladač ignoruje (kód knihovny **assert**):

```
#ifdef NDEBUG
#define assert(_Expression) ((void) 0)
#else
#define assert(_Expression) \
  (void) \
  ((!!(_Expression)) || \
  (_assert(#_Expression,__FILE__,__LINE__), 0))
#endif
```

Diagnostika a debugging
Diagnostické makro `offsetof(·)`

```
size_t offsetof(TYPE, MEMBER);
```

- Makro je definované v knihovně **stddef**.
- Vrací **offset** složky **`MEMBER`** struktury **`TYPE`**, tj. počet bytů, které je třeba přičíst k bázové adrese (**`&TYPE`**), aby získaný ukazatel ukazoval na počátek příslušné složky.
- Vzhledem k ***zarovnávání*** (*Alignment*) objektů na hranice jednotek granularity paměti (obvykle DWORD, 32 bitů) nemusí offset dané složky odpovídat prostému sečtení velikostí předcházejících složek.

```
struct complex {
   double re, im;
};

int main() {
  printf("complex.im start at %d.\n",
    offsetof(struct complex, im));
⋮
```

Vysokoúrovňové funkce
Obecné skutečnosti

- Knihovna **stdlib** obsahuje 2 generické vysokoúrovňové funkce, schopné pracovat s poli prvků libovolného typu:
  (i) pro řazení dat algoritmem ***QuickSort*** – `qsort(·)` a
  (ii) pro prohledávání technikou ***bisekce*** – `bsearch(·)`.

- Obě **funkce vyžadují předání adresy „porovnávací“ funkce**, která obdrží adresy dvou prvků (jako netypové ukazatele) a vrátí výsledek jejich vzájemného porovnání jako `int`
  → tímto způsobem je zajištěna genericita (vlastní řadicí/prohledávací funkce vůbec netuší, s čím pracuje).

- **Před prohledáváním bisekcí musí být pole prohledávaných prvků seřazeno vzestupně!** – není-li, pak bisekce nefunguje.

Vysokoúrovňové funkce
Prohledávání bisekcí (půlením intervalu)

```
void *bsearch(const void *key, const void *base,
        size_t nitems, size_t size,
        int (*cmp)(const void *, const void *));
```

- Funkce prohledává bisekcí pole o **`nitems`** prvcích, přičemž první z těchto prvků leží na adrese **`base`**.
- Hledaný prvek je předán jako netypový ukazatel **`key`** na „vzor“, se kterým jsou prvky pole porovnávány (to zajišťuje porovnávací funkce).
- Velikost každého prvku pole je dána argumentem **`size`**.
- Ukazatel **`cmp`** ukazuje na porovnávací funkci, kterou je nutné nadefinovat pro daný typ dat, která jsou uložena v prohledávaném poli.
- Porovnávací funkce vrací 0, pokud jsou si její argumenty rovny (ať už to znamená cokoli). Je-li první argument menší než druhý, vrací -1; v opačném případě 1.

Vysokoúrovňové funkce
Prohledávání bisekcí – ukázka

```
int cmpfunc(const void *a, const void *b) {
  return (*(int *) a - *(int *) b);
}

int values[] = {5, 20, 29, 32, 63};

void main() {
  int *item, key = 32;

  item = (int *) bsearch(&key, values, 5,
                         sizeof(int), cmpfunc);
  if (item != NULL) {
    printf("Found item = %d.\n", *item);
  }
  else {
    printf("Item = %d not found.\n", *item);
  }
}
```

Vysokoúrovňové funkce
QuickSort

```
void *qsort(const void *base,
            size_t nitems, size_t size,
            int (*cmp)(const void *, const void *));
```

- Funkce řadí metodou **QuickSort** pole o **`nitems`** prvcích, přičemž první z těchto prvků leží na adrese **`base`**.
- Velikost každého prvku pole je dána argumentem **`size`**.
- Ukazatel **`cmp`** ukazuje na porovnávací funkci, kterou je nutné nadefinovat pro daný typ dat, která jsou uložena v řazeném poli → lze samozřejmě použít tutéž funkci, která byla připravena pro potřeby funkce **`bsearch(·)`**.

- Funkce **`qsort(·)`** a **`bsearch(·)`** se obvykle používají **spolu**, protože před prohledáváním bisekcí je nutné pole seřadit, k čemuž se velmi dobře hodí QuickSort (tím spíš, že lze využít stejnou porovnávací funkci).

Vysokoúrovňové funkce
QuickSort – ukázka

```
#define COUNT 100

struct elem { int key; int data; } table[COUNT];

int cmpfunc(const void *a, const void *b) {
  int k1 = ((struct elem *) a)->key;
  int k2 = ((struct elem *) b)->key;
  return (k1 < k2) ? -1 : (k1 > k2) ? 1 : 0;
}

void main() {
  int i;

  for (i = 0; i < COUNT; i++) { … }

  qsort((void *) table, (size_t) COUNT,
    sizeof(struct elem), cmpfunc);
}
```

naplnění pole nějakými daty

Funkce s proměnným počtem argumentů
Obecné skutečnosti

- Jazyk ANSI C umožňuje definovat **funkce s proměnným** (neznámým) **počtem argumentů**. Datové typy a makra k tomu poskytuje knihovna **stdarg** → připojit hlavičkový soubor příkazem preprocesoru: `#include <stdarg.h>`

- Syntaktická podoba definice funkce s proměnným počtem argumentů vypadá takto:

```
void varargs(char *argtypes, ...) { …
```

- Makra `va_start(·)`, `va_arg(·)` a `va_end(·)` zajišťují extrakci nedeklarovaných argumentů ze seznamu argumentů funkce; stavová proměnná, která udržuje informace o seznamu argumentů, je typu `va_list`.

konstrukce '. . .' (3 tečky) nahrazuje deklaraci neznámého množství argumentů

Funkce s proměnným počtem argumentů
Počátek zpracování argumentů

```
void va_start(va_list args, last_arg);
```

- Makro **inicializuje** mechanismus extrakce neurčeného počtu argumentů funkce ze seznamu. Stavová proměnná **`args`** udržuje informace potřebné pro makro **`va_arg(·)`**, které provádí samotnou extrakci.
- Druhý argument **`last_arg`** je extrémně důležitý: Říká totiž, který z argumentů funkce s proměnným počtem argumentů je poslední deklarovaný („pevný"), tj. po kterém již následují nedeklarované argumenty.

POZOR! Je nezbytně nutné si do definované funkce s proměnným počtem argumentů **předat některým z deklarovaných argumentů počet** (a případně typ) **nedeklarovaných** argumentů → jinak je nebude možné extrahovat!

Funkce s proměnným počtem argumentů
Extrakce nedeklarovaných argumentů

```
type va_arg(va_list args, type);
```

- Makro **vyjímá ze seznamu argumentů** následující argument typu **`type`** a vrací jeho hodnotu. Typem může být kterýkoli datový typ jazyka ANSI C (včetně korektně definovaných uživatelských typů).
- Argument **`args`** je stavová proměnná, zinicializovaná voláním makra **`va_start(·)`**.

```
void va_end(va_list args);
```

- Makro deaktivuje mechanismus extrakce nedeklarovaných argumentů.
- **POZOR:** Pokud toto makro není zavoláno (a přitom bylo dříve voláno makro **`va_start()`**), funkce nemůže korektně doběhnout (návrat. hodnota není definována).

Funkce s proměnným počtem argumentů
Ukázka 1

```
void printargs(char *argtype, ...) {
   va_list arguments;
   int arg_int;
   char *arg_str, thisarg;

   va_start(arguments, argtype);
   while ((thisarg = *argtype++) != '\0') {
      switch (thisarg) {
      case 'i': arg_int = va_arg(arguments, int);
                printf("%d\n", arg_int);
                break;
      case 's': arg_str = va_arg(arguments, char *);
                printf("%s\n", arg_str);
      }
   }
   va_end(arguments);
}
```

poslední „pevný" argument před proměnnou částí

```
void main() {
   printargs("isis",
      5, "abc", 7, "def");
}
```

int string int string

Funkce s proměnným počtem argumentů
Ukázka 2

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

int sum(int num_args, ...) {
  va_list ap;
  int i, val = 0;

  va_start(ap, num_args);
  for (i = 0; i < num_args; i++) {
    val += va_arg(ap, int);
  }
  va_end(ap);

  return val;
}
```

```
void main() {
  printf("10 + 20 + 30 = %d\n",
    sum(3, 10, 20, 30));
  printf("1 + 2 + 3 + 4 = %d\n",
    sum(4, 1, 2, 3, 4));
}
```