Y36PJC Programování v jazyce C/C++

Moduly, projekty, make, Makefile

Ladislav Vagner

Dnešní přednáška

- Moduly v C/C++:
  - hlavičkové soubory,
  - implementační soubory,
  - oddělený překlad.
- Správa projektů:
  - Makefile, make,
  - nástroje ar, nm, ldd a jejich Windows ekvivalenty.

Minulá přednáška

- Abstraktní třídy.
- RTTI – run-time type info.
- Přetypování v C++.
- Výjimky.

Moduly

- Moduly v C/C++:
  - související kód (např. třída, skupina spolupracujících funkcí),
  - ucelená funkčnost,
  - poskytuje rozhraní pro využití ostatními moduly,
  - nezatěžuje ostatní moduly detaily své implementace.
- Realizace modulu:
  - deklarace (rozhraní) = hlavičkový soubor (.h),
  - definice (implementace) = zdrojový soubor (.cpp, .c).
- Výhody rozdělení do modulů:
  - přehlednost,
  - spolupráce více programátorů,
  - rychlejší překlad.

Moduly – hlavičkové soubory

```
// priklad .h
#ifndef __CRational_h_49057405__
#define __CRational_h_49057405__
const double PI = 3.1415926535;
class CRational
 {
    int numerator, denominator;
   public:
    CRational ( int n = 0, int d = 1 );
    friend CRational operator+ ( const CRational & a,
                                 const CRational & b );
    …
 };
CRational operator + ( const CRational & a,
                       const CRational & b );
#endif /* __CRational_h_49057405__ */
```

Moduly – hlavičkové soubory

- Hlavičkový soubor (.h):
  - deklarace tříd, metod, funkcí, konstant a proměnných,
  - definice inline metod a funkcí.
- Často obsahuje direktivy preprocesoru, které zajistí vložení pouze 1x:

```
#ifndef __CRational_h_49057405__
#define __CRational_h_49057405__
...
#endif /* __CRational_h_49057405__ */
```

- Hlavičkový soubor může obsahovat vložení jiných hlavičkových souborů (které využívá).
- Hlavičkový soubor by neměl používat direktivy, které mají globální dopad – např:

```
using namespace std;
```

Moduly – hlavičkové soubory

```
// priklad .h
#ifndef __CRational_h_49057405__
#define __CRational_h_49057405__
#include <iostream>
//using namespace std;  - sem ne

class CRational
 {
   …
   friend std::ostream operator+ ( std::ostream & os,
                                   const CRational & x );
   …
 };
std::ostream operator+ ( std::ostream & os,
                         const CRational & x );
#endif /* __CRational_h_49057405__ */
```

Moduly – hlavičkové soubory

- Pojmenování hlavičkových souborů:
  - volte pouze alfanumerické znaky anglické abecedy, číslice a podtržítko,
  - nepoužívejte národní znaky, speciální znaky, mezery, ani pokud to Váš kompilátor/OS umožňuje.
- Buďte velmi opatrní na malá/velká písmena:
  - Windows/DOS nerozlišují,
  - UNIX rozlišuje.
- Pokud používáte malá/velká písmena v názvech:
  - správně přepište jméno souboru do **`#include`**,
  - zkontrolujte jméno souboru na disku (Windows),
  - soubory zásadně přenášejte v archivu (zip, tgz, ...).

Moduly – implementační soubory

```
// priklad .cpp
#include <CRational.h>
#include <iostream>
using namespace std;  // zde ok
// priklad, lepsi jako private tridni metoda
static int gcd ( int a, int b )
 {
   if ( a == 0 || b == 0 ) return 1;
   while ( a != b )
    if ( a > b ) a -=b ; else b -= a;
   return ( a );
 }

CRational::CRational ( int n, int d )
 { int cd = gcd ( abs(n), abs(d) );
   numerator = n / cd; denominator = d / cd; }
```

Moduly – implementační soubory

- Implementační soubor - .c, .cpp:
  - definice metod a funkcí,
  - deklarace a definice tříd, funkcí, metod, konstant a proměnných, které nemají být vidět "ven z modulu",
  - lokální proměnné a funkce – označte **`static`**.
- Vkládá svůj hlavičkový soubor,
  - často vkládá i jiné hlavičkové soubory,
  - je nesmyslné vkládat jiné implementační soubory (.cpp, .c).
- Implementační soubor může bez omezení používat direktivy, které mají globální dopad – např:

```
using namespace std;
```

Moduly – oddělený překlad

```
/* sqr.h  - poskytuje funkci pro druhou mocninu */
#ifndef __sqr_h__3490304930493__
#define __sqr_h__3490304930493__

double sqr ( double x );

#endif  /* __sqr_h__3490304930493__ */

/* sqr.cpp - implementace modulu */
#include "sqr.h"

double sqr ( double x )
 {
   return ( x * x );
 }
```

Moduly – oddělený překlad

```
/* main.cpp */
#include <iostream>
#include <iomanip>
using namespace std;

#include "sqr.h"

int main ( int argc, char * argv [] )
 {
   double x;

   cout << "Zadej cislo" << endl;
   cin >> x;
   cout << x << "^2 = " << sqr ( x ) << endl;
   return ( 0 );
 }
```

Moduly – oddělený překlad

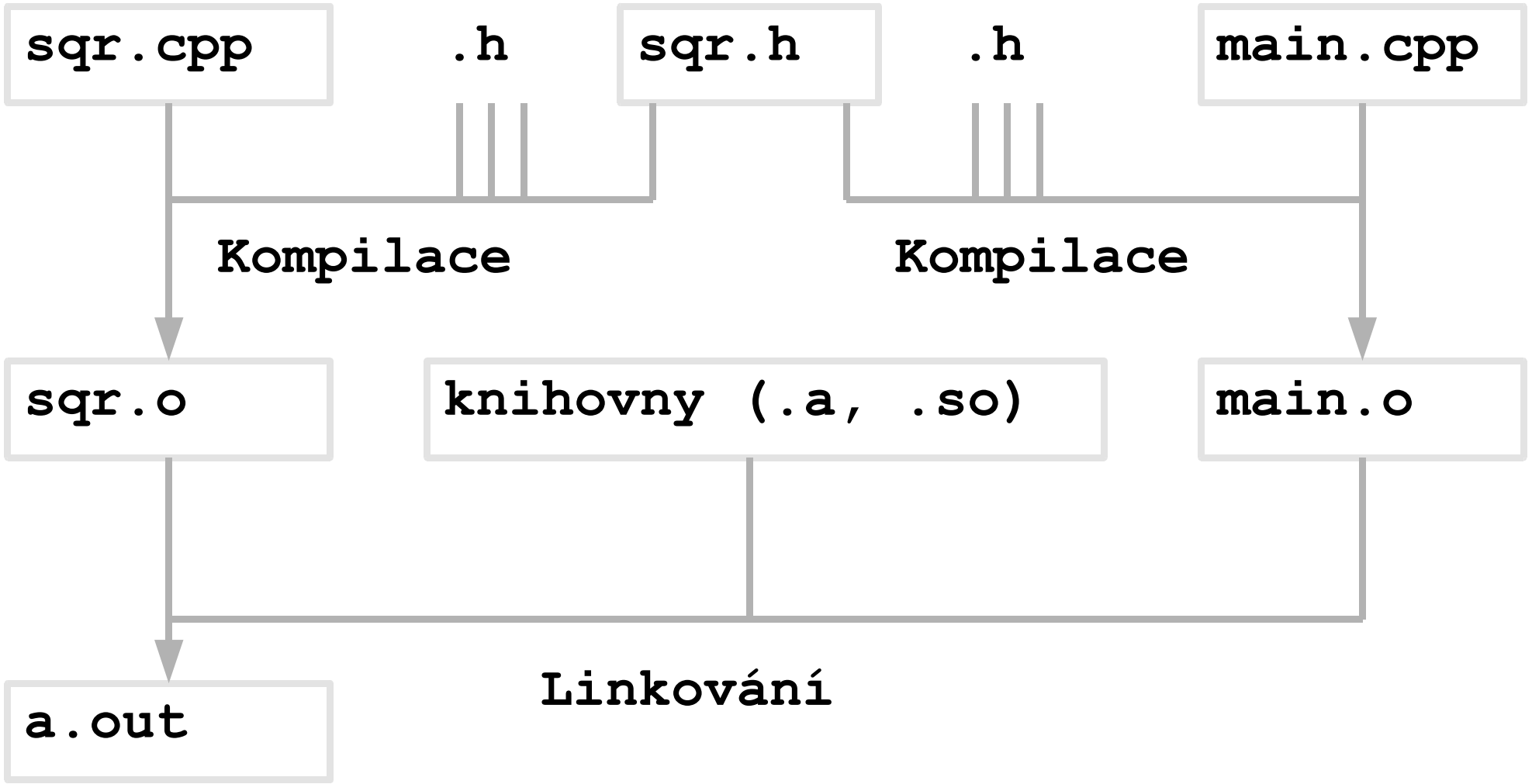
- Zkompilujete-li příklad, dostanete chybu:

```
> g++ -Wall -pedantic main.cpp

/tmp/cc4Oaaaa.o(.text+0x144):main.cpp: undefined
reference to `sqr(double)'
collect2: ld returned 1 exit status
```

- Linker hlásí, že funkce **`sqr`** neexistuje, nebyla definovaná.
- Přitom tato funkce je obsažena v implementačním souboru **`sqr.cpp`**.
- Proč to nefunguje?

Moduly – oddělený překlad



Moduly – oddělený překlad

- Kompilace:
  - ze zdrojových souborů a hlavičkových souborů vytvoří tzv. kód relativních adres (.o, .obj),
  - obsahuje instrukce CPU, ale nejsou dopočtené adresy,
  - .o/.lib obsahuje tabulku symbolů:
    - které jsou v .o souboru definované,
    - které je potřeba doplnit odjinud (**`extern`**).
- Linkování:
  - spojuje .o/.obj soubory a knihovny (.a/.so/.lib),
  - vytváří spustitelný program (dyn. knihovnu),
  - pro každý definovaný symbol (proměnnou, funkci, metodu) doplňuje adresu tam, kde je požadovaná,
  - kontroluje, že všechy odkazy jsou doplněné.

Moduly – oddělený překlad

- Oddělený překlad:
  - každý implementační (.cpp,.c) soubor se kompiluje nezávisle zvlášť,
  - kompilace je pomalá (zejména optimalizace),
  - soubory s kódem relativních adres se linkují,
  - linkování je rel. rychlé (doplňování tabulek).
- Výhody:
  - rychlejší překlad,
  - menší paměťové nároky překladače,
- Změna .c/.cpp – překlad pouze jednoho modulu a linkování (časté).
- Změna .h – překlad všech modulů, které jej vkládají, pomalejší (ale ne tak časté).

Moduly – oddělený překlad

- Správná kompilace ukázkového příkladu:

```
> g++ -Wall -pedantic -c -o main.o main.cpp
> g++ -Wall -pedantic -c -o sqr.o sqr.cpp
> g++ -o a.out sqr.o main.o
```

- První dva příkazy – kompilace modulů.
- Poslední příkaz – linkování.
- IDE zpravidla automaticky vytvářejí projekt (workspace, solution):
  - seznam zdrojových kódů,
  - parametry překladu,
  - parametry linkování.

Moduly – nesprávné řešení

```
/* main.cpp */
#include <iostream>
#include <iomanip>
using namespace std;

#include "sqr.cpp" // !! nesmyslne !!

int main ( int argc, char * argv [] )
 {
   double x;

   cout << "Zadej cislo" << endl;
   cin >> x;
   cout << x << "^2 = " << sqr ( x ) << endl;
   return ( 0 );
 }
```

Moduly – nesprávné řešení

- Ukázka sice funguje, ale popírá smysl modulů a odděleného překladu:
  - při změně libovolného .c/.cpp se musí kompilovat vše znovu (pomalá, náročné na paměť),
  - moduly nejsou oddělené,
  - chyba v jednom modulu se při překladu může hlásit v jiném modulu,
  - rozhraní definované v .h se nepoužívá.
- Nikdy nevkládejte .cpp do jiného .cpp.
- Nikdy neumisťujte definice funkcí/metod do vkládaných souborů.

Moduly – časté chyby

- Export proměnné z modulu:
  - máme modul a.cpp, v něm globální proměnnou **`X`**,
  - chceme zajistit, aby ostatní moduly měly k této proměnné přístup.
- Následující řešení nefunguje:

```
// a.h
…
int X; // !!
…
```

- Každý .cpp, který provede **`#include "a.h"`**, definuje proměnnou **`X`**.
- Proměnná **`X`** bude definovaná vícekrát.
- Linker oznámí chybu.

Moduly – časté chyby

- Správné řešení:
  - do hlavičkového souboru dáme deklaraci,
  - do a.cpp dáme definici proměnné **x**.
- Poznámka: je lepší se vyhnout exportům proměnných.

```
// a.h
…
extern int X;  // pouze deklarace
…

// a.cpp
#include "a.h"
…
int X;  // zde definice
…
```

Moduly – integrace C a C++ kódu

```
/* sqr.h  - poskytuje funkci pro druhou mocninu */
#ifndef __sqr_h__3490304930493__
#define __sqr_h__3490304930493__

double sqr ( double x );

#endif  /* __sqr_h__3490304930493__ */

/* sqr.c (ne .cpp) - implementace modulu */
#include "sqr.h"

double sqr ( double x )
 {
   return ( x * x );
 }
```

Moduly – integrace C a C++ kódu

```
/* main.cpp */
#include <iostream>
#include <iomanip>
using namespace std;

#include "sqr.h"

int main ( int argc, char * argv [] )
 {
   double x;

   cout << "Zadej cislo" << endl;
   cin >> x;
   cout << x << "^2 = " << sqr ( x ) << endl;
   return ( 0 );
 }
```

Moduly – integrace C a C++ kódu

- Takto připravený kód nebude fungovat i když správně zavoláme kompilaci na jednotlivé moduly.
- Linker hlásí neznámý symbol **`sqr`**.
- Příčina - C++ provádí tzv. name mangling (decoration):
  - do jména symbolu v .o souboru přidá informaci o typu parametrů,
  - rozlišení symbolů při přetěžování,
  - C toto nedělá (nemá přetěžování).
- Řešení - předepsat exportované funkci, že se mají použít pravidla C pro pojmenování.
- Používá se zejména u funkcí exportovaných z .so/.dll knihoven (každý kompilátor má jiná pravidla pro name mangling).

Moduly – integrace C a C++ kódu

```
/* sqr.h  - poskytuje funkci pro druhou mocninu */
#ifndef __sqr_h__3490304930493__
#define __sqr_h__3490304930493__

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

double sqr ( double x );

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif  /* __sqr_h__3490304930493__ */
```

Makefile, make

- Oddělený překlad ručně je pracný:
  - množství příkazů,
  - mnoho z nich je zbytečných (nedošlo ke změně),
  - ručně není zvládnutelné sledovat.
- **`make`** – program pro usnadnění překladu:
- **`Makefile`** – soubor s popisem pravidel pro překlad a popisem závislostí,
- **`make`** po spuštění:
  - načte Makefile,
  - zkontroluje, které soubory se změnily,
  - vyhodnotí závislosti,
  - zkompiluje/slinkuje pouze části, které se změnily.

Makefile

- Definice proměnných:

```
CXX=g++
CC=gcc
LD=g++
CFLAGS=-Wall -pedantic -Wno-long-long
```

- Sekce:

```
<cil>:<seznam predpokladu>
      prikaz
      prikaz
      …
```

- Cíl popisuje, co je výsledkem (zpravidla jméno souboru).
- Seznam předpokladů udává závislosti.

Makefile

- Příkazy v sekci se provedou:
  - pokud soubor s názvem cíle neexistuje,
  - nebo pokud je starší než libovolný z předpokladů,
  - kontrola aktuálnosti se provádí rekurzivně.
- Příkazy v sekci se provádějí, dokud končí bez chyb (návratová hodnota spuštěného progranu je 0).
- Příkazy v sekci jsou odsazeny tabulátorem.
- Zpravidla se přidává na první místo cíl s názvem **`all`**, ten se odkazuje na sestavení celého programu (případně všech programů).
- Zpravidla se přidává cíl **`clean`**, který vymaže všechny soubory, které se dají vytvořit kompilací/linkováním.

Makefile

```
CXX=g++
LD=g++
CFLAGS=-Wall -pedantic -Wno-long-long

all: a.out

a.out: main.o sqr.o
	$(LD) -o a.out main.o sqr.o
	strip --strip-unneeded a.out

main.o: main.cpp sqr.h
	$(CXX) $(CFLAGS) -c -o main.o main.cpp

sqr.o: sqr.cpp sqr.h
	$(CXX) $(CFLAGS) -c -o sqr.o sqr.cpp

clean:
	rm -f main.o sqr.o a.out
```

Makefile – použití

- sestavení prvního cíle (all):
  `make`
- vymazání všech mezivýsledků:
  `make clean`
- kompilace pouze **`main.o`**:
  `make main.o`

Makefile – vytváření

- Ruční vytváření Makefile – zdlouhavé, pracné, riziko chyb (chybějící závislosti, závislosti navíc).
- **`make`** zná makra a Makefile může být kratší.
- Automatizované vytváření závislostí:
  - gcc -MM,
  - automake,
  - configure skript,
  - podpora vytváření Makefile pro některé knihovny (qmake, xmkmf, ...).
- Většina IDE umí exportovat Makefile ze svého projektu.

Další nástroje

- Zobrazení tabulky symbolů z .o/.obj souboru:
  - UNIX: nm
  - Windows: dumpbin.exe

- Sestavení knihovny (.a/.lib) z .o/.obj souborů:
  - UNIX: ar
  - Windows: lib.exe

- Zobrazení závislostí na jiných dynamicky linkovaných knihovnách:
  - UNIX: ldd
  - Windows: depends.exe

Dotazy...

Děkuji za pozornost.