Modulární stavba programů

Aleš Smrčka

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole

smrcka@fit.vutbr.cz

13/12/21

Osnova přednášky

- Hlavičkové soubory
- Moduly
- Aplikační rozhraní (API)
- Zpětné volání (callback)
- Zásuvné moduly (plug-in)

Hlavičkové soubory

- Hlavičkové soubory (angl. header files) jako další část zdrojových souborů: soubory s příponou .h.
- Hlavička, protože bývají součástí “hlavičky” zdrojových souborů:

```
/**
 * Toto je ukazka programu.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include “obj_list.h”

int main(int argc, char *argv[])
{
    ...
   return 0;
}
```

Obsah hlavičkových souborů

- Důvod existence: společné (právě jedno) místo pro deklarace.
- Typicky si nevystačíme se základními datovými typy a operacemi (operátory) ⇒ tvoříme nové datové typy a nové operace/funkce.
- Pro znovupoužití (angl. reuse, český neinformatický překlad "opětovné použití"): přesunout deklarace do hlavičkového souboru.
- Hlavičkový soubor obsahuje deklarace:
  - datových typů (např. struct tm, FILE),
  - funkcí, tj. prototypy (např. printf, malloc, sqrt),
  - globálních proměnných (např. stdin, errno),
  - maker (např. assert),
  - identifikátorů (v rámci enum).

Hlavičkový soubor jako společné místo

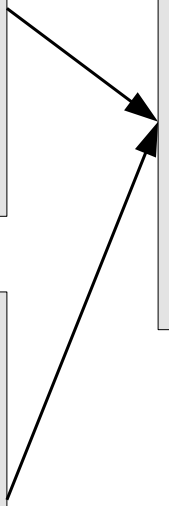
- Dva různé programy znovupoužívají stejné funkce nebo datové typy:

lekar-ui.c

```
#include “pacient.h”

...
void uprav_pacienta() {
  pridej_zaznam(...);
}
...
```

pacient-ui.c

```
#include “pacient.h”

...
void zobraz_moji_kontakty() {
  zobraz_kartu(...);
}
...
```

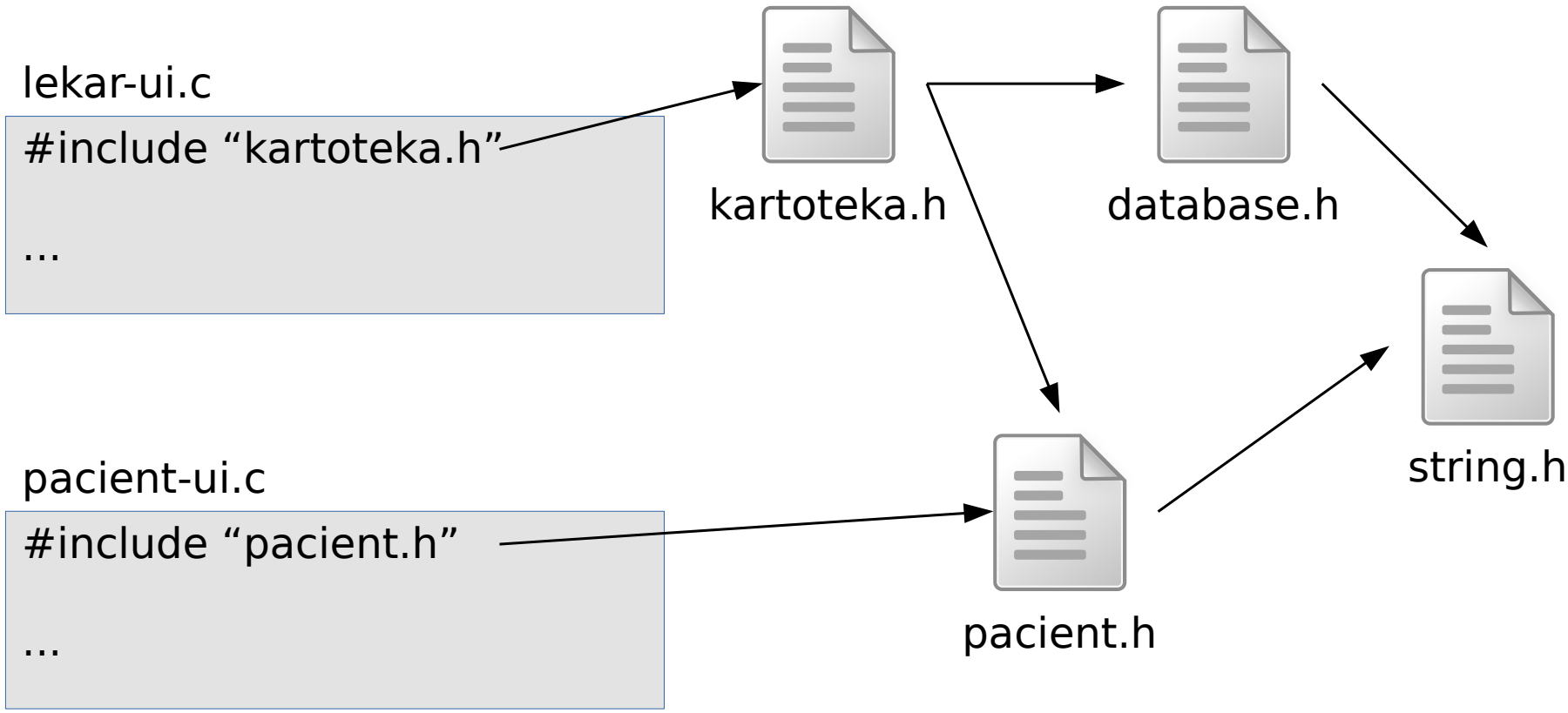


pacient.h

```
typedef struct { ... } TPacient;
typedef struct { ... } TZaznam;

void pridej_zaznam(TPacient *pac,
    TZaznam *zaznam);

void zobraz_kartu(TPacient *p);
```

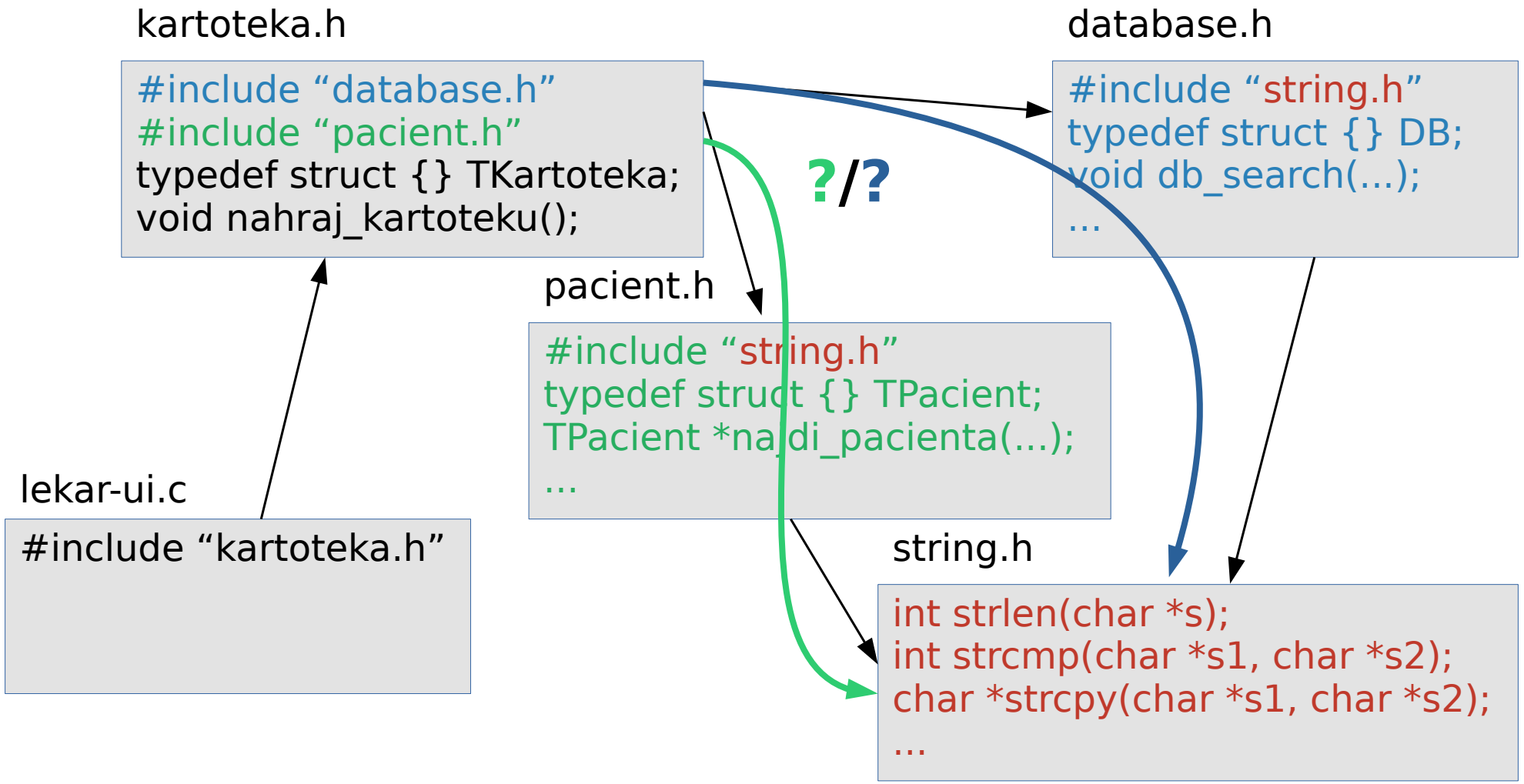
Hierarchie hlavičkových souborů

- Hlavičkové soubory mohou využívat deklarace v jiných hlavičkových souborech.
- Vzniká hierarchie hlavičkových souborů, např.:



Hierarchie hlavičkových souborů

- #include doslova zahrnuje obsah souboru.
- Vzniká problém s opětovnou deklarací:



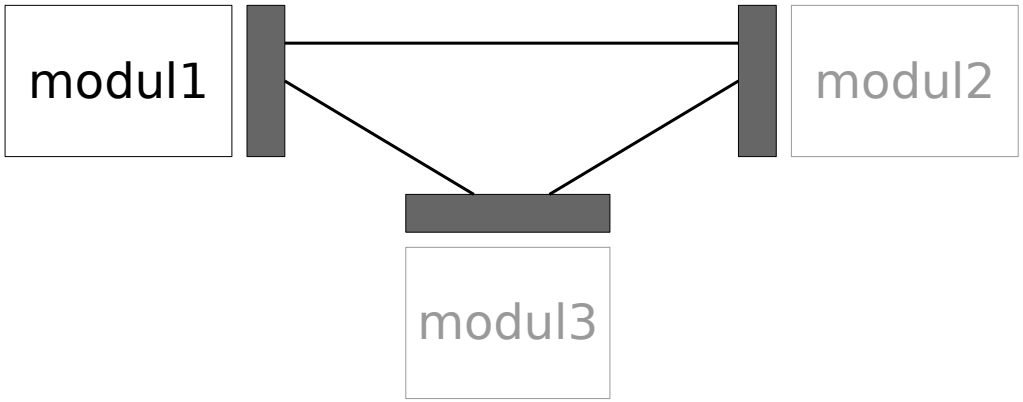
- Řešení kolizí v deklaraci: podmíněný překlad

```
#ifndef _STRING_H      // if not defined
#define _STRING_H 1

#include <stddef.h>    // size_t


...
size_t strlen(const char *s);

int strcmp(const char *s1, const char *s2);

char *stpcpy(char *dest, const char *src);
...

#endif
```

- Hlavičkové soubory
- Moduly
- Aplikační rozhraní (API)
- Zpětná volání (callback)
- Zásuvné moduly (plug-in)

Modul a modularita

- Modularita = vyšší stupeň strukturovaného programování, dekompozice velkých (složených) problémů na malé.
- Modul tvoří samostatnou ucelenou jednotku, kterou lze samostatně kompilovat a opakovaně využívat.
- Implementační detaily mohou být skryty (ve smyslu, že nezatěžují programátory).
- Moduly se používají prostřednictvím jejich rozhraní.



Modulární programování

- Modulární programování = technika vývoje softwaru
- Výhody:
  - srozumitelnosti (angl. clarity),
    - modul se zabývá pouze jedním problémem, navenek viditelné pouze jeho rozhraní,
  - spolehlivosti (angl. reliability),
    - jasně definovaná jednotka je lépe testovatelná,
  - udržovatelnosti (angl. maintainability),
    - malé části se lépe udržují, mění, upravují,
  - znovupoužití (angl. reusability),
    - efektiviní využití zdrojů, absence chyb z kopírování kódu,
  - možnost souběžného vývoje.
- Nevýhody:
  - problémy z integrace modulů,
    - skrývá vnitřní stav modulů (problém při špatně navrženém nebo využívaném rozhraní),
  - konkretizace modulů,
    - odvozené moduly se špatně udržují.

- Modul má své rozhraní = jak vystupuje navenek:
  - export,
    - množina funkcí, datových typů, identifikátorů, atd., které modul poskytuje.
  - import, někdy také *závislosti* (angl. depency),
    - množina funkcí, datových typů, identifikátorů, atd., které modul vyžaduje.
- Většinou je rozhraním myšlen pouze export modulu. Import modulu je dán instalací nebo pravidly pro sestavení nebo překlad modulu.

- Modul v C = přeložitelný celek (typicky jeden soubor).
- Modulem bývá často chápán binární soubor získaný jednorázovým překladem bez sestavení programu, tzv. objektový soubor (angl. object file) s příponou .o.
- Rozhraní modulu v hlavičkovém souboru.
  - Ne nutně. Někdy může modul vystupovat sám za sebe. Funkce, proměnné a konstantní proměnné lze překladačem automaticky zjistit (např. prostřednictvím tabulky symbolů binárních souborů).

Struktura modulu v jazyce C

- Hlavičkové soubory (.h) obsahují:
  - export modulu, tj. deklarace pro vybrané definice z .c souboru.
  - dokumentaci modulu (v podobě dokumentačních komentářů),
    - tedy vše o tom, „jak se to používá“.
  - minimální závislosti na dalších hlavičkových souborech.
- Zdrojové soubory (.c) obsahují:
  - definice funkcí,
  - deklarace globálních proměnných a konstantních proměnných,
    - tedy to, co “zabírá paměť”.
  - závislosti na dalších hlavičkových souborech.

Struktura modulu v jazyce C : Příklad

patient.h

```
#ifndef _PATIENT_H
#define _PATIENT_H
...
typedef struct { ... } Patient;
typedef struct { ... } FileRecord;
...
int patient_set_name(Patient *p, char *new_name);
int patient_set_address(Patient *p, char *address);
int patient_append_record(Patient *p, FileRecord *record);
...
#endif
```

export

patient.c

```
#include “patient.h”
#include <string.h>
#include <stdlib.h>
static void _free_name(Patient *p)
{
   ...
}
int patient_set_name(Patient *p, char *new_name)
{
   _free_name(p);
   p->name = strdup(new_name);
   except_mem_error(p->name == NULL);
}
...
```

import

skryté pomocné funkce

definice exportu

- Je vhodné ve zdrojovém souboru (.c) zahrnout hlavičkový soubor kvůli typové kontrole.

patient.h

```
#ifndef _PATIENT_H
#define _PATIENT_H
...
int patient_set_name(Patient *p, char *new_name);
...
#endif
```

patient.c

```
#include “patient.h”
#include <string.h>
#include <stdlib.h>
...
int patient_set_name(char *new_name, Patient *p)
{
   _free_name(p);
   p->name = strdup(new_name);
   except_mem_error(p->name == NULL);
}
```

- Jazyk C je připraven pro modulární překlad.
- Překlad programu = překlad modulů + sestavení
  - angl. build = compile + link
- Z hlediska vývojového prostředí urychlení celkového překladu,
  - překládají se pouze ty moduly, jejichž zdrojové kódy (nebo závislosti) se změnily,
  - sestavení vždy nad všemi moduly (sestavení je ale rychlé).

Moduly v jazyce C : Překlad

```
foo.c:
#include “foo.h”

foo.h:
#include <string.h>

bar.c:
#include “bar.h”
#include <stdlib.h>

bar.h:
#include “foo.h”

$ cc -MM *.c
bar.o: bar.c bar.h foo.h
foo.o: foo.c foo.h

$ make
cc    -c -o foo.o foo.c
cc    -c -o bar.o bar.c
cc foo.o bar.o -o program
```

```
$ touch bar.c

$ make
cc    -c -o bar.o bar.c
cc foo.o bar.o -o program
```

```
$ touch foo.h

$ make
cc    -c -o foo.o foo.c
cc    -c -o bar.o bar.c
cc foo.o bar.o -o program
```

- Hlavičkové soubory
- Moduly
- Aplikační rozhraní (API)
- Zpětná volání (callback)
- Zásuvné moduly (plug-in)

Aplikační rozhraní

- Aplikační rozhraní = množina procedur a protokolů pro tvorbu aplikací.
  - angl. API = Application Programming Interface,
  - rozhraní k většímu softwarovému celku, např. knihovna, subsystém, databázový systém, operační systém, apod. (obecně komponenta),
  - ucelený popis (formální i neformální), jak používat danou komponentu.
- Příklady:
  - Standardní knihovna jazyka C
  - MariaDB/MySQL C API
  - OpenGL API
  - Windows API
  - REST API
  - Standardní knihovna pro vlákna v C++
  - Unity3D Scripting
  - Arduino API
  - ...

- Komponenta s aplikačním rozhraním v C se nazývá knihovna:
  - např. standardní knihovna (libstdc), matematická knihovna (libm).
  - Z hlediska sestavení rozlišujeme knihovny statické (přípona .a) a dynamické (přípony .so/.dll).
    pozn.: Tvorba a užívání knihoven je nad rámec IZP.
- Rozhraní (tj. export knihovny) je definováno hlavičkovým souborem.
- Typická je hierarchie několika hlavičkových souborů a jeden kořenový, např.
  - math.h: math-finite.h mathcalls.h ...
  - gtk.h: gtkbutton.h gtkentry.h gtkdialog.h

- Nutnou součástí rozhraní je dokumentace zahrnující:
  - popis jednotlivých částí,
    - funkce, datové typy, konstanty,
  - závislosti volání,
    - zda je nutné datovou strukturu inicializovat, nastavit a jak, uvolnit, životní cyklus dat,
  - předpoklady,
    - např. strlen očekává patný ukazatel,
  - ale také (jako každá softwarová komponenta):
    - autor, licence, udržovatele projektu resp. originálního autora (angl. upstream),
    - reference na další závislosti, příručky, příklady použití, ...

- Aplikační rozhraní abstrahuje implementační detaily.
- Obsahuje pouze funkce, proměnné a datové typy minimálně nutné pro danou funkcionalitu.
- Speciální pozornost se dává bezpečnosti (angl. security):
  - na druhé straně API může být nedůvěryhodná komponenta,
  - vstupní data musí být ošetřena a “zdůvěryhodněna” ještě před samotnými operacemi.
- Tvorba dokumentace API pomocí např. Doxygen (viz předchozí přednáška).

- Hlavičkové soubory
- Moduly
- Aplikační rozhraní (API)
- Zpětná volání (callback)
- Zásuvné moduly (plug-in)

- Zpětná volání (angl. callback) slouží pro definici chování programu až za běhu.
  - pozn. rozlišujeme dobu kompilace a dobu běhu programu (angl. compile-time vs. run-time).
- Zpětná volání pro dvojí účel:
  1. generické algoritmy využívající proměnné operace,
     - algoritmus je hromadný nad funkcemi/operacemi, nejen nad daty, např.:
       - numerické výpočty (matematická funkce je parametrem algoritmu),
       - algoritmy vyhledání a řazení (kritérium je dané funkcí jako parametr algoritmu),
  2. neznalost požadované funkcionality v době překladu.

Zpětná volání v C

- Funkce v modulu/programu zabírá paměť:
  - Funkce resp. její definice obsahuje příkazy => instrukce pro procesor. Instrukce jsou binárně kódované a uloženy v paměti (viz inspektor binárních souborů).
- Ukazatel na funkci má hodnotu adresy první její instrukce.
- Všechny funkce splňují konvenci volání standardu jazyka C:
  - využívají se registry procesoru a zásobník,
  - jakým způsobem se předávají argumenty,
  - jakým způsobem se předává návratová hodnota,
  - nad rámec IZP.

Zpětná volání v C

- V jazyce C datový typ ukazatel na funkci.
- Je nutné dodržet typ funkce, tj. počet a datové typy parametrů funkce a datový typ návratové hodnoty.
- Proměnnou typu ukazatel na funkci lze volat.
- Příklad:

```
typedef double (*Callback)(double x);
double find_local_min(Callback fce,
          double a, double b);

void foo()
{
    Callback bar = sqrt;
    find_local_min(bar, -1, +1);
    find_root(log, 0.5, bar(4.0));
}
```

Zpětná volání v C : Příklad

```
#include <stdlib.h>

struct obj_t { int id; }

int obj_sort_compar(const void *a, const void *b)
{
    const struct obj_t *o1 = a;
    const struct obj_t *o2 = b;
    if (o1->id < o2->id) return -1;
    if (o1->id > o2->id) return 1;
    return 0;
}

void sort_cluster(struct obj_t *oarr, unsigned size)
{
    qsort(oarr, size, sizeof(struct obj_t),
          &obj_sort_compar);
}
```

- Hlavičkové soubory
- Moduly
- Aplikační rozhraní (API)
- Zpětná volání (callback)
- Zásuvné moduly (plug-in)

- Zásuvný modul (angl. plug-in) je programový modul (softwarová komponenta), která rozšiřuje základní funkcionalitu programu.
- Program musí být připraven na rozšíření formou zásuvných modulů => musí poskytovat aplikační rozhraní pro zásuvné moduly.
- Zásuvné moduly dělíme podle následujících kritérií:
  - podle životního cyklu zásuvného modulu,
  - podle typu,
  - podle principu komunikace.

Zásuvné moduly : Dělení podle životního cyklu

- Při spuštění programu jsou načteny všechny knihovní závislosti (toto je zajištěno podprogramem zvaným dynamic loader).
  - výhoda: efektivní načtení, menší chybovost v API, přenositelné,
  - nevýhoda: při překladu nutné znát počet zásuvných modulů, každý modul musí mít jiné rozhraní.
- Při běhu programu jsou knihovny vyhledány programem (např. na základě konfigurace programu nebo průchodem adresáře souborů):
  - soubor musí odpovídat konvencím zásuvného modulu (např. pojmenování souboru nebo umístěním ve speciálním adresáři),
  - načtení knihovny do paměti (dlopen, knihovna dl - POSIX, LoadLibrary - MS Windows),
  - zjištění ukazatelů na koncové funkce knihovny podle API zásuvných modulů (dlsym/GetProcAddress),
  - zavolání inicializační funkce,
  - práce se zásuvným modulem (typicky pomocí zpětných volání),
  - zavolání finalizační funkce a uvolněním knihovny z paměti (dlclose/FreeLibrary).

Zásuvné moduly : Dělení podle typu

- Různé aplikace nebo projekty (webové prohlížeče, informační systémy) mohou mít různou terminologii.
- Obecné však platí:
  - doplněk (angl. add-on),
    - rozšíření/doplnění základní funkcionality programu,
  - rozšíření (angl. extension),
    - rozšíření programu takové, že program lze použít i k jinému než původnímu účelu,
  - applet, servlet,
    - poskytnutý prostor (v okně aplikace nebo ve webovém rozhraní) pro funkcionalitu definovanou externím modulem,
  - zásuvný modul (angl. plug-in),
    - obecný pojem pro programově rozšiřitelnou funkcionalitu,
  - sdílená knihovna (angl. shared library),
    - technologie použitá pro rychlé operace se zásuvnými moduly.

Zásuvné moduly : Principy komunikace

Dělení podle principu komunikace se zásuvným modulem:

- Statický komunikační protokol,
  - malý počet funkcí API, předem známá sekvence, např.:

```
Callback foo_extension = init_plugin(plugin);
   do_something_with(foo_extension);
   fini_plugin(plugin);
```

- Konfigurovatelná zpětná volání,
  - tzv. registrace zpětných volání (callback registration),
  - typické u grafických uživatelských rozhraní:
    - onclick, onhover, onkeypressed, ...
- Veřejné vysílání (angl. broadcast),
  - jednotný komunikační kanál - sběrnice,
  - události (zprávy) jsou posílány na sběrnici,
  - příjemci (např. zásuvné moduly) mohou na událost reagovat posláním další události,
  - API nemusí definovat události, v krajním případě je program pouze poskytovatel sběrnice, např. dbus.

A to je konec…