# Jazyk C – Příklady

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 10

**A0B36PR2 – Programování 2**



## Dílčí příklady použití jazykových konstrukcí v projektu

- Program složený z více souborů
- Dynamická alokace paměti
- Načítání souboru
- Parsování čísel z textového souboru

- Měření času běhu programu
- Řízení kompilace projektu složeného z více souborů `Makefile`



## Zadání

- Vytvořte program, který načte orientovaný graf definovaný posloupností hran
  - Graf je zapsán v textovém souboru
- Navrhněte datovou strukturu pro reprezentaci grafu
- Počet hran není dopředu znám

  *Zpravidla však budou na vstupu grafy s průměrným počtem hran $3n$ pro $n$ vrcholů grafu.*
- Hrana je definována číslem vstupního a výstupního vrcholu a cenou (také celé číslo)
  - Ve vstupním souboru je každá hrana zapsaná samostatně na jednom řádku
  - Řádek má tvar:

    `from to cost`
  - kde `from`, `to` a `cost` jsou kladná celá čísla v rozsahu `int`
- Pro načtení hodnot hran použijte pro zjednodušení funkci `fscanf()`
- *Program dále rozšiřte o sofistikovanější, méně výpočetně náročné načítání*



## Pravidla překladu v gmake

- Pro řízení překladu použijeme předpis programu `GNU make`

  `make` nebo `gmake`
- Pravidla se zapisují do souboru `Makefile`

  `http://www.gnu.org/software/make/make.html`
- Pravidla jsou deklarativní ve tvaru definice cíle, závislostí cíle a akce, která se má provést

  **cíl** : **závislosti** *dvojtečka*

  **akce** *tabulátor*
- Cíl (podobně jako závislosti) může být například symbolické jméno nebo jméno souboru

  **tload.o** : **tload.c**

  **clang -c tload.c -o tload.o**
- Předpis může být napsán velmi jednoduše

  *Například jako v uvedené ukázce.*

Flexibilita použití však spočívám především v použití zavedených proměnných, vnitřních proměnných a využití vzorů, neboť většina zdrojových souborů se překládá identicky.

## Příklad – Makefile

- Definujeme pravidlo pro vytvoření souborů .o z .c
- Definice přeložených souborů vychází z aktuálních souborů v pracovním adresáři s koncovkou .c

```
CC:=ccache $(CC)
CFLAGS+=-O2

OBJS=$(patsubst %.c,%.o,$(wildcard *.c))

TARGET=tload

bin: $(TARGET)

$(OBJS): %.o: %.c
    $(CC) -c $< $(CFLAGS) $(CPPFLAGS) -o $@

$(TARGET): $(OBJS)
    $(CC) $(OBJS) $(LDFLAGS) -o $@

clean:
    $(RM) $(OBJS) $(TARGET)
```

*ccache CC=clang make vs CC=gcc make*

- **Při linkování záleží na pořadí souborů (knihoven)!**
- Jednou z výhod dobrých pravidel je možnost paralelního překladu nezávislých cílů

```
make -j 4
```



## Definice datové struktury grafu

- Zavedeme nový typ datové struktury hrana—edge_t,
- který použijeme ve struktuře grafu—graph_t

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

typedef struct {
   int from;
   int to;
   int cost;
} edge_t;

typedef struct {
   edge_t *edges;
   int length;
   int size;
} graph_t;

#endif
```

- Soubor budeme opakovaně vkládat (include) v ostatních zdrojových souborech, proto „zabraňujeme" opakované definici konstantou preprocesoru __GRAPH_H__



## Pomocné funkce pro práci s grafem

- Alokaci/uvolnění grafu implementujeme v samostatných funkcích
- Při načítání grafu budeme potřebovat postupně zvyšovat paměť pro uložení načítaných hran
- Proto využijeme dynamické alokace paměti pro „*nafukování*" paměti pro uložení hran grafu—enlarge_graph() o nějakou definovanou velikost

```
#ifndef __GRAPH_UTILS_H__
#define __GRAPH_UTILS_H__

#include "graph.h"

graph_t* allocate_graph(void);

void free_graph(graph_t **g);

graph_t* enlarge_graph(graph_t *g);

void print_graph(graph_t *g);

#endif
```



## Alokace paměti pro uložení grafu

- Testujeme úspěšnost alokace paměti—assert()
- Po alokaci nastavíme hodnoty proměnných na NULL

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "graph.h"

graph_t* allocate_graph(void) {
   graph_t *g = (graph_t*) malloc(sizeof(graph_t));
   assert(g != NULL);
   g->edges = NULL;
   g->length = 0;
   g->size = 0;
   /* or we can call calloc */
   return g;
}
```

- Alternativně můžeme použít funkci calloc()

## Uvolnění paměti pro uložení grafu

- Testujeme validní hodnotu argumentu funkce—`assert()`

*Pokud se stane chyba, tak funkci v programu špatně voláme.*

*Až program odladíme můžeme kompilovat s* `NDEBUG`.

```
void free_graph(graph_t **g) {
   assert(g != NULL && *g != NULL);
   if ((*g)->size > 0) {
      free((*g)->edges);
   }
   free(*g);
   *g = NULL;
}
```

- Po uvolnění paměti nastavíme hodnotu ukazatele na strukturu na hodnotou `NULL`



## Zvětšení paměti pro uložení hran grafu

- V případě nulové velikosti alokujme paměť pro `NSIZE` hran
- `NSIZE` můžeme definovat při překladu

např. `clang -D NSIZE=100 -c graph_utils.c`

```
#ifndef NSIZE
#define NSIZE 10
#endif

graph_t* enlarge_graph(graph_t *g) {
   assert(g != NULL);
   int n = g->size == 0 ? NSIZE : g->size * 2;
   /* double the memory */
   edge_t *e = (edge_t*)malloc(n * sizeof(edge_t));
   memcpy(e, g->edges, g->length * sizeof(edge_t));
   free(g->edges);
   g->edges = e;
   g->size = n;
   return g;
}
```

- Místo alokace nového bloku paměti a kopírování původního obsahu můžeme použít funkci `realloc()`



## Tisk hran grafu

- Pro tisk hran grafu využijeme pointerovou aritmetiku

```
void print_graph(graph_t *g) {
   assert(g != NULL);
   fprintf(stderr, "Graph has %d edges and %d edges are
     allocated\n", g->length, g->size);
   edge_t *e = g->edges;
   for(int i = 0; i < g->length; ++i, e++) {
      printf("%d %d %d\n", e->from, e->to, e->cost);
   }
}
```

- Informace vypisujeme na standardní chybový výstup
- Graf tiskneme na standardní výstup
- Při tisku a přesměrování standardního výstupu tak v podstatě můžeme realizovat kopírování souboru s grafem

*Např.* `./tload -p g > g2`



## Hlavní funkce programu – `main`

- V hlavní funkci zpracujeme předané argumenty programu
- V případě uvedení přepínače `-p` vytiskneme graf na `stdout`

```
int main(int argc, char *argv[]) {
   int ret = 0;
   int print = 0;
   char *fname;
   int c = 1;
   if (argc > 2 && strcmp(argv[c], "-o") == 0) {
      print = 1;
      c++;
   }
   fname = argc > 1 ? argv[c] : NULL;
   fprintf(stderr, "Load file '%s'\n", fname);
   graph_t *graph = allocate_graph();
   int e = load_graph_simple(fname, graph);
   fprintf(stderr, "Load %d edges\n", e);
   if (print) {
      print_graph(graph);
   }
   free_graph(&graph);
   return ret;
}
```

## Jednoduché načtení grafu – deklarace

- Prototyp funkce uvedeme v hlavičkovém souboru–`load_simple.h`

```
#ifndef __LOAD_SIMPLE_H__
#define __LOAD_SIMPLE_H__

#include "graph.h"

int load_graph_simple(const char *fname, graph_t *g);


#endif
```

- Vkládáme pouze soubor `graph.h`—pro definici typu `graph_t`

*Snažíme se zbytečně nevkládat nepoužívané soubory*



## Jednoduché načtení grafu – implementace 1/2

- Používáme funkci `enlarge_graph`, proto vkládáme `graph_utils.h`

```
#include <stdio.h>
#include "graph_utils.h"
int load_graph_simple(const char *fname, graph_t *g) {
   int c = 0;
   int exit = 0;
   FILE *f =  fopen(fname, "r");
   while(!feof(f) && !exit) {
      if (g->length == g->size) {
         enlarge_graph(g);
      }
      edge_t *e = g->edges + g->length;
      while(!feof(f) && g->length < g->size) {
   /* read and parse a single line */
      }
   }
   fclose(f);
   return c;
}
```

- `load_simple.h` vkládat nemusíme, obsahuje pouze prototyp funkce
- Obecně je to však dobrý zvykem nebo nutností (definice typů)



## Jednoduché načtení grafu – implementace 2/2

- Pro načtení řádku s definicí hrany použijeme funkci `fscanf()`

```
while(!feof(f) && g->length < g->size) {
   int r = fscanf(f, "%d %d %d\n", &(e->from), &(e->to
 ), &(e->cost));
   if (r == 3) {
      g->length++;
      c++;
      e++;
   } else {
      exit = 1;
      break;
   }
}
```

- Kontrolujeme počet přečtených parametrů a až pak zvyšujeme počet hran v grafu



## Spuštění programu 1/3

- Nechť máme soubor `g` definující graf o 1 000 000 uzlech

*Velikost souboru cca 62 MB (příkaz `du`–disk usage)*

```
% du g
62M     g

brettonia% ./tload g
Load file 'g'
Load 2998898 edges

% time ./tload g
Load file 'g'
Load 2998898 edges
./tload g  1.12s user 0.03s system 99% cpu 1.151 total
```

- Příkazem `time` můžeme změřit potřebný čas běhu programu

*strojový, systémový a reálný*

## Spuštění programu 2/3

- Příznakem `-p` a přesměrováním standardního výstupu můžeme vytisknout graph do souboru

*V podstatě vstupní soubor zkopírujeme.*

```
% time ./tload -p g  > g2
Load file 'g'
Load 2998898 edges
Graph has 2998898 edges and 5242880 edges are allocated
./tload -p g > g2  2.09s user 0.07s system 99% cpu 2.158
    total
% md5 g g2
MD5 (g) = d969461a457e086bc8ae08b5e9cce097
MD5 (g2) = d969461a457e086bc8ae08b5e9cce097
```

- Čas běhu programu je přibližně dvojnásobný
- Oba soubory se zdají být z otisku `md5` identické

*Na Linuxu* `md5sum` *případně lze použít otisk* `sha1` , `sha256` *nebo* `sha512`



## Spuštění programu 3/3

- Implementací sofistikovanějšího načítání

```
% /usr/bin/time ./tload g
Load file 'g'
Load 2998898 edges
0.19 real         0.16 user         0.03 sys
```

- lze získat výrazně rychlejší načítání

*160 ms vs 1100 s*

```
% /usr/bin/time ./tload g
Load file 'g'
Load 2998898 edges
1.15 real         1.05 user         0.10 sys
```



## Jak a za jakou cenu zrychlit načítání seznamu hran

- Zrychlit načítání můžeme přijmutím předpokladů o vstupu
- Při použití `fscanf()` je nejdříve načítán řetězec (řádek) pak řetěz reprezentující číslo a následně je parsováno číslo
- Převod na číslo je napsán obecně
- Můžeme použití postupné „bufferované" načítání
- Převod na číslo můžeme realizovat přímo po přečtení tokenu
- parsováním znaků (číslic) načtené posloupnosti bytů v obráceném pořadí

- Můžeme získat výrazně rychlejší kód, který je však komplexnější a pravděpodobně méně obecný