Zaawansowane zarządzanie pamięcią w .NET

Lepszy kod, wydajność i skalowalność

—

Konrad Kokosa

Apress®

Apress®

Konrad Kokosa

Zaawansowane zarządzanie pamięcią w .NET

Lepszy kod, wydajność i skalowalność

przekład: Jakub Niedźwiedź

APN Promise, Warszawa 2020

Zaawansowane zarządzanie pamięcią w .NET. Lepszy kod, wydajność i skalowalność

First published in English under the title
Pro .NET Memory Management; For Better Code, Performance, and Scalability
by Konrad Kokosa, edition: 1

Polish language edition published by APN PROMISE S.A., Copyright © 2020

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: Pro .NET Memory Management; For Better Code, Performance, and Scalability, by Konrad Kokosa, edition: 1, opublikowanego przez APress Media, LLC, oddział Springer Nature.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

ISBN: 978-83-7541-413-4 (druk), 978-83-7541-412-7 (ebook)

Przekład: Jakub Niedźwiedź
Redakcja merytoryczna: Konrad Kokosa
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Ukochanej żonie – Justynie, bez której nic naprawdę cennego nie zdarzyłoby się w moim życiu.

Spis treści

O autorze

Konrad Kokosa jest doświadczonym projektantem oprogramowania i programistą szczególnie interesującym się technologiami Microsoft, ale przyglądającym się z zaciekawieniem również wielu innym. Programuje od kilkunastu lat, rozwiązując problemy wydajnościowe i zagadki architektoniczne w świecie .NET oraz projektując i przyspieszając aplikacje .NET. Jest niezależnym konsultantem, prowadzi blog http://tooslowexception.com, aktywnie uczestniczy w konferencjach i spotkaniach członków społeczności oraz jest fanem Twittera (@konradkokosa). Jego pasją jest też prowadzenie szkoleń w obszarze .NET, zwłaszcza związanych z wydajnością aplikacji, dobrymi praktykami kodowania oraz diagnostyką. Posiada tytuł Microsoft MVP w kategorii Visual Studio and Development Tools. Jest współzałożycielem Dotnetos.org – inicjatywy trzech fanów .NET, mającej na celu organizowanie pokazów i konferencji dotyczących wydajności w .NET.

O recenzentach technicznych

Damien Foggon jest programistą, autorem i recenzentem technicznym w zakresie najnowszych technologii, który przyczynił się do wydania ponad 50 książek dotyczących .NET, C#, Visual Basic i ASP.NET. Jest współzałożycielem grupy użytkowników NEBytes z Newcastle (w sieci pod adresem http://www.nebytes.net), posiada wiele certyfikatów MCPD w zakresie .NET 2.0 i kolejnych wersji, a w sieci można go znaleźć pod adresem http://blog.fasm.co.uk.

Maoni Stephens jest architektką i główną programistką w zakresie .NET GC w firmie Microsoft. Jej blog można znaleźć pod adresem https://blogs.msdn.microsoft.com/maoni/.

Podziękowania

Przede wszystkim bardzo chciałbym podziękować swojej żonie. Bez jej wsparcia ta książka nigdy by nie powstała. Zaczynając pracę nad tą książką, nie wyobrażałem sobie, jak wiele czasu, który moglibyśmy spędzić razem, będę musiał poświęcić, aby ją napisać. Dziękuję Ci za cierpliwość, wsparcie i otuchę, jakimi obdarowałaś mnie w tym czasie!

Po drugie, chciałbym podziękować Maoni Stephens za wyczerpujące, dokładne i bezcenne uwagi podczas recenzowania pierwszych wersji tej książki. Bez cienia wątpliwości mogę powiedzieć, że dzięki niej ta książka jest lepsza. Fakt, że główna programistka .NET GC pomagała mi w pisaniu tej książki, jest dla mnie nagrodą samą w sobie! Wiele podziękowań kieruję też do innych członków zespołu .NET, którzy pomagali mi w zweryfikowaniu niektórych części tej książki przy wielkiej pomocy ze strony Maoni (w kolejności poświęconych nakładów pracy): Stephen Toub, Jared Parsons, Lee Culver, Josh Free i Omar Tawfik. Chciałbym też podziękować Markowi Probstowi z Xamarin za przejrzenie uwag dotyczących środowiska uruchomieniowego Mono. Specjalne podziękowania należą się Patrickowi Dussud, „ojcu .NET GC" za czas poświęcony na zrecenzowanie historii powstania CLR.

Po trzecie, chciałbym podziękować Damienowi Foggonowi, recenzentowi technicznemu z wydawnictwa Apress, który poświęcił wiele pracy na skrupulatną weryfikację wszystkich rozdziałów. Jego doświadczenie w wydawaniu i pisaniu książek było bezcenne i sprawiło, że ta książka stała się bardziej zrozumiała i spójna. Nieraz byłem zaskoczony dokładnością uwag i sugestii Damiena.

Chciałbym podziękować wszystkim pracownikom Apress, bez których ta książka nie zostałaby wydana. Specjalne podziękowania kieruję do Laury Berendson (redaktor prowadzący), Nancy Chen (redaktor koordynujący) i Joan Murray (starszy redaktor) za wsparcie i cierpliwość przy ciągłym przedłużaniu terminów. W pewnym momencie data dostarczenia ostatecznej wersji stała się tematem tabu w naszych rozmowach! Chciałbym też podziękować Gwenan Spearing, z którą zacząłem pracę nad tą książką, ale nie zdążyłem jej ukończyć, zanim odeszła z zespołu Apress.

Chciałbym podziękować świetnej społeczności .NET w Polsce i na całym świecie za inspiracje czerpane z wielu prezentacji, artykułów i wpisów w sieciach

społecznościowych, za całe poparcie i zachęty oraz za niekończące się pytania „Jak tam prace nad książką?". Takie podziękowania kieruję szczególnie do następujących osób (kolejność alfabetyczna): Maciej Aniserowicz, Arkadiusz Benedykt, Sebastian Gębski, Michał Grzegorzewski, Jakub Gutkowski, Paweł Klimczyk, Szymon Kulec, Paweł Łukasik, Alicja Musiał, Łukasz Olbromski, Łukasz Pyrzyk, Bartek Sokół, Sebastian Solnica, Paweł Sroczyński, Jarek Stadnicki, Piotr Stapp, Michał Śliwoń, Szymon Warda i Artur Wincenciak, a także do wszystkich posiadaczy tytułu MVP (zwłaszcza z zakresu Azure) oraz do wielu innych; przepraszam szczerze tych, których pominąłem; wielkie dzięki wszystkim, którzy uważają, że zasłużyli na takie podziękowania. Wymienienie tutaj wszystkich po prostu nie jest możliwe. Dziękuję za inspirację i otuchę.

Chciałbym podziękować wszystkim doświadczonym autorom, którzy znaleźli czas na udzielenie mi rad dotyczących pisania książek, szczególnie Tedowi Newardowi (`http://blogs.tedneward.com/`) i Jonowi Skeetowi (`https://codeblog.jonskeet.uk`) – choć założę się, że nie pamiętają tych rozmów! Andrzej Krzywda (`http://andrzejonsoftware.blogspot.com`) i Gynvael Coldwind (`https://gynvael.coldwind.pl`) również udzielili mi wielu cennych rad dotyczących pisania i wydawania książek.

Następnie chciałbym wymienić wszystkich twórców świetnych narzędzi i bibliotek, z których korzystałem podczas pisania tej książki: Andrey Shchekin, twórca SharpLab (`https://sharplab.io`); Andrey Akinshin, twórca BenchmarkDotNet (`https://benchmarkdotnet.org`) oraz Adam Sitnik, główny wspierający ten projekt; Sergey Teplyakov, twórca ObjectLayoutInspector (`https://github.com/SergeyTeplyakov/ObjectLayoutInspector`); 0xd4d, anonimowy twórca dnSpy (`https://github.com/0xd4d/dnSpy`); Sasha Goldshtein, twórca wielu przydatnych narzędzi pomocniczych (`https://github.com/goldshtn`); wszyscy twórcy takich świetnych narzędzi, jak PerfView i WinDbg (oraz wszystkich rozszerzeń związanych z .NET).

Chciałbym też podziękować swojemu wcześniejszemu pracodawcy, Bankowi Millennium za pomoc i wsparcie przy rozpoczęciu prac nad tą książką. Nasze drogi się rozeszły, ale zawsze będę pamiętać, że właśnie tam zaczęła się moja przygoda z pisaniem, prowadzeniem bloga i prezentacjami na konferencjach. Wiele podziękowań kieruję też zbiorczo do swoich byłych współpracowników za mnóstwo motywacji i zachęt poprzez dopytywanie się o postępy prac nad książką.

Chciałbym też podziękować wszystkim użytkownikom serwisu Twitter, którzy odpowiadali na moje ankiety i pytania związane z książką, dając mi wskazówki, co jest, a co nie jest ciekawe, przydatne i cenne z punktu widzenia użytkowników .NET.

Na koniec chciałbym zbiorczo podziękować całej mojej rodzinie i wszystkim przyjaciołom, którym brakowało mnie w czasie, gdy pracowałem nad tą książką.

Przedmowa

Gdy dołączyłam do zespołu Common Language Runtime (środowisko uruchomieniowe dla .NET) ponad 10 lat temu, nie miałam pojęcia, że element zwany odśmiecaniem pamięci (Garbage Collector) stanie się tym, czemu będę poświęcać większość swojego czasu w dalszym zawodowym życiu. Wśród kilku pierwszych osób, z którymi współpracowałam w tym zespole, był Patrick Dussud, który był zarówno architektem, jak i programistą CLR GC od jego powstania. Po kilku miesiącach przekazał mi pałeczkę i stałam się następną osobą poświęcającą swój czas w zespole CLR głównie na rozwijanie GC.

Tak zaczęła się moja przygoda z GC. Wkrótce odkryłam, jak fascynujący jest świat odśmiecania pamięci – byłam zdumiona złożonymi i rozległymi wyzwaniami związanymi z GC i bardzo podobało mi się znajdowanie dla nich sprawnych rozwiązań. W miarę jak środowisko CLR było używane w coraz większej liczbie scenariuszy przez coraz więcej użytkowników, a pamięć stanowiła jeden z najważniejszych aspektów wydajnościowych, pojawiały się nowe wyzwania w obszarze zarządzania pamięcią. Gdy zaczynałam, rzadko można było spotkać stertę GC o rozmiarze 200MB; obecnie sterta o wielkości 20GB nie jest wyjątkiem. Kilka z najbardziej obciążonych procesów na świecie jest obecnie obsługiwanych przez CLR. Sposoby lepszego obsługiwania pamięci w ich przypadku stanowią fascynujący problem.

W roku 2015 otworzyliśmy kod źródłowy CoreCLR. Gdy to zostało ogłoszone, członkowie społeczności pytali, czy źródła GC będą dostępne w repozytorium CoreCLR – co nie było takie oczywiste, gdyż nasza implementacja GC zawierała wiele innowacyjnych mechanizmów i zasad. Odpowiedź była twierdząca, a był to ten sam kod GC, który wykorzystywaliśmy w podstawowym środowisku CLR. To przyciągnęło wielu ciekawskich. Rok później byłam zachwycona, gdy dowiedziałam się, że jeden z naszych klientów zamierza napisać książkę ściśle dotyczącą naszej implementacji GC. Gdy pracownik z naszego polskiego biura zapytał mnie, czy mogłabym zrecenzować książkę Konrada, oczywiście się zgodziłam.

Gdy otrzymałam rozdziały książki od Konrada, stało się dla mnie jasne, że pilnie przestudiował on nasz kod GC. Byłam pod wrażeniem szczegółowości omawianych zagadnień. Można oczywiście samemu zbudować kod CoreCLR i przechodzić przez

niego krok po kroku. Książka ta z pewnością znacznie to wszystkim ułatwi. Ponieważ ważną część czytelników tej książki stanowią użytkownicy GC, Konrad zamieścił mnóstwo materiałów pozwalających lepiej zrozumieć zachowanie GC oraz wiele wzorców programowania pomagających wydajniej korzystać z GC. Na początku książki można też znaleźć podstawowe informacje na temat pamięci, a pod koniec książki omówienie wykorzystania pamięci w różnych bibliotekach. Uważam, że w książce panuje idealna równowaga pomiędzy wprowadzeniem do GC, omówieniem wewnętrznych mechanizmów i przykładami zastosowań.

Książka ta jest dla każdego, kto korzysta z .NET i komu zależy na wydajnym działaniu pamięci oraz dla każdego, kto po prostu jest ciekaw działania .NET GC i chce zrozumieć dokładnie działanie tej implementacji. Mam nadzieję, że czytanie tej książki sprawi wszystkim tyle radości, co mnie jej recenzowanie.

Maoni Stephens
Lipiec 2018

Wprowadzenie

W informatyce zawsze mieliśmy do czynienia z pamięcią – od kart perforowanych poprzez taśmy magnetyczne aż do dzisiejszych, skomplikowanych układów DRAM. Pamięć zawsze będzie obecna, w przyszłości być może w formie jakichś układów holograficznych lub jeszcze bardziej zadziwiających technologii, których jeszcze nie jesteśmy sobie w stanie wyobrazić. Oczywiście pamięć istnieje nie bez powodów. Dobrze wiadomo, że programy komputerowe to połączenie algorytmów i struktur danych. Bardzo lubię to określenie. Chyba każdy co najmniej raz słyszał o książce *Algorytmy + Struktury danych = Programy* napisanej przez Niklausa Wirtha (Prentice Hall, 1976), gdzie określenie to powstało.

Od samego początku inżynierii oprogramowania zarządzanie pamięcią było zagadnieniem znanym ze swojego znaczenia. Od pierwszych urządzeń komputerowych inżynierowie musieli myśleć o przechowywaniu algorytmów (kodu programu) i struktur danych (danych programu). Zawsze było ważne, jak i gdzie te dane są ładowane i przechowywane na później.

W tym aspekcie inżynieria oprogramowania i zarządzanie pamięcią zawsze były nieodłącznie powiązane. Sądzę też, że tak będzie zawsze. Pamięć jest ograniczonym zasobem i zawsze takim będzie. Dlatego do pewnego stopnia pamięć zawsze będzie zaprzątać umysły przyszłych programistów. Jeśli pewien zasób jest ograniczony, zawsze może pojawić się jakiś błąd lub nieprawidłowy sposób użycia, który doprowadzi do wyczerpania tego zasobu. Pamięć nie jest tutaj wyjątkiem.

To powiedziawszy, z pewnością jest jedna rzecz dotycząca zarządzania pamięcią, która się stale zmienia – ilość. Pierwsi programiści albo inżynierowie oprogramowania musieli zwracać uwagę na każdy pojedynczy bit w swoich programach. Później mieli do dyspozycji kilobajty pamięci. Z każdą dekadą liczby te rosły i dzisiaj żyjemy w czasach gigabajtów, gdy terabajty i petabajty czekają już grzecznie na progu na swoją kolej. W miarę jak rozmiar pamięci rośnie, maleją też czasy dostępu, co umożliwia przetwarzanie wszystkich tych danych w zadowalającym czasie. Choć jednak możemy stwierdzić, że pamięć jest szybka, proste algorytmy zarządzania pamięcią, próbujące przetwarzać gigabajty danych bez żadnych optymalizacji i bardziej skomplikowanego dostrajania, nie są wystarczające. Dzieje się tak głównie dlatego, że czasy dostępu do pamięci zmniejszają się wolniej niż poprawia się moc obliczeniowa procesorów, które

z niej korzystają. Trzeba zwracać szczególną uwagę, aby nie wprowadzać wąskich gardeł do zarządzania pamięcią, które ograniczałyby moc obecnych procesorów.

To sprawia, że zarządzanie pamięcią nie tylko jest bardzo istotne, ale też stanowi niezwykle fascynującą część informatyki. Jeszcze lepiej jest dzięki automatycznemu zarządzaniu pamięcią. Nie wystarczy tylko powiedzieć „zwolnijmy nieużywane obiekty". Co, jak i kiedy – te proste aspekty zarządzania pamięcią sprawiają, że jest to stale trwający proces poprawiania starych i wymyślania nowych algorytmów. Niezliczone prace naukowe i doktoraty zajmują się sposobami automatycznego zarządzania pamięcią w możliwie optymalny sposób. Wydarzenia, takie jak ISMM (International Symposium on Memory Management – międzynarodowe sympozjum dotyczące zarządzania pamięcią) pokazują co roku, jak wiele dzieje się w tej dziedzinie w zakresie odśmiecania pamięci, alokacji dynamicznej oraz interakcji z środowiskami uruchomieniowymi, kompilatorami i systemami operacyjnymi. Później badania akademickie zmieniają się w komercyjne i otwarte produkty, z których korzystamy w codziennej pracy.

.NET jest idealnym przykładem środowiska zarządzanego, gdzie cała ta złożoność jest ukryta pod spodem i jest dostępna dla programistów w postaci przyjemnej, gotowej do użycia platformy. W istocie możemy korzystać z niej, nie mając żadnej świadomości wewnętrznej złożoności, co jest świetnym osiągnięciem .NET. Jednakże im bardziej nasz program będzie wrażliwy na problemy wydajnościowe, tym mniej możliwe będzie obejście się bez wiedzy na temat wewnętrznego sposobu działania pamięci. Osobiście uważam też, że dobrze jest wiedzieć, jak działają rzeczy, z których codziennie korzystamy!

Napisałem tę książkę w taki sposób, jak chciałbym ją przeczytać wiele lat temu – gdy zaczynałem swoją przygodę w dziedzinie wydajności .NET i diagnostyki. Książka ta nie zaczyna się więc od typowego wprowadzenia do pojęć sterty i stosu oraz opisu wielu generacji. Zaczynam natomiast od najważniejszych ogólnych podstaw zarządzania pamięcią. Innymi słowy, próbowałem napisać tę książkę w sposób pozwalający dobrze poczuć ten bardzo ciekawy temat, a nie tylko przedstawić mechanizm odśmiecania pamięci w .NET i sposoby jego działania. Odpowiedzi na pytania co, jak, a przede wszystkim – dlaczego – powinny pomóc czytelnikom w zrozumieniu wewnętrznego działania zarządzania pamięcią w .NET. Wszystkie przyszłe lektury dotyczące tego tematu powinny być bardziej zrozumiałe dla czytelników po przeczytaniu tej książki. Próbuję przekazać też wiedzę nieco bardziej ogólną niż tylko związaną z .NET, zwłaszcza w pierwszych dwóch rozdziałach. Prowadzi to do głębszego zrozumienia tematu, który może mieć często zastosowanie również w innych zadaniach inżynierii oprogramowania (dzięki zrozumieniu algorytmów, struktur danych i po prostu dobrych zasad inżynierskich).

Chciałem napisać tę książkę w sposób przyjazny dla każdego programisty .NET. Bez względu na stopień zaawansowania, każdy powinien tutaj znaleźć coś ciekawego dla siebie. Choć zaczniemy od podstaw, młodzi programiści szybko będą mieli okazję zagłębić

się w wewnętrzne szczegóły platformy .NET. Zaawansowanych programistów bardziej zaciekawią szczegóły implementacyjne. Przede wszystkim niezależnie od doświadczenia każdy powinien być w stanie skorzystać z przedstawionych praktycznych przykładów kodu i diagnostyki problemów.

Wiedza uzyskana z tej książki powinna pomóc czytelnikom w pisaniu lepszego kodu – bardziej wydajnego i świadomego zarządzania pamięcią, z wykorzystaniem odpowiednich funkcji bez obawy przed ich niezrozumieniem. Będzie to też prowadzić do lepszej wydajności i skalowalności tworzonych aplikacji – im bardziej zorientowany na pamięć będzie pisany kod, tym mniej będzie podatny na wąskie gardła i nieoptymalne korzystanie z zasobów. Mam nadzieję, że po przeczytaniu tej książki każdy uzna podtytuł „Lepszy kod, wydajność i skalowalność" za uzasadniony.

Mam też nadzieję, że wszystko to sprawi, iż ta książka będzie bardziej ogólna niż prosty opis obecnego stanu platformy .NET i jej wewnętrznych mechanizmów. Niezależnie od tego, jak będą ewoluować przyszłe wersje platformy .NET, wierzę, że większość wiedzy wyniesionej z tej książki pozostanie przydatna przez długi czas. Nawet jeśli pewne szczegóły implementacyjne się zmienią, czytelnik będzie je w stanie łatwo zrozumieć, dzięki wiedzy uzyskanej z tej książki. Ogólne zasady po prostu nie będą się zmieniać tak szybko. Życzę wszystkim przyjemnej podróży przez ogromny i ciekawy temat automatycznego zarządzania pamięcią!

Chciałbym też podkreślić kilka kwestii, które nie są szczególnie obecne w tej książce. Temat zarządzania pamięcią, choć wydaje się na pierwszy rzut oka bardzo wąski i specjalistyczny, jest zaskakująco szeroki. Choć poruszam mnóstwo zagadnień, to czasami nie są one przedstawiane tak szczegółowo, jakbym chciał (ze względu na brak miejsca). Nawet przy takich ograniczeniach, książka ta ma ponad 1000 stron! Do pominiętych tematów należą na przykład wyczerpujące odwołania do innych środowisk zarządzanych (takich jak Java, Python czy Ruby). Przepraszam też fanów F# za tak nieliczne odwołania do tego języka. Po prostu nie starczyło stron na porządny opis tych zagadnień, a nie chciałem poruszać pewnych tematów bez wyczerpującego ich omówienia. Chciałbym też móc poświęcić więcej uwagi środowisku Linux, ale jest to zagadnienie tak świeże i bez wielu dostępnych narzędzi, że poruszam je tylko ogólnie w rozdziale 3 (i całkowicie pomijam świat macOs z tych samych powodów). Oczywiście pominąłem też dużą część innych zagadnień wydajnościowych w .NET niezwiązanych bezpośrednio z pamięcią, jak wielowątkowość.

Choć starałem się jak najlepiej przedstawić praktyczne zastosowanie omawianych zagadnień i technik, to nie zawsze jest to możliwe bez zrobienia tego w całkiem wyczerpujący sposób. Praktycznych zastosowań jest po prostu zbyt dużo. Raczej oczekuję od czytelnika, że będzie czytał uważnie, poświęci czas na przemyślenie danego zagadnienia i zastosuje zdobytą wiedzę w swojej codziennej pracy. Wystarczy zrozumieć, jak coś działa, aby być w stanie z tego korzystać!

Odnosi się to w szczególności do tzw. scenariuszy. Warto zauważyć, że wszystkie scenariusze zawarte w tej książce mają charakter poglądowy. Ich kod został ograniczony do niezbędnego minimum, aby łatwiej pokazać główną przyczynę pojedynczego problemu. W rzeczywistości może istnieć wiele różnych powodów zaobserwowanego błędnego działania programu (tak jak na wiele sposobów można zaobserwować wycieki pamięci). Scenariusze zostały przygotowane w sposób pomagający zilustrować takie problemy na pojedynczym przykładzie, ponieważ oczywiście nie da się zawrzeć wszystkich możliwych przyczyn danego problemu w jednej książce. Co więcej, w rzeczywistych scenariuszach sytuacja będzie zagmatwana przez szum powodowany dodatkowymi danymi oraz wybór ślepych ścieżek badania problemu. Często nie istnieje tylko jeden sposób na rozwiązanie przedstawionych problemów, ale wiele sposobów na znalezienie podstawowej przyczyny podczas analizy problemów. Takie rozwiązywanie problemów staje się mieszanką zadania czysto inżynierskiego z odrobiną sztuki wspartej własną intuicją. Warto też zauważyć, że scenariusze odwołują się do siebie nawzajem, aby nie powtarzać za każdym razem tych samych kroków, rysunków i opisów.

W szczególności powstrzymywałem się w tej książce od przywoływania różnych przypadków i źródeł problemów specyficznych dla konkretnych technologii. Są one po prostu... zbyt specyficzne dla poszczególnych technologii. Gdybym pisał tę książkę 10 lat temu, prawdopodobnie musiałbym wymienić różne scenariusze typowe dla wycieków pamięci w ASP.NET WebForms i WinForms. Kilka lat temu? ASP.NET MVC, WPF, WCF, WF,... Obecnie? ASP.NET Core, EF Core, Azure Functions i co jeszcze? Mam nadzieję, że widać, o co chodzi. Taka wiedza zbyt szybko staje się przestarzała. Książka naszpikowana przykładami wycieków pamięci w WCF mało kogo by dzisiaj interesowała. Jestem wielkim zwolennikiem powiedzenia: „Daj człowiekowi rybę; nakarmisz go na jeden dzień. Naucz człowieka łowić ryby; nakarmisz go na całe życie". Tak więc cała wiedza zawarta w tej książce, wszystkie scenariusze, mają uczyć, jak „łowić ryby". Wszystkie problemy, niezależnie od bazowej, specyficznej technologii, można diagnozować w taki sam sposób, jeśli zastosuje się odpowiednią wiedzę i podejście.

Wszystko to sprawia też, że czytanie tej książki jest dość wymagające, gdyż czasami jest pełna szczegółów i przekazuje dość przytłaczającą ilość informacji. Pomimo tego wszystkiego zachęcam do dogłębnego i powolnego czytania oraz opierania się pokusie pobieżnego przeglądania niektórych fragmentów. Na przykład, aby w pełni czerpać korzyści z tej książki, należy dokładnie zapoznać się z pokazanym kodem i przedstawionymi rysunkami (nie wystarczy tylko rzucić na nie okiem i uznać je za oczywiste).

Żyjemy w świetnych czasach środowiska uruchomieniowego CoreCLR z otwartym kodem źródłowym. Daje nam to dużo większe możliwości zrozumienia działania środowiska CLR. Nie ma tajemnic i zgadywania. Wszystko dostępne jest w kodzie, który można przeczytać i zrozumieć. Moje badania działania środowiska uruchomieniowego

są więc w znacznym stopniu oparte na kodzie GC w CoreCLR (kod ten jest też wykorzystywany przez platformę .NET Framework). Niezliczone dni i tygodnie spędziłem na analizowaniu tej ogromnej ilości dobrej inżynierskiej roboty. Myślę, że jest to świetny kod i sądzę, że wiele osób również chciałoby przebadać słynny plik `gc.cpp`, składający się z dziesiątek tysięcy wierszy kodu. Jest to jednak związane z bardzo stromą krzywą uczenia się. Żeby w tym pomóc, często pozostawiam wskazówki, od czego zacząć badanie kodu CoreCLR w odniesieniu do opisywanych tematów. Zachęcam gorąco do jeszcze głębszego zapoznania się z sugerowanymi przeze mnie fragmentami pliku `gc.cpp`!

Po przeczytaniu tej książki, czytelnik powinien być w stanie:

- Pisać wydajny kod uwzględniający zarządzanie pamięcią w .NET. Choć przedstawione przykłady są w języku C#, to myślę, że nabyta wiedza i poznany zestaw narzędzi pozwolą zastosować je też w językach F# i VB.NET.
- Diagnozować typowe problemy związane z zarządzaniem pamięcią w .NET. Ponieważ większość technik opiera się na danych ETW/LLTng i rozszerzeniu SOS, można je stosować w systemach Windows i Linux (bardziej zaawansowane narzędzia dostępne są w systemie Windows).
- Zrozumieć, jak działa CLR w obszarze zarządzania pamięcią. Poświęciłem sporo uwagi na wyjaśnienie nie tylko tego, jak pewne rzeczy działają, ale też, dlaczego.
- Czytać z pełnym zrozumieniem zgłoszenia wielu ciekawych problemów dotyczących C# i środowiska uruchomieniowego CLR w serwisie GitHub, a nawet samemu brać udział w dyskusji.
- Czytać kod GC w CoreCLR (zwłaszcza w pliku `gc.cpp`) ze zrozumieniem pozwalającym na dalsze badania i dociekania.
- Czytać z pełnym zrozumieniem informacje na temat odśmiecania pamięci i zarządzania pamięcią w różnych środowiskach, takich jak Java, Python lub Go.

Jeśli chodzi o samą zawartość tej książki, to przedstawia się ona następująco. Rozdział 1 jest bardzo ogólnym, teoretycznym wprowadzeniem do zarządzania pamięcią prawie bez jakichkolwiek szczególnych odwołań do .NET. Rozdział 2 podobnie jest ogólnym wprowadzeniem do zarządzania pamięcią na poziomie sprzętu i systemu operacyjnego. Oba rozdziały można traktować jako ważne, ale opcjonalne wprowadzenie. Stanowią one przydatne, szersze spojrzenie na to zagadnienie, które będzie przydatne w pozostałych częściach książki. Choć oczywiście usilnie zachęcam do ich przeczytania, to można je pominąć, jeśli ktoś się spieszy lub jest zainteresowany jedynie najbardziej praktycznymi tematami związanymi z .NET. Nawet zaawansowanych czytelników, którzy sądzą, że dobrze znają zagadnienia z tych dwóch pierwszych rozdziałów, zachęcam do ich przeczytania. Próbowałem w nich zawrzeć nie tylko oczywiste informacje.

Rozdział 3 jest poświęcony wyłącznie pomiarom i różnym narzędziom (niektóre z nich są bardzo często wykorzystywane w dalszych częściach książki). Jest to lektura zawierająca głównie listę narzędzi i opis ich używania. Jeśli kogoś ciekawi głównie teoretyczna część tej książki, może jedynie pobieżnie przejrzeć ten rozdział. Z drugiej strony, jeśli ktoś planuje intensywnie wykorzystywać wiedzę z tej książki w diagnostyce problemów, to prawdopodobnie będzie często wracać do tego rozdziału.

Rozdział 4 jest pierwszym, w którym zaczynamy intensywnie omawiać .NET, ale nadal w dość ogólny sposób, co pozwoli nam zrozumieć pewne istotne wewnętrzne mechanizmy, takie jak system typów .NET (w tym porównanie typów wartościowych i typów referencyjnych, internowanie łańcuchów znaków albo dane statyczne. Jeśli ktoś ma naprawdę mało czasu, może zacząć czytanie od tego rozdziału. Rozdział 5 opisuje pierwsze zagadnienie naprawdę związane z pamięcią – jak pamięć jest zorganizowana w aplikacjach .NET, wprowadzając pojęcie sterty małych obiektów i sterty dużych obiektów, a także segmentów. Rozdział 6 zagłębia się dalej w wewnętrzne mechanizmy związane z pamięcią i jest poświęcony alokowaniu pamięci. Zadziwiające, że całkiem duży rozdział może być poświęcony tak prostemu teoretycznie zagadnieniu. Ważną i dużą częścią tego rozdziału są opisy różnych źródeł alokacji w kontekście ich unikania.

Rozdziały od 7 do 10 są najważniejszymi częściami książki opisującymi, jak działa odśmiecanie pamięci w .NET i zawierają praktyczne przykłady i uwarunkowania wynikające z tej wiedzy. Aby nie przytłoczyć czytelnika nadmiarem informacji naraz, rozdziały te opisują najprostszą odmianę odśmiecania pamięci – tak zwane odśmiecanie niewspółbieżne w trybie stacji roboczej. Z kolei rozdział 11 jest poświęcony opisowi wszystkich pozostałych odmian z wyczerpującymi rozważaniami nad wyborem odpowiedniej z nich. Rozdział 12 kończy część książki poświęconą odśmiecaniu pamięci, opisując trzy ważne mechanizmy: finalizację, obiekty sprzątające po sobie i słabe odwołania.

Trzy ostatnie rozdziały stanowią „zaawansowaną" część książki w tym sensie, że wyjaśniają, jak wszystko działa poza podstawową częścią zarządzania pamięcią w .NET. Rozdział 13 wyjaśnia na przykład zagadnienie wskaźników zarządzanych i zajmuje się głębiej strukturami (w tym niedawno dodanymi strukturami typu ref). Rozdział 14 poświęca wiele uwagi typom i technikom zyskującym ostatnio coraz więcej popularności, takim jak typy `Span<T>` i `Memory<T>`. Część tego rozdziału jest poświęcona niezbyt dobrze znanemu tematowi projektowania zorientowanego na dane, a kilka słów poświęcono też nadchodzącym funkcjom języka C# (takim jak typy referencyjne dopuszczające wartość null i potoki). Rozdział 15 (ostatni) opisuje różne sposoby sterowania odśmiecaniem pamięci i jego monitorowania z poziomu kodu, w tym interfejs API klasy GC, hosting środowiska CLR oraz bibliotekę ClrMD.

Większość programów z tej książki jest dostępna w repozytorium GitHub pod adresem `https://github.com/Apress/pro-.net-memory`. Jest ono zorganizowane według rozdziałów, a większość z nich zawiera dwa rozwiązania: jedno dla przeprowadzanych

testów i jedno dla pozostałych programów. Warto zauważyć, że choć dołączone projekty zawierają listingi programów z książki, to zwykle jest w nich więcej kodu. Jeśli ktoś chce wykorzystać określony program z książki (lub z nim poeksperymentować), najłatwiej po prostu wyszukać numer danego listingu i go wypróbować. Zachęcam też do przejrzenia całych projektów dla określonych tematów, aby lepiej je zrozumieć.

Chciałbym też tu wspomnieć o kilku ważnych konwencjach stosowanych w tej książce. Najważniejszą jest rozróżnienie pomiędzy dwoma głównymi pojęciami wykorzystywanymi w całej książce:

- Odśmiecanie pamięci (GC – garbage collection) – ogólnie rozumiany proces odzyskiwania niepotrzebnej już pamięci.
- Mechanizm odśmiecania pamięci (the GC – the garbage collector) – konkretny mechanizm realizujący odśmiecanie pamięci, zwłaszcza w kontekście mechanizmu odśmiecania pamięci w .NET.

Ta książka stanowi zamkniętą całość i nie odwołuje się do wielu innych materiałów lub książek. Oczywiście jest wiele innych publikacji, z których można czerpać podobną wiedzę i wiele razy musiałbym się odwoływać do różnych źródeł. Zamiast tego pozwolę sobie na wymienienie wybranej przeze mnie listy proponowanych książek i artykułów, stanowiących uzupełniające źródło wiedzy:

- Książka *Pro .NET Performance* – Sasha Goldshtein, Dima Zurbalev i Ido Flatow (Apress, 2012).
- Książka *CLR via C#* – Jeffrey Richter (Microsoft Press, 2012).
- *Writing High-Performance .NET Code* – Ben Watson (Ben Watson, 2014).
- *Advanced .NET Debugging* – Mario Hewardt (Addison-Wesley Professional, 2009).
- *.NET IL Assembler* – Serge Lidin (Microsoft Press, 2012)
- *Shared Source CLI Essentials* – David Stutz (O'Reilly Media, 2003).
- Dokumentacja otwartego kodu środowiska uruchomieniowego .NET: „Book Of The Runtime", opracowywana równolegle z samym środowiskiem uruchomieniowym, dostępna pod adresem `https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md`.

Wiele cennych informacji można też znaleźć w różnych blogach i artykułach dostępnych w Internecie. Zamiast wymieniać je na tych stronach, wskażę na świetne repozytorium `https://github.com/adamsitnik/awesome-dot-net-performance` utrzymywane przez Adama Sitnika.

Rozdział 1

Podstawowe pojęcia

Zacznijmy od prostego, ale bardzo ważnego pytania. Kiedy powinniśmy przejmować się zarządzaniem pamięcią w .NET, skoro jest w całości zautomatyzowane? Czy w ogóle powinniśmy sobie zawracać tym głowę? Jak można by oczekiwać po tym, że napisałem tę książkę – gorąco zachęcam do pamiętania o pamięci w każdej sytuacji programistycznej. Jest to po prostu kwestia naszego profesjonalizmu. Konsekwencja tego, jak wykonujemy swoją pracę. Czy staramy się wykonywać ją jak najlepiej, czy po prostu ją wykonujemy? Jeśli dbamy o jakość swojej pracy, nie powinno wystarczać nam, aby efekty naszej pracy po prostu działały. Powinniśmy się też przejmować tym, jak działają? Czy tworzone oprogramowanie jest optymalne z punktu widzenia wykorzystania procesora i pamięci? Czy jest łatwe w utrzymaniu i testowaniu, otwarte na rozszerzenia, ale zamknięte na modyfikacje? Czy nasz kod spełnia zasady SOLID? Wierzę, że wszystkie te pytania odróżniają początkujących programistów od bardziej zaawansowanych i doświadczonych. Ci pierwsi są głównie zainteresowani wykonaniem zadania i nie przejmują się zbytnio powyższymi pozafunkcjonalnymi aspektami swojej pracy. Ci drudzy są na tyle doświadczeni, że biorą też pod uwagę jakość swojej pracy. Sądzę, że każdy chciałby do tego dążyć. Nie jest to oczywiście sprawa trywialna. Pisanie eleganckiego kodu, pozbawionego błędów i spełniającego wszystkie możliwe wymagania pozafunkcjonalne jest naprawdę trudne.

Czy jednak takie dążenie do doskonałości ma być jedynym wymogiem zdobywania większej wiedzy na temat zarządzania pamięcią w .NET? Uszkodzenia pamięci przejawiające się jako wyjątki `AccessViolationException` są niezwykle rzadkie[1]. Może się też wydawać, że podobnie jest z niekontrolowanym wzrostem wykorzystania pamięci. Czy mamy się więc czym przejmować? Dzięki wyrafinowanej implementacji środowiska

1 Wyjątek `AccessViolationException` albo inne uszkodzenia sterty mogą być często wzbudzane przez automatyczne zarządzanie pamięcią, nie dlatego, że stanowi ono ich przyczynę, ale ponieważ jest najcięższym składnikiem środowiska związanym z pamięcią. Dlatego ma największą możliwość ujawnienia wszelkich niespójnych stanów pamięci.

uruchomieniowego .NET na szczęście nie musimy poświęcać wiele uwagi aspektom pamięciowym. Z drugiej jednak strony, gdy zajmujemy się analizowaniem problemów wydajnościowych dużych aplikacji opartych na .NET, problemy z wykorzystaniem pamięci zawsze znajdują się wysoko na liście. Czy z długoterminowego punktu widzenia problemem jest, gdy mamy wyciek pamięci po wielu dniach ciągłego działania aplikacji? W Internecie można znaleźć zabawny mem dotyczący nienaprawionego wycieku pamięci w oprogramowaniu pewnego pocisku rakietowego, ponieważ pamięć była wystarczająca do czasu, aż pocisk osiągnął swój cel. Czy nasz system jest takim jednorazowym pociskiem? Czy zdajemy sobie sprawę z tego, czy zautomatyzowane zarządzanie pamięcią wprowadza duże obciążenie dla naszej aplikacji, czy nie? Może moglibyśmy wykorzystywać tylko dwa serwery zamiast dziesięciu? Co więcej nawet w czasach bezserwerowego przetwarzania w chmurze nie obędziemy się bez pamięci. Jednym z przykładów mogą być funkcje Azure Functions, za które opłaty wyliczane są w oparciu o jednostkę nazywaną „gigabajtosekundami" (GB-s). Są one obliczane poprzez przemnożenie średniego rozmiaru zajętej pamięci w gigabajtach przez czas w sekundach potrzebny do wykonania danej funkcji. Zużycie pamięci przekłada się więc bezpośrednio na wydawane pieniądze.

W każdym razie zaczynamy sobie zdawać sprawę, że nie mamy pojęcia, gdzie szukać prawdziwej przyczyny problemów i przydatnych pomiarów. W tym miejscu zaczynamy zdawać sobie sprawę, że warto zrozumieć wewnętrzne mechanizmy naszych aplikacji i bazowego środowiska uruchomieniowego.

Aby dogłębnie zrozumieć zarządzanie pamięcią w .NET, najlepiej zacząć od samego początku. Niezależnie od tego, czy ktoś jest nowicjuszem, czy zaawansowanym programistą. Zapraszam do wspólnego przejścia przez teoretyczne wprowadzenie zawarte w tym rozdziale. Da to pewien poziom wiedzy i zrozumienia podstawowych pojęć, na którym będzie bazować pozostała część książki. Aby nie była to tylko nudna teoria, czasami będę odwoływać się do konkretnych technologii. Przyjrzymy się też nieco historii tworzenia oprogramowania. Wiąże się to dobrze z rozwojem pojęć dotyczących zarządzania pamięcią. Zauważymy też dość ciekawe fakty, które, mam nadzieję, okażą się też interesujące dla czytelnika. Poznawanie historii danego zagadnienia jest zawsze najlepszym sposobem na uzyskanie szerszej perspektywy.

Nie ma się jednak czego obawiać. Nie jest to książka historyczna. Nie będę opisywał biografii wszystkich inżynierów zajmujących się rozwijaniem algorytmów odśmiecania pamięci od 1950 roku. Nie będzie też potrzebna wiedza o zamierzchłej historii. Mam jednak nadal nadzieję, że dla wszystkich ciekawe będzie poznanie, jak to zagadnienie się rozwijało i gdzie jest obecnie na linii czasu. Pozwoli nam to też na porównanie podejścia zastosowanego w .NET z wieloma innymi językami i środowiskami uruchomieniowymi.

Terminy związane z pamięcią

Zanim zaczniemy, dobrze będzie się przyjrzeć kilku bardzo ważnym definicjom, bez których trudno sobie wyobrazić omawianie tematu pamięci:

- *Bit* – jest to najmniejsza jednostka informacji wykorzystywana w informatyce. Reprezentuje dwa możliwe stany, odpowiadające zwykle wartościom liczbowym 1 i 0 lub wartościom logicznym prawda i fałsz. W rozdziale 2 krótko wspomnimy, w jaki sposób nowoczesne komputery przechowują pojedyncze bity. W przypadku większej wartości liczbowej potrzebna jest kombinacja wielu bitów do zakodowania jej w postaci liczby binarnej, jak wyjaśniono dalej. Przy określaniu rozmiaru danych bity oznaczane są małą literą b.
- *Liczba binarna* – całkowita wartość liczbowa reprezentowana przez sekwencję bitów. Każdy kolejny bit określa składową sumy danej wartości, będącą kolejną potęgą liczby 2. Na przykład, aby przedstawić liczbę 5, wykorzystujemy trzy kolejne bity o wartościach 1, 0 i 1, ponieważ 1x1 + 0x2 + 1x4 równa się 5. Liczba binarna n-bitowa może reprezentować maksymalną wartość 2^n – 1. Zwykle dodatkowy bit przeznaczony jest na reprezentowanie znaku wartości, aby kodować zarówno liczby dodatnie, jak i ujemne. Istnieją też inne, bardziej złożone sposoby kodowania wartości liczbowych w postaci binarnej, zwłaszcza dla liczb zmiennoprzecinkowych.
- *Kod binarny* – zamiast wartości liczbowych, sekwencja bitów może reprezentować określony zestaw innych danych – takich jak znaki tekstu. Każda sekwencja bitów jest przypisana do określonych danych. Najbardziej podstawowym i przez wiele lat najpopularniejszym kodem był kod ASCII, który wykorzystuje 7-bitowe sekwencje do reprezentowania tekstu i innych znaków. Istnieją też inne ważne kody binarne takie jak *opcodes* (kody operacji), które kodują instrukcje nakazujące komputerowi, co ma robić.
- *Bajt* – historycznie była to sekwencja bitów potrzebna do zakodowania pojedynczego znaku tekstu przy użyciu określonego kodu binarnego. Najbardziej typowym rozmiarem bajtu jest bajt 8-bitowy, choć zależy to od architektury komputera i może się różnić pomiędzy różnymi systemami. Ze względu na tę niejednoznaczność istnieje bardziej precyzyjny termin *oktet*, który oznacza jednostkę danych dokładnie 8-bitowej długości. W każdym razie standardem de facto jest rozumienie bajtu jako wartości o długości 8 bitów i w takiej formie termin ten stał się niekwestionowanym standardem definiowania rozmiarów danych. Obecnie raczej nie spotyka się innych architektur niż standardowa architektura z 8-bitowymi bajtami. Przy określaniu rozmiaru danych bajty oznaczane są wielką literą B.

Określając rozmiar danych, korzystamy z typowych przedrostków oznaczających rząd wielkości. Jest to powodem stałego zamieszania i nieporozumień, które warto w tym momencie wyjaśnić. Powszechnie używane terminy, takie jak kilo, mega i giga oznaczają kolejne potęgi tysiąca. Jeden *kilo* oznacza 1000 (i oznaczamy to w skrócie małą literą k), jeden *mega* oznacza milion (wielka litera M), itd. Z drugiej strony czasami popularne jest podejście wyrażające rzędy wielkości jako kolejne potęgi 1024. W takich przypadkach używamy jako przedrostka *kibi*, co oznacza 1024 (oznaczane w skrócie jako Ki), jeden *mebi* to 1024*1024 (oznaczane w skrócie jako Mi), jeden *gibi* (Gi) to 1024*1024*1024, itd. Wprowadza to powszechnie występującą niejednoznaczność. Gdy ktoś mówi o 1 „gigabajcie", w zależności od kontekstu możemy sądzić, że chodzi o miliard bajtów (1 GB) albo o 1024^3 bajtów (1 GiB). W praktyce niewiele osób przejmuje się dokładnym stosowaniem tych przedrostków. Bardzo powszechne obecnie jest określanie rozmiarów modułów pamięciowych w komputerach jako gigabajty (GB), gdy w rzeczywistości chodzi o gibibajty (GiB) albo odwrotnie w przypadku pojemności dysków twardych. Nawet norma JEDEC Standard 100B.01 „Terms, Definitions, and Letter Symbols for Microcomputers, Microprocessors, and Memory Integrated Circuits" (Terminy, definicje i symbole literowe dla mikrokomputerów, mikroprocesorów i zintegrowanych układów pamięciowych) odwołuje się do powszechnego użycia liter K, M i G do oznaczania potęg 1024. W takich sytuacjach musimy odwoływać się do zdrowego rozsądku, aby odpowiednio interpretować te prefiksy w danym kontekście.

Obecnie jesteśmy bardzo przyzwyczajeni do terminów, takich jak pamięć RAM albo pamięć trwała, które są instalowane w naszych komputerach. Nawet inteligentne zegarki są obecnie wyposażane w 8 GiB pamięci RAM. Łatwo można zapomnieć, że pierwsze komputery nie były wyposażone w takie luksusy. Można by powiedzieć, że w nic nie były wyposażone. Przyjrzenie się krótkiej historii rozwoju komputerów pozwoli nam inaczej spojrzeć na samą pamięć. Zacznijmy od początku.

Powinniśmy mieć na uwadze, że można się spierać, które urządzenie należałoby nazwać „całkiem pierwszym komputerem". Podobnie bardzo trudno jest wskazać jedynego „wynalazcę komputera". Jest to kwestia samej definicji, czym tak naprawdę jest „komputer". Zamiast więc zaczynać niekończące się dyskusje, co było pierwsze i kto był pierwszy, przyjrzyjmy się kilku najstarszym maszynom i temu, co oferowały programistom, choć samo słowo *programista* miało się pojawić wiele lat później. Na początku byli oni zwani *koderami* albo *operatorami*.

Należy podkreślić, że maszyny, które można zdefiniować jako pierwsze komputery, nie były w pełni elektroniczne, ale elektromechaniczne. Z tego powodu były bardzo powolne i pomimo ogromnych rozmiarów oferowały bardzo niewiele. Pierwszy z tych programowalnych elektromechanicznych komputerów został zaprojektowany w Niemczech przez Konrada Zuse i nazywał się komputerem Z3. Ważył jedną tonę! Jedna

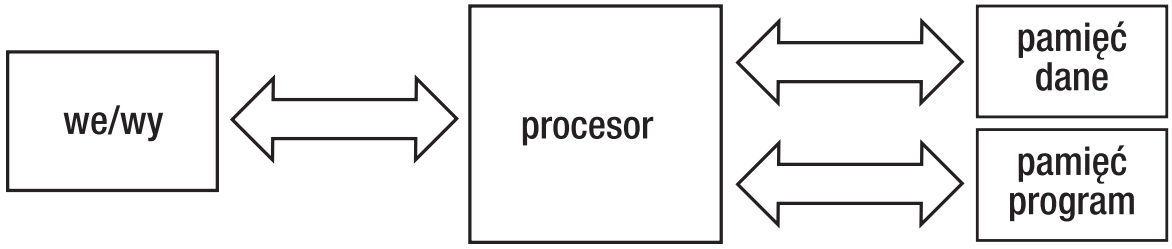
operacja dodawania zabierała około sekundy, a pojedyncze mnożenie zabierało 3 sekundy! Zbudowany był z 2000 elektromechanicznych przekaźników i oferował jednostkę arytmetyczną potrafiącą wykonywać jedynie operacje dodawania, odejmowania, mnożenia, dzielenia i wyciągania pierwiastka kwadratowego. Jednostki arytmetycznie zawierały też dwa 22-bitowe rejestry pamięciowe wykorzystywane w obliczeniach. Komputer ten oferował też 64 komórki pamięci ogólnego przeznaczenia, z których każda miała 22 bity długości. Obecnie moglibyśmy powiedzieć, że oferował 176 bajtów wewnętrznej pamięci do przechowywania danych!

Dane były wprowadzane poprzez specjalną klawiaturę, a program był wczytywany podczas wykonywania obliczeń z perforowanej taśmy celuloidowej. Możliwość przechowywania programu w wewnętrznej pamięci komputera miała zostać zaimplementowana kilka lat później i wkrótce do tego wrócimy, choć Zuse był w pełni świadomy tego pomysłu. W kontekście niniejszej książki ważniejsze jest pytanie o dostęp do pamięci Z3. Programując Z3 mieliśmy do dyspozycji tylko dziewięć instrukcji! Jedna z nich pozwalała nam załadować wartość jednej z 64 komórek pamięci do rejestru pamięciowego jednostki arytmetycznej. Inna zapisywała wartość z powrotem w komórce pamięci. I to wszystko, jeśli chodzi o „zarządzanie pamięcią" w tym pierwszym komputerze. Choć Z3 wyprzedzał swój czas pod wieloma względami, to z powodów politycznych i wybuchu II wojny światowej jego wpływ na rozwój komputerów był pomijalny. Firma Zuse rozwijała swoją linię komputerów przez wiele lat po wojnie, a ostatnia wersja komputera Z22 została zbudowana w roku 1955.

Podczas wojny i wkrótce po niej głównymi centrami rozwoju informatyki były Stany Zjednoczone i Zjednoczone Królestwo. Jednym z pierwszych komputerów zbudowanych w Stanach Zjednoczonych był Harvard Mark I opracowany przez IBM we współpracy z Uniwersytetem Harvarda i znany też jako ASCC (Automatic Sequence Controlled Calculator – kalkulator sterowany automatycznymi sekwencjami). Również był urządzeniem elektromechanicznym, jak wspomniany wcześniej Z3. Miał ogromne rozmiary, mierzył 2,4 m wysokości, 16 m długości i niemal metr głębokości. Ważył przy tym 5 ton! Jest nazywany największą maszyną obliczeniową w historii. Budowano go przez kilka lat, a pierwsze programy uruchomiono na nim pod koniec II wojny światowej w roku 1944. Służył marynarce wojennej, ale też Johnowi von Neumannowi podczas jego prac nad pierwszą bombą atomową w ramach projektu Manhattan. Mimo takich rozmiarów oferował jedynie 72 jednostki pamięciowe dla 23-cyfrowych liczb ze znakiem. Taka jednostka zwana była *akumulatorem* – i była dedykowanym niewielkim fragmentem pamięci, gdzie przechowywane były pośrednie wyniki operacji arytmetycznych i logicznych. Tłumacząc to na dzisiejsze miary, moglibyśmy powiedzieć, że ta 5-tonowa maszyna zapewniała dostęp do 72 komórek pamięci, z których każda miała 78 bitów długości (potrzebujemy 78 bitów do reprezentowania dużych liczb 23-cyfrowych); tak

więc oferowała pamięć o rozmiarze 702 bajtów! Programy były wtedy de facto ciągami obliczeń matematycznych działających na tych 72 komórkach pamięci. Były to języki programowania pierwszej generacji (określane skrótowo 1GL) albo języki maszynowe, gdzie programy były przechowywane na taśmie perforowanej, która była wprowadzana do maszyny na żądanie lub obsługiwano je przy pomocy przełączników na przednim panelu. Mógł wykonywać tylko trzy operacje dodawania lub odejmowania na sekundę. Pojedyncze mnożenie zabierało 20 sekund, a obliczenie funkcji sin(x) zabierało minutę! Tak jak w przypadku Z3, zarządzanie pamięcią w tej maszynie w ogóle nie istniało – można było tylko odczytywać lub zapisywać wartość w jednej z wyżej wspomnianych komórek pamięci.

Dla nas interesujący będzie fakt, że z tego komputera wywodzi się termin *architektura harwardzka* (zobacz rysunek 1-1). Zgodnie z tą architekturą kod programu i przechowywane dane są od siebie fizycznie oddzielone. Dane są przetwarzane przez jakiś rodzaj urządzenia elektronicznego lub elektromechanicznego (takiego jak centralna jednostka przetwarzająca – CPU, czyli procesor). Takie urządzenie często odpowiada też za sterowanie urządzeniami wejścia/wyjścia, takimi jak czytniki kart perforowanych, klawiatury lub urządzenia ekranowe. Choć komputery Z3 albo Mark I wykorzystywały tę architekturę ze względu na jej prostotę, to nie została ona całkiem zapomniana. Jak zobaczymy w rozdziale 2, jest ona wykorzystywana obecnie w niemal każdym komputerze w formie określanej jako *zmodyfikowana architektura harwardzka*. Możemy nawet zobaczyć jej wpływ na programy, które codziennie piszemy.



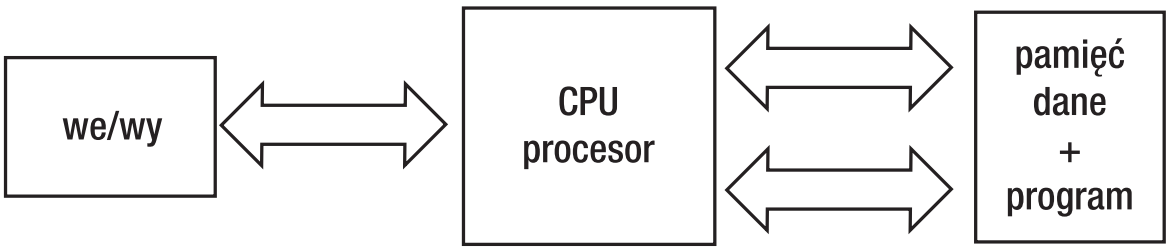
Rysunek 1-1 *Schemat architektury harwardzkiej*

Bardziej znany komputer ENIAC ukończony w 1946 roku był już urządzeniem elektronicznym opartym na lampach próżniowych. Oferował tysiące razy szybsze wykonywanie operacji matematycznych niż Mark I. Jednakże pod względem pamięci nadal wyglądał mało atrakcyjnie. Oferował jedynie 20 10-cyfrowych (ze znakiem) akumulatorów i nie posiadał pamięci wewnętrznej do przechowywania programów. Ze względu na II wojnę światową priorytetem było jak najszybsze budowanie maszyn obliczeniowych dla celów militarnych, a nie budowanie czegoś bardziej złożonego.

Naukowcy, tacy jak Konrad Zuse, Alan Turing i John von Neumann, zajmowali się jednak pomysłami wykorzystania wewnętrznej pamięci komputera do przechowywania

programu razem z jego danymi. Pozwoliłoby to na znacznie łatwiejsze programowanie (a w szczególności przeprogramowywanie) niż kodowanie poprzez karty perforowane lub mechaniczne przełączniki. John von Neumann napisał w 1945 roku bardzo wpływową pracę zatytułowaną „First Draft of a Report on the EDVAC" (Pierwszy szkic raportu dotyczącego maszyny EDVAC), w której opisuje architekturę zwaną *architekturą von Neumanna*. Warto podkreślić, że nie były to wyłącznie pomysły von Neumanna, gdyż inspirował się on też pracami innych naukowców z tego okresu.

Architektura von Neumanna pokazana na rysunku 1-2 jest uproszczoną architekturą harwardzką, w której jest pojedyncza jednostka pamięciowa do przechowywania zarówno danych, jak i programu. Z pewnością przypomina to nam obecne komputery i nie bez powodu. Z bardzo ogólnego punktu widzenia dokładnie w taki sposób są nadal konstruowane nowoczesne komputery, w których architektura von Neumanna i architektura harwardzka spotykają się w postaci zmodyfikowanej architektury harwardzkiej.



Rysunek 1-2 *Schemat architektury von Neumanna*

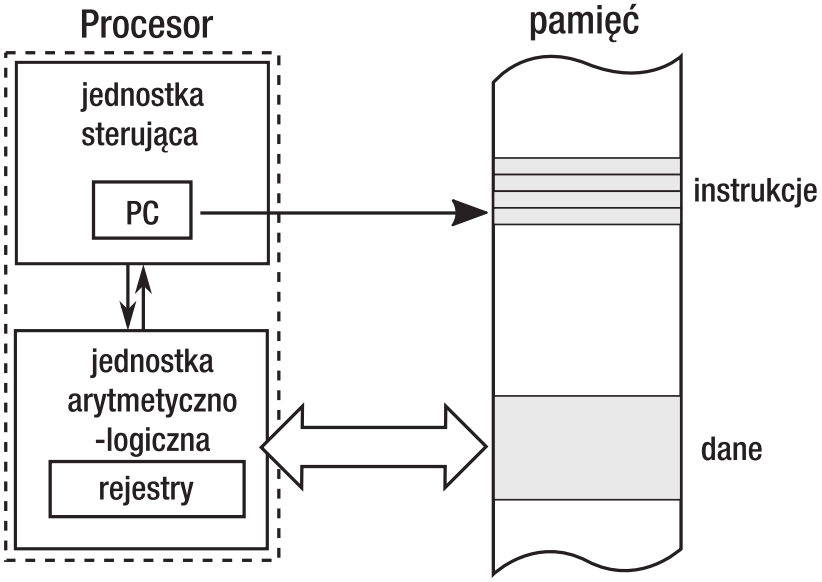
Maszyna Manchester Small-Scale Experimental Machine (SSEM) o przydomku „Baby", zbudowana w 1948 roku oraz powstały w 1949 roku w Cambridge komputer EDSAC były pierwszymi komputerami na świecie, które przechowywały instrukcje programu oraz dane w tym samym obszarze, a więc wcielały w życie architekturę von Neumanna. Komputer „Baby" był dużo bardziej nowoczesny i innowacyjny, ponieważ był pierwszym komputerem wykorzystującym nowy rodzaj pamięci – lampy Williamsa oparte na lampach kineskopowych (CRT). Lampy Williamsa mogą być uważane za pierwszą pamięć o dostępie swobodnym (RAM). Pamięć komputera SSEM składała się z 32 komórek, z których każda miała 32 bity długości. Możemy więc powiedzieć, że pierwszy komputer z pamięcią RAM miał jej całe 128 bajtów! Przeszliśmy więc drogę od 128 bajtów w 1949 roku do typowych 16 gibibajtów w 2018 roku. Lampy Williamsa stały się standardem na przełomie lat 40. i 50., gdy powstało wiele innych komputerów.

Prowadzi nas to do idealnego momentu historycznego, w którym możemy wyjaśnić wszystkie podstawowe pojęcia architektury komputera. Wszystko zestawiono poniżej i pokazano na rysunku 1-3:

- *Pamięć* – jest odpowiedzialna za przechowywanie danych i samego programu. Sposób implementacji pamięci ewoluował znacząco w czasie, poczynając od wspomnianych wcześniej kart perforowanych, przez pamięci magnetyczne i lampy kineskopowe, aż do obecnie wykorzystywanych tranzystorów. Pamięć można podzielić na dwie główne podkategorie:
  - *Pamięć o dostępie swobodnym (RAM – Random Access Memory)* – pozwala nam odczytywać dane z takim samym czasem dostępu, niezależnie od tego, do którego regionu pamięci sięgamy. W praktyce (jak zobaczymy w rozdziale 2) nowoczesna pamięć spełnia ten warunek tylko w przybliżeniu ze względów technologicznych.
  - *Pamięć o dostępie niejednorodnym (non-uniform access memory)* – przeciwieństwo RAM, czas wymagany na dostęp do pamięci zależy od położenia danej komórki na fizycznym nośniku. Dotyczy to oczywiście kart perforowanych, taśm magnetycznych, klasycznych dysków twardych, płyt CD i DVD, itd., gdzie nośnik pamięciowy musi być odpowiednio ustawiony (na przykład obrócony) przed uzyskaniem dostępu do danego fragmentu pamięci.
- *Adres* – reprezentuje określone położenie w całym obszarze pamięci. Jest zwykle wyrażany w formie bajtów, gdyż pojedynczy bajt jest najmniejszym możliwym do zaadresowania elementem na wielu platformach.
- *Jednostka arytmetyczno-logiczna* (*ALU – arithmetic and logic unit*) – odpowiada za wykonywanie operacji, takich jak dodawanie i odejmowanie. Jest to serce komputera, w którym odbywa się większość pracy. Obecnie komputery zawierają więcej niż jedną jednostkę arytmetyczno-logiczną, co pozwala na przeprowadzanie obliczeń równoległych.
- *Jednostka sterująca* – dekoduje instrukcje programu (*opcodes* – kody operacji) wczytywane z pamięci. W oparciu o wewnętrzny opis instrukcji wie, którą operację arytmetyczną lub logiczną należy wykonać i na których danych.
- *Rejestr* – miejsce w pamięci szybko dostępne dla jednostki arytmetyczno-logicznej i/lub jednostki sterującej (do których możemy odwoływać się zbiorczo jako do *jednostek wykonawczych*) i zwykle w niej zawarte. Wspomniane wcześniej akumulatory są specjalnymi, uproszczonymi typami rejestrów. Rejestry są niezwykle szybkie, jeśli chodzi o czas dostępu i właściwie żadne miejsce przechowywania danych nie jest bliższe jednostkom wykonawczym niż one.
- *Słowo maszynowe* – podstawowa jednostka danych o określonym rozmiarze, wykorzystywana przez dany projekt komputera. Widać jej odzwierciedlenie w wielu obszarach projektowych, takich jak rozmiar większości rejestrów, maksymalny adres albo największy blok danych przekazywany w pojedynczej operacji.

Najczęściej jest wyrażane w liczbie bitów (mówimy wtedy o *rozmiarze* lub *długości słowa maszynowego*). Większość dzisiejszych komputerów jest 32-bitowa lub 64-bitowa, więc w tym przypadku długość słowa maszynowego wynosi odpowiednio 32 bity lub 64 bity, rejestry są 32-bitowe lub 64-bitowe, itd.

Architektura von Neumanna ucieleśniona w maszynach SSEM lub EDSAC prowadzi do określenia *komputery z przechowywanymi programami*, co obecnie jest oczywiste, ale nie było takie w początkach ery komputerów. W takim założeniu kod programu, który ma być wykonywany, jest przechowywany w pamięci, więc może być dostępny, tak jak normalne dane – włączając w to takie przydatne operacje, jak modyfikowanie go i nadpisywanie nowym kodem programu.



Rysunek 1-3 *Schemat komputera z przechowywanym programem – pamięć + wskaźnik instrukcji*

Jednostka sterująca przechowuje dodatkowy rejestr zwany *wskaźnikiem instrukcji (IP – instruction pointer)* albo *licznikiem programu (PC – program counter)*, który wskazuje na aktualnie wykonywaną instrukcję. Normalne wykonywanie programu polega na zwiększaniu adresu przechowywanego w liczniku programu, aby przechodzić do kolejnych instrukcji. Operacje, takie jak pętle lub skoki, wymagają po prostu zmiany wartości wskaźnika instrukcji na inny adres, określając, gdzie chcemy przenieść wykonywanie programu.

Pierwsze komputery były programowane przy użyciu kodu binarnego, który bezpośrednio określał wykonywane instrukcje. Jednakże wraz ze zwiększającą się złożonością programów to rozwiązanie stawało się coraz bardziej uciążliwe. Nowy język programowania (określany jako druga generacja języków programowania – 2GL) został

zaprojektowany, tak aby opisywać kod w bardziej przystępny sposób przy pomocy tak zwanego *kodu asemblera (assembly code)*. Jest to tekstowy i bardzo zwarty opis poszczególnych instrukcji wykonywanych przez procesor. Było to znacznie wygodniejsze od bezpośredniego kodowania binarnego. Później zaprojektowano języki jeszcze wyższego poziomu (3GL), takie jak powszechnie znane języki C, C ++ lub Pascal.

Co ciekawe, wszystkie te języki muszą być przekształcone z formy tekstowej na binarną, a następnie umieszczone w pamięci komputera. Proces takiej transformacji jest nazywany *kompilacją*, a narzędzie, które ten proces przeprowadza nazywane jest *kompilatorem*. W przypadku kodu asemblera proces ten nazywamy *asemblacją* wykonywaną przez narzędzie zwane *asemblerem*. Końcowym wynikiem jest program w formie kodu binarnego, który można później wykonywać – ma on postać sekwencji kodów operacji oraz ich argumentów (operandów).

Wyposażeni w podstawową wiedzę możemy teraz zacząć swoją podróż po temacie zarządzania pamięcią.

Alokacja statyczna

Większość wczesnych języków programowania pozwalała jedynie na *statyczną alokację pamięci* – ilość i dokładne położenie pamięci musiały być znane podczas kompilacji, jeszcze przed wykonaniem programu. Przy stałych i predefiniowanych rozmiarach zarządzanie pamięcią było trywialne. Wszystkie „starodawne" języki programowania, począwszy od kodu maszynowego czy kodu asemblera, aż po pierwsze wersje języków FORTRAN i ALGOL, miały takie ograniczone możliwości. Mają też one jednak wiele minusów. Statyczne alokacje pamięci mogą łatwo prowadzić do niewydajnego wykorzystania pamięci – jeśli nie wiemy z góry, ile danych będzie przetwarzane, jak możemy ustalić, ile pamięci powinniśmy przydzielić? To sprawia, że programy są ograniczone i mało elastyczne. Ogólnie rzecz biorąc, taki program powinien zostać ponownie skompilowany, aby mógł przetwarzać większe ilości danych.

W najwcześniejszych komputerach wszystkie alokacje były statyczne, ponieważ wykorzystane komórki pamięci (akumulatory, rejestry albo komórki pamięci RAM) były ustalane podczas kodowania programu. Zdefiniowane „zmienne" istniały więc przez cały czas życia programu. Obecnie nadal wykorzystujemy alokację statyczną w takim sensie, gdy tworzymy globalne zmienne statyczne (i tym podobne), przechowywane w specjalnym segmencie danych programu. W późniejszych rozdziałach zobaczymy, gdzie są one przechowywane w przypadku programów .NET.

Maszyna rejestrowa

Jak dotąd widzieliśmy przykłady maszyn, które wykorzystywały rejestry (lub akumulatory w konkretnym przypadku) przy wykonywaniu operacji na jednostkach arytmetyczno-logicznych (ALU). Maszyna opierająca się na takich założeniach jest nazywana *maszyną rejestrową*. Podczas wykonywania programów na takim komputerze w istocie wykonujemy obliczenia na rejestrach. Jeśli chcemy wykonać operację dodawania, dzielenia lub jakąś inną, musimy załadować właściwe dane z pamięci do odpowiednich rejestrów. Następnie wywołujemy konkretne instrukcje, aby przeprowadzić na nich odpowiednie operacje oraz inną instrukcję, aby przepisać wynik z jednego z rejestrów do pamięci.

Załóżmy, że chcemy napisać program, który oblicza wyrażenie `s=x+(2*y)+z` w komputerze z dwoma rejestrami o nazwach `A` i `B`. Załóżmy też, że `s`, `x`, `y` oraz `z` są adresami w pamięci, pod którymi przechowywane są pewne wartości. Zakładamy też, że mamy jakiś niskopoziomowy kod pseudo-asemblera z instrukcjami typu Load (załaduj), Add (dodaj) i Multiply (pomnóż). Taką teoretyczną maszynę można zaprogramować przy pomocy następującego prostego programu (zobacz listing 1-1).

Listing 1-1 *Pseudokod przykładowego programu wykonującego obliczenie* `s=x+(2*y)+z` *na prostej maszynie rejestrowej z dwoma rejestrami. Komentarze pokazują stan rejestru po wykonaniu każdej instrukcji.*

```
Load      A, y        // A = y
Multiply  A, 2        // A = A * 2 = 2 * y
Load      B, x        // B = x
Add       A, B        // A = A + B = x + 2 * y
Load      B, z        // B = z
Add       A, B        // A = A + B = x + 2 * y + z
Store     s, A        // s = A
```

Jeśli ten kod przypomina komuś x86 lub inny znany kod asemblera – nie jest to przypadek. Wynika to z faktu, że prawie każdy nowoczesny komputer jest pewnego rodzaju złożoną maszyną rejestrową. Wszystkie procesory firm Intel i AMD, które wykorzystujemy w swoich komputerach, działają w taki sposób. Przy pisaniu kodu asemblera bazującego na x86/x64 operujemy na rejestrach ogólnego zastosowania, takich jak `eax`, `ebx`, `ecx`, itd. Oczywiście istnieje dużo więcej instrukcji, dodatkowe, wyspecjalizowane rejestry, itd. Ale ogólna zasada jest taka sama.

Uwaga Czy można by sobie wyobrazić maszynę z zestawem instrukcji, które pozwalają nam przeprowadzać operacje bezpośrednio w pamięci, bez konieczności ładowania danych do rejestrów? Kod w naszym języku pseudo asemblera mógłby wyglądać na bardziej zwarty i byłby zbliżony do języków wysokiego poziomu, ponieważ nie byłoby dodatkowych instrukcji ładujących dane z pamięci do rejestrów i zapisujących wartości z rejestrów w pamięci:

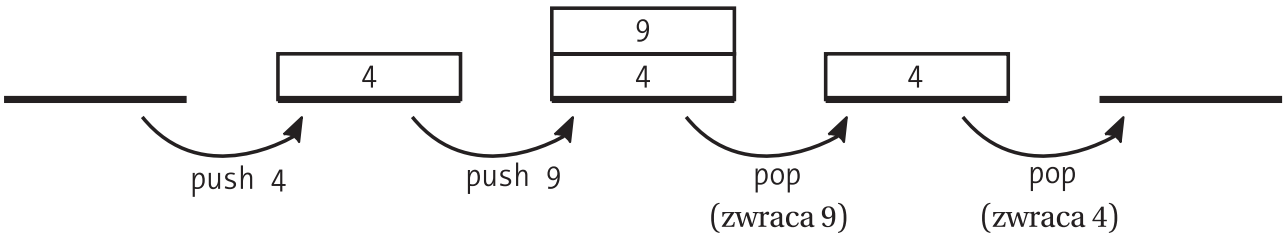
```
Multiply        s, y, 2     // s = 2 * y
Add             s, x        // s = s + x = 2 * y + x
Add             s, z        // s = s + z = 2 * y + x + z
```

Tak, były takie maszyny, np. IBM System/360, ale obecnie nie kojarzę żadnego powszechnie wykorzystywanego komputera tego rodzaju.

Stos

Koncepcyjnie stos jest strukturą danych, którą można po prostu opisać jako listę LIFO – „last in, first out" (ostatnie na wejściu, pierwsze na wyjściu). Pozwala na dwie główne operacje: odłożenie jakichś danych na wierzch stosu („push") i zdjęcie jakichś danych z wierzchu stosu („pop"), co zilustrowano na rysunku 1-4.

Stos od samego początku stał się nieodłącznie związany z programowaniem komputerów, głównie ze względu na pojęcie podprogramu. Obecnie .NET intensywnie wykorzystuje pojęcia „stosu wywołań" (call stack) i „stosu" (stack), więc przyjrzyjmy się, od czego to wszystko się zaczęło. Pierwotne znaczenie stosu jako struktury danych nadal obowiązuje (na przykład mamy w .NET dostępny typ kolekcji `Stack<T>`), ale zobaczmy, jak to znaczenie ewoluowało w odniesieniu do organizacji pamięci komputera.



Rysunek 1-4 *Operacje pop i push na stosie. Jest to tylko rysunek koncepcyjny, niepowiązany z żadnym konkretnym modelem pamięci czy implementacją.*

Najwcześniejsze komputery, o których mówiliśmy wcześniej, pozwalały jedynie na sekwencyjne wykonywanie programu, wczytując każdą instrukcję jedną po drugiej z karty perforowanej lub taśmy. Ale pomysł, aby pisać niektóre części programów

(*podprogramy*), tak aby można było z nich korzystać z różnych punktów całego programu, był bardzo kuszący. Możliwość wywoływania różnych części programu wymagała oczywiście, aby kod można było adresować, gdyż w jakiś sposób musimy wskazywać, jaką inną część programu chcemy wywołać. Pierwotne podejście było stosowane przez słyną Grace Hopper w systemie A-0 – zwanym pierwszym kompilatorem. Zakodowała ona zestaw różnych programów na taśmie, nadając każdemu kolejną liczbę, umożliwiającą komputerowi jego znalezienie. Właściwy „program" składał się z sekwencji liczb (indeksów podprogramów) i odpowiadających parametrów. Choć system ten faktycznie wywoływał podprogramy, był oczywiście w dużym stopniu ograniczony. Program mógł wywoływać podprogramy tylko kolejno po sobie i nie były możliwe zagnieżdżone wywołania.

Zagnieżdżone wywołania wymagają nieco bardziej złożonego podejścia, ponieważ komputery muszą jakoś pamiętać, gdzie mają kontynuować wykonywanie programu (gdzie wrócić) po zakończeniu wykonywania określonego podprogramu. Przechowywanie adresu powrotu w jednym z akumulatorów było pierwszym podejściem wynalezionym przez Davida Wheelera na maszynie EDSAC (metoda ta jest nazywana *„skokiem Wheelera"*). W jego uproszczonym podejściu nie były możliwe wywołania *rekurencyjne*, co oznacza wywoływanie podprogramu z poziomu jego samego.

Pierwsza wzmianka o pojęciu stosu w takim znaczeniu, w jakim znamy je dzisiaj w kontekście architektury komputerów, pojawiła się prawdopodobnie w raporcie Alana Turinga opisującym ACE (Automatic Computing Engine – automatyczny silnik obliczeniowy) z wczesnych lat 40. XX wieku. Opisywał on koncepcję maszyny typu von Neumanna, która była faktycznie komputerem z programami przechowywanymi w pamięci. Poza wieloma innymi szczegółami implementacyjnymi opisywał on dwie instrukcje – `BURY` (zakop) i `UNBURY` (odkop) – działające na głównej pamięci i akumulatorach:

- Przy wywoływaniu podprogramu (`BURY`) adres aktualnie wykonywanej instrukcji zwiększony o 1 (żeby wskazywać na następną instrukcję, do której należy wrócić z podprogramu) był zapisywany w pamięci. Inna, tymczasowo zachowywana wartość, służąca jako wskaźnik stosu, była zwiększana o 1.
- Przy powrocie z podprogramu (`UNBURY`) podejmowane było działanie odwrotne.

Stanowiło to pierwszą implementację stosu w sensie zorganizowanego w formie LIFO miejsca na adresy powrotów z podprogramów. Jest to rozwiązanie, które nadal jest używane w nowoczesnych komputerach i choć oczywiście znacząco ewoluowało od tego czasu, to podstawy są nadal takie same.

Stos jest bardzo ważnym aspektem zarządzania pamięcią, ponieważ przy programowaniu w .NET wiele naszych danych może być tam umieszczanych. Przyjrzymy się bliżej działaniu

stosu i jego wykorzystaniu w wywołaniach funkcji. Wykorzystamy przykładowy program z listingu 1-2 napisany w pseudokodzie podobnym do języka C, w którym wywołujemy dwie funkcje – najpierw `main` wywołuje funkcję `fun1` (przekazując dwa argumenty `a` i `b`), która ma dwie zmienne lokalne `x` i `y`. Następnie funkcja `fun1` wywołuje w pewnym momencie funkcję `fun2` (przekazując pojedynczy argument `n`), która ma pojedynczą zmienną lokalną `z`.

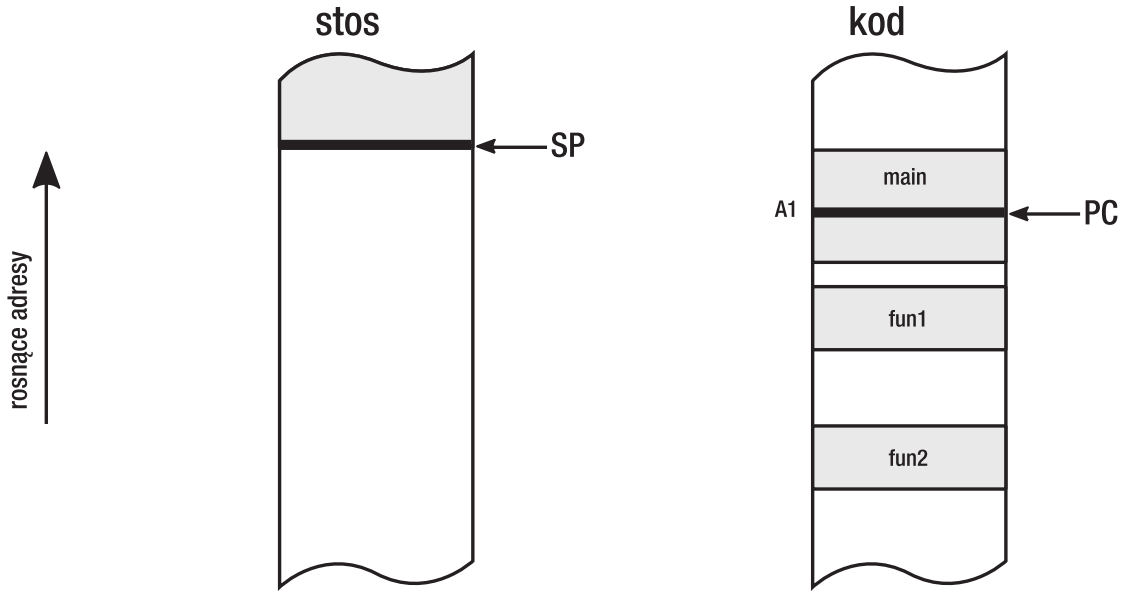
Listing 1-2 *Pseudokod programu wywołującego funkcję wewnątrz innej funkcji*

```
void main()
{
   ...
   fun1(2, 3);
   ...
}

int fun1(int a, int b)
{
   int x, y;
   ...
   fun2(a+b);
}

int fun2(int n)
{
   int z;
   ...
}
```

Najpierw wyobraźmy sobie ciągły obszar pamięci przeznaczony na obsługę stosu, narysowany w taki sposób, że adresy kolejnych komórek pamięci rosną zgodnie ze strzałką w górę (lewa część na rysunku 1-5a) oraz drugi region pamięci, gdzie znajduje się nasz kod (prawa część rysunku 1-5a), zorganizowany w taki sam sposób. Ponieważ kod funkcji nie musi leżeć obok siebie, bloki kodu dla funkcji `main`, `fun1` i `fun2` zostały narysowane oddzielone od siebie. Wykonywanie programu z listingu 1-2 można opisać w następujących krokach:

1. Tuż przed wywołaniem `fun1` wewnątrz funkcji `main` (zobacz rysunek 1-5a). Ponieważ program już działa, jakiś region na stosie został już utworzony (szara część regionu stosu na rysunku 1-5a). Wskaźnik stosu (SP) przechowuje adres wskazujący na aktualną granicę stosu. Licznik programu (PC) wskazuje na jakieś miejsce

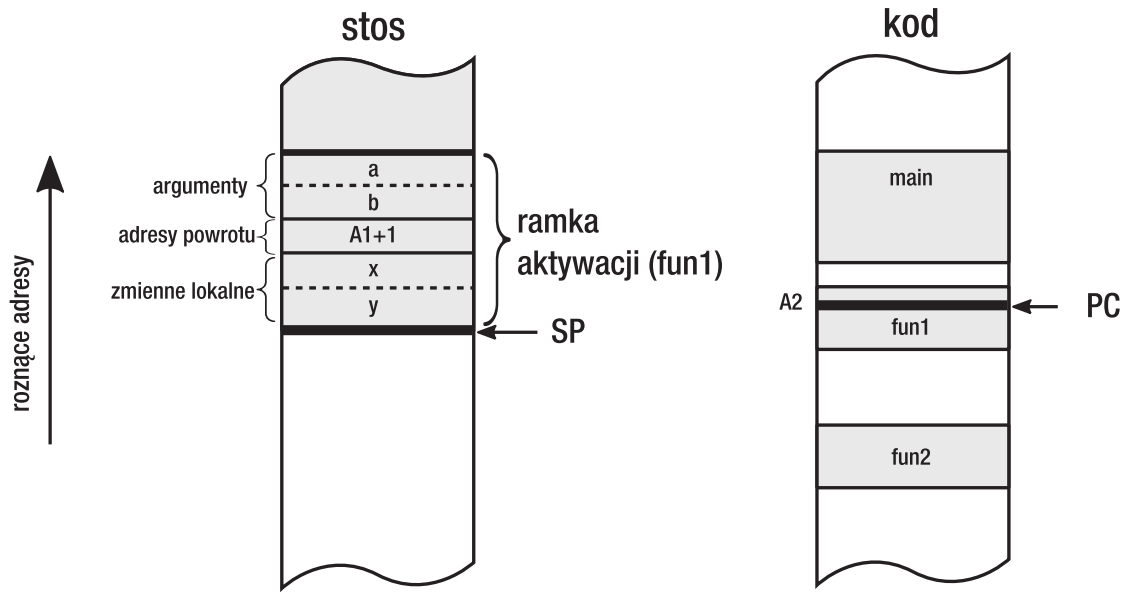
wewnątrz funkcji `main` (oznaczyliśmy to jako adres `A1`) tuż przed instrukcją wywołującą funkcję `fun1`.



Rysunek 1-5a *Regiony pamięci dla stosu i dla kodu – w momencie tuż przed wywołaniem funkcji* `fun1` *z listingu 1-2*

2. Po wywołaniu funkcji `fun1` wewnątrz `main` (zobacz rysunek 1-5b). Gdy funkcja jest wywoływana, stos jest rozszerzany przez przesunięcie wskaźnika SP, tak aby stos zawierał potrzebne informacje. Ten dodatkowy obszar obejmuje:
   - Argumenty – wszystkie argumenty funkcji mogą być zapisane na stosie. W naszym przykładzie zostały tu zapisane argumenty `a` oraz `b`.
   - Adres powrotu – aby mieć możliwość dalszego wykonywania funkcji `main` po wykonaniu wywołania funkcji `fun1`, adres następnej instrukcji tuż po wywołaniu funkcji jest zapisywany na stosie. W naszym przypadku oznaczamy go jako adres `A1+1` (wskazujący na następną instrukcję po instrukcji pod adresem `A1`).
   - Zmienne lokalne – miejsce na wszystkie zmienne lokalne, które mogą być również zapisane na stosie. W naszym przykładzie zostały tu zapisane zmienne `x` oraz `y`.

Taka struktura umieszczana na stosie, gdy wywoływany jest podprogram, jest nazywana *ramką aktywacji (activation frame)*. W typowej implementacji wskaźnik stosu jest zmniejszany o odpowiednie przesunięcie, aby wskazywał na miejsce, gdzie może zacząć się nowa ramka aktywacji. Dlatego często mówi się, że stos rośnie w dół.



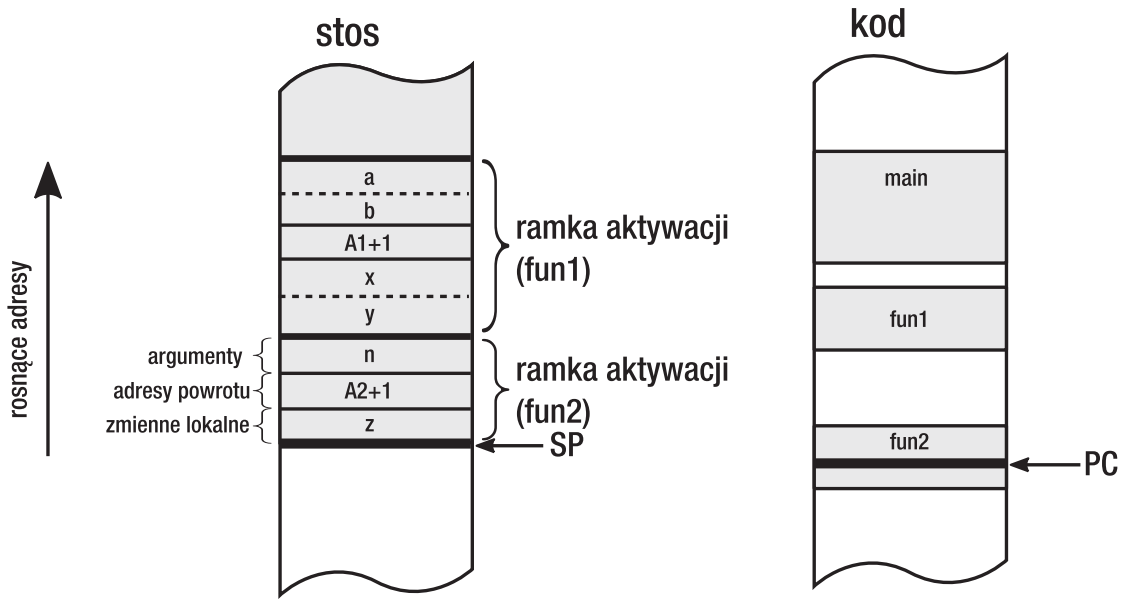
Rysunek 1-5b *Regiony pamięci dla stosu i dla kodu – w chwili po wywołaniu funkcji* `fun1` *z listingu 1-2*

3. Po wywołaniu funkcji `fun2` wewnątrz funkcji `fun1` (zobacz rysunek 1-5c). Powtarza się ten sam wzorzec tworzenia nowej ramki aktywacji. Tym razem zawiera ona region pamięci dla argumentu `n`, adresu powrotu `A2+1` oraz zmiennej lokalnej `z`.

Ramka aktywacji jest też nazywana bardziej ogólnie *ramką stosu (stack frame)*, co oznacza dowolne ustrukturyzowane dane zapisywane na stosie w określonym celu.

Jak widać, kolejne zagnieżdżone wywołania podprogramów po prostu powtarzają ten wzorzec, dodając pojedynczą ramkę aktywacji dla każdego wywołania. Im bardziej zagnieżdżone wywołania podprogramów, tym więcej ramek aktywacji będzie na stosie. Oczywiście sprawia to, że nieskończone zagnieżdżone wywołania podprogramów nie są możliwe, gdyż wymagałyby pamięci na nieskończoną liczbę ramek aktywacji[2]. To jest właśnie przypadek wyjątku `StackOverflowException`, z którym pewnie każdy programista kiedyś się spotkał. Oznacza on, iż wywołano tak dużo zagnieżdżonych podprogramów, że limit pamięci przeznaczonej na stos się wyczerpał.

2 Jest jeden ciekawy wyjątek zwany rekurencją ogonową (prawostronną), którego tutaj nie opisujemy dla zachowania zwięzłości.



Rysunek 1-5c *Regiony pamięci dla stosu i dla kodu – w chwili po wywołaniu funkcji* `fun2` *z funkcji* `fun1`

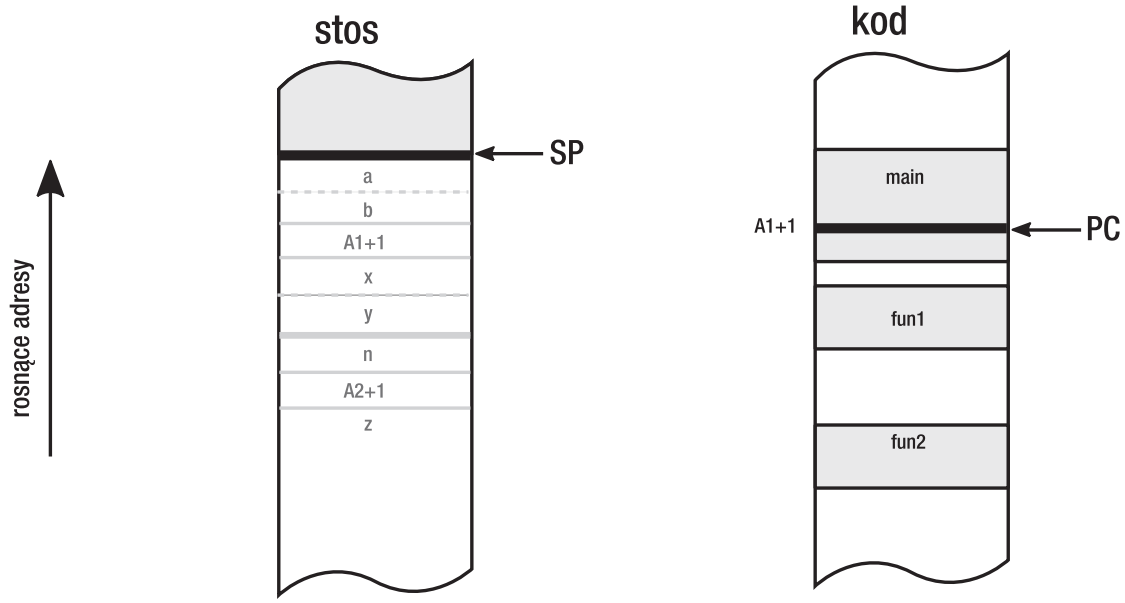
Jak widać, kolejne zagnieżdżone wywołania podprogramów po prostu powtarzają ten wzorzec, dodając pojedynczą ramkę aktywacji dla każdego wywołania. Im bardziej zagnieżdżone wywołania podprogramów, tym więcej ramek aktywacji będzie na stosie. Oczywiście sprawia to, że nieskończone zagnieżdżone wywołania podprogramów nie są możliwe, gdyż wymagałyby pamięci na nieskończoną liczbę ramek aktywacji[3]. To jest właśnie przypadek wyjątku `StackOverflowException`, z którym pewnie każdy programista kiedyś się spotkał. Oznacza on, iż wywołano tak dużo zagnieżdżonych podprogramów, że limit pamięci przeznaczonej na stos się wyczerpał.

Trzeba mieć na uwadze, że mechanizm przedstawiony tutaj jest tylko przykładowy i bardzo uogólniony. Faktyczne implementacje mogą się różnić pomiędzy różnymi architekturami i systemami operacyjnymi. W późniejszych rozdziałach przyjrzymy się dokładniej, jak ramki aktywacji i stos są wykorzystywane przez .NET.

Gdy wykonywanie podprogramu się kończy, jego ramka aktywacji jest odrzucana przez zwiększenie wskaźnika stosu o rozmiar aktualnej ramki aktywacji, natomiast zapisany adres powrotu jest wykorzystywany do ustawienia licznika programu (PC), żeby wykonywanie programu było kontynuowane w funkcji wywołującej. Innymi słowy, to, co było wewnątrz ramki stosu (zmienne lokalne, parametry), nie jest już potrzebne, więc zwiększenie wskaźnika stosu wystarcza do „zwolnienia" pamięci wykorzystywanej

3 Jest jeden ciekawy wyjątek zwany rekurencją ogonową (prawostronną), którego tutaj nie opisujemy dla zachowania zwięzłości.

przez tę ramkę. Dane te zostaną po prostu nadpisane przy następnym użyciu stosu (zobacz rysunek 1-6).



Rysunek 1-6 *Regiony pamięci dla stosu i dla kodu – po powrocie z funkcji* `fun1` *obie wcześniejsze ramki aktywacji są odrzucane*

Jeśli chodzi o implementację, to zarówno wskaźnik SP, jak i licznik PC są zwykle przechowywane w dedykowanych rejestrach. W tym momencie sam rozmiar tego adresu, określone obszary pamięci i rejestry nie są szczególnie istotne.

Stos w nowoczesnych komputerach jest obsługiwany zarówno przez sprzęt (poprzez zapewnianie dedykowanych rejestrów dla wskaźników stosu) oraz przez oprogramowanie (zapewnianą przez system operacyjny abstrakcję wątku i część pamięci wydzieloną jako stos).

Warto zauważyć, że można sobie wyobrazić wiele różnych implementacji stosu z punktu widzenia architektury sprzętowej. Stos może być przechowywany w dedykowanym bloku pamięci wewnątrz CPU albo na dedykowanym układzie scalonym. Może też wykorzystywać ogólną pamięć komputera. Tak jest dokładnie w przypadku najnowszych architektur, gdzie stos jest po prostu regionem o określonym rozmiarze w pamięci procesu. Mogą też istnieć implementacje z architekturą o wielu stosach. W takim przykładowym przypadku stos dla adresów powrotów z podprogramów mógłby być oddzielony od stosu z danymi – parametrami i zmiennymi lokalnymi. Może to być korzystne z powodów wydajnościowych, ponieważ pozwala na jednoczesny dostęp do dwóch oddzielnych stosów. Pozwala to na dodatkowe dostrajanie przetwarzania potokowego przez procesor i innych mechanizmów niskiego poziomu. W każdym razie w obecnych komputerach osobistych stos jest po prostu częścią głównej pamięci.

FORTRAN może być uważany za pierwszy szeroko używany wysokopoziomowy język programowania ogólnego przeznaczenia. W roku 1954, gdy był definiowany, możliwa była tylko alokacja statyczna. Wszystkie tablice musiały mieć rozmiary definiowane w czasie kompilacji a wszystkie alokacje były oparte na stosie. ALGOL był kolejnym bardzo ważnym językiem, który w większym lub mniejszym stopniu stanowił bezpośrednią inspirację dla mnóstwa innych języków (takich jak C/C++, Pascal, Basic, a poprzez języki Simula i Smalltalk – wszystkie nowoczesne języki zorientowane obiektowo, takie jak Python czy Ruby). ALGOL 60 miał jedynie alokacje na stosie – wraz z tablicami dynamicznymi (o rozmiarach określanych przez zmienne). Alan Perlis, wybitny członek zespołu, który stworzył język ALGOL, powiedział:

> *Algol 60 nie byłby w stanie odpowiednio działać bez pojęcia stosów. Choć mieliśmy stosy już wcześniej, dopiero w języku Algol 60 stosy zajęły centralne miejsce w projektowaniu procesorów.*

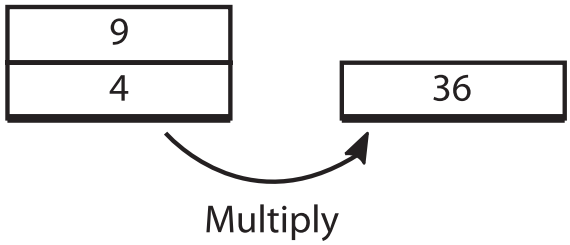
Podczas gdy rodzina języków ALGOL i FORTRAN była wykorzystywana głównie przez naukowców, to pojawił się inny nurt rozwijający języki programowania zorientowane biznesowo, począwszy od „A-0", FLOW-MATIC, poprzez COMTRANS do szerzej znanego języka COBOL (Common Business Language). We wszystkich brakowało bezpośredniego zarządzania pamięcią i operowały głównie na prymitywnych typach danych, takich jak liczby i łańcuchy znaków.

Maszyna stosowa

Zanim przejdziemy do innych pojęć związanych z pamięcią, pozostańmy przez chwilę w kontekście związanym ze stosem i omówmy tak zwane *maszyny stosowe*. W przeciwieństwie do maszyny rejestrowej, w maszynie stosowej wszystkie instrukcje są wykonywane na dedykowanym *stosie wyrażeń (expression stack)* albo *stosie obliczeń (evaluation stack)*. Trzeba mieć na uwadze, że ten stos nie musi być tym samym stosem, o którym mówiliśmy wcześniej. Taka maszyna mogłaby więc mieć zarówno dodatkowy „*stos wyrażeń*", jak i stos ogólnego zastosowania. Poza stosem może w ogóle nie być rejestrów. W takiej maszynie instrukcje domyślnie pobierają argumenty z wierzchu stosu wyrażeń – tyle, ile ich potrzebują. Wyniki również są zapisywane na wierzchu stosu. W takich przypadkach mamy do czynienia z *czystymi maszynami stosowymi* w przeciwieństwie do implementacji, w których operacje mogą mieć dostęp nie tylko do wartości z wierzchu stosu, ale mogą też sięgać głębiej.

Jak dokładnie wygląda operacja na stosie wyrażeń? Na przykład hipotetyczna instrukcja Multiply (pomnóż), wykonywana bez żadnych argumentów, pobierze dwie wartości

z wierzchu stosu wyrażeń, przemnoży je przez siebie i odłoży wynik na wierzch stosu wyrażeń (zobacz rysunek 1-7).



Rysunek 1-7 *Hipotetyczna instrukcja Multiply na maszynie stosowej – ściąga dwa elementy ze stosu i odkłada na stos wynik ich mnożenia przez siebie*

Wróćmy do przykładowego wyrażenia `s=x+(2*y)+z` znanego już z omówienia maszyny rejestrowej i przepiszmy go w stylu maszyny stosowej (zobacz listing 1-3).

Listing 1-3 *Pseudokod prostej maszyny stosowej wykonującej obliczenie* `s=x+(2*y)+z`. *Komentarze pokazują stan stosu wyrażeń.*

```
                  // pusty stos
Push 2            // [2] - pojedynczy element na stosie - wartość 2
Push y            // [2][y] - dwa elementy na stosie - wartość 2 oraz y
Multiply          // [2*y]
Push x            // [2*y][x]
Add               // [2*y+x]
Push z            // [2*y+x][z]
Add               // [2*y+x+z]
Pop l             // [] (z efektem ubocznym zapisania wartości pod l)
```

Ten pomysł prowadzi do bardzo przejrzystego i zrozumiałego kodu. Główne zalety można opisać następująco:

- Nie ma problemu z tym, jak i gdzie zapisywać wartości tymczasowe – czy miałyby to być rejestry, stos, czy pamięć główna. Koncepcyjnie jest to łatwiejsze niż próba optymalnego zarządzania wszystkimi tymi miejscami docelowymi. Upraszcza to implementację.
- Kody operacji mogą być krótsze w sensie wymaganej pamięci, gdyż istnieje wiele instrukcji bez operandów lub z pojedynczymi operandami. Pozwala to na wydajne binarne kodowanie instrukcji i daje bardziej zagęszczony kod binarny. Nawet

mimo tego, że liczba instrukcji może być większa niż w podejściu opartym na rejestrach ze względu na większą liczbę operacji load/store, to nadal jest to korzystne.

Było to ważną zaletą we wczesnych czasach komputerów, gdy pamięć była bardzo droga, a jej ilość była ograniczona. Może to być też korzystne obecnie w przypadku kodu pobieranego na smartfony lub w aplikacjach WWW. Gęste kodowanie binarne instrukcji oznacza też lepsze wykorzystanie pamięci podręcznej procesora.

Pomimo swoich zalet, założenia maszyny stosowej były rzadko implementowane w samym sprzęcie. Jednym z ważnych wyjątków były maszyny firmy Burroughs, takie jak B5000, które zawierały sprzętową implementację stosu. Obecnie chyba nie ma szeroko wykorzystywanej maszyny, która mogłaby być określona jako maszyna stosowa. Pewnym wyjątkiem jest jednostka zmiennoprzecinkowa x87 (wewnątrz procesorów kompatybilnych z x86), która była zaprojektowana jako maszyna stosowa, a ze względu na kompatybilność wsteczną nadal jest programowana w ten sposób nawet dzisiaj.

Dlaczego więc w ogóle wspominamy tego rodzaju maszyny? Ponieważ taka architektura jest świetnym sposobem projektowania niezależnych od platformy maszyn wirtualnych albo silników wykonawczych. Wirtualna maszyna Java firmy Sun oraz środowisko uruchomieniowe .NET są świetnymi przykładami maszyn stosowych. Są one wykonywane na dobrze znanych maszynach rejestrowych z architekturą x86 lub ARM, ale nie zmienia to faktu, że działają zgodnie z logiką maszyn stosowych. Zobaczymy to wyraźnie, gdy będziemy opisywać język pośredni (IL) w .NET w rozdziale 4. Dlaczego środowisko uruchomieniowe .NET i JVM (Java Virtual Machine) zostały zaprojektowane w ten sposób? Jak zawsze mamy tu do czynienia z pewną mieszanką powodów inżynierskich i historycznych. Kod maszyny stosowej jest kodem wyższego poziomu i lepiej odpowiada faktycznemu bazowemu sprzętowi. Środowisko uruchomieniowe .NET albo maszyna JVM mogłyby zostać napisane jako maszyny rejestrowe, ale wtedy, ile rejestrów byłoby potrzebnych? Ponieważ są tylko wirtualne, najlepszą odpowiedzią byłoby – nieskończona liczba rejestrów. Potrzebny byłby sposób ich obsługi i wielokrotnego wykorzystywania. Jak wyglądałaby optymalna, abstrakcyjna maszyna oparta na rejestrach?

Jeśli odłożymy takie problemy, pozwalając czemuś innemu (środowisku Java lub .NET w tym przypadku) dokonywać optymalizacji dla konkretnych platform, przetłumaczy to mechanizmy oparte na rejestrach lub oparte na stosie na określoną architekturę opartą na rejestrach. Maszyny oparte na stosie są jednak koncepcyjnie prostsze. Wirtualna maszyna stosowa (taka, która nie jest wykonywana przez prawdziwą, sprzętową maszynę stosową) może zapewniać dobrą niezależność od platformy, dając przy tym wysoce wydajny kod. Łącząc to ze wspomnianym lepszym zagęszczeniem kodu, otrzymujemy dobry wybór dla platformy, która ma być uruchamiana na szerokim zakresie urządzeń. Prawdopodobnie taki był powód, dla którego firma Sun zdecydowała się wybrać tę ścieżkę przy implementowaniu środowiska Java dla różnych, niewielkich

urządzeń. Firma Microsoft przy projektowaniu .NET również poszła tą drogą. Koncepcja maszyny stosowej jest po prostu elegancka, prosta i zwyczajnie działa. To sprawia, że implementowanie maszyny wirtualnej staje się przyjemniejszym zadaniem inżynierskim!

Z drugiej strony projekty maszyn wirtualnych opartych na rejestrach są bardziej zbliżone do projektów rzeczywistego sprzętu, na którym będą uruchamiane. Jest to bardzo pomocne w zakresie możliwych optymalizacji. Zwolennicy tego podejścia mówią, że pozwala ono osiągnąć dużo lepszą wydajność zwłaszcza w interpretowanych środowiskach uruchomieniowych. Interpreter ma znacznie mniej czasu na wykonanie zaawansowanych optymalizacji, im bardziej więc intepretowany kod będzie podobny do kodu maszynowego, tym lepiej. Dodatkowo operowanie na najczęściej używanym zestawie rejestrów zapewnia świetne wykorzystanie pamięci podręcznej[4].

Jak zawsze przy podejmowaniu decyzji trzeba poczynić pewne kompromisy. Dyskusja pomiędzy zwolennikami obu podejść jest długa i nierozstrzygnięta. W każdym razie faktem jest, że obecnie silnik wykonawczy .NET jest zaimplementowany jako maszyna stosowa, choć nie jest to całkiem czysta maszyna stosowa – zauważymy to w rozdziale 4. Zobaczymy też, jak stos obliczeń jest mapowany na bazowy sprzęt składający się z rejestrów i pamięci.

Uwaga Czy wszystkie maszyny wirtualne i silniki wykonawcze są maszynami stosowymi? Absolutnie nie! Jednym z wyjątków jest Dalvik, maszyna wirtualna w systemie Android firmy Google do wersji 4.4, która była implementacją JVM opartą na rejestrach. Był to interpreter pośredniego „kodu bajtowego Dalvik". Ale później kompilacja JIT (Just in Time) została wprowadzona w następcy maszyny Dalvik – środowisku Android Runtime (ART). Do innych przykładów należą BEAM – maszyna wirtualna dla Erlang/Elixir, Chakra – silnik wykonawczy JavaScript w przeglądarce IE9, Parrot (maszyna wirtualna dla Perl 6) oraz Lua VM (maszyna wirtualna dla Lua). Nie można więc powiedzieć, że tego rodzaju maszyna nie jest popularna.

---

4 Uwaga: W rozdziale 2 przyjrzymy się znaczeniu wzorców dostępu do pamięci w kontekście użycia pamięci podręcznej.

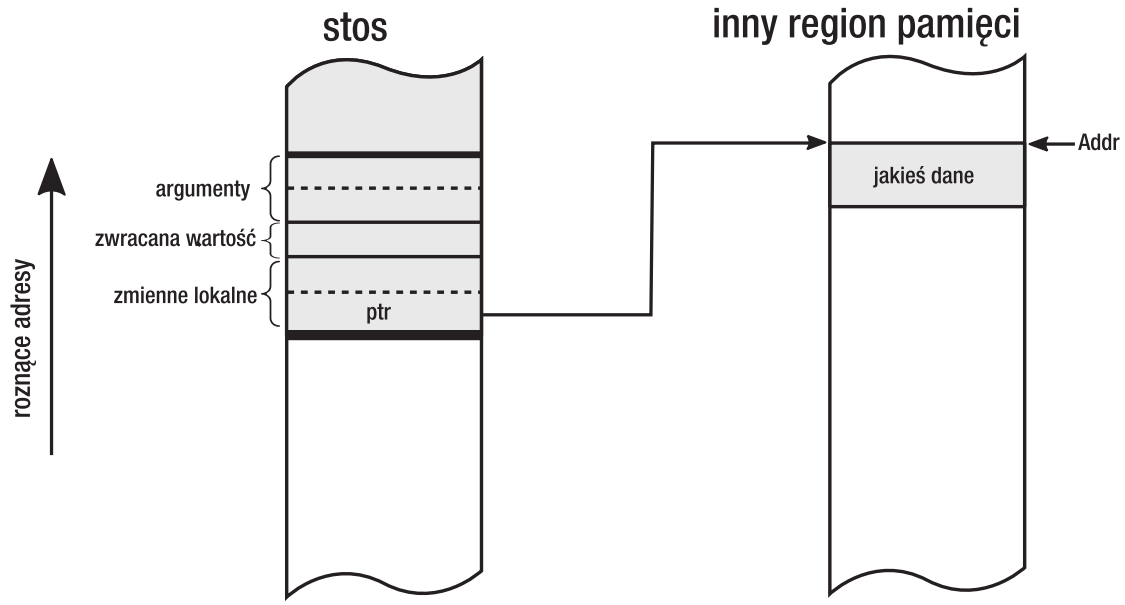
Wskaźnik

Jak dotąd wprowadziliśmy tylko dwa pojęcia pamięciowe: alokację statyczną i alokację na stosie (jako część ramki stosu). Pojęcie *wskaźnika (pointer)* jest bardzo ogólne i pojawia się od samego początku ery komputerowej – jak w przypadku wcześniej przedstawionego pojęcia wskaźnika instrukcji (licznika programu) albo wskaźnika stosu. Określone rejestry przeznaczone do adresowania pamięci, takie jak *rejestry indeksowe (index registers)* również mogą być traktowane jako wskaźniki[5].

Język PL/I został zaproponowany przez IBM ok. roku 1965 i w zamierzeniu miał mieć ogólne zastosowanie zarówno w świecie nauki, jak i biznesu. Choć ten cel nie został całkiem osiągnięty, to stanowi on ważny element historii informatyki, ponieważ był to pierwszy język, który wprowadził pojęcie wskaźników i alokacji pamięci. Harold Lawson, który brał udział w rozwoju języka PL/I, otrzymał w 2000 roku nagrodę od IEEE „za wymyślenie zmiennej wskaźnikowej i wprowadzenie tego pojęcia do PL/I, zapewniając w ten sposób po raz pierwszy możliwość elastycznej obsługi list łączonych w wysokopoziomowym języku ogólnego przeznaczenia". Dokładnie taka była potrzeba, która stała za wynalazkiem wskaźnika – zapewnienie możliwości przetwarzania list i działań na innych mniej lub bardziej złożonych strukturach danych. Pojęcie wskaźnika zostało następnie zastosowane podczas opracowywania języka C, który ewoluował z języka B (i jego poprzedników BCPL oraz CPL). Dopiero wersja języka FORTRAN 90 (następca FORTRAN 77) zdefiniowana w 1991 wprowadzała dynamiczną alokację pamięci (poprzez procedury `allocate/deallocate`), atrybut `POINTER`, przypisywanie wskaźników oraz instrukcję `NULLIFY`.

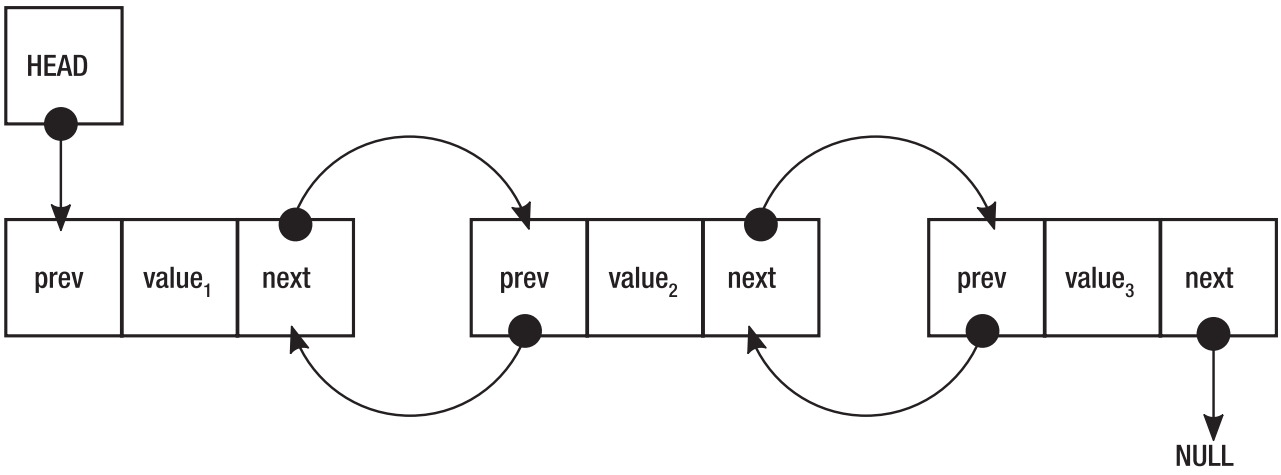
Wskaźnik jest zmienną, w której przechowujemy adres jakiegoś położenia w pamięci. Pozwala nam to odwoływać się do innych miejsc w pamięci poprzez ich adres. Rozmiar wskaźnika jest związany z długością słowa, o której wspominaliśmy wcześniej, co wynika z architektury danego komputera. Obecnie więc zwykle mamy do czynienia z 32-bitowymi lub 64-bitowymi wskaźnikami. Ponieważ sam wskaźnik zajmuje niewielki obszar w pamięci, możemy go umieszczać na stosie (na przykład jako zmienną lokalną lub argument funkcji) albo w rejestrze procesora. Rysunek 1-8 pokazuje typową sytuację, gdzie jedna ze zmiennych lokalnych (zapisana wewnątrz ramki aktywacji funkcji) jest wskaźnikiem do innego regionu w pamięci pod adresem `Addr`.

5 W kontekście adresowania pamięci ważnym ulepszeniem był rejestr indeksowy wprowadzony w maszynie Manchester Mark 1, następcy komputera „Baby". Rejestr indeksowy pozwalał nam na bezpośrednie odwoływanie się do pamięci, dodając jego wartość do innego rejestru. Dzięki temu mniej instrukcji było wymaganych do wykonywania operacji na ciągłych regionach pamięci, takich jak tablice.



Rysunek 1-8 *Zmienna lokalna w funkcji, będąca wskaźnikiem* `ptr` *prowadzącym do pamięci pod adresem* `Addr`

Prosta koncepcja wskaźników pozwala nam budować złożone struktury danych, takie jak listy łączone lub drzewa, ponieważ struktury danych w pamięci mogą się odwoływać do innych elementów, tworząc bardziej złożone struktury (zobacz rysunek 1-9).



Rysunek 1-9 *Wskaźniki używane do zbudowania struktury listy łączonej dwukierunkowej, gdzie każdy element wskazuje na swój element poprzedni oraz następny*

Co więcej, wskaźniki mogą zapewniać tak zwaną *arytmetykę wskaźnikową*. Można do nich dodawać lub odejmować od nich jakieś wartości, aby odwoływać się do miejsca w pamięci położonego odpowiednio względem danego wskaźnika. Na przykład

operator inkrementacji może zwiększać wartość wskaźnika o wartość rozmiaru wskazywanego obiektu, a nie o jeden bajt (jak można by się spodziewać).

W wysokopoziomowych językach, takich jak Java lub C# wskaźniki często nie są dostępne albo muszą być jawnie włączane w trybie tzw. kodu niebezpiecznego. Przyczyny takiego stanu rzeczy okażą się jaśniejsze, gdy będziemy omawiać ręczne zarządzanie pamięcią przy użyciu wskaźników w dalszej części tego rozdziału.

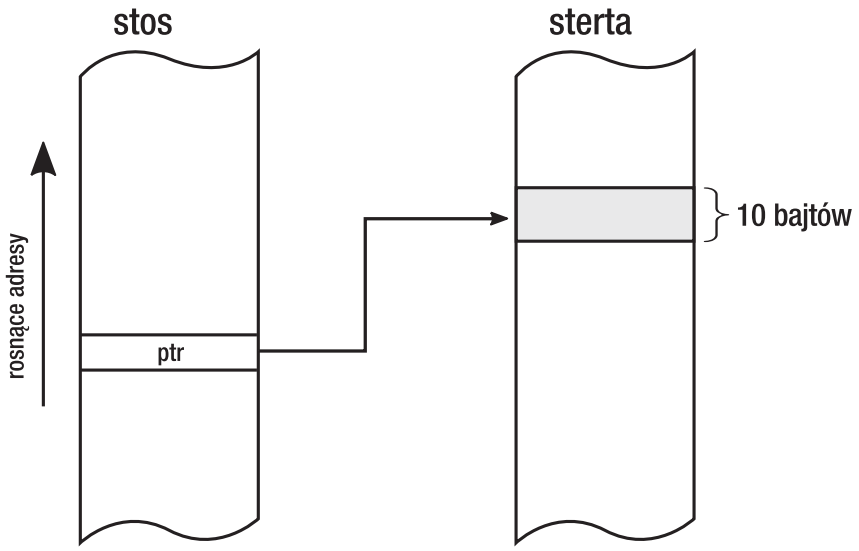
Sterta

Dochodzimy wreszcie do najważniejszego pojęcia w kontekście zarządzania pamięcią w .NET. *Sterta (heap)*, czasami też nazywana *magazynem swobodnym (Free Store)*, jest obszarem pamięci używanym do dynamicznego alokowania obiektów. Magazyn swobodny jest lepszą nazwą, ponieważ nie sugeruje żadnej konkretnej struktury wewnętrznej, ale raczej określa cel tego obszaru. Właściwie można by zapytać, jakie jest powiązanie pomiędzy stertą jako strukturą danych a stertą w odniesieniu do dynamicznego zarządzania pamięcią. Prawda jest taka, że nie ma takiego powiązania. Podczas gdy stos jest dobrze zorganizowany (jest oparty na pojęciu struktury danych LIFO), to sterta bardziej przypomina „czarną skrzynkę", którą możemy prosić o zapewnienie pamięci, nie zajmując się tym, skąd będzie ona pochodzić. Dlatego więc wcześniej wspomniane określenie „magazyn swobodny" albo „pula" byłoby chyba bardziej odpowiednią nazwą. Nazwa sterta została chyba użyta na początku w tradycyjnym znaczeniu angielskiego wyrazu „heap", określającym „bezładny zbiór rzeczy" w przeciwieństwie do dobrze uporządkowanego stosu. Historycznie rzecz biorąc, alokacja na stercie została wprowadzona w języku ALGOL 68, ale ten standard nie przyjął się zbyt szeroko. Prawdopodobnie stąd właśnie pochodzi określenie „sterta". Faktem jest, że prawdziwe historyczne pochodzenie tej nazwy jest obecnie dość trudne do ustalenia.

Sterta jest mechanizmem pamięciowym mogącym zapewniać ciągły blok pamięci o określonym rozmiarze. Ta operacja jest nazywana *dynamiczną alokacją pamięci*, ponieważ zarówno rozmiar, jak i faktyczne położenie danego obszaru pamięci nie muszą być znane w czasie kompilacji. Ponieważ położenie w pamięci nie jest znane w czasie kompilacji, do pamięci alokowanej dynamicznie musimy odwoływać się przez wskaźnik. Z tego względu pojęcia wskaźnika i sterty są ze sobą naturalnie powiązane.

Adres zwracany przez jakąś funkcję typu „przydziel mi X bajtów pamięci" powinien być oczywiście zapamiętany w jakimś wskaźniku, aby możliwe było dalsze odwoływanie się do utworzonego bloku pamięci. Wskaźnik ten może być przechowywany na stosie (zobacz rysunek 1-10), na samej stercie lub gdziekolwiek indziej.

```
PTR ptr = allocate(10);
```

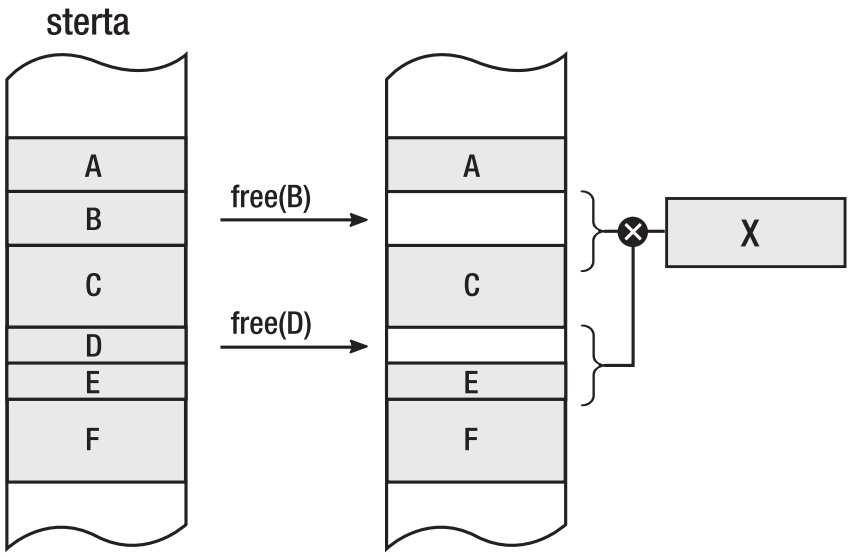


Rysunek 1-10 *Stos ze wskaźnikiem ptr oraz 10-bajtowy blok pamięci na stercie*

Operacja odwrotna do operacji alokacji jest nazywana *dealokacją* i polega na zwróceniu danego bloku pamięci do puli pamięci dostępnej do wykorzystania w przyszłości. Dokładny sposób alokowania przez stertę obszaru o danym rozmiarze stanowi szczegół implementacyjny. Istnieje wiele mechanizmów alokujących i niektóre z nich poznamy wkrótce.

Poprzez alokację i dealokację wielu bloków pamięci możemy doprowadzić do sytuacji, gdy nie będzie wystarczająco dużego ciągłego bloku na dany obiekt, choć sumaryczna dostępna na stercie ilość wolnego miejsca byłaby wystarczająca. Taka sytuacja jest określana *fragmentacją* sterty i może prowadzić do znacznej niewydolności wykorzystania pamięci. Rysunek 1-11 ilustruje taki problem, gdy nie ma wystarczająco dużego wolnego ciągłego miejsca na obiekt X. Istnieje wiele różnych strategii wykorzystywanych przez mechanizmy alokujące przy zarządzaniu pamięcią w możliwie optymalny sposób w celu unikania fragmentacji.

Warto też zauważyć, że kolejnym szczegółem implementacyjnym może być to, czy mamy tylko pojedynczą stertę, czy wiele wystąpień stert w pojedynczym procesie (zobaczymy to przy dokładniejszym omawianiu .NET).



Rysunek 1-11 *Fragmentacja – po usunięciu obiektów B i D nie ma wystarczającego miejsca na nowy obiekt X, choć w sumie jest dość wolnego miejsca*

Podsumujmy krótko różnice pomiędzy stosem i stertą w tabeli 1-1.

Tabela 1-1 *Porównanie funkcjonalności stosu i sterty*

Właściwość	Stos	Sterta
Czas życia	Zakres zmiennych lokalnych (odkładane na stos na wejściu do funkcji, zdejmowane ze stosu na wyjściu z funkcji)	Jawnie określony (przez alokację i ewentualne zwolnienie pamięci)
Zakres	Lokalny (dla wątku[1])	Globalny (dostępne dla każdego, kto ma wskaźnik)
Dostęp	Zmienna lokalna, argumenty funkcji	Wskaźnik
Czas dostępu	Szybki (prawdopodobnie region w pamięci podręcznej procesora)	Wolniejszy (może być nawet tymczasowo zapisywany na twardym dysku)
Alokacja	Przesuwanie wskaźnika stosu	Różne możliwe strategie
Czas alokacji	Bardzo szybki	Wolniejszy (zależy od strategii alokacji)
Zwalnianie	Przesuwanie wskaźnika stosu	Różne możliwe strategie
Użycie	Parametry podprogramów, zmienne lokalne, ramki aktywacji, niewielkie dane o rozmiarze znanym w czasie kompilacji	Wszystko

Właściwość	Stos	Sterta
Pojemność	Ograniczona (zwykle kilka MB na wątek)	Nieograniczona (w ramach dostępnej pojemności pamięci i dysku)
Zmienny rozmiar	Nie	Tak[2]
Fragmentacja	Nie	Prawdopodobna
Główne zagrożenia	Przepełnienie stosu	Wyciek pamięci (zapomnienie o zwolnieniu zaalokowanej pamięci), fragmentacja

1 Nie jest to całkiem prawdziwe, gdyż możemy przekazywać wskaźnik do zmiennej na stosie do innych wątków. Jednakże z pewnością nie jest to standardowe użycie.

2 Ze względu na dynamiczną naturę sterty, istnieją funkcje pozwalające nam zmieniać rozmiar (realokować) danego bloku pamięci.

Oprócz tych różnic stos i sterta najczęściej są lokalizowane po przeciwległych stronach przestrzeni adresowej procesu. Wrócimy do szczegółowego ułożenia stosu i sterty w przestrzeni adresowej procesu podczas rozważań nad niskopoziomowym zarządzaniem pamięcią w rozdziale 2. W każdym razie należy pamiętać, że jest to tylko szczegół implementacyjny. Dzięki zapewnieniu abstrakcji dla typów wartościowych i referencyjnych (które zostaną wprowadzone w rozdziale 4) nie powinniśmy przejmować się, gdzie są one tworzone.

Przejdźmy teraz do omówienia ręcznego i automatycznego zarządzania pamięcią. Ellis i Stroustrup pisali w podręczniku *The Annotated C++ Reference Manual*:

> *Programiści C uważają, że zarządzanie pamięcią jest zbyt ważnym zadaniem, żeby zostawiać je komputerowi. Programiści Lisp uważają, że zarządzanie pamięcią jest zbyt ważnym zadaniem, żeby zostawiać je użytkownikowi.*

Ręczne zarządzanie pamięcią

Jak na razie mieliśmy do czynienia z „ręcznym zarządzaniem pamięcią". Oznacza to w szczególności, że programista odpowiada za jawne alokowanie pamięci, a gdy nie jest już dłużej potrzebna, powinien ją dealokować. To naprawdę ręczna praca. To jak z ręczną skrzynią biegów w większości samochodów europejskich. Jestem z Europy i po prostu jesteśmy przyzwyczajeni do ręcznego zmieniania biegów. Musimy pomyśleć, czy już jest dobry moment na zmianę biegu, czy powinniśmy poczekać parę sekund, aż obroty silnika będą odpowiednio wysokie. Ma to jedną dużą zaletę – mamy pełną kontrolę nad samochodem. Sami odpowiadamy za to, czy silnik jest wykorzystywany optymalnie, czy nie. A ponieważ ludzie wciąż lepiej adaptują się do zmieniających się warunków, dobrzy kierowcy mogą

to robić lepiej niż automatyczna skrzynia biegów. Oczywiście jest też jedna duża wada. Zamiast myśleć o naszym głównym celu – dotarciu z punktu A do punktu B, musimy dodatkowo myśleć o zmienianiu biegów – setki, tysiące razy podczas długiej podróży. Jest to zarówno czasochłonne, jak i męczące. Znam osoby, które twierdzą, że jest to zajmujące, a oddanie kontroli automatycznej skrzyni biegów jest nudne. Mogę się nawet z nimi zgodzić. Podoba mi się jednak, jak ta przenośnia pasuje do zarządzania pamięcią.

Gdy mówimy o jawnym alokowaniu i dealokowaniu pamięci, odpowiada to dokładnie korzystaniu z ręcznej skrzyni biegów. Zamiast myśleć o naszym głównym celu, którym jest pewnie jakieś biznesowe zadanie do wykonania przez nasz kod, musimy też myśleć o tym, jak zarządzać pamięcią swojego programu. Odciąga to nas od naszego głównego celu i zajmuje naszą cenną uwagę. Zamiast myśleć o algorytmach, logice biznesowej i domenach, musimy również myśleć o tym, kiedy i jak dużo pamięci będziemy potrzebować. Jak długo będzie nam ona potrzebna? Kto będzie odpowiadać za jej zwalnianie? Czy to brzmi jak logika biznesowa? Oczywiście, że nie. Pytanie, czy to dobrze, czy nie, to już inna kwestia.

Dobrze znany język C został zaprojektowany przez Dennisa Rithiego gdzieś we wczesnych latach 70. XX wieku i stał się jednym z najszerzej używanych języków programowania na świecie. Historia wyewoluowania języka C z języka ALGOL poprzez pośrednie języki CPL, BCPL, i B jest sama w sobie dość ciekawa, ale w naszym kontekście ważnie jest, że język C wraz z językiem Pascal (wywodzącym się bezpośrednio z języka ALGOL) były w tym okresie dwoma najpopularniejszymi językami z jawnym zarządzaniem pamięcią. Jeśli chodzi o C, to mogę bez wątpienia powiedzieć, że kompilator tego języka został napisany dla niemal każdej architektury sprzętowej kiedykolwiek stworzonej. Nie zdziwiłbym się, gdyby statki kosmiczne obcych miały na swoich pokładach własny kompilator C (pewnie wraz z implementacją stosu TCP/IP). Wpływ tego języka na inne języki programowania jest niewyobrażalnie ogromny. Zatrzymajmy się na chwilę i przyjrzyjmy się mu dokładniej w kontekście zarządzania pamięcią. Pozwoli nam to wymienić kilka charakterystycznych cech ręcznego zarządzania pamięcią.

Popatrzmy na prosty kod przykładowy napisany w C i przedstawiony na listingu 1-4.

Listing 1-4 *Przykładowy program w języku C demonstrujący ręczne zarządzanie pamięcią*

```
#include <stdio.h>

void printReport(int* data)
{
    printf("Report: %d\n", *data);
}
```

```
int main(void) {
    int *ptr;
    ptr = (int*)malloc(sizeof(int));
    if (ptr == 0)
    {
        printf("ERROR: Out of memory\n");
        return 1;
    }
    *ptr = 25;
    printReport(ptr);
    free(ptr);
    ptr = NULL;
    return 0;
}
```

Jest to oczywiście nieco przesadzony przykład, ale dzięki niemu możemy jaśniej zilustrować problem. Możemy zauważyć, że ten krótki kod ma w istocie tylko jeden prosty cel biznesowy: wypisać „raport". Dla uproszczenia raport ten składa się w tym przypadku z pojedynczej liczby całkowitej, ale możemy sobie wyobrazić, że jest to bardziej skomplikowana struktura zawierająca też wskaźniki do innych struktur danych. Ten prosty cel biznesowy jest obudowany dużą ilością „rytualnego kodu" zajmującego się samą tylko pamięcią. Jest to esencja ręcznego zarządzania pamięcią.

Podsumowując powyższy fragment kodu, oprócz napisania logiki biznesowej, programista musi:

- Zaalokować odpowiednią ilość pamięci na potrzebne dane, korzystając z funkcji `malloc`.
- Rzutować zwrócony ogólny wskaźnik (`void*`) na właściwy typ wskaźnika (`int*`), aby określić, że wskazuje on na wartość liczbową (typ `int` w przypadku języka C).
- Zapamiętać wskaźnik do zaalokowanego regionu pamięci w lokalnej zmiennej wskaźnikowej `ptr`.
- Sprawdzić, czy udało się zaalokować potrzebną ilość pamięci (w przypadku niepowodzenia zwrócony zostanie adres 0).
- Uzyskać dostęp do pamięci wskazywanej przez adres zapisany we wskaźniku, aby zapisać pewne dane (liczbową wartość 25).
- Przekazać wskaźnik do innej funkcji `printReport`, która sama też odczytuje wartość z adresu wskazywanego przez wskaźnik.

- Zwolnić zaalokowaną pamięć, gdy nie jest już dłużej potrzebna, korzystając z funkcji `free`.
- Dla pewności powinniśmy zmienić wskaźnik na wartość specjalną `NULL` (co jest sposobem zaznaczenia, że ten wskaźnik na nic już nie wskazuje – wartość ta odpowiada wartości 0[6]).

Jak widać, musimy mieć wiele rzeczy na uwadze, gdy ręcznie zarządzamy pamięcią. Co więcej, w każdym z powyższych kroków możemy się pomylić albo o czymś zapomnieć, co może prowadzić do różnych poważnych problemów. Przechodząc przez każdy z tych kroków, zastanówmy się, co złego może się stać:

- Powinniśmy dokładnie wiedzieć, ile pamięci potrzebujemy. W naszym przykładzie wystarczy po prostu użycie `sizeof(int)`, ale co w przypadku, gdy mielibyśmy do czynienia z bardziej złożonymi, zagnieżdżonymi strukturami danych? Łatwo sobie można wyobrazić sytuację, w której alokujemy zbyt mało pamięci ze względu na jakiś drobny błąd w ręcznych obliczeniach wymaganego rozmiaru. Później, gdy będziemy chcieli zapisać coś do takiego regionu pamięci (lub z niego odczytać), otrzymamy prawdopodobnie *błąd segmentacji* (*Segmentation Fault*) – próbując uzyskać dostęp do pamięci, która nie została przez nas zaalokowana lub została zaalokowana do innego celu. Z drugiej strony możemy przez podobny błąd zaalokować zbyt dużo pamięci, co będzie skutkowało mało efektywnym wykorzystaniem pamięci.
- Rzutowanie zawsze może być podatne na błędy i może wprowadzać trudne do zdiagnozowania błędy, jeśli przez przypadek wprowadzimy niezgodność typów. Próbowalibyśmy interpretować wskaźnik jakiegoś typu, tak jakby był zupełnie innego typu, co łatwo prowadzi do niebezpieczeństwa naruszenia dostępu.
- Zapamiętywanie adresu jest prostą sprawą. Co jednak, jeśli zapomnimy to zrobić? Będziemy mieli zaalokowany obszar pamięci i brak możliwości jego zwolnienia – zapomnieliśmy właśnie, jaki jest jego adres! Jest to bezpośrednia droga do problemu wycieku pamięci, gdyż niezwolniona pamięć może się szybko rozrosnąć. Co więcej wskaźnik może być przechowywany w czymś bardziej skomplikowanym niż zmienna lokalna. Co będzie, jeśli zgubimy wskaźnik do złożonego grafu obiektów, ponieważ zwolniliśmy pamięć dla jakiejś struktury, która go zawierała?
- Pojedyncze sprawdzenie, czy byliśmy w stanie zaalokować żądaną ilość pamięci, nie jest zbyt uciążliwe. Ale powtarzanie tego setki razy w każdej funkcji z osobna na pewno takie będzie. Prawdopodobnie zdecydujemy się na pominięcie tego sprawdzania, ale może to doprowadzić nas do niezdefiniowanego zachowania

6 Szczegóły implementacyjne wartości NULL w przypadku .NET będą wyjaśnione w rozdziale 10.

w wielu miejscach naszej aplikacji, gdy będziemy próbować uzyskać dostęp do pamięci, której w ogóle nie udało się z powodzeniem zaalokować.

- Odczytywanie danych, do których prowadzą wskaźniki, jest zawsze niebezpieczne. Nigdy nie wiadomo, co będzie pod adresem, do którego prowadzi wskaźnik. Czy nadal mamy tam prawidłowy obiekt, czy może został on już zwolniony? Czy dany wskaźnik jest w ogóle prawidłowy? Czy wskazuje na właściwą przestrzeń adresową w pamięci użytkownika? Pełna kontrola nad wskaźnikami w językach, takich jak C prowadzi do tego rodzaju obaw. Ręczna kontrola nad wskaźnikami prowadzi do poważnych naruszeń bezpieczeństwa – sam programista musi zadbać o to, aby nie udostępniać danych poza obszarami, w których powinny być dostępne zgodnie z aktualnym modelem typów.
- Przekazywanie wskaźnika pomiędzy funkcjami i wątkami jedynie mnoży te obawy w środowisku wielowątkowym.
- Musimy pamiętać o zwalnianiu zaalokowanej pamięci. Jeśli pominiemy ten krok, uzyskamy wyciek pamięci. W przykładzie tak prostym, jak powyżej, oczywiście trudno będzie zapomnieć o wywołaniu funkcji `free`. Jest to jednak dużo bardziej problematyczne w bardziej złożonych bazach kodu, gdzie odpowiedzialność za poszczególne struktury danych może nie być tak oczywista i gdzie wskaźniki do takich struktur mogą być przekazywane w różne miejsca. Istnieje też jeszcze jedno ryzyko – nikt nas nie może powstrzymać przed zwolnieniem pamięci, która już została zwolniona. Jest to kolejna okazja do niezdefiniowanego zachowania i prawdopodobnie może spowodować błąd segmentacji.
- Wreszcie na koniec powinniśmy oznaczać nasz wskaźnik jako `NULL` (albo 0) w celu określenia, że już nie wskazuje na prawidłowy obiekt. W przeciwnym razie będziemy mieć do czynienia z tak zwanym *wiszącym wskaźnikiem (dangling pointer)*, który wcześniej lub później doprowadzi do błędu segmentacji lub innego niezdefiniowanego zachowania, ponieważ ktoś może próbować odczytać wskazywane przez niego dane, sądząc, że nadal prowadzi do prawidłowych danych.

Jak widać, z perspektywy programisty jawne alokowanie i dealokowanie pamięci może się stać naprawdę nieporęczne. Jest to bardzo potężna funkcjonalność, która z pewnością ma swoje zastosowania. Tam, gdzie znaczenie ma jak największa wydajność, a programista musi być w 100% pewien, co się dzieje wewnątrz programu – podejście to może być przydatne. Jednak „duża władza oznacza dużą odpowiedzialność", więc jest to broń obosieczna. W miarę rozwoju inżynierii oprogramowania również języki stawały się coraz bardziej zaawansowane, pomagając programiście uciec przed wszystkimi tymi troskami.

Idąc dalej, bezpośredni następca języka C – język C++ nie zmienił wiele w tym zakresie. Jednakże warto poświęcić kilka chwil językowi C++, ponieważ jest tak popularny

i wprowadza inne szeroko używane pojęcia. Jak wszyscy wiemy, jest to język z ręcznym zarządzaniem pamięcią. Tłumacząc poprzedni przykład na C++, uzyskamy kod pokazany na listingu 1-5.

Listing 1-5 *Przykładowy program w języku C++ pokazujący ręczne zarządzanie pamięcią*

```
#include <iostream>

void printReport(int* data)
{
    std::cout << "Report: " << *data << "\n";
}

int main()
{
    try
    {
        int* ptr;
        ptr = new int();
        *ptr = 25;
        printReport(ptr);
        delete ptr;
        ptr = 0;
        return 0;
    }
    catch (std::bad_alloc& ba)
    {
        std::cout << "ERROR: Out of memory\n";
        return 1;
    }
}
```

W kontekście naszych poprzednich rozważań możemy dostrzec kilka istotnych usprawnień:

- Operator `new` zajmuje się alokacją odpowiedniej ilości pamięci, wiedząc, ile jej będzie potrzebować, dzięki wsparciu kompilatora (który podpowiada rozmiar danego obiektu).

- Nie musimy rzutować otrzymanego wskaźnika na odpowiedni typ. To usuwa pewne problemy z bezpieczeństwem typów, które rozważaliśmy wcześniej.
- Obsługa błędów również się poprawiła, ponieważ nie jesteśmy zobligowani do ręcznego sprawdzania powodzenia alokacji, gdyż w przypadku problemu zostanie wyrzucony wyjątek.

Nadal jednak widać dużo dodatkowego kodu w tym przykładzie. Pojawił się również nowy problem. Co będzie, jeśli funkcja `printReport()` wyrzuci wyjątek? Bez odpowiedniej obsługi błędów możemy łatwo pominąć operator `delete` i doprowadzić do wycieku pamięci. Naprawienie naszego kodu przykładowego jest łatwe, ale może nie być to takie oczywiste w bardziej złożonych aplikacjach, gdzie odpowiedzialność za dane (kto i na jakim poziomie powinien usuwać takie wskaźniki) może nie być tak trywialna.

Wszystkie problemy, które widzieliśmy w tym rozdziale, są dodatkowo potęgowane w środowiskach wielowątkowych, gdy wskaźniki mogą być wspólnie wykorzystywane przez wiele jednostek wykonawczych. Trzeba brać pod uwagę odpowiednią synchronizację, aby nie dopuścić do wymieszania nieprawidłowych danych. Na przykład, co będzie, jeśli jeden wątek sprawdza, czy dany wskaźnik jest poprawny (nie `NULL`), natomiast inny zaraz potem zwolni pamięć przez niego wskazywaną? Takie sytuacje mogą prowadzić do nieregularnych i bardzo trudnych do diagnozowania problemów. W świecie jawnego zarządzania pamięcią, to programista odpowiada za zapewnianie odpowiedniego mechanizmu synchronizacji w celu uniknięcia takich sytuacji.

Przykład w języku C++ zaprezentowany na listingu 1-5 celowo nie jest zgodny z obecnymi wzorcami wykorzystania pamięci w tym języku. Powinien wykorzystywać jakąś technikę RAII (Resource Acquisition Is Initialization – inicjowanie przy pozyskaniu zasobu) – gdzie zasób (taki jak pamięć) jest reprezentowany przez zmienną lokalną typu implementującego jakiegoś rodzaju logikę związaną z własnością pamięci. Taki przykład zostanie przedstawiony później na listingu 1-10. Choć, jak się przekonamy, takie wzorce pomagają w rozwiązywaniu pewnych problemów, to nie zmieniają wiele w naszej ogólnej dyskusji na temat ręcznego i automatycznego zarządzania pamięcią.

Automatyczne zarządzanie pamięcią

Aby przezwyciężyć problemy z ręcznym zarządzaniem pamięcią i zapewnić programiście bardziej przyjazny sposób jej obsługi, pojawiły się różne podejścia do automatycznego zarządzania pamięcią. Dobrze jest wiedzieć, że nawet jeden z najstarszych wysokopoziomowych języków programowania – LISP (zaproponowany ok. 1958 roku – tylko kilka

lat po języku FORTRAN) miał sporo do zaoferowania na tym polu. W języku głównie funkcjonalnym i opierającym się w znacznym stopniu na przetwarzaniu list – ręczne zarządzanie pamięcią byłoby bardzo niewygodne. Paradygmat programowania funkcjonalnego traktuje programy jako obliczanie wyników połączonych funkcji i stanowczo unika modyfikowania danych (mutacji) i związanych z tym efektów ubocznych. Alokowanie i dealokowanie pamięci jest operacją mocno modyfikującą i ma oczywiste efekty uboczne. Obsługa pamięci w taki sposób w kodzie funkcjonalnym zaburzyłaby ten kod elementami imperatywnymi, podczas gdy LISP został zaprojektowany jako język wysoce deklaratywny. Jak powiedział twórca języka LISP: „konieczność jawnego czyszczenia list wyglądałaby niezwykle brzydko". Trzeba było więc opracować coś bardziej skomplikowanego. Pierwsze wersje języka LISP miały wbudowaną funkcję `eralist` (erase list – wyczyść listę), ale została ona usunięta po wprowadzeniu automatycznego zarządzania pamięcią.

LISP był bardzo innowacyjnym językiem i jego projekt pomógł w wynalezieniu wielu ważnych pomysłów informatycznych, a jednym z nich było automatyczne zarządzanie pamięcią. John McCarthy, wynalazca języka LISP, jest też uważany za jednego z „ojców założycieli" sztucznej inteligencji oraz autora pierwszych algorytmów odśmiecania pamięci. Wiele pomysłów wymyślonych na potrzeby tego języka nadal spełnia swoje zadanie i jest wykorzystywana w dzisiejszych językach programowania. Można z pewnością stwierdzić, że automatyczne zarządzanie pamięcią narodziło się w języku LISP. Pierwsza praca napisana przez McCarthy'ego w roku 1958 wprowadzała algorytm Mark and Sweep (oznacz i zamieć), który zbadamy dokładnie w późniejszych rozdziałach, ponieważ jest wciąż używany w środowisku .NET i wielu innych miejscach.

Język LISP dzięki swojej wyrazistości i zwięzłości może przedstawić nasz program przykładowy w prostej formie pokazanej na listingu 1-6.

Listing 1-6 *Przykładowy program w języku LISP pokazujący automatyczne zarządzanie pamięcią*

```
(defun printReport(data)
   (write-line (format nil "Report: ~a" data))
)

(prog
   ((ptr 25))
   (printReport ptr)
)
```

Dzięki automatycznemu zarządzaniu pamięcią cała otoczka kodu zniknęła i możemy wyraźnie zobaczyć właściwy cel biznesowy programu – wypisanie „raportu".

John McCarthy przytoczył ciekawą anegdotę w swojej pracy dotyczącej projektowania języka LISP: „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" (funkcje rekurencyjne wyrażeń symbolicznych i ich obliczanie przez maszynę, część I). Opisał tam krótko ten mechanizm, ale nazwał go po prostu „odzyskiwaniem". Później dodał przypis do tego fragmentu:

> *Nazywaliśmy już ten proces „odśmiecaniem pamięci", ale wycofałem się z użycia go w tej pracy – w przeciwnym razie korektorki z Research Laboratory of Electronics by jej nie przepuściły.*

Nie tylko sama nazwa, ale też cały pomysł na ten mechanizm był opisany i gotowy do implementacji. Obecnie określenia mechanizm automatycznego zarządzania pamięcią i *odśmiecanie pamięci (garbage collection)* są używane wymiennie. Możemy to zdefiniować jako mechanizm, który zdejmuje z programisty odpowiedzialność za ręczne zarządzanie pamięcią, tak że raz utworzone obiekty są automatycznie usuwane (a pamięć przez nie wykorzystywana jest odzyskiwana), gdy nie są już dłużej potrzebne.

Jednym z wniosków, jakie chciałbym przekazać w tej książce, jest fakt, że nawet gdy zarządzanie pamięcią jest w pełni automatyczne, może to powodować problemy. Na potwierdzenie warto zacytować interesujący fakt związany z pierwszą implementacją odśmiecania pamięci w języku LISP. Jak wspomina McCarthy w książce *History of Programming Languages I (Historia języków programowania I)*, podczas pierwszej publicznej demonstracji języka LISP na jednym z sympozjów na uczelni MIT urządzenie Flexowriter (elektryczna maszyna do pisania z tego okresu) zaczęło ze względu na drobne niedopatrzenie drukować wiele stron z komunikatem błędu zaczynającym się od zdania:

> *ODŚMIECANIE ZOSTAŁO WYWOŁANE. PONIŻEJ PODANO KILKA INTERESUJĄCYCH DANYCH*

Z tego względu prezentację trzeba było odwołać, a publiczność miała ubaw. Tylko John wiedział, że spowodowane to było błędnym użyciem mechanizmu odśmiecania pamięci. Choć był to raczej błąd ludzki niż błąd algorytmu, możemy powiedzieć, że mechanizmy odśmiecania pamięci miały kłopoty od samego początku!

Alokator, mutator i kolektor

Mutatory i inne pojęcia, z którymi zapoznamy się w tym rozdziale, są ważnymi terminami w akademickich badaniach nad automatycznym zarządzaniem pamięcią. Dzięki jasnym definicjom możemy je później jednoznacznie odróżniać w pracach akademickich i opracowaniach technicznych. Można na przykład powiedzieć o „koszcie mutatora" dla