

0x200

PROGRAMOVÁNÍ

Hacker je termín, kterým se označují nejenom ti, kdo píší kód, ale také ti, kdo kód exploitují. I když mají obě skupiny hackerů různé finální cíle, používají při řešení úloh podobné techniky. Protože těm, kdo exploitují, pomáhá, když umějí programovat, a naopak vědomosti o tom, jak se exploituje, pomáhají těm, kdo programují, dělají mnozí hackeři obojí. Zajímavé hacker-ské postupy najdete jak v technikách, které se používají proto, aby vznikl elegantní kód, tak i v technikách, jež se používají za účelem exploitace programů. Hacking opravdu znamená jen akt nalezení chytrého a překvapivého řešení dané úlohy.

Hacky, které najdete v exploitačních programech (říká se jim exploits), obvykle využívají počítačová pravidla (postupy, příkazy atd.) nevšedním způsobem, aby se obešla bezpečnostní opatření tak, jak to zatím nikoho ani nenapadlo. Programovací hacky jsou podobné v tom smyslu, že také využívají počítačová pravidla nově a vynalézavě, ale jejich konečným cílem je efektivnější nebo kratší zdrojový kód, nemusí se nutně jednat o narušení bezpečnosti. Programů, které řeší zadanou úlohu, se dá napsat nekonečně mnoho, ale většina z těchto řešení je zbytečně velkých, příliš složitých a ledabylých. Těch pár řešení, co zbudou, jsou kompaktní, efektivní a úhledná. Programy, které mají tyto kvality, jsou elegantní. Chytrým a vynalézavým řešením, která vedou k této efektivitě, se říká hacky. Hackeři obou skupin oceňují krásu elegantního kódu i důvtipnost chytrých hacků.

Ve světě byznysu se víc důrazu klade na to, aby se co nejdříve vychrlil kód, který funguje, než aby obsahoval chytré algoritmy a byl elegantní. Vzhledem k nesmírnému exponenciálnímu růstu výpočetní síly a paměti nemá, z hlediska byznysu, u moderních počítačů se zpracovatelskými cykly v gigahertzech a pamětí v gigabajtech valný smysl strávit pět hodin navíc jen proto, aby se vytvořil úsek kódu, který bude o něco málo rychlejší a efektivnější (co do využití paměti). I když optimalizaci doby trvání a využití paměti ponechá bez povšimnutí valná většina uživatelů kromě těch nejnáročnějších, je to charakteristika, kterou by šlo zpeněžit. Pokud jsou ale na prvním místě okamžité finanční výnosy, trávit čas tvorbou rafinovaných postupů se prostě nevyplatí.

Opravdové zhodnocení programovací elegance je tedy na hackerech: na počítačových nadšencích, jejichž konečným cílem není zisk, ale vymačkat až do mrtě ze svého starého Commodore 64 každý bit funkcionality; na autorech programů zvaných exploits, což jsou drobnoučké a okouzlující

fragmenty kódu, jimiž se dá proklouznout úzkými skulinami v zabezpečení, a na všech ostatních, kdo dokáží ocenit honbu za dokonalostí, kdo se stavějí čelem výzvě najít nejlepší možné řešení. To jsou lidé, které programování vzrušuje. Ti dokáží patřičně ohodnotit krásu elegantního fragmentu kódu nebo důvtipnost důmyslného postupu. Abyste ale pochopili, jak se dají programy explotovat, musíte nejprve dobře rozumět programování – toto je naprosto nezbytná podmínka. Proto je přirozené, že prvním předmětem našeho výkladu bude programování.

0x210 Co je programování?

Programování je velmi přirozený a intuitivní pojem. Program není nic víc než posloupnost příkazů napsaných v nějakém konkrétním jazyku. Programy jsou všude okolo nás – dokonce i ti, kdo mají panickou hrůzu z technologií, používají programy každý den. Itineráře pro jízdu autem, kuchařské recepty, fotbalové zápasy, DNA, to všechno jsou různé typy programů. Typický itinerář pro jízdu autem může vypadat například takto:

Jedte východním směrem po Main Street. Pokračujte po Main Street, dokud neuvidíte napravo kostel. Jestliže je ulice zavřená kvůli opravám, zde zabočte doprava na 15th Street, zabočte doleva na Pine Street, a pak doprava na 16th Street. Jinak prostě jedte dál a zabočte doprava na 16th Street. Pokračujte po 16th Street a zabočte doleva na Destination Road. Jedte rovně 5 mil po Destination Road, pak uvidíte napravo dům. Adresa je 743 Destination Road.

Kdo umí česky, tyto pokyny pochopí, a protože jsou napsány v češtině, bude se jimi při jízdě řídit. Jistě – není to žádná krásná literatura, ale každý z pokynů je jasný a snadno srozumitelný, přinejmenším pro ty, kdo umějí alespoň trochu česky.

Počítače ale žádnému lidskému jazyku nerozumějí, ani angličtině, natož češtině; rozumějí jen strojovému kódu. Chcete-li dát počítači pokyny, aby něco udělal, musejí být tyto pokyny, neboli instrukce, napsány v jeho jazyku. Strojový kód je ovšem tajemný a pracuje se s ním obtížně – skládá se z nijak nezpracovaných bitů a bajtů, a na každé architektuře je jiný. Pokud byste chtěli napsat program ve strojovém kódu pro procesor Intel x86, museli byste si zjistit hodnoty sdružené s jednotlivými instrukcemi, vypátrat jak spolu instrukce komunikují, a zvládnout myriády dalších nízkoúrovňových podrobností. Programování tohoto druhu je pracné a rozhodně neintuitivní.

Abychom překonali komplikace spojené s psaním ve strojovém kódu, potřebujeme překladač. Jednou z forem překladače strojového jazyka je assembler – to je program, který překládá kód assembleru do kódu, jenž bude stroj schopen přečíst. Assembler je méně záhadný než strojový kód, protože používá pro instrukce slovní názvy a proměnné, nikoliv pouhá čísla. Přesto je i assembler na hony vzdálen intuitivnímu jazyku. Názvy instrukcí jsou srozumitelné jen zasvěceným, přičemž jazyk je pro každou architekturu jiný. Právě tak, jako je strojový kód pro procesor Intel x86 jiný než strojový kód pro procesory Sparc, je i assembler určený pro x86 odlišný od assembleru pro Sparc. Program napsaný v assembleru pro architekturu jednoho procesoru nebude fungovat na

architektuře jiného procesoru. Je-li program napsaný v assembleru pro x86, musí se přepsat, má-li běžet na architektuře Sparc. A navíc – abyste mohli psát efektivní programy v assembleru, musíte rovněž znát řadu nízkoúrovňových detailů o architektuře procesoru, pro který píšete.

Uvedené potíže se dají zmírnit další formou překladače, kterému se říká kompilátor. Kompilátor převádí jazyk vysoké úrovně do strojového kódu. Vysokoúrovňové jazyky jsou mnohem intuitivnější než assembler (jsou obvykle založené na slovech a frázích z angličtiny) a dají se konvertovat do mnoha různých typů strojového kódu, tj. pro různé procesorové architektury. To znamená, že je-li program napsaný v nějakém vysokoúrovňovém jazyku, stačí ho napsat jednou; stejný programový kód se dá zkompilovat do strojového kódu různých konkrétních архитектур. Mezi vysokoúrovňové jazyky patří například C, C++ či Fortran. Ačkoliv program napsaný ve vysokoúrovňovém jazyku se mnohem snadněji čte a více se podobá angličtině než assembler nebo strojový kód, přesto se musejí dodržovat velmi striktní pravidla pro zápis jednotlivých instrukcí. V opačném případě kompilátor nebude schopen pochopit takový kód.

0x220 Pseudokód

Programátoři využívají ještě jednu formu programovacího jazyka, které se říká pseudokód. Pseudokód je jednoduchá řeč, která je uspořádána do obecné struktury podobající se nějakému vysokoúrovňovému jazyku. (Teoreticky se dá pseudokód psát v jakékoliv mluvené řeči, nicméně angličtina je nejvýhodnější, protože v ní se pseudokód nejvíc podobá skutečnému kódu, který na jeho základě vznikne.) Ačkoliv kompilátory, assembly a počítače pseudokódu nerozumějí, je to velmi šikovný způsob zápisu, jímž si programátor může uspořádat své instrukce. Pseudokód není konkrétně definovaný – každý programátor píše pseudokód trochu jinak. Je to jistý mlhavý druh chybějícího propojení mezi mluvenou řečí (obvykle angličtinou) a vysokoúrovňovými jazyky, jako je C. Pseudokód ovšem může posloužit jako skvělý úvodní nástroj pro vysvětlení běžných univerzálních programovacích pojmů.

0x230 Řídící struktury

Bez řídicích struktur by byl program jen pouhou sérií instrukcí, které se vykonávají sekvenčně (jedna za druhou). U velmi jednoduchých programů to stačí, ale většina programů, včetně našeho itineráře s pokyny k jízdě, tak jednoduchá není. Mezi pokyny k jízdě najdeme příkazy jako *Jeďte po Main Street, dokud neuvidíte napravo kostel* a *Jestliže je ulice zavřená kvůli opravám. . .* Tyto příkazy patří mezi řídicí struktury, protože mění tok vykonávání programu z jednoduchého sekvenčního zpracování (jedna instrukce za druhou) na složitější a užitečnější tok.

0x231 *If-Then-Else*

V našich pokynech k jízdě je zmíněno, že Main Street může být zavřená, protože se opravuje. Pokud tomu tak je, měla by se s takovou situací vypořádat speciální sada instrukcí, jinak se bude

v programu postupovat dál podle původní sady instrukcí. S těmito speciálními případy se dá v programu vypořádat pomocí jedné z nejpřirozenějších řídicích struktur: strukturou if-then-else (jestliže-pak-jinak). Všeobecně to vypadá nějak takhle:

```
If (podmínka) then
{
    Sada instrukcí, která se má vykonat, když podmínka platí;
}
Else
{
    Sada instrukcí, která se má vykonat, když podmínka neplatí;
}
```

V této knize budeme občas používat pseudokód podobající se jazyku C, takže každá instrukce bude končit středníkem, sady instrukcí budou obklopeny složenými závorkami a budou odsazené. Struktura if-then-else předchozích pokynů k jízdě může v pseudokódu vypadat takto:

```
Jedte po Main Street;
If (ulice je zavřená kvůli opravám)
{
    Zabočte doprava na 15th Street;
    Zabočte doleva na Pine Street;
    Zabočte doprava na 16th Street;
}
Else
{
    Zabočte doprava na 16th Street;
}
```

Povšimněte si, že každá instrukce je na samostatném řádku, a že jednotlivé sady podmínkových instrukcí jsou obklopeny složenými závorkami (a že jsou rovněž odsazené, aby se text lépe četl). V C a v mnohých dalších programovacích jazycích se klíčové slovo then předpokládá implicitně, takže se nemusí uvádět – v předchozím pseudokódu jsme ho vynechali.

V jiných jazycích může být samozřejmě stanoveno, že klíčové slovo then je v syntaxi povinné – sem patří nejenom BASIC a Fortran, ale také Pascal. Tyto druhy syntaktických odlišností v jednotlivých programovacích jazycích jsou pouze drobné detaily; podkladová struktura je stále stejná. Jakmile programátor pochopí pojmy, které se jazyky snaží vyjádřit, zvládne hravě různé variace syntaxe. Protože v následujících sekcích budeme pracovat s jazykem C, budeme v této knize používat pseudokód, jehož syntaxe se bude podobat syntaxi jazyka C. Uvědomte si ovšem, že pseudokód může mít různé, někdy i dost odlišné formy.

Další běžné pravidlo syntaxe à la C říká, že pokud sada instrukcí uzavřených ve složených závorkách obsahuje pouze jedinou instrukci, nejsou složené závorky povinné. Kvůli lepší čitelnosti a jasnosti úmyslů je ovšem vhodné takové instrukce odsazovat (z hlediska syntaxe to povinné není).

Výše uvedené pokyny k jízdě lze s využitím tohoto pravidla přepsat následovně (stále ovšem máme ekvivalentní fragment pseudokódu):

```
Jeďte po Main Street;  
If (ulice je zavřená kvůli opravám)  
{  
    Zabočte doprava na 15th Street;  
    Zabočte doleva na Pine Street;  
    Zabočte doprava na 16th Street;  
}  
Else  
    Zabočte doprava na 16th Street;
```

Toto pravidlo o sadách instrukcí platí pro všechny řídicí struktury zmiňované v této knize. Toto pravidlo samotné je rovněž možné popsat prostřednictvím pseudokódu:

```
If (v sadě instrukcí je jen jediná instrukce)  
    Složené závorky vymezující skupinu instrukcí nejsou povinné;  
Else  
{  
    Složené závorky vymezující skupinu instrukcí jsou povinné;  
    Musíme mít k dispozici nějaký způsob, jak tyto instrukce logicky seskupit;  
}
```

I samotný popis syntaxe se dá považovat za jednoduchý program. Existují různé varianty if-then-else, jako je například příkaz select/case. Logika je v zásadě pořád stejná – jestliže podmínka platí, udělej tyto věci, jinak udělej tamty jiné věci (mezi ně mohou patřit i další příkazy if-then).

0x232 *Cykly While/Until*

Dalším elementárním programovacím pojmem je řídicí struktura `while`, což je druh cyklu. Programátor si často přeje vykonat jistou sadu instrukcí víckrát za sebou, ne pouze jednou. Program může tento úkol splnit prostřednictvím cyklu, požaduje k tomu ovšem sadu podmínek, které mu řeknou, kdy má s cyklováním přestat, jinak by cykloval až do skonání věků. Cyklus `while` říká, aby se následná sada instrukcí cyklu vykonávala tak dlouho, dokud (`while`) platí podmínka cyklu. Jednoduchý program pro hladovou myš by mohl vypadat takto:

```
While (máš hlad)  
{  
    Najdi něco k snědku;  
    Sněz to;  
}
```

Sada dvou instrukcí za příkazem `while` se bude opakovat tak dlouho, dokud (`while`) bude mít myš hlad. Množství jídla, které myška v jednotlivých průchodech cyklem najde, může být různé, od nepatrného drobečku až k celému bochníku chleba. A obdobně – počet průchodů sadou instrukcí v cyklu `while` může být závislý na tom, kolik jídla myš vlastně najde.

Jistou variantou cyklu `while` je cyklus `until`, což je syntaxe, která je k dispozici například v programovacím jazyku Perl (v C se tato syntaxe nepoužívá). Cyklus `until` je prostě cyklus `while` s invertovanou podmínkou cyklu. Tentýž program krmení myši vypadá s cyklem `until` takto:

```
Until (nemáš hlad)
{
    Najdi něco k snědku;
    Sněz to;
}
```

Je jasné, že každý příkaz `until` se dá převést na cyklus `while`. Výše uvedené pokyny k jízdě obsahují příkaz `Jedte po Main Street`, dokud neuvidíte napravo kostel. Toto se dá snadno změnit na standardní cyklus `while`, když prostě podmínku obrátíme na opačnou.

```
While (napravo není kostel)
Jedte po Main Street;
```

0x233 *Cykly For*

Další řídicí konstrukcí cyklu je cyklus `for`. Obvykle se používá tehdy, když chce programátor provést konkrétní počet iterací. Pokyn k jízdě `Jedte přímo 5 mil po Destination Road` se dá převést na cyklus `for` v tomto tvaru:

```
For (5 iterací)
Jedte přímo 1 míli;
```

Ve skutečnosti je cyklus `for` vlastně cyklem `while` s čítačem (počítadlem průchodů). Stejný příkaz lze totiž zapsat i takto:

```
Nastavit čítač na 0;
While (hodnota čítače je menší než 5)
{
    Jedte přímo 1 míli;
    Přičíst 1 k čítači;
}
```

V syntaxi pseudokódu podobném jazyku C je podstata cyklu `for` ještě zřejmější:

```
For (i=0; i<5; i++)
    Jedte přímo 1 míli;
```

V tomto případě se čítač jmenuje `i` a příkaz `for` je rozdělen do tří sekcí oddělených středníky. První sekce deklaruje čítač a nastavuje ho na počáteční hodnotu, v tomto případě na 0 (nulu). Druhá sekce je něco jako příkaz `while` využívající čítač: dokud (`while`) čítač splňuje uvedenou podmínku, pokračovat v cyklování. Třetí, poslední, sekce popisuje, jaká akce se má podniknout s čítačem při každé iteraci. V tomto případě je touto akcí `i++`, což je zkrácený zápis výroku "Přičti 1 k čítači `i`".

S využitím všech dosud probraných řídicích struktur se pokyny k jízdě dají převést do pseudokódu ve stylu podobném jazyku C takto:

```
Jeďte východním směrem po Main Street;
While (napravo není kostel)
Jeďte po Main Street;
If (ulice je zavřená)
{
    Zabočte doprava na 15th Street;
    Zabočte doleva na Pine Street;
    Zabočte doprava na 16th Street;
}
Else
    Zabočte doprava na 16th Street;
For (i=0; i<5; i++)
    Jeďte přímo 1 míli;
Zastavte na 743 Destination Road;
```

0x240 Další základní programovací pojmy

V následujících sekcích probereme univerzálnější programovací pojmy, které se používají v mnohých programovacích jazycích (i když s jistými drobnými syntaktickými odlišnostmi). Poté, co tyto pojmy uvedu, je budu prostřednictvím syntaxe podobné jazyku C integrovat do příkladu pseudokódu. Na konci se bude tento pseudokód velmi podobat kódu C.

0x241 *Proměnné*

Čítač, který jsme použili v cyklu `for`, je ve skutečnosti jeden z typů proměnných. Proměnná se dá jednoduše chápat jako objekt obsahující data, která lze měnit – odtud ten název. Existují i proměnné, které se nemění, a trefně se jim říká konstanty. Vrátime-li se k našemu příkladu s pokyny k jízdě, rychlost auta lze označit za proměnnou, zatímco barva auta je konstanta. Ačkoliv v pseudokódu jsou proměnné jednoduché abstraktní pojmy, v jazyce C (a v mnoha dalších) se proměnné musejí deklarovat a musí se jim přidělit typ. Teprve pak se mohou použít. Je tomu tak proto, že program C se bude nakonec kompilovat do spustitelného programu. Podobně jako u kuchařského receptu, v němž se seznam potřebných ingrediencí uvádí před pokyny, jak kuchtit, deklarace proměnné umožňují vykonat potřebné přípravné akce, než se pustíme do hlavní části programu. Nakonec se

všechny proměnné uloží někde v paměti, přičemž jejich deklarace umožňují kompilátoru, aby si tuto paměť zorganizoval efektivněji. Ovšem na úplném konci, navzdory všem těm deklaracím typů proměnných, je všechno pouze paměť.

V C dostane každá proměnná typ, který popisuje, jaké informace hodláte do této proměnné ukládat. Mezi nejběžnější typy patří `int` (celočíslné hodnoty), `float` (hodnoty s pohyblivou desetinnou čárkou) a `char` (hodnota obsahující jediný znak). Proměnné se deklarují velmi jednoduše – dané klíčové slovo se uvede před seznamem proměnných, jako to vidíte zde:

```
int a, b;
float k;
char z;
```

Proměnné `a` a `b` jsou nyní definované jako celočíselné. Do `k` se dá uložit hodnota s desetinným místem (například hodnota 3.14). O proměnné `z` se předpokládá, že v ní bude uložený nějaký znak, jako například `A` nebo `w`. Hodnoty se dají proměnným přiřadit už při deklaraci, nebo kdykoliv později, prostřednictvím operátoru `=`.

```
int a = 13, b;
float k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

Až se výše uvedené instrukce vykonají, bude proměnná `a` obsahovat hodnotu 13, `k` bude obsahovat číslo 3.14, `z` bude obsahovat znak `w` a `b` bude obsahovat hodnotu 18, protože 13 plus 5 se rovná 18. Stručně řečeno – proměnné jsou prostě jeden ze způsobů, jak se dají zapamatovat hodnoty; v jazyce C však musíte nejprve deklarovat typ každé proměnné.

0x242 Aritmetické operátory

Příkaz `b = a + 7` je ukázka velmi jednoduchého aritmetického operátoru. V C se pro jednotlivé aritmetické operace používají symboly uvedené v následující tabulce.

První čtyři operace v tabulce by měly být zřejmé. Ačkoliv modulo vám možná připadá jako nový pojem, jedná se pouze o zbytek po dělení. Má-li `a` hodnotu 13, pak 13 děleno 5 se rovná 2, zbytek je 3, což znamená, že `a % 5 = 3`. Dále, protože proměnné `a` i `b` jsou celá čísla (`int`), příkaz `b = a / 5` vede na výsledek 2, který se uloží do `b`, protože je to celá část výsledku operace dělení. Chcete-li získat přesnější výsledek 2,6, musíte použít proměnné s pohyblivou čárkou (`float`).

Operace	Symbol	Ukázka
Sčítání	+	<code>b = a + 5</code>

Operace	Symbol	Ukázka
Odčítání	-	$b = a - 5$
Násobení	*	$b = a * 5$
Dělení	/	$b = a / 5$
Modulo	%	$b = a \% 5$

Pokud chcete dostat tyto koncepty do programu, musíte mluvit jeho jazykem. Jazyk C dále poskytuje pro aritmetické operace několik zkrácených zápisů. Jeden z nich jsem už zmínil výše a běžně se používá v cyklech `for`.

Výraz	Zkrácený zápis	Vysvětlení
$i = i + 1$	<code>i++</code> nebo <code>++i</code>	Přičte 1 k proměnné.
$i = i - 1$	<code>i--</code> nebo <code>--i</code>	Odečte 1 od proměnné.

Tyto zkrácené zápisy se dají zkombinovat s ostatními aritmetickými operacemi, takže mohou vzniknout složitější výrazy. Zde pak lépe vynikne rozdíl mezi `i++` a `++i`. První výraz znamená zvýšit hodnotu `i` o 1 poté, co se vyhodnotí aritmetická operace, zatímco druhý znamená zvýšit hodnotu `i` o 1 předtím, než se vyhodnotí aritmetická operace. Podívejte se na tento příklad:

```
int a, b;
a = 5;
b = a++ * 6;
```

Na konci této sady instrukcí bude `b` obsahovat 30 a `a` bude obsahovat 6, protože zkrácený zápis `b = a++ * 6;` je ekvivalentní těmto příkazům:

```
b = a * 6;
a = a + 1;
```

Pokud bychom ale použili instrukci `b = ++a * 6;`, pořadí operací sčítání se změní, což znamená, že tentokrát budou ekvivalentní tyto instrukce:

```
a = a + 1;
b = a * 6;
```

Protože se změnilo pořadí operací, `b` bude nyní obsahovat hodnotu 36; `a` bude stále obsahovat 6.

V programech dost často potřebujeme modifikovat obsah proměnné na místě. Například potřebujete k proměnné přičíst nějakou hodnotu, řekněme 12, a výsledek ihned uložit zpět do této proměnné (například `i = i + 12`). To se děje tak často, že i pro tyto výrazy existují zkrácené zápisy, viz následující tabulka.

Výraz	Zkrácený zápis	Vysvětlení
$i = i + 12$	$i += 12$	Přičte danou hodnotu k proměnné.
$i = i - 12$	$i -= 12$	Odečte danou hodnotu od proměnné.
$i = i * 12$	$i *= 12$	Vynásobí proměnnou danou hodnotou.
$i = i / 12$	$i /= 12$	Vydělí proměnnou danou hodnotou.

0x243 Porovnávací operátory

Proměnné se hojně využívají v podmínkových příkazech řídicích struktur, které jsme si vysvětlili dříve. Podmínkové příkazy jsou založeny na jistém druhu porovnávání. V jazyce C se porovnávací operátory zapisují prostřednictvím zkrácené syntaxe, která se velmi běžně používá v různých programovacích jazycích.

Podmínka	Symbol	Ukázka
Menší než	<	$(a < b)$
Větší než	>	$(a > b)$
Menší nebo rovno	<=	$(a <= b)$
Větší nebo rovno	>=	$(a >= b)$
Je rovno	==	$(a == b)$
Není rovno	!=	$(a != b)$

Většina operací je zřejmá, takže k nim není potřeba nic vysvětlovat, jen upozorňuji, abyste si dobře zapamatovali, jakým způsobem se zapisuje operace je rovno – jsou dvě rovnítka za sebou. Tato dvě rovnítka znamenají test na ekvivalenci, zatímco jediné rovnítko se používá pro přiřazení hodnoty do proměnné. Příkaz $a = 7$ znamená uložit hodnotu 7 do proměnné a , zatímco $a == 7$ znamená zkontrolovat, zdali má proměnná a v sobě uloženou hodnotu 7. (Některé programovací jazyky, jako například Pascal, používají pro přiřazení do proměnné operátor $:=$, aby vizuálně eliminovaly možnost záměny těchto operací.) Dále si všimněte, že vykřičník všeobecně znamená ne. Tento symbol se dá použít i samostatně – v takovém případě pak mění výraz na opačný.

$!(a < b)$ je ekvivalentní k $(a >= b)$

Porovnávací operátory se dají řetězit pomocí zkrácených zápisů logických operací OR a AND.

Logika	Symbol	Ukázka
OR (NEBO)		$((a < b) (a < c))$
AND (A)	&&	$((a < b) \&\& !(a < c))$

Ukázkový příkaz skládající se ze dvou dílčích podmínek spojených logikou OR (NEBO) se vyhodnotí jako pravdivý (true), jestliže je a menší než b, NEBO jestliže je a menší než c. Obdobným způsobem se ukázkový příkaz skládající se ze dvou dílčích podmínek spojených logikou AND (A) vyhodnotí jako pravdivý, jestliže je a menší než b A ZÁROVEŇ a není menší než c. Dílčí podmínky lze pro lepší přehlednost dávat do kulatých závorek a může jich být více než dvě.

Pomocí proměnných, porovnávacích operátorů a řídicích struktur lze vyjádřit mnoho věcí. Vraťme se k příkladu myši, která shání něco k snědku. Mít nebo nemít hlad se dá přeložit do logické proměnné s hodnotou true nebo false. Hodnota 1 pochopitelně znamená true, 0 znamená false.

```
While (hlad == 1)
{
    Najdi něco k snědku;
    Sněz to;
}
```

A nyní se seznámíme s další zkratkou, kterou programátoři a hackeři používají dost často. C vlastně nemá logické operátory, takže jakákoli nenulová hodnota se považuje za true, a příkaz se považuje za false, obsahuje-li 0. A skutečně – porovnávací operátory opravdu vracejí hodnotu 1, jestliže se porovnávání vyhodnotí na true, a hodnotu 0, jestliže se vyhodnotí na false. Při testu, zda je proměnná hlad rovna 1, se vrátí 1, jestliže se hlad rovná 1, a 0, jestliže se hlad rovná 0. Protože se jedná pouze o dva případy, je možné porovnávací operátor vyhodit.

```
While (hlad)
{
    Najdi něco k snědku;
    Sněz to;
}
```

Následující program chytřejší myši s více vstupy předvádí, jak se dají porovnávací operátory zkombinovat s proměnnými.

```
While ((hlad) && !(je_tu_kočka))
{
    Najdi něco k snědku;
    If(!(jídlo_v_pastičce))
        Sněz to;
}
```

V tomto příkladu se předpokládá, že existují proměnné, které popisují přítomnost kočky a kde se nachází jídlo, přičemž hodnota 1 znamená true a 0 znamená false. Zapamatujte si ovšem, že libovolná nenulová hodnota se považuje za true, a že pouze hodnota 0 se považuje za false.

0x244 *Funkce*

Tu a tam se vyskytne sada instrukcí, o které programátor předem ví, že ji bude několikrát potřebovat. Takové instrukce se dají seskupit do menšího podprogramu zvaného funkce. V některých jiných programovacích jazycích se jim říká subrutiny nebo procedury. Například akce pro změnu směru jízdy auta se skládá z mnoha dílčích akcí: zapnout patřičný blinkr, zpomalit, zkontrolovat okolní provoz, otočit volantem v patřičném směru atd. To znamená, že pokyny k jízdě, které byly uvedeny na začátku této kapitoly, bychom měli propracovat do daleko větších podrobností. Kdybychom ovšem měli explicitně vypisovat všechny dílčí instrukce pro každý pokyn, bylo by to velmi pracné, nehledě na to, že samotný plán jízdy by se velmi špatně četl. Potřebné proměnné se ovšem dají předat jako argumenty do funkce, aby se dalo modifikovat, jak má funkce fungovat. V našem případě předáváme do funkce proměnnou, která udává směr (doleva, doprava):

```
Function Zatočit(proměnná_směr)
{
    Zapnout blinkr proměnná_směr;
    Zpomalit;
    Zkontrolovat okolní provoz;
    while(něco jede)
    {
        Stát;
        Sledovat okolní provoz;
    }
    Otočit volantem proměnná_směr;
    while(otočení není dokončené)
    {
        if(rychlost < 5 mph)
            Zrychlit;
    }
    Otočit volantem zpět do původní pozice;
    Vypnout blinkr proměnná_směr;
}
```

Tato funkce popisuje všechny instrukce, které potřebujete, pokud chcete změnit směr jízdy. Když program, který potřebuje zatočit, ví o této funkci, stačí mu funkci zavolat. Když se funkce zavolá, vykonají se instrukce, které jsou uvnitř, s hodnotami argumentů předaných do funkce; poté se vykonávání vrátí do programu za volání funkce. Do funkce se dá předat hodnota vyjadřující buď doleva, nebo doprava, což způsobí, že funkce zatočí autem v předaném směru.

V jazyce C mohou funkce standardně vracet hodnotu volajícímu. Pro ty z vás, kdo důvěrně znáte matematické funkce, to je nepochybně zcela jasné. Představte si funkci, která vypočítává faktoriál zadaného čísla – je přirozené, že vrátí výsledek.

V C nejsou funkce označovány klíčovým slovem "function"; deklarují se datovým typem proměnné, kterou vracejí. Je to formát, který se velmi podobá deklaraci proměnné. Jestliže má funkce vrátit celé číslo (předpokládejme, že se jedná o funkci, která má vypočítat faktoriál čísla x), může taková funkce vypadat takto:

```
int factorial(int x)
{
    int i;
    for(i=1; i < x; i++)
        x *= i;
    return x;
}
```

Funkce je deklarována jako celočíselná, protože vynásobí všechna celá čísla od 1 do x a vrátí výsledek, což je také celé číslo. Příkaz `return`, který vidíte na konci funkce, předává zpět obsah proměnné x a ukončuje funkci. Tato funkce pro výpočet faktoriálu se pak dá používat v hlavní části programu, který o ní ví, jako celočíselná proměnná.

```
int a=5, b;
b = factorial(a);
```

Na konci tohoto kratičkého programu bude proměnná b obsahovat hodnotu 120, protože funkce `factorial` byla zavolána s argumentem 5 a vrátila 120.

V C je to zařízeno tak, že kompilátor musí "vědět" o funkcích dříve, než je může použít. To se dá jednoduše zařídit tak, že se celá funkce napíše předtím, než se v programu později zavolá, nebo můžete použít prototyp funkce (function prototype). Prototyp funkce je prostě způsob, jak říct kompilátoru, že má očekávat funkci s daným názvem, s daným datovým typem návratové hodnoty a s danými datovými typy argumentů funkce. Skutečná funkce může pak být umístěna ke konci programu, dá se ale volat kdekoliv, protože kompilátor už ví o její existenci. Prototyp funkce `factorial()` může vypadat následovně:

```
int factorial(int);
```

Prototypy funkce se obvykle umísťují poblíž začátku programu. Nemusejí nutně definovat nějaké názvy proměnných, protože tohle se udělá až ve skutečné funkci. Kompilátor potřebuje znát pouze název funkce, datový typ návratové hodnoty a datové typy argumentů funkce.

Pokud nemá funkce nic vracet, měla by být deklarována jako `void`, což je případ funkce `Zatočit()`, kterou jsem uvedl o něco dříve v této kapitole. Funkce `Zatočit()` ovšem stále ještě nepokrývá veškerou funkcionalitu, kterou potřebujeme pro pokyny zajišťující změnu směru jízdy. Při každém zatočení potřebujeme znát směr a název ulice. To znamená, že funkce pro zatáčení by měla mít dvě proměnné: směr zatočení, a ulici, do které chceme vjet. To je pro naši funkci jistá komplikace, protože ulice, kam se má zatočit, se musí najít ještě předtím, než začneme zatáčet. Následující výpis obsahuje pseudokód složitější zatáčecí funkce (samozřejmě v syntaxi podobné jazyku C).

```

void Zatočit(proměnná_směr, název_cílové_ulice)
{
    Vyhledat ceduli s názvem ulice;
    název_aktuální_křižovatky = přečíst ceduli s názvem ulice;
    while(název_aktuální_křižovatky != název_cílové_ulice)
    {
        Vyhledat ceduli jiné ulice;
        název_aktuální_křižovatky = přečíst ceduli s názvem ulice;
    }
    Zapnout blinkr proměnná_směr;
    Zpomalit;
    Zkontrolovat okolní provoz;
    while(něco jede)
    {
        Zastavit;
        Sledovat okolní provoz;
    }
    Otočit volantem proměnná_směr;
    while(otočení není dokončené)
    {
        if(rychlost < 5 mph)
            Zrychlit;
    }
    Otočit volantem zpět do původní pozice;
    Vypnout blinkr proměnná_směr;
}

```

Tato funkce obsahuje sekci, ve které se vyhledá patřičná křižovatka tím, že se vyhledá cedule s názvem ulice a přečte se z ní název ulice, jenž se uloží do proměnné `název_aktuální_křižovatky`. Poté se pokračuje s vyhledáváním a čtením ulic tak dlouho, dokud se nenajde cílová ulice. V tomto okamžiku se vykonají instrukce týkající se zabočení. Pokyny k jízdě v pseudokódu mohou být nyní upraveny tak, aby se v nich použila výše uvedená zatáčecí funkce.

```

Jedte východním směrem po Main Street;
    While (napravo není kostel)
Jedte po Main Street;
If (ulice je zavřená)
{
    Zatočit(vpravo, 15th Street);
    Zatočit(vlevo, Pine Street);
    Zatočit(vpravo, 16th Street);
}
else

```