AISDI - Laboratorium wstępne

Cel laboratorium

Zapoznanie się z narzędziami przydatnymi przy realizacji kolejnych ćwiczeń.

Zaprezentowane narzędzia znajdują zastosowanie nie tylko w ramach tych ćwiczeń, ale stanowią podstawę warsztatu dobrego programisty.

Istnieje wiele systemów ułatwiających współpracę z przedstawionymi narzędziami, ale celem laboratorium jest przekazanie pewnej ogólnej idei badania działania programu, a nie nauka obsługi konkretnych narzędzi (zwłaszcza, że różne języki/środowiska zawierają inne narzędzia, jednakże zawsze realizujące zadania podobne do poniższych).

Przebieg ćwiczenia

1. Przygotowanie środowiska

   Każdy normalny projekt informatyczny składa się z więcej niż jednego pliku. Proszę przygotować swoje ulubione środowisko (CodeBlocks, vim + make, itp.), tak aby był w nim projekt zawierający co najmniej 3 pliki: `main.cpp`, `MyClass.cpp`, `MyClass.h` (zamiast `MyClass` proszę wymyślić jakąkolwiek prostą klasę (`Car`, `Box`, itp.), funkcja `main` powinna być zaimplementowana w pliku `main.cpp` i w jakiś sposób używać wprowadzonej klasy.

   Należy zadbać, aby projekt budował się z wykorzystaniem C++11 i zawierał co najmniej dwie konfiguracje:

   - Debug - z symbolami debugowymi i bez optymalizacji (`-g -O0` w gcc)
   - Release - bez symboli i z optymalizacją (`-O2` w gcc)

2. Testy jednostkowe

   Do projektu należy dodać wsparcie dla testów jednostkowych. W ramach tego laboratorium najwygodniejsze może być wykorzystanie dostępnej biblioteki z pakietu Boost. Przykład działającej konfiguracji z laboratoriów PROI jest dostępny tutaj.

   W przykładzie wykorzystane są Makefile'e, ale można wykorzystać widoczne w nich opcje do poprawnego skonfigurowania np. IDE CodeBlocks.

   Ważne: zazwyczaj testy jednostkowe buduje się niezależnie od głównej aplikacji (projekt buduje wtedy co najmniej dwa pliki wykonywalne). Dla uproszczenia można tak zmodyfikować `main.cpp`, żeby budował tylko testy jednostkowe (testowanie algorytmów i struktur nie wymaga istnienia programu, testy jednostkowe zazwyczaj wystarczają).

W celu weryfikacji poprawnej konfiguracji należy napisać w nowym pliku `MyClassTests.cpp` dowolny przechodzący test jednostkowy.

3. Błąd krytyczny w aplikacji

   W istniejącym projekcie należy doprowadzić do wystąpienia błędu krytycznego aplikacji, np. poprzez próbę dostępu do zastrzeżonych fragmentów pamięci. Na przykład - dostęp do obiektu zwolnionego wcześniej operacją `delete`, próba wywołania metody wirtualnej na wskaźniku o wartości `nullptr`, sięgnięcie do obszaru poza granicami zaalokowanej tablicy itp. itd. Efektem próby uruchomienia programu powinno być uzyskanie komunikatu o jej błędzie od systemu operacyjnego (np. `Segmentation fault`).

   Oczywiście zazwyczaj programista nie jest świadomy, gdzie jest umiejscowiony błąd w kodzie, powodujący takie efekty działania aplikacji. Często wygodnymi narzędziami do znalezienia źródła takich problemów są różnego typu tzw. debuggery. Najprostszym z nich jest `gdb`. Aby zdiagnozować w nim taki problem wystarczy uruchomić go poleceniem `gdb ./nazwa_programu_do_badania`, następnie w jego konsoli wykonać polecenie `run`, uruchamiające badany program. Gdy nastąpi jego błąd, `gdb` zatrzyma się. Można wtedy uzyskać ścieżkę wywołań, która doprowadziła do tego błędu, wpisując polecenie `bt`. Korzystając z polecenia np. `print` można popodglądać wartości lokalnych zmiennych.

   Ważne: proszę porównać efekty pracy z narzędziem `gdb` w zależności od wykorzystanej konfiguracji kompilacji (Debug vs. Release). Powinno to wyjaśnić, czemu stosuje się często takie dwie kompilacje: Debug dla programistów, Release dla klienta.

   Dla zainteresowanych: proszę spróbować włączyć symbole debugowe w kompilacji z optymalizacją i spróbować obejrzeć wtedy, co się dzieje przy próbie użycia `gdb` (wysokie prawdopodobieństwo pozornie losowych przejść pomiędzy liniami programu).

   Takie narzędzia, jak `gdb` są pożyteczne, gdy dostęp do programu jest tylko przez konsolę (np. zdalne podłączenie po SSH do serwera klienta). Przy normalnej pracy wygodniejsze jest skorzystanie z narzędzi graficznych. Prostym opakowaniem na `gdb` jest program `ddd`. IDE CodeBlocks także wspiera uruchomienie programu w trybie debug i obserwację jego zachowania.

   W wybranym przez siebie środowisku należy przećwiczyć:

   - obserwację efektów błędu programu,
   - podglądanie wartości zmiennych,
   - umieszczanie punktów wstrzymania aplikacji (brakpoint).

   Proszę pamiętać, że na kolejnych laboratoriach programy, które będą się tak zachowywać (zamykać z błędem), nie będą oceniane. Dlatego warto zapoznać się z narzędziami pozwalającymi usunąć takie błędy.

4. Problemy z zarządzaniem zasobami aplikacji

   Programy zaprezentowane w poprzednim punkcie wykorzystują możliwości nadzoru nad uruchomioną aplikacją, udostępnione przez system operacyjny. Pozwala to na badanie aplikacji w jej "naturalnym środowisku", ale nie pozwala wykryć pewnych rodzajów błędów. W tym celu między innymi istnieją systemy wykorzystujące tzw. instrumentację czy symulację, czyli uruchamiające badaną aplikację w środowisku pozwalającym na lepszą obserwację zachowania programu. Na przykład, podmieniając biblioteczne funkcje `malloc` i `free`, można śledzić wycieki pamięci. Między innymi tym zajmuje się program `valgrind`.

   Na początku proszę użyć `valgrind --tool=memcheck ./badana_aplikacja` na aplikacji z poprzedniego punktu. Narzędzie powinno wyświetlić raport o podobnej zawartości do wywołania debuggera. Zauważalne powinno być też zwiększenie czasu wykonania programu - `valgrind` wprowadza dodatkową warstwę pomiędzy system operacyjny i aplikację, znacząco zmniejszając jej wydajność.

   Następnie należ naprawić aplikację z poprzedniego punktu, a po czym doprowadzić w niej do wycieku pamięci (`new` bez `delete`).

   Wywołując `valgrind --tool=memcheck --leak-check=full ./aplikacja` proszę zaobserwować wyniki zwracane przez narzędzie. Zazwyczaj są wystarczające by odnaleźć większość wycieków w aplikacji. Niestety zdarzają się też raporty o fałszywych wyciekach, ale każdy warto przeanalizować.

   Wszystkie kolejne projekty powinny być sprawdzone przy użyciu `valgrind`a na obecność wycieków. Wykrycie wycieku pamięci przez prowadzącego będzie skutkowało znaczącym obcięciem punktów, z niezaliczeniem ćwiczenia włącznie.

5. Profilowanie aplikacji

   Często ważny aspektem podczas wyboru odpowiedniego algorytmu, bądź struktury danych, jest wydajność danego rozwiązania. Podstawowa zasada optymalizacji brzmi: najpierw zmierzyć, potem optymalizować. Oczywiście nie oznacza to, że można bezkarnie sięgać po algorytm o złożoności `O(N^3)` gdy dostępny jest `O(NlogN)`, ale że gdy pojawia się problem z niewystarczającą wydajnością programu, najpierw warto zmierzyć, jak działa i co w nim działa najwolniej, nim zacznie się losowo optymalizować kod.

   Narzędzia wykorzystywane do takich pomiarów to tzw. profile'ry. Ich zastosowanie to złożony problem i nie mieści się w zakresie tego laboratorium. Dlatego wykorzystane będzię najprostsze narzędzie badające czas wykonania programu: instrukcja `time` dostępna w systemach Linux. Wywołanie `time jakakolwiek_instrukcja` zmierzy czas wykonania tej instrukcji i wyświetli kilka różnych czasów. "Real" to ten czas, jaki upłynął z punktu widzenia użytkownika, "system" to czas, jaki system operacyjny

spędził na obsłudze wywołań systemowych badanej aplikacji, a "user" to ile czasu spędziła sama aplikacja, wykorzystując procesor. Ten ostatni czas jest najciekawszy, gdy bada się złożoność czasową/obliczeniową, bo liczy tylko, ile zajęło samo przetwarzanie.

Proszę spróbować sprofilować jakąś swoją aplikację. Pierwsza obserwacja może być taka, że aplikacja zakańcza się bardzo szybko. Dlatego, by móc zmierzyć czas jakiejś operacji, trzeba ją często powtórzyć w pętli, bądź na dużym zbiorze danych. Profilowanie krótkich wywołań daje zazwyczaj wyniki niskiej jakości.

Proszę zaimplementować jakikolwiek prosty algorytm (np. odwrócenie kolejności elementów w wektorze) i zmierzyć czas działania tej operacji, jak skaluje się w zależności od ilości elementów w wektorze i jakie są różnice między konfiguracjami Debug i Release.

Każde kolejne ćwiczenie będzie wymagało przygotowania krótkiego raportu z pomiarów czasu wykonania zaimplementowanych projektów. Będą też ćwiczenia polegające na porównaniu różnych implementacji.

Dla zainteresowanych: instrukcję `time` można wykorzystać, by uzyskać dużo więcej informacji z systemu o działaniu aplikacji. Na przykład ilość danych wysłanych i odebranych z sieci. Proszę sprawdzić `man time`.

Dla bardzo zainteresowanych: oprócz złożoności czasowej wciąż bywa znacząca złożoność pamięciowa. Proste badanie "kto alokuje najwięcej pamięci" można wykonać z użyciem `valgrind --tool=massif`.