

Několik tipů na závěr

1258 Několik slov o optimalizaci



Obvykle se rozlišují tři úrovně optimalizace:

1. *Nejnižší úroveň.* Překladač při ní volí mezi instrukcemi, které mají stejný účinek, ale zabírají jiný počet bajtů a jiný počet taktů procesoru. Například uložení hodnoty 0 do registru EAX lze na procesorech Intel dosáhnout instrukcí
`XOR EAX, EAX`
nebo
`MOV EAX, 0`
První z těchto instrukcí zabírá v paměti třikrát tolik bajtů než druhá, je ale dvakrát rychlejší.
Tuto úroveň používají překladače zpravidla vždy, bez ohledu na to, zda jsou optimalizace povoleny nebo zakázány.
2. *Střední úroveň.* Jde o optimalizace typu eliminace společných podvýrazů, přesunutí výpočtu, který nezávisí na parametrech cyklu, před jeho tělo, ukládání proměnných do registrů atd. Na této úrovni může překladač využít nápovědu od programátora, jako je deklarace registrových proměnných, deklarace funkcí inline atd.
3. *Nejvyšší úroveň.* Tato úroveň je výlučnou doménou programátora. Jedině programátor může vědět něco bližšího o rozsahu a povaze dat, se kterými bude jeho program pracovat, a podle toho volit algoritmy, které se použijí.

O skutečné použitelnosti programu rozhoduje vlastně pouze třetí úroveň. Jestliže programátor použije k řazení pole o miliardě prvků bublinkové třídění, protože o jiném algoritmu jakživ neslyšel a neví, že standardní knihovna mu nabízí již hotové funkce, které implementují quicksort, nesmí se divit, že jeho program běží opravdu pomalu, a neměl by svádět vinu na neschopnost překladače účinně optimalizovat.

1259 Standard popisuje pozorovatelné chování



Standardy jazyků C a C++ neurčují způsob implementace funkcí a datových struktur, které jsou popsány v knihovnách; neurčují ani způsob implementace jiných konstrukcí, které popisují.

Tyto standardy vycházejí z představy abstraktního počítače a popisují jeho *pozorovatelné chování*. To znamená, že například nikde není řečeno, že funkce `sort<>()` musí implementovat algoritmus *quicksort*. Dozvíme se pouze, že má jít o algoritmus, jehož průměrná složitost – potřebný počet operací – je úměrný $N \cdot \ln N$, kde N je počet prvků tříděné posloupnosti, a že vyžaduje iterátory s náhodným přístupem. Tomuto požadavku vyhovuje vedle *quicksortu* třeba také algoritmus *introsort* (introspektivní třídění), ale i třídění haldou.

Podobně z popisu kontejneru `list<>` se dozvíme, že má určité operace, podporuje obousměrné iterátory atd.; jinými slovy, že podle chování ho nejde rozlišit od obousměrně zřetěženého seznamu. To ale neznamená, že nutně musí být jako obousměrně zřetěžený seznam implementován. Implementátoři mohou přijít na jiný způsob, jak dosáhnout požadovaného pozorovatelného chování, a ten použít.

1260 Ukazatele na metody



Běžné ukazatele na funkce nelze použít pro práci s metodami objektových typů. Proto nabízí C++ tzv. *třídní ukazatele* (member pointers). Třídní ukazatel `F` na metodu třídy `T` bez parametrů vracující typ `X` deklarujeme zápisem

```
X (T::*F)();
```

Struktura deklarace je podobná jako struktura deklarace obyčejného ukazatele na funkci, pouze před identifikátorem deklarované proměnné je kromě hvězdičky ještě jméno třídy, do níž tato metoda patří, připojené rozlišovacím operátorem `::`. Závorky okolo kvalifikovaného identifikátoru ukazatele jsou nezbytné.

1261 Jak získáme ukazatel na metodu



Ukazatel na metodu získáme pomocí operátoru `&` aplikovaného na jméno metody kvalifikované jménem třídy pomocí operátoru `::`.

Vezměme např. třídu reprezentující celá čísla

```
class numero {
    int n;
public:
    numero(int nn): n(nn){}
    int sqr() {return n*n;}
    // a další metody
};
```

Ukazatel na metodu `sqr()` získáme příkazem

```
int (numero::*F)(void) = &numero::sqr;
```

Metoda musí být v místě tohoto příkazu dostupná (nesmějí její použití zakazovat přístupová práva). Operátor `&` nelze vynechat.

1262 Třídní ukazatel na datové složky



Datová složka instance je proměnná jako každá jiná, takže pro práci s nimi můžeme používat obyčejné ukazatele. Vedle toho ale můžeme použít i *třídní ukazatele*. Deklarujeme je podobně jako obyčejný ukazatel, před znak `*` však připojíme operátorem `::` jméno třídy; ukazatel pak bude ukazovat na složky instancí této třídy.

Například ukazatel na složku typu `int` instance třídy `numero` z tipu 1161 můžeme deklarovat takto:

```
int numero::*p;
```

1263 Jak získáme hodnotu třídního ukazatele na datovou složku



Ke získání třídního ukazatele na datovou složku použijeme opět operátor `&`, který aplikujeme na jméno složky kvalifikované pomocí operátoru `::` jménem třídy. To znamená, že ukazatel na složku `n` třídy `numero` z tipu 1161 získáme příkazem

```
int numero::*p = &numero::n;
```

1264 Dereferencování třídního ukazatele na metodu



Chceme-li zavolat metodu, na kterou ukazuje třídní ukazatel `F`, musíme mít k dispozici buď instanci, pro kterou tuto metodu zavoláme, nebo ukazatel na takovou instanci. Máme-li k dispozici instanci, použijeme binární operátor `.`, kterému jako levý operand zadáme onu instanci a jako pravý operand třídní ukazatel na metodu. Instanci spolu s třídním ukazatelem uzavřeme do závorek a za tento výraz připojíme operátor volání funkce – závorky obsahující v případě potřeby skutečné parametry.

Máme-li ukazatel na instanci, použijeme operátor `->`, kterému jako levý operand zadáme onen ukazatel na instanci a jako pravý operand třídní ukazatel na metodu. Vše ostatní je stejné.

Je-li `cis` instance třídy `numero` z tipu 1161, použijeme metodu, na kterou ukazuje `F`, v příkazu

```
int i = (cis.*F());
```

Kdybychom místo instance `cis` měli ukazatel `pcis` na instanci tohoto typu, napsali bychom

```
int i = (pcis->*F());
```

1265 Dereferencování třídního ukazatele na datovou složku



Máme instanci `cis` třídy `numero` z tipu 1161 a ukazatel získaný příkazem

```
int numero::*p = &numero::n;
```

ukazuje na datovou složku typu `int`. Datové složce instance `cis`, na niž ukazuje `p`, přiřadíme hodnotu příkazem

```
cis.*p = 54;
```

Budeme-li místo instance třídy `numero` mít ukazatel `pcis` na tuto instanci, napíšeme

```
pcis->p = 54;
```



Upozornění: Priorita operátorů `.`, `*` a `->` je nižší než priorita operátoru `*` pro dereferencování běžných ukazatelů.

1266 A co statické metody a složky?



Jazyk C++ nedovoluje získat třídní ukazatel na statickou metodu, konstruktor nebo destruktork. Nelze také získat třídní ukazatel na statickou datovou složku.

Na druhé straně statické metody jsou vlastně volné funkce ukryté do třídy, a tak s nimi můžeme pracovat pomocí obyčejných ukazatelů na funkce.

1267 Třídní ukazatele a pozdní vazba



Při dereferencování třídního ukazatele pomocí operátorů `.`, `*` a `->` můžeme uvést nejen instanci typu uvedeného v deklaraci, ale i jakéhokoli odvozeného typu. Je samozřejmé, že při volání virtuální metody pomocí třídního ukazatele se uplatní pozdní vazba.

1268 Třídní ukazatele: pohled pod pokličku



Důvodem, proč nelze pro práci s metodami používat běžné ukazatele na funkce, je, že metoda má vždy jeden skrytý parametr navíc – ukazatel `this`. Tím, že k dereferencování třídního ukazatele použijeme instanci nebo ukazatel na ni, zajistíme, že metoda tento skrytý parametr dostane.

Třídní ukazatel na datovou složku si můžeme zjednodušeně představit jako ukazatel, obsahuje relativní adresu datové složky vzhledem k začátku instance; neříkáme však které. Tím, že operátoru `.` nebo `->` předáme instanci nebo ukazatel na ni, řekneme, ve které instanci se má tato relativní adresa uplatnit.

Poznamenejme, že tento popis třídních ukazatelů je hodně zjednodušený, neboť třídní ukazatel se musí umět vyrovnat i s komplikovanou strukturou virtuálních potomků.

1269 Jak zjistit aktuální čas



Chceme-li zjistit aktuální čas, můžeme použít funkci `time()` definovanou v hlavičkovém souboru `<time.h>`, jež vrací hodnotu typu `time_t` (jde o počet sekund od půlnoci 1.1.1970). Tuto funkci předáme jako parametr ukazatel na proměnnou, do níž chceme výsledek uložit; předáme-li hodnotu 0, funkce výsledek neuloží, funkce ho pouze vrátí.

Pomocí funkce `gmtime()` můžeme převést hodnotu typu `time_t` na hodnotu typu `struct tm`, jež obsahuje složky představující den, měsíc, rok, hodinu, minutu atd. S tím jsme se setkali v tipu 1195.

1270 Jak zjistit čas procesu



Chceme-li určit čas, který uplynul od spuštění aktuálního procesu, použijeme funkci `clock()` definovanou v hlavičkovém souboru `<time.h>`. Tato funkce je bez parametrů a vrací hodnotu typu `clock_t`, což je zpravidla celočíselný typ (`long`) obsahující počet taktů od spuštění procesu. Chceme-li zjistit čas v sekundách, vydělíme získanou hodnotu konstantou `CLOCKS_PER_SEC`.

1271 Přesnější měření času (nestandardní řešení)



Pro přesnější měření času – např. při různých testech – můžeme často použít funkci `ftime()` (v některých implementacích se jmenuje `_ftime()`). Jedná se o nestandardní, ale poměrně běžné rozšíření; najdeme ji v hlavičkovém souboru `<sys/timeb.h>`.

Tato funkce je typu `void` a jako parametr očekává ukazatel na strukturu `_timeb`, do níž uloží aktuální lokální čas.

Struktura `_timeb` má čtyři složky:

- Složka `dstflag` je nenulová, platí-li právě letní čas.
- Složka `millitm` obsahuje část časového údaje vyjádřenou v milisekundách.
- Složka `time` obsahuje čas v sekundách od 1. ledna 1970 UTC.
- Složka `timezone` obsahuje rozdíl mezi UTC a lokálním časem v minutách.

1272 Typově generická makra V C99



Pro mnohé z matematických funkcí definovaných v hlavičkových souborech `<math.h>` a `<complex.h>` jsou ve standardní knihovně C99 v hlavičkovém souboru `<tgmath.h>` definována tzv. *typově generická makra* (type-generic macros). Představují jakousi náhradu za přetěžování funkcí z C++. Jestliže zavoláme některou z uvedených funkcí pomocí „základního“ identifikátoru (který se normálně používá pro parametr typu `double`), určí se skutečně volaná funkce podle typu skutečného parametru.

To znamená, že napíšeme-li např. `sin(x)`, zavolá se `sin()`, `sinf()`, `sinl()`, `csin()`, `csinf()` nebo `csinl()` podle typu skutečného parametru.

Je-li skutečný parametr celočíselný, zavolá se verze pro typ `double`.

1273 Generování náhodných čísel



V knihovně jazyka C najdeme funkci `rand()` bez parametrů, jež vrací náhodné celé číslo s rovnoměrným rozdělením na množině celých čísel od 0 do `RAND_MAX` (tedy každá z hodnot v tomto rozmezí je stejně pravděpodobná). Tato konstanta by měla být rovna nejméně 32767.

Ve skutečnosti jde o pseudonáhodnou posloupnost, tedy o členy deterministické posloupnosti s velmi složitým chováním. Jestliže použijeme tuto funkci bez dalších úprav, dostaneme při každém běhu programu stejnou posloupnost těchto čísel.

1274 Randomizace náhodné posloupnosti



Chceme-li získat při každém běhu jinou posloupnost náhodných čísel, musíme tuto náhodnou posloupnost randomizovat. K tomu slouží funkce `srand()`, které předáme novou počáteční hodnotu náhodné posloupnosti. Typicky se používá hodnota odvozená od aktuálního času, např.

```
srand( (unsigned)time( NULL ) );
```

1275 Generátory náhodných čísel v C++: hudba budoucnosti



Generátor nabízený standardem jazyka C nepatří mezi nejkvalitnější – zpravidla má poměrně krátkou periodu. Navíc standard nehovoří o tom, jaký algoritmus bude pro generování použit, a zpravidla to dodavatelé také neuveřejňují.

Jazyk C++ v současné podobě tento problém neřeší vůbec, tj. nenabízí žádné další nástroje. Mezi očekávanými rozšířeními, která by měla být součástí nového standardu, je ovšem i skupina tříd, které implementují některé známé velmi kvalitní generátory pseudonáhodných čísel, mezi jiným i generátor Mersenne Twister. Měly by být k dispozici šablony `linear_congruential<>`, `mersenne_twister<>` a `subtract_with_carry<>`.

Dále by měly být k dispozici připravené generátory Bernoulliho rozdělení, binomického, geometrického, Poissonova, normálního, exponenciálního a rozdělení gama. Podrobněji viz [OČRo].

1276 Náhrada operátorů



Standard jazyka C++ počítá s tím, že na některých klávesnicích nejsou běžně dostupné znaky `&`, `|` a některé další, a proto umožňuje nahradit některé operátory klíčovými slovy.

Tyto náhrady ukazuje následující tabulka.

<code>and</code>	<code>&&</code>
<code>bitor</code>	<code> </code>
<code>not_eq</code>	<code>!=</code>
<code>xor</code>	<code>^</code>
<code>and_eq</code>	<code>&=</code>
<code>compl</code>	<code>~</code>
<code>or</code>	<code> </code>
<code>xor_eq</code>	<code>^=</code>
<code>bitand</code>	<code>&</code>
<code>not</code>	<code>!</code>
<code>or_eq</code>	<code> =</code>

I když tyto náhrady předepisuje standard, většina překladačů tyto identifikátory nezná. Chceme-li je používat, musíme do programu vložit direktivou `#include` hlavičkový soubor, který se obvykle jmenuje `<ciso646>` nebo `<iso646.h>`; v něm jsou tyto náhrady definovány.

1277 Co jsou to signály



Dojde-li v programu k některé z možných událostí, je program přerušen a dostane *signál* o tom, co se stalo. Signály mají přidělena čísla, která říkají, o co jde. Makra závěšující identifikátory pro tato čísla jsou definována v hlavičkovém souboru `<signal.h>`. Standard jazyka C požaduje alespoň následující hodnoty:

Signál	Význam
SIGABRT	abnormální ukončení programu způsobené např. voláním funkce <code>abort()</code>
SIGFPE	chybná aritmetická operace, jako je dělení nulou nebo přetečení; závisí ovšem na nastavení výpočetního prostředí – v dnešních překladačích to zpravidla chybu nezpůsobí a signál se neposílá
SIGILL	chybně přeložená funkce, např. neplatná instrukce
SIGINT	bylo přijato interaktivní upozornění
SIGSEGV	chyba při přístupu do paměti
SIGTERM	program obdržel žádost o ukončení

1278 Obsluha signálu



Pro všechny signály je definována jakási implicitní obsluha. Programátor ovšem může předepsat svou vlastní pomocí funkce `signal()` definované v hlavičkovém souboru `<signal.h>`. Této funkci předá jako první parametr číslo signálu, jehož obsluhu chce definovat, a jako druhý parametr ukazatel na funkci, která má být nastavena jako nová obsluha.

Funkce `signal()` vrátí ukazatel na předchozí obsluhu tohoto signálu.

Obsluha signálu je buď funkce typu `void` s jedním parametrem typu `int` nebo jedno z následujících maker:

SIG_DFL	příkazuje použít implicitní obsluhu
SIG_ERR	znamená chybu
SIG_IGN	příkazuje signál ignorovat

1279 Vyvolání signálu



Chceme-li vyvolat signál, použijeme funkci `raise()`, které předáme číslo signálu. Tato funkce vrátí 0 v případě úspěchu, nenulovou hodnotu v opačném případě.

Je-li volána obsluha signálu, skončí funkce `raise()` až poté, co skončí tato obsluha.

1280 Čtení z paměťového proudu



Máme znakový řetězec `lajna` typu `string`, který začíná celým číslem (číslem řádku – mohou mu předcházet mezery). Chceme toto číslo zjistit.

Jedním z možných řešení je použít vstupní paměťový proud `istream` definovaný v hlavičkovém souboru `<sstream>`.

```
istream vstup(lajna);
int n;
vstup >> n;
```

Kdybychom pracovali se širokými znaky, použili bychom proud typu `wistream`.

1281 Zápis do paměťového proudu



Potřebujeme získat znakovou reprezentaci (typu `string`) dat z programu. Jednou z možností je použití výstupního paměťového proudu typu `ostream`:

```
ostream VEN;
VEN << "celá hodnota je " << n;
```

Data uložená v tomto proudu získáme pomocí metody `str()`:

```
string text = VEN.str();
```

1282 Chceme přejmenovat soubor



Chceme-li přejmenovat existující soubor, zavoláme funkci `rename()` deklarovanou v hlavičkovém souboru `<stdio.h>`. Tato funkce očekává jako první parametr řetězec obsahující „staré“, tedy současné jméno souboru a jako druhý parametr řetězec obsahující „nové“, tedy požadované jméno. Vrábí 0, jestliže se operace podaří, a nenulovou hodnotu v případě chyby.

Například příkazem

```
i = rename("data.txt", "data.dta");
```

přejmenujeme soubor "data.txt", ležící v aktuálním adresáři, na "data.dta".

Operace se nepodaří, například když v udaném adresáři neexistuje soubor s uvedeným jménem. Jestliže naopak existuje soubor s požadovaným novým jménem, závisí výsledek na implementaci; v mém překladači to způsobí, že se operace nepodaří.

1283 Chceme odstranit soubor



Chceme-li odstranit existující soubor, použijeme funkci `remove()` deklarovanou v hlavičkovém souboru `<stdio.h>`. Tato funkce očekává jako parametr řetězec obsahující jméno odstraňovaného souboru. Vrábí 0, jestliže se operace podaří, a nenulovou hodnotu v případě chyby.

Například příkazem

```
int i = remove("data.dta");
```

smažeme soubor "data.dta" ležící v aktuálním adresáři.

1284 Potřebujeme dočasný soubor



V programu občas potřebujeme dočasné pomocné soubory. V takové situaci lze využít služeb funkce `tmpfile()`, která je bez parametrů a vrací ukazatel na strukturu `FILE`. Tato funkce vytvoří pomocný soubor se jménem, které se bude lišit od jmen všech souborů v daném prostředí. Tento soubor bude otevřen v režimu "wb+" a při řádném ukončení programu bude automaticky odstraněn.

Tato funkce je deklarována v hlavičkovém souboru `<stdio.h>`.

1285 Jak vytvořit platné jméno dočasného souboru



Někdy potřebujeme pomocný soubor otevřený v jiném režimu, než je "wb+". Pak nám nepomůže funkce `tmpfile()`, ale funkce `tmpnam()`, jež generuje jméno, které se bude lišit od jmen všech souborů v daném prostředí. Jako parametr jí předáme dostatečně dlouhé pole znaků; funkce `tmpnam()` do něj uloží vytvořené jméno a zároveň tento řetězec vrátí.

Jestliže jméno z jakýchkoli důvodů nevytvoří, vrátí 0.

Jestliže této funkci předáme 0, vrátí ukazatel na vnitřní statické pole s výsledkem.

1286 Co zavádí tato deklarace?



Na závěr jedno malé cvičení: Zkuste si určit, co zavádí následující deklarace. Než se do toho pustíte, zakryjte si odpověď pod následujícím řádkem:

```
void (*abc(int def, void (*ghi)(int)))(int);
```

Jde o deklaraci funkce `abc()`, která vrací ukazatel na funkci typu `void` s jedním parametrem typu `int`. Prvním parametrem funkce `abc()` je číslo typu `int`, druhým parametrem je ukazatel na funkci typu `void` s jedním parametrem typu `int`. (Ve skutečnosti jde o prototyp funkce `signal()`, jen se změněnými identifikátory.)