IJC — Jazyk C

Petr Peringer
peringer AT fit.vutbr.cz

Vysoké učení technické v Brně
Fakulta informačních technologií,
Božetěchova 2, 612 66 Brno

(Verze: 2024-02)
Přeloženo: 5. února 2024

Úvod

Tyto slajdy jsou určeny pro studenty předmětu IJC na FIT. Obsahují popis jazyka ISO C vhodný pro studenty, kteří již zvládli triviální základy programování v C. Obsah slajdů je velmi stručný, podrobnější informace jsou součástí výkladu. *Předpokládá se též samostatná práce studentů s literaturou.*

Programy jsou sázeny obvykle bez potřebných `#include` na šířku maximálně 52 znaků:

```
---------|---------|---------|---------|---------|--
text programu a poznámky
```

Pro úsporu místa jsou někdy formátovány nestandardně.
V textu jsou občas použity rámečky pro zvýraznění.

Pravidla

IJC =

- přednášky
- úkoly
- konzultace
- hodnocení celkem 100 bodů:
  - 30b 2 úkoly
  - 70b zkouška

Poznámka: Je dobré číst odbornou literaturu.

Zdroje informací

- Oficiální stránka:
  `https://www.fit.vut.cz/study/course/IJC/`
- Aktuální informace pro studenty:
  `https://www.fit.vut.cz/study/course/IJC/public/`
- Dobrá literatura
- WWW odkazy
- `man 3`
- Nápověda v integrovaném prostředí
- ...

Literatura


Herout, P.: *Učebnice jazyka C*, 6. vydání, Kopp, 2010


ISO: Programming languages – C, WG14 N3896 Committee Draft, April 1, 2023 (C23)


Kerninghan, B.; Ritchie, D.: *The C Programming Language*, 2nd edition, Addison-Wesley, 1988


Kerninghan, B.; Ritchie, D.: *Programovací jazyk C*, Computer Press, 2006


Kernighan B., Pike R.: *The Practice of Programming*, Addison-Wesley, 1999

Historie jazyka C

C	(1972)	K&R
ANSI C	(1989)	norma pro USA
ISO C90	(1990)	mezinárodní norma
ISO C90 + TC	(1994)	doplňky a opravy
ISO C99	(1999)	mezinárodní norma
ISO C99 + TC	(2004)	doplňky a opravy
ISO C11	(2011)	mezinárodní norma
ISO C17	(2018)	= platný standard
ISO C23	(2024)	bude nový standard

Kerninghan, Ritchie: The C Programming Language (1978, 1988).

Aktuální norma: `ISO/IEC 9899:2018` (C17)
Překladače a vývojová prostředí pro jazyk C — viz WWW

Novinky v C11 (C17 jen opravuje chyby)

- některé části normy jsou nepovinné (”optional”)
- podpora vláken (”threads”)
- možnost specifikovat zarovnávání (”alignment”)
- UNICODE znaky a řetězce (bylo v revizi 2004)
- typově generické výrazy
- `static_assert`
- anonymní struktury a unie
- funkce bez návratu (atribut `noreturn`)
- výlučný přístup k souborům
- doplnění charakteristik pro float/double/...
- volitelná podpora pro kontrolu mezí a analýzu
- likvidace funkce `gets` (nahrazeno `gets_s`)

Novinky v C23

- změna: `true`, `false`, `bool`, ... jsou klíčová slova.
- podpora `nullptr`
- podpora `auto` (jen pro inferenci typů proměnných)
- operátor `typeof(expr)`
- volitelně `_Decimal64` atd.
- inicializace nulami `{}` (včetně VLA)
- `#embed`, `#warning`, `__has_include`
- změna: `void f()` je funkce bez parametrů (jako v C++)
- povinné *variably-modified types* (parametry fcí, ne VLA)
- `enum E : long { .... }`
- literály se separátory `1.123'456'789`
- binární literály `0b10101010`, `%b` formát
- speciální celočíselné typy `_BitInt(N)`
- `constexpr` (jen pro proměnné)
- nová syntaxe pro atributy: `[[noreturn]]`
- nové knihovní funkce, ...

Charakteristika jazyka C

- Obecně využitelný programovací jazyk tradičně používaný pro systémové programování.
- C je relativně jednoduchý jazyk 'nižší úrovně'.
- Otevřený jazyk; malé jádro je snadno rozšiřitelné (knihovny).
- *Dobře napsané* programy v C jsou efektivní.
- Existuje velké množství programů napsaných v C.
- Programy jsou přenositelné *při dodržování jistých pravidel*.
- Existují překladače na prakticky všechny platformy.
- Populární díky operačnímu systému UNIX.
- Je standardizovaný (ISO/ANSI).

Nevýhody jazyka C

- Nedisciplinovanému uživateli umožňuje psát zcela nesrozumitelné programy (`i["abcdef"]`, viz IOCCC)
- Je nutné dodržovat jisté konvence nekontrolované překladačem (ale existují prostředky na dodatečné kontroly — například program `lint`):
  - parametry funkce `printf`
  - meze polí
  - konzistence při sestavování programu
  - nedefinované chování: int overflow, ...
  - ...
- Manuální správa paměti. (Ne vždy jde o nevýhodu. Existuje možnost použít *"garbage collector"* – viz např. `libgc`)
- ...

Doporučený styl psaní programů

```
if ( a > b ) {                          /* varianta A */
   text odsazený o 2-8 znaků
}
```

```
if ( a > b )                            /* varianta B */
{
   text odsazený o 2-8 znaků
}
```

Zarovnávání textu programu je důležité pro jeho dobrou čitelnost. Překladače zarovnání nevyžadují ani nekontrolují.

Poznámky:

- Automatické formátování zdrojových textů (GNU `indent`)
- Automatické generování dokumentace ze zdrojových textů programů (např. `doxygen`).

První program

Soubor `ahoj.c`

```
#include <stdio.h>          /* vloží rozhraní */

int main(void)  {
  printf("Ahoj!\n");
}
```

Způsob zpracování

```
cc ahoj.c              # pro Linux, UNIX, GNU C
./a.out
```

Poznámka: Struktura programu v jazyku C

Základní datové typy

`void`	prázdná množina hodnot
`char`	znak
`short`	krátké celé číslo
`int`	celé číslo (+-podle registru procesoru)
`long`	dlouhé celé číslo (min 32 bitů)
`long long`	velmi dlouhé celé číslo (min 64 bitů)
`float`	reálné číslo
`double`	reálné číslo s dvojnásobnou přesností

Implicitní typové konverze

`int` $\Longleftrightarrow$ `float`

`char` $\Longleftrightarrow$ `int`

Řetězce

Řetězcové literály (C11)

```
"řetězec může obsahovat speciální znaky (char[])"
u8"text UTF-8 (char[])"
u"text UTF-16 (char16_t[])"
U"text UTF-32 (char32_t[])"
L"text (wchar_t[])"

\\   = znak \
\"   = znak "
\t   = tabelátor (posun na následující sloupec N*8)
...
```

Poznámky:

Pozor na jména souborů v MS-DOS stylu (`"c:\text.txt"`)

Pozor na další omezení – např. `printf("100%%");`

Funkce `printf`

```
printf("formátovací řetězec" [, parametry]);
```

- Knihovní funkce, není součástí jazyka
- Formátovací řetězec popisuje způsob tisku parametrů, znak % má speciální význam – označuje začátek formátu

Základní formáty

```
%f      číslo v plovoucí čárce
%d      desítkové celé číslo
%o      oktalové celé číslo
%x      šestnáctkové celé číslo
%c      znak
%s      řetězec znaků
%%      znak %
```

Příkaz `for`

```
for(počáteční_nastavení; test; nastav_další_krok)
```

```
for(;;) { .... }   /* nekonečný cyklus */
```

Příklad:

```
int main(void)  {
  int i;
  for(i=1; i<=10; i++)
    printf(" %f \n", 1.0/i);
}
```

Symbolické konstanty

Makra:

```
#define  JMENO  hodnota
#define  ZACATEK 1
```

`const` nebo `enum` není zcela ekvivalentní, ale obecně lepší:

```
const int ZACATEK = 1;
enum { ZACATEK = 1 };
```

Standardní vstup - příklad

počítání znaků

```
int main(void) {
  long nc = 0;    /* zaručený rozsah do 2e9 */
  while(getchar()!=EOF)      /* čtení znaku */
    nc++;
  printf("%ld znaků \n", nc);
}       /*  ^------------ long */
```

druhá verze

```
int main(void) {
   long nc;
   for( nc = 0; getchar()!=EOF; nc++ )
     ;    // prázdný příkaz - vše je v záhlaví
   printf("%ld znaků \n", nc);
}
```

Příklad: kopírování souboru

Standardní vstup → standardní výstup

```
#include <stdio.h>

int main(void) {
   int c;        // POZOR na chybné použití char c;
   while ( (c = getchar()) != EOF )  // priorita =
      putchar(c);
}
```

Příklad: počítání slov

```
int main(void) {
  int c;
  long nl=0, nw=0, nc=0; // počet řádků, slov a znaků
  bool inword = false;   // stav automatu
  while( (c = getchar()) != EOF ) {
    nc++;
    if( c == '\n' ) nl++;          // nový řádek
    if( c == ' ' || c == '\n' || c == '\t' )
      inword = false;              // oddělovač
    else if( !inword ) {
      inword = true;               // začíná slovo
      nw++;
    }
  }
  printf("%ld %ld %ld\n", nl, nw, nc);
}
```

Stavové automaty (*Finite-state machines*)

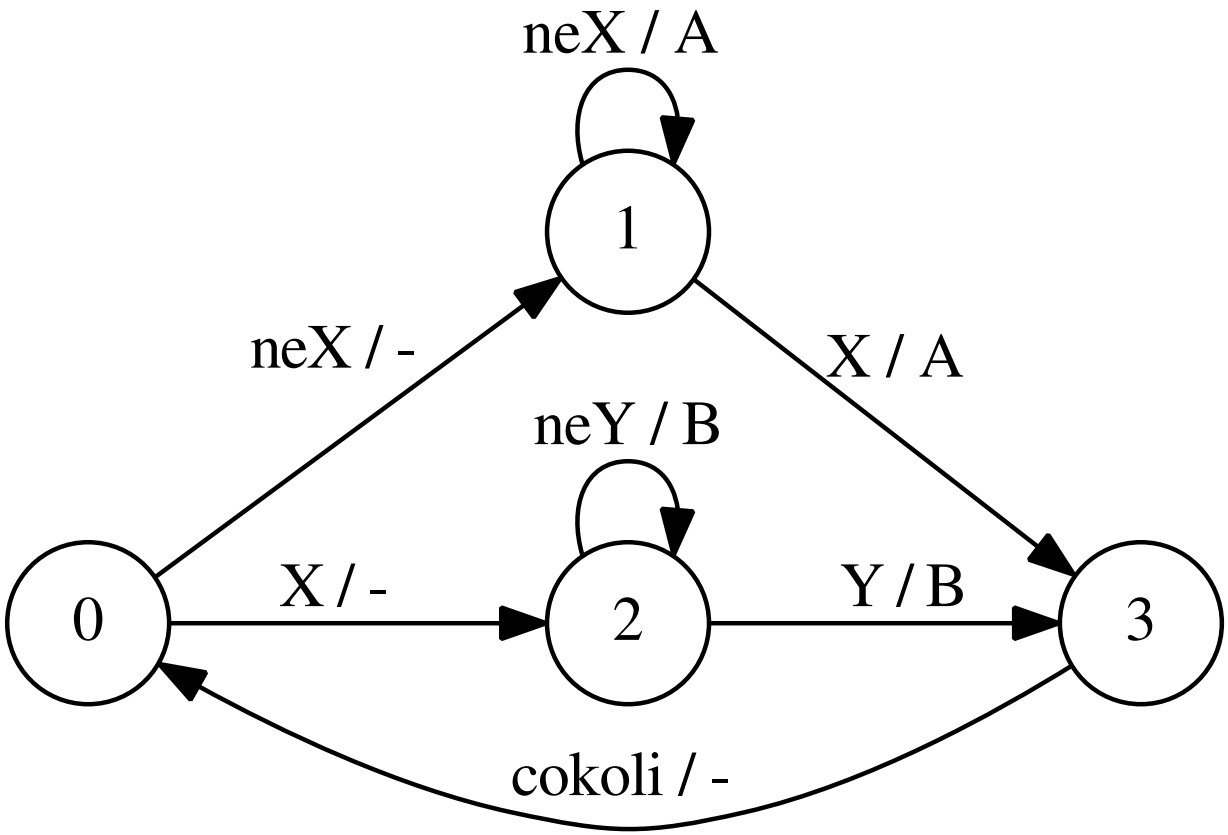
- Stavy: konečný počet, počáteční stav, koncové stavy
- Vstupní abeceda
- Výstupní abeceda
- Přechodová funkce (hrany v grafové reprezentaci)
- Výstupní funkce (Mealy/Moore)

Implementace: `switch(stav)`, tabulka+interpret, ...

Snadno modifikovatelné, čitelnější kód

Použití: regulární výrazy, zpracování textu, HW, řízení, ...

Příklad: automat1 — grafová reprezenatce

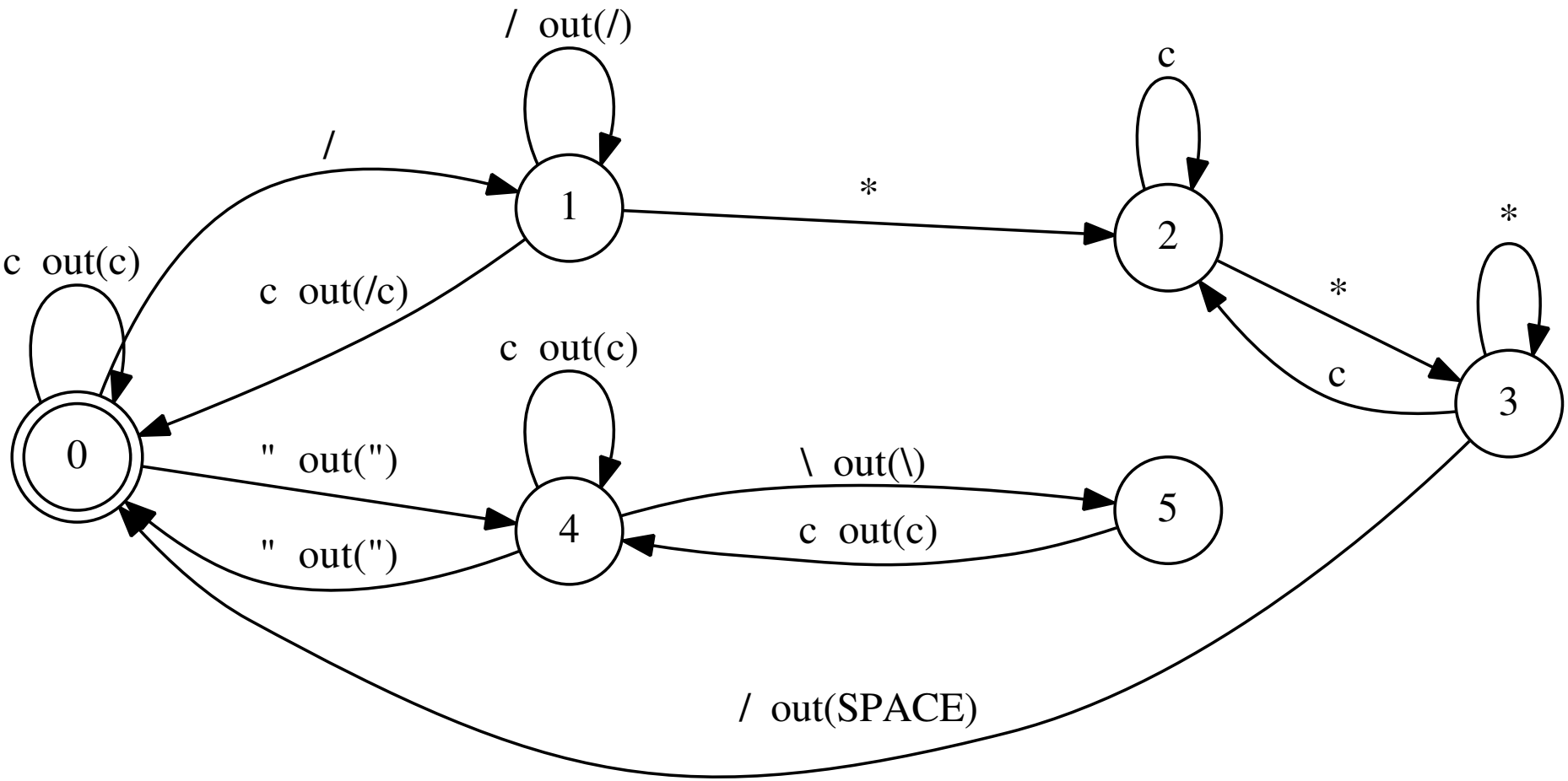


Příklad: automat1 — typická implementace

```
int stav = 0;   // počáteční stav
T vstup;
while ((vstup=DALSI_SYMBOL()) != KONEC) {
  switch(stav) {
    case 0: if (vstup==X) stav=2;
            else          stav=1;
            break;
    case 1: if (vstup==X) stav=3;
            Vystup(A);
            break;
    case 2: if (vstup==Y) stav=3;
            Vystup(B);
            break;
    case 3: stav=0; break;
  }
} // end while
```

Příklad: automat2 pro vynechání poznámek

Graf popisující automat na odstranění `/* ... */` poznámek



Příklad: automat2 — kód

```
int stav = 0;   // počáteční stav
int c;
while ((c=getchar()) != EOF) {
  switch(stav) {
    case 0: if(c=='/')  stav=1;
            else if (c=='"') { stav=4; putchar(c); }
            else putchar(c);
            break;
    case 1: if(c=='*')  stav=2;
            else if(c=='/') putchar(c);
            else { stav=0; putchar('/'); putchar(c); }
            break;
    case 2: if(c=='*') stav=3;
            break;
```

Příklad: automat2 — dokončení

```
    case 3: if(c=='/') { stav=0; putchar(' '); }
            else if(c!='*') stav=2;
            break;
    case 4: if(c=='\\') stav=5;
            else if(c=='"') stav=0;
            putchar(c);
            break;
    case 5: stav=4;
            putchar(c);
            break;
  } // end switch
} // end while
if(stav!=0) fprintf(stderr, "Error\n");
```

Poznámka: Chybí zpracování `'"'`, `'\''` a komentářů `//`

Příklad: histogram počtu číslic

```
int main(void)
{
   int c;                    // načtený znak nebo EOF
   int ndigit[10] = { 0, };
   while( (c=getchar()) != EOF )
      if(isdigit(c))                 // je číslice?
         ndigit[c-'0']++;            // znak --> číslo
   printf("digit: number\n");
   for(int i=0; i<10; i++)
     printf("'%c': %d\n", i+'0', ndigit[i]);
   printf("\n");
}
```

Definice funkce

ISO C

```
int power(int x, int n) {          /* umocňování */
  int p;
  for( p=1; n>0; n-- )
    p = p * x;
  return p;                  /* návrat s hodnotou p */
}
```

Volání funkce

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Poznámky:

Předávání parametrů hodnotou (kromě pole)

Pořadí vyhodnocování argumentů nedefinováno

Prototyp funkce

K&R definice funkce

Podle K&R (velmi zastaralé, zrušeno v C23, nepoužívat)

```
power(x, n)                /* K&R: implicitní typ int */
int x, n;                  /* K&R: deklarace parametrů */
{
   int i, p;
   for( i=p=1; i<=n; i++ )
      p = p * x;
   return p;
}
```

Poznámky:

K&R nezná pojem *prototyp funkce*

K&R deklarace funkcí nutná jen pro typ různý od int

Příklad: nalezení nejdelšího řádku v textu ...

```
unsigned get_line(const unsigned lim, char s[lim]) {
    unsigned i = 0;
    int c;
    while ((c = getchar()) != EOF) {
        s[i++] = c;      // uložit znaky včetně '\n'
        if (c == '\n')     break; // konec řádku
        if (i >= lim - 1) break; // nevejde se celý
    }
    s[i] = '\0';         // řetězec končí znakem '\0'
    return i;
}

void copy(char s[], const char s1[]) { // s:=s1
  while( (*s++ = *s1++) != '\0' );
}
```

Příklad: ... pokračování

```
#define MAXLINE 1000    // implementační limit

int main(void) {
  char line[MAXLINE];   // načtený řádek
  char save[MAXLINE];   // nejdelší řádek a
  unsigned max = 0;     // jeho délka
  unsigned len;
  while( (len=get_line(MAXLINE,line)) > 0 )
    if( len > max ) {   // je delší?
      max = len;
      copy(save, line); // pole předá odkazem
    }
  if( max > 0 )
    printf("%s", save);
}
```

Jazyk C – systematická definice

Základní množina znaků (1 byte)

- písmena: `a-zA-Z`
- číslice: `0-9`
- grafické znaky:

```
!   &   *   .   <   [   ^   |
"   '   +   /   =   \   _   }
#   (   ,   :   >   ]   {   ~
%   )   -   ;   ?
```

- mezera, nový řádek, BS, HT, VT, FF

V identifikátorech, znakových/řetězcových literálech, jménech hlavičkových souborů a poznámkách mohou být i jiné znaky.

Poznámky (comments)

```
/* text C poznámky */

// text C99+ poznámky do konce řádku
// poznámka s trigraph ??/
   pokračuje i na dalším řádku
```

ISO C nedovoluje vnořené poznámky. (Použijte `#if 0`)
Poznámka je *přepsána na jednu mezeru* až po rozvoji maker

Poznámka:
Některé zastaralé (ne ISO C) implementace nevkládaly mezeru

Operátory a separátory

```
!     &=    +     --    /=    <<    ==    >>=   ^     |=
%     (     ++    -=    :     <<=   >     ?     ^=    ||
%=    )     +=    ->    ;     <=    >=    [     {     }
&     *     ,     .     <     =     >>    ]     |     ~
&&    *=    -     /
```

Poznámka:
Operátor % a záporná čísla:

- C89: "*implementation defined*",
- C99, C11: LIA (Language Independent Arithmetic)
- Poznámka: pro znaménkový typ zbytek $\neq$ modulo

Identifikátory

- rozlišují se velká a malá písmena (*case sensitive*)
- minimální rozlišovaná délka 63 znaků (31 pro extern)
- konvence pro zápis identifikátorů:
  - identifikátory preprocesoru velkými písmeny
    `#define YES 1`
  - ostatní identifikátory malými písmeny
    `int i;`
  - dlouhé identifikátory s podtržítkem mezi slovy
    `int muj_dlouhy_identifikator;`
  - speciální jména začínají znakem _ (`__func__`)
- Zvláštní prostory jmen pro:
  - návěští (”labels”),
  - označení struktur/unií/výčtů (”tags”),
  - členy struktur/unií.

Klíčová slova (C11)

```
auto        else        long        switch
break       enum        register    typedef
case        extern      restrict    union
char        float       return      unsigned
const       for         short       void
continue    goto        signed      volatile
default     if          sizeof      while
do          inline      static
double      int         struct

_Alignas  _Atomic  _Complex  _Imaginary  _Static_assert
_Alignof  _Bool    _Generic  _Noreturn   _Thread_local
```

Jednoduché typy a velikost údajů

`void`	nezabírá paměť
`bool`	pouze hodnoty `true` a `false`
`char`	znak (vždy 1 bajt, celočíselný typ)
`short int`	krátké celé číslo (min 16 bitů)
`int`	standardní celé číslo
`long int`	celé číslo, min 32 bitů
`long long int`	celé číslo, min 64 bitů
`float`	reálné číslo (malá přesnost)
`double`	reálné číslo, dvojnásobná přesnost
`long double`	reálné číslo, extra přesnost (např. 10B)
`float complex`	komplexní číslo, přesnost float
`double complex`	komplexní číslo, přesnost double

Poznámka: C23: `_Float32`, ..., `_Decimal128`

Typy a velikost údajů

`|char| <= |short| <= |int| <= |long| <= |long long|`

Příklady `sizeof(T)` pro různé architektury

	PC/64bit	PC/32bit	PC/16bit
`char`	1	1	1
`short`	2	2	2
`int`	4 nebo 8	4	2
`long`	8	4	4
`long long`	8	8	(8)
`float`	4	4	4
`double`	8	8	8
`long double`	16	12	10
`void*`	8	4	2 nebo 4

Poznámka: LP64, ILP64, LLP64, ILP32, LP32

signed, unsigned (celočíselné typy)

```
unsigned char          signed char
unsigned               unsigned int
unsigned long
```

const, volatile (*type qualifiers*)

```
const T
```

Objekt takto označený nelze modifikovat přiřazením.

```
volatile T
```

Tato specifikace zabrání překladači provádět optimalizace při přístupu k objektu. Významné zvláště při komunikaci mezi paralelními procesy (např. sdílená paměť).

Poznámky: *"const-correctness"*, volatile $\neq$ atomic

Literály: celá čísla

Celočíselné literály

- desítkové: `číslice1-9 číslice0-9`
- osmičkové: `0 číslice0-7`
- šestnáctkové: `0x číslice0-9, a-f, A-F`
- binární (C23): `0b číslice0-1`

přípona	typ
bez přípony	`int`
`l` nebo `L`	`long`
`ll` nebo `LL`	`long long`
`u` nebo `U`	+specifikace `unsigned`
`wb` nebo `WB`	`_BitInt(N)` $\quad$ `N` $= \mathit{min}$ (C23)

Příklady:

```
1'234567'890123'456789LL,  077,  0xFFuL,  0b1010uWB
```

Literály v plovoucí řádové čárce

přípona	typ
bez přípony	`double`
`l` nebo `L`	`long double`
`f` nebo `F`	`float`
`dd` nebo `DD`	`_Decimal64 (DF: 32, DL: 128)`

Příklady:

```
0.    .0    1.0e-3L    3.14159F    3e1    -2e+9    0.1DD
```

Poznámky:

C17 šestnáctkový formát: `0xfeeP10F` (exponent 2^{10})

C23 desítkový základ (např. finanční aplikace): `_Decimal64`

Znakové literály

`'<znak>'` znakový literál, je typu `int`

Speciální znaky (”escape-sequence”)

`\'`	apostrof
`\a`	alert – pípnutí
`\b`	backspace – posun zpět (BS)
`\f`	nová stránka (FF)
`\n`	nový řádek (LF, CRLF)
`\r`	návrat vozíku (CR)
`\t`	horizontální tabelátor (HT)
`\\`	znak `\`
`\x<číslo>`	znak s ordinálním číslem hexadecimálně
`\<číslo>`	znak s ordinálním číslem oktalově

Příklady:

```
'n'     '\\'     '\n'     '\x80'     '\0'          '\''
```

Víceznakové a ’široké’ literály

Znakové konstanty

anglický termín	zápis	typ
multi-byte character	`'xx'`	`int`
UTF-8 character	`u8'x'`	`char8_t`
wide character	`L'x'`	`wchar_t`
UTF-16 character	`u'x'`	`char16_t`
UTF-32 character	`U'x'`	`char32_t`

Poznámky:

- UNICODE kódování: `'\u03A9'` Ω, `'\U0001F596'`
- Víceznaková (multi-byte) reprezentace: UTF-8
- Kódování zdrojového textu

Řetězcové literály

Řetězcové konstanty

zápis	typ
`"<znaky (i speciální a multi-byte)>"`	`char[]`
`u8"<znaky (i speciální a multi-byte)>"`	`char8_t[]`
`L"<znaky (i speciální a multi-byte)>"`	`wchar_t[]`
`u"<znaky (i speciální a multi-byte)>"`	`char16_t[]`
`U"<znaky (i speciální a multi-byte)>"`	`char32_t[]`

- Překladač doplní na konec řetězce znak `'\0'`
- Překladač překóduje obsah řetězce podle implementací definované lokalizace. Použije funkce `mbstowcs`, `mbrtoc16*`, `mbrtoc32*`

Řetězcové literály – příklady

```
  ""              // prázdný řetězec (char[1])

  "text se znakem \" v řetězci"

  "text jako v originálním K&R C a \
pokračování textu na následujícím řádku"

  "řetězec1" <mezery, nové řádky>
  "řetězec2"    // ANSI-C "řetězec1řetězec2"

  "\0" "12"     // char[4]: '\0' '1' '2' '\0'

  u8"něco"      // minimálně char[6]

  "a" "b" U"c"  // U"abc"
```

Deklarace

```
datový_typ  deklarátor [=ini_hodnota], ... ;
```

Příklady:

```
char  line[100];  /* definice pole 100 znaků     */
void  f(void);    /* deklarace funkce - prototyp */

void (*fp)(void);          /* ukazatel na funkci */
void (*p[9])(void);  /* pole ukazatelů na funkce */

const char  backslash = '\\';
int         i;
int * const p = &i;       /* konstantní ukazatel */
double      eps = 1.0e-5;
```

Okamžik inicializace

- statické (obvykle globální) proměnné se inicializují při překladu, bez inicializace mají hodnotu `0`
- automatické (lokální) proměnné se inicializují, když tok řízení dosáhne místa deklarace, bez inicializace mají nedefinovanou hodnotu
- dynamické (`malloc()`) nelze přímo inicializovat

Příklad:

```
int x;          // inicializováno při překladu na 0
void f(void) {
  int i;                    // i neinicializováno
  int j = 1;  // inicializováno při každém volání
  char *p = malloc(10);  // p inicializováno, *p ne
  static int c;  // c inicializováno na 0 jen 1x
}
```

Rozsah deklarace

- identifikátor deklarovaný na globální úrovni má rozsah od místa deklarace do konce zdrojového textu modulu
- identifikátor uvedený jako formální parametr funkce má rozsah od místa deklarace do konce těla funkce
- identifikátor deklarovaný uvnitř bloku má rozsah do konce bloku
- návěští má rozsah funkce, ve které je definováno
- jméno makra má rozsah od příkazu `#define` do konce textu modulu nebo do příkazu `#undef`

Viditelnost identifikátoru

Příklad:

```
int x = 10;

int main() {
  double x = 1.1;    /* ve funkci platí double x */
  printf("%f",x);
}

void printx(void) {           /* zde platí int x */
  printf("%d",x);
}
```

typedef

Vytváření nových jmen (synonym) datových typů.
Nejde o vytvoření nového typu.

Příklady:

```
typedef int length_t;
length_t len, maxlen=100;

typedef int (*ptr2f_t)(void); /* ukaz. na funkci */
ptr2f_t numcmp, swap;
```

```
typedef char * string_t;
string_t  s, lineptr[10];
```

je ekvivalentní:

```
char *s, *lineptr[10];
```

Ukazatel, adresa

- Deklarace/definice ukazatele

```
int *px = NULL;
```

- Získání ukazatele na objekt (proměnnou)

```
int x, y;
px = &x;  // operátor získání adresy objektu
```

- Odkaz na objekt (proměnnou)

```
y = *px;  // operátor zpřístupnění objektu
```

```
px = &x;
y  = *px; // v tomto kontextu ekvivalent y = x
```

- Výskyt na levé straně přiřazení (pojem L-hodnota)

```
*px = 0;
*px += 4;
(*px)++;
```

Použití ukazatelů

- Předávání parametrů odkazem

```
void f(char *s, int (*p)[3], void (*fp)(void));
```

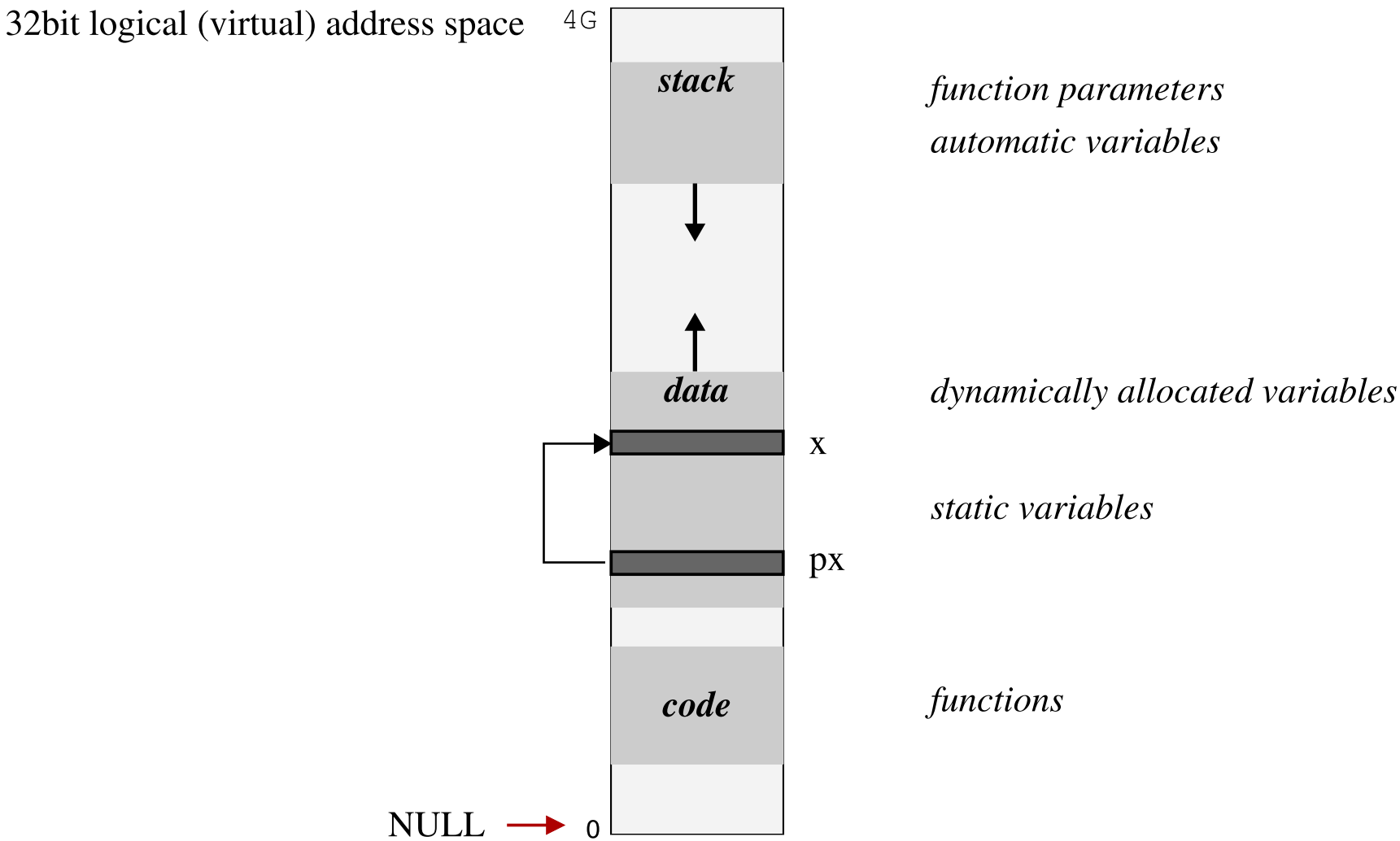
- Dynamické přidělování paměti

```
double *s = malloc(100*sizeof(double));
```

- Dynamické datové struktury

```
// Linux lists:
struct list_head queue = LIST_HEAD_INIT(queue);
// ...
struct list_head *pos;
list_for_each(pos, queue) {
    struct node *d;
    d = list_entry(pos, struct node, head);
    d->data = 0;
}
```

Adresový prostor – obrázek



Ukazatele jako argumenty funkcí

Příklad: Vzájemná záměna hodnot proměnných

```
void swap(int *px, int *py) {
  register int temp = *px;
  *px = *py;
  *py = temp;
}

Volání: int x,y;
        /*...*/
        swap(&x,&y);
             ^^^^^ pozor!
```

Poznámka: C++ a typ reference

Ukazatel a pole

```
int a[10], *pa;
pa = &a[0];  // ukazatel na první prvek pole
x = *pa;     // je ekvivalentní  x = a[0]
```

Ukazuje-li `pa` na `i`-tý prvek pole, potom `(pa+1)` ukazuje na prvek `i+1`. V našem příkladu `*(pa+1)` je stejné jako `a[1]`
Je možné místo `pa = &a[0];` napsat `pa = a;` a potom platí: `a[i] == *(pa+i)`

Obecně pro jazyk C vždy platí:

```
a[i] == *(a+i)
```

Poznámky:

- Není možné zapsat `a++`
- Meze polí se nekontrolují

Adresová aritmetika

S ukazateli lze provádět aditivní operace.

- Jednotkou výpočtu je velikost cílového objektu.
- Operace mají smysl jen v rámci jednoho pole.
- Například:
  - posun ukazatele o 1 prvek: `p++` (vpřed), `p--` (vzad)
  - posun ukazatele o N prvků: `p+=5` (vpřed o 5 prvků)
  - rozdíl ukazatelů: `p2 - p1` (počet prvků mezi ukazateli)

Příklad: Posun ukazatele na další prvek pole

```
int *ip = pole;
ip++;           // posun adresy o sizeof(int) bajtů
```

Omezení operací s ukazateli

Dva ukazatele nelze sčítat, násobit, dělit, posouvat ani kombinovat s float a double — překladač by měl hlásit chybu.

Pozor — chybné operace s ukazatelem

```
const char *p1 = "abcdef";
const char *p2 = "ghijkl";
p1 += p2;     // chyba - nesmyslná adresa
p1 *= 2;      // chyba - nesmyslná adresa
p1 >>= 2;     // chyba - nesmyslná adresa
p1 += 3.14;   // chyba - nesmysl

const void *p = p1;
p += 3;       // chyba - nelze pro void*
```

Příklad: délka řetězce

```
size_t strlen(const char *s)  {
   size_t n;
   for(n=0; *s!='\0'; s++) n++;
   return n;
}
```

Efektivnější varianta:

```
size_t strlen(const char *s) {
   const char *p = s;
   while( *p ) p++;
   return p-s;          // rozdíl ukazatelů
}
```

Poznámky:

- Je možné i volání `f(&a[2])` resp. `f(a+2)`
- Nejefektivnější je obvykle `strlen` z knihovny.

Přehled povolených operací:

`ptr+int, ptr+=int, ptr++`	posun
`ptr-int, ptr-=int, ptr--`	posun
`ptr1 - ptr2`	rozdíl
`ptr[i]`	indexování
`*ptr`	zpřístupnění cíle
`ptr1 = ptr2`	přiřazení

Upozornění:
Operace s neinicializovaným ukazatelem jsou nebezpečné a nelze je většinou kontrolovat překladačem!

Pozor — typická chyba

```
char *s;   // Neinicializováno nebo implicitní NULL
*s = 'A';  // Zápis znaku na chybné místo v paměti
```

Řetězce

Řetězec je pole znaků, lze jej tedy ”přiřadit”ukazateli:

```
char *message = "";              // inicializace
message = "Now is the time."; // nejde o kopii!
```

Operace s řetězci:

- Kopie znakových řetězců:

```
char *strcpy(char *s1, const char *s2)  {
  char *s = s1;
  while( *s++ = *s2++ );
  return s1;
}
```

  Upozornění:
  - obrácené pořadí parametrů: `strcpy(kam,odkud)`
  - nekontroluje velikost řetězců (možné buffer overflow)
  - chování nedefinováno pro překrývající se `s1` a `s2`

Řetězce – pokračování

- kopie řetězců s omezením délky:

```
char *strncpy(char *s1, const char*s2, size_t n);
```

- spojování řetězců:

```
char *strcat(char *s1, const char*s2);
char *strncat(char *s1, const char*s2, size_t n);
```

- lexikální porovnání řetězců:

```
int strcmp(const char*s1, const char *s2);
int strncmp(const char*s1, const char *s2, size_t n);
int strcoll(const char *s1, const char *s2);
```

- mnoho dalších funkcí (`memcpy`, `strchr`, `strstr`, `strlen`, C11: `strcpy_s`, `strnlen_s`, `strcat_s`, ...)
  viz std. rozhraní `<string.h>`

Vícerozměrná pole

- prvkem pole může být opět pole (do libovolné úrovně)

```
static int day_tab[2][13] = {
  { 0,31,28,31,30,31,30,31,31,30,31,30,31 },
  { 0,31,29,31,30,31,30,31,31,30,31,30,31 },
};
```

- prvky jsou uloženy po řádcích
- nelze zkracovat `a[x][y]` jako `a[x,y]` (není hlášena chyba!)
- parametry typu pole se předávají vždy odkazem
- v parametrech není nutné udávat rozměr nejlevějšího indexu (počet řádků). Pro mapovací funkci to není podstatné. (Příklad: `int f(int day_tab[][13]);`)
- parametr typu pole (`T[]`) je ekvivalentní parametru typu ukazatel (`T*`)

Inicializace pole ukazatelů

```
const char *month_name(int n)
{ // pole ukazatelů na řetězce
  static const char *name[] = {
    "illegal month",
    "January", "February", "March",
    "April",   "May",      "June",
    "July",    "August",   "September",
    "October", "November", "December"
  };
  return (n<1 || n>12)? name[0] : name[n];
}
```

Poznámka:

```
char a[10][20]; // pole znakových polí
char *b[10];    // pole ukazatelů na řetězce
```

Argumenty příkazového řádku (command line)

Programu TEST lze zadat argumenty při spuštění:

```
TEST     argument1 argument2 .... poslední_argument
argv[0] argv[1]    argv[2]          argv[argc-1]
```

- `argv[0]` je jméno programu (nebo `""`),
- `argv[argc]` má hodnotu `NULL`

Příklad: Výpis argumentů – program echo

```
int main(int argc, char *argv[]) {
  for(int i=1; i<argc; i++)
    printf("%s ", argv[i]);
  printf("\n");
}
```

Druhá (horší) varianta výpisu argumentů

```
int main(int argc, char *argv[]) {
  while(--argc>0)
    printf("%s ", *++argv);     // špatně čitelné
  printf("\n");
}
```

Poznámka: getopt a různé formáty argumentů

```
program  -i
program  -i hodnota
program  -i=hodnota
program  --include hodnota
program  --include[=hodnota]
```

Struktury

```
struct Date {                          // tag
  unsigned char day;
  unsigned char month;
  int year;
};

struct Date d = {21,10,1993};
struct Date dat;
```

Odkaz na člen struktury:

```
x = d.year;  // jméno_proměnné . jméno_členu
```

Je možné vnořování struktur

Struktury lze předávat jako parametry funkcím a vracet jako funkční hodnotu

Struktury — omezení

- Povolené operace jsou pouze:

`.`	zpřístupnění položky
`->`	zpřístupnění položky přes ukazatel
`=`	přiřazení (nebo inicializace) struktury
`&`	získání ukazatele na proměnnou

- Ostatní operace jsou zakázané (struktury nelze sčítat atd.).
  Poznámka: v C++ lze definovat potřebné operátory

Anonymní struktury a unie (C11)

```
struct X {
    int x;
    union /* no tag */ {
        int i;
        float f;
        struct /* no tag */ {
            int y;
            int z;
        };
    };
} s;

s.y = 1;
s.f = 3.14159F;
```

- jednodušší a čitelnější zápis

Neúplná deklarace struktury

```
struct ABC;
```

Lze použít pro definici ukazatelů:

```
struct ABC *p;          // O.K.
```

Lze použít pro externí deklaraci proměnné:

```
extern struct ABC x;    // O.K.
```

Nelze použít pro definici proměnné:

```
struct ABC x;           // chyba
```

Použití ukazatele na strukturu

```
struct Date *ptr = &d;
```

`x = ptr->year;` je ekvivalentní příkazu `x = (*ptr).year;`

Operátory `.` a `->` mají vysokou prioritu (viz tabulka operátorů)
`++ptr->day` je proto ekvivalentní `++(ptr->day)`

Poznámky:
Dávejte přednost čitelnější formě: `ptr->member`
Nepřehánět: `ptr->next->next->next->prev`

Velikost struktury v bajtech

operátor `sizeof`

```
sizeof(struct XYZ)
sizeof(dat1)
sizeof expr   // bez závorek jen pro výrazy
```

- Velikost struktury nemusí být součtem velikostí jednotlivých složek. Důvodem je zarovnání (*alignment*) složek i celkové velikosti struktury.
- Pozor na vložené místo (*padding*) — při kopírování celých struktur (`memcpy`, `fwrite`, ...) je možný únik informací.

Poznámky: *Little endian / big endian*,
`_Alignof(typ)`
`_Alignas(typ)` `_Alignas(const-expr)`

Příklad — počítání slov v textu ...

Struktura může obsahovat ukazatel na sebe:

```
struct treenode {
  char *word;              // řetězec
  int count;
  struct treenode *left;         // levý následník
  struct treenode *right;        // pravý následník
};
int main()  {              // neúplná implementace
  struct treenode *root = NULL;
  char word[MAXWORD];
  int len;
  while( (len=getword(word, MAXWORD)) != EOF)
    if(len>0)
      root = tree_insert(root,word);
  tree_print(root);
}
```

Příklad — ... vložení do stromu

```
struct treenode *
tree_insert(struct treenode *p, const char *w) {
  int cond;
  if( p == NULL ) {      // nové slovo
    p = tree_alloc();    // malloc s kontrolou NULL
    p->word = strdup(w);
    p->count = 1;
    p->left = p->right = NULL;
  } else if( (cond=strcmp(w,p->word)) == 0 )
    p->count++;          // == slovo se opakuje
  else if( cond<0 )      // <  do levého podstromu
    p->left  = tree_insert(p->left,w);
  else                   // >  do pravého podstromu
    p->right = tree_insert(p->right,w);
  return p;
}
```

Bitová pole (*bitfields*)

- Vhodné pro úsporné uložení dat na několika bitech
- Lze použít pouze uvnitř struktur (a unií)

Příklad:

```
struct flags {
  unsigned is_kw   : 1;  // uloženo v jednom bitu
  unsigned is_ext  : 1;
  unsigned         : 2;  // výplň 2 bity
  unsigned is_stat : 1;
  unsigned         : 0;  // zarovnání na unsigned
  unsigned num_bit : 12; // uloženo ve 12 bitech
} flags; // velikost 2*unsigned
```

Bitová pole — pokračování

Nevýhody:

- Pomalejší přístup k položkám
- Nelze použít operátory `&` (získání adresy), `sizeof` ani makro `offsetof`.
- Některé detaily jsou definovány implementací:
  - Pořadí obsazování bitů (od LSB nebo od MSB)
  - Položka přesahující konec alokační jednotky je nebo není zarovnána
  - `int b:3;` může mít hodnoty 0..7 nebo −4..3
    Používejte např. `unsigned int b:3;`

Poznámka: Lze nahradit explicitní prací s bitovými operacemi

Unie (1.p union)

- Všechny položky začínají na stejné adrese (začátek unie) a překrývají se. Platná (aktivní) je vždy jen jedna položka.
- Použití složek je stejné jako u struktur
- Inicializovat lze pouze první složku (C90), podle C99+ jakoukoli složku `{ .name = value }`

Příklad:

```
union u_tag {
  int ival;
  float fval;
  char *pval;
} u;

union u_tag x = { 1 }; //C99+: = { .fval = 0.1 };
```

```
if(typ==INT)
  printf("%d\n", u.ival);
else if(typ==FLOAT)
  printf("%f\n", u.fval);
else if(typ==STRING)
  printf("%s\n", u.pval);
else
  error("chyba");
```

Ukazatel na funkci

obsahuje adresu kódu funkce, může být použit pouze k volání funkce (ukazatelová aritmetika není použitelná). Ukazatel:

```
T (*fp)(void);
```

reprezentuje *ukazatel na funkci bez parametrů vracející typ T*. Máme-li funkci:

```
T funkce(void) { /* kód funkce */ }
```

potom přiřazení do ukazatele má v ISO C tvar:

```
fp = &funkce;  // & lze vynechat
```

Použití ukazatele k volání funkce:

```
(*fp)();       // volání cílové funkce
fp();          // lze zkrátit (doporučuji)
```

Příklad — řazení pole ukazatelů na objekty ...

```
//*** řadicí algoritmus - netestováno, doladit!
void sort( void *v[], unsigned n,
           int (*cmp)(const void *p1, const void *p2))
{
  int gap, i, j;
  for( gap=n/2; gap>0; gap/=2 )
    for( i=gap; i<n; i++ )
      for( j=i-gap; j>=0; j-=gap ) {
        void *tmp;
        if(cmp(v[j],v[j+gap]) <= 0)  // porovnání
          break;
        tmp = v[j];                  // swap
        v[j] = v[j+gap];
        v[j+gap] = tmp;
      }
}
```

Příklad — dokončení

Příklad: Řazení pole ukazatelů — použití:

```
char *lineptr[100]; // pole řetězců
int  nlines;
....
sort(lineptr,nlines,&strcmp);
                    ^^^^^^^ Problém: konverze
```

kde `int strcmp(const char *s1, const char *s2);` je standardní funkce z knihovny (viz `<string.h>`).

Poznámka:
Standardní knihovna obsahuje `qsort()`

Výčtový typ

- Konstanty ve výčtu jsou typu `int` a mají hodnoty rostoucí postupně od nuly
- Lze explicitně specifikovat hodnoty položek (konstant)
- Proměnné výčtového typu jsou kompatibilní s typem `int`
- Označení (*tag*) výčtu nesmí kolidovat:
  `struct A {}; enum A {}; union A {};` je chyba!

Příklady

```
enum dny { PO, UT, ST, CT, PA, SO, NE };
typedef enum Boolean { FALSE, TRUE } Boolean;
enum bitmask { bit0=1, LSB=bit0, bit1,
               bit2=1<<2, bit3=1<<3, bit4=1<<4 };
enum { RAZ, DVA, TRI } e; // proměnná, "no tag"
```

Použití výčtu

Příklad:

```
enum dny d=PO;
int i = d;          // hodnota 0
d++;                // přičte 1, pozor na výsledek
printf("%d", d);    // vytiskne 1

d = 2;          // v C přípustné, ale NEPOUŽÍVAT!
```

Poznámka: V C++ jsou i lepší výčtové typy (`enum class`)

Operátory

Aritmetické operátory

+ -	(binární i unární)
* /	
%	modulo

Pro komutativní operátory není definováno pořadí vyhodnocení operandů
Pozor na operátor % a na výpočty v plovoucí čárce

Relační operátory

==	!=	<	>	<=	>=

Operátory – pokračování

Logické operátory

`||`	logické nebo (OR)
`&&`	logické a (AND)
`!`	logické ne (NOT)

Zkrácené vyhodnocování:

`A || B`	je-li `A!=0`, pak se `B` nevyhodnocuje
`A && B`	je-li `A==0`, pak se `B` nevyhodnocuje

Příklad: Přestupný rok (Gregoriánský kalendář)

```
(year%4 == 0 && year%100 != 0 || year%400 == 0)
```

Implicitní konverze typů

- *Celočíselná rozšíření* (*integral promotions*):
  zachovávají hodnotu včetně znaménka
  `char`, `short int`, bitová pole → `int` nebo `unsigned`

  Poznámka: argumenty funkcí (...), K&R `float` → `double`

- *Obvyklé aritmetické konverze* u binárních aritmetických operací:
  1. Je-li jeden operand typu `long double`,
     je druhý převeden na `long double` a výsledek je také `long double`
  2. jinak, je-li `double` ...
  3. jinak, je-li `float` ...

Implicitní konverze typů – pokračování

Jinak se provedou celočíselná rozšíření pro oba operandy a potom:

1. jestliže oba operandy mají stejný typ neprovádí se další konverze,
2. jinak, jsou-li oba operandy `signed` nebo oba `unsigned` převede se vše na typ s větším rozsahem.
3. Jinak, jestliže `unsigned` typ má větší nebo stejný rozsah, je druhý operand převeden na `unsigned`.
4. Jinak, jestliže `signed` operand může reprezentovat všechny hodnoty `unsigned` operandu, je druhý operand převeden na `signed`.
5. Jinak jsou oba převedeny na odpovídající `unsigned` typ.

Problém `signed/unsigned char`

– vzniká při převodu znaků na int:

`signed char` → `int`	rozsah −128 .. 127
`unsigned char` → `int`	rozsah 0 .. 255

Příklad:

```
int getchar(void);
/*  vrací EOF (tj. -1) nebo znak 0 .. 255  */

int c;                  /* musí být typu int !!! */
c = getchar();
if(c==EOF) ....         /* pro char c; bude chybné */
```

Explicitní konverze

```
(typ) výraz
```

Příklad: problém kontextové operace dělení

```
int i1,i2;
double f;
...
f = i1/i2;              // celočíselné dělení
f = (double)i1/i2;

f = i1/(i2+1.0)
```

Operátory – pokračování

Operátory ++ a --

`++` increment (zvýšení o 1)
`--` decrement (snížení o 1)

`++i`	před použitím hodnoty	prefixový zápis
`i++`	po použití hodnoty	postfixový zápis

Příklad:

```
int x, n=5;
x = n++;      /* x = 5, n = 6 */
x = ++n;      /* x = 7, n = 7 */
```

Poznámka: Tyto unární operátory lze použít pouze pro proměnné! (přesněji: L-hodnoty).
Nelze například napsat `(55+5)++`

Příklad

Příklad: odstranění zadaného znaku z řetězce

```
void squeeze( char s[], char c )  {
   int i,j;
   for( i=j=0; s[i] != '\0'; i++ )
      if( s[i] != c )
         s[j++] = s[i];
   s[j] = '\0';
}
```

Operátory – pokračování

Logické operátory po bitech

`&`	bitové AND (*bitwise and*)
`|`	bitové OR
`^`	bitové XOR
`~`	bitová negace (NOT)

Posuny bitů

`<<`	posun vlevo
`>>`	posun vpravo

Poznámka: `unsigned` operand: logický posun,
`signed` operand: nedefinováno/aritmetický posun

Příklad

Příklad: čtení bitového pole z proměnné

```
int getbits(unsigned x, unsigned p, unsigned n) {
  return (x>>(p+1-n)) & ~( ~0 << n) ;
}        |           |   |         |
         \-----------/   \---------/
         posun na pravý  maska 000000111111
         okraj slova                 n jedniček
```

Jak se vyhnout závislosti na počtu bitů slova:

`~0` = samé jedničky ve slově libovolné délky (nezávislé)

`x & ~077` nezávislé na délce slova

`x & 0177700` závislé – uvažuje 16 bitů

Operátory – pokračování

Přiřazovací operátory a výrazy

```
+=   -=   *=   /=   %=   <<=   >>=   &=   ^=   |=
```

Přiřazení má hodnotu a typ levé strany.
`c1 op= c2` je prakticky ekvivalent `c1 = c1 op (c2)`
ale podvýraz `c1` vyhodnotí jen jednou.

Příklad: (pozor na závorky)

```
x *= y + 1                    x = x * (y + 1)
```

Není nutné optimalizovat společné podvýrazy.
Často je čitelnější: `yyval[yypv[p3+p4]+yypv[p1+p2]] += 2;`

Příklad: součet jednotkových bitů

```
int bitcount(unsigned n)  {
   int b;
   for( b=0; n != 0; n >>= 1 )
      if(n & 1)  // hodnota nejnižšího bitu
         b++;
   return b;
}
```

Poznámka: Pozor na vedlejší efekty

```
i = i++;                  // nedefinováno!
pole[j] = pole[k]++;      // nedefinováno pro j==k
*ptr1 = (*ptr2)++;        // nedefinováno pro ptr1==ptr2
```

Operátory – pokračování

Podmíněný výraz

```
e1 ? e2 : e3
```

Je-li `e1!=0`, pak výraz má hodnotu e2, jinak e3
Vyhodnocuje se pouze jeden z výrazů e2, e3

Příklad: formátování tisku

```
for(int i=0; i < N; i++)
  printf("%6d%c", a[i], (i%10==9||i==N-1)?'\n':' ');
```

Příklad: výběr maxima

```
int max(int a, int b) {
   return  (a>b) ? a : b;
}
```

Priorita a asociativita operátorů

operátory	asociativita
`( ) [ ] -> .`	→
`! ~ + - ++ -- & * (typecast) sizeof`	←
`* / %`	→
`+ -`	→
`<< >>`	→
`< <= > >=`	→
`== !=`	→
`&`	→
`^`	→
`|`	→
`&&`	→
`||`	→
`?:`	←
`= *= /= %= += -= &= ^= |= <<= >>=`	←
`,`	→

Příkazy

Výraz-příkaz

Výraz se stane příkazem, zapíšeme-li za něj středník.
Hodnota takového výrazu se zanedbá.

Příklad:

```
x = 0;
i++;
printf("xxx");
```

Složený příkaz – blok

```
{ deklarace příkaz1 příkaz2 .... příkazN }
```

Poznámky:

Za } není středník! (pozor na makra)
C99: deklarace proměnných mohou být mezi příkazy

Příkazy – pokračování

Podmíněný příkaz

```
if( výraz )  příkaz1
[ else  příkaz2 ]
```

Poznámka: nejednoznačnost se řeší jako v Pascalu a jiných jazycích: `else` patří k poslednímu volnému `if`

Příklad: binární vyhledávání

```
/* hledáme  x  v poli  v[]  o rozměru  n */
int binary( int x, int v[], int n)  {
  int low = 0;
  int mid;
  int high = n - 1;
  while( low<=high ) {
    mid = (low + high) / 2;
    if( x<v[mid] )
      high = mid - 1;
    else if ( x>v[mid] )
      low = mid + 1;
    else
      return mid;      /* nalezeno mid */
  }
  return -1;           /* nenalezeno */
}
```

Příkazy – pokračování

Příkaz `switch`

```
switch( výraz )  {
  case konstantní_výraz : příkaz1
  ....
  [ default : příkazNPLUS1; ]
}
```

Poznámka: Pozor: funguje jinak než v Pascalu!

Příklad: počítání číslic, prázdných znaků a ostatních

```
int main()  {
  int c, i, nwhite=0, nother=0, ndigit[10] = { 0, };
  while( (c=getchar()) != EOF )
    switch(c) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
                ndigit[c-'0']++;
                break;  /* nepokračovat */
      case ' ': case '\n': case '\t':
                nwhite++;
                break;  /* ukončí switch */
      default : nother++;
                break;  /* i zde je dobré */
    } /* switch i while */
  printf("digits = ");
  for(i=0; i<10; i++)
    printf(" %d ", ndigit[i]);
  printf("\n ws=%d, other=%d\n", nwhite, nother);
}
```

Příkazy – pokračování

Cyklus `while`

```
while ( <výraz> )
  <příkaz>
```

Cyklus `for`

```
for( <výraz1>; <výraz2>; <výraz3> )
  <příkaz>

for(;;) <příkaz>          // nekonečný cyklus

for(int i=0; i<N; i++) // běžné použití
  pole[i]=0;
```

Příkazy – pokračování

Příklad: Algoritmus řazení (shell-sort)

```
// funkce řadí pole v[] o rozměru n vzestupně

void shell(int v[], int n) {
  for(int gap=n/2; gap>0; gap/=2)
    for(int i=gap; i<n; i++)
      for(int j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
        int temp = v[j];
        v[j] = v[j+gap];
        v[j+gap] = temp;
      }
}
```

Operátor čárka

Postupné vyhodnocení výrazů, použití výsledku posledního.

Příklady:

```
void reverse(char s[]) { // obrácení řetězce
  int i, j;
  for( i=0, j=strlen(s)-1; i<j; ++i, --j) {
    int c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
}

assert( ("Index x mimo meze", x>=0 && x<MAX) );
```

Poznámka:
Čárky oddělující argumenty funkcí nejsou operátory!

Příkazy – pokračování

Cyklus `do-while`

```
do
  příkaz
while( výraz );  // cyklí dokud podmínka platí
```

Poznámky:
Provede se alespoň jednou.
Podle statistik cca 5% cyklů.

Příklad: konverze čísla na znaky v opačném pořadí

```
do {
  s[i++] = n % 10 + '0';
}while( (n/=10) > 0 );
```

Příkaz `break`

Ukončuje nejblíže nadřazený příkaz `switch`, `while`, `for`, nebo `do-while`

Příklad: vynechání koncových mezer a tabulátorů

```
int main() {
  int n;
  char line[MAXLINE];
  while( (n=getline(line,MAXLINE)) > 0 )  {
    while( --n >= 0 )
      if( !isspace(line[n]) )
        break;
    line[n+1] = '\0';
    printf("%s\n", line);
  }
}
```

Příkaz `continue`

Přeskočí zbytek těla cyklu a pokračuje podmínkou cyklu

```
for( i=0; i<N; i++ ) {
  if( a[i] < 0 )
    continue;
  ....  /* pouze pro >= 0 */
}
```

Poznámka: Eliminace zanoření těla cyklu

Příkaz `goto` a návěští

Použitelné pouze v rámci jedné funkce

```
void f(void) {
  // ....
  goto identifikátor;
  // ....
  // ....
  identifikátor : příkaz;
  // ....
}
```

Poznámky:
Používat opatrně, minimalizovat použití.
Nikdy neskákat do strukturovaných příkazů!

Příklad – praktické použití skoku

```
int f(void) {
    // ....
    if (chyba)
        goto error_exit;
    for (....)
        for (....) { // 2. úroveň
            // .... výpočet
            if (chyba)
                goto error_exit;
        }
    return kladny_vysledek;
  error_exit:
    // .... ošetření chyby
    return -1;
}
```

Funkce

```
typ jméno ( deklarace_parametrů )
{
  deklarace lokálních proměnných
  příkazy   // C99: i deklarace
}
```

Funkce může vracet struktury, unie ale *ne pole*

Příkaz `return`

```
return;   // jen pro funkce typu void f()
return e; // e = výraz kompatibilní s typem fce
```

Poznámky:

- C99: `inline` funkce, C11: `_Noreturn`

Rekurze

viz Rekurze.

Příklad: výpis celého čísla desítkově

```
void printd(int n) {
    int i;
    if(n<0)  {
        putchar('-');
        n = -n; // nefunguje pro INT_MIN
    }
    if((i=n/10) != 0)
        printd(i);          // rekurze
    putchar(n%10 + '0');
}
```

Poznámka: optimalizace, *tail recursion*

Argumenty funkcí

- jsou předávány hodnotou
- pole se předávají odkazem (ukazatel na první prvek)
- problémy s funkcemi s proměnným počtem argumentů
- C99: `__func__`, ...

Poznámka: API, ABI

Argumenty funkcí – pokračování

Funkce s proměnným počtem argumentů

```
typ f(typ1 parametr1, ... ); // variadic function
```

- Musí být alespoň jeden pevný parametr
- Funkce musí mít informace o skutečném počtu argumentů při zavolání
- `va_list`, `va_start()`, `va_arg()`, `va_end()`

Příklad: Standardní funkce s proměnným počtem argumentů

```
int printf(const char *fmt, ... );
int sprintf(char *s, const char *fmt, ... );
```

Externí proměnné

Program = množina globálních proměnných a funkcí

Globální proměnné

- Jsou to statické proměnné
- Inicializace ”při překladu”
- (Použití např. pro omezení počtu argumentů funkcí)

Deklarace funkce je implicitně externí (`extern` je zbytečné):

```
extern int plus(int,int);     int plus(int,int);
```

U proměnných je podstatný rozdíl:

```
extern int a;                 int a; // definice
```

Použití: extern deklarace před použitím, modularita

Poznámka: C89: nedeklarované funkce: `extern int f();`

Rozsah platnosti

(*scope*)
Úsek programu, ve kterém je jméno definované

automatické proměnné	blok
globální proměnné	od deklarace do konce souboru
jména parametrů	prototyp, funkce
návěští	funkce

Statické proměnné

existují trvale bez ohledu na aktivaci funkcí

Pozor: proměnná nebo funkce označená `static` je platná pouze v rámci souboru (modulu) a není viditelná z jiných modulů

Příklad:

```
int count;                          // statická a externí
static char buffer[100];            // statická a ne-externí
static int plus(int a, int b) // statická=ne-externí
{
    static int s = 1;               // statická a lokální
    int n = a + buffer[b];          // automatická=lokální
    return s = (n + count*s);
}
```

Proměnné `register`

- Pokud je to možné, jsou uloženy v registru procesoru (rychlost)
- Pouze pro automatické proměnné
- Nelze získat ukazatel na registrovou proměnnou

Příklad:

```
int swap(int *x, int *y) {
  register int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Bloková struktura programu

- Hierarchie bloků (zanoření do ”libovolné”úrovně)
- V bloku lze deklarovat proměnné
- Lokální deklarace překrývá ostatní (globální i lokální), platí to i pro jména formálních parametrů

Poznámky:

- Nelze vnořovat definice funkcí
- Makra nerespektují tuto strukturu (nejsou ”hygienická”)

Inicializace proměnných

- Statické jsou (”při překladu”) implicitně inicializovány hodnotou nula, případná inicializace proběhne při překladu (vyžaduje konstantní výraz)

```
const int nula;
static int dva = 2;
void f(int p) {
   static bool stav=true; // musí být konstanta
   // ....
}
```

- Automatické a registrové mají *nedefinovanou hodnotu*, případná inicializace se provede při běhu programu

```
void g(int v[], int n)  {
  int x;                  // nedefinovaná hodnota
  int high = v[n-1]; // nemusí být konst. výraz
  // ....
}
```

Příklady inicializace strukturovaných proměnných

Inicializace polí

```
int pole1[] = { 1, 1, 1, 0, 0, };
int pole2[10] = { 1, [5]=0, 1, }; // jen C99+

char string1[] = "the";         // použití řetězce
char string2[] = { 't', 'h', 'e', '\0' };

char nonstring[3] = "the";      // není řetězec!
```

Inicializace struktur

```
struct MyExtraComplex {
   double Re;
   int Im;
} c1 = { .Re=1.25, .Im=3 }; // C99 inicializace
```

Překladové jednotky (moduly)

Modul je samostatná překladová jednotka – soubor `*.c`

Příklad: pozor, NEVHODNÝ styl – hrozí nekonzistence

```
// === modul1.c ===
int x; // definice x
static int s;

void f2(double d);

int main() {
  f2(5);
  s--;
  return 0;
}
```

```
// === modul2.c ===
extern int x; // deklarace x
static int s;

static void f(void) {
  s++;
}
void f2(double d) {
  x = 10*d;
  f();
}
```

Překlad a sestavení více modulů: `cc modul1.c modul2.c`

Překladové jednotky – správné řešení

- Moduly zveřejňují rozhraní v souboru `*.h` (header file).
  V rozhraní jsou pouze:
  - deklarace proměnných
  - deklarace funkcí
  - definice typů
  - definice maker
  - definice inline funkcí

  (Jen pokud jsou sdílené více moduly.)
- Rozhraní se vkládá (`#include`) do modulů, aby byla při překladu zajištěna konzistence deklarací a kontrola definic.

  Poznámka: Aby to opravdu fungovalo, potřebujete správně použít program ”make”nebo podobný.

Překladové jednotky – správné řešení 2

```
// === rozhrani.h ===
extern int x;
void f2(double d);
```

```
// === modul1.c ===
#include "rozhrani.h"
static int s;
int x; /* definice */

int main() {
  f2(5);
  s--;
  return 0;
}
```

```
// === modul2.c ===
#include "rozhrani.h"
static int s;

static void f(void) {
  s++;
}
void f2(double d) {
  x = 10*d;
  f();
}
```

Preprocesor jazyka C

Direktivy začínají znakem #

Vložení souboru s rozhraním

```
#include <stdio.h>
#include "modul2.h"      // hledá i v ./
```

Definice makra bez parametrů

```
#define JMENO  text \
               text na dalším řádku
// ... každý výskyt JMENO se rozvine na text
#undef JMENO
```

Příklad:

```
#define EOF  -1
#define NULL ((void*)0)
```

Makra s parametry

Příklad: pozor na priority a vedlejší efekty

```
#define SQR(a)  ((a)*(a))
  // použití:
  x = SQR(p+q); // ((p+q)*(p+q)) O.K.
  x = SQR(i++); // ((i++)*(i++)) nedefinováno


#define ABS(a)     ( ((a)<0) ? -(a) : (a) )

#define PRIKAZ(x)  do { neco(x); } while(0)
  //....
  if(a) PRIKAZ(a);
  else  PRIKAZ(b);
```

Podmíněný překlad

Vynechání úseků programu

```
#ifdef JMENO                    #ifndef JMENO
  .....                           .....
#endif /* JMENO */              #endif /* !JMENO */
```

```
#if konstantní_výraz
  .....
#elif konstantní_výraz
  .....
#else
  .....
#endif
```

V podmínce jsou použitelné pouze konstantní výrazy:

```
defined(linux) && defined(i386)
__BORLANDC__ >= 0x0300
```

Spojování parametrů a vytváření řetězců

#	vytvoření řetězce (”stringize”)
##	spojení identifikátorů

Příklad: ukázka použití

```
#define SPOJ(a,b)   a##b
#define PRINT(x)    printf(#x " = %d\n", x)
```

Použití a výsledek:

```
SPOJ(file,id)    fileid
PRINT(pocet)     printf("pocet" " = %d\n", pocet)
```

Parametry překladu

Nastavení různých parametrů překladu:

```
#pragma  <parametry>
```

Příklad: Borland C / DOS

```
#pragma option -K        /* unsigned char */
#pragma warn amb
```

Příklad: GCC

```
#pragma message "Compiling " __FILE__ "..."
#pragma GCC optimize ("-O1")
```

Poznámka: `_Pragma`

Standardní ISO-C knihovny

soubor	co obsahuje
`assert.h`	Makro assert pro ladění.
`ctype.h`	Makra pro klasifikaci znaků.
`errno.h`	Definuje pojmenování chybových kódů.
`float.h`	Parametry pro floating-point funkce.
`limits.h`	Rozsahy celých čísel.
`locale.h`	Funkce pro národní/jazykovou podporu.
`math.h`	Matematické funkce.
`setjmp.h`	Typy pro funkce `longjmp` a `setjmp`.
`signal.h`	Konstanty a deklarace pro funkce `signal` a `raise`
`stdarg.h`	Makra pro práci s proměnným počtem argumentů.
`stddef.h`	Některá makra a datové typy.
`stdio.h`	Typy,makra a funkce pro standardní vstup/výstup
`stdlib.h`	Obecně použitelné funkce (konverze, řazení)
`string.h`	Funkce pro práci s řetězci a pamětí.
`time.h`	Typy a funkce pro práci s časem.

`assert.h` – Makro pro ladění

Implementace:

```
#define assert(podminka) if(!podminka) .....
```

- diagnostika logických chyb – testování podmínek (precondition, postcondition)
- při nesplnění podmínky vypíše:

text	`"Assertion failed: "`
text podmínky	`x < 0`
jméno souboru	`__FILE__`
číslo řádku	`__LINE__`
C99: jméno funkce	`__func__`

- lze vypnout definováním makra `NDEBUG`

```
#define NDEBUG
#include <assert.h>
```

Příklad

```
//#define NDEBUG
#include <assert.h>

double logarithm(const double x) {
   assert( x > 0.0 );
   /* zde platí podmínka */
   ....
}
```

`ctype.h` – makra pro klasifikaci znaků

`int isalnum(int c)`	písmeno nebo číslice [A-Za-z0-9]
`int isalpha(int c)`	písmeno [A-Za-z]
`int iscntrl(int c)`	řídicí znak
`int isdigit(int c)`	číslice [0-9]
`int isgraph(int c)`	tisknutelný znak bez mezery
`int islower(int c)`	malé písmeno [a-z]
`int isprint(int c)`	tisknutelný znak včetně mezery
`int ispunct(int c)`	tisknutelný znak bez alnum a mezery
`int isspace(int c)`	oddělovač
`int isupper(int c)`	velké písmeno [A-Z]
`int isxdigit(int c)`	šestnáctková číslice [0-9A-Fa-f]
`int toupper(int c)`	převod znaku na velké písmeno
`int tolower(int c)`	převod znaku na malé písmeno

Poznámky: Rozsah parametru: unsigned char + `EOF`
Makra vrací hodnoty `==0` a `!=0` (NE `0` nebo `1`)

`errno.h` – chybové kódy

`int errno`
je globální (pseudo)proměnná nastavovaná std. funkcemi,
na začátku programu je nulová

Norma definuje pouze základní hodnoty:

`EDOM`	doménová chyba
`EILSEQ`	C99: chyba konverze na `wchar_t`
`ERANGE`	přetečení nebo podtečení

implementace pak podle OS doplňuje další

Poznámka: `perror()`

`float.h` – charakteristiky floating-point typů

- Zaokrouhlování: `FLT_ROUNDS`
- `FLT_EVAL_METHOD`
- Základ: `FLT_RADIX`
- Minimum, maximum a přesnost: `{FLT|DBL|LDBL}`
  `*_MANT_DIG`, `*_DIG`, `DECIMAL_DIG`,
  `*_MAX`, `*_MIN`, `*_EPSILON`,
  `*_MIN_EXP`, `*_MIN_10_EXP`,
  `*_MAX_EXP`, `*_MAX_10_EXP`,

`limits.h` – rozsahy celočíselných typů

`CHAR_BIT` = počet bitů typu `char`
`MB_LEN_MAX` = max počet bajtů v `'xxx'`

```
SCHAR_MIN, SCHAR_MAX, UCHAR_MAX, CHAR_MIN, CHAR_MAX
SHRT_MIN, SHRT_MAX, USHRT_MAX
INT_MIN, INT_MAX, UINT_MAX
LONG_MIN, LONG_MAX, ULONG_MAX
```

C99: `LLONG_MIN, LLONG_MAX, ULLONG_MAX`

`locale.h` – funkce pro národní/jazykovou podporu

Konvence pro formáty: datum, čas, měna, znaky abecedy, řazení řetězců, ...
Ovlivňuje chování standardních funkcí (`isalpha()`, ...)
Definuje typ `struct lconv`,
makra `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`
a funkce

```
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);
```

Po startu programu platí:

```
setlocale(LC_ALL, "C");
```

`math.h` – matematické funkce

Definuje všechny běžné matematické funkce pro `double`, `float` (přípona `f`), a `long double` (přípona `l`):

```
sin, asin, cos, acos, tan, atan, atan2, ...
sqrt, pow, exp, log, log10, ...
lgamma, tgamma, ...
fabs, fmin, fmax, ...
floor, ceil, round, fmod, modf, ...
```

makra `INFINITY`, `isinf()`, `NAN`, `isnan(x)`

Poznámky:

C99: `complex` varianty: `csin`, `csinf`, ...

`setjmp.h` – nelokální skoky

- deklarace:

```
int setjmp(jmp_buf jmpb);
void longjmp(jmp_buf jmpb, int retval);
```

- `setjmp` – příprava pro nelokální skok
- `longjmp` – provede nelokální skok
- volání `longjmp` obnoví stav programu tak, jakoby `setjmp` skončil s návratovou hodnotou `retval`. (`longjmp` nemůže předat hodnotu `0` v `retval`, v takovém případě je změněna na `1`.)
- `longjmp` může být voláno pouze z funkce, která byla zavolána z funkce, která volala příslušný `setjmp`.
- `setjmp` lze volat jen z vhodných míst a dojde k narušení ne-volatile proměnných, které byly změněny mezi voláním `setjmp` a `longjmp`.

`setjmp.h` – nelokální skoky

- `setjmp` je použitelné pro ošetření chyb a výjimek (někdy se používá i pro implementaci kooperativního zpracování úloh - viz ”coroutines”)

Návratové hodnoty:

- `setjmp` vrací `0` když je volán. Když dojde k návratu z `setjmp` po volání `longjmp`, `setjmp` vrací nenulovou hodnotu.
- `longjmp` se nikdy nevrací

Příklad použití `setjmp()`, `longjmp()`

```
void subroutine(jmp_buf jmp)  {
   longjmp(jmp,1);
}

int main()  {
   int value;
   jmp_buf jmp;
   value = setjmp(jmp);
   if (value != 0)  {
     printf("Byl volán longjmp(jmp,%d)\n", value);
     exit(value);
   }
   printf("Volání podprogramu ... \n");
   subroutine(jmp);
}
```

`signal.h` – zpracování signálů

Komunikace procesů, výjimky

```
void (*signal(int sig, void (*func)(int s)))(int);
int raise(int sig);
```

`raise`	vyšle signál číslo `sig` procesu
`signal`	určuje jak bude přijatý signál zpracován

Definované konstanty:

`SIG_DFL`	nastaví implicitní obsluhu
`SIG_ERR`	indikuje chybu při návratu z funkce signal
`SIG_IGN`	ignoruje signál

Uživatelem specifikované obslužné funkce mohou končit `return` nebo voláním `abort`, `_exit`, `exit`, nebo `longjmp`.

Typy signálů

`SIGABRT`	abnormální ukončení
`SIGFPE`	chyba operace v plovoucí čárce (dělení nulou)
`SIGILL`	nelegální operace
`SIGINT`	ctrl-C
`SIGSEGV`	chyba přístupu k paměti
`SIGTERM`	požadavek ukončení programu

Poznámky:

- Problémy: nedefinované chování při signálu během obsluhy jiného signálu
- ISO C signály nevhodné – používat POSIX signály
- UNIX definuje cca 30 typů signálů

`stdarg.h` – funkce s proměnným počtem argumentů

`va_list` typ ukazatele na argumenty

Makra pro přenositelný přístup k argumentům:

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

- Makro `va_start` musí být použito jako první. Nastaví `ap` tak, aby ukazoval na první z volitelných argumentů funkce. Například:

  ```
  void f(int i, char *lastfix, ...);
  ```

- Jednotlivé hodnoty argumentů vrací makro `va_arg`. Druhý parametr `type` určuje typ argumentu. (Nelze použít typy `char`, `unsigned char`, nebo `float`.)
- Nakonec `va_end` uzavře zpracování argumentů.

Příklad: Součet seznamu čísel ukončeného nulou

```
int sum(int i1, ...)  {
   int total = i1;
   va_list ap;              // ukazatel
   int arg;
   va_start(ap, i1);
   while ((arg = va_arg(ap,int)) != 0) // 0=konec
      total += arg;
   va_end(ap);
   return total;
}

int main(void) {
   printf("Součet = %d\n", sum(1, 2, 3, 4, 0));
}
```

`stddef.h` – některá makra a datové typy

`ptrdiff_t`	rozdíl ukazatelů
`size_t`	velikost objektů
`wchar_t`	’široké’ znaky (UNICODE)
`NULL`	
`offsetof( typ, člen )`	

Příklady:

```
struct xx {
  int a, b;
};
size_t off = offsetof( struct xx, b );
size_t sz  = sizeof( struct xx );
int p[10];
ptrdiff_t dif = (char*)&p[10] - (char*)&p[0];
wchar_t c = L'H';
```

`stdio.h` – standardní vstup/výstup

- Typy:
  - `size_t` velikost objektů v paměti
  - `FILE` soubor
  - `fpos_t` pozice v souboru
- Makra:

  `NULL, EOF`
  `FOPEN_MAX, FILENAME_MAX`
  `SEEK_CUR, SEEK_END, SEEK_SET`
- Proměnné:

  `stdin, stdout, stderr`

`stdio.h` – standardní vstup/výstup

- Funkce:

```
remove, rename, tmpfile
fopen, freopen, fread, fwrite, fclose
fprintf, fscanf, printf, scanf, sprintf, sscanf
vfprintf, vprintf, vsprintf,
fgetc, fgets, fputc, fputs,
getc, getchar, putc, putchar, puts
ungetc,
fgetpos, fseek, fsetpos, ftell, rewind
clearerr, feof, ferror, perror
```

Poznámka: C99: orientace, `mbstate_t`, `fwide()`, `getwchar()`, `wprintf()`, ...

funkce `getchar`, `putchar`

```
int getchar(void);
```

čte jeden znak `stdin`. Narazí-li na konec souboru, vrací hodnotu `EOF`. Ekvivalent `getc(stdin)`.

```
int putchar(char c);
```

zapisuje jeden znak do `stdout`. Nelze-li zapsat, vrací `EOF`, jinak vrací `c`. Ekvivalent `putc(c,stdout)`.

Příklad: konverze na malá písmena (filtr)

```
#include <stdio.h>
#include <ctype.h>
int main() {
  int c;
  while((c=getchar())!=EOF)
    putchar(tolower(c));
}
```

Funkce `fgets`

```
char *fgets(char *s, int size, FILE *stream);
```

čte řádek ze souboru `stream`. Vrací `s` při úspěšném čtení, `NULL` při `EOF`. Znak `'\n'` přečte a uloží do `s` (na rozdíl od `gets`).

Funkce `gets` už není v ISO C11

```
char *gets(char *s);
```

Nepoužívat – nekontroluje délku vstupu!

Funkce `puts`

```
int puts(const char *s);
```

zapíše řetězec do `stdout` a přejde na nový řádek (na rozdíl od fputs). Vrací `EOF` při chybě, jinak vrací nezápornou hodnotu.

Funkce `printf` – formátovaný výstup do `stdout`

```
int printf(const char *fmt, ...);
```

`fmt`: řetězec obsahující formát tisku

```
%  - prefix formátu
d  - desítkově celé číslo
o  - oktalově celé číslo
x  - šestnáctkově celé číslo
u  - desítkově bez znaménka
c  - znak
s  - řetězec
e  - pohyblivá čárka s exponentem
f  - pohyblivá čárka bez exponentu
g  - kratší z %e nebo %f
```

Funkce `printf` – příklady

`%5.2lf`	5 míst, 2 desetinná, `long float`
`%10s`	min. délka 10
`%-10s`	zarovnání doleva
`%10.5s`	délka 10, tiskne pouze 5 znaků, zleva mezery
`%.10s`	tisk řetězce, je-li delší max. 10 znaků
`% d`	znaménko '`-`' nebo '` `' na začátku čísla
`%#g`	vždy desetinná tečka, ponechá koncové nuly
`%06x`	ponechá úvodní nuly: `-00001`

Funkce `scanf` – formátovaný vstup ze `stdin`

```
int scanf(const char *fmt, ...);
```

vrací počet úspěšně načtených formátů podle specifikace:

- znaky `' '`, `'\t'`, `'\n'` se ignorují
- jiné znaky se musí shodovat se vstupním textem
- % – prefix formátu
- `*` – potlačí přiřazení
- volitelně šířka pole

formát	vstup	argument
`d, ld`	desítkové číslo	`int*, long*`
`o, lo`	osmičkové číslo	`int*, long*`
`x, lx`	šestnáctkové číslo	`int*, long*`
`h`	short	`short*`
`c`	jediný znak	`char*`
`s`	řetězec	`char[], char*`
`f, lf`	reálné číslo	`float*, double*`

Funkce `scanf` – příklady

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %2s", &i, &x, name); /* & */
```

Vstup: `56789 0123 45a72`

Výsledek: `i = 56; x = 789.0; name = "45"`

Funkce *printf *scanf

`fprintf`, `fscanf` – tisk a čtení ze souboru

```
int fprintf(FILE *f, const char *fmt, ...);
int fscanf(FILE *f, const char *fmt, ...);
```

`sprintf`, `sscanf` – formátové konverze v paměti

```
int sprintf(char *s, const char *fmt, ...);
int sscanf(const char *s, const char *fmt, ...);
```

Poznámky:

- konzistenci parametrů a formátu zajišťuje programátor
- `scanf` vyžaduje ukazatele jako parametry
- `scanf`, `sprintf` — nebezpečí přepsání paměti
- implicitní konverze/rozšíření argumentů

Příklad:

```
int i;
char c;
scanf(" %d ", i );    /* pozor - chyba! */
scanf(" %d ", &c );   /* pozor - chyba! */
```

Práce se soubory

Funkce `fopen` – otevření souboru

```
FILE *fopen(const char *name, const char *mode);
```

`name` je jméno souboru, `mode` je režim otevření:

`"r"`	`"rb"`	čtení, čtení binárního souboru
`"w"`	`"wb"`	zápis
`"a"`	`"ab"`	přidávání na konec
`"wx"`		chyba pokud soubor už existuje

Poznámka: kombinace `"r+"`, `"w+b"`, `"rb+"` atd...
Při chybě vrací `NULL`

Příklad:

```
const char *name = "test.txt";
FILE *fp = fopen( name, "r" );
if( fp == NULL )
   error("soubor %s nelze otevřít pro čtení", name);
```

funkce `fclose` – uzavření souboru

```
int fclose(FILE *f);
```

Vrací `EOF` v případě chyby, jinak nulu

Příklad: zřetězení souborů na stdout (cat)

```
void filecopy(FILE *fp);
int main(int argc, char *argv[])  {
  FILE *fp;
  if( argc==1 )
    filecopy(stdin);
  else
    while( --argc > 0 )
      if( (fp=fopen(*++argv,"r")) == NULL ) {
        printf("cat: can't open %s \n", *argv);
        continue; /* další soubor */
      }
      else {
        filecopy(fp);
        fclose(fp);
      }
}
```

Příklad: `cat` – dokončení

```
void filecopy(FILE *fp) { /* neefektivní */
  int c;
  while( (c=getc(fp)) != EOF )
    putc( c, stdout );
}
```

`stdlib.h` – obecně použitelné funkce

`div_t, ldiv_t`	typy pro `div`, `ldiv`
`EXIT_FAILURE, EXIT_SUCCESS`	parametry `exit()`
`RAND_MAX`	maximum `rand()`
`MB_CUR_MAX`	max. počet bajtů v `'xxx'`

Funkce:

```
abs, div, labs, ldiv
bsearch, qsort,
rand, srand
atof, atoi, atol, strtod, strtol, strtoul
mblen, mbstowcs, mbtowc, wcstombs, wctomb
calloc, free, malloc, realloc
abort, atexit, exit, getenv, system
```

Převod řetězce na číslo

```
double atof( const char * s);
int atoi( const char * s );
long atol( const char * s );

double strtod( const char *s, char **endptr);
long strtol( const char *s, char **endptr, int base);
unsigned long strtoul(const char *s, char **e, int b);
```

`endptr` – ukazatel do řetězce po konverzi (není-li `NULL`)
`base` – základ číselné soustavy

Funkce `atexit`

Volání zadaných funkcí na konci programu

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

- registruje funkci pro zavolání před skončením programu (pořadí LIFO)
- dovoluje registraci min. 32 funkcí
- vrací nulu v případě úspěchu

funkce `exit` – zpracování chyb

```
void exit(int e);
```

ukončí program s návratovým kódem `e`

Funkce `atexit` – příklad

```
#include <stdio.h>
#include <stdlib.h>
FILE *fp;
void Close(void) { fputs("\nEXIT\n",fp); fclose(fp); }

int main(int argc, char *argv[])  {
  // test argc ...
  if( (fp=fopen(argv[1],"w")) == NULL ) {
    fprintf(stderr, "can't open %s \n", argv[1]);
    exit(1); // ukončení s chybou
  }
  atexit(Close);
  DoSomething(fp);       // může volat exit()
  // ukončení bez chyby - následuje exit(0);
}
```

Dynamické přidělování paměti

```
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void *calloc(size_t memb, size_t size);
void free(void *ptr);
```

`malloc`	přidělí paměť o zadané velikosti
`free`	uvolní přidělenou paměť
`calloc`	přidělí a nuluje paměť
`realloc`	zvětší/zmenší přidělenou paměť (může přesunout)

Příklad: `malloc` a `free`

```
#include <stdlib.h>
typedef struct prvek {
  int data;
  struct prvek *dalsi;
} prvek;

prvek *novy_prvek(void)  {
  prvek *p = malloc(sizeof(prvek));
  if( p == NULL ) /* pozor! musí se testovat! */
    error("chyba: málo paměti");
  return p;
}

void zrus_prvek(prvek *ptr)  {
  free(ptr);
}
```

`string.h` – práce s řetězci a pamětí

`strlen`	délka řetězce
`strcpy`	kopie řetězce
`strncpy`	kopie řetězce (max. n znaků)
`strcat`	připojení řetězce
`strncat`	připojení řetězce (max. n znaků)
`strcmp`	porovnání dvou řetězců
`strncmp`	porovnání dvou řetězců (max. n znaků)
`strcoll`	porovnání podle národní abecedy (`LC_COLLATE`)
`strchr`	vyhledání znaku
`strrchr`	vyhledání znaku od konce
`strstr`	vyhledání podřetězce

`size_t, NULL`

Práce s pamětí

`memcpy`	kopie bloků paměti
`memmove`	kopie překrývajících se bloků paměti
`memcmp`	porovnání bloků paměti
`memchr`	vyhledání bajtu
`memset`	nastavení bajtů
...	

Počet bajtů je dalším parametrem těchto funkcí.

Příklad:

```
void f(void) {
  int pole[SIZE];
  int pole2[SIZE];
  memset(pole, 0, SIZE * sizeof(pole[0]) );
  memcpy(pole2, pole, sizeof(pole2) );
}
```

`time.h` – práce s časovými údaji

`CLOCKS_PER_SEC`	počet tiků za sekundu
`clock_t`	systémový čas [tiky]
`time_t`	čas [s]
`struct tm`	`tm_sec`, `tm_mday`, ...

funkce:

`clock`	čas od začátku programu [tiky]
`difftime`	rozdíl časů v sec
`mktime`	převod na interní reprezentaci času
`time`	současná hodnota času [s]

`asctime, ctime, gmtime, localtime, strftime`

Příklad: Práce s časovými údaji

```
struct tm t;
t.tm_year = 1995 - 1900;    /* rok    od 1900 */
t.tm_mon  = 10 - 1;         /* měsíc  0..11   */
t.tm_mday = 6;              /* den    1..31   */
t.tm_hour = 12;
t.tm_min  = 0;
t.tm_sec  = 1;
t.tm_isdst = -1;            /* letní čas? */

if( mktime(&t) != -1 )      /* OK */
  if(t.tm_wday == 5)
    puts("pátek");
```

Přehled vybraných vlastností C99

Nové vlastnosti jazyka C podle normy C99:

- typ `long long` a související funkce
- negeneruje implicitní prototypy funkcí
- `inline` funkce (§6.7.4)
  Někdy `inline` definice vyžaduje další `extern` deklaraci v jediném modulu a není specifikováno, zda se funkce rozvine nebo zavolá.
  Pozor na velmi staré verze GCC (4.3+ a -std=c99 je O.K.)

```
inline int plus1(int a, int b) { return a+b; }
inline int plus2(int a, int b) { return a+b; }
int main(void) {  // použijte gcc -stdc=c99
    return plus1(1,2)+plus2(3,4); // chyba bez -O2
}
// extern int plus1(int a, int b); // +deklarace v modulu
```

- Strukturované literály: `(typ) { inicializace }`
- Automatická pole nekonstantních rozměrů (pouze ve funkcích)

```
T f(int n) {
    int pole[n]; // "variable-length array", VLA
    // ...
}

// následující prototypy jsou ekvivalentní:
T f(int n, int m, int a[n][m]);
T f(int n, int m, int a[*][*]);
T f(int n, int m, int a[ ][*]);
T f(int n, int m, int a[ ][m]);
```

viz ISO C99 §6.7.5.3 *Variably modified types*

- pole na konci struktury (”*Flexible array member*”)

  ```
  struct S { int n; T array[]; };
  ```

- Typ `_Bool` a makra v `<stdbool.h>` (konstanty `true`, `false` a typ `bool`).
- Komplexní čísla. V `<complex.h>` jsou definovány typy `complex`, `imaginary` a celá řada funkcí. Například:

  ```
  double complex cacos(double complex z);
  float complex cacosf(float complex z);
  long double complex cacosl(long double complex z);
  ```

  jsou definice funkce `acos`.

- `restrict` ukazatel je jedinou přístupovou cestou k objektu. Vhodné pro lepší optimalizaci; kvalifikátor `restrict` lze kdykoli vynechat bez změny významu.

Příklad:

```
void f (char * restrict s1,
        const char * restrict s2,
        int n) {
   while(n--)
      *s1++ = *s2++;
}
```

```
void g1 (int p[restrict][32]);
void g2 (int (* restrict p)[32]);
```

- Nové funkce: `snprintf`, `vscanf`, ...
- Identifikátor jména funkce: `__func__`
- Znakové literály `\uXXXX \UXXXXXXXX`
- Makra s proměnným počtem argumentů

```
#define debug(...)      fprintf(stderr, __VA_ARGS__)
#define showlist(...)   puts(#__VA_ARGS__)
```

- Standardní `#pragma` definice

```
#pragma STDC  co  jak
co  := {FP_CONTRACT|FENV_ACCESS|CX_LIMITED_RANGE}
jak := {ON|OFF|DEFAULT}
```

- Pragma operátor

```
#define M  _Pragma(string-literal) // v makrech
```

Standardní knihovny C99

soubor	co obsahuje
`complex.h`	podpora pro komplexní čísla
`fenv.h`	nastavení parametrů prostředí pro výpočty v plovoucí čárce (režim zaokrouhlování, zpracování výjimek)
`inttypes.h`	rozšíření definic celočíselných typů v `stdint.h` o makra pro formáty pro `printf` a deklarace funkcí `strtoimax`, `strtoumax`, `wcstoimax` a `wcstoumax`
`stdbool.h`	podpora typu `bool`
`stdint.h`	celočíselné typy – například `intmax_t`, `uintmax_t`, `int8_t`, `uint8_t`, `int_least32_t`, `int_fast32_t`, atd.
`tgmath.h`	typově generická makra

Standardní knihovny C99 – pokračování

soubor	co obsahuje
`iso646.h`	AMD1: podpora omezených znakových sad (`and` `==` `&&` atd.)
`wchar.h`	AMD1: podpora `wchar_t`, definuje například funkce: `fwprintf`, `swprintf`, `wcstol`, `wcscpy`, `wcsmemcpy`, atd.
`wctype.h`	AMD1: `ctype.h` pro typ `wchar_t` — definuje typy a funkce `iswdigit` atd.

- Generická makra v `<tgmath.h>`

`<math.h>` funkce	`<complex.h>` funkce	generické makro
acos	cacos	acos
asin	casin	asin
atan	catan	atan
cos	ccos	cos
sin	csin	sin
tan	ctan	tan
exp	cexp	exp
log	clog	log
pow	cpow	pow
sqrt	csqrt	sqrt
fabs	cabs	fabs
...	...	...

ISO-C11 — nové vlastnosti

- některé části normy jsou nepovinné (”optional”)
- podpora vláken (”threads”)
- možnost specifikovat zarovnávání (”alignment”)
- UNICODE znaky a řetězce (bylo už v revizi 2004)
- typově generické výrazy (`_Generic`)
- `static_assert(1 + 1 == 2, "chyba: 1+1 není 2");`
- funkce bez návratu (`noreturn void abort();`)
- anonymní struktury a unie
- výlučný přístup k souborům (`fopen("file","w+x");`)
- konečná likvidace funkce `gets`
- volitelná podpora pro kontrolu mezí a analýzu
- ...

Volitelné části normy C11

- `__STDC_ANALYZABLE__` hodnota 1 ⇒ podporuje přílohu L
- `__STDC_IEC_559__` 1 ⇒ příloha F (IEC 60559 floating-point arithmetic).
- `__STDC_IEC_559_COMPLEX__` 1 ⇒ příloha G (IEC 60559 compatible complex arithmetic).
- `__STDC_LIB_EXT1__` hodnota `201ymmL` ⇒ příloha K (Bounds-checking interfaces).
- `__STDC_NO_ATOMICS__` 1 ⇒ NEpodporuje atomické typy
- `__STDC_NO_COMPLEX__` 1 ⇒ NE komplexní čísla
- `__STDC_NO_THREADS__` 1 ⇒ NE `<threads.h>`
- `__STDC_NO_VLA__` 1 ⇒ NE pole s nekonstantní velikostí

complex.h

Volitelná část normy C11 (bylo povinné pro C99)
CMPLX makra vrací komplexní hodnotu $x + iy$

```
#include <complex.h>

double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
```

Poznámka: Pokud má makro `__STDC_NO_COMPLEX__` hodnotu 1, implementace nepodporuje komplexní čísla a `complex.h` neexistuje.

Standardní knihovny přidané v C11

soubor	co obsahuje
`stdalign.h`	`alignas`, operátor `alignof`, ...
`stdatomic.h`	atomické operace
`stdnoreturn.h`	specifikace `noreturn`
`threads.h`	`thread_local`, funkce, mutex, cvar,...
`uchar.h`	`char16_t`, `char32_t`, funkce

\+ doplňky v ostatních hlavičkových souborech

ISO-C23 — nové vlastnosti

- `true`, `false`, `bool`, ... jsou klíčová slova.
- podpora `nullptr`
- podpora `auto` (jen pro inferenci typů proměnných)
- operátor `typeof(expr)`
- volitelně `_Decimal64` atd.
- inicializace nulami `{}` (včetně VLA)
- Preprocesor: `#embed`, `#warning`, `__has_include`, ...
- povinné *variably-modified types* (parametry fcí, ne VLA)
- `enum E : long { .... }`
- literály se separátory `1.123'456'789`
- binární literály `0b10101010`, `%b` formát
- speciální celočíselné typy `_BitInt(N)`
- `constexpr` (jen pro proměnné)
- nová syntaxe pro atributy: `[[noreturn]]`

Standardní knihovny doplněné do C23

soubor	co obsahuje
`stdbit.h`	makra pro počítání bitů v int atd.
`stdckdint.h`	makra pro operace $+ - *$ s kontrolou přetečení

\+ doplňky v ostatních hlavičkových souborech

Ladění programů

Při programování máme dvě možnosti:

- psát programy bez chyb
- nebo se naučit hledat a odstraňovat chyby v programech (debugging)
  - ladění je časově náročné
  - je třeba se učit z vlastních i cizích chyb
  - vliv použitého programovacího jazyka

Prevence chyb

Techniky pro omezení chyb:

- dobrý návrh programu (rozhraní, ...)
- dobrý styl psaní programů
- důkladné testování okrajových podmínek, assert(), ...
- omezení globálních dat
- používání nástrojů pro kontrolu správnosti programů (lint, valgrind, electric fence, ...)

Poznámka: Pozor na chyby způsobené nepochopením kódu

Ladicí program (*debugger*)

- interaktivní nástroj vhodný pro zjištění stavu
- možnosti ladicích programů:
  - krokování programu
  - výpis obsahu paměti, registrů, ...
  - výpis stavu zásobníku (stack-trace)
  - breakpoint (podmíněný, HW)
  - watchpoint (podmíněný, HW)
  - automatické spuštění při výskytu problému
  - post-mortem ladění: `core`
  - připojení k již běžícímu procesu
  - vzdálené ladění
  - ...
- nejsou vždy dostupné (embedded, některé jazyky, ...)
- problémy: paralelní procesy/vlákna, operační systémy, distribuované systémy

Závěr: ne vždy vhodné

Techniky ladění programů

- Použití ladicích programů.
- Výpisy (log-file) a jejich analýza (výhody: součást programu, lze zapnout/vypnout, někdy efektivnější).

Zkoumání možných příčin chyb – doporučení

jednodušší případy:

- hledání obvyklých chyb: chybějící & v `scanf`, chybné formáty `printf`/`scanf`, a podobné (nástroje: `gcc -Wall`, program `lint`, ...)
- soustředit se na poslední změny kódu
- neopakovat stejné chyby, opravit *všechny* výskyty chyby
- neodkládat ladění na pozdější dobu (příklad: Mars Pathfinder – reset)
- sledovat ”stack-trace”– hodnoty parametrů funkcí
- nedělat unáhlené opravy
- vysvětlit svůj kód někomu jinému (i když tomu nerozumí)

složitější případy:

- zajistit reprodukovatelnost chyby
- sledovat četnost výskytů chyby (například každých X znaků je chyba – asi problém +-1 – prohledat kód na výskyt konstant v okolí hodnoty X)
- kontrolní výpisy (log-file), grep
- používání nástrojů: grep, diff, awk, ...
- přidávat testy do kódu
- vizualizace výstupů programu (graf)
- vytvářet testovací případy; redukce kódu s chybou na minimum
- zaznamenávat průběh ladění v případě dlouhotrvajících problémů

nejsložitější případy:

- zkontrolovat lehce přehlédnutelné chyby

```
if (a & 1 == 0) {  /* <<<< priorita */
    /* nikdy se neprovede ... */
}
switch (x) {
    /* ... */
  defaut:           /* <<<< překlep */
    /* nikdy se neprovede ... */
}
memset(p, n, 0);   /* <<<< chybné pořadí argumentů */

DPRINTF(("výpis %d\n", i++)); /* <<<< co až zmizí? */
```

- nehledat chyby, které nejsou (vzniknou například chybným testováním)
- zkontrolovat uvolňování zdrojů (memory leaks, file desc.)
- velmi zřídka může být chyba v překladači, knihovnách, OS, HW (Příklad: `isprint(getchar())` a dvojí vyhodnocení)

Nereprodukovatelné chyby

- zkontrolovat inicializaci proměnných
- pokud po přidání testovacího kódu chyba zmizí – možná chyba alokace paměti (i `printf` může alokovat paměť a způsobit změnu chování)
- když kód vypadá správně – nějaká jiná část programu přepisuje paměť (neinicializovaný ukazatel, vrácení odkazu na lokální proměnnou, použití dynamicky alokované paměti po jejím uvolnění, dvakrát free, ...)
  Nástroje: speciální malloc/free – knihovny (dmalloc, efence, ...), program valgrind, ...
- chyba v programu závisí na prostředí:
  kontrola nastavení proměnných prostředí (např. LANG), přístupová práva, konfigurace systému, ...
  (Příklad: MSDOS: ctrlZ je EOF — nefunguje stdin s bin. daty)

Ladění cizího kódu

- Je chyba opravdu v cizím kódu?
- Není chyba už známa nebo opravena v nové verzi? (bugzilla, ...)
- Vytvořit minimální příklad chyby a nahlásit

Nástroje pro ladění

- debuggery: GDB, DDD, Insight, ...
- výpisy informací o činnosti procesu:

strace, ptrace	komunikace s jádrem systému
ltrace	volání funkcí sdílených knihoven
ps, top	stav procesu
script	záznam výstupů na terminál
lsof	výpis otevřených souborů
fuser	procesy pracující se souborem

- výpisy informací o programu:

nm	symboly v modulech/knihovnách
size	velikost kódu, dat, ...
ldd	které dyn. knihovny jsou použity
strings	textové řetězce v bin. souboru
od, hexdump	obsah souboru (různé formáty)
objdump	obsah modulu/programu (disassembly,...)

- pomocné textové nástroje: grep, cut, sort, diff, cmp, comm

Testování programů

- Systematické pokusy o vyvolání chyby testovaného programu.
- Souvisí s laděním, ale není to totéž.
- Může odhalit chyby, ale nemůže dokázat jejich nepřítomnost.

Testování v průběhu psaní programu

- testujte mezní případy:
  - vstup odpovídající velikosti pole +-1
  - prázdný vstup
  - obrovský vstup
  - speciální znaky na vstupu
- testujte pre- a post-conditions: záporné hodnoty nebo nula tam, kde se očekává kladné číslo (příklad USS Yorktown: dělení nulou)
- defenzivní programování — testovat i ”případy které nikdy nenastanou”
- testujte návratové hodnoty funkcí (malloc, fopen, ...)

Systematické testování

- testujte kód v průběhu implementace
- testujte jednoduché části jako první
- je třeba znát, jaký výstup lze očekávat
- ověřujte invariantní vlastnosti (počty zpracovaných záznamů, kontrolní součty, ...)
- porovnejte nezávislé implementace stejných algoritmů
- ověřte, zda testujete všechny varianty kódu

Automatizace testů

- automatické regresní testy – porovnání s předchozí verzí
- nezávislé testy – obsahují vstupy a očekávané výstupy
- ...

Poznámka: ”unit testing”, ”code coverage analysis”

Tipy pro testování

- vždy testujte své programy
- upravte kód tak, aby se ověřilo ošetření chyb (speciální alokátor paměti vracející chybu, zmenšení velikostí polí, ...) inicializujte pole a proměnné speciálními hodnotami
- testujte v různých prostředích (jiný OS, překladač, HW)
- umožněte regulaci množství testovacích výpisů (`--verbose`)
- před odevzdáním vypněte testovací kód (`#define NDEBUG`)

Úvod do C++

Historie

C with classes	(1981)	Bjarne Stroustrup, Bell Labs
ISO C++	(1998)	první norma C++98
ISO C++	(2011)	C++11
ISO C++	(2014)	C++14
ISO C++	(2017)	C++17
ISO C	(2018)	aktuální norma C18
ISO C++	(2020)	aktuální norma C++20

Původní definicí je kniha *Stroustrup: The C++ Programming Language (Addison-Wesley 1985, 1991, 1997, 2013)*.
Platí norma *ISO/IEC 14882:2020* (neformálně: C++20).

Poznámka: Překladače a vývojová prostředí pro jazyk C++ viz WWW.

Charakteristika jazyka C++

- Obecně využitelný programovací jazyk vyšší úrovně.
- Je standardizovaný (ISO/ANSI)
- Podporuje abstraktní datové typy a vytváření knihoven.
- Nástupce jazyka C (zpětná kompatibilita)
- Efektivita
- Objektová orientace (třídy, dědičnost)
- Možnost přetěžovat operátory
- Generické třídy a funkce (šablony)
- Obsluha výjimek
- Mnoho různých implementací překladačů
- Množství prostředků pro různé aplikace (GUI, ...)

Nevýhody C++

- Je *podstatně* větší a složitější než C
- Není čistě objektově orientovaný (např. typ int není třída a nelze z něj dědit)
- Zdědil některé problémy jazyka C (indexování polí se nekontroluje, manuální správa paměti, ...)
- Není zcela kompatibilní s jazykem C
- Ne všechny překladače dostatečně vyhovují normě.

Poznámka: Citace z Internetu: ”There are only two kinds of programming languages: those people always bitch about and those nobody uses.”

Příklady – překlad a sestavení programu

Soubor `ahoj.cc`:

```
#include <iostream>
int main() {
  std::cout << " Ahoj! \n";  // tisk řetězce
}
```

Způsob zpracování (UNIX, překladač GNU C++):

```
g++ -o ahoj ahoj.cc       # překlad, sestavení
./ahoj                    # spuštění
```

Poznámka: Přípony `.cc`, `.cpp`, `.C`, `.c++`, `.cxx`

Optimalizace a ladění (UNIX, GNU C++):

```
g++ -O2 -o prog prog.cc   # +optimalizace
g++ -g -o prog prog.cc    # +ladicí informace
gdb prog                  # ladění
```

Příklad: čtení a tisk C++ řetězce

```
#include <iostream>
#include <string>       // std::string

int main() {
  using namespace std; // === nemusíme psát std::
  cout << "C++ string" << endl;
  string s;
  cout << "s = " << s << endl;
  cout << "string s (libovolná délka): " << flush;
  cin >> s ;  // sledujte jak funguje (čte slova)
  cout << "s = " << s << endl;
}
```

Příklad: čtení a tisk C řetězce v C++ (nevhodné)

```
// hrozí chyba typu "buffer overflow" [NEPOUŽÍVAT]

#include <iostream>
#include <string>

using namespace std;   // nemusíme psát std::

int main() {
  cout << "C string" << endl;
  char s[100] = "";
  cout << "s = " << s << endl;
  cout << "string s (max 99 zn): " << flush;
  cin >> s ; // čte slovo, pozor na "buffer overflow"
  cout << "s = " << s << endl;
}
```

Příklad: četnost slov

```
// g++ -std=c++11 (nebo c++0x)
#include <iostream>
#include <string>
#include <map>          // kontejner std::map

int main() {
  std::string word;
  std::map<std::string,int> m;  // asociativní pole
  while( std::cin >> word )     // čte slova
    m[word]++;
  for(auto x: m)                // iterace a tisk
    std::cout << x.first <<"\t"<< x.second <<"\n";
}
```

Příklad: četnost slov (zastaralé, C++98)

```
#include <iostream>
#include <string>
#include <map>          // kontejner std::map

typedef std::map<std::string,int>  map_t;
typedef map_t::iterator            mapiter_t;

int main() {
  std::string word;
  map_t m; // asociativní pole
  while( std::cin >> word ) // čte slova
    m[word]++;
  for(mapiter_t i=m.begin(); i!=m.end(); ++i) // tisk
    std::cout << i->first <<"\t"<< i->second <<"\n";
}
```

Příklad: řazení čísel, iterator

```
// g++ -std=c++11
#include <iostream>    // cout
#include <vector>      // vector
#include <algorithm>   // sort, copy
#include <iterator>    // ostream_iterator

int main() {
  using namespace std;
  vector<int> v{4, 2, 5, 1, 3};  // inicializace
  sort(begin(v), end(v));        // řazení
  ostream_iterator<int> o(cout,"\n"); // iterátor
  copy(begin(v), end(v), o);     // tisk
}
```

Příklad: řazení čísel — varianta 2

```
#include <vector>       // vector
#include <algorithm>    // sort, copy
#include <iostream>     // cout
#include <iterator>     // ostream_iterator

int main() {
  using namespace std;
  vector<int> v;
  int i;
  while ( cin >> i )    // čtení čísel ze vstupu
    v.push_back(i);     // vektor se zvětšuje
  sort(v.begin(), v.end());       // řazení
  ostream_iterator<int> o(cout, "\n");
  copy(v.begin(), v.end(), o);    // tisk
}
```

Příklad: lambda funkce

```
// g++ -std=c++11
#include <iostream>    // cout
#include <vector>      // vector
#include <algorithm>   // sort, copy, for_each

int main() {
  using namespace std;
  vector<int> v{ 4, 2, 5, 1, 3 };  // inicializace
  sort(v.begin(), v.end(),
       [](int x, int y){return x>y;}); // řazení
  int a = 10;                          // lokální objekt
  for_each(v.begin(),v.end(),
           [&a](int &x){ x += a++; }); // "closure"
  for(auto x: v)                       // tisk
    cout << x << endl;
}
```

Rozdíly mezi C a C++

- C++ vychází z jazyka C11 (C++98 z C90, C++11 z C99)
- Dobře napsané C programy jsou též C++ programy (s několika výjimkami: nová klíčová slova, povinné prototypy funkcí, silnější typová kontrola, ...)
- Rozdíly mezi C a C++ jsou zjištěny překladačem kromě několika málo výjimek:
  - Znakové literály jsou typu `char`

```
sizeof('a') == sizeof(int)  // C
sizeof('a') == sizeof(char) // C++
```

- ...
  - Výčtový typ není ekvivalentní typu `int`

```
enum e { A };
sizeof(A) == sizeof(int) // C
sizeof(A) == sizeof(e)   // C++ != sizeof(int)
```

  - Jméno struktury v C++ může překrýt jméno objektu, funkce, výčtu nebo typu v nadřazeném bloku:

```
int x[99];
void f()  {
  struct x { int a; };
  sizeof(x); /* pole v C, struktura v C++ */
}
```

Rozšíření C++ proti C99

- typ reference
- anonymní unie (jsou v C11)
- přetěžování funkcí a operátorů
- operátory `new`, `delete`, `new[]` a `delete[]`
- třídy (`class`), abstraktní třídy
  - automatická inicializace (konstruktory, destruktory)
  - zapouzdření (`private`, `public`, `protected`)
  - dědičnost, násobná dědičnost
  - polymorfismus (virtuální funkce)
  - uživatelem definované konverze

Rozšíření C++ proti C99 – pokračování

- jméno třídy a výčtu je jméno typu
- ukazatele na členy tříd
- v inicializaci statických objektů je dovolen obecný výraz
- generické datové typy – šablony (`template`, `typename`)
- obsluha výjimek (`try`, `catch`, `throw`)
- prostory jmen (`namespace`, `using`)
- nové způsoby přetypování (`static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`)
- informace o typu za běhu programu (`typeid`, `type_info`)
- klíčové slovo `mutable`

Novinky v C++11 proti C++98

- R-hodnotové reference, ”*move*” konstruktory
- `constexpr`
- Změny ve specifikaci dat ”POD = *Plain Old Data*”
- `extern template`
- Inicializační seznamy
- Sjednocený zápis inicializace (`int a{5};`)
- Inference typů (auto,decltype)
- for cyklus přes rozsah kontejneru
- Lambda funkce a výrazy
- Alternativní syntaxe funkcí
  `[]fce(int x)->int{return x;}`
- Speciální identifikátory `override`, `final`
- `nullptr`

Novinky v C++11 – pokračování

- Nové, silně typované výčty
- `template<vector<T>>` (problém v C++98)
- `explicit` pro konverzní operátory
- `alias` šablony
- Šablony s proměnným počtem parametrů (”variadic templates”)
- Nové řetězcové literály
- Uživatelem definované literály (10kg, 12345bignum)
- Podpora vláken (`thread_local`, ...)
- `default` a `delete` konstruktory atd.
- `long long`
- `static_assert`
- Možnost implementovat ”garbage collector”

C++14

Malé změny a zrušení různých omezení:

- `auto f() { .... return x; }`
- `auto` parametry lambda funkcí
- šablony proměnných
- C99 inicializace `{ .field=value, .... }`
- binární literály `0b0110`
- oddělovače v numerických literálech `10'000'000`
- vylepšení a rozšíření std knihovny
- ...

C++17

Řada menších změn jazyka:

- zrušení "*trigraphs*"
- float literály v šestnáckové soustavě
- lepší optimalizace
- ...

Vylepšení a rozšíření std knihovny:

- práce se soubory a adresáři (viz `boost::filesystem`)
- `std::string_view`
- Další matematické funkce
- `std::variant`
- `std::byte`
- ...

Poznámky: Závislost na C11, podpora C++17 v překladačích

C++20

Jazyk:

- Moduly (*modules*): `import`, `module` (identifikátory), `export`
- Korutiny (*coroutines*): `co_await`, `co_return`, `co_yield`
- Koncepty (*concepts*): `concept`, `requires`
- (*three-way comparison operator*): `operator <=>`
- Funkce prováděné při překladu (immediate functions): `consteval`
- Inicializace proměnné při překladu: `constinit`

Knihovna: rozsahy (*ranges*), `std::span`, `std::format`, ...
Makra: `__has_cpp_attribute(a)`, `__cpp_concepts`, ...

C++

Poznámky

```
/* text poznámky */
// text poznámky platí až do konce řádku
```

Vyhrazené identifikátory

- vše co obsahuje dvojité podtržení __ na libovolné pozici
- vše co začíná podtržením _ následovaným velkým písmenem
- globální symboly začínající podtržením _

Speciální identifikátory v C++11

`override`
`final`

Klíčová slova C++

alignas *	decltype *	namespace	struct
alignof *	default	new	switch
and	delete	noexcept *	template
and_eq	do	not	this
asm	double	not_eq	thread_local*
auto	dynamic_cast	nullptr *	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	(register)	unsigned
char16_t *	for	reinterpret_cast	using
char32_t *	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
const_cast	int	static	while
constexpr *	long	static_assert *	xor
continue	mutable	static_cast	xor_eq

Alternativní reprezentace

`<%`	`{`	`and`	`&&`	`and_eq`	`&=`
`%>`	`}`	`bitand`	`&`	`bitor`	`|`
`<:`	`[`	`compl`	`~`	`not`	`!`
`:>`	`]`	`not_eq`	`!=`	`or`	`||`
`%:`	`#`	`or_eq`	`|=`	`xor`	`^`
`%:%:`	`##`	`xor_eq`	`^=`		

Poznámka: Digraphs (`<:`)

Poznámka: Trigraphs (`??/`) (zrušeno v C++17)

```
// Provede se následující příkaz??/
i++;
```

Literály

Syntaxe číselných literálů je stejná jako v C.

C++11: uživatelem definované literály (`operator ""`)
Příklad:

```
BigNumber operator "" _big(const char * literal_string);
BigNumber some_variable = 12345_big;
```

Znakové literály jsou typu:

- `char` v C++
- `int` v C, v C++ pouze víceznakové (mbc)

Poznámka: V C++ existují 3 různé znakové typy: `char`, `unsigned char` a `signed char`

Typová kontrola

Je v C++ silnější než v C:

- Deklarace:

  ```
  void (*funptr)();
  ```

  je ukazatel na fci vracející `void` v C, ukazatel na fci vracející `void` bez parametrů v C++
- Ukazatel na konstantní objekt nelze přiřadit do ukazatele na nekonstantní objekt.
- Typová kontrola při sestavování programu rozliší funkce s různými parametry

Výčtové typy:

- lze přiřadit pouze konstantu daného typu
- lze vynechat klíčové slovo `enum` při použití
- `sizeof` výčtového typu závisí na hodnotách prvků

Poznámky

- Rozsah deklarace proměnné cyklu ve `for`

```
for(int i=1; i<10; i++) {
   // zde platí i (jako v C99+)
}
```

- Je chybou, když je příkazem skoku přeskočena inicializace proměnné (překladač to kontroluje).
- Pozor na `setjmp` a `longjmp` v C++
- Pozor na pořadí inicializace globálních/statických objektů

Typ reference

Definice:

```
T & x = Lhodnota_typu_T;
```

- Blízké ukazatelům (ale neexistuje obdoba NULL)
- Použitelné pro předávání parametrů odkazem
- Nelze vytvořit:
  - referenci na referenci (např. `T & & r`),
    Pozor: v šablonách je dovoleno, ale jen nepřímo
  - referenci na bitová pole,
  - ukazatele na reference,
  - pole referencí.

Výhodou referencí je jednoduchost použití (na rozdíl od `*ptr`)

Poznámka: R-hodnotové reference `Typ &&` (C++11)

Typ reference – příklady

```
double x = 1.23456;
double & xref = x;  // Typické použití

double & yref;      // CHYBA! chybí inicializace
extern int & zref;  // extern může být bez inicializace

// Předání parametru odkazem:
void Transpose(Matrix & m);

// Vracení reference:
int & f(param);     // Pozor na to _co_ se vrací!

f(p) = 1;           // Volání funkce a použití výsledku
```

Reference — chybná nebo netypická použití

```
const int & i = 7;  // Vytvoří pomocnou proměnnou
int & i = 7;        // CHYBA! nelze pro nekonst. referenci

float f = 3.14;
const int & ir = f; // pomocná_proměnná = 3
ir = 5;             // CHYBA! konstantu nelze změnit
```

Poznámka:

Proč nelze použít R-hodnotu s nekonstantní referencí:

```
void incr( int& refint )  { refint++; }
void g() {              // Pozor - toto není C++
    double d = 1;
    incr(d);            // Záludná chyba: nezmění d !
}
```

Operátory C++ podle priority (zjednodušeno)

operátory	asociativita
`( ) [ ] -> :: .`	→
`! ~ + - ++ -- & * (Typ) sizeof new delete`	←
`.*  ->*`	→
`* / %`	→
`+ -`	→
`<< >>`	→
`< <= > >=`	→
`== !=`	→
`&`	→
`^`	→
`|`	→
`&&`	→
`||`	→
`?:`	←
`= *= /= %= += -= &= ^= |= <<= >>=`	←
`,`	→

Operátory C++

operátor	popis
`::`	kvalifikátor
`.*`	dereference ukazatele na člen třídy přes objekt
`->*`	dereference ukazatele na člen třídy přes ukazatel na objekt
`new`	dynamické vytvoření objektu
`delete`	zrušení objektu
`static_cast`, `reinterpret_cast`, `const_cast`, `dynamic_cast`	nové operátory přetypování

Operátory — příklady

```
// alokace paměti operátorem new:
T *p = new T[10*n];     // dynamická alokace pole
T *p2 = new T(5);       // dynamická alokace objektu

// uvolnění paměti operátorem delete:
delete [] p;            // uvolnění paměti pole
delete p2;              // uvolnění paměti objektu

// alokace a rušení pole bajtů:
char *s = new char[100];
delete [] s;
// char *s2 = static_cast<char*>(std::malloc(100));
// std::free(s2);
```

Operátor : :

- Přístup ke globální proměnné:

```
double x;
void f() {
   int x;                 // lokální x
   ::x = 3.1415926;       // globální x
}
```

- Explicitní specifikace třídy:

```
class T {
  public:
    int metoda();         // deklarace metody
};
int T::metoda() { }       // definice mimo třídu
```

- Specifikace prostoru jmen:

```
prostor::identifikátor                    std::cin
prostor::podprostor::identifikátor
```

Standardní konverze v C++

Implicitní konverze probíhají automaticky (jsou-li nutné) při vyhodnocování binárních operací:

- Každý ’malý’ celočíselný typ se konvertuje takto:

typ	konverze na	metoda
`char`	`int`	podle nastavení
`unsigned char`	`int`	doplní nuly
`signed char`	`int`	rozšíří znaménko
`short`	`int`	stejná hodnota
`unsigned short`	`unsigned int`	stejná hodnota
`enum`	`int`	stejná hodnota
`bool`	`int`	0 nebo 1

Potom je každá hodnota operandu buď `int` (včetně `long` a `unsigned` modifikátorů) `double`, `float` nebo `long double`.

Standardní konverze v C++ — pokračování

1. Je-li některý operand `long double`, je druhý konvertován na `long double`
2. Jinak, je-li operand `double`, konvertuje druhý na `double`
3. Jinak, je-li operand `float`, konvertuje druhý na `float`
4. Jinak, je-li operand `unsigned long`, konvertuje druhý na `unsigned long`
5. Jinak, je-li operand `long`, konvertuje druhý na `long`
6. Jinak, je-li operand `unsigned`, konvertuje druhý na `unsigned`
7. Jinak, jsou oba operandy typu `int`

Výsledek odpovídá typu obou operandů po konverzi.

Standardní konverze — příklady

Poznámka:
Při porovnávání čísla `int` s číslem `unsigned` může dojít k (pro někoho neočekávaným) problémům:

```
int i = -1;
unsigned u = 1234;

if(i<u)                    // sledujte varování překladače
   printf(" i < u ");  // nevytiskne nic!
```

Příklad z praxe:

```
unsigned long f(unsigned a, unsigned b) {
    return a*b;  // CHYBA! 32b*32b->32b -> 64b
}
```

Explicitní konverze

Explicitní konverze uvádí programátor do textu programu (a přebírá za ně veškerou odpovědnost):

```
(typ) výraz
typ(výraz)
static_cast<typ>(výraz)
```

Příklady: Explicitní přetypování v C++

```
double(int1)/int2
complex(3.14)
int('c')
static_cast<char*>(ptr)
reinterpret_cast<long>(ptr)
```

extern "C"

Možnost použití funkcí z knihoven jazyka C, případně jiných jazyků (ASM, FORTRAN):

```
extern "C" int f(int);

extern "C" {
    int g(int);
    int h(int);
    // ...
}
```

Poznámky:
Prostory jmen (`namespace`)

Preprocesor

Je stejný jako v ISO C, je vhodné minimalizovat jeho používání:

- `#define K1 10` lze nahradit za:

```
const int K1 = 10;
```

- `#define f(x) (výraz_x)` lze většinou nahradit za:

```
inline int f(int x) { return výraz_x; }
```

případně lze použít generické funkce:

```
template<typename T>
inline T f(T x) { return výraz_x; }
```

Makro `__cplusplus` definováno překladačem (`202002L`)

Knihovny a sestavování programů

Základní koncepty

knihovna = množina přeložených modulů v jednom souboru
(přípony: `*.a`, `*.lib`, `*.so`, `*.dll`)

object file = přeložený (binární) modul
(přípony: `*.o`, `*.obj`)

sestavování programu = vytvoření spustitelného souboru spojením modulů a knihoven (program volá funkce z knihoven/jiných modulů a ty musí být dostupné *nejpozději v okamžiku volání*)

Typy sestavení:

statické – při vytváření programu

dynamické – při/po spuštění programu

Poznámky: relokace, Position Independent Code (PIC), linker, loader

Formáty souborů

- COFF, PE (Windows), ELF (POSIX), ...
- Typická struktura souboru:
  - hlavička – typ, obsah
  - sekce `.text` - kód
  - sekce `.rodata` - konstanty
  - sekce `.data` - inicializovaná data,
  - sekce `.debug*` - ladicí informace
  - sekce `.rel*` - relokační informace
  - sekce `.plt` - tabulka pro dyn. sestavení
  - ...

Poznámka:
Výpis obsahu souborů: programy `objdump`, `nm`, `readelf`

Vytváření statických knihoven

- Statické knihovny obsahují kód modulů a index pro zrychlení sestavování
- Vytvářejí se speciálním programem (librarian). POSIX používá program `ar` – archivář.
  1. překlad modulů:
     ```
     cc -c  moduly.c
     ```
  2. vytvoření knihovny:
     ```
     ar  parametry  knihovna.a  moduly.o
     ```
     Pozor – parametry jsou důležité!
  3. vytvoření indexu:
     ```
     ranlib  knihovna.a
     ```

Vytváření sdílených knihoven

- obsahují kód modulů a tabulky odkazů
- vytvářejí se překladačem,
  1. `cc -c -fPIC moduly.c`
  2. `cc -shared -fPIC moduly.o -o knihovna.so`
- mají příponu `.so` (shared object)
  nebo `.DLL` (Dynamically Linked Library)
- musí obsahovat kód nezávislý na umístění v paměťovém prostoru (PIC)
- ELF: volání funkcí přes PLT (Procedure Linkage Table)

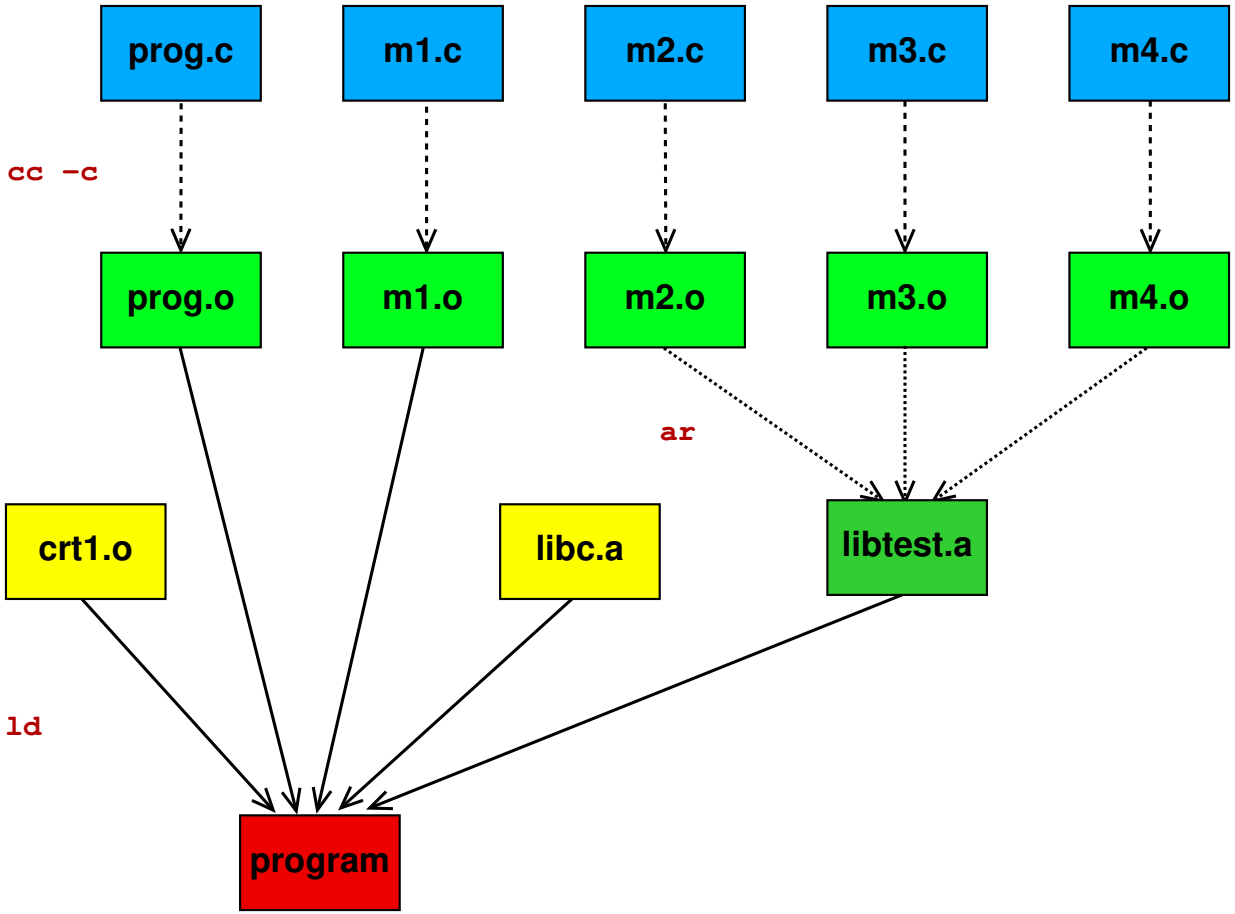
Poznámka: libtool

Sestavování programu

- překladač (compiler):
  - vloží do přeloženého modulu informace potřebné pro sestavení (exportované symboly, nedefinované symboly, relokační informace, ...)
- sestavovací program (linker):
  - umístí kód modulů do adresového prostoru (někdy je nutná relokace),
  - přepíše odkazy skutečnými adresami funkcí (pokrytí odkazů)
- zavaděč (loader):
  - při spuštění programu zajistí jeho načtení do paměti a případné dynamické sestavení

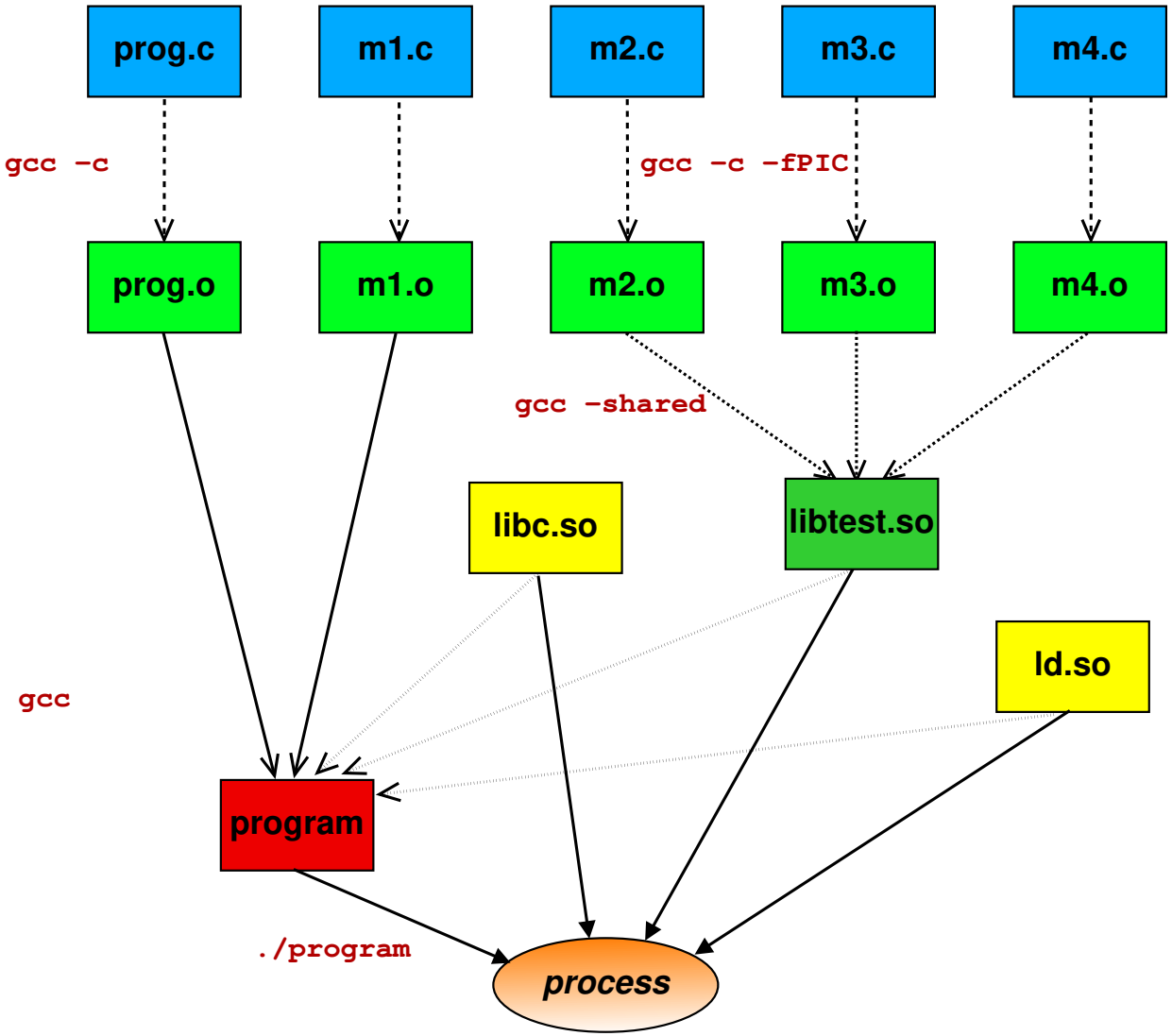
Statické sestavení

- výsledný program obsahuje kód všech funkcí
- typicky se sestavují celé moduly z knihoven
- Sestavovací programy:
  - `gcc -static` (implicitně sestavuje dyn.)
  - `ld` (POSIX),
  - `tlink` (BC), ...
- Výhody:
  - nezávislost programu na knihovnách (jeden soubor)
- Nevýhody:
  - větší spustitelné soubory,
  - nelze sdílet kód knihoven


prog.c
m1.c
m2.c
m3.c
m4.c
cc -c
prog.o
m1.o
m2.o
m3.o
m4.o
ar
crt1.o
libc.a
libtest.a
ld
program

Dynamické sestavení

- program neobsahuje kód funkcí,
- k sestavení dojde *až po spuštění programu*
- `ld.so` = dynamic linker/loader
- Výhody:
  - možnost nezávislé aktualizace knihoven,
  - menší programy,
  - sdílení kódu knihoven
- Nevýhody:
  - pomalejší start procesu,
  - závislost programu na dalších souborech
  - závislost na verzích knihoven


prog.c
m1.c
m2.c
m3.c
m4.c
gcc -c
gcc -c -fPIC
prog.o
m1.o
m2.o
m3.o
m4.o
gcc -shared
libc.so
libtest.so
ld.so
gcc
program
./program
process

Použití knihoven

Příklad: Program s knihovnou `libtest.a`

```
gcc -o program -static m1.c m2.c -L. -ltest
```

Příklad: Program s knihovnou `libtest.so`

```
gcc -o program m1.c m2.c -L. -ltest
```

Poznámky:

- Pozor na umístění knihoven!
- Lze kombinovat statické i dynamické
- Pořadí argumentů je významné (POSIX)
  `cc prog.c library1.a library2.a library1.a`
  (z knihoven se vybírají jen potřebné symboly, při cyklických závislostech musíme knihovnu uvést dvakrát)
- Problémy: DLL hell, nekompatibilita verzí knihoven, ...

Použití knihoven – pokračování

- Program `ldd(1)` – výpis sdílených knihoven
- Umístění sdílených knihoven v adresářové struktuře (`/lib`, `/usr/lib`, ...)
- Jména souborů se sdílenými knihovnami:

```
/usr/lib/libJMENO.so.7.8.9  -- real name
/usr/lib/libJMENO.so.7      -- so name (symlink)
/usr/lib/libJMENO.so        -- linker name (symlink)
```

- Parametrizace dynamického sestavování:
  `LD_LIBRARY_PATH` – adresáře, ve kterých jsou hledány sdílené knihovny
  `LD_PRELOAD` – možnost nahradit některé funkce jinými z přednostně sestavených knihoven
  (další informace viz `man ld.so`)
- Zrychlení sestavování: `/etc/ld.so.cache`, `ldconfig(8)`

Program `make`

- řeší závislosti při vytváření programů
- podle zadaných pravidel uspořádá provádění akcí
- použije se čas poslední změny souboru:
  pokud (čas cíle $<$ čas některého zdroje), provede akci,
  která vytvoří nový cílový soubor

Příklad: Pravidla v souboru Makefile:

```
program: modul1.o modul2.o
modul1.o: modul1.c
modul2.o: modul2.c
```

Poznámky: GNU make: implicitní pravidla a akce

```
make CFLAGS=-O2
```

Příklad

Použití proměnných

```
CFLAGS   = -O2 -Wall -std=c99   # ?=
CXXFLAGS = -g
PROGS    = program1 program2

all: $(PROGS)
program1: program1.c
        $(CC) $(CFLAGS) -o $@  $<
program2: program2.c
        $(CC) $(CFLAGS) -o $@  $<
clean:
        rm -f $(PROGS) ### pozor na <TAB>
```

Implementace
GNU make, BSD make

Alternativy k `make`
Jam, ...

Automatické generování závislostí
`gcc -MM *.cc >depend`
vložení do Makefile: `-include depend`

Programy pro generování Makefile

- GNU Autoconf
  `./configure; make; make install`
- CMake
- qmake – Qt toolkit
- imake – X Window System

Výkonnost programů (performance)

Základní pravidlo: *raději neoptimalizovat*
Pravidlo *90 — 10* (někdy i 95—5)
Hledání kritických míst v programech:

- Odhad (často zavádějící)
- Měření času:
  příkaz `time`,
  funkce `time()`, `clock()`,
  instrukce `RDTSC`, *performance counters*, ...
- Použití programů typu 'profiler' (`perf`, `gprof`, `cachegrind`, ...).
- Zjišťování paměťové náročnosti (`memprof`, ...)

Poznámky:
Příliš mnoho souvislostí — často připomíná černou magii
Vliv vyrovnávacích pamětí, podsystému virtuální paměti (výpadky stránek, TLB miss), ...

Časová náročnost

Strategie:

- Použít lepší algoritmy/datové struktury
- Zapnout optimalizace při překladu (gcc -O3)
- Provést úpravy kritických částí kódu:
  - přesun invariantů,
  - vyhodnocení společných podvýrazů,
  - použití jiných operací(`a<<=2`),
  - rozvinutí cyklů,
  - uložit často potřebné výsledky (cache),
  - vyrovnávání vstupu/výstupu,
  - speciální alokace malých objektů,
  - předem vypočítané hodnoty (tabulky),
  - počítání s menší přesností,
  - zlepšení lokality odkazů,
  - přepsání do jiného jazyka (asm)
  - ...

Poznámka: *Neoptimalizujte co nemá smysl*

Paměťová náročnost

Souvislost: *časová* x *paměťová* náročnost
Strategie:

- používat co nejmenší datové typy
  (pozor na zarovnávání, bitfields raději nepoužívat)
- neukládat co lze snadno znovu vypočítat
- volit vhodné datové formáty
  (např. de/komprese může být časově náročná)
- velikost paměti a cache

Poznámky:

- lokalita odkazů, vliv CPU cache
- problém rekurze (příklad: Ackerman)
- memory profiler: valgrind/cachegrind, ...

Přenositelnost (portability)

Přenositelný program lze přeložit na více různých platformách (například UNIX, Linux/x86, Windows, MacOS X, ...)

- obvykle se odhalí více chyb
- náročnější na dodržování standardů
- nevyužívat speciality platforem, platformově závislý kód dávat do samostatných modulů
- pozor na implementací definované vlastnosti prostředí (velikost `int`, pořadí vedlejších efektů, zarovnání, little/big endian, bitfields, un/signed char, ...)
- používat textové formáty na výměnu dat (pozor na CRLF)
- ...

Přenositelnost – pokračování

- Přenositelné (multiplatform) knihovny – podstatně usnadňují psaní přenositelných programů (příklady: SDL, Qt, WxWidgets, ...).
- Vhodné vývojové nástroje (GCC, make, gdb, ...).
- ”Cross development”– editace/překlad/sestavení neprobíhá na cílové platformě (důležité především pro ”embedded”systémy).
- Automatická detekce konfigurace při překladu – GNU autoconf:
  `configure; make; make install`
  (GNU build system: Automake, Autoconf, Libtool)

Konverzní programy

Programy pro překlad z jiných jazyků do C:

- p2c – Pascal
- f2c – Fortran
- Některé jazyky používají C jako vysokoúrovňový assembler (pojem ”translator”, příklady: Cfront C++, Modelica)
- ...

Poznámka: + podpůrné knihovny

Programy pro správu revizí

Tyto programy (Revision Control, Version Control Systems) dovolují sledování změn při vývoji programů.
Příklady:

- SCCS (Bell-labs, 1980), RCS (198x), ...
- Klient-server: CVS (1986), Subversion (2000), ...
- Distribuované: Git, GNU arch, Bazaar, Monotone, ...

Princip:

- archiv souborů (repository),
- operace vložení (commit, checkin),
- operace výběru (checkout),
- ukládá pouze změny (algoritmus `diff`),
- přístup k libovolné verzi v historii změn,
- možnost větvení (branching),
- je možný současný přístup více vývojářů.

Git

Distribuovaný systém, používán velkými projekty (Linux, ...)
Jednoduchý úvod (lokální repozitář):

```
mkdir adresar
cd adresar
git init          ## založení (prázdného) repozitáře

## editujeme soubory v adresáři a podadresářích
git add .         ## přidáme soubory do sledovaných
git commit -a     ## zapíše změny (TODO: -m)
git tag -a -m "Brand New Release" 1.0

## editujeme soubory v adresáři a podadresářích

git checkout 1.0          ## návrat
gitk              ## GUI pro sledování revizí
```

Git – pokračování

Jednoduchý úvod (vzdálený repozitář):

```
git clone host:/cesta/adresar ## stažení repozitáře
cd adresar

## editujeme soubory v adresáři a podadresářích
git add .          ## přidáme soubory do sledovaných
git commit -a      ## zapíše změny (TODO: -m)
git push           ## zápis na server

## editujeme soubory v adresáři a podadresářích

git pull           ## stažení aktualizace
                   ## možné kolize!
# a další (rebase,stash,...) viz literatura
```

Obsah souborů je (po commit) uložen na alespoň 2 místech:
repo1-vzdálené — repo2-lokální — soubory

Příklady API

- Jednoduchý příklad: dlist.h – dvojsměrně vázaný seznam (+varianty: Linux lists)
- Knihovny:
  - GMP – počítání s (téměř) libovolnou přesností
  - zlib, libbzip2 – komprese dat
  - libpng, libgd, SDL, OpenGL – grafika
  - libsnd, libvorbis, OpenAL – zvuk
  - GSL, atlas, BLAS, LAPACK – numerické výpočty
  - regex – regulární výrazy
  - guile – vestavěný jazyk Scheme
- Plug-in, komponenty: CORBA, XPCOM, ...

Bezpečnost — základy

- Problém typu ”buffer overflow”(zásobník, heap, ...) – je nutné dobře ošetřit vstupy
- Problém ”temporary file creation”– pozor na postup vytváření jmen dočasných souborů
- ...
- Problémový HW: Spectre, Meltdown, ...
- ...
- Příklady
- Velmi důležité zvláště u SUID programů a serverů

UCS – přehled

- ISO 10646, Universal Character Set (UCS), 2^{31}, ”Planes”
- kód (U+0041) + jméno znaku (”Latin capital letter A”)
- Rozsah U+0000 – U+00FF odpovídá ISO 8859-1 (Latin-1)
- Unicode je kompatibilní s UCS, navíc algoritmy, atd.
- Kódování UCS-2, UCS-4, UTF-8, UTF-16, ...
- Kombinace znaků (”combining char”, např. `r+háček = ř`)
- Úrovně implementace (Level 1-3)

Problémy:

- různá kódování, BOM
- možnost více kódů pro jeden znak:
  `U+00B5 MICRO SIGN = U+03BC GREEK SMALL LETTER MU`

Poznámky: International Components for Unicode (ICU), Pango, Qt

UTF-8 – základy

UTF = ”UCS Transformation Format”
UTF-8 (autor: Ken Thompson, kolem 1992)

```
U00000000 - U0000007F: 0xxxxxxx
U00000080 - U000007FF: 110xxxxx 10xxxxxx
U00000800 - U0000FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U00010000 - U001FFFFF: 11110xxx 10xxxxxx*3
U00200000 - U03FFFFFF: 111110xx 10xxxxxx*4
U04000000 - U7FFFFFFF: 1111110x 10xxxxxx*5
```

Z bezpečnostních důvodů dekodér akceptuje pouze minimální nutnou reprezentaci.
UNICODE definuje normalizované formy (NFD,NFC,...).

UTF-8 – použití

Požadavky:

- Systém: libc + vygenerovaná lokalizační data (locale -a)
- `LC_*` nastaveno na UTF-8 (locale; locale charmap)
- Terminál/editor pracující s UTF-8
- Použití `setlocale(LC_CTYPE, "")`
- ? Překladač pracující s lokalizacemi UTF-8 (literály)
- ? `wchar_t`, mbs, wcs — přímo použitelné jen pokud je definováno makro `__STDC_ISO_10646__` s hodnotou `yyyymmL`

Možnosti zpracování:

- Vstup, zpracování i výstup v UTF-8
- Vstup/výstup UTF-8, zpracování v `wchar_t`

Nástroje:

- Použití libc a lokalizace
- Vlastní implementace (potenciálně rychlejší)

UTF-8 – použití

Konverze:

```
size_t wcstombs(char *dest, const wchar_t *src, size_t n);
size_t mbstowcs(wchar_t *dest, const char *src, size_t n);
Thread-safe: wcsrtombs, mbsrtowcs
```

Funkce nezávislé na lokalizaci a kódování:

```
strcpy strcat strcmp strstr
```

Funkce závislé na lokalizaci ale ne na kódování:

```
strcoll strxfrm
```

UTF-8 – použití

Délka řetězce:

- počet bajtů (pro alokaci)
- počet znaků (málo potřebné) `mbstowcs(NULL,s,0)`
- počet pozic na displeji (`wcwidth`,`wcswidth`)

Editace:

- vkládání znaku
- rušení znaku
- pozice znaku na obrazovce (čínské zabírají 2 pozice)

UTF-8 – příklady

Základní použití nevyžaduje zapnutí lokalizace:

```
#include <stdio.h>

int main() {
  printf("Hello, UTF-8 world: ěščřžýáíé \n");
}
```

UTF-8 – příklady

Tisk širokých znaků:

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

int main() {
  if(!setlocale(LC_CTYPE, "")) {
    fprintf(stderr, "Nelze nastavit lokalizaci.\n");
    return 1;
  }
  printf("%ls\n", L"ěščřžýáíé öüß");     // režim char
  // wprintf(L"ěščřžýáíé öüß");          //  wchar_t
}
```

Závěr – poznámky a souvislosti

- Free Software, Projekt GNU
- Licence (GPL, LGPL, BSD, MIT, MPL, Apache, ...)
- WWW stránka s odkazy
- ...
- Co bude u zkoušky