

1. Generování cílového kódu v C/C++

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Program v C s více moduly

```
1 int sum (int * a, int n);
3 int array[2] = {1, 2};
5 int main (void)
6 {
7     int val = sum (array, 2);
8     return val;
9 }
```

main.c

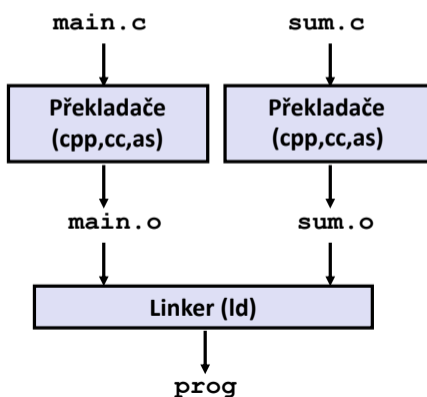
```
1 int sum (int *a, int n)
2 {
3     int i, s= 0;
4
5     for (i = 0; i < n; i++) {
6         s += a[i];
7     }
8
9     return s;
10 }
```

sum.c

Překlad a linkování

- Programy v C/C++ jsou přeloženy a linkovány **kompilátorem**

```
standa@teplaky:~$ gcc -Og -o prog main.c sum.c
```



Zdrojové kódy

*Odděleně přiložené
relokovatelné objektové soubory*

*Spustitelný objektový soubor
(obsahuje kód a data všech funkcí
definovaných v main.c a sum.c)*

Výhody použití linkeru

1. Modularita

- Program může být rozepsán do několika souborů (modulů)
- Lze vytvářet knihovny často používaných funkcí

2. Efektivita

- Čas kompilace – oddělená kompilace
 - změním jeden soubor → zkompiluji → přilinkuji k ostatním
 - nezměněné soubory není třeba znova kompilovat
 - lze kompilovat různé moduly nezávisle
- Velikost kódu – knihovny
 - **statické knihovny**
 - spustitelné soubory a obrazy v paměti obsahují pouze knihovní kód, který používají
 - **dynamické knihovny**
 - spustitelné soubory neobsahují žádný knihovní kód
 - za běhu může být jedna kopie knihovního kódu sdílena napříč více procesy

1. Rozlišování symbolů

- Programy definují a referencují symboly (globální proměnné a funkce)

```
void swap() {...}      /* define symbol swap */
swap();                /* reference symbol swap */
int *xp = &x;         /* define symbol xp, reference x */
```

- Definice symbolů jsou umístěny v objektovém souboru (assemblerem) v **tabulce symbolů**
 - Tabulka symbolů je pole záznamů
 - Každý záznam obsahuje název, velikost a umístění symbolu

Během kroku rozlišování symbolů provede linker asociaci referencí s právě jednou definicí.

Symboly v příkladu

```
1  int sum (int * a, int n);
3  // array - definice
4  int array[2] = {1, 2};
6  // main - definice
7  int main (void)
8  {
9      // sum - reference
10     int val = sum (array, 2);
11     return val;
12 }
```

main.c

```
1  // sum - definice
2  int sum (int *a, int n)
3  {
4      int i, s= 0;
5
6      for (i = 0; i < n; i++) {
7          s += a[i];
8      }
9
10     return s;
11 }
```

sum.c

2. Relokace

- uspořádání objektů v adresním prostoru programu
 - strojový kód používá absolutní adresy
 - při nahrání kódu na jinou adresu by kód nefungoval
- modifikace absolutních adres
 - podle relokační tabulky nebo rozlišení symbolů
 - relokační tabulka je součástí spustitelného souboru
- spustitelný produkt linkeru může vyžadovat další relokační při načtení do paměti
 - např. při použití dynamických knihoven
 - u hardwaru nabízejícího virtuální paměť je tento průchod obvykle vynechán – každý program je umístěn do vlastního adresního prostoru, takže nedochází ke konfliktu, i když se všechny programy načítají na stejnou základní adresu

Objektové soubory

• Obsah objektových souborů

- Informace o souboru a překladači
- Cílová architektura, struktura, ...
- Kód
- Data
- Ostatní informace potřebné k běhu/linkování/debugování
 - Relokace, informace o symbolech atd.
 - Ladicí informace

Tyto informace v oddílech (sekcích)

• Formáty objektových souborů

- a.out (UNIX (od 1971)/starý Linux)
- ELF (Executable and Linkable Format, Linux, od 1988 majoritní *nix formát)
- COM (DOS, čistě kód+data, nic víc)
- COFF (Common Object File Format, UNIX (1983), IRIX, nahrazen formátem ELF)
- MZ (DOS .exe)
- PE (Portable Executable, Windows .exe, vychází z COFF)

Typy objektových souborů

Relokovatelný objektový soubor (*.o)

- Obsahuje kód a data v takové formě, která může být kombinována s jinými relokovatelnými objektovými soubory a společně mohou vytvořit spustitelný kód
- Každý zdrojový kód (modul) produkuje právě jeden objektový kód

Spustitelný objektový soubor (a.out)

- Obsahuje kód a data v takové formě, která může být přímo nakopírována do paměti a spuštěna

Sdílený objektový soubor (*.so)

- Speciální typ relokovatelného objektového souboru, který může být nahrán do paměti a dynamicky linkován, případně nahrán za běhu programu
- Windows: Dynamic Linked Libraries (DLLs)

ELF – Executable and Linkable Format

- Hlavní souborový formát OS Linux – přenositelný, relokovatelný, rozšiřitelný
- Struktura
 - ELF hlavička – délka slova, endianita, typ objektového souboru, platforma
 - Hlavičky sekcí – velikost sekce, virtuální adresy
- Sekce
 - vytvářené překladačem/linkerem
 - čtené překladačem/linkerem

Programy pro práci s objekty

```
objdump
readelf
hd
```

Předdefinované sekce

- `.text` kód
- `.data` nekonstantní data (proměnné)
- `.rodata` konstantní data (řetězce apod.)
- `.bss` data, která se inicializují na 0
- `.interp` interpret
- `.symtab` tabulka symbolů pro ladění (strip)
- `.dynsym` symboly pro dynamický linker

Symboly linkeru

Globální symboly

- Symboly definované v modulu **m**, které mohou být referencované z jiných modulů
- Např. nestatické C funkce nebo nestatické globální proměnné

Externí symboly

- Globální symboly, na které se odkazuje z modulu **m**, ale jsou definovány v jiném modulu

Lokální symboly

- Symboly, které jsou definovány a referencovány výlučně v modulu **m**
- Např. C funkce a globální proměnné definované s atributem `static`
- Lokální symboly linkeru nejsou to samé, jako lokální proměnné programu

Rozlišování symbolů

Reference na globální symbol...

... definovaný zde

```
1 int sum (int * a, int n);
3 int array[2] = {1, 2};
5 int main (void)
6 {
7     int val = sum (array, 2);
8     return val;
9 }
```

main.c

Globální definice

Linker neví nic o val

Reference na globální symbol...

```
1 int sum (int *a, int n)
2 {
3     int i, s= 0;
5     for (i = 0; i < n; i++) {
6         s += a[i];
7     }
9     return s;
10 }
```

sum.c

... definovaný zde

Identifikace symbolů

- Který z následujících symbolů bude zastouper v tabulce symbolů `symbols.o`?

```
1  int incr = 1;
3  static int foo(int a) {
4      int b = a + incr;
5      return b;
6  }
8  int main(int argc,
9  char* argv[]) {
10     printf("%d\n", foo(5));
11     return 0;
12 }
```

- incr
- foo
- a
- argc
- argv
- b
- main
- printf
- "%d\n"

```
readelf -s symbols.o
```

Lokální symboly

- Lokální nestatické proměnné vs. lokální statické proměnné
 - lokální nestatické proměnné jsou uloženy v zásobníku
 - lokální statické proměnné: uložené v `.bss` nebo `.data`

```
1  static int x = 15;
3  int f() {
4      static int x = 17;
5      return x++;
6  }
8  int g() {
9      static int x = 19;
10     return x += 14;
11 }
13 int h() {
14     return x += 27;
15 }
```

Překladač alokuje místo v `.data` pro každou definici `x`

Jsou vytvořeny lokální symboly v tabulce symbolů s unikátními jmény, např. `x`, `x.1721` a `x.1724`

Rozlišení duplicitních definic symbolů

- Symboly v programu jsou buď silné nebo slabé
 - silné** funkce a inicializované globální proměnné
 - slabé** neinicializované globální proměnné (nebo deklarované jako `extern`)

p1.c

```
int foo=5; ← silný
```

```
p1() {} ← silný
```

p1.c

```
int foo ← slabý
```

```
p2() {} ← silný
```

Pravidla pro symboly

1. Více silných symbolů se stejným jménem není dovoleno
 - Silný symbol může být definován pouze jednou
 - Jinak dojde k chybě při linkování
2. Pokud existuje silný symbol a více slabých, linker vybírá silný
 - Reference na slabé symboly se mění na reference na silné
3. Pokud existuje více slabých symbolů, vybírá linker náhodně
 - Lze potlačit při překladu: `gcc -fno-common`

Příklady

```
int x;  
p1() {}
```

```
p1() {}
```

Chyba při linkování, dva silné symboly (`p1`)

```
int x;  
p1() {}
```

```
int x;  
p2() {}
```

Reference na `x` odkazují na tu samou neinicializovanou proměnnou `int`. To chceme?

```
int x, y;  
p1() {}
```

```
double x;  
p2() {}
```

Zápis do `x` v `p2` může přepsat hodnotu `y`

```
int x=7;  
int y=5;  
p1() {}
```

```
double x;  
p2() {}
```

Zápis do `x` v `p2` může přepsat hodnotu `y`

```
int x=7;  
p1() {}
```

```
int x;  
p2() {}
```

Reference `x` odkazují na tu samou inicializovanou proměnnou

Příklad zmatení typů

```
1  /* Slaby symbol */
2  long int x;
4  int main(int argc, char *argv[]) {
5      printf("%ld\n", x);
6      return 0;
7  }
```

```
1  /* Globalni silny symbol */
2  double x = 3.14;
```

- Kompilace proběhne bez chyb nebo varování
- Co se vytiskne na standardní výstup?

Globální proměnné

Pokud je to možné, vyhněte se jim.

- Pokud to jinak nejde
 - používejte atribut `static`, pokud je to možné
 - pokud definujete globální proměnnou, inicializujte ji
 - použijte `extern`, pokud odkazujete na globální proměnnou v jiném modulu

```
1 #include <stdio.h>
2 #include "global.h"
4 int g = 0;
6 int main() {
7     printf("f: %i\n", f());
8     return 0;
9 }
```

```
1 #include "global.h"
3 int f() {
4     return g+1;
5 }
```

Jak bude vypadat `global.h`?

Jak připravit a distribuovat často používané funkce?

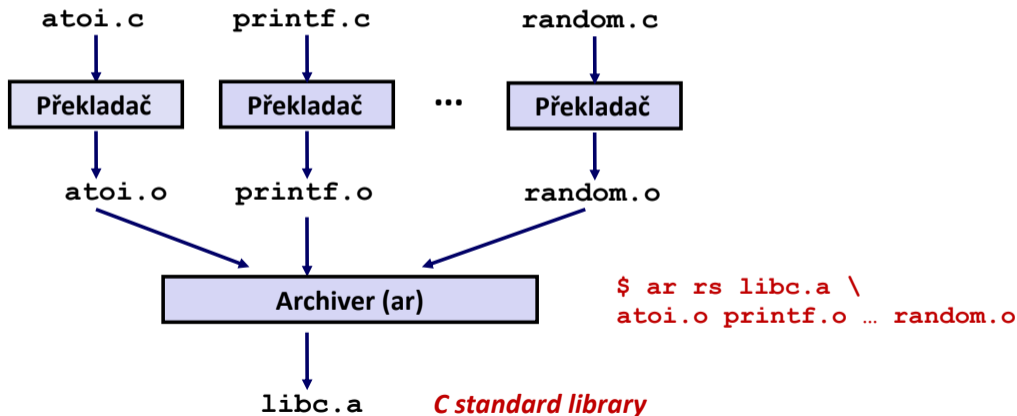
- Typicky matematické funkce, práce s pamětí, manipulace s řetězci, vstup a výstup, ...

Využití linkeru

- **1.** Vložení všech funkcí do jednoho velkého souboru
 - Programátor linkuje velký objektový soubor do svých programů
 - Není příliš efektivní
- **2.** Vytvoření více separátních souborů (třeba pro každou funkci)
 - Programátor explicitně linkuje objekty s potřebnou funkcionalitou
 - Z hlediska kódu efektivnější, z hlediska programátora moc ne :-)

Statické knihovny

- Spojení relokovatelných objektových souborů do jednoho souboru – **archivu**
- Archiv umožňuje inkrementální změny



Běžně používané knihovny

`libc.a` (standardní C knihovna)

`libm.c` (matematická knihovna)

```
$ ar -t /usr/lib/libc.a | sort
...
fork.o
fprintf.o
fpu_control.o
fputc.o
...
```

```
$ ar -t /usr/lib/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
...
```

Co dělat, když nemám knihovnu v `/lib` ani v `/usr/bin`?

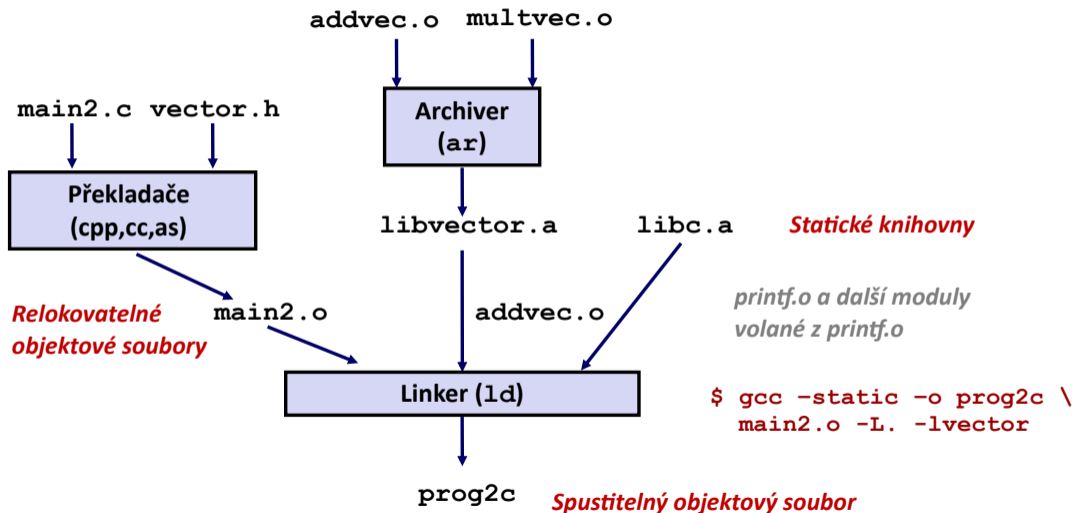
```
$ gcc --print-file-name=libc.a
/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../x86_64-linux-gnu/libc.a
```

```
1  #include <stdio.h>
2  #include "vector.h"
4  int x[2] = {1, 2};
5  int y[2] = {3, 4};
6  int z[2];
8  int main()
9  {
10     addvec(x, y, z, 2);
11     printf("z = [%d %d]\n",
12           z[0], z[1]);
13     return 0;
14 }
```

libvector.a

```
1  void addvec(int *x, int *y,
2  int *z, int n) {
4     for (int i = 0; i < n; i++)
5         z[i] = x[i] + y[i];
6 }

1  void multvec(int *x, int *y,
2  int *z, int n) {
4     for (int i = 0; i < n; i++)
5         z[i] = x[i] * y[i];
6 }
```



Používání statických knihoven

- Algoritmus linkeru pro řešení externích referencí
 - Projdi *.o a *.a soubory v pořadí určeném příkazem kompilace
 - Během průchodu udržuj seznam nevyřešených referencí
 - Při nalezení nového *.o nebo *.a se snaž vyřešit symboly z tabulky symbolů
 - Pokud je na konci procesu nějaký nevyřešený symbol, došlo k chybě

Problém:

- Záleží na pořadí!
- Doporučení: knihovny dávejte na konec

```
$ gcc -static -o prog2c -L. -lvector main2.o  
main2.o: In function 'main':  
main2.c:(.text+0x19): undefined reference to 'addvec'  
collect2: error: ld returned 1 exit status
```

Sdílené knihovny

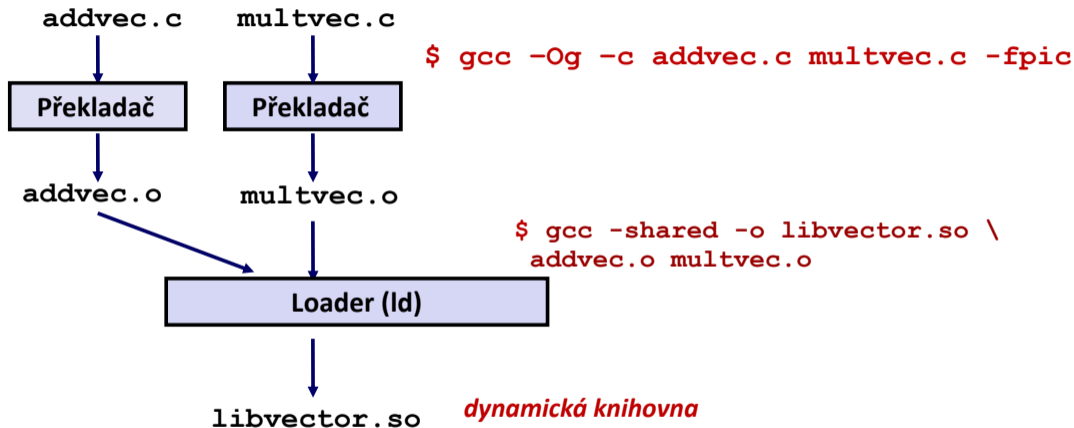
- **Nevýhody statických knihoven**
 - Duplikace ve uložených spustitelných souborech (každá funkce potřebuje `libc`)
 - Duplikace v běžících spustitelných souborech
 - I malá změna v systémové knihovně znamená nutnost přelinkování aplikace
- **Moderní řešení: sdílené knihovny**
 - Objektové soubory, obsahující kód a data, která jsou nahrávána do aplikace dynamicky (při spuštění nebo za běhu)
- **Dynamické linkování může proběhnout při zavádění do paměti (load-time linking)**
 - Typicky v Linuxu, automatizace dynamickým linkerem (`ld-linux.so`)
- **Dynamické linkování může také proběhnout po startu programu (run-time linking)**
 - V Linuxu se řeší před rozhraní `dlopen()`
 - Distribuovaný SW, výkonné webové servery, ...
- **Sdílené knihovny lze sdílet mezi více procesy**
 - Pokud zbyde čas na příští přednášce

Co dynamické knihovny potřebují?

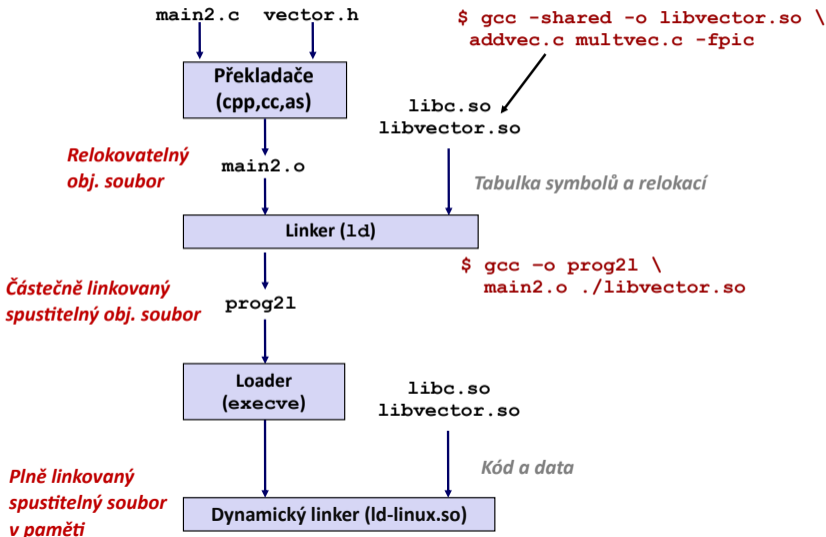
- `.interp` sekce
 - Specifikuje dynamický linker (např. `ld-linux.so`)
- `.dynamic` sekce
 - Obsahuje informace (jména, ...) použitých dynamických knihoven
 - Viz příklad
- Kde je možné knihovny nalézt?
 - Program `ldd`

```
$ ldd prog
linux-vdso.so.1 => (0x00007ffcf2998000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
/lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

Příklad dynamické knihovny



Load-time dynamické linkování



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <dlfcn.h>
4
5  int x[2] = {1, 2}, y[2] = {3, 4}, z[2];
6
7  int main(int argc, char** argv) {
8      void *handle;
9      void (*addvec)(int *, int *, int *, int);
10     char *error;
11     /* Dynamically load the shared library that contains addvec() */
12     handle = dlopen("./libvector.so", RTLD_LAZY);
13     if (!handle) {
14         fprintf(stderr, "%s\n", dlerror());
15         exit(1);
16     }
17     ...
```

```
1  /* Get a pointer to the addvec() function we just loaded */
2  addvec = dlsym(handle, "addvec");
3  if ((error = dlerror()) != NULL) {
4      fprintf(stderr, "%s\n", error);
5      exit(1);
6  }
7  /* Now we can call addvec() just like any other function */
8  addvec(x, y, z, 2);
9  printf("z = [%d %d]\n", z[0], z[1]);
11 if (dlclose(handle) < 0) { /* Unload the shared library */
12     fprintf(stderr, "%s\n", dlerror());
13     exit(1);
14 }
15 return 0;
16 }
```

