12

Správa paměti a ukazatele

V této kapitole se budeme zabývat různými hledisky správy paměti a přístupu do paměti. Většiny starostí se správou paměti zbavuje programátory běhový systém ; přesto je však užitečné rozumět fungování správy paměti a důležité jsou i znalosti efektivní práce s neřízenými prostředky.

Dobré porozumění správě paměti a možnostem ukazatelů v jazyku C# vám umožní lépe integrovat kód v C# se starším kódem a pomůže vám zvládnout účinnou manipulaci s pamětí v systémech kritických z hlediska výkonu.

Hlavní témata této kapitoly:

- Jak běhový systém přiděluje prostor v zásobníku a v haldě
- Jak funguje úklid
- Jak lze pomocí destruktorů a rozhraní `System.IDisposable` zajistit správné uvolnění neřízených prostředků
- Syntaxe použití ukazatelů v jazyce C#
- Jak lze pomocí ukazatelů implementovat vysoce výkonná pole uložená v zásobníku

Technické principy správy paměti

Mezi výhody programování v C# patří, že se programátoři nemusí podrobně starat o správu paměti. Konkrétně problematiku čištění paměti automaticky zajišťuje automatická správa paměti. Díky tomu získáváte účinnost blížící se jazykům typu C++, aniž byste se problematikou správy paměti museli zabývat sami jako v C++. I když však paměť nemusíte spravovat ručně, je nadále vhodné rozumět dění v pozadí, abyste mohli psát efektivní kód. V této části analyzujeme procesy, ke kterým dochází v paměti počítače při přidělování paměti proměnným.

> Většina informací uvedených v této části není přesně dokumentována. Následující informace byste měli považovat za zjednodušený úvod do obecných algoritmů pro správu paměti, nikoli popis skutečné implementace.

Hodnotové datové typy

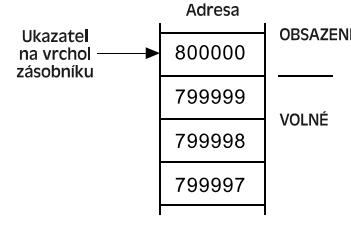
Windows používá systém označovaný jako *virtuální adresování*, ve kterém operační systém zcela spravuje přiřazování paměťových adres, které jsou viditelné programům, skutečným místům v hardwarové paměti. V důsledku toho „vidí“ každý proces v 32bitovém procesoru 4 GB dostupné paměti bez ohledu na to, kolik hardwarové paměti je v počítači skutečně nainstalováno (u 64bitových procesorů bude tato hodnota větší). V těchto 4 GB paměti jsou umístěny všechny části programu, včetně spustitelného kódu, případných knihoven DLL načtených kódem a obsahu všech proměnných, které program při své činnosti používá. Tato paměť velikosti 4 GB se nazývá *virtuální adresní prostor* nebo *virtuální paměť*. Pro zjednodušení ji budeme v této kapitole zkráceně označovat jako *paměť*.

Paměťová místa v dostupných 4 GB jsou číslována od nuly. Chcete-li získat přístup k hodnotě uložené na konkrétním místě v paměti, musíte uvést číslo, které dané paměťové místo reprezentuje. Ve všech překládaných vysokoúrovňových jazycích včetně C#, Visual Basicu, C++ a Javy převádí překladač názvy proměnných srozumitelných člověku na paměťové adresy, se kterými může pracovat procesor.

V určitém místě virtuální paměti procesu se nachází oblast označovaná jako *zásobník*. Do zásobníku se ukládají hodnotové datové typy, které nejsou členy objektů. Navíc při volání metody slouží zásobník k uložení kopie parametrů předaných této metodě. Chcete-li pochopit fungování zásobníku, musíte porozumět tomu, jaký význam má v jazyce C# obor proměnné. *Vždy* platí, že když se proměnná `a` dostane do oboru platnosti před proměnnou `b`, tak se proměnná `b` dostane mimo obor jako první. Podívejte se na tento kód:

```
{
  int a;
  // Nějaké operace
  {
    int b;
    // Nějaké jiné operace
  }
}
```

Nejdříve je deklarována proměnná `a`. Potom je ve vnitřním bloku kódu deklarována proměnná `b`. Dále je vnitřní blok kódu ukončen a proměnná `b` se dostává mimo obor platnosti. Po ní se mimo obor dostává proměnná `a`. Obor platnosti proměnné `b` je tedy zcela vnořen do oboru platnosti proměnné `a`. Pro fungování zásobníku je klíčová koncepce, podle které se paměť přidělená proměnným vždy navrací v opačném pořadí, než v jakém byla přidělena.



Obrázek 12.1

Přesné místo zásobníku v adresním prostoru není známo – při vývoji v C# tuto informaci nepotřebujete. Další volné místo v zásobníku určuje *ukazatel na vrchol zásobníku* („*ukazatel zásobníku*“, proměnná, kterou udržuje operační systém). Při prvním spuštění programu směřuje ukazatel na vrchol zásobníku těsně za konec bloku paměti, který je pro zásobník vyhrazen. Zásobník se plní směrem dolů od vyšších adres k nižším. Během vkládání dat do zásobníku se odpovídajícím způsobem mění i uka-

zatel na vrchol zásobníku, aby vždy směřoval těsně za následující volné místo. Tento princip je znázorněn na obrázku 7.1, kde vidíte ukazatel na vrchol zásobníku s hodnotou 800000 (0xC3500 hexadecimálně) a další volné místo na adrese 799999.

Následující kód požádá překladač o místo v paměti pro uložení proměnných typu int a double, jejichž paměťová místo budou označena jako nRacingCars a engineSize. Řádky s deklarací proměnných představují místo, od kterého budete požadovat přístup k příslušné proměnné. V místě ukončovací složené závorky bloku, ve kterém byly deklarovány, se obě proměnné dostávají mimo obor platnosti.

```
{
  int nRacingCars = 10;
  double engineSize = 3000.0;
  // Výpočty;
}
```

Předpokládejme, že používáte zásobník znázorněný na obrázku 12.1. Když se proměnná nRacing Cars dostane do oboru platnosti a získá hodnotu 10, je hodnota 10 uložena na adresy 799996 až 799999, což jsou čtyři bajty přímo pod místem, na které směřuje ukazatel zásobníku. (Čtyři bajty jsou nutné proto, že tuto velikost vyžaduje uložení typu int.) Přitom se od hodnoty ukazatele zásobníku odečte číslo 4, takže ukazatel nyní směřuje na místo s adresou 799996, ihned za novým prvním volným místem (799995).

Na dalším řádku kódu je deklarována proměnná engineSize (typu double) a inicializována hodnotou 3000.0. Typ double zabírá 8 bajtů. Hodnota 3000.0 bude proto v zásobníku uložena na adresách 799988 až 799995 a ukazatel na vrchol zásobníku se sníží o 8. Opět tedy směřuje přímo za následující volné místo v zásobníku.

Když se proměnná engineSize dostane mimo obor platnosti, není již nadále potřeba. Životnosti lokálních proměnných jsou vždy takto vnořeny. Díky tomu lze zaručit, že bez ohledu na operace provedené v době, kdy byla proměnná engineSize ve svém oboru platnosti, směřuje nyní ukazatel zásobníku na místo, kde je uložena proměnná engineSize. Aby se tato proměnná odstranila ze zásobníku, zvýší se hodnota ukazatele na vrchol zásobníku o 8. Nyní tedy směřuje na místo, které bezprostředně následuje za koncem oblasti vyhrazené proměnné engineSize. V tomto místě kódu se vyskytuje ukončovací složená závorka a mimo svůj obor platnosti se dostává i proměnná nRacingCars. Ukazatel zásobníku se zvýší o 4. Když se po odstranění proměnných engineSize a nRacingCars ze zásobníku dostane do oboru platnosti jiná proměnná, přepíše se paměť pod místem s adresou 799999, kde se dříve nacházela proměnná nRacingCars.

Jestliže překladač narazí na řádek typu int i, j, zdánlivě nelze určit pořadí, ve kterém se proměnné dostanou do svého oboru platnosti. Obě proměnné jsou deklarovány současně a zároveň se také dostanou mimo svůj obor. Na pořadí odstranění těchto dvou proměnných z paměti v této situaci nezáleží. Překladač interně vždy zajistí, aby proměnná umístěná do paměti jako první byla odstraněna jako poslední. Tím dodržuje pravidlo, že nesmí dojít k překřížení životnosti proměnných.

Referenční datové typy

Zásobník sice poskytuje vysoký výkon, nelze jej ale použít pro všechny proměnné. Požadavek na vnořování životnosti proměnných je v mnoha případech příliš omezující. Často je nutno přidělit

nějaké metodě paměť k uložení určitých dat, která mají být dostupná dlouho poté, co tato metoda skončí. Tato možnost je k dispozici pro všechny referenční typy; požadavky na úložné místo vznášíme pomocí operátoru `new`. V těchto případech se uplatňuje *řízená halda*.

Jestliže jste se při programování v C++ zabývali nízkoúrovňovou správou paměti, již jste s haldou obeznámeni. Řízená halda se poněkud liší od haldy používané v C++. Funguje pod kontrolou automatické správy paměti a oproti klasickým haldám přináší významné výhody.

Řízená halda (nebo krátce jen halda) není nic jiného než další oblast paměti v rámci 4 GB vyhrazených procesu. Následující kód ukazuje, jak halda funguje a jak je referenčním datovým typům přidělována paměť:

```
void DoWork()
{
  Customer arabel;
  arabel = new Customer();
  Customer otherCustomer2 = new EnhancedCustomer();
}
```

Uvedený kód předpokládá existenci dvou tříd: `Customer` a `EnhancedCustomer`, kde `EnhancedCustomer` je odvozena od `Customer`.

Nejdříve deklarujete odkaz na třídu `Customer` s názvem `arabel`. Prostor pro tento odkaz bude přidělen v zásobníku. Nezapomeňte však, že se jedná pouze o odkaz, nikoli o skutečný objekt typu `Customer`. Odkaz `arabel` zabírá 4 bajty, které postačují k uložení adresy, na které bude objekt typu `Customer` uložen. (K reprezentaci paměťové adresy mezi 0 a 4 GB jako celé číslo jsou nutné čtyři bajty.)

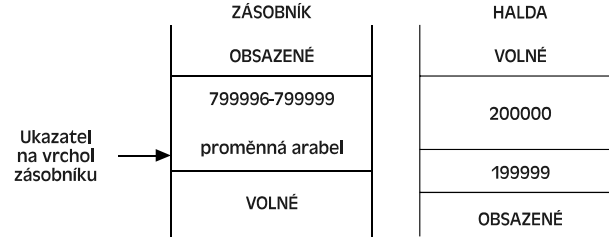
Další řádek,

```
arabel = new Customer();
```

plní několik funkcí. Zaprvé přidělí paměť v haldě, kde bude uložen objekt typu `Customer` (skutečný objekt, nikoli pouze adresa). Zadruhé uloží do proměnné `arabel` na adresu místa v paměti, která přidělil novému objektu typu `Customer`. (Také zavolá příslušný konstruktor `Customer()`, který inicializuje datové složky v instanci této třídy, ale tím se zde nebudeme zabývat.)

Instance třídy `Customer` není umístěna do zásobníku, ale do haldy. Z tohoto příkladu nelze přesně určit, kolik bajtů objekt typu `Customer` zabírá. Pro účely tohoto výkladu však předpokládejme, že má velikost 32 bajtů. Těchto 32 bajtů obsahuje instanční složky třídy `Customer` a také určité informace, pomocí nichž platforma .NET identifikuje a spravuje instance svých tříd.

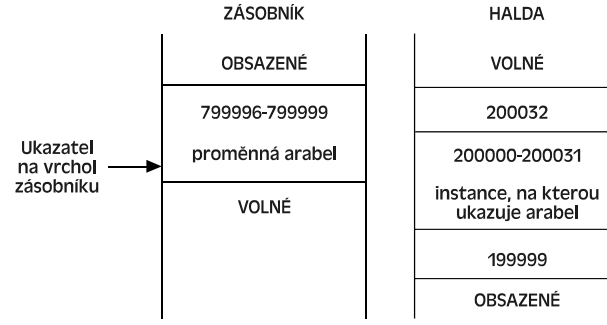
Aby mohl běhový systém.NET najít v haldě místo, kam bude uložen nový objekt typu `Customer`, prohledá haldu a použije první souvislý nepoužívaný blok velikosti 32 bajtů. Předpokládejme dále, že k tomu náhodou dojde na adrese `200000` a že se odkaz



Obrázek 12.2

`arabel` v zásobníku nachází v místech `799996` až `799999`. To znamená, že před vytvořením instance objektu `arabel` bude obsah paměti odpovídat schématu na obrázku 12.2.

Po přidělení paměti novému objektu typu `Customer` bude obsah paměti popsán obrázkem 12.3. Všimněte si, že na rozdíl od zásobníku se paměť v haldě přiděluje směrem nahoru, takže volné místo je k dispozici nad použitou oblastí.



Obrázek 12.3

Další řádek kódu deklaruje odkaz na objekt typu `Customer` a zároveň vytváří jeho instanci. V tomto případě je pomocí jediného řádku kódu přidělen prostor v zásobníku pro odkaz `otherCustomer2` a prostor v haldě pro object typu `EnhancedCustomer`:

```
Customer otherCustomer2 = new EnhancedCustomer();
```

Tento řádek vyhradí v zásobníku 4 bajty pro uložení odkazu `otherCustomer2`, který bude uložen na adresách `799992` až `799995`. Objektu typu `EnhancedCustomer` přidělí místo v haldě, které začíná adresou `200032`.

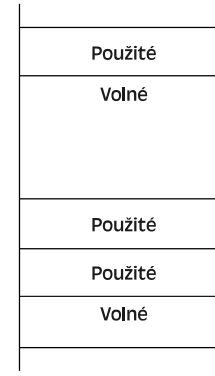
Z příkladu je zřejmé, že vytvoření referenční proměnné je složitější než vytvoření hodnotové proměnné. S tím souvisí i výkonnostní režie. Ve skutečnosti jsme zde uvedený proces poněkud zjednodušili, protože běhový systém .NET musí udržovat informace o stavu haldy a tyto informace je nutné aktualizovat při každém přidání nových dat do haldy. Bez ohledu na uvedenou režii máte nyní k dispozici mechanismus přidělování paměti proměnným, který nepodléhá omezením zásobníku. Přiřadíte-li hodnotu jedné referenční proměnné jiné proměnné stejného typu, získáte dvě proměnné, které odkazují na stejný objekt v paměti. Když se referenční proměnná dostane mimo svůj obor, je odstraněna ze zásobníku způsobem popsaným v předchozí části, ale data odkazovaného objektu jsou nadále umístěna v haldě. Data zůstanou v haldě uložena buď do ukončení programu, nebo dokud je neodstraní automatická správa paměti, což se stane pouze v případě, kdy na ně již neodkazuje žádná proměnná (neexistuje na ně žádný odkaz).

Referenční datové typy jsou výkonným nástrojem, který budete ve svém kódu v C# často používat. Získáte totiž vysoký stupeň kontroly nad životností svých dat, protože je zaručeno, že zůstanou umístěna v haldě tak dlouho, dokud na ně směřují nějaké odkazy.

Úklid

V předchozím výkladu a na příslušných schématech jsme si ukázali, že řízená halda funguje velmi podobně jako zásobník, mj. i v tom, že po sobě vytvořené objekty jsou umístěny v paměti vedle sebe. Chcete-li tedy zjistit, kam umístit další objekt, můžete k tomu použít ukazatel haldy. Tento ukazatel určuje další volné paměťové místo a jeho hodnota se přizpůsobuje pokaždé, když do haldy přidáte další objekt. Věci však komplikuje fakt, že životnost objektů uložených v haldě není spřažena s oborem platnosti jednotlivých proměnných v zásobníku, které na ně odkazují.

Automatická správa paměti z haldy při spuštění odstraní všechny objekty, na které již nevede žádný odkaz. Výsledkem je, že v haldě budou roztroušeny objekty, mezi kterými se nachází právě uvolněná paměť (viz obrázek 12.4).



Obrázek 12.4

Kdyby zůstala řízená halda v tomto stavu, bylo by přidělování místa pro nové objekty velmi komplikované. Běhový systém by totiž musel v haldě pokaždé hledat blok paměti takové velikosti, která by postačovala k uložení nového objektu. Automatická správa paměti však neponechává haldu v uvedeném stavu. Ihned po uvolnění všech nepoužívaných objektů zkomprimuje automatická správa paměti haldu tak, že přesune všechny zbývající objekty, aby tvořily jediný souvislý blok paměti. To znamená, že s ohledem na algoritmus hledání místa pro uložení nových objektů může halda nadále fungovat podobně jako zásobník. Při přesunutí objektů je samozřejmě nutno aktualizovat všechny odkazy na tyto objekty s použitím správných nových adres, ale i o to se automatická správa paměti postará.

Touto fází komprese pomocí automatické správy paměti se fungování řízené haldy zásadně liší od klasických neřízených hald. Chcete-li najít místo k uložení nových dat, stačí v případě řízené haldy pouze načíst hodnotu ukazatele haldy a není nutné procházet spojový seznam adres. Z tohoto důvodu je vytvoření instance objektu v platformě .NET mnohem rychlejší. Zajímavé je, že bývá rychlejší i přístup k objektům, protože jsou v haldě shromážděny ve stejné oblasti paměti, což omezuje stránkování. Společnost Microsoft se domnívá, že tyto výkonnostní přínosy více než vyváží ztrátu výkonu, kdykoli musí automatická správa paměti komprimovat haldu a změnit všechny odkazy na přesunuté objekty.

> Obecně platí, že automatická správa paměti se spouští tehdy, když běhový systém .NET zjistí nutnost úklidu. Spuštění automatické správy paměti v určitém bodě svého kódu můžete sami vynutit voláním metody `System.GC.Collect()`. `System.GC` je třída .NET, která reprezentuje automatickou správu paměti, a metoda `Collect()` inicializuje úklid. Třída `GC` je určena pro výjimečné situace, kdy víte, že je vhodný čas ke spuštění automatické správy paměti (jestliže jste například v kódu zrušili odkazy na mnoho objektů). Vzhledem k logice automatické správy paměti však není zaručeno, že všechny objekty, na které nevede odkaz, budou z haldy odstraněny v jediném průchodu automatické správy paměti.

Uvolňování neřízených prostředků

Díky existenci automatické správy paměti se obvykle nemusíte starat o objekty, které již nepotřebujete. Stačí, když se všechny odkazy na tyto objekty dostanou mimo obor platnosti a automatická správa paměti uvolní paměť automaticky. Automatická správa paměti však neumí uvolnit neřízené prostředky (například popisovače souborů, síťová připojení a databázová připojení). Když řízené třídy zapouzdřují přímé nebo nepřímé odkazy na neřízené prostředky, musíte preventivně zajistit uvolnění těchto neřízených prostředků v okamžiku, kdy instanci třídy zruší automatická správa paměti.

V definici třídy můžete uvolnění neřízených prostředků automatizovat dvěma způsoby. Tyto mechanismy se často implementují společně, protože každý z nich řeší problém poněkud odlišným způsobem. Jedná se o následující mechanismy:

- Deklarace destruktoru (neboli finalizéru) jako členu třídy
- Implementace rozhraní `System.IDisposable` ve třídě

V následujících částech si postupně rozebereme oba mechanismy a potom si předvedeme, jak lze dosáhnout optimálních výsledků díky jejich společné implementaci.

Destruktory

Již jste se dozvěděli, že konstruktory umožňují určit akce, které je nutné provést při vytvoření jednotlivé instance třídy. Naopak destruktory jsou volány předtím, než je objekt pomocí automatické správy paměti zničen. Vzhledem k tomuto chování by se na první pohled mohlo zdát, že destruktory budou ideálním místem pro kód pro uvolňování neřízených prostředků a provedení obecného úklidu. Věci však bohužel nejsou tak jednoduché.

> Zmiňujeme zde sice destruktory v jazyce C#, ale v základní architektuře .NET se tyto prvky označují jako finalizéry. Když v jazyce C# definujete destruktor, překladač ve skutečnosti umístí do sestavení metodu s názvem `Finalize()`. Tento fakt nemá žádný vliv na zdrojový kód, ale měli byste si ho uvědomit, až budete zkoumat obsah sestavení.

Vývojářům v C++ bude syntaxe destruktoru povědomá. Připomíná metodu se stejným názvem jako třída, která tuto metodu obsahuje, její název však začíná vlnovkou (~). Nemá žádný návratový typ, nepřijímá žádné parametry a nemá přístupové modifikátory. Uveďme si příklad:

```
class MyClass
{
  ~MyClass()
  {
    // implementace destruktoru
  }
}
```

Když překladač C# přeloží destruktor, implicitně převede kód destruktoru na metodu `Finalize()`. Přitom zajistí, že bude spuštěna metoda `Finalize()` rodičovské třídy. Následující příklad představuje kód C# odpovídající kódu zprostředkujícího jazyka (IL), který by překladač generoval pro destruktor `~MyClass()`:

```
protected override void Finalize()
{
  try
  {
    // implementace destruktoru
  }
  finally
  {
    base.Finalize();
  }
}
```

Jak je patrné, kód implementovaný v destruktoru `~MyClass()` je zabalen do bloku `try`, který je součástí metody `Finalize()`. Zavolání metody `Finalize()` bázové třídy je zajištěno uložením odpovídajícího příkazu do bloku `finally`. Bloky `try` a `finally` se budeme zabývat v kapitole 14, „Chyby a výjimky“.

Zkušení vývojáři v C++ používají destruktory velmi často. Kromě uvolňování prostředků pomocí nich někdy také získávají ladicí informace nebo zajišťují jiné úkoly. Destruktory v C# se používají mnohem méně než jejich ekvivalenty v C++. Problém s destruktory v C# v porovnání s jejich protějšky v jazyce C++ spočívá v tom, že nejsou deterministické. Při zničení objektu v C++ je příslušný destruktor spuštěn okamžitě. Vzhledem ke způsobu, jakým funguje automatická správa paměti v C#, však nelze nijak zjistit, kdy destruktor objektu v C# bude skutečně spuštěn. Proto nemůžete do destruktoru umístit žádný kód, který vyžaduje spuštění v konkrétním čase, a neměli byste se ani spoléhat na to, že destruktory různých instancí třídy budou volány v určitém pořadí. Pokud vaše objekty uchovávají vzácné a kritické prostředky, které je nutné uvolnit co nejdříve, není vhodné čekat na automatický úklid.

Destruktory C# působí ještě další problém: implementace destruktoru zpomaluje konečné odstranění objektu z paměti. Objekty bez destruktoru jsou odstraněny z paměti v jediném průchodu automatické správy paměti, ale objekty s destruktory vyžadují ke svému zničení dva průchody: při prvním z nich se volá destruktor, ale objekt se neodstraní, a teprve při druhém průchodu je objekt skutečně odstraněn. Běhový systém navíc používá ke spuštění metod `Finalize()` všech odstraňovaných objektů jediný podproces. Pokud používáte destruktory často a provádíte pomocí nich zdlouhavé úklidové operace, může to mít zřetelný dopad na výkon.

Rozhraní IDisposable

Doporučenou alternativou k destruktorům v C# je použití rozhraní `System.IDisposable`. Rozhraní `IDisposable` definuje schéma (s podporou na úrovni jazyka), které poskytuje deterministický mechanismus uvolnění neřízených prostředků a zabraňuje problémům souvisejícím s automatickou správou paměti, které jsou vlastní destruktorům. Rozhraní `IDisposable` deklaruje jedinou metodu s názvem `Dispose()`, která nepřijímá žádné parametry a vrací `void`. Uveďme si implementaci třídy `MyClass`:

```
class MyClass: IDisposable
{
  public void Dispose()
  {
    // implementace
  }
}
```

Implementace metody `Dispose()` by měla explicitně uvolnit všechny neřízené prostředky přímo použité objektem a zavolat metodu `Dispose()` pro všechny zapouzdřené objekty, které rozhraní `IDisposable` také implementují. Díky tomu umožňuje metoda `Dispose()` přesně kontrolovat čas uvolnění neřízených prostředků.

Předpokládejme, že máte třídu s názvem `ResourceGobbler`, která se spoléhá na použití určitých externích prostředků a implementuje rozhraní `IDisposable`. Chcete-li vytvořit instanci této třídy, použít ji a poté ji zlikvidovat, můžete to provést takto:

```
ResourceGobbler theInstance = new ResourceGobbler();

// zpracování

theInstance.Dispose();
```

Tento kód bohužel neuvolní prostředky spotřebované instancí `theInstance`, jestliže při zpracování dojde k výjimce. Proto byste měli kód napsat následujícím způsobem s použitím bloku `try` (který si podrobně popíšeme v kapitole 14):

```
ResourceGobbler theInstance = null;

try
{
  theInstance = new ResourceGobbler();

  // zpracování
}
finally
{
  if (theInstance!= null) theInstance.Dispose();
}
```

Tato verze zajišťuje, že metoda `Dispose()` bude pro instanci `theInstance` volána vždy a že budou vždy uvolněny všechny prostředky spotřebované touto instancí, i když při zpracování dojde k výjimce. Pokud byste však pokaždé museli opakovat obdobnou konstrukci, zhoršila by se přehlednost kódu. Jazyk C# poskytuje syntaxi, která zaručuje automatické volání metody `Dispose()` pro objekt implementující rozhraní `IDisposable`, když se jeho odkaz dostane mimo obor platnosti. Slouží k tomu klíčové slovo `using`, které se tentokrát uplatňuje v odlišném kontextu, který nijak nesouvisí s jmennými prostory. Následující kód generuje kód jazyka IL ekvivalentní bloku `try` v předchozí ukázce:

```
using (ResourceGobbler theInstance = new ResourceGobbler())
{
  // zpracování
}
```

Příkaz `using`, po kterém následuje závorka s deklarací a vytvořením instance referenční proměnné, zajistí, že obor platnosti proměnné bude nastaven na doprovodný blok příkazů. Pokud se navíc daná proměnná dostane mimo obor platnosti, bude její metoda `Dispose()` zavolána automaticky, i když dojde k výjimce. Jestliže však již používáte bloky `try` k zachycení jiných výjimek, můžete se příkazu `using` vyhnout a jednoduše zavolat metodu `Dispose()` v klauzuli `finally` existujícího bloku `try`. Tento postup je čistší a omezuje počet odsazení v kódu.

> Pro určité třídy je označení úklidové metody identifikátorem `Close()` logičtější než `Dispose()` – to platí, pracujete-li například se soubory nebo s databázovými připojeními. V těchto případech je běžné implementovat rozhraní `IDisposable` a pak implementovat samostatnou metodu `Close()`, která jednoduše zavolá metodu `Dispose()`. Tento přístup zpřehledňuje práci s třídami, ale podporuje také příkaz `using` poskytovaný jazykem C#.

Implementace rozhraní IDisposable a destruktoru

V předchozích částech jsme si vysvětlili dvě alternativy, jak uvolnit neřízené prostředky používané uživatelskými třídami:

- Spuštění destruktoru vynucuje běhový systém, ale tento proces není deterministický a neúnosně zvyšuje režii běhového systému, což je dáno principem fungování automatické správy paměti.
- Rozhraní `IDisposable` dovoluje uživatelům třídy řídit, kdy budou prostředky uvolněny, ale vyžaduje disciplínu při volání metody `Dispose()`.

Nejlepší přístup je obecně založen na implementaci obou mechanismů, což spojuje jejich výhody a překonává jejich nedostatky. Rozhraní `IDisposable` implementujete s předpokladem, že většina programátorů zavolá metodu `Dispose()` správně, ale poskytnete i destruktor jako bezpečnostní mechanismus pro případ, že k volání metody `Dispose()` nedojde. Uveďme si příklad této podvojné implementace:

```
using System;

public class ResourceHolder: IDisposable
{

  private bool isDisposed = false;

  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (!isDisposed)
    {
      if (disposing)
      {
        // Úklid řízených objektů voláním jejich
        // metod Dispose()
      }
      // Úklid neřízených objektů
    }
    isDisposed = true;
  }

  ~ResourceHolder()
  {
    Dispose (false);
  }

  public void SomeMethod()
  {
    // Před spuštěním libovolné metody se kontroluje,
    // zda objekt již nebyl zlikvidován
```

```
    if(isDisposed)
    {
      throw new ObjectDisposedException("ResourceHolder");
    }

    // Implementace metody...
  }
}
```

Je patrné, že tento kód obsahuje další přetížení metody `Dispose()` s přístupem `protected`, které přijímá jeden parametr typu `bool`. Uvedená metoda také zajišťuje veškeré čištění. Metoda `Dispose(bool)` je volána destruktorem i metodou `IDisposable.Dispose()`. Tento přístup zajišťuje, že se veškerý kód pro úklid nachází na jediném místě.

Parametr předaný metodě `Dispose(bool)` určuje, zda byla metoda `Dispose(bool)` volána destruktorem nebo metodou `IDisposable.Dispose()`. Metoda `Dispose(bool)` by neměla být volána z žádného jiného místa kódu. Vycházíme z následujících předpokladů:

- Jestliže klient implementuje rozhraní `IDisposable.Dispose()`, dává tím najevo, že je vhodné uklidit všechny řízené a neřízené prostředky přidružené k danému objektu.
- Všechny prostředky je nutné uklidit i tehdy, když byl zavolán destruktor. V tomto případě však víte, že jej musela zavolat automatická správa paměti, a neměli byste se pokoušet o přístup k jiným řízeným objektům, protože si již nemůžete být jisti jejich stavem. V této situaci je optimální uklidit známé neřízené prostředky a doufat, že všechny odkazované spravované objekty také mají destruktory, které zajistí jejich vlastní úklid.

Členská proměnná `isDisposed` uvádí, zda byl objekt již zlikvidován, a zajišťuje, že se o likvidaci členských proměnných nepokusíte opakovaně. Před spuštěním libovolných instančních metod také umožňuje testovat, zda již objekt nebyl zlikvidován, jak to ukazuje metoda `SomeMethod()`. Tento zjednodušený přístup není bezpečný z hlediska podprocesů a spoléhá na to, že metodu nebude souběžně volat několik podprocesů. Požadavek, aby si klient vynutil synchronizaci, je však rozumný a opakovaně se uplatňuje v rámci knihoven tříd .NET (např. ve třídách kolekcí). Prací s podprocesy a synchronizací se budeme zabývat v kapitole 19, „Podprocesy a synchronizace“.

Metoda `IDisposable.Dispose()` nakonec obsahuje volání metody `System.GC.SuppressFinalize()`. `GC` je třída reprezentující automatickou správu paměti a metoda `SuppressFinalize()` sděluje automatické správě paměti, že již není nutné volat destruktor této třídy. Vzhledem k tomu, že vaše implementace metody `Dispose()` již zajistila veškerý požadovaný úklid, destruktor nadále nemá žádný smysl. Volání metody `SuppressFinalize()` znamená, že automatická správa paměti bude s příslušným objektem nakládat tak, jako by vůbec žádný destruktor neměl.

Nebezpečný kód

Z předchozího textu je zřejmé, že díky automatické správě paměti a použití odkazů dokáže jazyk C# před vývojářem velmi dobře skrývat většinu aspektů základní správy paměti. Někdy však budete chtít přímý přístup do paměti. Můžete například požadovat přístup k funkci v externí knihovně DLL (mimo .NET), která vyžaduje předání ukazatele jako parametru (což platí pro mnoho funkcí rozhraní Windows API). Přímý přístup může být také užitečný z výkonnostních důvodů. V této části se zaměříme na vlastnosti jazyka C#, které poskytují přímý přístup k obsahu paměti.

Přímý přístup do paměti pomocí ukazatelů

Představujeme zde sice *ukazatele*, jako by to bylo nové téma, ale ve skutečnosti o žádnou novinku nejde. Ve svém kódu jste již použili mnoho odkazů, což jsou v zásadě typově bezpečné ukazatele. Už jste viděli, že proměnné reprezentující objekty a pole prostě obsahují paměťovou adresu, na které jsou uložena odpovídající data (*odkazovaný objekt*). Ukazatel je jednoduše proměnná, která uchovává adresu jiných dat stejným způsobem jako odkaz. Rozdíl spočívá v tom, že jazyk C# neumožňuje přímý přístup k adrese, která je obsažena v referenční proměnné. V případě odkazu se s proměnnou syntakticky zachází tak, jako by obsahovala skutečnou hodnotu odkazované proměnné.

Odkazy v C# jsou navrženy tak, aby se jazyk snáze používal a nemohli jste náhodou provést operaci, která by poškodila obsah paměti. Oproti tomu v případě ukazatelů máte k dispozici skutečnou paměťovou adresu. Díky tomu získáváte silný nástroj k novým typům operací. K dané adrese můžete například přičíst 4 bajty, abyste mohli prozkoumat, nebo dokonce upravit libovolná data, která jsou uložena na adrese právě o 4 bajty dále.

Ukazatele se používají ze dvou hlavních důvodů:

- **Zpětná kompatibilita:** Bez ohledu na všechny funkce poskytované běhovým systémem .NET je stále možné volat nativní funkce rozhraní Windows API a někdy se může jednat o jediný způsob, jak určitou operaci provést. Tyto funkce rozhraní API jsou obvykle vytvořeny v jazyce C a často vyžadují jako parametry ukazatele. V mnoha případech však můžete napsat deklaraci `DllImport` tak, abyste se použití ukazatelů vyhnuli – například pomocí třídy `System.IntPtr`.
- **Výkon:** V situacích, kdy potřebujete získat maximální rychlost, mohou ukazatele posloužit k optimalizaci výkonu. Víte-li, co děláte, můžete zajistit nejefektivnější způsob přístupu k datům nebo manipulace s nimi. Uvědomte si však, že velmi často kód obsahuje jiné části, kde lze provést potřebné výkonnostní úpravy, aniž byste se museli uchylovat k použití ukazatelů. Pokuste se vyhledat úzké profily svého kódu pomocí nástroje Code Profiler. Jeden z těchto nástrojů se dodává spolu s Visual Studiem.

Nízkoúrovňový přístup k paměti má svou cenu. Syntaxe použití ukazatelů je složitější než u referenčních typů a není pochyb o tom, že se s ukazateli pracuje obtížněji. Chcete-li ukazatele uplatnit úspěšně, potřebujete dobré programátorské znalosti a vynikající schopnost pečlivě a logicky promýšlet činnost svého kódu. Nebudete-li pečliví, můžete kvůli ukazatelům do svého programu velmi snadno zanést nenápadné a obtížně zjistitelné chyby. Je například snadné přepsat jiné proměnné, způsobit přetečení zásobníku, přistupovat k oblastem paměti, které neobsahují žádné proměnné, nebo dokonce přepsat informace o vlastním kódu vyžadované běhovým systémem .NET, což bude mít za následek havárii programu.

Používáte-li ukazatele, musí navíc mechanismus zabezpečení přístupu ke kódu v běhovém systému udělit vašemu kódu vysokou úroveň důvěryhodnosti, aby jej bylo možné vůbec spustit. Na základě výchozí zásady zabezpečení přístupu ke kódu je to možné pouze tehdy, je-li kód spuštěn v místním počítači. Jestliže je nutné kód spustit ze vzdáleného místa, např. z Internetu, musí uživatelé udělit kódu dodatečná oprávnění, aby fungoval. Uživatelé tato oprávnění zpravidla neudělí, pokud vám i vašemu kódu nedůvěřují. Zabezpečením přístupu ke kódu se budeme podrobněji zabývat v kapitole 20, „Zabezpečení“.

Navzdory těmto problémům jsou ukazatele velmi silný a pružný nástroj při psaní efektivního kódu.

Rozhodně nedoporučujeme používat ukazatele zbytečně, protože to nejen komplikuje psaní a ladění kódu, ale takový kód navíc neprojde kontrolami typové bezpečnosti, které klade CLR. O tomto modulu jsme se zmínili v kapitole 1, „Architektura .NET".

Psaní nebezpečného kódu s klíčovým slovem unsafe

Vzhledem k rizikům souvisejícím s ukazateli umožňuje jazyk C# použít ukazatele pouze v blocích kódu, které k tomuto účelu speciálně označíte. Slouží k tomu klíčové slovo `unsafe`. Klíčovým slovem `unsafe` lze i označit jednotlivou metodu takto:

```
unsafe int GetSomeNumber()
{
  // kód, který používá ukazatele
}
```

Klíčovým slovem `unsafe` můžete označit libovolnou metodu nezávisle na dalších modifikátorech, které pro danou metodu použijete (jde například o modifikátory `static` nebo `virtual`). V případě metod se modifikátor `unsafe` vztahuje na parametry metody, což umožňuje použít jako parametry ukazatele. Jako `unsafe` je také možné označit celou třídu nebo strukturu, což znamená, že se za nebezpečné budou považovat všechny její členy:

```
unsafe class MyClass
{
  // S ukazateli nyní může pracovat
  // libovolná metoda v této třídě
}
```

Podobně můžete klíčovým slovem `unsafe` označit členskou proměnnou:

```
class MyClass
{
  // Deklarace datové složky ukazatele ve třídě
  unsafe int *pX;
}
```

Modifikátorem `unsafe` lze také označit blok kódu uvnitř metody:

```
void MyMethod()
{
  // kód, který nepoužívá ukazatele
  unsafe
  {
    // Nebezpečný kód, který používá ukazatele
  }
  // "Bezpečnější" kód, který nepoužívá ukazatele
}
```

Poznamenejme však, že jako `unsafe` nelze označit samotnou lokální proměnnou:

```
int MyMethod()
{
```

```
  unsafe int *pX;  // CHYBA
}
```

Chcete-li použít nebezpečnou lokální proměnnou, musíte ji deklarovat a použít uvnitř nebezpečné metody nebo bloku. Práce s ukazateli má ještě jednu podmínku. Překladač C# odmítne nebezpečný kód, pokud mu neoznámíte, že váš kód obsahuje nebezpečné bloky. K tomu je určen příznak `unsafe`. Chcete-li tedy přeložit soubor s názvem `MySource.cs`, který zahrnuje nebezpečné bloky, je nutné zadat následující příkaz (za předpokladu, že nepoužíváte žádné další možnosti překladače):

```
csc /unsafe MySource.cs
```

nebo:

```
csc -unsafe MySource.cs
```

> Pracujete-li v prostředí Visual Studia 2005 nebo 2008, naleznete také možnost překladu nebezpečného kódu na kartě Build okna vlastností projektu.

Syntaxe ukazatelů

Když označíte blok kódu klíčovým slovem `unsafe`, můžete deklarovat ukazatel pomocí následující syntaxe:

```
int* pWidth, pHeight;
double* pResult;
byte*[] pFlags;
```

V tomto kódu se deklarují čtyři proměnné: `pWidth` a `pHeight` jsou ukazatele na čísla typu `int`, `pResult` je ukazatel na typ `double` a `pFlags` je pole ukazatelů na hodnoty typu `byte`. Názvy ukazatelů běžně začínají písmenem `p`, které označuje, že se jedná o ukazatele (pointer). Symbol * v deklaraci proměnné indikuje, že deklarujete ukazatel (tj. něco, co obsahuje adresu proměnné uvedeného typu).

> Vývojáři v C++ by si měli všimnout syntaktických rozdílů mezi jazyky C++ a C#. Příkaz `int* pX, pY;` jazyka C# odpovídá příkazu `int *pX, *pY;` v jazyce C++. V jazyce C# je symbol * přidružen k typu, nikoli k identifikátoru proměnné.

Jakmile deklarujete proměnné typu ukazatel, můžete s nimi pracovat stejně jako s normálními proměnnými. Nejdříve se ale musíte seznámit s dalšími dvěma operátory:

- `&` znamená *„získej adresu proměnné“* a převádí hodnotový datový typ na ukazatel, například `int` na `*int`. Tento operátor se nazývá *operátor získání adresy*.
- * znamená *„získej obsah této adresy“* a převádí ukazatel na hodnotový datový typ (například `*float` na `float`). Tento operátor se označuje jako *operátor nepřímého přístupu* (někdy také *operátor dereferencování*).

Z těchto definic je zřejmé, že operátory `&` a * mají vzájemně opačný efekt.

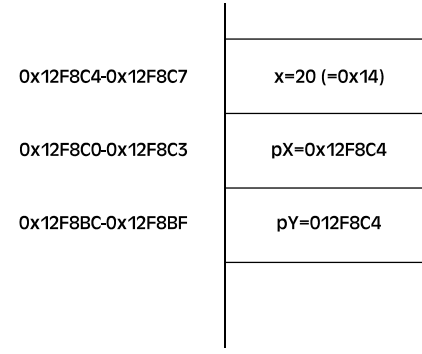
> Možná přemýšlíte nad tím, jak je možné používat symboly `&` a * tímto způsobem, když tyto symboly také znamenají operátory bitové konjunkce (`&`) a násobení (*). Ve skutečnosti je vždy možné poznat správný význam, protože v novém významu souvisejícím s ukazateli se tyto symboly vždy vyskytují

jako unární operátory – fungují pouze s jednou proměnnou a v kódu jsou uvedeny před ní. Na druhé straně operace bitové konjunkce a násobení jsou binární operace, které vyžadují dva operandy.

Následující kód obsahuje příklady použití těchto operátorů:

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
```

Nejdříve deklarujete proměnnou `x` typu `int` s hodnotou 10, po které následují dva ukazatele na celá čísla `pX` a `pY`. Potom nastavíte ukazatel `pX` tak, aby směřoval na proměnnou `x` (tj. uložíte do proměnné `pX` adresu proměnné `x`). Dále přiřadíte hodnotu ukazatele `pX` ukazateli `pY`, aby i ukazatel `pY` ukazoval na proměnnou `x`. Nakonec v příkazu `*pY = 20` uložíte hodnotu 20 na místo, na které směřuje ukazatel `pY`. Tím ve skutečnosti změníte obsah proměnné `x` na hodnotu `20`, protože ukazatel `pY` právě na proměnnou `x` směřuje. Všimněte si, že mezi proměnnými `pY` a `x` neexistuje žádná pevná vazba. Aktuálně pouze platí, že ukazatel `pY` směřuje na paměťové místo, kde je uložena proměnná `x`.



Obrázek 12.5

Chcete-li lépe porozumět tomu, co se zde odehrává, představte si, že je celé číslo `x` uloženo v zásobníku na paměťových místech `0x12F8C4` až `0x12F8C7` (`1243332` až `1243335` dekadicky). Jedná se o 4 místa, protože typ `int` zabírá 4 bajty. Vzhledem k tomu, že zásobník přiděluje paměť směrem dolů, bude proměnná `pX` uložena v místech `0x12F8C0` až `0x12F8C3` a proměnná `pY` se objeví v místech od `0x12F8BC` do `0x12F8BF`. Také proměnné `pX` a `pY` obsazují po čtyřech bajtech. Nesouvisí to s faktem, že typ `int` má velikost 4 bajty. Je to způsobeno tím, že 4 bajty jsou nutné k uložení adresy u 32bitového procesoru. V případě uvedených adres bude zásobník po spuštění předchozího kódu vypadat jako na obrázku 12.5.

Tento proces předvádíme na celých číslech, která jsou u 32bitového procesoru uložena v zásobníku za sebou, což však neplatí pro všechny datové typy. Důvod spočívá v tom, že 32bitové procesory fungují nejlépe, když načítají data z paměti ve shlucích velikosti 4 bajtů. Paměť u takových počítačů je obvykle rozdělena do bloků této velikosti a všechny takové bloky se v systému Windows někdy označují pojmem DWORD, protože tak se ve Windows v době před příchodem platformy .NET nazýval typ 32bitových celých čísel bez znaménka. (DWORD je zkratka anglického označení Double Word – dvojslovo.) Nejúčinnější je manipulovat v paměti s hodnotami typu DWORD – ukládání dat přes hranice těchto hodnot obvykle způsobuje pokles výkonu hardwaru. Běhový system .NET proto zpravidla doplňuje datové typy tak, aby velikost jimi obsazené paměti byla násobkem čísla 4. Například hodnota typu `short` má velikost 2 bajty, ale pokud ji umístíte do zásobníku, bude ukazatel zásobníku přesto snížen o 4 bajty, nikoli o 2, aby další proměnná uložená do zásobníku také začínala na hranici dvojslova.

Můžeme deklarovat ukazatel na libovolný hodnotový typ (tj. libovolný z předdefinovaných typů `uint`, `int`, `byte` atd., nebo na strukturu). Není však možné deklarovat ukazatel na třídu nebo pole, protože by to způsobovalo problémy automatické správě paměti. Automatická správa paměti potřebuje ke svému správnému fungování přesné informace o tom, které instance třídy byly v haldě vytvořeny a kde se nacházejí. Pokud by však kód začal manipulovat se třídami pomocí ukazatelů, mohl by velmi snadno poškodit v haldě informace týkající se tříd, které běhový systém. NET udržuje pro automatickou správu paměti. V tomto kontextu se libovolný datový typ, ke kterému má přístup automatická správa paměti, označuje jako *řízený typ*. Ukazatele lze deklarovat pouze jako *neřízené typy*, protože s nimi automatická správa paměti nemůže pracovat.

Přetypování ukazatelů na celočíselné typy

Ukazatel ve skutečnosti obsahuje celé číslo, které představuje adresu. Proto vás jistě nepřekvapí, že adresu v libovolném ukazateli lze převést na libovolný celočíselný typ nebo zpět. Převody mezi ukazateli a celočíselnými typy musí být explicitní. Implicitní konverze nejsou v tomto případě dovoleny. Následující zápis je například zcela v pořádku:

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int* pD = (int*)y;
```

Adresa obsažená v ukazateli `pX` je přetypována na typ `uint` a uložena do proměnné `y`. Potom je proměnná `y` přetypována zpět na typ `int*` a uložena do nové proměnné `pD`. Proměnná `pD` tedy nyní také ukazuje na hodnotu proměnné `x`.

Primární důvod pro přetypování hodnoty ukazatele na celočíselný typ je její zobrazení. Metody `Console.Write()` a `Console.WriteLine()` nemají žádná přetížení, která by přijímala ukazatele, ale přijmou a zobrazí hodnoty ukazatelů, které byly přetypovány na celočíselné typy:

```
Console.WriteLine("Address is " + pX);    // Špatně - výsledkem je chyba při překladu
Console.WriteLine("Adresa je " + (uint)pX); // OK
```

Ukazatel lze přetypovat na libovolný z celočíselných typů. Adresa však v 32bitových systémech zabírá 4 bajty. Přetypujete-li proto ukazatel na jakýkoli jiný typ než `uint`, `long` či `ulong`, téměř jistě dojde k přetečení. (Typ `int` způsobuje problémy proto, že má rozsah asi od –2 miliard do 2 miliard, zatímco adresy nabývají hodnot od nuly přibližně do 4 miliard.) Po vydání jazyka C# pro 64bitové procesory bude adresa zabírat 8 bajtů. V těchto systémech tedy pravděpodobně nastanou chyby přetečení, přetypujete-li ukazatel na libovolný typ kromě `ulong`. Je také důležité si uvědomit, že klíčové slovo `checked` se nevztahuje na převody ukazatelů. Tyto převody nezpůsobí výjimku při výskytu přetečení ani v kontextu klíčového slova `checked`. Běhový systém.NET předpokládá, že když pracujete s ukazateli, víte, co děláte, a možná přetečení vám nevadí.

Přetypování mezi typy ukazatelů

Můžete také explicitně převádět ukazatele, které směřují na různé typy. Například:

```
byte aByte = 8;
byte* pByte= &aByte;
double* pDouble = (double*)pByte;
```

Tento kód je naprosto správný, ačkoli opět platí, že při psaní podobného kódu musíte dbát opatrnosti. Pokud se v tomto příkladu podíváte na hodnotu typu `double`, na kterou směřuje ukazatel `pDouble`, ve skutečnosti budete zjišťovat obsah paměti, kde je uložena hodnota typu `byte` (`aByte`) a spolu s ní jiná data. S celou oblastí paměti budete zacházet tak, jako by obsahovala hodnotu typu `double`, která však nedává žádný smysl. Převody mezi typy však můžete využít při implementaci ekvivalentu unie v jazyce C nebo můžete přetypovat ukazatele na jiné typy na ukazatele na typy `sbyte`, chcete-li prozkoumat jednotlivé bajty v paměti.

Ukazatele typu void

Jestliže potřebujete pracovat s ukazatelem, ale nechcete uvést, na jaký typ dat ukazuje, můžete jej deklarovat jako ukazatel na typ `void`:

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

Tuto možnost lze využít hlavně při volání funkcí z rozhraní API Win32, které vyžadují parametry typu `void*`. V jazyce C# nemají ukazatele typu `void` příliš široké uplatnění. Konkrétně překladač označí jako chybu, jestliže se pokusíte dereferencovat ukazatel typu `void*` pomocí operátoru `*`.

Aritmetika ukazatelů[4]

Ukazatele umožňují přičítání a odečítání celých čísel. Překladač však přitom postupuje poměrně inteligentně. Předpokládejme například, že máte ukazatele na hodnotu `int` a pokusíte se k jeho hodnotě přičíst číslo 1. Překladač bude předpokládat, že ve skutečnosti chcete vyhledat paměťové místo, které následuje za proměnnou typu `int`, takže zvýší hodnotu o 4 bajty, což je velikost typu `int`. Bude-li se jednat o ukazatel na typ `double`, přičtení čísla 1 zvýší hodnotu ukazatele o 8 bajtů, tj. o velikost typu `double`. Pouze pokud ukazatel směřuje na proměnnou typu `byte` nebo `sbyte` (každá z nich má velikost 1 bajt), bude po přičtení čísla 1 k ukazateli jeho hodnota skutečně zvýšena o jednotku.

Při práci s ukazateli lze použít operátory +, -, +=, -=, ++ a --, jestliže se na pravé straně těchto operátorů vyskytuje proměnná typu `long` nebo `ulong`.

> Aritmetické operace s ukazateli typu void nejsou povoleny.

Předpokládejme například tyto definice:

```
uint u = 3;
byte b = 8;
```

[4] Poznámka českého vydavatele: Aritmetiku ukazatelů má smysl používat pro práci s poli alokovanými pomocí operátoru `stackalloc`, o nichž se v knize hovoří dále. Pro jiné účely se v podstatě nehodí.

```
double d = 10.0;
uint* pUint= &u;      // Typ uint má velikost 4 bajty
byte* pByte = &b;     // Typ byte má velikost 1 bajt
double* pDouble = &d;  // Typ double má velikost 8 bajtů
```

Dále předpokládejme, že tyto ukazatele směřují na adresy:

- `pUint`: 1243332,
- `pByte`: 1243328,
- `pDouble`: 1243320.

Potom spusťte tento kód:

```
++pUint;
// Přičte k ukazateli pUint (1*4) = 4 bajty

pByte -= 3;
// Odečte od ukazatele pByte (3*1) = 3 bajty
double* pDouble2 = pDouble + 4;
// pDouble2 = pDouble + 32 bajtů (4*8 bajtů)
```

Ukazatele nyní obsahují následující hodnoty:

- `pUint`: 1243336,
- `pByte`: 1243325,
- `pDouble2`: 1243352.

> Obecně platí, že po přičtení čísla X k ukazateli na typ T s hodnotou P získáte výsledek P + X*(sizeof(T)).

> Předchozí pravidlo musíte mít na paměti. Pokud jsou následné hodnoty určitého typu uloženy na paměťových místech, která na sebe navazují, lze ukazatele mezi jednotlivými místy pomocí přičítání celých čísel velmi dobře přesunovat. Jestliže ale pracujete s typy jako `byte` nebo `char`, jejichž velikosti nejsou násobkem čísla 4, nebudou následné hodnoty standardně uloženy v sousedních paměťových místech.

Lze také odečítat jeden ukazatel od jiného ukazatele, a to za předpokladu, že oba ukazatele směřují na stejný datový typ. V tomto případě je výsledkem hodnota typu `long`, jejíž hodnota je dána rozdílem mezi hodnotami ukazatelů děleným velikostí typu, na který tyto ukazatele směřují:

```
double* pD1 = (double*)1243324;
// Všimněte si, že je naprosto v pořádku
// inicializovat ukazatel tímto způsobem.

double* pD2 = (double*)1243300;
long L = pD1-pD2;
// Dá výsledek 3 (=24/sizeof(double))
```

Operátor sizeof

V této části se zmiňujeme o velikosti různých datových typů. Pokud potřebujete do svého kódu zadat velikost typu, můžete použít operátor `sizeof`, který jako parametr přijímá název datového typu a vrací počet bajtů, které daný typ zabírá. Například:

```
int x = sizeof(double);
```

Tento příkaz uloží do proměnné `x` hodnotu `8`.

Výhoda použití operátoru `sizeof` spočívá v tom, že nemusíte velikosti datových typů do programu zadávat pevně, takže je kód lépe přenositelný. U předdefinovaných typů vrací operátor `sizeof` tyto hodnoty:

```
sizeof(sbyte) = 1;          sizeof(byte) = 1;
sizeof(short) = 2;          sizeof(ushort) = 2;
sizeof(int) = 4;            sizeof(uint) = 4;
sizeof(long) = 8;           sizeof(ulong) = 8;
sizeof(char) = 2;           sizeof(float) = 4;
sizeof(double) = 8;         sizeof(bool) = 1;
```

Operátor `sizeof` můžete také použít pro struktury, které jste sami definovali. Pak však výsledek závisí na tom, jaké datové složky struktura obsahuje. Operátor `sizeof` nelze použít pro třídy a lze jej uvést pouze v bloku kódu označeném klíčovým slovem `unsafe`.

Ukazatele na struktury: operátor nepřímého přístupu ke složkám

Ukazatele na struktury fungují přesně stejně jako ukazatele na předdefinované hodnotové typy. Je však nutno splnit jednu podmínku: struktura nesmí obsahovat žádné referenční typy. To je dáno výše uvedeným omezením, že ukazatele nesmějí směřovat na referenční typy. Překladač tomu zabraňuje tak, že oznámí chybu, jestliže se pokusíte vytvořit ukazatel na libovolnou strukturu, jejíž součástí je nějaký referenční typ.

Předpokládejme, že máte strukturu s následující definicí:

```
struct MyStruct
{
  public long X;
  public float F;
}
```

Pak lze definovat ukazatel na tuto strukturu takto:

```
MyStruct* pStruct;
```

Poté jej můžete inicializovat tímto způsobem:

```
MyStruct Struct = new MyStruct();
pStruct = &Struct;
```

Ukazatel také umožňuje přístup k složkám struktury:

```
(*pStruct).X = 4;
(*pStruct).F = 3.4f;
```

Tato syntaxe je však poněkud komplikovaná. Proto definuje jazyk C# další operátor, který dovoluje přístup k členům struktur pomocí ukazatelů s použitím jednodušší syntaxe. Tento operátor se označuje jako *operátor nepřímého přístupu ke složkám* (*operátor šipka*) a jeho symbolem je pomlčka následovaná znakem větší než, takže vypadá jako šipka: `->`.

> Vývojářům v C++ je operátor nepřímého přístupu ke složkám určitě povědomý, protože jazyk C++ používá ke stejnému účelu stejný symbol.

Pomocí operátoru nepřímého přístupu ke složkám lze předchozí kód přepsat takto:

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

Můžete také přímo nastavit ukazatele příslušného typu tak, aby směřovaly na datové složky dané struktury:

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

nebo lze použít ekvivalentní zápis:

```
long* pL = &(pStruct->X);
float* pF = &(pStruct->F);
```

Ukazatele na složky třídy

Jak jsme již uvedli, nelze vytvořit ukazatel na třídu. Je to dáno tím, že automatická správa paměti neudržuje žádné informace o ukazatelích, pouze o odkazech. Pokud by tedy existovaly ukazatele na třídy, nemusela by automatická správa paměti fungovat správně.

Většina tříd však obsahuje členy hodnotového typu a může být vhodné vytvořit ukazatele na tyto složky. To je možné, ale vyžaduje to speciální syntaxi. Předpokládejme například, že přepíšete strukturu z předchozího příkladu jako třídu:

```
class MyClass
{
  public long X;
  public float F;
}
```

Potom můžete chtít vytvořit ukazatele na složky této třídy `X` a `F` stejným způsobem jako prve. Tento postup však bohužel způsobí chybu při překladu:

```
MyClass myObject = new MyClass();
long* pL = &(myObject.X);   // Špatně - chyba při překladu
float* pF = &(myObject.F);  // Špatně - chyba při překladu
```

Proměnné `X` a `F` jsou sice neřízeného typu, ale jsou zapouzdřeny v objektu, který je umístěn v haldě. Během úklidu může automatická správa paměti přesunout objekt `MyObject` na nové místo, takže by ukazatele `pL` a `pF` směřovaly na chybné paměťové adresy. Vzhledem k tomu překladač neumožňuje přiřadit adresy složek řízených typů ukazatelům tímto způsobem.

Řešení představuje klíčové slovo `fixed`, které automatické správě paměti oznámí, že mohou existovat ukazatele směřující na členy určitých objektů, takže tyto objekty nesmí přesunout. Při deklaraci pouze jednoho ukazatele se klíčové slovo `fixed` používá takto:

```
MyClass myObject = new MyClass();
fixed (long* pObject = &(myObject.X))
{
  // Nějaké operace
}
```

Proměnnou typu ukazatele lze definovat a inicializovat v závorkách za klíčovým slovem `fixed`. Obor platnosti této proměnné typu ukazatel (`pObject` v ukázce) odpovídá bloku `fixed`, který je uzavřen ve složených závorkách. Tento zápis informuje automatickou správu paměti, že bude-li zavolána při provádění kódu v bloku `fixed`, nesmí objekt `myObject` přesunout.

Chcete-li deklarovat několik ukazatelů, můžete před stejný blok kódu umístit odpovídající počet příkazů `fixed`:

```
MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
fixed (float* pF = &(myObject.F))
{
  // Nějaké operace
}
```

Pokud chcete pevně nastavit několik ukazatelů s různou dobou platnosti, můžete celé bloky `fixed` vnořovat:

```
MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
{
  // Nějaké operace s ukazatelem pX
  fixed (float* pF = &(myObject.F))
  {
    // Nějaké jiné operace s ukazatelem pF
  }
}
```

V jednom bloku `fixed` lze také inicializovat několik proměnných za předpokladu, že mají stejný typ:

```
MyClass myObject = new MyClass();
MyClass myObject2 = new MyClass();
fixed (long* pX = &(myObject.X), pX2 = &(myObject2.X))
{
  // atd.
```

Ve všech uvedených případech nezáleží na tom, zda různé deklarované ukazatele směřují na datové složky ve stejném objektu či v různých objektech nebo na statické datové složky, které nejsou přidruženy k žádné instanci třídy.

Příklad ukazatele: PointerPlayaround

V této části si rozebereme příklad, který používá ukazatele. Následující kód tvoří příklad s názvem `PointerPlayaround`. Provádí jednoduchou manipulaci s ukazateli a zobrazuje výsledky, takže můžete sledovat, co se v paměti děje a kde jsou proměnné uloženy:

```
using System;

namespace Wrox.ProCSharp.Memory
{
  class MainEntryPoint
  {
    static unsafe void Main()
    {
      int x=10;
      short y = -1;
      byte y2 = 4;
      double z = 1.5;
      int* pX = &x;
      short* pY = &y;
      double* pZ = &z;

      Console.WriteLine("Adresa proměnné x: 0x{0:X}, velikost: {1}, hodnota: {2}",
        (uint)&x, sizeof(int), x);
      Console.WriteLine("Adresa proměnné y: 0x{0:X}, velikost: {1}, hodnota: {2}",
        (uint)&y, sizeof(short), y);
      Console.WriteLine("Adresa proměnné y2: 0x{0:X}, velikost: {1}, hodnota: {2}",
        (uint)&y2, sizeof(byte), y2);
      Console.WriteLine("Adresa proměnné z: 0x{0:X}, velikost: {1}, hodnota: {2}",
        (uint)&z, sizeof(double), z);
      Console.WriteLine("Adresa ukazatele "
        + "pX=&x: 0x{0:X}, velikost: {1}, hodnota: 0x{2:X}",
        (uint)&pX, sizeof(int*), (uint)pX);
      Console.WriteLine("Adresa ukazatele "
        + "pY=&y: 0x{0:X}, velikost: {1}, hodnota: 0x{2:X}",
        (uint)&pY, sizeof(short*), (uint)pY);
      Console.WriteLine("Adresa ukazatele: "
        + "pZ=&z: 0x{0:X}, velikost: {1}, hodnota: 0x{2:X}",
        (uint)&pZ, sizeof(double*), (uint)pZ);

      *pX = 20;
      Console.WriteLine("Po nastavení *pX, x = {0}", x);
      Console.WriteLine("*pX = {0}", *pX);

      pZ = (double*)pX;
      Console.WriteLine("Proměnná x zpracovaná jako typ double = {0}", *pZ);
      Console.ReadLine();
```

```
    }
  }
}
```

Tento kód deklaruje čtyři hodnotové proměnné:

- proměnnou `x` typu `int`,
- proměnnou `y` typu `short`,
- proměnnou `y2` typu `byte`,
- proměnnou `z` typu `double`.

Deklaruje také ukazatele na tři z těchto hodnot: `pX`, `pY` a `pZ`.

Dále zobrazíte hodnoty těchto proměnných a také jejich velikosti a adresy. Všimněte si, že při práci s adresami proměnných `pX`, `pY` a `pZ` se v podstatě díváte na ukazatel *na* ukazatel – adresu adresy hodnoty. V souladu s běžnou praxí při zobrazení adres se v příkazech `Console.WriteLine()` používá specifikátor formátu `{0:X}`, který zajišťuje zobrazení paměťových adres v hexadecimálním formátu.

Nakonec pomocí ukazatele `pX` změníte hodnotu proměnné `x` na `20` a vyzkoušíte několik operací přetypování ukazatelů. Zjistíte tak, co se stane, když budete s obsahem proměnné `x` nakládat, jako by se jednalo o hodnotu typu `double`.

Při překladu a spuštění tohoto kódu dostanete následující výstup. Tento výstup na obrazovku také ukazuje, jaký výsledek má pokus o překlad s příznakem `/unsafe` a bez něj:

```
csc PointerPlayaround.cs
Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround.cs(7,26): error CS0227: Unsafe code may only appear if
        compiling with /unsafe

csc /unsafe PointerPlayaround.cs
Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround
Adresa proměnné x: 0x12F4B0, velikost: 4, hodnota: 10
Adresa proměnné y: 0x12F4AC, velikost: 2, hodnota: -1
Adresa proměnné y2: 0x12F4A8, velikost: 1, hodnota: 4
Adresa proměnné z: 0x12F4A0, velikost: 8, hodnota: 1.5
Adresa ukazatele pX=&x: 0x12F49C, velikost: 4, hodnota: 0x12F4B0
Adresa ukazatele: pY=&y: 0x12F498, velikost: 4, hodnota: 0x12F4AC
Adresa ukazatele pZ=&z: 0x12F494, velikost: 4, hodnota: 0x12F4A0
Po nastavení *pX, x = 20
*pX = 20
Proměnná x zpracovaná jako typ double = 2.86965129997082E-308
```

Kontrolou těchto výsledků si můžete ověřit správnost popisu fungování zásobníku, který jsme si uvedli v části „*Technické principy správy pamětí*“ výše v této kapitole. Zásobník přiděluje následným proměnným paměťová místa s klesajícími adresami. Všimněte si také, že bloky paměti jsou v zásobníku skutečně přidělovány po násobcích čtyřech bajtů. Například proměnná `y` je typu `short` (s velikostí dvou bajtů) a má (dekadickou) adresu `1242284`. Z toho je patrné, že jsou pro ni vyhrazena paměťová místa na adresách `1242284` až `1242287`. Kdyby běhový systém .NET ukládal proměnné bezprostředně vedle sebe, zabírala by proměnná `Y` pouze dvě adresy: `1242284` a `1242285`.

Další příklad předvádí aritmetiku ukazatelů a také ukazatele na struktury a členy tříd. Tento příklad je pojmenovaný `PointerPlayaround2`. Na začátku definujete strukturu s názvem `Currency Struct`, která reprezentuje finanční částku v dolarech a centech. Definujete také odpovídající třídu označenou `CurrencyClass`:

```
internal struct CurrencyStruct
{
  public long Dollars;
  public byte Cents;

  public override string ToString()
  {
    return "$" + Dollars + "." + Cents;
  }
}

internal class CurrencyClass
{
  public long Dollars;
  public byte Cents;

  public override string ToString()
  {
    return "$" + Dollars + "." + Cents;
  }
}
```

Když jste nyní definovali příslušnou strukturu a třídu, můžete na ně nastavit několik ukazatelů. Kód nového příkladu je uveden dále. Vzhledem k tomu, že je poměrně dlouhý, projdeme si jej podrobně. Nejdříve zobrazíte velikost struktury `CurrencyStruct`, vytvoříte dvojici instancí struktury `CurrencyStruct` a několik ukazatelů na strukturu `CurrencyStruct`. Pomocí ukazatele `pAmount` inicializujete datové složky struktury `CurrencyStruct` s názvem `amount1` a potom zobrazíte adresy svých proměnných:

```
public static unsafe void Main()
{
  Console.WriteLine("Velikost struktury CurrencyStruct: " + sizeof(CurrencyStruct));
  CurrencyStruct amount1, amount2;
  CurrencyStruct* pAmount = &amount1;
  long* pDollars = &(pAmount->Dollars);
```

```
byte* pCents = &(pAmount->Cents);

Console.WriteLine("Adresa struktury amount1: 0x{0:X}", (uint)&amount1);
Console.WriteLine("Adresa struktury amount2: 0x{0:X}", (uint)&amount2);
Console.WriteLine("Adresa ukazatele pAmount: 0x{0:X}", (uint)&pAmount);
Console.WriteLine("Adresa ukazatele pDollars: 0x{0:X}", (uint)&pDollars);
Console.WriteLine("Adresa ukazatele pCents: 0x{0:X}", (uint)&pCents);
pAmount->Dollars = 20;
*pCents = 50;
Console.WriteLine("Struktura amount1 obsahuje " + amount1);
```

Nyní si vyzkoušíte několik manipulací s ukazateli, které využívají vašich znalostí fungování zásobníku. Vzhledem k pořadí deklarace proměnných víte, že struktura `amount2` bude uložena na adrese ihned pod strukturou `amount1`. Operátor `sizeof(CurrencyStruct)` vrací hodnotu 16 (jak dokládá výstup na obrazovku), takže struktura `CurrencyStruct` zabírá násobek čtyř bajtů. Po dekrementaci tedy bude ukazatel směřovat na strukturu `amount2`:

```
--pAmount;  // po této operaci by měl ukazatel směřovat na amount2
Console.WriteLine("Struktura amount2 má adresu 0x{0:X} a obsahuje {1}",
  (uint)pAmount, *pAmount);
```

Všimněte si, že voláním metody `Console.WriteLine()` zobrazíte obsah struktury `amount2`, kterou jste však zatím neinicializovali. Zobrazí se nesmyslné hodnoty, které byly náhodou uloženy v daném paměťovém místě před spuštěním příkladu. Na tomto místě je důležité poznamenat, že normálně by překladač C# neinicializovanou proměnnou neumožnil použít, ale když začnete používat ukazatele, můžete velmi snadno obejít mnoho běžných kontrol při překladu. V tomto případě jste to udělali proto, že překladač nemůže nijak zjistit, že ve skutečnosti zobrazujete obsah struktury `amount2`. Víte to pouze vy, protože díky svým znalostem fungování zásobníku dokážete určit, jaký výsledek bude mít dekrementace ukazatele `pAmount`. Jakmile začnete používat aritmetiku ukazatelů, zjistíte, že můžete přistupovat ke všem typům proměnných a paměťových míst, ke kterým by vám překladač standardně neumožnil přístup. Proto se kód uplatňující aritmetiku ukazatelů označuje jako nebezpečný.

Dále použijete aritmetiku ukazatelů pro ukazatel `pCents`. Ukazatel `pCents` aktuálně směřuje na proměnnou `amount1.Cents`, ale pokusíte se jej nastavit tak, aby směřoval na proměnnou `amount2.Cents`. Opět přitom uplatníte operace s ukazateli a nebudete překladači přímo sdělovat, co máte v úmyslu. Chcete-li to provést, potřebujete dekrementovat adresu obsaženou v ukazateli `pCents` o hodnotu `sizeof(Currency)`:

```
// Nějaké inteligentní přetypování, aby ukazatel pCents
// směřoval na centy uvnitř struktury amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) ( --pTempCurrency );
Console.WriteLine("Adresa ukazatele pCents je nyní 0x{0:X}", (uint)&pCents);
```

Nakonec pomocí klíčového slova `fixed` vytvoříte několik ukazatelů, které budou směřovat na datové složky v instanci třídy, a nastavíte pomocí těchto ukazatelů hodnoty v příslušné instanci. Poznamenejme, že je to poprvé, kdy máte možnost vyhledat adresu položky uložené v haldě, a nikoli v zásobníku:

```
Console.WriteLine("\nNyní s třídami");
// Nyní to vyzkoušejte se třídami
CurrencyClass amount3 = new CurrencyClass();

fixed(long* pDollars2 = &(amount3.Dollars))
fixed(byte* pCents2 = &(amount3.Cents))
{
  Console.WriteLine("Proměnná amount3.Dollars má adresu 0x{0:X}", (uint)pDollars2);
  Console.WriteLine("Proměnná amount3.Cents má adresu 0x{0:X}", (uint) pCents2);
  *pDollars2 = -100;
  Console.WriteLine("Struktura amount3 obsahuje " + amount3);
}
```

Po překladu a spuštění tohoto kódu získáte přibližně následující výstup:

```
csc /unsafe PointerPlayaround2.cs
Microsoft (R) Visual C# 2008 Compiler version 3.05.20706.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround2
Velikost struktury CurrencyStruct: 16
Adresa struktury amount1: 0x12F4A4
Adresa struktury amount2: 0x12F494
Adresa ukazatele pAmount: 0x12F490
Adresa ukazatele pDollars: 0x12F48C
Adresa ukazatele pCents: 0x12F488
Struktura amount1 obsahuje $20.50
Struktura amount2 má adresu 0x12F494 a obsahuje $0.0
Adresa ukazatele pCents je nyní 0x12F488

Nyní s třídami
Proměnná amount3.Dollars má adresu 0xA64414
Proměnná amount3.Cents má adresu 0xA6441C
Struktura amount3 obsahuje $-100.0
```

Všimněte si, že v tomto výstupu je zobrazena neinicializovaná hodnota struktury `amount2` a že velikost struktury `CurrencyStruct` je `16` bajtů – to je poněkud více, než byste mohli očekávat vzhledem k velikost jejích datových složek (hodnoty typu `long` a `byte` by měly společně zabírat 9 bajtů).

Optimalizace výkonu pomocí ukazatelů

Všechny dosavadní příklady byly vytvořeny tak, aby demonstrovaly různé možnosti práce s ukazateli. Manipulovali jsme s pamětí způsobem, který je nejspíš zajímavý pouze pro ty, kteří chtějí poznat technické principy, ale v podstatě jej nelze využít k psaní lepšího kódu. V této části použijete své znalosti ukazatelů a vyzkoušíte si příklad, ve kterém díky rozumnému uplatnění ukazatelů výrazně zlepšíte výkonnost programu.

Vytvoření polí fungujících v zásobníku

V této části se podíváme na jednu z hlavních oblastí, kde mohou být ukazatele užitečné: při vytvoření vysoce výkonných polí s nízkou režií v zásobníku. Jak jsme uvedli v kapitole 2, „Základy jazyka C#“, poskytuje tento jazyk bohatou podporu manipulace s poli. Jazyk C# umožňuje velmi snadnou práci s jednorozměrnými a vícerozměrnými obdélníkovými nebo nepravidelnými (jagged) poli. Nevýhodou těchto polí však je, že se jedná o objekty, které jsou instancemi třídy `System.Array`. To znamená, že tato pole jsou uložena v haldě s veškerou režií, která s tím souvisí. V některých případech může být vhodné vytvořit krátkodobá výkonná pole, která nebudou vyžadovat režii referenčních objektů. Můžete k tomu použít ukazatele, ačkoli jak v této části zjistíte, lze to snadno provést pouze u jednorozměrných polí.

Chcete-li vytvořit vysoce výkonné pole, potřebujete nové klíčové slovo `stackalloc`. Příkaz `stackalloc` požádá běhový systém .NET o přidělení části paměti v zásobníku. Když zavoláte příkaz `stackalloc`, musíte doplnit dva údaje:

- typ dat, která chcete uložit,
- množství datových položek, které budete potřebovat uložit.

Jestliže například potřebujete přidělit dostatek paměti k uložení 10 datových položek typu `decimal`, můžete napsat:

```
decimal* pDecimals = stackalloc decimal[10];
```

Tento příkaz pouze přidělí paměť v zásobníku – nepokusí se inicializovat paměť žádnou výchozí hodnotou. Pro účely tohoto příkladu to vyhovuje, protože vytváříte vysoce výkonné pole a zbytečná inicializace by zhoršila výkon.

Obdobně při ukládání 20 datových položek typu `double` zadáte:

```
double* pDoubles = stackalloc double[20];
```

Ačkoli tento řádek kódu uvádí konstantní počet proměnných, které budou vytvořeny, může se stejně tak jednat o hodnotu vypočtenou během činnosti programu. Předchozí ukázku tedy můžete napsat takto:

```
int size;
size = 20;   // Nebo nějaká jiná hodnota vypočítaná za běhu
double* pDoubles = stackalloc double[size];
```

Z těchto úseků kódu je zřejmé, že syntaxe příkazu `stackalloc` je poněkud neobvyklá. Ihned za příkazem je uveden název datového typu, který chcete uložit (a musí se jednat o hodnotový typ). Pak následuje v hranatých závorkách počet položek, pro které potřebujete vyhradit místo. Počet přidělených bajtů bude určen vynásobením této hodnoty výrazem `sizeof(`*datový typ*`)`. Z použití hranatých závorek v předchozím kódu lze usoudit na pole, což příliš nepřekvapuje. Pokud jste přidělili místo na 20 hodnot typu `double`, získáte pole s 20 hodnotami typu `double`. Nejjednodušší typ pole, který můžete získat, je blok paměti, ve kterém je uložen jeden prvek za druhým (viz obrázek 12.6).

Tento diagram také znázorňuje ukazatel vrácený příkazem `stackalloc`, což je vždy ukazatel na přidělený datový typ, který směřuje na horní mez nově přiděleného bloku paměti. Chcete-li blok paměti použít, jednoduše dereferencujete vrácený odkaz. Pokud například potřebujete přidělit

místo pro 20 hodnot typu `double` a potom uložit do prvního prvku (prvek pole s indexem 0) hodnotu `3.0`, zadejte:

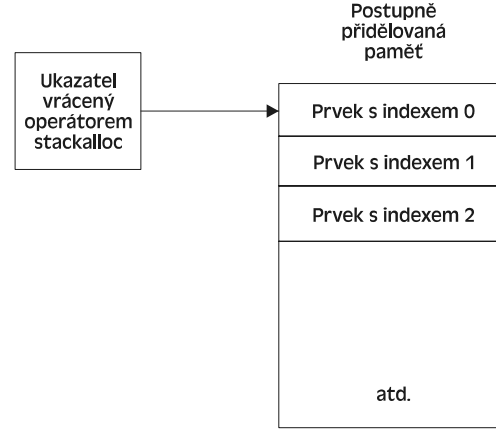
```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
```

Pro přístup k dalšímu prvku pole využijete aritmetiku ukazatelů. Jak jsme si již vysvětlili, jestliže přičtete k ukazateli číslo 1, zvýší se jeho hodnota o velikost libovolného datového typu, na který směřuje. V uvedeném případě to bude stačit k tomu, abyste se dostali na další volné paměťové místo v přiděleném bloku. Druhý prvek pole (prvek číslo `1`) lze tedy nastavit na hodnotu `8.4` následovně:

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
*(pDoubles+1) = 8.4;
```

Na základě stejného principu můžete získat přístup k prvku s indexem `X` pole pomocí výrazu `*(pDoubles+X)`.

K prvkům svého pole můžete v zásadě přistupovat tímto způsobem, ale pro praktické uplatnění je uvedená syntaxe příliš složitá. Jazyk C# naštěstí definuje alternativní syntaxi s použitím hranatých závorek – tedy obyčejné indexování, jak se na pole sluší a patří. Jazyk C# velmi přesně definuje význam hranatých závorek v kontextu ukazatelů. Pokud je proměnná `p` libovolného typu ukazatele a `X` je celočíselná proměnná, pak je výraz `p[X]` překladačem vždy interpretován jako `*(p+X)`. Platí to pro všechny ukazatele, nejen pro ukazatele inicializované klíčovým slovem `stackalloc`. Díky této zkrácené notaci máte nyní k dispozici velmi pohodlnou syntaxi pro přístup ke svému poli. V praxi to znamená, že můžete pro přístup k jednorozměrným polím vytvořeným v zásobníku přistupovat stejným způsobem jako k polím v haldě, která jsou reprezentována třídou `System.Array`:



Obrázek 12.6

```
double* pDoubles = stackalloc double [20];
pDoubles[0] = 3.0;  // pDoubles[0] je totéž jako *pDoubles
pDoubles[1] = 8.4;  // pDoubles[1] je totéž jako *(pDoubles+1)
```

Tato myšlenka aplikovat syntaxi polí na ukazatele není nová. Byla základní součástí jazyků C i C++ již od jejich vzniku. Vývojáři v C++ jistě poznali, že pole fungující v zásobníku, která lze získat klíčovým slovem `stackalloc`, jsou v podstatě shodná s klasickými poli založenými v zásobníku v jazycích C a C++. Mimo jiné díky této syntaxi a způsobu propojení ukazatelů a polí si jazyk C v 70. letech získal takovou oblibu a jedná se také o hlavní důvod, proč si programátoři v jazycích C a C++ programátorskou techniku využívající ukazatelů tolik oblíbili.

Vaše vysoce výkonné pole je sice přístupné stejným způsobem jako běžné pole v jazyce C#, je však nutné uvést jedno varování. Tento kód v jazyku C# způsobí výjimku:

```
double[] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

K výjimce dojde, protože se pokoušíte o přístup k poli pomocí indexu, který je mimo hranice pole. Index má hodnotu `50`, zatímco maximální povolená hodnota je `19`. Pokud ale deklarujete odpovídající pole příkazem `stackalloc`, není pole zabaleno do žádného objektu, který by meze pole kontroloval. Následující kód proto výjimku *nezpůsobí*:

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

V tomto kódu přidělíte dostatek paměti k uložení 20 hodnot typu `double`. Potom nastavíte, že v paměťových místech velikosti `sizeof(double)` počínaje místem, které je dáno přičtením výrazu `50*sizeof(double)` k začátku této paměti, bude uložena hodnota `3.0` typu `double`. Toto paměťové místo je naneštěstí daleko mimo oblast paměti, kterou jste k uložení hodnot typu double přidělili. Předem nelze nijak určit, jaká data budou na této adrese uložena. V nejlepším případě se může jednat o aktuálně nevyužitou paměť. Stejně tak je ale možné, že tímto způsobem přepíšete některé místo v zásobníku, které sloužilo k uložení jiných proměnných, nebo dokonce návratovou adresu z aktuálně prováděné metody. Opět vidíte, že za vysoký výkon získaný při práci s ukazateli se platí. Musíte s naprostou jistotou vědět, co děláte, jinak se při běhu setkáte s velmi podivnými chybami.

Příklad QuickArray

Náš výklad o ukazatelích ukončíme příkladem `QuickArray`, ve kterém si ukážeme použití klíčového slova `stackalloc`. V tomto příkladu se program jednoduše dotáže uživatele, kolik prvků chce poli vyhradit. Kód potom pomocí příkazu `stackalloc` vyhradí pole s daným počtem položek typu `long`. Prvky tohoto pole jsou inicializovány druhými mocninami celých čísel počínaje nulou a výsledky jsou zobrazeny na konzole:

```
using System;

namespace QuickArray
{
  internal class MainEntryPoint
  {
    private static unsafe void Main()
    {
      Console.Write("Jak velké pole chcete? \n> ");
      string userInput = Console.ReadLine();
      uint size = uint.Parse(userInput);

      long* pArray = stackalloc long [(int)size];
      for (int i = 0 ; i < size ; i++)
      {
        pArray[i] = i*i;
```

```
      }

      for (int i = 0 ; i < size ; i++)
      {
        Console.WriteLine("Prvek {0} = {1}", i, *(pArray+i));
      }

      Console.ReadLine();
    }
  }
}
```

Následuje výstup příkladu `QuickArray`:

```
QuickArray
Jak velké pole chcete?
> 15
Prvek 0 = 0
Prvek 1 = 1
Prvek 2 = 4
Prvek 3 = 9
Prvek 4 = 16
Prvek 5 = 25
Prvek 6 = 36
Prvek 7 = 49
Prvek 8 = 64
Prvek 9 = 81
Prvek 10 = 100
Prvek 11 = 121
Prvek 12 = 144
Prvek 13 = 169
Prvek 14 = 196
```

Shrnutí

Nezapomeňte, že chcete-li se stát opravdu profesionálním programátorem v C#, musíte do hloubky rozumět tomu, jak funguje přidělování a automatická správa paměti a úklid neřízených prostředků. V této kapitole jsme popsali, jak modul CLR spravuje a přiděluje paměť v haldě a v zásobníku. Také jsme si rozebrali, jak psát třídy, které správně uvolňují neřízené prostředky, a jak v jazyku C# používat ukazatele. V obou případech se jedná o pokročilá témata, kterým začínající programátoři špatně rozumějí, a často se při implementaci dopouštějí chyb.

Tuto kapitolu byste měli považovat za nezbytnou součást toho, co se dovíte v kapitole 14 o obsluze chybových stavů a v kapitole 19 o vláknech. V příští kapitole se podíváme na reflexi v C#.