

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Zarządzanie projektami informatycznymi. Subiektywne spojrzenie programisty

Autor: Joel Spolsky

Tłumaczenie: Mikołaj Szczepaniak

ISBN: 83-7361-869-4

Tytuł oryginału: [Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity](#)

Format: B5, stron: 360



Poznaj skuteczne i techniki kierowania pracą zespołu programistów

- Dowiedz się, jak przygotować specyfikację funkcjonalną produktu, którą wszyscy odpowiednio rozumieją
- Stwórz możliwy do zrealizowania harmonogram prac nad projektem
- Zbuduj zespół projektowy, zatrudniając odpowiednich ludzi
- Pamiętaj o prawach Murphy'ego

Projekty informatyczne są dość specyficznym typem projektów. Zarządzanie nimi też różni się od tradycyjnego zarządzania projektami. Informatycy to ludzie pracujący w sposób zdecydowanie odmienny od standardowo przyjętego, co z kolei sprawia, że kierowanie zespołem informatyków wymaga odpowiedniego podejścia.

Metody zarządzania projektami zaczerpnięte z innych branż często nie sprawdzają się przy projektach IT. Na szczęście metody skutecznego zarządzania projektami informatycznymi naprawdę istnieją. Poznanie ich jest niezbędne do tego, aby kierowany przez Ciebie projekt zakończył się w terminie i zmieścił w wyznaczonym budżecie.

„Zarządzanie projektami informatycznymi. Subiektywne spojrzenie programisty” to zbiór zasad zarządzania projektami IT spisanych przez programistę, którego niespodziewanie mianowano kierownikiem projektu. Czytając tę książkę, dowiesz się, jak zbudować skutecznie działający zespół programistów, jak stworzyć realny harmonogram prac i pisać specyfikacje, które rzeczywiście okażą się przydatne. Przekonasz się, że stosowanie gotowych rozwiązań dostępnych w internecie nie zawsze zdaje egzamin, i nauczysz się, jak ważne jest testowanie kodu na każdym etapie projektu. Przeczytasz o strategii tworzenia oprogramowania i dowiesz się, dlaczego pozornie skazane na sukces projekty IT upadają.

- 12 zasad skutecznego zarządzania projektami IT
- Tworzenie użytecznych specyfikacji funkcjonalnych
- Planowanie harmonogramu realizacji projektu
- Testowanie i usuwanie błędów
- Tajniki skutecznej rekrutacji
- Korzystanie z rozwiązań open source
- Dobór odpowiedniej technologii

**Naucz się skutecznie zarządzać projektami,
w których dotychczas brałeś udział jako programista**



Spis treści

O Autorze	9
Wprowadzenie	11
Część I Bity i bajty: praktyczne elementy programowania	15
Rozdział 1. Wybór języka programowania	17
Rozdział 2. Powrót do podstaw	19
Rozdział 3. Test Joela: 12 kroków ku lepszemu oprogramowaniu	29
Rozdział 4. Absolutne minimum na temat formatu Unicode i innych systemów kodowania znaków, które nie może być obce żadnemu programiście (bez wyjątku!)	43
Rozdział 5. Niegroźne specyfikacje funkcjonalne. Część 1: Po co to wszystko?	55
Rozdział 6. Niegroźne specyfikacje funkcjonalne. Część 2: Czym właściwie jest specyfikacja?	63
Rozdział 7. Niegroźne specyfikacje funkcjonalne. Część 3: Ale... jak?	73
Rozdział 8. Niegroźne specyfikacje funkcjonalne. Część 4: Wskazówki	77
Rozdział 9. Niegroźne harmonogramy tworzenia oprogramowania	85
Rozdział 10. Codzienne kompilacje są Twoimi sprzymierzeńcami	97
Rozdział 11. Bezwzględne usuwanie błędów	103
Rozdział 12. Pięć światów	109
Rozdział 13. Przygotowywanie papierowych prototypów	117
Rozdział 14. Nie pozwól, aby astronauty architektury decydowali o Twoich projektach	119
Rozdział 15. Ognia i naprzód	123
Rozdział 16. Rzemiosło	127

Rozdział 17. Trzy błędne opinie w świecie informatyki	133
Rozdział 18. Dwukulturowość	139
Rozdział 19. Zbieraj od użytkowników raporty o błędach — rób to automatycznie!	147
Część II Zarządzanie zespołem programistów	157
Rozdział 20. Partyzancki poradnik rekrutacji	159
Rozdział 21. Szkodliwy wpływ „motywującego” systemu kar i nagród	173
Rozdział 22. Pięć wymówek, z powodu których nie korzystamy z pomocy testerów	177
Rozdział 23. Szkodliwe skutki przełączania zadań rozdzielonych pomiędzy pracowników	185
Rozdział 24. Rzeczy, których nigdy nie należy robić: część pierwsza	189
Rozdział 25. Sekret góry lodowej: rozwiązanie	195
Rozdział 26. Prawo Nieszczelnych Abstrakcji	203
Rozdział 27. Lord Palmerston o programowaniu	209
Rozdział 28. Mierniki	217
Część III Być jak Joel: Przypadkowo wybrane opinie na nieprzypadkowe tematy	219
Rozdział 29. Rick Chapman w poszukiwaniu głupoty	221
Rozdział 30. Czym zajmują się psy w tym kraju?	225
Rozdział 31. Realizacja zadań z perspektywy początkującego programisty	231
Rozdział 32. Dwie historie	237
Rozdział 33. McDonald’s kontra The Naked Chef	243
Rozdział 34. Nic nie jest tak proste, na jakie wygląda	249
Rozdział 35. W obronie syndromu NIH	253
Rozdział 36. Pierwszy list w sprawie strategii: Ben & Jerry kontra Amazon	257
Rozdział 37. Drugi list w sprawie strategii: problem jajka i kury	267
Rozdział 38. Trzeci list w sprawie strategii: wróćmy do podstaw!	275
Rozdział 39. Czwarty list w sprawie strategii: Bloatware i mit 80-20	281
Rozdział 40. Piąty list w sprawie strategii: ekonomia otwartego dostępu do kodu źródłowego	285
Rozdział 41. Tydzień szaleństwa prawa Murphy’ego	295
Rozdział 42. Jak Microsoft przegrał wojnę na interfejsy API	299

Część IV Przydługi komentarz na temat technologii .NET	315
Rozdział 43. Microsoft oszalał	317
Rozdział 44. Nasza strategia .NET	323
Rozdział 45. Przepraszam szanownego pana, mogę prosić o program łączący?	327
Dodatki	331
Dodatek A Najlepsze pytania do Joela	333
Skorowidz	347

Rozdział 1.

Wybór języka programowania

Niedziela, 5 maja 2002 roku.

Dlaczego programiści wybierają do realizowanych przez siebie zadań pewne języki programowania, rezygnując z wszystkich pozostałych?

Kiedy potrzebuję przede wszystkim szybkości, często wybieram surowy język C.

Kiedy chcę opracować możliwie mały program, który będzie działał w systemie operacyjnym Windows, zazwyczaj decyduję się na język C++ ze statycznie dołączanymi klasami biblioteki MFC.

Jeśli jednak potrzebuję graficznego interfejsu użytkownika (GUI), który będzie prawidłowo funkcjonował w systemach Mac, Windows i Linux, zazwyczaj wybieram Javę. Chociaż mechanizmy obsługi tego typu interfejsów dostępne w tym języku nie są doskonałe, utworzone rozwiązanie z pewnością będzie przenośne.

Kiedy niezbędne jest zastosowanie techniki szybkiego tworzenia oprogramowania z odpowiednim interfejsem GUI, zwykle używam Visual Basic, chociaż mam świadomość związanych z tym ograniczeń oraz ścisłych związków z jedną platformą (systemem operacyjnym Windows).

W przypadku narzędzi wykonywanych w wierszu poleceń, które będą uruchamiane wyłącznie na komputerach z systemami UNIX i które nie muszą być specjalnie szybkie, możemy użyć Perla.

Jeśli tworzone oprogramowanie ma być wykonywane w przeglądarce internetowej, jedynym sensownym rozwiązaniem jest zastosowanie języka JavaScript. Natomiast w przypadku procedury składowanej SQL-a przeważnie musimy wybrać odpowiednią pochodną standardu języka SQL stosowaną w danym serwerze bazy danych.

Co jest najważniejsze?

Prawda jest taka, że najczęściej wybieramy język programowania, kierując się wyłącznie jego składnią. Zawsze wołałem języki z nawiasami klamrowymi wyznaczającymi poszczególne bloki kodu (C/C++, C# i Javę). Słyszałem już mnóstwo opinii na temat czynników decydujących o tym, czy składnia języka programowania jest *dobra* czy *zła*. Nie dałbym się jednak przekonać do składni, która wymusza zajmowanie 20 MB przez same średniki.

Szczególnie interesującym elementem technologii .NET jest strategia programowania dla wielu platform w różnych językach. Jej idea jest możliwość wybrania dowolnego, odpowiadającego nam języka programowania (których jest mnóstwo) i napisania oprogramowania, którego działanie nie będzie się różnić od funkcjonowania odpowiednich programów napisanych w innych językach.

Języki VB.NET i C#.NET są niemal identyczne — wszelkie niezgodności między nimi wynikają z drobnych różnic składniowych. Wszystkie pozostałe języki programowania, które mają ambicje wkroczyć do świata technologii .NET, muszą obsługiwać przy najmniej minimalny zestaw mechanizmów i typów, który będzie współdziałał z innymi elementami tej technologii. Jednak w jaki sposób można tworzyć w technologii .NET programy wiersza poleceń dla systemu UNIX? Czy ta technologia umożliwia tworzenie małych (mniejszych od 16 KB) programów dla systemu Windows?

Wydaje się zatem, że technologia .NET umożliwia „wybór” języka na podstawie jego cechy, która interesuje nas najbardziej — składni.

Rozdział 2.

Powrót do podstaw

Wtorek, 5 grudnia 2001 roku.

Znaczna część dyskusji prowadzonych na mojej witrynie internetowej dotyczyła „ogólnych” kwestii, takich jak: analiza porównawcza technologii .NET i języka Java, omówienie strategii XML, problem przechodzenia na nowe technologie, zestawianie konkurencyjnych strategii, projektowanie oprogramowania, architektur itp. Każde z tych zagadnień w pewnym sensie przypomina przekładaniec (rodzaj ciasta). Na najwyższym poziomie jest ogólna strategia oprogramowania, bezpośrednio pod nią architektura (np. .NET), pod którą działają poszczególne produkty, np. narzędzia wytwarzania oprogramowania (jak Java lub platformy, np. Windows).

Chcesz poznać kolejne warstwy? Proszę bardzo: biblioteki DLL, obiekty, funkcje. Jeszcze niżej? W końcu dotrzemy do pojedynczych wierszy kodu napisanych w wybranym języku programowania.

Poziom wyrażen języka programowania jest dla Ciebie zbyt wysoki? Poniżej chciałbym się zająć procesorami, czyli kawałkiem krzemu przetwarzającym bajty. Przypuśćmy, że jesteś początkującym programistą. Zapomnij o wszystkim, czego dowiedziałeś się o programowaniu, oprogramowaniu i zarządzaniu, aby wrócić na najniższy poziom podstawowego modelu Von Neumanna. Wyrzuć z pamięci wszelkie doświadczenia związane z pracą w środowisku J2EE. Myśl tylko o *bajtach*.

Dlaczego w ogóle to robimy? Sądzę, że największym błędem popełnianym przez programistów — nawet na najwyższych poziomach architektur oprogramowania — jest niedostateczna lub obciążona błędami wiedza na temat procesów zachodzących na najniższych poziomach. Można w ten sposób zbudować wspaniały pałac na grząskim gruncie; zamiast betonowych fundamentów powstanie niestabilne rumowisko. Taki pałac co prawda ładnie wygląda, ale od czasu do czasu popękają kafelki w łazience i nie będzie wiadomo, co jest źródłem braku stabilności naszego dzieła.

Spróbuj więc, przynajmniej teraz, się odprężyć — zapraszam na krótki spacer z atrakcjami w postaci praktycznych ćwiczeń stosowania języka programowania C.

Pamiętasz sposób obsługi łańcuchów w języku C? Każdy z nich składa się z pewnej liczby bajtów, z których ostatni reprezentuje znak `null` równy 0¹. Wybór takiego rozwiązania wiąże się z dwoma dość oczywistymi następstwami:

1. Brak możliwości sprawdzenia, gdzie dany łańcuch się kończy (a więc wyznaczenia jego długości) bez jego iteracyjnego przeszukania pod kątem miejsca przechowywania kończącego znaku `null`.
2. Łańcuch nie może zawierać żadnych zer, zatem nie można umieszczać w łańcuchach języka C dowolnych nieprzetworzonych danych binarnych, takich jak obrazy w formacie JPEG.

Dlaczego łańcuchy języka programowania C działają w taki sposób? Wynika to wprost z tego, że mikroprocesor PDP-7, dla którego zaprojektowano system operacyjny UNIX i właśnie język C, przetwarzał łańcuchy typu ASCII. Skrót ASCII oznacza „format *ASCII* z *Z* (zerem) na końcu”.

Czy to jedyny sposób składowania łańcuchów w tym języku? Nie. W rzeczywistości jest to jedna z najgorszych istniejących metod przechowywania łańcuchów. W przypadku bardziej rozbudowanych programów, interfejsów API, systemów operacyjnych i bibliotek klas, należy jak ognia unikać łańcuchów typu ASCII. Dlaczego?

Rozważania zacznę od napisania własnej wersji kodu funkcji `strcat`, która dodaje jeden łańcuch na koniec drugiego:

```
void strcat(char* dest, char* src)
{
    while(*dest) dest++;
    while(*dest++ = *src++);
}
```

Wystarczy krótka analiza tej funkcji, aby przekonać się, jakie jest jej faktyczne działanie. Po pierwsze, przeglądnij pierwszy łańcuch w poszukiwaniu kończącego znaku `null`. Kiedy uda się to zrobić, zobacz iteracyjnie znaki drugiego łańcucha i kolejno je skopiuuj do pierwszego.

Takie podejście do problemu obsługi i konkatencji łańcuchów było może dobre w czasie, gdy swoje opracowanie przygotowywali Kernighan i Ritchie², ale w rzeczywistości stwarza mnóstwo problemów. Oto jeden z nich. Przypuśćmy, że chcesz umieścić zbiór imion w jednym, dużym łańcuchu:

```
char bigString[1000]; /* Nigdy nie wiem, ile przydzielić pamięci... */
bigString[0] = '\0';
strcat(bigString, "Jan, ");
strcat(bigString, "Paweł, ");
strcat(bigString, "Jerzy, ");
strcat(bigString, "Józef ");
```

¹ Więcej informacji na temat łańcuchów znaków w języku programowania C znajdziesz na stronie internetowej www-ee.eng.hawaii.edu/Courses/EE150/Book/chap7/subsection2.1.1.2.html.

² B. Kernighan, D. Ritchie, *The C Programming Language, Second Edition*, Prentice Hall 1988.

Powyższy przykład powinien działać, prawda? Tak. Raczej nie będzie z nim żadnych problemów.

Jaka jest efektywność zastosowanej techniki? Czy nasz program jest wystarczająco szybki? Czy nie można zrobić tego szybciej? Czy jest skalowalny? Czy mając milion łańcuchów, powinieneś użyć analogicznego rozwiązania?

Nie. Zaprezentowany kod wykorzystuje *algorytm roztargnionego malarza*. Czym malarz zasłużył sobie na ten przydomek? Powinien to wyjaśnić poniższy dowcip:

Malarz dostał zlecenie pomalowania jezdni — narysowania linii przerywanej na środku ulicy. Pierwszego dnia położył na krawężniku puszkę farby i pomalował 100 metrów. „Nieźle!” — usłyszał od swojego szefa. — „Jesteś naprawdę szybki!”. Do kieszeni malarza trafiła złotówka premii.

Następnego dnia udało mu się pomalować tylko 50 metrów drogi. „Cóż, dziś nie było tak dobrze jak wczoraj, ale nadal jesteś wydajnym pracownikiem. 50 metrów to także sporo”. Malarz znowu dostał złotówkę premii.

Następnego dnia pomalował tylko 10 metrów jedni. „Co? 10 metrów?” — krzyknął rozszłoszczony szef. — „To nie do przyjęcia! Pierwszego dnia pomalowałeś 10 razy dłuższy odcinek! Co się dzieje?”.

„Nic na to nie poradzę” — odparł malarz. — „Codziennie mam dalej do mojej puszkę z farbą!”.

(Pytanie dodatkowe: jakie naprawdę powinny być liczby wymienione w tym dowcipie?³). Ten przeciętny dowcip dosyć dokładnie oddaje to, co dzieje się podczas wywoływania przedstawionej przed momentem funkcji `strcat`. Ponieważ pierwsza część tej funkcji za każdym razem przegląda cały łańcuch docelowy w poszukiwaniu kończącego znaku `null`, funkcja jest zdecydowanie za wolna i nie zapewnia odpowiedniej skalowalności. Okazuje się, że ten sam problem dotyczy znacznej części wykorzystywanego na co dzień kodu. Wiele systemów plików jest implementowanych w taki sposób, że umieszczenie zbyt dużej liczby plików w pojedynczym katalogu ma bardzo negatywny wpływ na wydajność ich przetwarzania, ponieważ efektywność tego typu operacji dramatycznie spada wraz ze wzrostem liczby obsługiwanych elementów (w tym przypadku tysięcy plików). Aby się o tym przekonać, spróbuj otworzyć bardzo zapchany *Kosz* w systemie Windows — zanim zostaną wyświetlone umieszczone tam pliki, minie mnóstwo czasu, który z pewnością nie jest liniowo zależny od liczby tych plików. Autorzy systemu widocznie musieli gdzieś zastosować algorytm roztargnionego malarza. Za każdym razem, gdy stwierdzisz, że mechanizm, którego wydajność powinna rosnąć liniowo wraz z poszerzanym zbiorem danych wejściowych, funkcjonuje raczej jak mechanizm kwadratowo zależny od ilości tych danych, koniecznie powinieneś poszukać w kodzie ukrytych malarzy z puszkami farby na krawężniku. Tego typu niedociągnięcia często można znaleźć w naszych bibliotekach. Widok ciągu wywołań funkcji `strcat` lub pętli z takimi wywołaniami często nie jest wystarczającym sygnałem, że ma się do czynienia ze złożonością kwadratową, a właśnie tak jest w tym przypadku.

³ Matematyczną analizę tego zagadnienia znajdziesz na stronie discuss.fogcreek.com/techInterview/default.asp?cmd=show&ixPost=153.

Jak można to poprawić? Kilku sprytnych programistów języka C zaimplementowało własne metody `mystrcat` w następujący sposób:

```
char* strcat(char* dest, char* src)
{
    while(*dest) dest++;
    while(*dest++ = *src++);
    return --dest;
}
```

Co tak naprawdę zrobiono? Minimalnym kosztem udało się wprowadzić mechanizm zwracania wskaźnika na koniec *nowego*, dłuższego łańcucha. Dzięki temu kod wywołujący naszą funkcję może wymusić dodawanie kolejnych łańcuchów bez konieczności ponownego przeszukiwania łańcucha docelowego:

```
char bigString[1000]; /* Nigdy nie wiem, ile przydzielić pamięci... */
char *p = bigString;
bigString[0] = '\0';
p = strcat(p, "Jan, ");
p = strcat(p, "Paweł, ");
p = strcat(p, "Jerzy, ");
p = strcat(p, "Józef ");
```

Takie rozwiązanie oczywiście gwarantuje liniową (zamiast kwadratowej) złożoność obliczeniową, zatem nie jest narażone na spadek wydajności w przypadku wielokrotnego wykonywania operacji łączenia łańcuchów.

Projektanci języka programowania Pascal mieli świadomość tego problemu i „rozwiązali” go przez umieszczanie licznika bajtów w pierwszym bajcie łańcucha. Łańcuchy konstruowane na podstawie tego mechanizmu są w związku z tym nazywane *łańcuchami Pascala*. Ponieważ jednak największa wartość, którą można reprezentować za pomocą pojedynczego bajta, to 255, długości łańcuchów Pascala jest ograniczona do 255 znaków; ale ponieważ łańcuchy nie zawierają kończącego znaku `null`, ostatecznie zajmują tyle samo pamięci co łańcuchy ASCII. Ogromną zaletą łańcuchów Pascala jest to, że nigdy nie musisz stosować kosztownych pętli tylko po to, by określić ich długość. Zamiast tego, wyznaczenie długości łańcucha Pascala wymaga użycia pojedynczego polecenia. Różnica w wydajności obu mechanizmów jest ogromna.

Stary system operacyjny Macintosh wykorzystywał łańcuchy Pascala wszędzie, gdzie się dało. Wielu programistów języka C pracujących na innych platformach wykorzystywało łańcuchy Pascala właśnie ze względu na ich szybkość. Są one wewnętrznie wykorzystywane np. w Excelu — to wyjaśnia, dlaczego w wielu miejscach ich długość jest ograniczana do 255 bajtów, ale też pozwala odpowiedzieć na pytanie, z czego wynika niesamowita szybkość Excela.

Bardzo długo było tak, że kiedy chciałeś umieścić łańcuch Pascala w naszym kodzie języka C, musiałeś używać podobnych operacji przypisania:

```
char* str = "\006Witaj!";
```

To prawda, musiałeś samodzielnie zliczać bajty i kodować je na stałe w pierwszym bajcie tworzonego łańcucha. Bardziej leniwi programiści stosowali następujący zabieg, który dodatkowo spowalniał ich programy:

```
char* str = "Witaj!";  
str[0] = strlen(str) - 1;
```

Zwróć uwagę, że mamy w tym przypadku do czynienia zarówno z łańcuchem zakończonym znakiem `null` (za wstawienie tego znaku odpowiada kompilator), jak i z łańcuchem Pascala. Zwykle nazywałem tego typu konstrukcje *cholernymi łańcuchami*, ponieważ używanie sformułowania *łańcuchy Pascala zakończone znakiem null* zajmowało zbyt wiele czasu. Ponieważ jednak ta książka może trafić także w ręce dzieci, trzeba raczej używać tego drugiego, dość niezręcznego określenia.

Odszedłem trochę od zagadnienia, które interesuje nas najbardziej. Pamiętasz ten wiersz kodu?

```
char bigString[1000]; /* Nigdy nie wiem, ile przydzielić pamięci... */
```

Ponieważ mieliśmy się zająć bitami, z pewnością nie zignoruję problemu przydziału właściwej ilości pamięci. Powinienem podejść do tej kwestii bardziej profesjonalnie, a więc określić liczbę potrzebnych bajtów i przydzielić odpowiedni obszar w pamięci.

Myślisz, że nie powinienem?

Warto pamiętać, że jakiś sprytny haker, gdy uzyska dostęp do mojego kodu źródłowego, zauważy, że przydzielam tylko 1000 bajtów *w nadziei*, że to wystarczy. Wtedy znajdzie sposób zmuszenia mojego programu do połączenia w moim łańcuchu danych o łącznej długości 1100 bajtów. W ten sposób nadpisze ramkę stosu i tak zmieni adres zwracania, aby moja funkcja, kończąc pracę, przekazywała sterowanie do kodu napisanego przez tego hakera. Właśnie tak w praktyce wygląda problem nazywany często wrażliwością na atak *przepełnienia bufora*. Taki błąd był pierwszym wykrytym słabym punktem programu Microsoft Outlook, który umożliwiał włamania komputerowe nawet początkującym adeptom tej sztuki.

No dobrze, przyjmijmy, że wszyscy programiści są równie leniwi. Tak czy owak, powinni zawsze starać się w rozsądny sposób określać, ile pamięci będzie potrzebne do składowania definiowanej struktury danych.

Język C z pewnością tego *nie* ułatwia. Wróć do przykładu z imionami:

```
char bigString[1000]; /* Nigdy nie wiem, ile przydzielić pamięci... */  
char *p = bigString;  
bigString[0] = '\0';  
p = strcat(p, "Jan, ");  
p = strcat(p, "Paweł, ");  
p = strcat(p, "Jerzy, ");  
p = strcat(p, "Józef ");
```

Ile pamięci powinno się przydzielić? Spróbuj zrobić to zadanie w jedyny słuszny sposób:

```
char bigString;  
int i = 0;  
i = strlen("Jan, ")  
+ strlen("Paweł, ")  
+ strlen("Jerzy, ");
```

```
+ strlen("Józef ");
bigString = (char*)malloc(i+1);
/* Pamiętaj o przestrzeni na kończący znak null! */
...
```

Nie wierzę własnym oczom! Myślę, że jednak nie powinieneś pokazywać tej książki swoim dzieciom. Nie chcę zrzucić winy na Ciebie, więc trzymaj się blisko, bo mamy do czynienia z czymś naprawdę interesującym.

Musisz przeglądać wszystkie łańcuchy wejściowe tylko po to, by określić, jak długi powinien być łańcuch wyjściowy, dopiero potem przeglądaj je ponownie w trakcie właściwej operacji ich łączenia. W przypadku łańcuchów Pascala przynajmniej operacja `strlen` była szybka. Może powinieneś spróbować napisać taką wersję funkcji `strcat`, która będzie automatycznie przydzielała odpowiedni obszar pamięci?

Przedstawione rozwiązanie wywołuje kolejne problemy — tym razem pułapką jest przydzielanie pamięci (tzw. *alokatorów pamięci*). Czy wiesz, jak działa funkcja `malloc`? Jej działanie bazuje na utrzymywanej jednokierunkowej liście dostępnych bloków pamięci nazywanej *wolnym łańcuchem*. Kiedy wywołujesz funkcję `malloc`, lista ta jest przeglądana w poszukiwaniu pierwszego bloku pamięci, który będzie odpowiednio duży, by zrealizować bieżące żądanie (pomieścić wskazaną strukturę danych). Wybrany blok jest następnie dzielony na dwa — jeden o żądanym rozmiarze i drugi zawierający bajty nadmiarowe. Pierwszy blok przechowuje naszą strukturę danych, drugi (jeśli istnieje) jest z powrotem umieszczany na liście wolnych bloków. Kiedy wywołasz funkcję `free`, zwolniony blok zostanie ponownie dołączony do listy wolnych bloków pamięci. Z czasem bloki pamięci są więc dzielone na coraz mniejsze kawałki i kiedy zażadasz większego obszaru pamięci, może się okazać, że odpowiedni blok nie istnieje. Funkcja `malloc` poprosi wówczas o czas i zacznie naprawiać wolny łańcuch, sortując całą listę (według adresów) i łącząc sąsiadujące ze sobą małe bloki danych w coraz większe obszary. Może to zabrać mnóstwo czasu. Efekt jest taki, że funkcja `malloc` nigdy nie gwarantuje należytej szybkości, ponieważ za każdym razem musi przeszukać listę wolnych bloków pamięci, a niekiedy (w trudnych do przewidzenia momentach) dodatkowo porządkuje tę listę i wydajność całej funkcji jest bardzo mała (wydajność funkcji `malloc` przypomina wówczas raczej mechanizmy czyszczenia pamięci, zatem wszelkie teorie na temat niepotrzebnych opóźnień powodowanych przez procedury czyszczenia pamięci są nie do końca prawdziwe, ponieważ typowa implementacja funkcji `malloc` jest obciążona ryzykiem bardzo podobnych opóźnień, chociaż być może w mniejszym zakresie).

Sprytniejsi programiści minimalizują potencjalne opóźnienia generowane przez funkcję `malloc`, stosując operacje przydzielania bloków pamięci, których rozmiary są zawsze potęgami liczby 2; wiadomo: 2 bajty, 4 bajty, 8 bajtów, 16 bajtów, 18446744073709551616 bajtów itd. Z oczywistych względów (przynajmniej dla osób, które kiedykolwiek bawiły się klockami lego) w ten sposób można zminimalizować liczbę niepożądanych operacji fragmentowania bloków z wolnego łańcucha. Chociaż takie działania są z pozoru niepotrzebną stratą przestrzeni pamięciowej, nietrudno obliczyć, że maksymalna nadwyżka tej przestrzeni nigdy nie przekroczy 50 procent. Nasz program nie zajmie więc więcej niż dwukrotność pamięci, której rzeczywiście potrzebuje, co w większości przypadków nie stanowi większego problemu.

Przypuśćmy, że napisałeś udoskonaloną wersję funkcji `strcat`, która w razie konieczności automatycznie przydziela potrzebną pamięć dla bufora danych wyjściowych (łańcucha docelowego). Czy ta funkcja zawsze powinna przydzielać łańcuchowi obszar dokładnie odpowiadający jego rozmiarowi? Mój nauczyciel i mentor, Stan Eisenstat⁴, zasugerował mi kiedyś, że wywołując funkcję `realloc`, zawsze powinienem podwajać obszar pamięci, który był wcześniej przydzielony. W takim przypadku nigdy nie musiałbym wywołać tej funkcji więcej niż $\lg(n)$ razy, co jest niezłym wynikiem nawet dla ogromnych łańcuchów (co więcej, takie rozwiązanie oznacza, że nigdy bezproduktywnie nie zajmuje się więcej niż 50 procent pamięci).

Tak czy inaczej tu, w świecie bajtów, życie stale się komplikuje. Czyż nie byłeś zadowolony, kiedy odkryłeś, że nie musisz już programować w języku C? Mamy obecnie do dyspozycji tak znakomite języki programowania, jak: Perl, Java, Visual Basic oraz XSLT, które całkowicie zwalniają z obowiązku analizowania opisanych przed chwilą problemów — po prostu w jakiś sposób same o wszystko dbają. Niekiedy jednak coś przestaje działać w najmniej spodziewanym czasie i miejscu (np. instalacja wodna pęka na środku pokoju gościnnego) i dopiero wówczas musisz się zastanowić, czy należy używać klasy `String` czy może `StringBuilder`. Podobne problemy najczęściej wynikają z tego, iż współczesne kompilatory nadal nie potrafią same ocenić, co próbujemy osiągnąć za pomocą pewnych struktur i na siłę ratują nas przed przypadkowym wprowadzeniem w kodzie algorytmu roztargnionego malarza.

Niniejsze rozważania sprowokowałem, publikując na swojej witrynie internetowej krótką rozprawę, w której stwierdziłem, że nie jest możliwe zaimplementowanie szybkiej obsługi typowego zapytania języka SQL (np. `SELECT author FROM books`), jeśli przetwarzane dane są składowane w formacie XML⁵. Na wypadek, gdyby ktoś wówczas nie zrozumiał, co miałem na myśli i niejako przy okazji omawiania efektywnego wykorzystywania procesora, warto bliżej przeanalizować ten problem.

W jaki sposób relacyjne bazy danych implementują zapytanie `SELECT author FROM books`? W relacyjnej bazie danych każdy wiersz tabeli (w tym przypadku tabeli `books`) ma dokładnie taką samą długość (wyrażoną w bajtach) jak wszystkie pozostałe. Co więcej, każde pole takiego wiersza jest przesunięte o stałą liczbę bajtów względem początku wiersza. Przykładowo, jeśli każdy wiersz tabeli `books` ma 100 bajtów, a pole `author` jest przesunięte względem początku wiersza o 23 bajty, to nazwiska autorów są przechowywane w bajtach 23, 123, 223, 323 itd. Jak wobec tego powinien wyglądać kod przechodzący do następnego rekordu w wyniku tego zapytania? Na przykład tak:

```
pointer += 100;
```

To tylko jeden rozkaz procesora, musi więc działać baaardzo szybko.

Zobacz teraz tabelę `books` w wejściowym dokumencie XML:

```
<?xml bla bla bla>
<books>
  <book>
    <title>UI Design for Programmers</title>
```

⁴ Patrz strona internetowa www.cs.yale.edu/people/faculty/eisenstat.html.

⁵ Patrz strona internetowa www.joelonsoftware.com/articles/fog0000000296.html.

```
<author>Joel Spolsky</author>
</book>
<book>
  <title>The Chop Suey Club</title>
  <author>Bruce Weber</author>
</book>
</books>
```

Krótkie pytanie: jak powinien wyglądać kod przechodzący do następnego rekordu?

Hm...

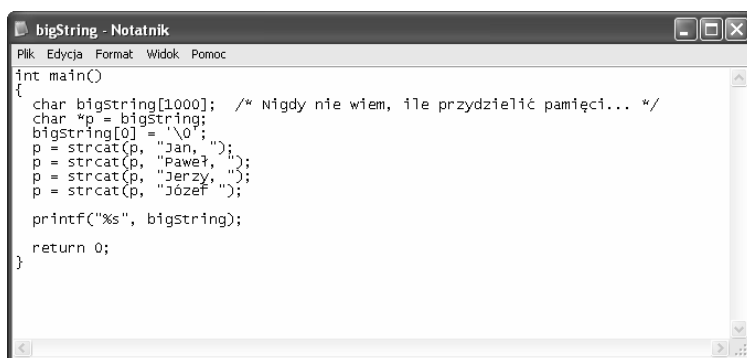
Dobry programista powiedziałby teraz, że należy przetworzyć ten kod XML i zbudować w pamięci odpowiednie drzewo, które umożliwiłoby stosunkowo szybką pracę z tymi danymi. Ilość pracy, jaką w takim przypadku musiałby wykonać nasz procesor (tylko po to, by wykonać zapytanie `SELECT author FROM BOOKS`), niejednego człowieka zanudziłaby na śmierć. Każdy, kto miał kiedyś do czynienia z pisanem kompilatorów, doskonale zdaje sobie sprawę z tego, że właśnie analiza leksykalna i składniowa stanowi najwolniejszą część całego procesu kompilacji. Wystarczy powiedzieć, że takie przetwarzanie danych (analiza leksykalna, składniowa i budowa drzewa) wymaga stosowania wielu operacji na łańcuchach, które — jak wiemy — są bardzo wolne. Co więcej, rozwiązanie bazujące na drzewie zakłada, że *dysponujemy* odpowiednią ilością pamięci, aby umieścić tam wszystkie dane wejściowe. W przypadku relacyjnych baz danych czas potrzebny na przejście do kolejnego rekordu jest stały i w praktyce wymaga użycia *pojedynczego rozkazu procesora*. To bardzo duża różnica, a dzięki plikom odwzorowania pamięci dodatkowo istnieje możliwość wczytywania tylko tych stron pamięci dyskowej, które będą po chwili przetwarzane. W przypadku danych zapisanych w formacie XML, jeśli przeprowadzi się operację wstępnego przetworzenia i umieszczenia potrzebnych informacji w pamięci, czas przechodzenia pomiędzy rekordami także będzie stały, ale bardzo wydłuży się czas uruchamiania aplikacji. Jeśli natomiast zrezygnuje się z przetwarzania tych danych na początku, czas wykonywania operacji przejścia z jednego rekordu do następnego będzie się różnić w zależności od długości bieżącego rekordu, a operacja ta zawsze będzie wymagała wykonania setek rozkazów procesora.

Z przedstawionej analizy wynika, że nie można stosować formatu XML, jeśli przy dużej ilości danych oczekuje się wysokiej wydajności systemu. Jeśli jednak operujemy na niewielkiej ilości danych lub jeśli opracowywane oprogramowanie nie musi być szybkie, format XML w zupełności wystarczy. Gdy chcemy korzystać z zalet obu światów (relacyjnych baz danych i formatu XML), trzeba opracować mechanizm składowania takich metadanych dołączanych do tradycyjnych dokumentów XML (jak w przypadku licznika bajtów w łańcuchach Pascala), które będą stanowiły czytelne wskazówki o lokalizacji danych, aby nie musieć ich dodatkowo przetwarzać. Wówczas nie będzie jednak można wykorzystywać edytorów tekstu do modyfikowania naszych plików XML, ponieważ mogłoby to zniszczyć te metadane — nie będą to więc prawdziwe dokumenty XML.

Zwracam się do kilku czytelników, którzy dotarli ze mną aż tutaj — mam nadzieję, że moje rozważania czegoś was nauczyły i zmobilizowały do ponownego przemyślenia poruszonych kwestii. Myślę, że przedstawiona analiza nudnych zagadnień, typowych dla pierwszego roku studiów informatycznych (jak choćby faktycznego działania funkcji `strcat` i `malloc`) pokazuje, że warto czasem wrócić do korzeni podczas podejmowania

decyzji odnośnie strategii i architektury nowoczesnych rozwiązań na podstawie technologii XML. Jako zadanie domowe zastanów się, dlaczego układy firmy Transmeta są tak powolne. Spróbuj także stwierdzić, dlaczego oryginalna specyfikacja języka HTML dla tabel została tak skonstruowana, że użytkownicy modemów analogowych tracą mnóstwo czasu, czekając na otwarcie stron z wielkimi tabelami. Odpowiedz też na pytanie, dlaczego komponenty modelu COM są bardzo szybkie tylko do momentu, w którym zmusisz je do przetwarzania międzyprocesowego. I dlaczego programiści, którzy zaprojektowali system NT, umieścili sterowniki ekranu w przestrzeni jądra zamiast w przestrzeni użytkownika?

Wszystkie te kwestie wymagają przeprowadzenia odpowiedniej analizy na poziomie bajtów, a uzyskane w ten sposób odpowiedzi mają zasadniczy wpływ na decyzje podejmowane na najwyższym poziomie — podczas doboru architektury i strategii. Dlatego właśnie moja wizja nauczania studentów pierwszego roku informatyki przewiduje konieczność wprowadzania technik programowania od podstaw, czyli najlepiej od języka programowania C wraz z wyjaśnieniem rozkazów procesora kryjących się za poszczególnymi funkcjami tego języka. Jestem szczerze zadowolony, kiedy słyszę, że wiele programów nauczania informatyki traktuje Javę jako dobry język do nauki, tylko dlatego że Java jest „prosta” — nie wymaga analizy tych wszystkich kwestii związanych z łańcuchami i przydzielaniem pamięci, a pozwala szybko budować programy obiektowe złożone z wielu modułów. Takie podejście jest z pedagogicznego punktu widzenia nie do przyjęcia i w niedługim czasie doprowadzi do katastrofy. Całe pokolenia magistrów informatyki będą na każdym kroku wprowadzały do oprogramowania algorytmy roztargnionego malarza, nawet nie zdając sobie z tego sprawy, ponieważ będzie im brakowało elementarnej wiedzy na temat faktycznych technik reprezentowania łańcuchów, które nie są przecież widoczne w kodzie Perla. Jeśli chcesz kogoś nauczyć czegoś naprawdę pożytecznego, musisz rozpocząć zajęcia od wprowadzenia zagadnień związanych z najniższym poziomem funkcjonowania oprogramowania. Trzeba najpierw przez trzy tygodnie przekazywać najprostsze i jednocześnie najistotniejsze informacje, aby przygotować właściwy grunt do skutecznego rozwiązywania naprawdę trudnych problemów.



```
bigString - Notatnik
Plik Edycja Format Widok Pomoc

int main()
{
    char bigString[1000]; /* Nigdy nie wiem, ile przydzielić pamięci... */
    char *p = bigString;
    bigString[0] = '\0';
    p = strcat(p, "Jan, ");
    p = strcat(p, "Paweł, ");
    p = strcat(p, "Jerzy, ");
    p = strcat(p, "Józef ");

    printf("%s", bigString);

    return 0;
}
```