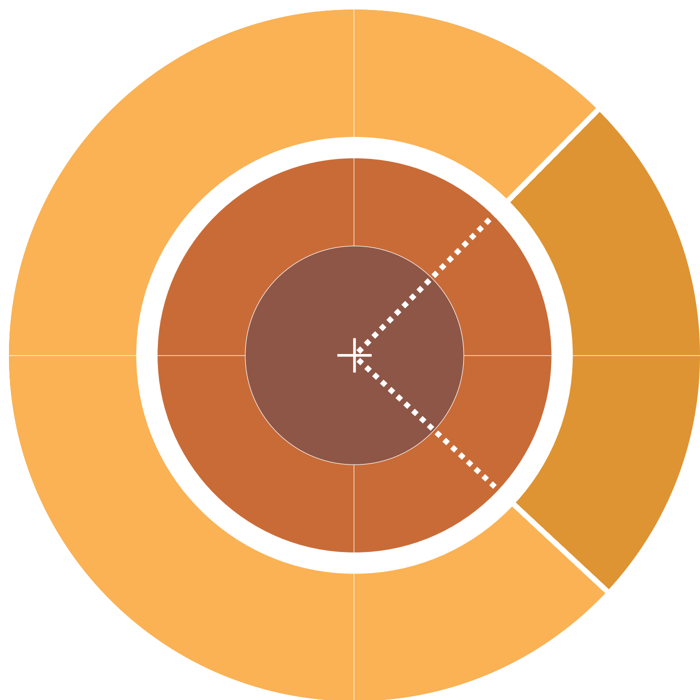
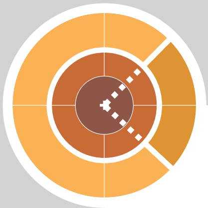
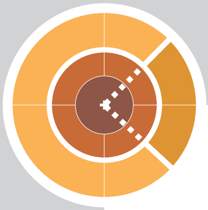




Datum a čas

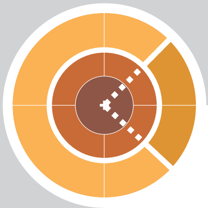
- funkce pro práci s (kalendářním) **datem**, systémovým **časem** a **časem** běžícího **procesu** poskytuje knihovna `time` => připojit pomocí `#include <time.h>`
- kalendářní čas je reprezentován buď jako **aritmetická hodnota** (obvykle počet sekund od 1.1.1980) nebo jako struktura s definovanými položkami (den, měsíc, rok, ...)
- funkce `clock()` vrací čas běžícího procesu (integrální, obvykle v mikrosekundách) - je tedy **vhodná k měření doby běhu programu**



Měření doby běhu programu

```
clock_t clock();
```

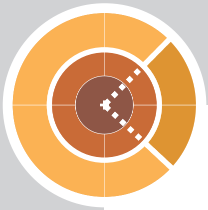
- funkce vrací (přibližnou) hodnotu **času procesoru**, který používá běžící proces
- čas je vyjádřen **v jednotkách**, které se liší podle implementace (tikách) - většinou jsou to mikrosekundy
- **POZOR!** na přetečení: je-li `clock_t` např. 32-bitový `long` (což je celkem obvyklé), přeteče hodnota přibližně po 36 minutách
- počet časových jednotek (tiků) za sekundu je definován makrem `CLOCKS_PER_SEC`
- není-li čas procesoru k dispozici (z důvodů specifických pro OS), vrací funkce `clock()` -1



Měření doby běhu programu (příklad)

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

int main() {
    clock_t t1, t2;

    t1 = clock();
    process();
    t2 = clock();

    printf("Proces pracoval %f sekund",
        ((double) (t2 - t1)) / CLOCKS_PER_SEC);

    return 0;
}
```

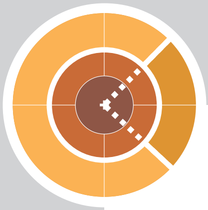
ANSI C dovoluje přetypování na libovolný vhodný číselný typ



Kalendářní čas - time()

```
time_t time(time_t *tptr);
```

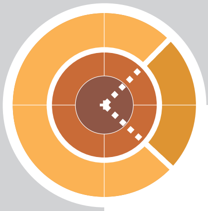
- funkce vrací kalendářní čas kódovaný jako celé číslo typu **long** (obvykle)
- je-li ukazatel **tptr** různý od **NULL**, uloží se vrácená hodnota také na adresu, kam ukazuje **tptr**
- nastane-li chyba, vrací -1
- ANSI C dovoluje, aby **time_t** byl libovolného aritmetického typu
- typicky se vrácená hodnota předává do funkce **ctime()** nebo **asctime()**, které ji převedou do čitelné podoby
- výpočet intervalu mezi dvěma kalendářními časy se provádí funkcí **difftime()**



Převod času na řetězec - ctime() a asctime()

```
char *ctime(const time_t *tptr);
char *asctime(const struct tm *ts);
```

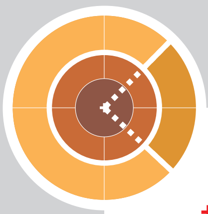
- obě funkce vrací ukazatel na řetězec, který se skládá z času a data v tisknutelné **srozumitelné podobě**, např.
  `Wed Nov 30 08:52:21 2005\n\0`
- `asctime()` má jako parametr ukazatel na strukturu, kterou tvoří `localtime()` nebo `gmtime()` z aritmetického času, který vrací `time()`; tedy
  `ctime(tp)` je shodné s `asctime(localtime(tp))`
- ve většině implementací vrací ukazatel na **statickou** oblast, je tedy nutné buď řetězec okamžitě vypsat nebo přesunout pomocí `strcpy()` předtím, než se funkce zavolá znovu



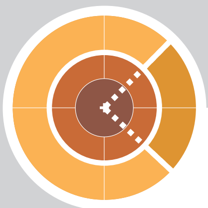
Převod času do struktury - gmtime(), localtime() a mktime()

```
struct tm *gmtime(const time_t *tptr);
struct tm *localtime(const time_t *tptr);
time_t mktime(struct tm *tmptr);
```

- **gmtime()** a **localtime()** převádí číselnou hodnotu času na strukturovaný formát typu **struct tm**
- **gmtime()** převádí na Greenwich Mean Time
- **localtime()** převádí na místní čas, podle časového pásma a případného letního času
- při chybě vracejí **NULL**
- oblast, na kterou ukazuje návratová hodnota je typicky **statická**, přepisuje se při dalším volání, tj. je třeba ji po volání fce někam zkopírovat
- **mktime()** tvoří ze "strukturovaného" času číselnou hodnotu typu **time_t** (pokud nelze převést, vrací -1)



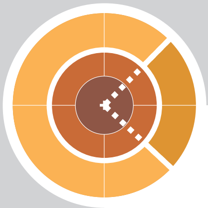

```
struct  tm {
   /* seconds after the minute -- [0,61] */
   int  tm_sec;
   /* minutes after the hour   -- [0,59] */
   int  tm_min;
   /* hours after midnight     -- [0,23] */
   int  tm_hour;
   /* day of the month         -- [1,31] */
   int  tm_mday;
   /* months since January     -- [0,11] */
   int  tm_mon;
   /* years since 1900                   */
   int  tm_year;
   /* days since Sunday        -- [0,6]  */
   int  tm_wday;
   /* days since January 1     -- [0,365]*/
   int  tm_yday;
   /* Daylight Savings Time flag */
   int  tm_isdst;
};
```



Rozdíl časů - difftime()

```
double difftime(time_t t1, time_t t0);
```

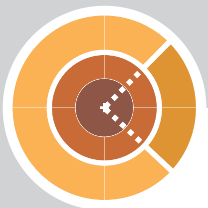
- **`difftime()`** existuje pouze v ANSI C
- vrací **rozdíl mezi časovými okamžiky** danými hodnotami **`t1`** a **`t0`** (získanými např. **`time()`**)
- výsledek je sekundách jako typ **`double`**
- programátor se nemůže spolehnout na to, že v **`time_t`** je čas v mikrosekundách => použít **`difftime()`**



Příklad na měření času

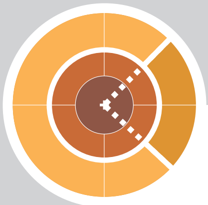
```
#include <time.h>
...
struct tm dat0 = {0};
time_t now, then;

dat0.tm_year = 90;
dat0.tm_month = 0;
dat0.tm_mday = 1;

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

if (then != (time_t) -1) {
   printf("Dnes je %s\n", ctime(&now));
   printf("Od 1.1.1990 uplynulo %f sekund\n",
      difftime(now, then));
}
```

všechny položky struktury nastavit na 0



Řídicí funkce a diagnostika

- dále uvedené funkce poskytují další možnosti řízení výpočtu v programech napsaných v jazyce ANSI C
- většina se nachází v knihovně **`stdlib`** - u těch, které jsou v jiných knihovnách, je na to výslovně upozorněno
- některé z těchto funkcí představují částečnou náhradu výjimkového aparátu (**`atexit()`**, **`signal()`**, atd.)



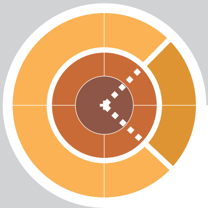
Ladicí makro assert()

```
#include <assert.h>
#ifndef NDEBUG
void assert(int expression);
#else
#define assert(x) ((void) 0)
#endif
```

```
#include <assert.h>

int f(int x) {
    assert(x > 0 && x < 10);
    ...
}
```

není-li podmínka za běhu programu splněna, program se ukončí s výpisem souboru a řádky, kde je assert() umístěn

- po odladění vyhodit (NDEBUG)



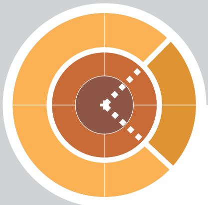
Volání příkazového procesoru - system()

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

- parametr **`command`** se předá příkazovému procesoru OS (tj. např. **`cmd`** ve Win32 nebo **`sh`** v UNIXu)
- vrací stav provedeného příkazu
- lze volat s parametrem **`NULL`**, pak vrací 0, když v OS **neexistuje příkazový procesor** nebo nenulovou hodnotu, když příkazový procesor existuje

```
int main() {
   int retv;
   retv = system("dir");
   printf("Ret code %d\n", retv);
   return 0;
}
```

vrátí 0, pokud se **`dir`** úspěšně provedl



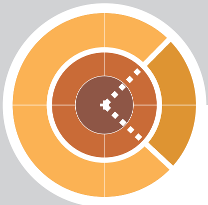
Ukončení programu - exit() a abort()

```
#define EXIT_SUCCESS
#define EXIT_FAILURE
void exit(int status);
void abort(void);
```

hodnota parametru status v případě normálního ukončení

v případě chyby

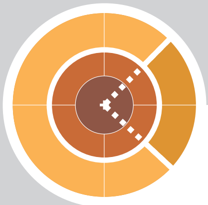
- ukončují **okamžitě** činnost programu, za normálních okolností se řízení z těchto funkcí nevrací
- `exit()` provádí ukončovací akce:
  (1) zavolají se všechny funkce, registrované pomocí `atexit()`
  (2) otevřené výstupní proudy se vyprázdní, všechny proudy se uzavřou
  (3) zruší se případné dočasné soubory
  (4) řízení se předá nadřízenému prostředí s informací o stavu ukončení (normál, havárie)
- `abort()` končí "nenormálně", ihned, neprovádí úklid



Registrace ukončovací funkce - atexit()

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

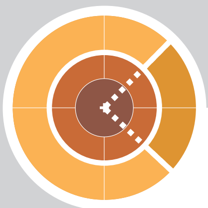
- registruje funkci, která se bude volat při ukončení programu funkcí `exit()` nebo návratem z funkce `main()`
- registrovaná funkce se nevolá při nenormálním ukončení
- ANSI norma předepisuje, že lze registrovat min. 32 funkcí
- registrované funkce se volají **v opačném pořadí**, než byly zaregistrovány
- každá funkce se volá bez parametrů a nic nevrací
- je-li funkce registrovaná **vícekrát**, pak se právě tolikrát zavolá při ukončení programu
- registrované funkce se nesmí odkazovat na vnější objekty, které mají paměťovou třídu `auto` nebo `register`



Registrace ukončovací funkce - atexit() (příklad)

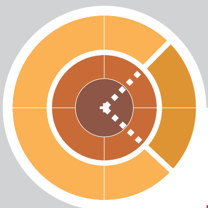
```
#include <stdlib.h>

FILE *finp;

void cleanup() {
    if (finp != NULL) fclose(finp);
}

int main() {
    int status;

    status = atexit(cleanup);
    assert(status == 0);

    finp = fopen("inp.dat", "r");
    …
}
```

nepodaří-li se ukončovací funkci zaregistrovat, ukončí se program (včas)



Signály

- mechanismus ošetření výjimečných stavů, potřebné funkce a konstanty jsou v knihovně **`signal`**, tj. připojit pomocí **`#include <signal.h>`**
- signály jsou potenciálně **asynchronní události** označené celočíselnými hodnotami (lze identifikovat pomocí konstant z knihovny **`signal`**: SIGINT, SIGABRT, ...
- signály se objevují v důsledku činnosti **mechanismu detekce chyb**, **uživatelských akcí** (vyvoláním signálu funkcí **`raise()`**) nebo externích událostí (stisk spec. klávesy)
- ke každému signálu může program definovat ***ovladač***, což je funkce, která se volá v případě, že se signál objevil - očekává se, že provede jisté sanační akce a vrátí řízení
- **ovladač signálu** je běžná funkce, která má jeden parametr, číslo signálu, a nevrací hodnotu




```c
#include <signal.h>

FILE *finp = NULL;

void sig_handler(int sigid) {
    if (finp != NULL) fclose(finp);
    exit(EXIT_FAILURE);
}

int main() {
    int i = 0, j = 1, k;
    if (signal(SIGFPE, &sig_handler) == SIGERR)
        exit(EXIT_FAILURE);
    finp = fopen("inp.dat", "r");
    k = j / i;
    fclose(finp);
    return EXIT_SUCCESS;
}
```