Jak pisać przenośny kod. Wstęp do programowania wieloplatformowego

Autor: Brian Hook
Tłumaczenie: Maciej Jezierski
ISBN: 83-246-0625-4
Tytuł oryginału: Write Portable Code: An Introduction to Developing Software for Multiple Platforms
Format: B5, stron: 272

Przenieś swoje programy na inne platformy systemowe

- Poznaj techniki przenoszenia kodu
- Stwórz uniwersalne interfejsy użytkownika
- Uwzględnij różnice pomiędzy systemami operacyjnymi

W branży informatycznej ogromny nacisk kładzie się na jak najszybsze ukończenie produktu dla konkretnej platformy. Jednak gdy produkt staje się popularny, a użytkownicy innych systemów operacyjnych oczekują od producenta wersji możliwej do uruchomienia na swoim sprzęcie, wiele firm staje przed poważnym problemem. Kod źródłowy tworzony z myślą o określonym systemie operacyjnym lub procesorze zawiera elementy bardzo trudne do „przełożenia" na inną platformę. Istnieją jednak techniki programowania, których zastosowanie zdecydowanie ułatwia późniejszą konwersję, a odpowiednie wykorzystanie nie przedłuża czasu przygotowania innej wersji aplikacji.

Czytając książkę „Jak pisać przenośny kod. Wstęp do programowania wieloplatformowego", poznasz te techniki. Znajdziesz tu uniwersalne zasady tworzenia przenośnego oprogramowania. Dowiesz się, jak zaplanować nowy projekt tak, aby jego przeniesienie na inną platformę nie stanowiło problemu. Nauczysz się przerabiać istniejący kod i dostosowywać go do specyfiki innych platform. Poznasz sposoby unikania błędów wynikających z różnic pomiędzy platformami. Znajdziesz w tej książce także przykłady i wzorce, dzięki którym będziesz w stanie tak pisać kod, aby przeniesienie go na inną platformę odbywało się szybko i bez kłopotów.

- Planowanie procesu przenoszenia kodu
- Techniki wykorzystywane podczas przenoszenia kodu
- Systemy kontroli plików źródłowych
- Różnice pomiędzy procesorami
- Preprocesory i kompilatory
- Tworzenie interfejsów użytkownika
- Implementacja operacji sieciowych
- Korzystanie z bibliotek dynamicznych
- Operacje na systemie plików
- Lokalizacja aplikacji

Stwórz uniwersalne programy

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

1

Koncepcje przenośności

ZANIM ZAGŁĘBIMY SIĘ W SZCZEGÓŁY PRZENOSZENIA, MUSIMY WYKONAĆ KROK W TYŁ I SKUPIĆ SIĘ NA SAMEJ KONCEPCJI PRZENOŚNOŚCI. OCZYWIŚCIE ŁATWO BYŁOBY OD RAZU RZUCIĆ SIĘ NA GŁĘBOKĄ WODĘ i rozpocząć od pokazania konkretnego przykładu przerobienia programu z Linuksa tak, żeby można go było uruchomić w Windows, ale wtedy odniósłbym się tylko do jednego aspektu przenośności.

W niniejszym rozdziale omówię różne kwestie dotyczące tworzenia przenośnego oprogramowania: zasady, techniki i działania podejmowane w celu umożliwienia przenoszenia pozwalające na pisanie programów, które mogą być łatwo przenoszone z jednej platformy na drugą, bez względu na specyfikę pierwotnego i docelowego środowiska. Poszczególnymi problemami i ich rozwiązaniem zajmę się, jak tylko przebrniemy przez podstawy.

Przenośność jest stanem umysłu, nie sposobem programowania

Programowanie jest często działaniem podzielonym na kilka etapów. Twój mózg przełącza kolejne biegi, w miarę jak edytujesz, kompilujesz, wyszukujesz błędy, optymalizujesz, dokumentujesz i testujesz. Możesz skłaniać się ku temu, żeby zrobić z przenoszenia osobny etap, taki jak edycja czy wyszukiwanie błędów, ale „myślenie przenośne" nie jest kolejnym krokiem, jest całościowym stanem umysłu, który powinien być aktywny przy każdym poszczególnym działaniu podejmowanym przez programistę. Gdzieś w Twoim umyśle pomiędzy zasadami „nadawaj znaczące nazwy zmiennym" a „nie koduj na sztywno tych stałych" cały czas powinno być obecne „myśl przenośnie".

Podobnie jak chwasty rozprzestrzeniające się w Twoim ogrodzie, problemy przenośności mają w zwyczaju przenikać cały proces tworzenia oprogramowania. Jeśli kodowanie jest procesem zmuszania komputera do wykonania poszczególnych zestawów działań poprzez przemawianie do niego językiem, który rozumie, a tworzenie przenośnego oprogramowania jest procesem unikania wszelkich zależności czy założeń dotyczących danego komputera, to istnieje pośredni, ale wyraźny konflikt pomiędzy tymi zadaniami. Wymaganie, żeby coś działało na bieżącej platformie, rywalizuje z chęcią, żeby działało równie dobrze na pozostałych platformach.

To ważne, żeby widzieć różnice pomiędzy przenoszeniem kodu a pisaniem przenośnego kodu. Pierwsze jest lekarstwem, drugie profilaktyką. Jeśli miałbym wybór, wolałbym raczej uodpornić programistę na złe nawyki teraz, niż próbować naprawiać efekty uboczne tych praktyk później. To „szczepienie" można osiągnąć przez praktykowanie nawyków przenośnego kodowania tak energicznie, że proces ten stanie się drugą naturą — głęboko zakorzenionym, intuicyjnym zrozumieniem, istniejącym cały czas w podświadomości programistów.

Rozwijaj dobre przenośne nawyki

Bardzo często programiści po raz pierwszy zaznajamiający się ze światem przenośnego oprogramowania za bardzo przejmują się poszczególnymi technikami czy problemami, ale doświadczenie uczy ich, że przenośność można osiągnąć znacznie łatwiej dzięki nawykom i filozofii, która zachęca, jeśli nie zmusza, programistę do pisania przenośnego kodu. Żeby wytworzyć dobre nawyki przenośnego programowania, musisz po pierwsze odrzucić pokusę skupiania się na szczegółach takich jak uporządkowanie bajtów czy problemy wyrównania.

Nie ma znaczenia, jak dobrze znasz przenośność w teorii — bardzo często praktyka obnaży braki w tej teorii. Teoretycznie pisanie kodu zgodnego ze standardami powinno zaowocować tym, że kod będzie bardziej przenośny. Jednakże przyjęcie takiego założenia bez przetestowania go może prowadzić do powstania wielu różnych problemów. Na przykład nie ma znaczenia, czy standard ANSI C

nakazuje określone zachowanie, jeśli niekompatybilny lub wadliwy kompilator po prostu nie trzyma się standardu. Kod zgodny ze standardem nie pomoże, jeśli używasz niezgodnych narzędzi.

Klasycznym tego przykładem jest Microsoft Visual C++ 6.0, który nieprawidłowo obsługiwał specyfikację C++ dotyczącą zasięgu zmiennych w instrukcji `for`:

```
for( int i = 0; i< 100; i++)
/*  */;
i = 10; /* W MSVC++ 6.0 ta zmienna ciągle istnieje... W kompilatorach zgodnych ze
           standardem jest poza zasięgiem, zaraz po opuszczeniu pętli for, i w związku
           z tym ta linia w kompilatorach zgodnych ze standardem spowoduje
           pojawienie się błędu */
```

Programiści Microsoftu poprawili to zachowanie w wersji 7.x swojego kompilatora C++, jednakże spowodowało to z kolei problemy z wsteczną kompatybilnością kodu napisanego dla wersji 6, powodując tym samym niezgodność z domyślnym zachowaniem. Oznacza to, że programista może przewidująco pisać kod w sposób, który uważa za bezpieczny i przenośny w kompilatorze Microsoft Visual C++ tylko po to, by odkryć, że nie będzie on działać podczas kompilacji na kompilatorze GNU Compiler Collection (GCC).

Ale takie problemy możesz łatwo wyłapać, jeśli ćwiczysz dobre praktyki programowania przenośnego, takie jak częste testowanie i tworzenie oprogramowania w wielu środowiskach w miarę postępów prac. To zaoszczędzi Ci kłopotów z zapamiętywaniem wszystkich specyficznych błędów i standardowych osobliwości, które możesz napotkać.

Dobre nawyki są lepsze od szczegółowej znajomości błędów i standardów

Popatrzmy więc, jak wyglądają te dobre nawyki.

Przenoś wcześnie i przenoś często

Żaden kod nie jest przenośny, zanim nie zostanie przeniesiony, dlatego też powinieneś przenosić swój kod wcześnie i często. Pozwoli to uniknąć powszechnej pomyłki pisania „przenośnego" kodu i testowania go później, na dalszym etapie tworzenia kodu, tylko po to, żeby odkryć wszystkie małe problemy z przenośnością za jednym razem. Dzięki testowaniu przenośności swojego kodu wcześnie możesz wyłapać wszystkie problemy wtedy, kiedy jeszcze masz czas na ich poprawienie.

Twórz w środowisku heterogenicznym

Można uniknąć dwuetapowego procesu „napisz i przenieś", tworząc w środowisku heterogenicznym. Ten nawyk pozwala także zminimalizować ryzyko zanikania podstaw kodu, kiedy jedna część Twojego projektu rozwija się, podczas gdy inna pozostaje w tyle z powodu braku uwagi programistów.

Na przykład na wczesnym etapie projektu mogłeś uruchamiać swój kod w Linuksie, Mac OS X i Windows, ale ze względu na terminy wersja dla Linuksa stała się nieaktualna. Pół roku później musisz uruchomić swoje oprogramowanie w Linuksie, ale odkrywasz, że wiele zmian nie zostało uwzględnionych na tej platformie (na przykład z powodu warunkowych dyrektyw kompilacji).

Pierwszym krokiem przy tworzeniu w środowisku heterogenicznym jest upewnienie się, że programiści używają tylu różnych systemów i narzędzi, ile z praktycznego punktu widzenia jest rozsądne. Jeśli dostarczasz projekt, który będzie używany w systemie Solaris na procesorze Sun Sparc, w Microsoft Windows na procesorze Intela i w Mac OS X na komputerach Macintosh, upewnij się, że wszyscy członkowie zespołu używają wszystkich tych systemów jako podstawowych systemów tworzenia oprogramowania. Jeśli nie wymaga się od nich używania wszystkich tych systemów jako podstawowych systemów produkcyjnych, może wziąć górę skłonność do podejścia „najpierw niech działa, przeniesiemy to później".

TWORZENIE W WIELU ŚRODOWISKACH NA PRZYKŁADZIE BIBLIOTEKI SAL

Bibliotekę Simple Audio Library (SAL) pisałem równocześnie na laptopie z systemem Windows XP, używając Microsoft Visual C++ 6.0, na komputerze Apple G4 Power Mac z systemem OS X 10.3 (używając XCode/GCC), na komputerze z procesorem AMD Athlon XP i systemem Linux (ArkLinux, jedna z dystrybucji oparta na Red Hat), używając GCC i sporadycznie wykorzystując Embedded Visual C++ dla Pocket PC. Większość kodu napisałem na komputerze z systemem Windows XP, sporadycznie przenosząc go i weryfikując na innych platformach co kilka dni.

Od czasu do czasu przeprowadzałem szybkie testy weryfikacyjne, używając innych kompilatorów, takich jak Metrowerks CodeWarrior i Visual C++ 7.x, co czasami uwidaczniało problemy, a nawet błędy.

Kod nigdy za bardzo nie uległ rozwidleniu lub zanikowi. Jednakże obsługa Power PC została wprowadzona stosunkowo późno podczas pisania SAL i pociągnęła za sobą dużo żmudnej pracy, ponieważ wiele założeń przyjętych w SAL było nieprawidłowych w przypadku tej platformy. W szczególności problem spowodowało poleganie w programie testowym na istnieniu funkcji `main()`. Ponieważ w Pocket PC nie ma aplikacji konsolowych, dlatego też należy dostarczyć tam metodę `WinMain()`. Inne problemy były spowodowane naciskiem, jaki w Power PC położono na łańcuchy znaków rozszerzonych (w celu umożliwienia internacjonalizacji).

Użyłem kilku różnych niskopoziomowych interfejsów API, aby zaimplementować kluczowe cechy takie jak synchronizacja wątków. Znalazłem wspólne abstrakcje i umieściłem je w osobnej warstwie abstrakcji, która stała się w końcu warstwą samą dla siebie we właściwej implementacji dla każdej z platform. To oznaczało, że przeniesienie biblioteki ze schematu muteksów Win32 do wątkowego schematu muteksów POSIX było bardzo łatwe do osiągnięcia, ponieważ kod jądra biblioteki SAL nie zawierał wstawek specyficznych dla Win32.

Nawet jeśli pracujemy w podobnych systemach, na przykład w Windows na komputerach PC, dobrym pomysłem będzie użycie różnorodnego sprzętu (kart wideo, procesorów, kart muzycznych, kart sieciowych itd.) oraz oprogramowania, dzięki czemu więcej problemów związanych z konfiguracją może zostać wykrytych wcześniej niż później. Pomaga to odrzucić stwierdzenia programistów „U mnie działa!", kiedy błędy pojawiają się w już trakcie używania programu.

Używaj różnych kompilatorów

Powinieneś także używać różnych kompilatorów tak często, jak to tylko możliwe. W rozmaitych systemach docelowych może być możliwość stosowania tylko wybranego kompilatora, ale czasami można od tego uciec, używając tego samego kompilatora w całkowicie odmiennych systemach. Na przykład kompilator GCC dostępny jest na wielu różnych platformach.

Prawidłowa kompilacja na wielu platformach umożliwi uniknięcie trudnej sytuacji, jeśli producent preferowanego przez Ciebie kompilatora nagle zniknie. Zapewni to także, że kod nie będzie bazował na nowych (i niesprawdzonych) cechach języka lub kompilatora.

Testuj na kilku platformach

Większość projektów ma ściśle zdefiniowany zestaw docelowych platform określonych przez zmiany na rynku. Znacznie ułatwia to testowanie i nadzór jakościowy, ale także jest początkiem ryzykownego przyjmowania precyzyjnych założeń. Nawet jeśli wiesz, że program będzie uruchamiany na jednym systemie docelowym, nie zaszkodzi użyć innych platform docelowych — procesorów, pamięci RAM, pamięci masowych, systemów operacyjnych itd. — tylko do testowania.

W takim przypadku, jeśli system docelowy zostanie zmieniony z powodów wymagań rynkowych lub zmieniających się zależności biznesowych, będziesz mógł spać spokojnie, wiedząc, że Twoje oprogramowanie nie jest na sztywno przywiązane do pojedynczej platformy.

Obsługuj wiele bibliotek

Tworzenie większości dzisiejszego oprogramowania jest w mniejszym stopniu pisaniem nowego kodu niż łączeniem ze sobą większych fragmentów istniejącego kodu. Jeśli jesteś zależny od grupy bibliotek lub interfejsów API dostępnych w jednym kompilatorze lub systemie, przeniesienie kodu na nową platformę będzie trudne. Jednakże, jeśli od początku poświęcisz czas na obsługę wielu alternatywnych bibliotek, które wykonują to samo zadanie, będziesz miał o wiele większy wybór w sytuacji, kiedy producent zakończy działalność albo odmówi udostępnienia swojego oprogramowania na innych platformach. Jest jeszcze jedna, mniej znacząca zaleta — można licencjonować lub otworzyć swój kod bez przejmowania się zależnościami od zamkniętego kodu bibliotek innych producentów.

Klasycznym tego przykładem jest wybór pomiędzy obsługą OpenGL a Direct3D, dwoma najważniejszymi interfejsami API grafiki dostępnymi dzisiaj. OpenGL jest międzyplatformowy i dostępny w licznych systemach, włączając w to wszystkie najważniejsze systemy operacyjne komputerów PC. Z drugiej strony Direct3D jest oficjalnym interfejsem API grafiki w Windows, dostępnym tylko w Windows. To stawia twórców przed trudnym wyborem: optymalizować dla Windows, największego rynku użytkowników na świecie, czy próbować obsługiwać wiele platform za jednym razem, używając OpenGL?

Najlepszym rozwiązaniem byłoby stworzenie abstrakcyjnej warstwy, która mogłaby wykorzystywać obydwa interfejsy API. Może to oznaczać dużo pracy, więc rozgałęzienie abstrakcji musi być dobrze przemyślane, zanim zaczniesz je realizować. Jednakże w momencie przenoszenia oprogramowania na nową platformę praca poświęcona na tworzenie abstrakcji zwróci się wielokrotnie.

Planuj przenośność dla nowego projektu

Odczuwam niepohamowaną, czystą radość, kiedy rozpoczynam pracę nad nowym projektem. Zakładanie nowego katalogu, czekającego na wypełnienie doskonałym kodem źródłowym stworzonym na podstawie wieloletniego doświadczenia, jest w pewnym sensie podobne do poczucia zapachu nowego samochodu.

Kiedy znajdziesz się już w tej wyjątkowej sytuacji zaczynania czegoś od nowa, masz możliwość zaplanowania w jaki sposób działań, żeby Twój projekt był przenośny. Jeśli przed rozpoczęciem weźmiesz pod uwagę kilka zasad, zaoszczędzisz sobie później mnóstwo czasu i kłopotów.

Implementuj przenośność w prosty sposób

Tak jak z wieloma innymi rodzajami dobrych nawyków, szanse nabycia dobrych nawyków dotyczących przenośności są wprost proporcjonalne do tego, jak łatwo jest ich używać. Jeśli stosowana metodologia powoduje, że tworzenie przenośnego oprogramowania jest żmudne i nieefektywne, porzucisz je szybciej niż zacząłeś.

To ważne, żeby tworzyć procedury, biblioteki i mechanizmy tak, żeby pisanie przenośnego kodu było drugą naturą, a nie żmudnym, niekończącym się zadaniem. Na przykład programista nie powinien zajmować się kwestią uporządkowania bajtów, chyba że jest to konieczne.

Wybierz rozsądny poziom przenośności

O ile za każdym razem można próbować napisać stuprocentowo przenośny kod, o tyle w praktyce jest to niemalże niemożliwe bez znaczącego zmniejszenia funkcji użytkowych oprogramowania.

Nie możesz za wszelką cenę wymuszać przenośności! Twoje oprogramowanie powinno być na tyle przenośne, na ile jest to uzasadnione w praktyce, ale nigdy bardziej. Poświęcanie czasu i wysiłku na przenośność, która ma minimalną

wartość użytkową, jest analogiczne do stracenia tygodnia na optymalizację funkcji, która jest wywoływana jednokrotnie. Nie jest to wydajny sposób wykorzystania czasu.

Z tego powodu założenie zasadniczej i realistycznej podstawy, zestawu reguł określających rozsądny poziom przenośności, jest tak ważne dla projektu. Bez tego projekt będzie skazany na nijakie, przeciętne kodowanie po to tylko, żeby działał wszędzie… przeciętnie.

Każda platforma ma własny zestaw udziwnień, włączając w to komputery, kompilatory, narzędzia, procesory, sprzęt, systemy operacyjne itd. Są tysiące różnych sposobów na to, żeby uszkodzić program podczas przenoszenia z jednej platformy na drugą. Na szczęście wiele z tych udziwnień jest wspólnych, co ułatwia zadanie pisania przenośnego oprogramowania. Określenie wspólnych cech jest jednym z pierwszych kroków projektowania i pisania przenośnego kodu.

Jak omówię to później w rozdziale 14., duża część przenośności odnosi się do skalowalności (możliwości uruchomienia w systemach z dużym zróżnicowaniem wydajności i cech) w ramach podstawowych założeń. Skalowalność jest ważna, ale musi mieścić się w ramach dobrze określonych parametrów, żeby była postrzegana jako istotna.

Pomijając same funkcjonalności wybranych przez Ciebie platform, musisz zrobić również założenia co do ich podstawowej wydajności. Jest zupełnie możliwe napisanie oprogramowania, które skompiluje się i będzie działać identycznie zarówno na 8 MHz mikrokontrolerze Atmel AVR i na 3,2 GHz procesorze Intel Pentium 4, ale to, czy wynik będzie istotny i interesujący w obu przypadkach, jest wątpliwe. Algorytmy i struktury danych użyte dla stacji roboczej klasy PC są całkowicie odmienne od tych dla wbudowanych mikrokontrolerów i ograniczanie wysokowydajnych maszyn do operacji, które działają równie wydajnie w zasadniczo różniących się architekturach, nie ma większego sensu.

STUDIUM PRZYPADKU — OBLICZENIA ZMIENNOPRZECINKOWE

Język ANSI C obsługuje operacje zmiennoprzecinkowe o pojedynczej i podwójnej precyzji poprzez użycie słów kluczowych `float` i `double`, odpowiednio z przypisanymi do nich operandami matematycznymi. Większość programistów używa tej funkcjonalności bez zastanawiania się nad jej szczegółami. Niestety, niektóre dzisiejsze i wiele starszych urządzeń zapewniało niezwykle słabą obsługę obliczeń zmiennoprzecinkowych. Na przykład procesory używane w większości osobistych urządzeń przenośnych (PDA) nie są w stanie same z siebie wykonywać instrukcji zmiennoprzecinkowych, muszą więc używać dużo wolniejszych bibliotek, które je emulują.

Oczywiście może się zdarzyć, że bardzo wolne obliczenia zmiennoprzecinkowe są akceptowalne w przypadku poszczególnych projektów, ponieważ rzadko się ich używa (nawet jeśli z tego powodu rozmiar pliku wykonywalnego może się zwiększyć, ponieważ biblioteka emulująca operacje zmiennoprzecinkowe musi być skonsolidowana, mimo że wykorzystuje się ją tylko kilka razy). Ale w przypadku projektów, które wymagają wysokiej wydajności zmiennoprzecinkowej, kiedy szybko trzeba przenieść kod do systemu, który nie posiada wbudowanej obsługi operacji zmiennoprzecinkowych, sprawy mogą przybrać zły obrót.

Jedną z powszechnych metod rozwiązywania takiego problemu jest zapisywanie operacji matematycznych przy użyciu specjalnych makropoleceń, które na urządzeniach pozbawionych macierzystej obsługi operacji zmiennoprzecinkowych wywołują operacje stałoprzecinkowe zamiast zmiennoprzecinkowych. Oto przykład:

```
#if defined NO_FLOAT
typedef int32_t real_t
extern real_t FixedMul( real_t a, real_t b );
extern real_t FixedAdd( real_ a, real_t b );
#define R_MUL( a, b ) FixedMul( (a),(b))
#define R_ADD( a, b ) FixedAdd( (a),(b))
#else
typedef float real_t;
#define R_MUL( a, b ) ((a)*(b))
#define R_ADD( a, b ) ((a)+(b))
#endif /* NO_FLOAT */
```

Trzyelementowy iloczyn skalarny będzie napisany następująco:

```
real_t R_Dot3( const real_t a[ 3 ], const real_t b[ 3 ] )
{
   real_t x = R_MUL( a[ 0 ], b[ 0 ] );
   real_t y = R_MUL( a[ 1 ], b[ 1 ] );
   real_t z = R_MUL( a[ 2 ], b[ 2 ] );
   return R_ADD( R_ADD( x, y ), z);
}
```

Jednakże zwykły zapis zmiennoprzecinkowy jest znacznie łatwiejszy do odczytania i zrozumienia:

```
Float R_Dot3( const float a[ 3 ], const float b[ 3 ] )
{
   return a[ 0 ] * b[ 0 ] + a[ 1 ] * b[ 1 ] + a[ 2 ] * b[ 2 ];
}
```

Jeśli musisz obsługiwać systemy nieposiadające operacji zmiennoprzecinkowych albo jeśli myślisz, że istnieje duże prawdopodobieństwo, że będzie to potrzebne, wtedy używanie makropoleceń będzie prawdopodobnie dobrym pomysłem. Jednak jeśli masz możliwość określenia macierzystej obsługi operacji zmiennoprzecinkowych jako części swoich założeń, będziesz czerpał korzyści ze zwiększenia czytelności i zwięzłości kodu.

Przenośność jest dobrym pomysłem, a pisanie przenośnego kodu jest dobrym podejściem, jeśli jednak doprowadzisz to do skrajności albo będziesz pisać nadmiernie przenośny kod tylko po to, żeby zaspokoić ideologiczny dogmat, Twój kod może na tym ucierpieć. Przenośność jest środkiem do osiągnięcia celu, nie celem samym w sobie.

Aplikacja sieciowa, która opiera swoją architekturę na szerokopasmowej komunikacji o małych opóźnieniach, przestanie działać w przypadku pracy z modemem. Więc o ile aplikacja może być skompilowana i uruchomiona wszędzie, w praktyce nie jest przenośna do pewnych rodzajów sieci z powodu podstawowych założeń dotyczących sieci, w których będzie działać.

Ustanowienie punktu wyjścia jest kluczowym elementem przy tworzeniu przenośnego oprogramowania, ponieważ umożliwia poczynienie pewnych założeń, które są całkowicie uzasadnione i umożliwiają praktyczne tworzenie oprogramowania efektywnego na ograniczonej liczbie platform.

Istnieje różnica pomiędzy kodem przenośnym aż do przesady a przenośnym wystarczająco. Jeśli Twój projekt jest przeznaczony na pojedynczą platformę docelową, ale wiesz, że w pewnym momencie może zajść potrzeba zmiany kompilatora, skoncentruj się na utrzymaniu przenośności swojego kodu pomiędzy kompilatorami i nie przejmuj się tak bardzo systemami docelowymi, których najprawdopodobniej nie będziesz obsługiwać.

Nie przywiązuj swoich projektów do produktów innych firm

Nowoczesne tworzenie oprogramowania jest niewiarygodnie złożone i nawet proste projekty mogą składać się z dziesiątków tysięcy linii kodu źródłowego. Ta wysoka złożoność często wymaga użycia (w najlepszym przypadku) dobrze przetestowanych i dobrze udokumentowanych komponentów firm zewnętrznych, takich jak biblioteki i narzędzia. Używanie istniejących komponentów oszczędza czas, ale powoduje także pojawienie się mnóstwa nowych problemów z przenośnością. Zapewnienie przenośności Twojego oprogramowania jest wystarczająco trudne i czasochłonne, ale jeśli dodasz do tego obce wpływy, może stać się to zadaniem zdecydowanie beznadziejnym. Za każdym razem, kiedy zewnętrzny komponent jest włączany do projektu, elastyczność i kontrola coraz bardziej się zmniejszają.

Nawet w przypadku najlepszych scenariuszy, bibliotek o otwartym kodzie, musisz sprawdzić, czy ten kod da się skompilować i uruchomić wszędzie tam, gdzie potrzebujesz. A jeśli chcesz, żeby biblioteka o otwartym kodzie obsługiwała kolejną platformę, musisz całą pracę związaną z jej przeniesieniem wykonać sam (co jest na szczęście możliwe dzięki naturze otwartego kodu).

Niestety, użycie bibliotek o zamkniętym kodzie zewnętrznych firm uniemożliwia skorzystanie z tego rozwiązania. W takiej sytuacji możesz stanąć przed poważnym problemem, jeśli dostawca nie chce, bądź nie może (na przykład zaprzestał działalności) obsługiwać platformy, której wymagasz. W najgorszym przypadku będziesz musiał ponownie zaimplementować od podstaw bibliotekę zewnętrznej firmy na nową platformę. Przywiązywanie Twoich projektów do komponentów innych firm może być bardzo niebezpieczne na dłuższy dystans. Wielu informatycznych weteranów może wyrecytować opowieści o projektach nierozerwalnie związanych z porzuconą biblioteką albo zestawem narzędzi i o tym, jaki wpływ to miało na cały proces rozwijania oprogramowania.

Na przykład wielu twórców używa na platformie PC biblioteki sieciowej DirectPlay Microsoftu, ponieważ jest ona darmowa, a jej autorzy twierdzą, że zapewnia wiele funkcjonalności, których ponowne napisanie zajęłoby mnóstwo czasu. Nisko wiszący owoc darmowej i prostej technologii jest kuszący, ale ci, którzy po niego sięgną, mogą wpaść w tarapaty, kiedy będą próbować przenieść oprogramowanie na platformę inną niż Microsoft Windows, taką jak Macintosh albo konsola do gier. Często muszą potem przepisać całą warstwę sieciową od zera, żeby zrównoważyć nierozważne użycie technologii zewnętrznej firmy.

PODSTAWOWE ZAŁOŻENIA I UŻYCIE INTERFEJSÓW API NA PRZYKŁADZIE BIBLIOTEKI SAL

Biblioteka SAL ma dosyć skromną funkcjonalność i podstawowe złożenia co do wydajności. Jest napisana w ANSI C89 (z wyjątkiem jednego pliku napisanego w Objective-C i ograniczonego do użycia na platformie Mac OS X), co umożliwia jej dostępność na największym możliwym zakresie platform. Dwa kluczowe komponenty technologiczne to mikser i obsługa wątków w określony sposób.

Mikser jest oparty na liczbach całkowitych z założeniem, że 32-bitowe operacje na liczbach całkowitych, w szczególności mnożenie, będą stosunkowo szybkie. Z tego powodu może nie pracować szczególnie dobrze na 16-to bitowych platformach takich jak Palm OS 4.

Jednakże dwa kluczowe parametry — maksymalna liczba równocześnie odtwarzanych dźwięków i rozmiar bufora — są określane przez użytkownika w czasie działania, umożliwiając bibliotece SAL wysoką skalowalność w szerokim zakresie, w zależności od możliwości systemów. Dla szczególnie wolnych systemów rozmiar bufora może być zwiększony kosztem większego opóźnienia. Ponadto ilość aktywnych dźwięków może zostać zmniejszona, żeby zminimalizować ilość pracy wykonywanej w wewnętrznej pętli miksującej. Wysokowydajny system może bez problemu poradzić sobie ze 128-głosową polifonią, ale w razie potrzeby możesz używać SAL w trybie jednogłosowym na mało wydajnych urządzeniach.

Podstawowa implementacja modelu obsługi wątków dla SAL tworzy osobny wątek odpowiedzialny za pobieranie danych próbek dla aktywnych głosów. Taka jest koncepcja modelu SAL używana w przypadku większości platform. Jednakże przynajmniej jedna platforma (OS X) używa zamiast tego wywołań zwrotnych do dźwiękowego interfejsu API CoreAudio. (Wywołania zwrotne są wykonywane z innego wątku, więc z technicznego punktu widzenia używany jest inny wątek, ale SAL go nie tworzy.) Bez względu na to, jak system dźwiękowy tworzy zmiksowane dane, podstawą ciągle jest założenie, że dzieje się to asynchronicznie, więc oczekuje się, że będą dostępne podstawowe operacje do synchronizacji (w postaci muteksów). Proste jednowątkowe systemy operacyjne (takie jak Mac OS 9 albo Microsoft MS-DOS) w teorii mogą być obsługiwane, ale wymaga to szczegółowego zaprojektowania, ponieważ architektury te używają przerwań do sterowania systemem dźwiękowym.

W niektórych częściach SAL wymagane jest, żeby niewielka (jak na standardy PC) ilość pamięci była cały czas dostępna — około kilkaset kilobajtów. Implementacja odtwarzania próbek przy wykorzystaniu modulacji kodowo-impulsowej (PCM) zakłada, że w pamięci obecne są dane PCM. Jednakże bez problemu można użyć zamiast tego strumieniowania dźwięku, dzięki czemu zapotrzebowanie na pamięć znacznie się zmniejszy. Wymaga to większej pracy od programisty aplikacji, ale istnieje taka możliwość.

Jądro implementacji SAL wymaga standardowej biblioteki uruchomieniowej C (`free()`, `malloc()`, `vsnprintf ()`, `memset()`, `fprintf()` itd.). Jednakże z minimalnymi modyfikacjami (składającymi się głównie z podmiany funkcji `vsnprintf()` i `memset()`) może działać efektywnie w środowiskach wolno stojących (bez biblioteki uruchomieniowej C czy systemu operacyjnego).

Biblioteka SAL nie używa żadnych konkretnych interfejsów API w kodzie jądra. Jednakże używa kilku specyficznych dla platform interfejsów API (Win32, pthreads, CoreAudio, Cocoa, OSS, ALSA itd.) w celu zaimplementowania części architektury. Wewnętrzne interfejsy API SAL znajdują się w warstwie ponad tymi interfejsami API. Na przykład `_SAL_lock_mutex()` wywołuje `WaitForSingleObject()` na Win32 i `pthread_mutex_lock()` na Linuksie.

Nie ma żadnych elementów jądra SAL, które nie mogłyby być przeniesione na platformę uwzględniającą podstawowe założenia SAL. Jednakże biblioteki są często dużo łatwiejsze do przeniesienia niż aplikacje.

Jeśli uważasz, że komponent zewnętrznej firmy jest znaczącą częścią Twojej pracy, powinieneś oddzielić ją przynajmniej jednym poziomem abstrakcji, tak żeby jego podmiana albo rozszerzenie miały jak najmniejszy wpływ na pozostałą

część projektu. Jeśli się okaże, że musisz użyć konkretnego zestawu narzędzi albo biblioteki, spróbuj otrzymać licencję na pełny kod źródłowy lub, w ostateczności, zapewnij, żeby producent umieścił kod źródłowy w depozycie, co zabezpieczy Cię przed wycofaniem się firmy z rynku.

Przenoś stary kod

Niestety, rzeczywistość rzadko pozwala nam na luksus pracowania nad zupełnie nowym projektem. W wielu przypadkach musimy radzić sobie z problemami z przenośnością niewynikającymi z naszej winy, w momencie przenoszenia czyjegoś nieprzenośnego kodu z jednej platformy na inną.

W takiej sytuacji kilka ogólnych wskazówek i zasad może pomóc Ci zarządzać tym procesem.

Załóż, że kod jest nieprzenośny, dopóki nie zostanie przeniesiony

Wielu programistów myśli, że są nieźli w pisaniu przenośnego kodu, i wielu faktycznie jest. Ale problemem jest to, że w ich mniemaniu kod jest przenośny i powinien po prostu „skompilować się i ruszyć" na nowej platformie. To niestety bardzo rzadki przypadek. Bez znaczenia jak bardzo Cię zapewniano, że kod jest przenośny, zawsze zrób założenie, że nie jest. Zanim kod nie zostanie przeniesiony na nową platformę i przetestowany, powinien być traktowany z najwyższą ostrożnością.

Podczas tworzenia biblioteki SAL ciągle natrafiałem na problemy z przenośnością, za każdym razem, kiedy rekompilowałem na innym systemie albo zmieniałem kompilator. Po prostu jest zbyt wiele rzeczy, o których trzeba pamiętać, a i tak wszystko jak zwykle wychodzi w praktyce. Przenoszenie jest papierkiem lakmusowym tego, jak przenośny jest Twój kod.

Oczywiście możesz twierdzić, że Twój kod jest „przyjazny przenośności", co jest rozsądnym określeniem dla oprogramowania pisanego z założeniem przenośności, ale nie przeniesionego na wybraną platformę. Jest olbrzymia różnica pomiędzy kodem, o którym wiemy, że jest nieprzenośny (próbowałeś go przenieść, ale nie mogłeś), kodem, o którym nie wiemy, że jest przenośny (nikt nie próbował go przenieść), i kodem, o którym wiemy, że jest przenośny (był przeniesiony na inną platformę).

Zmieniaj tylko konieczne rzeczy

Przenoszenie oprogramowania wymaga wielu zmian i z tego powodu pojawia się możliwość wprowadzenia nowych błędów i uszkodzenia oprogramowania na oryginalnej platformie. Chociaż chciałoby się przejrzeć i oczyścić kod niezwiązany z przenoszeniem, unikaj tego za wszelką cenę. Utrzymywanie czystej podstawy kodu zapewnia dobrą podstawę do testów regresywnych.

Praca z dużą ilością kodu źródłowego, który już działa na jednej platformie, może być trudna. Za każdym razem, kiedy edytujesz plik, istnieje pewna szansa, że właśnie zepsułeś coś w innym miejscu. Z tego powodu rozszerz zasadę „nie zmieniaj niczego, czego nie musisz" i włącz do niej zalecenie: „idź po najmniejszej linii oporu".

Oznacza to, że najczęściej istnieje jakaś logiczna linia podziału, która pozwoli Ci wyraźnie oddzielić Twój nowy kod od starego kodu. Jeśli potrafisz ją znaleźć, przenoszenie będzie o wiele łatwiejsze, ponieważ będziesz mógł się przełączać pomiędzy zmienionym przez Ciebie i pierwotnym kodem źródłowym.

Zaplanuj atak

Zanim napiszesz lub zmienisz jedną linię, musisz wyraźnie zrozumieć, co zamierzasz zrobić. Przenoszenie oprogramowania różni się od pisania nowego oprogramowania i Twoje podejście także powinno być odmienne. Zidentyfikuj prawdopodobne punkty zapalne podczas przenoszenia, dzięki czemu będziesz dokładnie wiedział, jakie zadania należy wykonać, aby przenieść oprogramowanie na nową platformę. Jak tylko zrobisz listę, możesz spokojnie usiąść i opracować dokładny plan ataku, który wykorzystasz podczas procesu przenoszenia.

Na przykład przenoszenie aplikacji z Windows na Linuksa mogłoby mieć taką (bardzo ogólną) listę kontrolną:

- ☑ Usuń wszystkie odwołania do plików nagłówkowych specyficznych dla Windows.
- ☑ Zaktualizuj kod obsługi plików przeznaczony dla Windows do funkcji używanych w Linuksie.
- ☑ Wyizoluj i zaktualizuj całą ścieżkę CreateWindow.
- ☑ Zaktualizuj ładowanie zasobów do funkcji dostępu do plików Linuksa.
- ☑ Zmień kod wykorzystujący rejestr na obsługujący lokalne ustawienia oparte na plikach.

Mając dobrze zdefiniowaną listę zadań, możesz spróbować przewidzieć, na jakie problemy natrafisz później w procesie przenoszenia i odpowiednio go zaplanować.

Bardzo często pierwszym odruchem jest po prostu przeniesienie oprogramowania na nowy system i kompilowanie z jednoczesnym poprawianiem błędów zgłaszanych przez kompilator i program konsolidujący. Jak tylko pogram zostanie skompilowany i skonsolidowany, próbujesz go uruchomić w debugerze, aż zacznie działać. Nie jest to jednak bardzo wydajna metoda procesu przenoszenia. Powinieneś zidentyfikować wszystkie najważniejsze miejsca, które należy zmodyfikować w pierwszej kolejności, aby uniknąć zbytniego zagłębiania się w kod, co wymagałoby pracy, która powinna być wykonana później.

Na przykład może się okazać, że pierwsze poprawki, które przyszły Ci do głowy przy rozwiązywaniu jednego z problemów z przenoszeniem, są mało praktyczne, kiedy pojawi się inny problem. W takim przypadku musisz wycofać pierwsze zmiany i opracować strategię, która rozwiąże oba problemy z przenośnością.

Dokumentuj wszystko w systemie śledzenia zmian

Jeśli dotąd nie wyraziłem się wystarczająco jasno, każda wprowadzona przez Ciebie zmiana może okazać się potencjalnie niszczycielska. Z tego powodu powinieneś dokumentować wszystkie zmiany.

Używanie systemu śledzenia zmian jest niemalże obowiązkowe, kiedy tworzysz skomplikowane oprogramowanie, które będzie rozwijać się w czasie. Podczas przenoszenia jest to jeszcze ważniejsze, ponieważ każda zmiana może niezauważalnie uszkodzić coś niezwiązanego z Twoją bieżącą pracą, a szukanie tej usterki jest znacznie łatwiejsze, jeśli masz przejrzyste zapisy zmian.

Po rozpoczęciu przenoszenia programiści czują niepohamowaną chęć, żeby zobaczyć program działający na nowej platformie tak szybko, jak to tylko możliwe. Ale jeśli rozpoczniesz zadanie bez odpowiedniego zaplanowania, możesz stracić mnóstwo czasu, przeszukując ślepe zaułki i wycofując poprzednie zmiany.