JAK DZIAŁAJĄ FUNKCJE – PODZIAŁ PAMIĘCI

Gdy wywołujesz daną funkcję, program przechodzi do tej funkcji, przekazywane są parametry i następuje wykonanie ciała funkcji. Gdy funkcja zakończy działanie, zwracana jest wartość (chyba, że zwracana jest wartość typu **void**) i sterowanie powraca do funkcji wywołującej.

Jak to się odbywa? Skąd kod wie, gdzie skoczyć? Gdzie są przechowywane przekazywane zmienne? Co się dzieje ze zmiennymi zadeklarowanymi w ciele funkcji? W jaki sposób jest przekazywana wartość zwracana przez funkcję? Skąd kod wie, w którym miejscu ma wznowić działanie po powrocie z funkcji?

Większość książek wprowadzających w zagadnienia programowania nie próbuje odpowiadać na te pytania, ale bez zrozumienia tych mechanizmów pisanie programów wciąż pozostaje „programowaniem z elementami magii.” Wyjaśnienie zasad działania funkcji wymaga poruszenia tematu pamięci komputera.

1.Poziomy abstrakcji

Jednym z największych wyzwań dla początkujących programistów jest konieczność posługiwania się wieloma poziomami abstrakcji. Oczywiście, komputery są jedynie urządzeniami elektronicznymi. Nie mają pojęcia o oknach czy menu, nie znają programów ani instrukcji, a nawet nie wiedzą nic o zerach i jedynkach. W rzeczywistości jedyne zmiany, jakie zauważają, to zmiany napięcia mierzonego w odpowiednich punktach układów elektronicznych. Nawet to jest dla nich pewną abstrakcją: w rzeczywistości elektryczność jest tylko wygodną intelektualną koncepcją dla zaprezentowania działania cząstek subatomowych, które z kolei są abstrakcją dla czegoś innego (!).

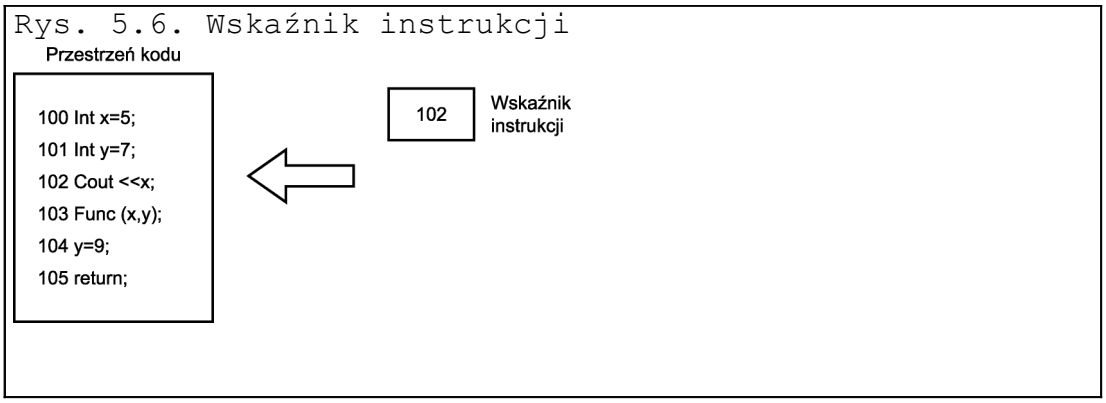
Bardzo niewielu programistów zadaje sobie trud zejścia poniżej poziomu wartości w pamięci RAM. W końcu nie trzeba znać fizyki cząsteczkowej, aby prowadzić samochód, robić kanapki czy kopać piłkę; nie trzeba też znać się na elektronice, aby programować komputer.

Konieczne jest jednak zrozumienie, w jaki sposób jest zorganizowana pamięć komputera. Bez wyraźnego obrazu tego, gdzie znajdują się tworzone zmienne i w jaki sposób przekazywane są wartości między funkcjami, programowanie nadal pozostanie tajemnicą.

2.Podział pamięci

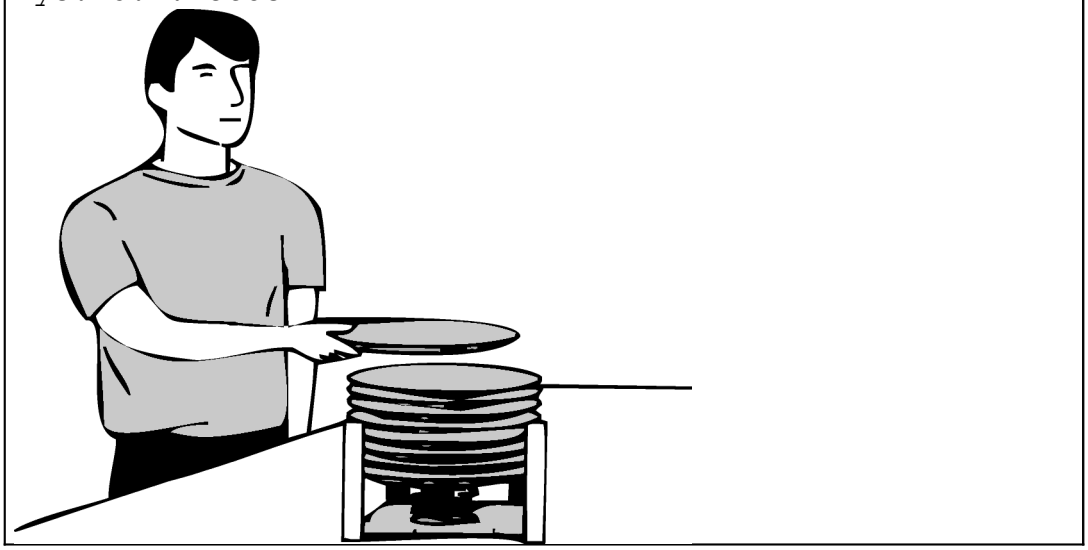
Gdy uruchamiasz program, system operacyjny (taki jak DOS, Unix czy Microsoft Windows) przygotowuje różne obszary pamięci (w zależności od wymagań kompilatora). Jako programista C++, często będziesz miał do czynienia z globalną przestrzenią nazw, stertą, rejestrami, przestrzenią kodu oraz stosem.
Zmienne globalne występują w globalnej przestrzeni nazw. O globalnej przestrzeni nazw i stercie pomówimy dokładniej w następnych rozdziałach, teraz skupimy się na rejestrach, przestrzeni kodu oraz stosie.
Rejestry są specjalnym obszarem pamięci wbudowanym w procesor (CPU, Central Processing Unit). Odpowiadają za wewnętrzne wykonywanie programu przez procesor. Większość tego, co dzieje się w rejestrach, wykracza poza tematykę tej książki; interesuje nas tylko zestaw rejestrów, który w danej chwili wskazuje następną instrukcję kodu. Zestaw rejestrów nosi wspólną nazwę *wskaźnika instrukcji* (ang. *instruction pointer*). Zadaniem wskaźnika instrukcji jest śledzenie, która linia kodu ma zostać wykonana jako następna. przetłumaczona na serię instrukcji procesora, z których każda znajduje się w pamięci pod określony adresem.

Rys. 5.6. Wskaźnik instrukcji



Stos jest specjalnym obszarem pamięci, zaalokowanym przez program w celu przechowywania danych potrzebnych wszystkim funkcjom programu. Jest nazywany stosem, gdyż stanowi kolejkę LIFO (last-in, first-out — ostatni wchodzi, pierwszy wychodzi), przypominającą stos talerzy w restauracji (pokazany na rysunku 5.7).

Rys. 5.7. Stos



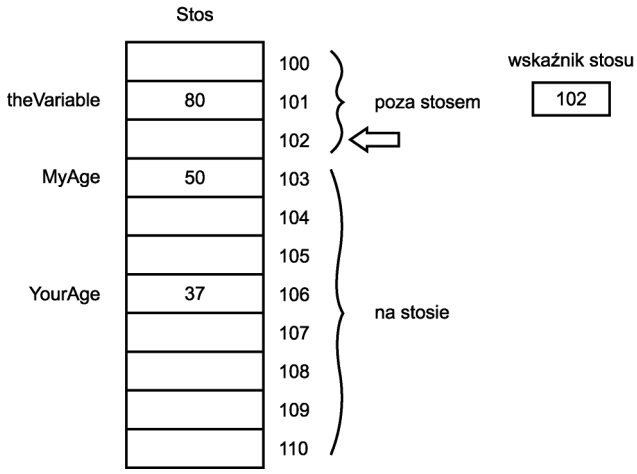
„Ostatni wchodzi, pierwszy wychodzi” – oznacza, że to, co zostanie umieszczone na stosie jako ostatnie, zostanie z niego zdjęte jako pierwsze. Większość kolejek przypomina kolejki w sklepie: pierwsza osoba w kolejce jest obsługiwana jako pierwsza. Stos przypomina stos monet: gdy ułożysz na stole dziesięć monet, jedna na drugiej, a następnie część z nich zabierasz, zabierasz najpierw te monety, które ułożyłeś jako ostatnie.

Gdy dane są umieszczane (ang. *push*) na stosie, stos rośnie; gdy są zdejmowane ze stosu (ang. *pop*), stos maleje. Nie ma możliwości wyjęcia talerza ze stosu bez zdjęcia wszystkich talerzy, które zostały umieszczone na nim później.

Stos talerzy jest najczęściej przedstawianą analogią. Jest ona poprawna, ale działanie pamięci wygląda nieco inaczej. Bardziej odpowiednie jest wyobrażenie sobie szeregu pojemników ułożonych jeden na drugim. Szczytem stosu jest ten pojemnik, na który w danej chwili wskazuje *wskaźnik stosu* (ang. *stack pointer*), będący jeszcze jednym rejestrem.

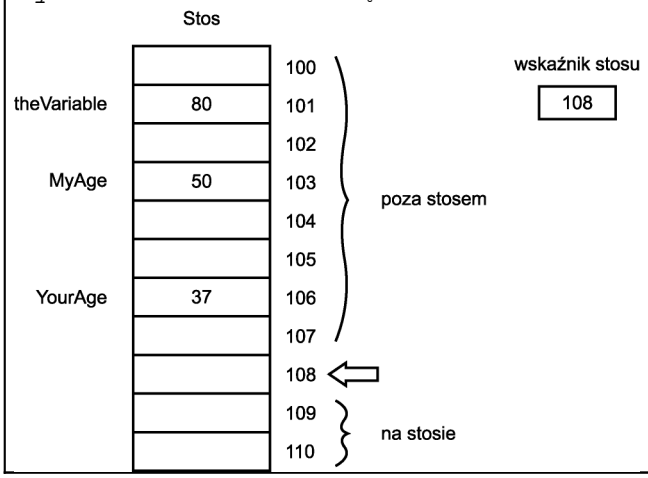
Każdy z pojemników ma kolejny adres, a jeden z tych adresów jest przechowywany w rejestrze wskaźnika stosu. Wszystko, co znajduje się poniżej tego magicznego adresu, znanego jako szczyt stosu, jest uważane za zawartość stosu. Wszystko, co znajduje się powyżej szczytu stosu, jest uważane za znajdujące się poza stosem, a co za tym idzie, niepoprawne. Ilustruje to rysunek 5.8.

Rys. 5.8. Wskaźnik stosu



Gdy odkładasz daną na stos, jest ona umieszczana w pojemniku znajdującym się powyżej wskaźnika stosu, a następnie wskaźnik stosu jest przesuwany o jeden pojemnik w górę. Gdy zdejmujesz daną ze stosu, jedyną czynnością odbywającą się w rzeczywistości jest przesunięcie wskaźnika stosu o jeden pojemnik w dół. Pokazuje to rysunek 5.9.

Rys. 5.9. Przesunięcie wskaźnika stosu



Dane powyżej wskaźnika stosu (czyli poza stosem) mogą (ale nie muszą) ulec zmianie w dowolnej chwili. Wartości te nazywamy „odpadami” (aby lepiej uświadomić sobie, że nie powinniśmy na nie liczyć)

.

3.Stos i funkcje

Poniżej przedstawiono przybliżony opis tego, co się dzieje, gdy program przechodzi do wykonania funkcji. (Poszczególne rozwiązania różnią się, w zależności od systemu operacyjnego i kompilatora).

1. Zwiększany jest adres we wskaźniku instrukcji i wskazuje on instrukcję następną po tej, która wywołuje funkcję. Ten adres jest następnie umieszczany na stosie; stanowi adres powrotu z funkcji.
2. Na stosie jest tworzone miejsce dla zadeklarowanego typu wartości zwracanej przez funkcję. Gdy zwracany typ jest zadeklarowany jako int, w przypadku systemu z dwubajtowymi liczbami całkowitymi, na stos są odkładane dwa kolejne bajty, ale nie jest w nich umieszczana żadna wartość („odpady", które się w nich dotąd znajdowały, pozostają tam nadal).
3. Do wskaźnika instrukcji jest ładowany adres wywoływanej funkcji (ten adres jest zawarty w kodzie aktualnie wykonywanej instrukcji wywołania funkcji), dzięki czemu następna wykonywana instrukcja będzie już instrukcją funkcji.
4. Odczytywany jest adres bieżącego szczytu stosu, następnie zostaje on umieszczony w specjalnym wskaźniku nazywanym *ramką stosu* (ang. *stack frame*). Wszystko, co zostanie umieszczone na stosie od tego momentu, jest uważane za „lokalne" dla funkcji.
5. Na stosie umieszczane są argumenty funkcji.
6. Wykonywana jest instrukcja wskazywana przez wskaźnik instrukcji (następuje wykonanie pierwszej instrukcji w funkcji).
7. W trakcie ich definiowania, lokalne zmienne zostają umieszczane na stosie.

Gdy funkcja jest gotowa do powrotu, zwracana wartość jest umieszczana w miejscu stosu zarezerwowanym w kroku 2. Następnie stos jest zwijany (tzn. wskaźnik stosu przesuwa się) aż do wskaźnika ramki stosu, co oznacza odrzucenie wszystkich lokalnych zmiennych i argumentów funkcji.
Zwracana wartość jest zdejmowana ze stosu i przypisywana jako wartość instrukcji wywołania funkcji. Następnie ze stosu zdejmowana jest wartość odłożona w kroku 1.; wartość ta zostaje umieszczona we wskaźniku instrukcji. Program, posiadając wartość zwróconą przez funkcję, wznawia działanie od instrukcji następującej bezpośrednio po instrukcji wywołania funkcji.

Niektóre ze szczegółów tego procesu zmieniają się w zależności od kompilatora i komputera, ale podstawowy jego przebieg jest nie-

zmienny. Gdy wywołujesz funkcję, na stosie odkładany jest adres powrotu i argumenty. W trakcie działania tych funkcji, na stos są odkładane zmienne lokalne. Gdy funkcja wraca, ze stosu zostaje usunięte wszystko.

W następnych rozdziałach poznamy inne miejsca pamięci, używane do przechowywania danych, które muszą istnieć dłużej niż czas życia funkcji.

Bibliografia

1. Jesse Liberty, „C++ dla każdego”, Wydawnictwo Helion.