MYŚL JAK PROGRAMISTA

TECHNIKI KREATYWNEGO ROZWIĄZYWANIA PROBLEMÓW

V. ANTON SPRAUL

Tytuł oryginału: Think Like a Programmer: An Introduction to Creative Problem Solving

Tłumaczenie: Jacek Janusz

ISBN: 978-83-246-7284-4

Original edition copyright © 2012 by V. Anton Spraul.
All rights reserved.

Published by arrangement with No Starch Press, Inc.

Polish edition copyright © 2013 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/myprog
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/myprog.zip

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

PODZIĘKOWANIA **7**

WSTĘP **9**
O książce 11
Wymagania wstępne 11
Wybrane zagadnienia 12
Styl programowania 12
Ćwiczenia 12
Dlaczego C++? 13

1
STRATEGIE ROZWIĄZYWANIA PROBLEMÓW **15**
Klasyczne łamigłówki 17
Lis, gęś i kukurydza 17
Łamigłówki z przesuwanymi elementami 22
Sudoku 26
Zamek Quarrasi 30
Ogólne techniki rozwiązywania problemów 32
Miej zawsze jakiś plan 32
Ponownie zaprezentuj problem 33
Podziel problem 34
Rozpocznij z wiedzą, którą posiadasz 35
Uprość problem 36
Szukaj analogii 37
Eksperymentuj 38
Nie popadaj we frustrację 38
Ćwiczenia 40

Kup ksi k
Pole ksi k

2
PRAWDZIWE ŁAMIGŁÓWKI 41
Elementy języka C++ wykorzystywane w tym rozdziale 42
Tworzenie wzorów na wyjściu 42
Przetwarzanie danych wejściowych 48
Analiza problemu 49
Łączenie wszystkich elementów w całość 58
Śledzenie stanu 60
Podsumowanie 73
Ćwiczenia 74

3
ROZWIĄZYWANIE PROBLEMÓW ZA POMOCĄ TABLIC 77
Podstawowe informacje o tablicach 78
Przechowywanie danych 79
Kopiowanie 79
Odczytywanie i przeszukiwanie 80
Sortowanie 81
Obliczenia statystyczne 84
Rozwiązywanie problemów za pomocą tablic 85
Optymalizacja 89
Tablice ze stałymi wartościami 91
Tablice z wartościami nieskalarnymi 94
Tablice wielowymiarowe 96
Kiedy należy używać tablic 99
Ćwiczenia 104

4
ROZWIĄZYWANIE PROBLEMÓW
ZA POMOCĄ WSKAŹNIKÓW I PAMIĘCI DYNAMICZNEJ 107
Podstawowe informacje o wskaźnikach 108
Korzyści z używania wskaźników 109
Struktury danych o wielkości definiowanej w trakcie działania programu 109
Struktury danych o zmiennych rozmiarach 110
Współdzielenie pamięci 110
Kiedy należy używać wskaźników? 111
Pamięć ma znaczenie 112
Stos i sterta 113
Rozmiar pamięci 116
Czas życia 118
Rozwiązywanie problemów za pomocą wskaźników 119
Łańcuchy o zmiennej długości 119
Listy powiązane 130
Wnioski i następne działania 139
Ćwiczenia 139

Kup ksi k Pole ksi k

5
ROZWIĄZYWANIE PROBLEMÓW ZA POMOCĄ KLAS 143
Przegląd podstawowych informacji o klasach 144
Cele użycia klas 146
Enkapsulacja 146
Ponowne użycie kodu 147
Dzielenie problemu 147
Hermetyzacja 148
Czytelność 150
Wyrazistość 150
Tworzenie przykładowej klasy 151
Podstawowy schemat klasy 152
Metody wspierające 156
Klasy z danymi dynamicznymi 160
Dodawanie węzła 162
Reorganizacja listy 165
Destruktor 169
Kopiowanie głębokie 170
Obraz całości dla klas z pamięcią dynamiczną 175
Błędy, jakich należy unikać 176
Klasa fikcyjna 176
Jednozadaniowce 177
Ćwiczenia 178

6
ROZWIĄZYWANIE PROBLEMÓW ZA POMOCĄ REKURENCJI 181
Przegląd podstawowych informacji o rekurencji 182
Rekurencja nieogonowa i ogonowa 182
Wielki Pomysł Rekurencyjny 191
Często popełniane błędy 194
Zbyt wiele parametrów 195
Zmienne globalne 196
Używanie rekurencji w dynamicznych strukturach danych 198
Rekurencja i listy powiązane 198
Rekurencja i drzewa binarne 201
Funkcje opakowujące 204
Kiedy należy wybierać rekurencję? 207
Argumenty przeciwko rekurencji 207
Ćwiczenia 211

7
ROZWIĄZYWANIE PROBLEMÓW
ZA POMOCĄ PONOWNEGO WYKORZYSTANIA KODU 213
Poprawne i niewłaściwe wykorzystanie kodu 214
Przegląd podstawowych informacji o komponentach 215

Kup ksi k
Pole ksi k

Blok kodu ... 215
Algorytmy ... 216
Wzorce ... 216
Abstrakcyjne typy danych ... 217
Biblioteki ... 218
Zdobywanie wiedzy o komponentach ... 219
Eksploracyjne zdobywanie wiedzy ... 219
Zdobywanie wiedzy w razie potrzeby ... 223
Wybór typu komponentu ... 232
Wybór komponentu w praktyce ... 234
Porównanie wyników ... 238
Ćwiczenia ... 239

8
MYŚLENIE JAK PROGRAMISTA ... 241
Tworzenie własnego planu głównego ... 242
Uwzględnienie mocnych i słabych stron ... 242
Budowanie planu głównego ... 248
Rozwiązywanie każdego problemu ... 250
Opracowywanie metody oszukiwania ... 252
Wymagane podzadania dla metody oszukiwania w grze wisielec ... 254
Wstępny projekt ... 256
Kodowanie wstępne ... 257
Analiza wstępnych wyników ... 266
Sztuka rozwiązywania problemów ... 267
Zdobywanie nowych umiejętności programistycznych ... 268
Nowe języki ... 269
Zdobywanie nowych umiejętności w języku, który już znasz ... 271
Nowe biblioteki ... 272
Bierz udział w szkoleniach ... 273
Podsumowanie ... 274
Ćwiczenia ... 275

SKOROWIDZ ... 276

Kup ksi k

Pole ksi k

4

Rozwiązywanie problemów za pomocą wskaźników i pamięci dynamicznej

W TYM ROZDZIALE NAUCZYMY SIĘ ROZWIĄZYWANIA PROBLEMÓW ZA POMOCĄ WSKAŹNIKÓW I PAMIĘCI DYNAMICZNEJ, CO POZWOLI NAM NA PISANIE UNIWERSALNYCH PROGRAMÓW ZARZĄDZAJĄCYCH DANYMI, KTÓRYCH ROZMIARY nie są znane przed uruchomieniem kodu. Wskaźniki i dynamiczne przydzielanie pamięci są programowaniem ekstremalnym. Gdy piszesz programy, które rezerwują pamięć w locie, przypisują ją do przydatnych struktur, a następnie przed zakończeniem działania sprzątają po sobie w taki sposób, by nic nie pozostało, nie jesteś po prostu kimś, kto potrafi trochę kodować — jesteś programistą.

Ponieważ wskaźniki są trudnym zagadnieniem, a wiele popularnych języków programowania, na przykład Java, obywa się bez nich, niektórzy początkujący programiści starają się przekonać samych siebie, że ten obszar wiedzy mogą zupełnie pominąć. Jest to błąd. Wskaźniki i niebezpośredni dostęp do pamięci będą

Kup ksi k Pole ksi k

zawsze używane w mechanizmach języków wysokiego poziomu. Aby więc rzeczywiście myśleć jak programista, musisz umieć swobodnie poruszać się wśród wskaźników oraz problemów wymagających ich użycia.

Zanim jednak zajmiemy się rozwiązywaniem problemów ze wskaźnikami, dokładnie przeanalizujemy wszystkie pierwszo- i drugoplanowe aspekty związane z ich działaniem. Dzięki temu osiągniemy dwie korzyści. Po pierwsze, zdobyta wiedza pozwoli wykorzystać wskaźniki w najbardziej wydajny sposób. Po drugie, poprzez ujawnienie tajemnic związanych ze wskaźnikami będziemy mogli ich bezpiecznie używać.

Podstawowe informacje o wskaźnikach

Podobnie jak w przypadku zagadnień omawianych w poprzednich rozdziałach, również tutaj zakładam, że masz już pewną podstawową wiedzę o wskaźnikach. Aby się jednak upewnić, że „nadajemy na tej samej fali”, przedstawię poniżej krótki przegląd ich możliwości.

Wskaźniki w języku C++ są reprezentowane przez znak gwiazdki (*). W zależności od kontekstu gwiazdka oznacza deklarację wskaźnika lub pamięć, do której się on odwołuje, a nie sam wskaźnik. Aby zadeklarować wskaźnik, umieszczamy znak gwiazdki między nazwą typu i identyfikatorem:

```
int *intPointer;
```

Powyższy kod zawiera deklarację zmiennej `intPointer`, będącej wskaźnikiem do typu `int`. Zwróć uwagę na to, że gwiazdka łączy się z identyfikatorem, a nie typem. W poniższej deklaracji zmienna `variable1` jest wskaźnikiem do typu `int`, a `variable2` po prostu zmienną typu `int`:

```
int *variable1, variable2;
```

Znak `&` przed zmienną działa jak operator *adresu*. Za pomocą poniższej instrukcji moglibyśmy więc przypisać adres zmiennej `variable2` do `variable1`:

```
variable1 = &variable2;
```

Możemy także bezpośrednio przypisać wartość jednej zmiennej wskaźnikowej do drugiej:

```
intPointer = variable1;
```


Kup ksi k Pole ksi k

Być może najważniejsze jest to, że w trakcie działania programu możemy przydzielać pamięć, która jest dostępna jedynie poprzez wskaźnik. Jest to możliwe przy użyciu operatora new:

```
double *doublePointer = new double;
```

Dostęp do pamięci wskazywanej przez wskaźnik jest zwany *wyłuskiwaniem* (*dereferencją*). Uzyskuje się go za pomocą gwiazdki umieszczonej po lewej stronie identyfikatora wskaźnika. Jak już wspomniałem, jest to ten sam sposób umieszczania znaku gwiazdki, jaki stosuje się podczas deklaracji wskaźnika, jednakże inny kontekst sprawia, że obie operacje różnią się od siebie. Oto przykład:

```
❶ *doublePointer = 35.4;
❷ double localDouble = *doublePointer;
```

Do obszaru pamięci typu double przydzielonego we wcześniejszym przykładzie przypisujemy wartość ❶, a następnie kopiujemy ją do zmiennej localDouble ❷.

Aby zwolnić niepotrzebną pamięć przydzieloną za pomocą operatora new, stosujemy słowo kluczowe delete:

```
delete doublePointer;
```

Sposób działania tego procesu został dokładnie przedstawiony w dalszej części rozdziału, w podrozdziale „Pamięć ma znaczenie”.

Korzyści z używania wskaźników

Wskaźniki dają możliwości niedostępne przy użyciu statycznego przydzielania pamięci, a także pozwalają na jej efektywne wykorzystanie. Trzy główne korzyści wynikające z używania wskaźników są następujące:

- Struktury danych o wielkości definiowanej w trakcie działania programu.
- Struktury danych o zmiennych rozmiarach.
- Współdzielenie pamięci.

Przyjrzyjmy się każdej z nich bardziej szczegółowo.

Struktury danych o wielkości definiowanej w trakcie działania programu

Dzięki użyciu wskaźników możemy utworzyć tablicę o rozmiarze ustalonym w trakcie działania programu, a nie definiować go jeszcze przed kompilacją aplikacji. Dzięki temu unikamy sytuacji ewentualnego braku miejsca w tablicy

Kup ksi k Pole ksi k

oraz definiowania jej zbyt dużego rozmiaru, a przez to najczęściej niewykorzystania większości przydzielonej pamięci. Z dynamicznym ustalaniem rozmiaru struktury danych mieliśmy już do czynienia w rozdziale 3., w podrozdziale „Kiedy należy używać tablic". Ta idea zostanie również wykorzystana w dalszej części niniejszego rozdziału, w podrozdziale „Łańcuchy o zmiennej długości".

Struktury danych o zmiennych rozmiarach

Możemy także stworzyć wskaźnikowe struktury danych, które w razie konieczności powiększają się lub zmniejszają w trakcie działania programu. Najprostszą strukturą danych o zmiennych rozmiarach jest lista powiązana, o której już wspominaliśmy. Do jej danych można uzyskać dostęp jedynie w sposób sekwencyjny. Rezerwuje ona jednak tylko tyle miejsca, ile wynosi rozmiar samych danych, bez żadnego marnowania pamięci. Jak zobaczysz w dalszej części książki, inne, bardziej złożone wskaźnikowe struktury danych zawierają opcje porządkowania i „profile", które umożliwiają lepsze zarządzanie powiązaniami między zapamiętanymi danymi, niż ma to miejsce w przypadku tablic. Z tego powodu, mimo że tablica umożliwia uzyskanie pełnego dostępu swobodnego, operacja *wyszukiwania* (podczas której w strukturze wyszukujemy element najlepiej spełniający określone kryterium) może być szybciej zrealizowana w strukturze opartej na wskaźnikach. Z tej właściwości skorzystamy w dalszej części tego rozdziału, aby stworzyć strukturę danych do przechowywania informacji o studentach, która powiększa się w razie konieczności.

Współdzielenie pamięci

Wskaźniki mogą poprawić wydajność programu poprzez umożliwienie współdzielenia bloków pamięci. Na przykład jeśli wywołujemy funkcję, za pomocą *parametrów referencji* możemy przekazać tylko wskaźnik do obszaru pamięci, zamiast kopiować cały blok. Najprawdopodobniej widziałeś już taką operację w praktyce. Referencje mają umieszczony znak `&` między typem i nazwą na liście parametrów formalnych:

```
void refParamFunction (int ❶ & x) {
   ❷ x = 10;
}

int number = 5;
refParamFunction(❸ number);
cout << ❹ number << "\n";
```

UWAGA *Spacje przed i po znaku `&` nie są wymagane. Umieściłem je wyłącznie ze względów estetycznych. W kodach innych programistów będziesz mógł zauważyć takie zapisy, jak `int& x`, `int &x`, a być może nawet `int&x`.*

Kup ksi k Pole ksi k

W powyższym kodzie formalny parametr x ❶ nie jest kopią argumentu number ❸, lecz odwołaniem do miejsca w pamięci, w którym przechowywana jest zmienna number. Dlatego też po zmianie x ❷ pamięć zajmowana przez zmienną number zostaje zmodyfikowana, a w wyniku tego program wyświetla wartość 10 ❹. Jak przedstawiono w powyższym przykładzie, parametry referencyjne mogą być używane w mechanizmach umożliwiających zwracanie wartości z funkcji. Mówiąc ogólniej, pozwalają one współdzielić ten sam obszar pamięci zarówno przez funkcję wywoływaną, jak i wywołującą, a przez to zmniejszać narzut. Jeśli zmienna będąca parametrem zajmuje 1 kilobajt pamięci, przekazanie jej jako referencji wymaga skopiowania tylko 32- lub 64-bitowego wskaźnika zamiast całego kilobajta. Poprzez zastosowanie słowa kluczowego const możemy poinformować, że używamy parametru referencyjnego w celu poprawy wydajności, a nie zwracania wartości:

```
int anotherFunction(const int & x);
```

Dzięki umieszczeniu słowa const przed deklaracją parametru referencyjnego x funkcja anotherFunction otrzyma referencję do przekazanego argumentu, lecz nie będzie mogła go zmodyfikować, podobnie jak dzieje się w przypadku każdego innego parametru zdefiniowanego jako const.

Mówiąc ogólnie, możemy używać wskaźników w powyższy sposób, aby pozwolić różnym częściom programu lub strukturom znajdującym się w nim na dostęp do tego samego obszaru danych bez potrzeby jego kopiowania.

Kiedy należy używać wskaźników?

Podobnie jak tablice, również wskaźniki mają potencjalne wady i powinny być używane jedynie w odpowiednich sytuacjach. W jaki sposób możemy wiedzieć, kiedy użycie wskaźnika jest odpowiednie? Znając już zalety stosowania wskaźników, możemy stwierdzić, że powinny być one używane jedynie wówczas, gdy należy wziąć pod uwagę co najmniej jedną z korzyści przez nie oferowanych. Jeśli Twój program wymaga złożonej struktury w celu przechowywania danych, lecz przed jego uruchomieniem nie możesz dokładnie oszacować, ile miejsca ona zajmie; jeśli potrzebujesz struktury, która może się powiększać i zmniejszać w trakcie działania aplikacji; jeśli obsługujesz duże obiekty lub inne bloki danych, które są przekazywane w różnych miejscach programu — wówczas powinieneś zastosować wskaźniki. W przypadku gdy nie masz do czynienia z którąś z wymienionych sytuacji, należy jednak unikać użycia wskaźników i dynamicznego przydzielania pamięci.

Znając złą sławę wskaźników, uważanych za jedną z najtrudniejszych cech języka C++, mógłbyś stwierdzić, że żaden z programistów nie spróbuje nawet ich użyć, gdy nie będzie to niezbędne. Byłem jednak wielokrotnie zaskoczony sytuacją zupełnie odwrotną. Czasem programiści po prostu oszukują samych

Kup ksi k Pole ksi k

siebie, starając się udowodnić, że użycie wskaźników jest konieczne. Wyobraź sobie, że wywołujesz funkcję napisaną przez kogoś innego, na przykład znajdującą się w bibliotece lub interfejsie programowania aplikacji. Używasz w tym celu następującego prototypu:

```
void compute(int input, int *output);
```

Moglibyśmy założyć, że ta funkcja została napisana w języku C, a nie C++, dlatego używa wskaźnika zamiast referencji (&), by zdefiniować parametr wyjściowy. Nierozważny programista mógłby wywołać tę funkcję w poniższy sposób:

```
int num1 = 10;
int *num2 = new int;
compute(num1, num2);
```

Powyższy kod jest niewydajny pamięciowo, ponieważ tworzy wskaźnik, mimo że nie jest to wymagane. Oprócz dwóch zmiennych całkowitych rezerwuje on miejsce na dodatkowy wskaźnik. Kod jest także niewydajny czasowo, ponieważ zbędna alokacja pamięci zajmuje czas procesora (jak wyjaśniono w następnym podrozdziale). Tego wszystkiego można uniknąć poprzez zastosowanie innego aspektu operatora &, który pozwoli na uzyskanie adresu dla statycznie zadeklarowanej zmiennej, na przykład:

```
int num1 = 10;
int num2;
compute(num1, &num2);
```

Mówiąc dokładniej, mimo że w kolejnej wersji naszego kodu wciąż stosujemy wskaźnik, używamy go jednak w sposób domyślny, nie tworząc nowej zmiennej wskaźnikowej ani nie deklarując dynamicznej pamięci.

Pamięć ma znaczenie

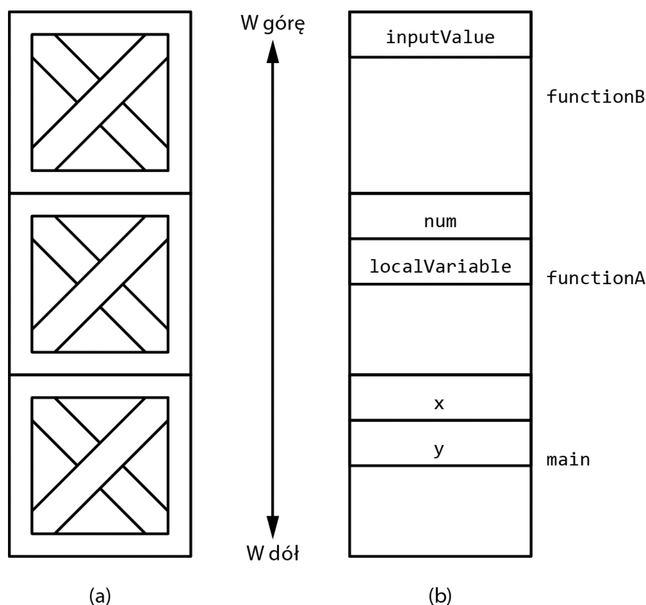
Aby zrozumieć, w jaki sposób dynamiczne przydzielanie pamięci pozwala nam na powiększanie i zmniejszanie zasobów w czasie działania programu, powinniśmy dowiedzieć się więcej o tym, jak ogólnie działa alokacja pamięci. Według mnie jest to jedno z zagadnień, z których mogą skorzystać początkujący programiści uczący się języka C++. Wszyscy programiści muszą w końcu zrozumieć, jak działają systemy pamięci w nowoczesnym komputerze, a język C++ po prostu zmusza Cię do stanięcia twarzą w twarz z tym zagadnieniem. Inne języki tak dokładnie ukrywają szczegóły zarządzania pamięcią, że nowicjusze przekonują samych siebie, iż ta wiedza ich nie dotyczy, co jednak nie jest prawdą.

Kup ksi k Pole ksi k

Szczegóły może nie są ważne, dopóki wszystko działa prawidłowo. Gdy jednak tylko pojawiają się kłopoty, ignorowanie niskopoziomowych modeli pamięci powoduje powstanie pomiędzy programistą i rozwiązaniem problemu przeszkód nie do pokonania.

Stos i sterta

Język C++ alokuje pamięć w dwóch miejscach: na *stosie* (ang. *stack*) i na *stercie* (ang. *heap*). Jak wynika z samych nazw, stos jest uporządkowany i schludny, a sterta chaotyczna i niechlujna. Nazwa „stos" jest szczególnie opisowa, ponieważ pozwala wyobrazić sobie metodę zwartego przydzielania pamięci. Załóżmy, że masz do czynienia ze stosem skrzyń przedstawionym na rysunku 4.1 (a). Gdy musisz przechować nową skrzynię w magazynie, umieszczasz ją na samej górze stosu. Aby usunąć określoną skrzynię, zdejmujesz najpierw te, które są nad nią. W praktycznej terminologii programowania oznacza to, że po przydzieleniu bloku pamięci (skrzyni) na stosie nie można zmienić jego rozmiaru, ponieważ w dowolnym momencie mogą istnieć inne obszary, znajdujące się bezpośrednio za nim (inne skrzynie nad nią).



Rysunek 4.1. Stos skrzyń i stos wywołań funkcji

W języku C++ możesz jawnie stworzyć własny stos, by użyć go w określonym algorytmie, lecz bez względu na to w systemie istnieje jeden stos, który zawsze będzie używany przez Twój program. Jest on zwany *stosem wywołań* (ang. *runtime stack*). Za każdym razem gdy wywołana zostaje funkcja (oznacza to także funkcję `main`), na szczycie stosu wywołań zarezerwowany zostaje blok pamięci. Jest on zwany *rekordem aktywacji* (ang. *activation record*). Pełna analiza jego

Kup ksi k Pole ksi k

zawartości wykracza poza ramy tej książki, lecz będąc osobą zajmującą się rozwiązywaniem problemów, powinieneś wiedzieć, że podstawowym zadaniem rekordu aktywacji jest udostępnianie miejsca do przechowywania zmiennych. Jest w nim przydzielana pamięć dla wszystkich zmiennych lokalnych, włącznie z parametrami funkcji. Przyjrzyjmy się następującemu przykładowi:

```
int functionB(int inputValue) {
   ❶ return inputValue - 10;
}
int functionA(int num) {
   int localVariable = functionB(num * 10);
   return localVariable;
}
int main()
{
   int x = 12;
   int y = functionA(x);
   return 0;
}
```

W powyższym kodzie funkcja `main` wywołuje `functionA`, która z kolei wywołuje `functionB`. Na rysunku 4.1 (b) przedstawiono uproszczoną wersję stanu stosu wywołań zaraz przed wykonaniem instrukcji powrotu z funkcji `functionB` ❶. Rekordy aktywacji dla wszystkich trzech funkcji zostały umieszczone na stosie zwartej pamięci, przy czym funkcja `main` znajduje się na jego samym dole (aby jeszcze bardziej skomplikować zagadnienie, chciałbym dodać, że możliwe jest, by — przeciwnie do naturalnego zachowania — stos zaczynał się od najwyższego możliwego adresu w pamięci, a następnie rósł w kierunku niższych adresów. Nic nie stoi jednak na przeszkodzie, by zignorować tę możliwość). Logicznie rozumując, rekord aktywacji funkcji `main` znajduje się na dole stosu, rekord aktywacji funkcji `functionA` nad nim, a rekord aktywacji funkcji `functionB` na samej górze stosu. Żaden z dwóch niższych rekordów aktywacji nie może zostać usunięty, zanim nie zlikwidujemy rekordu aktywacji dla funkcji `functionB`.

Stos jest dokładnie uporządkowany, lecz w przeciwieństwie do niego sterta jest dość chaotyczna. Wyobraź sobie, że znów musisz przechowywać skrzynie, lecz tym razem są one delikatne i nie możesz układać jednej na drugiej. Na początku dysponujesz dużym, pustym pomieszczeniem i możesz w nim układać skrzynie w dowolnym miejscu na podłodze. Są one jednak ciężkie, więc po ich ułożeniu na podłodze pozostają tam aż do momentu, gdy trzeba się będzie ich pozbyć. W porównaniu ze stosem, taki system ma pewne zalety i wady. Z jednej strony jest uniwersalny i pozwala w dowolnym momencie uzyskać dostęp do zawartości każdej ze skrzyń. Z drugiej strony jednak w pomieszczeniu szybko zapanuje nieporządek. Jeśli skrzynie mają różne rozmiary, szczególnie trudno będzie wykorzystać każdą wolną przestrzeń dostępną na podłodze. W końcu

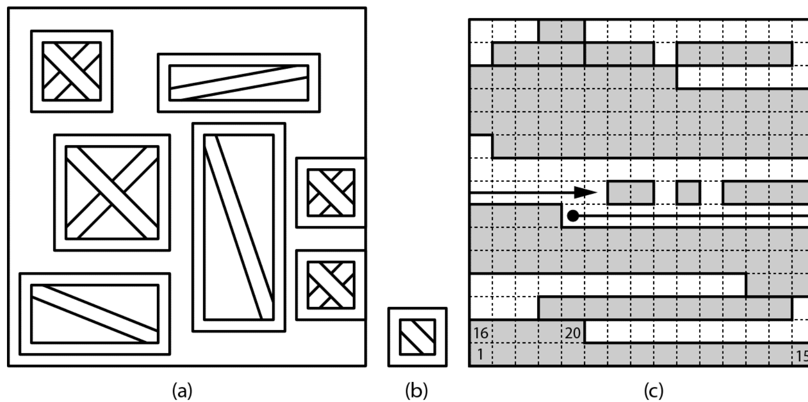
Kup ksi k Pole ksi k

między skrzyniami będzie istnieć mnóstwo wolnych miejsc, które będą jednak miały zbyt małą powierzchnię, by umieścić na nich kolejny element. Ponieważ skrzynie nie mogą być w prosty sposób przemieszczane, usuwanie niektórych z nich spowoduje, że pojawią się raczej kolejne, trudne do zlikwidowania wolne przestrzenie, a nie obszar, który byłby uporządkowany i zbliżony wyglądem do naszej początkowej pustej podłogi. W praktycznej terminologii programowania nasza sterta jest jak podłoga w opisanym właśnie pomieszczeniu. Blok pamięci jest zwartym ciągiem adresów. W trakcie działania programu, który wykonuje wiele rezerwacji i zwolnień pamięci, pojawia się mnóstwo wolnych przestrzeni między zaalokowanymi obszarami. Problem ten jest znany jako *fragmentacja pamięci* (ang. *memory fragmentation*).

Każdy program ma własną stertę, w której można dynamicznie przydzielać pamięć. W języku C++ oznacza to zazwyczaj użycie słowa kluczowego `new`, lecz możesz także stosować standardowe funkcje języka C służące do alokowania pamięci, takie jak `malloc`. Każde wywołanie `new` (lub `malloc`) rezerwuje odpowiedni obszar na stercie i zwraca wskaźnik do niego, natomiast każde wywołanie `delete` (lub `free` w przypadku, gdy do alokacji użyto `malloc`) powoduje, że jest on zwalniany do puli dostępnej pamięci. Z powodu fragmentacji nie wszystkie obszary pamięci na stercie są równie użyteczne. Jeśli nasz program rozpoczyna swoje działanie od przydzielenia pamięci na stercie dla zmiennych A, B i C, możemy się spodziewać, że te trzy bloki będą zwarte. Jeśli jednak zwolnimy pamięć dla zmiennej B, pojawi się wolne miejsce, które może zostać użyte jedynie przez żądanie wymagające rezerwacji obszaru o rozmiarze B lub mniejszym, chyba że zostanie również zwolniona pamięć dla zmiennej A lub C.

Omawianą sytuację wyjaśniono na rysunku 4.2. W części (a) widzimy podłogę naszego pomieszczenia z chaotycznie umieszczonymi skrzyniami. W pewnym momencie pomieszczenie było prawdopodobnie uporządkowane, lecz z upływem czasu skrzynie były umieszczane w sposób przypadkowy. Obecnie mamy niewielką skrzynię (b), która jednak nie może zmieścić się na żadnym wolnym miejscu na podłodze, mimo że całkowita nieużywana powierzchnia jest dużo większa od powierzchni samej skrzyni. W części (c) przedstawiliśmy niewielką stertę. Linie przerywane oznaczają najmniejsze (niepodzielne) fragmenty pamięci, które w zależności od rodzaju menedżera sterty mogłyby być pojedynczym bajtem, słowem lub jeszcze czymś większym. Obszary zacienione reprezentują rezerwacje zwartej pamięci. Jeden z nich w celu lepszej klarowności zawiera pewne fragmenty, które zostały ponumerowane. Podobnie jak nieuporządkowana podłoga, również stos z fragmentacją zawiera wiele oddzielnych fragmentów nieprzydzielonej pamięci, co zmniejsza ich przydatność. Mamy 85 nieużywanych fragmentów pamięci, lecz największy zwarty obszar, wskazywany przez symbol strzałki, składa się tylko z 17 elementów. Inaczej mówiąc, jeśli każdy z fragmentów byłby bajtem, ta sterta nie mogłaby pozwolić na wykonanie polecenia `new` przydzielającego obszar pamięci większy od 17 bajtów, mimo że całkowita ilość wolnego miejsca wynosi 85 bajtów.

Kup ksi k Pole ksi k



Rysunek 4.2. Nieuporządkowana podłoga, skrzynia, której nie można na niej umieścić, a także pamięć z fragmentacją

Rozmiar pamięci

Pierwszym praktycznym zagadnieniem związanym z pamięcią jest ograniczanie jej użycia tylko do tego, co niezbędne. Nowoczesne systemy komputerowe mają tak dużo pamięci, że łatwo można założyć, iż jej pojemność jest nieskończona, lecz w rzeczywistości każdy z programów ma dostęp do jej ograniczonej ilości. Programy również muszą używać pamięci w sposób efektywny, aby uniknąć spowolnienia działania całego systemu. W wielozadaniowym systemie operacyjnym (co oznacza praktycznie każdy nowoczesny system) każdy bajt pamięci zmarnowany przez dany program zwiększa prawdopodobieństwo pojawienia się sytuacji, w której zestaw działających aplikacji nie ma już wystarczającej ilości pamięci do poprawnej pracy. W tym momencie system operacyjny zaczyna w sposób ciągły kopiować fragmenty pamięci z jednego programu do drugiego, powodując znaczące spowolnienie działania aplikacji. Taka sytuacja zwana jest *szamotaniem* (ang. *thrashing*).

Zauważ, że poza koniecznością utrzymania jak najmniejszego rozmiaru dla całego programu należy również dążyć do tego, by stos i sterta były jak największe. Aby to udowodnić, zacznijmy rezerwować pamięć na stercie po jednym kilobajcie za każdym razem, aż do momentu gdy coś przestanie działać:

```
const int intsPerKilobyte = 1024 / sizeof(int);
while (true) {
   int *oneKilobyteArray = new int[intsPerKilobyte];
}
```

Chciałbym zaznaczyć, że jest to okropny kod, napisany wyłącznie w celu zademonstrowania problemu. Jeśli chciałbyś sprawdzić go w swoim systemie, sugeruję, abyś wcześniej — po prostu dla bezpieczeństwa — zapisał wszystkie dane, na których pracujesz. Program zawiesi się, a Twój system operacyjny poinformuje, że kod wywołał wyjątek `bad_alloc`, lecz nie mógł go obsłużyć. Jest on

Kup ksi k Pole ksi k

zgłaszany przez instrukcję `new`, gdy żaden blok nieprzydzielonej pamięci na stercie nie jest na tyle duży, by można było poprawnie obsłużyć żądanie. Brak miejsca na stercie jest zwany *przepełnieniem sterty* (ang. *heap overflow*). W niektórych systemach taka sytuacja może się zdarzać często, a w innych błędne działanie programu może spowodować pojawienie się długotrwałego stanu szamotania, zanim zostanie zgłoszony wyjątek `bad_alloc` (w moim systemie użycie instrukcji `new` nie powiodło się dopiero wówczas, gdy za pomocą wcześniejszych wywołań zaalokowałem 2 gigabajty pamięci).

Podobna sytuacja zdarza się w przypadku stosu wywołań. Każde wywołanie funkcji rezerwuje miejsce na stosie, a oprócz tego dla każdego rekordu aktywacji istnieje stały narzut, nawet w przypadku funkcji nieposiadającej parametrów lub lokalnych zmiennych. Najprostszym sposobem zademonstrowania takiej sytuacji jest niekontrolowane wywołanie funkcji rekurencyjnej:

```
❶ int count = 0;
void stackOverflow() {
   ❷ count++;
   ❸ stackOverflow();
}
int main()
{
   ❹ stackOverflow();
   return 0;
}
```

Powyższy kod zawiera zmienną globalną ❶, co w większości przypadków świadczy o złym stylu programowania. W tej sytuacji potrzebujmy jednak zmiennej, która będzie istnieć podczas wszystkich wywołań rekurencyjnych. Ponieważ jest ona zadeklarowana poza funkcją, nie ma potrzeby rezerwowania dla niej miejsca w rekordzie aktywacji. Nie istnieją również żadne inne lokalne zmienne ani parametry. Funkcja tylko zwiększa licznik ❷, a następnie wykonuje wywołanie rekurencyjne ❸. Rekurencja zostanie dokładnie omówiona w rozdziale 6., lecz w tym przykładzie jest ona używana tylko w celu stworzenia możliwie najdłuższego łańcucha wywołań funkcji. Rekord aktywacji funkcji pozostaje na stosie do momentu jej zakończenia. Gdy więc z funkcji `main` zostaje po raz pierwszy wywołana funkcja `stackOverflow` ❹, rekord aktywacji jest umieszczany na stosie wywołań i nie może zostać usunięty, dopóki nie zakończy się jej działanie. Nie zdarzy się to jednak nigdy, ponieważ funkcja wykonuje kolejne wywołanie `stackOverflow`, umieszczając następny rekord aktywacji na stosie, następnie tworzy trzecie wywołanie itd. Liczba rekordów aktywacji rośnie na stosie aż do momentu, gdy zaczyna brakować na nim miejsca. W moim systemie wartość `count` wynosiła około 4900, gdy program przestał działać. Środowisko projektowe Visual Studio, w którym pracuję, domyślnie przydziela programowi 1 MB pamięci na stosie, co oznacza, że każde wywołanie funkcji, nawet takiej, która nie ma lokalnych zmiennych ani parametrów, tworzy rekord aktywacji o wielkości ponad 200 bajtów.

Kup ksi k
Pole ksi k

Czas życia

Czas życia (ang. *lifetime*) zmiennej to okres między jej alokacją a zwolnieniem pamięci. W przypadku zmiennej korzystającej ze stosu, czyli zmiennej lokalnej lub parametru, czas życia jest zarządzany w sposób domyślny. Zmienna zostaje zaalokowana podczas wywołania funkcji, a usunięta po jej zakończeniu. W przypadku zmiennej korzystającej ze sterty, czyli dynamicznie zaalokowanej przy użyciu instrukcji `new`, czas jej życia jest w naszych rękach. Zarządzanie czasem życia dynamicznie alokowanych zmiennych jest zmorą każdego programisty C++. Najczęściej znanym problemem jest groźny *wyciek pamięci*, czyli sytuacja, w której pamięć zostaje przydzielona na stercie, lecz nigdy nie jest zwolniona ani wykorzystana przez żaden wskaźnik. Oto prosty przykład:

```
❶ int *intPtr = new int;
❷ intPtr = NULL;
```

W powyższym kodzie deklarujemy wskaźnik do liczby całkowitej ❶, inicjalizując go poprzez przydzielenie odpowiedniego miejsca na stercie. Następnie w drugim wierszu do tego wskaźnika przypisujemy wartość `NULL` ❷ (która jest po prostu innym identyfikatorem liczby zero). Wartość całkowita, którą zaalokowaliśmy za pomocą instrukcji `new`, wciąż jednak istnieje. Samotna i porzucona znajduje się na swoim miejscu na stercie, czekając na operację zwolnienia pamięci, która nigdy nie wystąpi. Nie możemy zwolnić pamięci dla zaalokowanej zmiennej całkowitej, ponieważ taka operacja wymaga użycia słowa kluczowego `delete` z parametrem będącym wskaźnikiem do usuwanego bloku, a my już nim nie dysponujemy. Jeśli chcielibyśmy jednak wykonać polecenie `delete intPtr`, zakończyłoby się ono błędem, ponieważ wartość `intPrt` jest równa zeru.

Czasem zamiast problemu z pamięcią, której nie można zwolnić, mamy do czynienia z czymś odwrotnym. Jest to próba zwolnienia tej samej pamięci dwukrotnie, co powoduje powstanie błędu czasu wykonania. Wydaje się, że takiej sytuacji można łatwo uniknąć: wystarczy nie wywoływać dwukrotnie instrukcji `delete` dla tej samej zmiennej. Jest to jednak bardziej skomplikowane, ponieważ możemy mieć wiele zmiennych wskazujących na ten sam obszar pamięci. Jeśli wiele zmiennych wskazuje na ten sam blok pamięci i zostanie wykonane polecenie `delete` dla jednej z nich, wówczas w rzeczywistości zwolnimy pamięć dla wszystkich. Jeśli nie przypiszemy im jawnie wartości `NULL`, wówczas będziemy mieli do czynienia ze *wskaźnikami zawieszonymi*, dla których wywołanie `delete` będzie powodować pojawienie się błędu czasu wykonania.

Kup ksi k Pole ksi k

Rozwiązywanie problemów za pomocą wskaźników

W tym momencie jesteś już gotowy do rozwiązywania problemów, dlatego przyjrzyjmy się kilku z nich i spróbujmy użyć wskaźników oraz dynamicznego przydzielania pamięci, by je rozwiązać. Najpierw zajmiemy się dynamicznie alokowanymi tablicami, co pozwoli na zaprezentowanie metody kontrolowania pamięci sterty podczas ich używania. Następnie zanurzymy się w prawdziwie dynamicznej strukturze.

Łańcuchy o zmiennej długości

W pierwszym problemie zamierzamy stworzyć funkcje do obsługi łańcuchów tekstowych. Ten termin jest tu użyty w najbardziej ogólnym znaczeniu — jako sekwencja znaków, bez względu na sposób ich przechowywania. Załóżmy, że nasz typ łańcuchowy powinien być wspierany przez trzy funkcje.

PROBLEM: OBSŁUGA ŁAŃCUCHÓW O ZMIENNEJ DŁUGOŚCI

Stwórz implementację opartą na stercie dla trzech funkcji obsługujących łańcuchy:

- **`append`** — funkcja wymaga podania łańcucha i znaku, a w wyniku swojego działania dołącza ten znak do końca łańcucha.
- **`concatenate`** — funkcja używa dwóch łańcuchów i dołącza znaki z drugiego do pierwszego.
- **`characterAt`** — funkcja wymaga użycia łańcucha oraz odpowiedniej liczby, zwracając znak znajdujący się na wskazywanej przez nią pozycji w łańcuchu (pierwszy znak łańcucha ma indeks równy zeru).

Napisz kod, zakładając, że funkcja `characterAt` będzie wywoływana często, a dwie pozostałe stosunkowo rzadko. Względna wydajność operacji powinna odzwierciedlać częstotliwość ich stosowania.

W tym przypadku powinniśmy zaprojektować taką reprezentację łańcucha, która pozwoli na szybkie wykonywanie funkcji `characterAt`, co oznacza, że musimy wymyśleć szybki sposób na uzyskanie dostępu do dowolnego znaku. Jak prawdopodobnie pamiętasz z poprzedniego rozdziału, to jest właśnie najlepsza zaleta tablic — dostęp swobodny. Rozwiążmy więc nasz problem przy użyciu tablic typu `char`. Funkcje `append` i `concatenate` zmieniają rozmiar łańcucha, co oznacza, że napotkamy tu wszystkie rodzaje problemów dotyczących tablic, o których już wspominaliśmy. Ponieważ w problemie nie zdefiniowano ograniczenia dotyczącego wielkości łańcucha, nie możemy określić początkowego rozmiaru dla naszych tablic i mieć następnie nadzieję, że wszystko będzie działać. Zamiast tego musimy zmieniać go w trakcie działania programu.

Kup ksi k Pole ksi k

Aby rozpocząć, zdefiniujmy nazwę typu dla naszego łańcucha za pomocą słowa kluczowego `typedef`. Wiemy, że tablice będziemy tworzyć w sposób dynamiczny, więc musimy zaprojektować nasz typ łańcuchowy jako wskaźnik do `char`:

```
typedef char * arrayString;
```

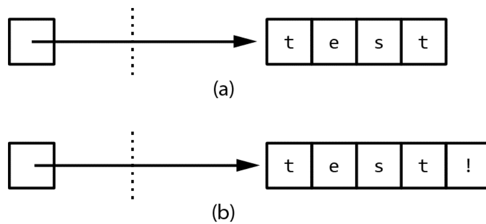
Mając gotowy typ, zajmijmy się funkcjami. Wykorzystując zasadę rozpoczynania od tego, co już potrafimy zrobić, napiszmy szybko funkcję `characterAt`:

```
char characterAt(arrayString s, int position) {
   ❶ return s[position];
}
```

Jak pamiętasz z rozdziału 3., gdy wskaźnikowi został przypisany adres tablicy, możemy uzyskać dostęp do jej elementów, wykorzystując w tym celu zwykłą notację tablicową ❶. Zauważ jednak, że w przypadku gdy wartość `position` nie będzie poprawnym indeksem tablicy `s`, mogą pojawić się problemy. Dodatkowo powyższy kod przerzuca odpowiedzialność za kontrolowanie poprawności drugiego parametru na funkcję wywołującą. Inne rozwiązania tego zagadnienia zostaną przedstawione w ćwiczeniach. W chwili obecnej zajmijmy się funkcją `append`. Możemy sobie ogólnie wyobrazić, co powinna ona wykonywać, lecz aby poznać szczegóły, powinniśmy rozważyć użycie przykładu. Jest to technika, którą nazywam *rozwiązywaniem poprzez przykład*. Rozpocznij od zdefiniowania nietrywialnego zestawu danych wejściowych dla funkcji lub programu. Zapisz wszystkie szczegóły związane z danymi wejściowymi, a także te, które dotyczą wyników. Gdy będziesz tworzył kod, zajmiesz się ogólnym przypadkiem, a także dokładnie sprawdzisz, w jaki sposób na każdym etapie działania programu zmodyfikowane zostają dane wejściowe, aby upewnić się, że osiągasz wymagany stan wyjściowy. Ta metoda jest szczególnie przydatna podczas pracy ze wskaźnikami i dynamicznie przydzielaną pamięcią, ponieważ w takich przypadkach program działa w większości poza bezpośrednią kontrolą. Śledzenie stanu na papierze zmusza Cię do obserwowania wszystkich zmieniających się wartości w pamięci — nie tylko tych, które są bezpośrednio reprezentowane przez zmienne, lecz także samej sterty.

Załóżmy, że rozpoczniemy od łańcucha `test`, co oznacza, że na stercie mamy tablicę zawierającą znaki `t`, `e`, `s` `i` `t` — w tej właśnie kolejności, a przy użyciu funkcji `append` chcielibyśmy dodać do niej znak wykrzyknika. Na rysunku 4.3 zaprezentowano stan pamięci przed (a) i po (b) wykonaniu tej operacji. To, co znajduje się po lewej stronie pionowej, kropkowanej linii, jest pamięcią stosu (lokalnymi zmiennymi lub parametrami). Wszystko po jej prawej stronie dotyczy pamięci na stercie, dynamicznie przydzielonej za pomocą słowa kluczowego `new`.

Kup ksi k Pole ksi k



Rysunek 4.3. Proponowane stany pamięci przed i po wywołaniu funkcji append

Spoglądając na rysunek, od razu widzę ewentualny problem, który może się pojawić w naszej funkcji. Wykorzystując zdefiniowaną przez nas implementację łańcuchów tekstowych, funkcja zamierza stworzyć nową tablicę, mającą liczbę elementów większą o jeden od pierwotnej, a następnie skopiować wszystkie znaki z pierwszej tablicy do drugiej. Lecz w jaki sposób będziemy wiedzieć, jaki rozmiar ma pierwsza tablica? Z poprzedniego rozdziału wiemy, że powinniśmy sami śledzić wielkość naszych tablic. Tak więc czegoś tu brakuje.

Jeśli mamy jakieś doświadczenie w pracy z łańcuchami ze standardowej biblioteki języków C i C++, wiemy już, czym jest brakujący element. Jeśli nie, możemy go szybko znaleźć. Pamiętaj, że jedną z naszych technik rozwiązywania problemów jest *poszukiwanie analogii*. Być może powinniśmy się zastanowić nad innymi problemami, w których wiedza o wielkości jakiegoś składnika jest nieznana. W rozdziale 2. podczas rozwiązywania problemu walidacji sumy kontrolnej Luhna przetwarzaliśmy numery identyfikacyjne o dowolnej liczbie cyfr. Nie wiedzieliśmy wówczas, ile cyfr wprowadzi użytkownik, jednak napisaliśmy pętlę `while` działającą aż do momentu, w którym nie został wczytany ostatni znak, czyli koniec wiersza.

Niestety, na końcu naszych tablic nie ma znaku końca wiersza. Ale dlaczego nie moglibyśmy *umieścić* tego znaku jako ostatniego elementu we wszystkich naszych tablicach znakowych? Wówczas będziemy mogli wyznaczyć długość łańcucha w taki sam sposób, jak ustalaliśmy liczbę cyfr w numerze identyfikacyjnym. Jedyną wadą tej metody jest to, że w naszych łańcuchach, z wyjątkiem ich zakończenia, nie moglibyśmy już używać znaków końca wiersza. Nie jest to zbyt wielkie ograniczenie i zależy od sytuacji, w której wykorzystamy nasze łańcuchy. Dla uzyskania maksymalnej uniwersalności powinniśmy jednak wybrać taką wartość, która nie będzie kolidować z żadnym znakiem możliwym do zastosowania przez użytkownika. Do oznaczania końca naszych łańcuchów wybierzemy więc zero, ponieważ reprezentuje ono wartość pustą w ASCII i innych systemach kodowania znaków. Jest to dokładnie taka sama metoda, którą zastosowano w standardowej bibliotece dla języków C i C++.

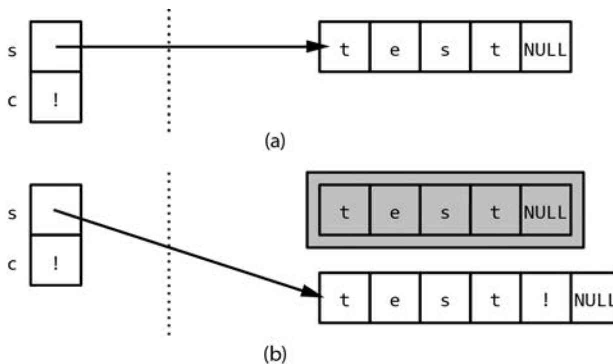
Po rozwiązaniu powyższego problemu możemy się bardziej skoncentrować na tym, co powinna robić funkcja `append` z naszymi przykładowymi danymi. Wiemy, że będzie ona miała dwa parametry: pierwszy typu `arrayString` ma być wskaźnikiem do łańcucha znaków na stercie, a drugi typu `char` powinien być znakiem dołączanym do niego. Aby zbytnio nie komplikować, weźmy się do pracy i stwórzmy szkic funkcji `append`, a także kod, który powinien ją testować:

Kup ksi k Pole ksi k

```
void append(❶ arrayString& s, char c) {
}
void appendTester() {
   ❷ arrayString a = new char[5];
   ❸ a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] =  0;
   ❹ append(a, '!')
   ❺ cout << a << "\n";
}
```

Funkcja `appendTester` przydziela pamięć na stercie dla naszego łańcucha ❷. Zwróć uwagę na to, że rozmiar tablicy wynosi 5, co jest niezbędne, ponieważ musimy przypisać cztery litery słowa `test` oraz kończący łańcuch znak pusty ❸. Następnie wywołujemy funkcję `append` ❹, która w tym momencie nie zawiera żadnej logiki. Podczas pisania nagłówka funkcji zdałem sobie sprawę, że parametr typu `arrayString` powinien być referencją (`&`) ❶, ponieważ funkcja zamierza utworzyć na stercie nową tablicę. Wynika stąd, że wartość zmiennej `a`, która jest przekazywana do funkcji `append`, zostanie zmieniona po jej zakończeniu, gdyż musi wskazywać na nowy łańcuch. Zauważ, że ze względu na to, iż nasze tablice używają kończącego znaku pustego stosowanego w bibliotekach standardowych, możemy przekazać tablicę poprzez wskaźnik bezpośrednio do strumienia wyjściowego, aby sprawdzić jej wartość ❺.

Na rysunku 4.4 pokazano analizę tego, co powinna zrobić funkcja z danymi testowymi. Znaki kończące tablicę znajdują się we właściwych miejscach; dla większej czytelności oznaczono je jako `NULL`. W stanie (b) oczywiste jest, że zmienna `s` wskazuje na nowy blok pamięci. Poprzednia tablica jest obecnie przedstawiona w kolorze szarym. W tego typu rysunkach stosuję taki kolor w celu poinformowania, że dany obszar pamięci został zwolniony. Umieszczanie pamięci przydzielonej w naszych diagramach pozwala nie zapomnieć o jej zwalnianiu.

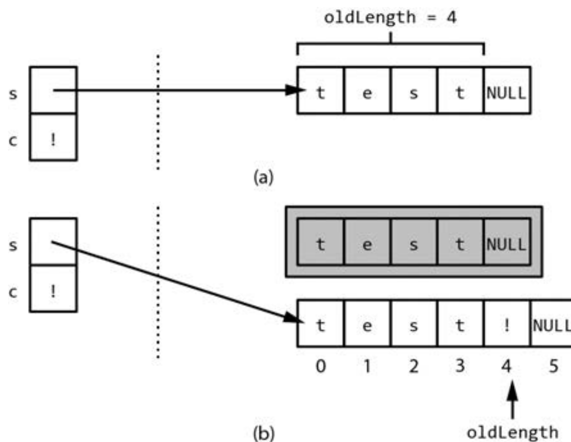


Rysunek 4.4. Zaktualizowane i uzupełnione stany pamięci przed (a) i po (b) wywołaniu funkcji append

Mając wszystko właściwie narysowane, możemy napisać samą funkcję:

```
void append(arrayString& s, char c) {
   int oldLength = 0;
   ❶ while (s[oldLength] != 0) {
      oldLength++;
   }
   ❷ arrayString newS = new char[oldLength + 2];
   ❸ for (int i = 0; i < oldLength; i++) {
      newS[i] = s[i];
   }
   ❹ newS[oldLength] = c;
   ❺ newS[oldLength + 1] = 0;
   ❻ delete[] s;
   ❼ s = newS;
}
```

W kodzie dzieje się wiele rzeczy, więc przeanalizujmy go krok po kroku. Na początku funkcji mamy pętlę, która odszukuje znak `NULL` kończący tablicę ❶. Po jej zakończeniu zmienna `oldLength` będzie zawierać liczbę właściwych znaków w naszej tablicy (to znaczy bez kończącego ją znaku pustego). Na stercie przydzielamy miejsce dla nowej tablicy, która będzie miała rozmiar `oldLength + 2` ❷. Jest to jeden ze szczegółów, którego nie można łatwo zrozumieć, gdy starasz się to zrobić tylko w swoim umyśle, lecz staje się to proste po przedstawieniu na rysunku. Podążając za kodem i korzystając z przykładu przedstawionego na rysunku 4.5, możemy zauważyć, że wartość `oldLength` powinna być równa 4. Wiemy, że tak ma być, ponieważ słowo `test` składa się z czterech znaków. Nowa tablica, przedstawiona w części (b), wymaga alokacji sześciu znaków, gdyż musimy mieć dodatkowe miejsce na dołączany wykrzyknik oraz kończący znak pusty.



Rysunek 4.5. Zależności między zmienną lokalną, parametrami i przydzieloną pamięcią przed i po wykonaniu funkcji append

Kup ksi k Pole ksi k

Mając już przydzieloną pamięć, kopiujemy wszystkie znaki właściwe ze starej tablicy do nowej ❸, a następnie w odpowiednich miejscach umieszczamy dołączany ❹ oraz kończący znak `NULL` ❺. Przypomnę jeszcze raz, że stosowanie diagramu pozwala na zachowanie porządku. Aby wszystko wyglądało jeszcze prościej, na rysunku 4.5 zaprezentowano sposób wyznaczania wartości `oldLength`, a także pozycję, którą wskazuje ta zmienna w nowej tablicy. Dzięki tej metodzie można łatwo określić poprawne indeksy dla obu instrukcji przypisania.

Ostatnie trzy wiersze w funkcji `append` dotyczą szarego prostokąta znajdującego się na rysunku w części (b). Aby uniknąć wycieku pamięci, musimy zwolnić pamięć na stercie dla tablicy, na którą pierwotnie wskazywał parametr `s` ❻. Następnie przypisujemy mu adres nowej, większej tablicy ❼. Niestety, jednym z powodów tego, że sytuacje wycieku pamięci są tak często spotykane podczas programowania w języku C++, jest to, iż dopóki całkowita wielkość takiej pamięci nie jest duża, system nie zgłasza żadnych problemów. Wynika stąd, że wyciek pamięci może zostać zupełnie niezauważony przez programistę podczas przeprowadzania testów programu. Jako programiści musimy więc być staranni i zawsze brać pod uwagę czas życia bloków pamięci przydzielonych na stercie. Za każdym razem gdy używasz słowa kluczowego `new`, pomyśl o tym, gdzie i kiedy pojawi się odpowiednie słowo `delete`.

Zauważ, że wszystko, co zawarliśmy w funkcji, wynika bezpośrednio z naszych diagramów. Skomplikowane programowanie staje się dużo prostsze przy użyciu dobrze zaprojektowanych rysunków. Chciałbym, aby początkujący programiści poświęcali trochę czasu na ich stworzenie przed rozpoczynaniem właściwego kodowania. Jest to związane z naszą najważniejszą zasadą rozwiązywania problemów: zawsze należy mieć jakiś plan. Dobrze zdefiniowany diagram dla przykładowego problemu jest czymś w rodzaju dokładnie zaplanowanej trasy prowadzącej do punktu docelowego przed rozpoczęciem wyjazdu na wakacje. Na początku wymagane jest poświęcenie dodatkowego czasu, by na końcu uniknąć niepotrzebnego wysiłku i frustracji.

TWORZENIE DIAGRAMÓW

Wszystko, czego potrzebujesz do narysowania diagramu, to ołówek i papier. Jeśli jednak masz więcej czasu, sugeruję użycie odpowiedniego programu graficznego. Zawiera on narzędzia do rysowania oraz szablony dedykowane specjalnie dla potrzeb rozwiązywania problemów programistycznych. Jednakże każdy program obsługujący grafikę wektorową będzie odpowiedni (użyty tu termin *wektor* oznacza, że aplikacja pozwala na tworzenie linii prostych i krzywych, a nie przetwarza grafiki rastrowej, tak jak Photoshop). Ilustracje do tej książki wykonałem za pomocą darmowego programu Inkscape. Tworzenie diagramów w komputerze pozwala na ich uporządkowanie i przechowywanie w tym samym miejscu, w którym znajduje się odpowiedni kod. Są one zazwyczaj także bardziej staranne, dzięki czemu można je łatwiej zrozumieć, gdy po jakimś czasie do nich wrócisz. Wreszcie diagram stworzony na komputerze można łatwo skopiować i zmodyfikować, podobnie jak zrobiłem to w przypadku tworzenia rysunku 4.5 na podstawie poprzedniego. Jeśli chcesz umieścić na nim jakąś tymczasową uwagę, możesz to łatwo zrobić na wydrukowanej kopii.

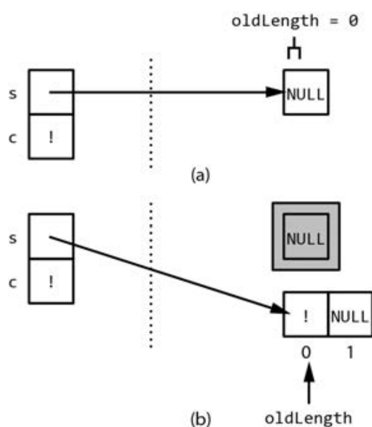
Kup ksi k Pole ksi k

Wracając do naszej funkcji `append` — kod wygląda na godny zaufania, lecz pamiętaj, że stworzyliśmy go na podstawie określonego przykładu. Nie możemy więc być zbyt pewni siebie i zakładać, że będzie działał w każdym przypadku. W szczególności należy przetestować przypadki specjalne. *Przypadkiem specjalnym* w programowaniu nazywana jest sytuacja, w której poprawne dane spowodują typowe działanie kodu dającego błędne wyniki.

Zauważ, że nasz problem nie dotyczy niepoprawnych danych, takich jak wartości spoza zakresu. W przypadku kodu zawartego w tej książce założyliśmy, że programy i funkcje będą otrzymywać poprawne dane. Na przykład jeśli program oczekuje podania szeregu liczb całkowitych oddzielonych przecinkami, zakładamy, że je otrzyma, a nie pojawią się żadne dodatkowe znaki, wartości nieliczbowe itp. Takie założenie jest niezbędne w celu uzyskania programu o sensownej długości, a także uniknięcia powtarzania tego samego kodu sprawdzającego poprawność danych. W świecie rzeczywistym powinniśmy jednak stosować odpowiednie zabezpieczenia przeciwko błędnym danym wejściowym. Taka właściwość programu jest zwana odpornością. *Odporna* aplikacja działa poprawnie nawet w przypadku złych danych wejściowych. Na przykład tego typu program mógłby wyświetlić komunikat błędu, zamiast spowodować awarię działania.

Testowanie przypadków specjalnych

Przyjrzyjmy się ponownie funkcji `append`, sprawdzając ją pod względem przypadków specjalnych; inaczej mówiąc, upewniając się, że dla poprawnych wartości wejściowych nie pojawią się żadne dziwne zachowania programu. Najlepszymi przypadkami specjalnymi są sytuacje ekstremalne, takie jak minimalna i maksymalna wartość wejściowa. Wprawdzie w przypadku funkcji `append` nie istnieje maksymalny rozmiar tablicy znakowej, jednakże mamy określone minimum. Jeśli łańcuch nie ma właściwych znaków, wówczas odpowiada tablicy jednoelementowej (jedynym elementem jest końcowy znak **`NULL`**). Podobnie jak poprzednio, stwórzmy teraz diagram, aby naświetlić sytuację. Załóżmy, że dołączyliśmy znak wykrzyknika do pustego łańcucha, tak jak pokazano na rysunku 4.6.



Rysunek 4.6. Testowanie przypadku najmniejszej wartości dla funkcji append

Kup ksi k Pole ksi k

Rysunek wygląda sensownie, lecz powinniśmy analizowany przypadek sprawdzić jeszcze w naszej funkcji. Dodajmy poniższy kod do funkcji testującej `appendTester`:

```
arrayString b = new char[1];
b[0] = 0;
append(b, '!');
cout << b << "\n";
```

Kod działa poprawnie. Czy funkcja `append` podoba się nam, gdy upewniliśmy się, że będzie działać poprawnie? Kod wydaje się prosty i nie wyczuwam żadnych „złych zapachów”, lecz jest chyba zbyt długi dla tak prostej operacji. Gdy myślę o przyszłej funkcji `concatenate`, stwierdzam, że podobnie jak `append`, będzie ona musiała ustalać długość jednej lub dwóch tablic znakowych. Ponieważ obie operacje wymagają pętli, która znajduje znak pusty kończący łańcuch, moglibyśmy stworzyć oddzielną funkcję dla tego kodu, która w razie potrzeby byłaby wywoływana z `append` i `concatenate`. Zróbmy to i odpowiednio zmodyfikujmy funkcję `append`:

```
int length(arrayString s) {
   ❶ int count = 0;
   while (s[count] != 0) {
      count++;
   }
   return count;
}

void append(arrayString& s, char c) {
   ❷ int oldLength = length(s);
   arrayString newS = new char[oldLength + 2];
   for (int i = 0; i < oldLength; i++) {
      newS[i] = s[i];
   }
   newS[oldLength] = c;
   newS[oldLength + 1] = 0;
   delete[] s;
   s = newS;
}
```

Kod funkcji `length` ❶ został w zasadzie skopiowany z początku funkcji `append`. W niej samej umieściliśmy w tym miejscu wywołanie funkcji `length` ❷. Funkcja `length` jest zwana *funkcją pomocniczą* (ang. *helper function*). Jej zadaniem jest realizowanie operacji wspólnych dla kilku innych funkcji. Eliminacja nadmiarowego kodu oznacza oprócz zmniejszenia jego długości także to, że staje się on bardziej niezawodny i prostszy do modyfikacji. Pozwala także w rozwiązaniu problemu, ponieważ funkcje pomocnicze dzielą kod na mniejsze kawałki, pozwalając nam łatwiej odkryć możliwości jego ponownego wykorzystania.

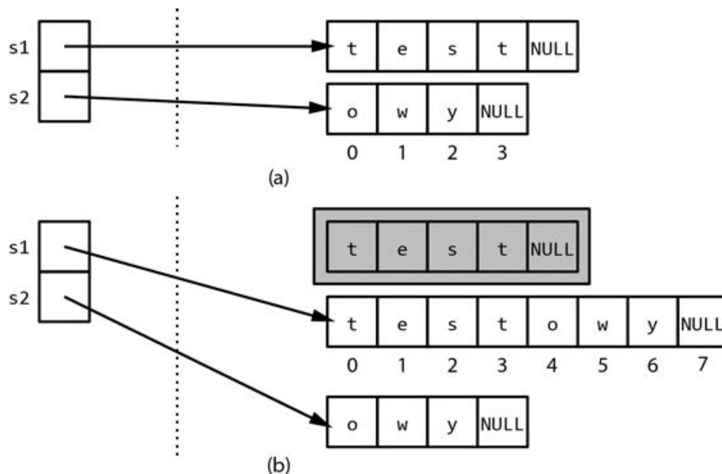
Kup ksi k   Pole ksi k

Kopiowanie dynamicznie przydzielonych łańcuchów

Nadszedł czas na zajęcie się funkcją `concatenate`. Zastosujemy to samo podejście co w przypadku funkcji `append`. Najpierw stworzymy pustą wersję funkcji `concatenate`, aby zdefiniować jej parametry i ich typy danych. Następnie narysujemy diagram dla przypadku testowego, a wreszcie napiszemy odpowiadający mu kod. Oto szablon funkcji razem z dodatkowym kodem testującym:

```
void concatenate(❶ arrayString& s1, ❷ arrayString s2) {
}
void concatenateTester() {
   arrayString a = new char[5];
   a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] =  0;
   arrayString b = new char[4];
   b[0] = 'o'; b[1] = 'w'; b[2] = 'y'; b[3] = 0;
   concatenate(a, b);
}
```

Pamiętaj, że opis funkcji mówi, iż znaki drugiego łańcucha (drugiego parametru) powinny zostać dołączone na końcu pierwszego. Pierwszy parametr funkcji `concatentate` będzie więc referencją ❶, podobnie jak było w przypadku pierwszego parametru funkcji `append`. Drugi parametr ❷ nie powinien jednak zostać zmieniony przez funkcję, dlatego będzie wartością. W naszym testowym przypadku połączymy łańcuchy `test` i `owy`. Sytuacja przed i po wykonaniu tej operacji została przedstawiona na rysunku 4.7.



Rysunek 4.7. Stan przed (a) i po (b) wykonaniu metody concatenate

Szczegóły diagramu powinny być Ci już znane z analizy funkcji `append`. W obecnym przypadku rozpoczynamy od dwóch tablic, dynamicznie przydzielonych na stosie i wskazywanych przez nasze dwa parametry `s1` oraz `s2`. Po zakończeniu

Kup ksi k Pole ksi k

działania funkcji parametr s1 powinien wskazywać na nową tablicę na stosie, która zawiera 9 znaków. Ta, która była wskazywana poprzednio, powinna zostać usunięta. Parametr s2 i jego tablica pozostają niezmienione. Mimo że umieszczenie parametru s2 i tablicy owy na diagramie może wydawać się bezcelowe, w celu uniknięcia błędów kodowania śledzenie niezmienianych elementów jest równie ważne jak tych, które ulegają modyfikacji. Ponumerowałem również elementy starej i nowej tablicy, ponieważ przydało się to w przypadku funkcji append. Wszystko znajduje się na swoim miejscu, możemy więc zająć się pisaniem funkcji:

```
void concatenate(arrayString& s1, arrayString s2) {
   ❶ int s1_OldLength = length(s1);
   int s2_Length = length(s2);
   int s1_NewLength = s1_OldLength + s2_Length;
   ❷ arrayString newS = new char[s1_NewLength + 1];
   ❸ for(int i = 0; i < s1_OldLength; i++) {
      newS[i] = s1[i];
   }
   for(int i = 0; i < s2_Length; i++) {
       newS[❹ s1_OldLength + i] = s2[i];
   }
   ❺ newS[s1_NewLength] = 0;
   ❻ delete[] s1;
   ❼ s1 = newS;
}
```

Na początku obliczamy długości obu łańcuchów, które powinny zostać połączone ❶, a następnie sumujemy je, aby wyznaczyć długość docelowego łańcucha. Pamiętaj, że te wszystkie wartości dotyczą liczby właściwych znaków, bez uwzględniania znaku zakończenia NULL. Gdy tworzymy tablicę na stercie w celu przechowania nowego łańcucha ❷, przydzielamy więc jej o jeden znak więcej, niż wynika to z sumy długości, ponieważ musimy także zapamiętać kończący znak pusty. Następnie kopiujemy znaki z dwóch źródłowych łańcuchów do docelowego ❸. Pierwsza pętla jest prosta, zwróć jednak uwagę na wyznaczanie indeksu w drugiej pętli ❹. Kopiujemy tam dane z tablicy s2 w środek tablicy newS. Jest to kolejny przykład translacji jednego zakresu wartości na inny, co robimy już od rozdziału 2. Dzięki przyjrzeniu się numeracji elementów na diagramie można sprawdzić, jakie zmienne powinno się połączyć ze sobą, by wyznaczyć poprawny indeks w tablicy docelowej. W pozostałej części funkcji umieszczamy znak NULL na końcu nowego łańcucha ❺. Podobnie jak w przypadku append, zwalniamy ze sterty pierwotny blok pamięci, wskazywany przez pierwszy parametr ❻, a następnie przypisujemy mu nowo utworzony łańcuch ❼.

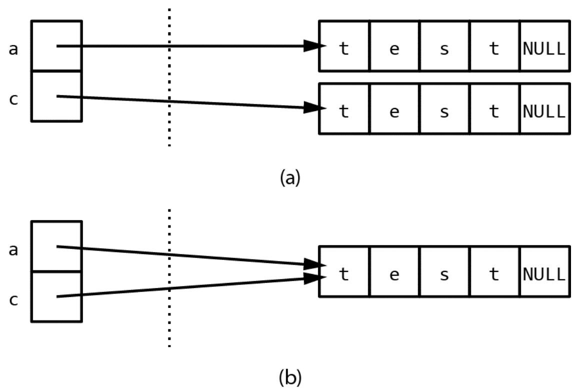
Kod wygląda na działający, lecz podobnie jak przedtem, chcemy się upewnić, że nieumyślnie nie stworzyliśmy funkcji, która działa poprawnie w naszym przykładzie, lecz nie we wszystkich przypadkach. Najbardziej prawdopodobne sytuacje sprawiające kłopoty mogą polegać na użyciu jednego lub dwóch parametrów o długości zero (zawierających sam znak zakończenia łańcucha). Powinniśmy

Kup ksi k Pole ksi k

jawnie przetestować oba przypadki przed dalszym działaniem. Pamiętaj, że podczas sprawdzania poprawności kodu, który używa wskaźników, powinieneś zwracać uwagę na nie same, a nie na wartości na stercie, które są przez nie wskazywane. Oto jeden przypadek testowy:

```
arrayString a = new char[5];
a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] =  0;
arrayString c = new char[1];
c[0] = 0;
concatenate(c, a);
cout << a << "\n" << c << "\n";
❶ cout << (void *) a << "\n" << (void *) c << "\n";
```

Chciałem się upewnić, że wywołanie funkcji `concatenate` spowoduje, iż wskaźniki `a` i `c` będą wskazywać na łańcuch `test`, to znaczy na tablice o takiej samej zawartości. Równie ważne jest jednak to, że powinny one wskazywać na *różne* łańcuchy, jak przedstawiono na rysunku 4.8 (a). Ten przypadek jest sprawdzany w drugiej instrukcji wyjściowej poprzez zmianę typów zmiennych na `void *`. Powoduje to, że strumień wyjściowy wyświetla właściwą wartość wskaźnika ❶. Jeśli wskaźniki miałyby taką samą wartość, oznaczałoby to, że zostały *połączone krzyżowo* (ang. *cross-linked*), jak pokazano na rysunku 4.8 (b). Gdy wskaźniki zostaną z nieznanych powodów połączone ze sobą krzyżowo, mogą pojawić się dziwne problemy, ponieważ zmiana zawartości jednej zmiennej na stosie powoduje w sposób tajemniczy modyfikację innej (w rzeczywistości tej samej) zmiennej. W przypadku złożonej aplikacji takie zjawisko może być jednak trudne do zaobserwowania. Pamiętaj także, że jeśli dwa wskaźniki są połączone krzyżowo, wówczas po zwolnieniu pamięci za pomocą instrukcji `delete` dla jednego z nich drugi staje się wskaźnikiem zawieszonym. Wynika stąd, że musimy starannie analizować nasz kod i zawsze sprawdzać go pod kątem wystąpienia potencjalnych połączeń krzyżowych.



Rysunek 4.8. Funkcja concatenate powinna tworzyć dwa różne łańcuchy (a), a nie dwa wskaźniki połączone ze sobą krzyżowo (b)

Kup ksi k Pole ksi k

Po zaimplementowaniu wszystkich trzech funkcji: `characterAt`, `append` i `concatenate` możemy stwierdzić, że problem został rozwiązany.

Listy powiązane

A teraz spróbujemy czegoś trudniejszego. Obsługa wskaźników będzie bardziej skomplikowana, lecz będziemy prezentować wszystko w prosty sposób, ponieważ wiemy, jak należy tworzyć diagramy.

PROBLEM: OBSŁUGA REJESTRU STUDENTÓW O NIEZNANEJ LICZBIE ELEMENTÓW

Dla tego problemu napiszesz funkcje przechowujące i modyfikujące kolekcję rejestrów studentów. Rejestr studenta zawiera numer studenta i jego ocenę; dane te są liczbami całkowitymi. Powinny zostać zaimplementowane następujące funkcje:

- **`addRecord`** — funkcja używa wskaźnika do kolekcji rejestrów studentów, zawierających numery studentów i ich oceny, aby dodać do niej nowy rejestr z wypełnionymi danymi.
- **`averageRecord`** — funkcja wymaga podania wskaźnika do kolekcji rejestrów studentów i zwraca średnią ocen studentów w kolekcji jako liczbę typu `double`.

Kolekcja może mieć dowolny rozmiar. Operacja `addRecord` będzie wykonywana często, więc powinna zostać zaimplementowana w wydajny sposób.

Istnieje szereg sposobów rozwiązania powyższego zagadnienia, które spełniają przedstawioną specyfikację. My jednak wybierzemy taki, który pozwoli nam przećwiczyć metody rozwiązywania problemów w oparciu o wskaźniki, czyli listy powiązane. Być może słyszałeś już wcześniej o tej strukturze. Jeśli nie, powinieneś wiedzieć, że wprowadzenie list powiązanych do naszych rozważań spowoduje przejście na zupełnie inny poziom dyskusji. Zdolny programista być może potrafiłby rozwiązać wcześniejsze problemy, mając do dyspozycji wystarczającą ilość czasu i umiejętność przeprowadzania starannej analizy. Większość programistów nie wymyśliłaby jednak list powiązanych bez dodatkowej pomocy. Gdy tylko je poznasz i opanujesz podstawy ich użycia, będziesz mógł tworzyć inne struktury w swoim umyśle, a wtedy wszystko szybko zacznie poprawnie działać. Lista powiązana jest prawdziwą strukturą dynamiczną. Nasze tablice łańcuchów przechowywaliśmy w dynamicznie przydzielonej pamięci, lecz po stworzeniu były one statycznymi strukturami bez możliwości ich powiększania lub zmniejszania i mogły tylko być zamieniane na inne. W przeciwieństwie do tego lista powiązana jest rosnącą w czasie kolekcją elementów — czymś w rodzaju łańcucha.

Kup ksi k Pole ksi k

Tworzenie listy węzłów

Stwórzmy przykładową listę powiązaną rejestrów studentów. Aby to wykonać, potrzebujesz struktury, która przechowuje dane, jakie powinny się znaleźć w kolekcji, a oprócz tego zawiera wskaźnik do niej samej. W naszym przypadku w strukturze będzie się znajdować numer studenta i jego ocena.

```
struct ❶ listNode {
   ❷ int studentNum;
   int grade;
   ❸ listNode * next;
};
❹ typedef listNode * studentCollection;
```

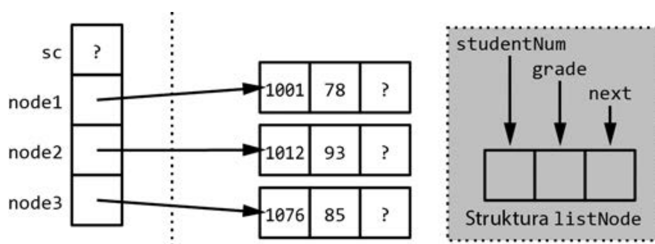
Nasza struktura nazywa się `listNode` ❶. Struktura używana do tworzenia listy powiązanej jest zawsze nazywana *węzłem* (ang. *node*). Prawdopodobnie nazwa ta powstała jako analogia do terminu botanicznego oznaczającego miejsce na pniu, z którego wyrasta nowa gałąź. Węzeł składa się z numeru studenta ❷ i oceny, które razem tworzą jego właściwą treść. Zawiera on jednak także wskaźnik do tego samego typu struktury, który definiujemy ❸. Taki zapis dla większości programistów widzących go po raz pierwszy jest dezorientujący, a nawet wydaje się niemożliwy pod względem składniowym. W jaki sposób można stworzyć strukturę, korzystając podczas jej definiowania z niej samej? Jest to jednak prawidłowa operacja, a jej sens zostanie niebawem wyjaśniony. Zwróć uwagę na to, że wskaźnik odwołujący się w węźle do samego siebie ma zazwyczaj nazwę *next*, *nextPtr*[1] itd. Wreszcie kod zawiera deklarację `typedef` dla typu wskaźnika do naszego węzła ❹. A teraz stwórzmy naszą przykładową listę powiązaną przy użyciu zadeklarowanych typów:

```
❶ studentCollection sc;
❷ listNode * node1 = new listNode;
❸ node1->studentNum = 1001; node1->grade = 78;
listNode * node2 = new listNode;
node2->studentNum = 1012; node2->grade = 93;
listNode * node3 = new listNode;
❹ node3->studentNum = 1076; node3->grade = 85;
❺ sc = node1;
❻ node1->next = node2;
❼ node2->next = node3;
❽ node3->next = NULL;
❾ node1 = node2 = node3 = NULL;
```

[1] W języku angielskim oznacza ona „następny” lub „następny wskaźnik” — *przyp. tłum.*

Kup ksi k Pole ksi k

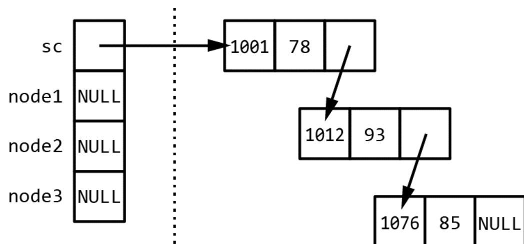
Rozpoczynamy od zadeklarowania zmiennej `sc` typu `studentCollection` ❶, która ostatecznie będzie nazwą naszej listy powiązanej. Następnie deklarujemy zmienną `node1` ❷, która jest wskaźnikiem do struktury `listNode`. Jak już wspomniano, nazwa `studentCollection` jest synonimem `listNode *`, jednakże w celu lepszej czytelności będę używać typu `studentCollection` tylko dla zmiennych, które odwołują się do całej struktury listy. Po zadeklarowaniu zmiennej `node1` i przypisaniu jej wskaźnika typu `listNode` do nowo przydzielonego miejsca na stercie ❷ inicjalizujemy wartości pól `studentNum` i `grade` ❸. W tym momencie pole `next` nie jest jeszcze zainicjalizowane. W tej książce nie omawiamy składni języka C++, lecz jeśli nie widziałeś wcześniej notacji `->`, chciałbym wyjaśnić, że jest ona używana do określania pola w strukturze (lub klasie), do której odwołujemy się za pomocą wskaźnika. Tak więc zapis `node1->studentNum` oznacza: „Pole `studentNum` w strukturze wskazywanej przez `node1`" i jest równoważny wyrażeniu `(*node1).studentNum`. Następnie powtarzamy tę samą czynność dla `node2` i `node3`. Po przypisaniu wartości pól dla ostatniego węzła stan pamięci przedstawia się zgodnie z rysunkiem 4.9. W poniższych diagramach użyłem metody „podzielonych kontenerów", którą wcześniej zastosowałem w przypadku tablic znakowych.



Rysunek 4.9. W połowie drogi do stworzenia przykładowej listy powiązanej

Gdy mamy już wszystkie węzły, możemy je połączyć ze sobą, by stworzyć listę powiązaną. Tego właśnie dotyczy pozostała część kodu. Po pierwsze, wskazujemy poprzez naszą zmienną `studentCollection` pierwszy węzeł ❺, następnie przypisujemy polu `next` z pierwszego węzła adres drugiego węzła ❻, a wreszcie polu `next` z drugiego węzła adres trzeciego ❼. W dalszej kolejności przypisujemy wartość `NULL` (jak już wspomniałem, jest to po prostu synonim zera) dla pola `next` w ostatnim węźle (8.). Wykonujemy tę czynność z tego samego powodu, z jakiego w poprzednim problemie umieszczaliśmy pusty znak na końcu naszych tablic: aby ograniczyć strukturę. Tak jak potrzebowaliśmy specjalnego znaku wskazującego koniec tablicy, analogicznie musimy użyć wartości równej zeru w polu `next` ostatniego węzła naszej listy powiązanej, aby upewnić się, że *jest* on rzeczywiście ostatni. Wreszcie, aby uporządkować kod i uniknąć pojawienia się ewentualnych problemów połączeń krzyżowych, przypisujemy wartość `NULL` do każdego z poszczególnych wskaźników do węzłów ❾. Końcowy stan pamięci został przedstawiony na rysunku 4.10.

Kup ksi k Pole ksi k



Rysunek 4.10. Ukończona przykładowa lista powiązana

Mając rysunek przed oczami, możemy łatwo zrozumieć, dlaczego nasza struktura jest zwana listą powiązaną: każdy jej węzeł jest połączony z następnym. Listy powiązane są często przedstawiane graficznie w postaci liniowej, lecz ja wolę widok prezentujący je w postaci rozproszonej w pamięci, ponieważ uwypukla on to, że węzły nie mają ze sobą żadnego związku z wyjątkiem połączeń — każdy z nich mógłby znajdować się w dowolnym miejscu na stercie. Upewnij się, że przeanalizowałeś kod tak dokładnie, iż zrozumiałeś, że jest on zgodny z diagramem.

Zwróć uwagę na to, że w docelowej konfiguracji jest używany tylko jeden wskaźnik wykorzystujący pamięć stosu. Jest nim zmienna `sc` typu `studentCollection`, która wskazuje pierwszy węzeł. Wskaźnik spoza listy (czyli taki, który nie jest polem `next` jednego z jej węzłów) wskazujący jej pierwszy węzeł jest zwany *wskaźnikiem głowy* (ang. *head pointer*). Na poziomie symbolicznym reprezentuje on całą listę, lecz oczywiście w rzeczywistości wskazuje bezpośrednio tylko jej pierwszy węzeł. Aby przejść do drugiego węzła, musimy po drodze minąć pierwszy, by dojść do trzeciego, musimy przejść przez dwa pierwsze itd. Oznacza to, że listy powiązane pozwalają jedynie na dostęp sekwencyjny — w przeciwieństwie do dostępu swobodnego, oferowanego przez tablice. Dostęp sekwencyjny jest wadą struktur wykorzystujących listy powiązane. Jak wcześniej wspomnieliśmy, ich zaletą jest możliwość powiększania i pomniejszania rozmiaru całej listy poprzez dodawanie i usuwanie węzłów bez potrzeby tworzenia nowej struktury i kopiowania do niej danych, tak jak robiliśmy w przypadku tablic.

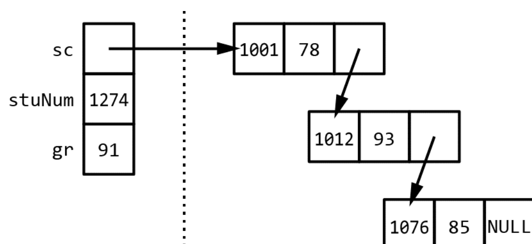
Dodawanie węzłów do listy

Obecnie zajmiemy się implementacją funkcji `addRecord`. Jej zadaniem jest tworzenie nowego węzła i dołączanie go do istniejącej listy powiązanej. Użyjemy tej samej techniki, którą wykorzystaliśmy w poprzednim problemie. Na początku stworzymy nagłówek pustej funkcji i jej przykładowe wywołanie. Do celów testowych dodamy w taki sposób odpowiedni kod do poprzedniego listingu, by zmienna `sc` była już wskaźnikiem głowy dla listy składającej się z trzech węzłów.

```
void addRecord(studentCollection& sc, int stuNum, int gr) {
}
❶ addRecord(sc, 1274, 91);
```

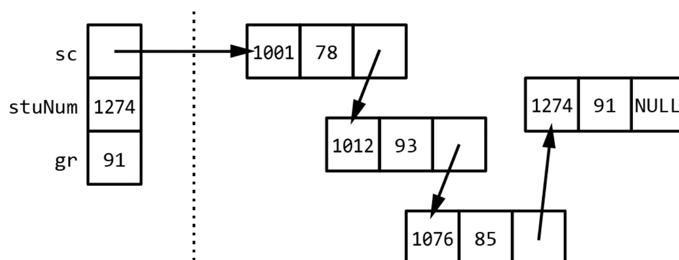

Kup książkę Poleć książkę

Wywołanie ❶ powinno się pojawić na końcu poprzedniego listingu. Mając szkic funkcji wraz z jej parametrami, możemy przedstawić na diagramie stan pamięci przed jej wywołaniem, co zaprezentowano na rysunku 4.11.



Rysunek 4.11. Stan pamięci przed wywołaniem funkcji addRecord

Jeśli chodzi o stan po wywołaniu funkcji, mamy jednak możliwość wyboru. Możemy przypuszczać, że utworzymy nowy węzeł na stercie, a następnie skopiujemy wartości parametrów `stuNum` i `gr` do odpowiednich jego pól `studentNum` i `grade`. Pytanie brzmi: w którym logicznym miejscu listy umieścimy ten węzeł? Najbardziej oczywistym wyborem byłoby umieszczenie go na jej końcu: pole `next` zawiera wartość `NULL`, idealnie nadając się do wskazywania na nowy węzeł. Takie rozwiązanie przedstawiono na rysunku 4.12.



Rysunek 4.12. Propozycja stanu pamięci po wywołaniu funkcji addRecord

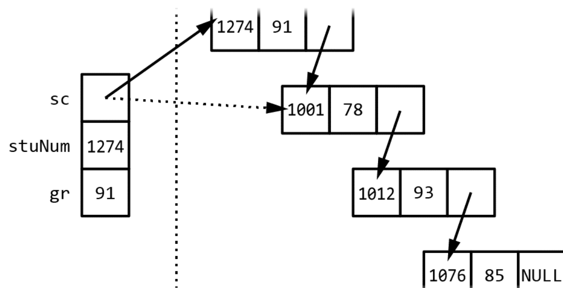
Jeśli jednak założymy, że kolejność rejestrów nie ma znaczenia (czyli nie musimy przechowywać ich w takiej samej kolejności, w jakiej zostały dodane do kolekcji), wówczas nasza decyzja nie będzie poprawna. Aby to zrozumieć, rozważ kolekcję składającą się nie z trzech, lecz trzech tysięcy rejestrów. By uzyskać dostęp do ostatniego elementu naszej listy powiązanej w celu zmodyfikowania pola `next`, musielibyśmy przejść przez wszystkie 3000 węzłów. Taka operacja jest szczególnie nieefektywna, ponieważ możemy umieścić nowy węzeł na liście bez potrzeby „podróżowania" przez *wszystkie* istniejące elementy.

Rozwiązanie przedstawiono na rysunku 4.13. Po utworzeniu nowego węzła zostaje on dołączony na *początku* listy, a nie na jej końcu. Po tej operacji nasz wskaźnik głowy `sc` wskazuje nowy węzeł, a jego pole `next` wskazuje element, który wcześniej był pierwszym węzłem na liście, czyli strukturę zawierającą numer studenta 1001. Zauważ, że podczas przypisywania wartości do pola `next`

Kup ksi k Pole ksi k

w nowym węźle jedynym wskaźnikiem, który ulega modyfikacji, jest sc, a żadne inne wartości dla istniejących węzłów nie zmieniają się ani nawet nie są odczytywane. Na podstawie diagramu możemy więc stworzyć kod:

```
void addRecord(studentCollection& sc, int stuNum, int gr) {
   ❶ listNode * newNode = new listNode;
   ❷ newNode->studentNum = stuNum;
   newNode->grade = gr;
   ❸ newNode->next = sc;
   ❹ sc = newNode;
}
```

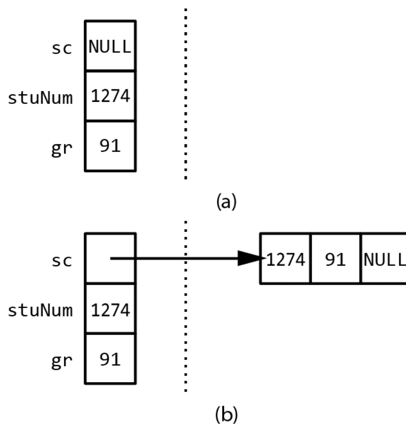


Rysunek 4.13. Akceptowany stan po wykonaniu funkcji addRecord. Strzałka z linii kropkowanej wskazuje poprzednią wartość wskaźnika przechowywanego w zmiennej sc

Jak już wspominałem, zamiana diagramu na kod jest dużo prostsza niż analizowanie wszystkiego wyłącznie w swoim umyśle. Kod może zostać napisany bezpośrednio na podstawie rysunku. Tworzymy nowy węzeł ❶, a następnie przypisujemy polom struktury numer studenta i ocenę, pobrane z parametrów ❷. W dalszej kolejności przyłączamy nowy węzeł do listy, wskazując za pomocą jego pola `next` wcześniejszy pierwszy węzeł (przypisując mu wartość `sc`) ❸, a następnie wskazując za pomocą zmiennej `sc` ten nowy element ❹. Zauważ, że ostatnie dwa kroki powinny być wykonane właśnie w takiej kolejności: musimy użyć pierwotnej wartości zmiennej `sc`, zanim zostanie ona zmieniona. Weź również pod uwagę to, że ze względu na modyfikację zmiennej `sc` musi być ona parametrem referencyjnym.

Jak zwykle podczas tworzenia kodu na podstawie przykładu powinniśmy przetestować ewentualne przypadki specjalne. W naszej sytuacji oznacza to sprawdzenie, czy funkcja działa z pustą listą. W przypadku tablic znakowych pusty łańcuch był poprawnym wskaźnikiem, ponieważ mieliśmy tablicę, na którą można wskazywać — tablicę zawierającą tylko kończący znak `NULL`. Tu liczba węzłów jest równa liczbie rejestrów, a pusta lista oznacza wskaźnik głowy wskazujący wartość `NULL`. Czy więc nasz kod zadziała, jeśli na liście, której wskaźnik głowy jest równy `NULL`, spróbujemy umieścić przykładowe dane? Na rysunku 4.14 prezentujemy stany przed i po wykonaniu tej operacji.

Kup ksi k Pole ksi k



Rysunek 4.14. Stany przed i po wywołaniu funkcji addRecord dla pustej listy

Analizując ten przykład na podstawie kodu, możemy stwierdzić, że działa on prawidłowo. Nowy węzeł jest tworzony tak jak poprzednio. Ponieważ zmienna `sc` przed operacją jest równa `NULL`, poprawnym działaniem jest jej skopiowanie ❸ do pola `next` naszego nowego węzła, ponieważ dzięki temu lista jednoelementowa zostaje poprawnie zakończona. Zauważ, że w sytuacji, w której używalibyśmy innej metody implementacyjnej, polegającej na umieszczeniu nowego węzła na końcu listy zamiast na jej początku, wstępnie pusta lista *mogłaby* być przypadkiem szczególnym, ponieważ tylko wtedy zmienna `sc` byłaby modyfikowana.

Przeglądanie listy

Nadszedł czas na opracowanie funkcji `averageRecord`. Jak poprzednio, rozpoczniemy od stworzenia nagłówka pustej funkcji i diagramu. Odpowiedni kod został przedstawiony poniżej. Zakładamy, że wywołanie ❶ wystąpi po stworzeniu naszej pierwotnej listy przykładowej, jak zaprezentowano na rysunku 4.10.

```
double averageRecord(studentCollection sc) {
}
❶ int avg = averageRecord(sc);
```

Jak widzisz, postanowiłem wyznaczać wartość średnią jako liczbę całkowitą, podobnie jak czyniliśmy w przypadku tablic z poprzedniego rozdziału. W zależności od problemu może się jednak okazać, że lepiej będzie ją wyliczać jako wartość zmiennoprzecinkową. Potrzebujemy jeszcze diagramu, lecz mamy przecież dostępny stan przed operacją, zaprezentowany na rysunku 4.9. Dla przedstawienia stanu po wywołaniu funkcji nie wymagamy diagramu, ponieważ nie zmienia ona dynamicznej struktury listy, a jedynie tworzy na jej podstawie raport. Powinniśmy po prostu znać oczekiwany wynik, który w naszym przypadku wynosi około 85,3333.

Kup ksi k Pole ksi k

Jak więc będziemy wyliczać średnią? Mamy już ogólny pomysł oparty na naszych doświadczeniach z wyznaczania średniej ze wszystkich elementów tablicy. Musimy dodawać każdą wartość z kolekcji, a następnie podzielić sumę przez liczbę elementów. W przypadku kodu stosowanego dla tablic odczytywaliśmy dane w pętli `for`, poczynając od 0 aż do wartości o 1 mniejszej od rozmiaru tablicy, przy czym licznik pętli był zarazem indeksem. W obecnej sytuacji nie możemy użyć pętli `for`, ponieważ nie wiemy z góry, ile liczb jest zapisanych w liście powiązanej. Zamiast tego powinniśmy kontynuować działania aż do osiągnięcia wartości `NULL` w polu `next`, która oznacza koniec listy. Sugeruje to użycie pętli `while`, czyli rozwiązania, które wykorzystaliśmy już wcześniej w tym rozdziale do przetwarzania tablic o nieznanej długości. Tego typu przetwarzanie jest zwane *przeglądaniem listy*. Jest to jedna z podstawowych operacji, jakie można wykonywać na listach powiązanych. Zaimplementujmy więc naszą ideę przeglądania listy, by mogła pomóc nam rozwiązać problem:

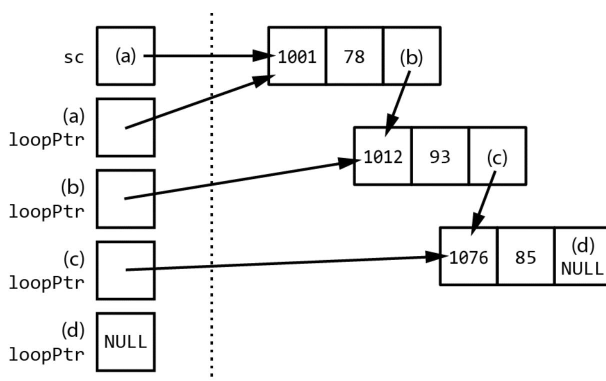
```
double averageRecord(studentCollection sc) {
   ❶ int count = 0;
   ❷ double sum = 0;
   ❸ listNode * loopPtr = sc;
   ❹ while (loopPtr != NULL) {
      ❺ sum += loopPtr->grade;
      ❻ count++;
      ❼ loopPtr = loopPtr->next;
   }
   ❽ double average = sum / count;
   return average;
}
```

Rozpoczynamy od zadeklarowania zmiennej `count`, która będzie przechowywać liczbę węzłów napotkanych na liście ❶. Będzie to również liczba wartości występujących w kolekcji, której użyjemy do wyznaczenia średniej. Następnie deklarujemy zmienną `sum` przeznaczoną do przechowywania sumy częściowej wartości ocen na liście ❷. Oprócz tego tworzymy deklarację wskaźnika do struktury `listNode`, nazwanego `loopPtr`, który wykorzystamy do przeglądania listy ❸. Jest to odpowiednik naszej zmiennej typu całkowitego, której używaliśmy w pętli `for` podczas przetwarzania tablic. Pamięta ona bieżące położenie na liście powiązanej nie za pomocą licznika, lecz przy wykorzystaniu wskaźnika do węzła, który właśnie przetwarzamy.

W tym miejscu kodu rozpoczyna się właściwe przeglądanie listy. Pętla przeglądania trwa aż do momentu, gdy nasz wskaźnik uzyska wartość `NULL` ❹. Wewnątrz pętli dodajemy wartość pola `grade` bieżącego węzła do zmiennej `sum` ❺. Zwiększamy licznik `count` ❻, a następnie przypisujemy pole `next` bieżącego węzła do naszego wskaźnika sterującego pętlą ❼. Jest to trudniejsza część kodu, dlatego upewnij się, że dobrze ją rozumiesz. Na rysunku 4.15 zaprezentowano, w jaki sposób zmienna węzła ulega modyfikacji w trakcie działania programu. Symbole od (a) do (d) oznaczają różne momenty życia zmiennej `loopPtr` oraz

Kup ksi k Pole ksi k

miejsca, które ona wskazuje. Punkt (a) oznacza po prostu początek pętli: zmienna `loopPtr` została właśnie zainicjalizowana wartością `sc`, dlatego, podobnie jak ona, wskazuje na pierwszy węzeł na liście. Podczas pierwszej iteracji pętli do zmiennej `sum` zostaje dodana wartość oceny równa `78`, pobrana z pierwszego węzła. Wartość pola `next` zostaje skopiowana do `loopPtr`, więc obecnie zmienna ta wskazuje drugi węzeł na liście — jest to punkt (b). Podczas drugiej iteracji dodajemy wartość `93` do zmiennej `sum` oraz kopiujemy pole `next` drugiego węzła do zmiennej `loopPtr` — jest to punkt (c). Wreszcie podczas trzeciej i ostatniej iteracji pętli dodajemy `85` do zmiennej `sum` i przypisujemy wartość `NULL` pobraną z pola `next` trzeciego węzła do zmiennej `loopPtr` — jest to punkt (d). Gdy ponownie osiągamy początek pętli `while`, od razu z niej wychodzimy, ponieważ `loopPtr` jest równa `NULL`. W każdej iteracji zwiększaliśmy licznik `count`, którego wartość wynosi 3.



Rysunek 4.15. Zmiany lokalnej zmiennej loopPtr podczas iteracji w pętli dla funkcji averageRecord

Po zakończeniu pętli po prostu dzielimy wartość `sum` przez `count` i zwracamy wynik ❽.

Kod działa poprawnie dla naszych testowych danych, lecz jak zwykle powinniśmy sprawdzić ewentualne przypadki specjalne. Jak już wspominałem, najbardziej oczywistym przypadkiem specjalnym dla naszej struktury jest lista pusta. Co się stanie z naszym kodem, gdy funkcja rozpocznie swoje działanie, a wartość `sc` będzie równa `NULL`?

Czy potrafisz zgadnąć? Kod wykonał niepoprawną operację (musiałem sprawić, by jeden z przypadków specjalnych spowodował pojawienie się poważnego problemu, ponieważ w przeciwnym razie nie traktowałbyś moich słów poważnie). Sama pętla przetwarzająca listę powiązaną działa poprawnie. Jeśli `sc` wynosi `NULL`, wówczas zmienna `loopPtr` zostaje również zainicjalizowana tą wartością, a pętla kończy się zaraz po jej rozpoczęciu. Wartość zmiennej `sum` jest wówczas równa 0, co wydaje się sensowne. Problem pojawia się w momencie, gdy chcemy wykonać dzielenie w celu wyznaczenia średniej ❽. Wartość `count` jest również równa 0, co oznacza, że dzielimy przez zero, a w wyniku tego program

Kup ksi k Pole ksi k

przestaje działać lub otrzymujemy bezsensowny wynik. Aby uniknąć tego problemu, powinniśmy sprawdzać pod koniec funkcji, czy zmienna `count` jest równa 0. Dlaczego jednak nie moglibyśmy obsługiwać tej sytuacji na początku i sprawdzać `sc`? Dodajmy następujący wiersz na samej górze funkcji `averageRecord`:

```
if (sc == NULL) return 0;
```

Jak widać na powyższym przykładzie, obsługa przypadków specjalnych jest zazwyczaj całkiem prosta. Musimy mieć po prostu czas, by móc je zidentyfikować.

Wnioski i następne działania

W tym rozdziale tylko powierzchownie przeanalizowaliśmy rozwiązywanie problemów za pomocą wskaźników i pamięci dynamicznej. Tego typu zagadnienia pojawią się także w dalszej części książki. Na przykład obiektowo zorientowane techniki programowania, które omówimy w rozdziale 5., są szczególnie przydatne podczas pracy ze wskaźnikami. Dzięki nim można obudowywać wskaźniki w taki sposób, by nie martwić się o wycieki pamięci, wskaźniki zawieszone i inne związane z nimi niebezpieczeństwa.

Mimo że do zdobycia jest jeszcze dużo wiedzy związanej z rozwiązywaniem problemów z tego obszaru, będziesz mógł rozwijać swoje umiejętności przy wykorzystaniu coraz bardziej złożonych struktur opartych na wskaźnikach, jeśli zastosujesz się do głównych zasad omówionych w tym rozdziale. Po pierwsze, stosuj ogólne reguły dotyczące rozwiązywania problemów. Następnie wykorzystuj określone zasady dotyczące wskaźników, a także używaj diagramów lub innych narzędzi pozwalających na zwizualizowanie każdego z rozwiązań przed rozpoczęciem kodowania.

Ćwiczenia

Nie żartuję, wspominając o ćwiczeniach. Mam nadzieję, że nie czytasz samego rozdziału i zaraz potem nie przechodzisz do następnego.

4.1. Samodzielnie zdefiniuj własne zadanie. Weź pod uwagę problem, który jest ograniczony rozmiarem tablicy i dla którego znasz rozwiązanie przy jej użyciu. Zmień kod w taki sposób, by usunąć ograniczenie i zastosować tablicę przydzielaną dynamicznie.

4.2. Dla dynamicznie przydzielanych łańcuchów stwórz funkcję `substring` z następującymi trzema parametrami: zmienną typu `arrayString`, liczbą całkowitą oznaczającą początkową pozycję, liczbą całkowitą określającą długość łańcucha. Funkcja powinna zwrócić wskaźnik do nowo przydzielonego bloku pamięci zawierającego tablicę znaków. Musi ona zawierać znaki pochodzące z oryginalnego

Kup ksi k Pole ksi k

łańcucha, poczynając od określonej pozycji i o podanej długości. Pierwotny łańcuch nie powinien zostać zmodyfikowany. Jeśli więc oryginalnym łańcuchem był `abcdefg`, pozycją początkową 3, a długością 4, wówczas nowy łańcuch powinien być równy `cdef`.

4.3. Dla naszych dynamicznie przydzielanych łańcuchów stwórz funkcję `replaceString`, która wykorzystuje trzy parametry, każdy o typie `arrayString`: `source`, `target` i `replaceText`. Funkcja zamienia każde wystąpienie łańcucha `target` w łańcuchu `source` łańcuchem `replaceText`. Na przykład jeśli `source` wskazuje tablicę zawierającą ciąg znaków `abcdabee`, target wskazuje `ab`, a `replaceText` ciąg `xyz`, wówczas po zakończeniu funkcji parametr `source` powinien wskazywać tablicę zawierającą łańcuch `xyzcdxyzee`.

4.4. Zmień implementację naszych łańcuchów w taki sposób, aby element `location[0]` tablicy przechowywał jej rozmiar (czyli pozycja `location[1]` powinna zawierać pierwszy faktyczny znak łańcucha), w przeciwieństwie do zastosowania ograniczającego znaku pustego. Zaimplementuj każdą z funkcji: `append`, `concatenate` i `characterAt`, wykorzystując w miarę możliwości sposób zapamiętania informacji o rozmiarze. Ponieważ nie będziemy już wykorzystywać znaku pustego oczekiwanego przez standardowy strumień wyjściowy, musisz napisać własną funkcję `output`, która będzie przetwarzać łańcuch, wyświetlając jego elementy.

4.5. Napisz funkcję `removeRecord`, która używa wskaźnika do struktury `studentCollection` oraz numeru studenta, a następnie usuwa z kolekcji rejestr z tym właśnie numerem.

4.6. Stwórz implementację łańcuchów, która zamiast wykorzystywać dynamicznie alokowane tablice, używa listy powiązanej zawierającej znaki. Tak więc będziesz mieć listę powiązaną, w której właściwymi danymi będą pojedyncze znaki. Pozwoli to na operacje powiększania bez potrzeby ponownego tworzenia całego łańcucha od nowa. Rozpocznij od zaimplementowania funkcji `append` i `characterAt`.

4.7. Korzystając z poprzedniego ćwiczenia, zaimplementuj funkcję `concatenate`. Pamiętaj, że w przypadku wywołania `concatenate(s1, s2)`, w którym oba parametry są wskaźnikami do pierwszych węzłów odpowiednich list powiązanych, funkcja powinna kopiować każdy z węzłów `s2` i dołączać go na koniec `s1`. Oznacza to, że funkcja nie powinna po prostu przypisać polu `next` ostatniego węzła w liście `s1` adresu pierwszego węzła listy `s2`.

4.8. Dodaj funkcję `removeChars` do implementacji łańcuchów wykorzystującej listę powiązaną. Funkcja powinna usuwać część znaków z łańcucha, korzystając z parametrów określających pozycję i długość. Na przykład wywołanie `removeChars(s1, 5, 3)` powinno usunąć trzy znaki, poczynając od piątej pozycji w łańcuchu. Upewnij się, że pamięć wykorzystywana przez usunięte węzły zostanie prawidłowo zwolniona.

4.9. Wyobraź sobie listę powiązaną, w której zamiast znaków przechowujemy cyfry, czyli liczby całkowite z zakresu od 0 do 9. Dzięki temu moglibyśmy reprezentować dodatnie liczby o dowolnym rozmiarze. Na przykład liczba 149 byłaby listą powiązaną, w której pierwszym węźle przechowywalibyśmy cyfrę 1, w drugim 4, a w trzecim i ostatnim 9. Napisz funkcję `intToList`, która wykorzystuje jako

Kup ksi k Pole ksi k

parametr liczbę całkowitą i tworzy tego typu listę powiązaną. Wskazówka: łatwiej będzie stworzyć listę przechowującą cyfry w odwrotnej kolejności, czyli dla wartości 149 w pierwszym węźle byłaby przechowywana cyfra 9.

4.10. Dla struktury z poprzedniego ćwiczenia napisz funkcję, która używa dwóch list jako parametrów i w wyniku swojego działania tworzy nową listę, zawierającą sumę liczb w nich zapisanych.

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

Skorowidz

A

abstrakcyjne typy danych, 149, 217, 232
algorytm, 216
  Luhna, 50, 61
  quicksort, 89
  wyszukiwania interpolacyjnego, 239
  wyszukiwania sekwencyjnego, 92
alokacja pamięci, 112, 115
analiza
  kodu, 271
  wyników, 266
API, 218
ASCII, 69

B

biblioteka
  cstring, 218
  DirectX, 218
biblioteki standardowe, 218
blok kodu, 215
błąd semantyczny, 243

C

cechy rekurencji
  wydajność, 207
  wymagania pamięciowe, 207
  złożoność konceptualna, 207
czas życia zmiennej, 118
czytelność kodu, 150, 197

D

definiowanie iteratora, 227
deklarowanie
  klasy, class declaration, 144
  konstruktorów, 228
  obiektu, 144
  tablicy, 79
  wektora, 102
  węzła struktury, 161
dekodowanie wiadomości, 61
delegowanie pracy, 186
destruktor, 169
diagram, 124, 135, 165
długość łańcucha, 128
dodawanie
  węzła, 162
  węzła do listy, 133
dokumentacje
  bibliotek, 272
  funkcji, 273
dominanta zbioru, 85
dostęp
  do pamięci, 109
  sekwencyjny, 78
  swobodny, 78, 103, 119
drzewo binarne, 201, 204, 212
dynamiczne
  przydzielanie pamięci, 107, 112, 170
  struktury danych, 160, 198
  ustalanie rozmiaru struktury, 110
dyspozytor, dispatcher, 192
dziedziczenie, 147
dzielenie przez zero, 245

Kup ksi k
Pole ksi k

E

efektywność
  czasowa, 103
  pamięciowa, 103, 112
enkapsulacja, encapsulation, 146, 148, 152

F

fikcyjny rekord, 230
fragmentacja pamięci, memory fragmentation, 115
funkcja
  addRecord, 130, 136
  append, 119, 121, 125
  appendTester, 122
  averageRecord, 130, 138
  characterAt, 119
  compareFunc, 89
  concatenate, 119, 127, 129
  length, 126
  malloc, 115
  qsort, 82, 88, 220, 234, 238
  recordWithNumber, 160
  removeRecord, 160
  strcmp, 218, 221
funkcje
  iteracyjne, 192, 194
  opakowujące, wrapper functions, 196, 204
  rekurencyjne, 117, 206
  składowe, 152

G

gra wisielec, 254

H

hermetyzacja, 148, 154
histogram, 89, 90

I

implementacja metod, 154
indeks, 78
informacje
  o klasach, 144
  o komponentach, 215
  o rekurencji, 182
  o tablicach, 78
  o wskaźnikach, 108
instrukcja
  if, 88, 97
  return, 168
  switch, 91
interfejs, 148
interfejs programowania aplikacji, API, 218
iterator, 226, 230

J

JDBC, 218
język proceduralny, 13

K

klasa, 143, 177
  scIterator, 228, 230
  stack, 208
  studentCollection, 162, 169, 221, 226
  wektorowa, 101
  wordList, 266
klasy
  fikcyjne, 177
  kontenerowe, 226
  pochodne, subclass, 144
klient, 148
kod
  gry, 257–266
  znaku, 53
kodowanie, 242
komponenty, 215, 219, 232
  niskopoziomowe, 239
  wyższego poziomu, 239
kompozycja, composition, 161
konstruktor, 144
  domyślny, default constructor, 145, 155
  kopiujący, copy constructor, 174
  z parametrami, 155
kopia projektu, 267
kopiowanie
  głębokie, deep copy, 172
  listy, 172
  łańcuchów, 127
  płytkie, shallow copy, 171
  tablicy, 79
korzeń, 201

Kup ksi k
Pole ksi k

L

licznik count, 137, 197
LIFO, 149
listy powiązane, 130, 133, 198
listy powiązane jednokierunkowe, 198
literał numeryczny, 244

Ł

łamigłówka
  lis, gęś i kukurydza, 17
  przesuwane elementy, 22
  sudoku, 26
  zamek Quarrasi, 30
łańcuch tekstowy, 119

M

metoda, method, 144
  addRecord, 160
  get, 153
  push_back, 101
  removeRecord, 165
  set, 153
  setFirstStudentPolicy, 221
metody
  chronione, 159
  prywatne, 159
  wspierające, 156
modyfikowanie
  danych obiektu, 156
  interfejsu, 148

N

nieuczciwa strategia, 252
nowe umiejętności, 268, 271

O

obiekt studentRecord, 230
obiekty, 143
obliczanie
  długości łańcuchów, 128
  kosztu licencji, 93
  średniej, 84, 244
obsługa
  łańcuchów tekstowych, 119
  pamięci dynamicznej, 176
  wyjątków, 165
odczytywanie pliku tekstowego, 258
operator
  &, 112
  ||, 164
  ++, 230
  adresu, 108
  delete[], 100
  modulo, 51, 55
  new, 109
  przypisania, 173, 243
  równości, 243
optymalizacja kodu, 89
oszukiwanie w grze wisielec, 254

P

pakiety, 272
pamięć, 112
pętla
  for, 43
  while, 56, 168
pętle zagnieżdżone, 96
pisanie kodu gry, 257
plik nagłówkowy iostream, 42
pole, data member, 144
polecenia wyjściowe, 42
polecenie delete intPtr, 118
ponowne wykorzystanie kodu, 63, 147, 218
porównywanie
  elementów tablicy, 220
  wyników, 238
problem
  czytanie liczby, 64
  dekodowanie wiadomości, 60, 72
  dyżurny, 220, 223
  ile papug, 183
  kontrola poprawności, 49, 54
  najlepszy klient, 186
  obsługa łańcuchów, 119
  obsługa rejestru studentów, 130
  odliczanie w dół, 44
  oszukująca gra w wisielca, 251
  połówka kwadratu, 42
  porządek wyświetlania listy powiązanej, 209
  przekształcenie cyfry w liczbę, 52
  sortowanie niektórych elementów, 234
  szkolny dziennik, 151
  trójkąty połączone bokami, 46
  wartości dodatnie, 57
  wydajne przetwarzanie, 224, 231

Kup ksi k
Pole ksi k

problem
  wyliczanie sumy liczb, 192
  wyszukiwanie w drzewie binarnym, 202
  wyznaczanie dominanty, 85
  wyznaczanie liczby liści, 204
  zarządzanie nieznaną liczbą rejestrów, 160
  zliczanie liczb w liście, 200
programista
  mocne strony, 247
  słabe strony, 245
programowanie
  proceduralne, 144
  sterowane testami, 246
  zorientowane obiektowo, 13, 144, 176
projekt wstępny gry, 256
projektowanie, 242
przeciążanie operatora, operator overloading, 173
przeglądanie listy, 136
przepełnienie sterty, heap overflow, 117
przerost funkcjonalności, 248
przeszukiwanie tablicy, 80
przydzielanie pamięci, 113, 124
przypisanie wielokrotne, 174
punkt przywracania, 267

R

redundancja danych, 157
referencja, 112
referencja do elementu, 229
rekord aktywacji, activation record, 113
rekurencja, 117, 181
  bezpośrednia, 182
  nieogonowa, head recursion, 185, 191, 198
  ogonowa, tail recursion, 184, 189
  pośrednia, 182
rekurencyjne przetwarzanie listy, 199
rozmiar
  pamięci, 116
  struktury, 110
  tablicy, 100
  wektora, 102
rozwiązywanie problemów, 9, 16, 250, 267
  analogie, 37
  dzielenie problemu, 34, 147
  eksperymentowanie, 38
  plan działania, 32
  prezentowanie problemu, 33
  upraszczanie problemu, 36
  za pomocą klas, 143
  za pomocą ponownego wykorzystania kodu, 213
  za pomocą rekurencji, 181
  za pomocą wskaźników, 119

S

schemat klasy, 152
skalowalność rozwiązania, 91
składowa
  chroniona, protected member, 144
  publiczna, public member, 144
słabe strony
  kodowania, 243
  projektowania, 245
słowo kluczowe
  const, 92
  delete, 109, 118
  get, 153
  new, 115
  operator, 173
  this, 153
sortowanie, 81
  przez wstawianie, 83, 216
  rejestrów, 237
  tablicy obiektów, 235
  ze wstawianiem, 235
specyfikator dostępu, access specifier, 144
spowolnienie działania aplikacji, 116
sprawdzanie poprawności, 158
stała ARRAY_SIZE, 81
stan pamięci, 121, 134
standardowe biblioteki, 272
sterta, heap, 113, 212
stos wywołań, runtime stack, 113
stos, stack, 113
stosowanie
  klas, 146
  komponentów, 227, 233
  rekurencji, 198, 207
  tablic, 99
  wskaźników, 109, 111
struktura, 145
  agentStruct, 99
  binaryTreeNode, 208
  listNode, 131
  student, 94, 152
  treeNode, 201
struktury
  danych, 109, 110
  dynamiczne, 130, 160

Kup ksi k
Pole ksi k

strumień cout, 259
styl programowania, 12
suma kontrolna, 49, 57
szablony klas, 178
szamotanie, thrashing, 116

T

tablica, array, 78
  sortArray, 238
  struktur, 94
  typu char, 119
  z wartościami nieskalarnymi, 94
  ze stałymi wartościami, 91
tablice
  dynamiczne, 100, 101
  wielowymiarowe, 96
test dopasowania do wzorca, 261
testowanie, 125, 246
tryb
  dekodowania, 60
  małych liter, 60
  wielkich liter, 60
  znaków interpunkcyjnych, 60, 70
tworzenie
  diagramów, 124
  funkcji iteracyjnej, 194
  iteratora, 227
  klasy, 150, 156, 176
  kodu produkcyjnego, 269
  listy węzłów, 131
  planu głównego, 242, 248
  stosu, 113
typ
  całkowity, 52, 60
  łańcuchowy, 120
  wyliczeniowy, 70

U

uniwersalność komponentu, 232
usuwanie węzła z listy, 165–168

W

walidacja, 158, 159
wartościowanie skrócone, 164
wartość NULL, 118, 163
wektor, 77, 101
węzeł, node, 131
właściwość, 153
WPR, 191
wskaźnik, 107, 119
  do char, 120
  do liczby całkowitej, 118
  do struktury węzła, 205
  głowy, head pointer, 133
wskaźniki
  połączone krzyżowo, 129
  zawieszone, 118
współdzielenie pamięci, 110
wybór komponentu, 232, 234
wyciek pamięci, 124
wydajność, 224
wyjątek bad_alloc, 117
wykorzystanie kodu
  niewłaściwe, 214
  poprawne, 214
wyłuskiwanie, 109
wyrażenie, 44
wyrażenie inicjalizujące, 79
wyszukiwanie
  komponentu, 225, 226
  największej wartości, 98
  określonej wartości, 80
  oparte na kryterium, 81
wzorce projektowe, 216
wzorzec iterator, 227

Z

zapis node1->studentNum, 132
zdobywanie wiedzy
  eksploracyjne, 219
  w praktyce, 219
  w razie potrzeby, 223
zmienne
  globalne, 196
  lokalne, 123, 198
  obiektowe, 270
  statyczne, 197
znak
  %, 51
  &, 108, 110
  [EOL], 68
  gwiazdki, 108
  końca wiersza, 121
  tyldy, 170
zwalnianie pamięci, 109

Kup ksi k
Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

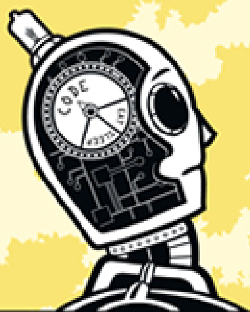
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA



OPANUJ SZTUKĘ MYŚLENIA JAK PROGRAMISTA!

Nauka programowania to w rzeczywistości nauka sposobu myślenia. Jako programista musisz biegle analizować problemy, dzielić je na części oraz starać się je rozwiązać w optymalny sposób. Składnia języka i środowisko programistyczne to tylko podstawowe narzędzia, których obsługi może się nauczyć każdy. Jednak nie każdy potrafi myśleć jak programista.

Dzięki tej książce masz szansę zostać profesjonalistą! W trakcie lektury poznasz najlepsze sposoby rozwiązywania problemów, opanujesz rekurencję i przekonasz się, że wcale nie jest ona taka straszna. Zobaczysz również, jak tworzyć kod nadający się do ponownego użycia, i opanujesz zagadnienia z obszaru programowania obiektowego. Przykłady w tej książce zostały napisane w języku C++, ale nie stanowi to bariery, żeby przenieść te idee na inne języki programowania. Warto poświęcić tej książce parę wieczorów i zmienić swój sposób patrzenia na programowanie!

Poznaj:

- strategie rozwiązywania problemów
- sposoby rozwiązywania problemów za pomocą rekurencji
- korzyści z wykorzystania wskaźników i pamięci dynamicznej
- metody zdobywania nowych umiejętności programistycznych

helion.pl księgarnia internetowa

Nr katalogowy: 14479

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-246-7284-4

Cena: 49,00 zł