PB111 --> „reálne“ C

(kľúčové body sú z prednášok PB071 doc. Švendu z roku 2023)

Prekladač (kompilátor), štandardy, normy, rozšírenia

používame prekladač GNU Compiler Collection vo verzii **11.5.0**

príkaz: `gcc`

používame C v štandarde **ISO/IEC 9899:1999** (C99)

prepínač: `-std=c99`

používame rozšírenie **POSIX.1-2008** (200809)

prepínač: `-D_POSIX_C_SOURCE=200809L`

Prekladač (kompilátor), jeho výsosť

pomenovanie výstupného súboru

prepínač: `-o <meno>`

zapnutie ladiacich (debug) informácii

prepínač: `–g`

striktné dodržiavanie štandardu (C99) vo forme upozornení

prepínač: `–pedantic`

upozornenie na pochybné konštrukcie v kóde

prepínače: `-Wextra -Wall`

upozornenia sú považované za chyby

prepínač: `-Werror`

Dátové typy

veľkosti **rozdielne na rôznych strojoch**, zachované iba nerovnosti

zistiteľné pomocou makra: `sizeof(<typ>)`

typ/veľkosť v bajtoch	PB111	Aisa (x86_64)
`(unsigned/signed)` **`char`**	1	1
`(unsigned)` **`int`**	2	4
`bool`	1/8	1/8
`float`	N/A	4
`double`	N/A	8
`smerník (pointer)`	2	8

Modifikátory typov

veľkosť vieme upravovať ľubovoľnému typu modifikátorom

Modifikátor	Zmena
`unsigned`	neznamienkový (0 – 2 ^ veľkosť)
`signed`	znamienkový (-2^(veľkosť-1) – 2^(veľkosť-1)-1)
`short`	kratšia verzia typu
`long`	dlhšia verzia typu (potrebná podpora prekladača)
`long long`	ešte dlhšia verzia typu (potrebná podpora)

Makrá

pomenované fragmenty kódu

definujeme pomocou `#define <meno> <hodnota>`

v kóde sa nahradzujú **textovo** (tzn. bez sémantiky, iba kus za kus)

pozor preto na bodkočiarky (;) na konci, zátvorkovanie, ...

príklad:

```
#define MAKRO_A (n + m)
#define MAKRO_B n + m
int n = 3, m = 5;
int a = MAKRO_A * 5;  // a = 40
int b = MAKRO_B * 5;  // b = 28
```

(Štandardná) knižnica

knižnice slúžia pre rozšírenie funkcionality

vkladáme pomocou makra `#include <meno>`

knižnica priložená ku prekladaču – **štandardná knižnica**

príklad:

```
#include <stdio.h>    // funkcie printf(), scanf()
#include <stdlib.h>   // funckie malloc(), calloc(), free()
#include <stdbool.h>  // typ bool, makrá true, false
#include <unistd.h>   // funkcie write(), close()
#include <stdint.h>   // fixné číselné typy napr. uint64_t, int32_t
#include <assert.h>   // makro assert()
```

Manuálové stránky

informácie o implementácii konkrétnych funkcií na konkrétnych strojoch

na internete https://en.cppreference.com/w/c.html (verzia pre C, nie C++)

na Linuxe zavoláme pomocou príkazového riadku

pre POSIX implementáciu: `man 3p <funkcia>`

pre bežnú C implementáciu: `man 3 <funkcia>`

príklad:

```
man 3 malloc
```

Textové reťazce (string)

na vyjadrenie textu v programe, napr. pri výpise (s funkciami zo `<stdio.h>`)

pole znakov (`char`) ukončených znakom binárnej nuly (`'\0‘`)

definujeme (napr.) pomocou: `char *<meno> = "Text v reťazci";`

príklad:

```
char *textovy_literal = "Text v literali"; // neupravovateľný
char textove_pole[] = "Text v literali";   // upravovateľný po max. veľkosť
char *dynamicky_retazec = malloc( <dlzka_textu> + 1 );
strcpy( dynamicky_retazec, ukazatel_na_text );
// vždy je uložený taký počet znakov, ktorý reprezentuje text plus jeden pre binárnu nulu
// "text" ~ {'t', 'e', 'x', 't', '\0'}
```

Argumenty funkcie `main()`

na vkladanie vstupných informácii, napr. pri volaní `./program <arg1> <arg2>`

tri vzájomne zameniteľné predpisy (signatúry) funkcie `main()`

```
int main( void );
int main( int argc, char *argv[] );
int main( int argc, char *argv[], char **envp );
```

`int argc` obsahuje počet parametrov vložených o programu, ak `argc > 1`, tak prvý argument je cesta ku spusteniu programu

`char *argv[]` je pole reťazcov o veľkosti `argc` obsahujúce jednotlivé argumenty pre program, `argv[ 0 ]` je cesta ku spusteniu programu

`char **envp` je nulou (`NULL`) ukončené pole textových reťazcov obsahujúce premenné prostredia

Funkčné smerníky (funkčné ukazatele, function pointers)

smerník, ktorý obsahuje adresu umiestnenia kódu funkcie

smerník získame bežným spôsobom: `&<meno_funkcie>`

vieme uložiť do premennej:

```
<návratový typ> (*<meno_premennej>) (<typ_argumentu1>, <typ_argumentu2>, ...)
```

využitie napríklad pri porovnávaní pomocou funkcie `qsort()` zo `<stdlib.h>`

príklad:

```
int funkcia( int a, int b )
{ ... }
int ( *smernik_na_funkcu )( int, int ) = &funkcia;
smernik_na_funkcu( 3, 5 );  // volanie funkcie, rovnako ako funkcia( 3, 5 )
```

Preklad

v 4 častiach**: predprocesovanie**, **preklad**, **zostavenie**, **linkovanie**

predprocesovanie (preprocessing)

spracovanie makier (expanzia #include, nahradzovanie #define, ...)

preklad (compilation)

syntaktická kontrola kódu, výsledok assembly kód

zostavenie (assembly)

assembly kód do strojového kódu

linkovanie (linking)

nahradzovanie relatívnych adries absolútnymi

Ladenie (debug) a úniky pamäte (memory leaks)

systémové volania ladíme pomocou nástroja `strace`

```
strace –fyi ./<nazov_skompilovaneho_suboru>,
```

beh programu ladíme pomocou nástroja `gdb`

```
gdb ./<nazov_skompilovaneho_suboru>
```

úniky pamäte hľadáme pomocou nástroja `valgrind`, konkrérne

```
valgrind --leak-check=full --show-leak-kinds=all --track-fds=yes
--track-origins=yes -s ./<nazov_skompilovaneho_suboru>
```