8

WSKAŹNIKI

Im bardziej zaglądał do środka,
tym bardziej nic tam nie było.
A. A. Milne „Kubuś Puchatek”

Dwa poprzednie rozdziały upłynęły nam na poznawaniu różnorodnych aspektów programowania obiektowego. Nawet teraz, w kilkanaście lat po powstaniu, jest ona czasem uważana może nie za awangardę, ale poważną nowość i „odstępstwo” od „klasycznych” reguł programowania.
Takie opinie, pojawiające się oczywiście coraz rzadziej, są po części echem dawnej popularności języka C. Fakt, że C++ zachowuje wszystkie właściwości swego poprzednika, zdaje się usprawiedliwać podejście, iż są one ważniejsze i bardziej znaczące niż „dodatki” wprowadzone wraz z dwoma plusami w nazwie języka. Do owych „dodatków” ma rzecz jasna należeć programowanie obiektowe.

Sprawia to, że ogromna większość kursów i podręczników języka C++ jest usystematyzowana wedle osobliwej zasady. Otóż mówi ona, że najpierw należy wyłożyć wszystkie zagadnienia związane z C, a dopiero potem zająć się „nowinkami”, w które został wyposażony jego następca.
Zastanawiając się nad tym bliżej, można nieomal nabrać wątpliwości, czy w ten sposób nadal uczymy się przede wszystkim programowania, czy może bardziej zajmują nas już kwestie formalne danego języka? Jeżeli nawet nie odnosimy takiego wrażenia, to nietrudno znaleźć szczęśliwsze i bardziej naturalne drogi poznania tajników kodowania.

Pamiętajmy, że programowanie jest raczej praktyczną dziedziną informatyki, a jego nauka jest w dużej mierze zdobywaniem umiejętności, a nie tylko samej wiedzy. Dlatego też wymaga ona mniej teoretycznego nastawienia, a więcej wytrwałości w osiąganiu coraz lepszego „wtajemniczenia” w zagadnienia programistyczne. Naturalną koleją rzeczy jest więc uszeregowanie tych zagadnień według wzrastającego poziomu trudności czy też ze względu na ich większą lub mniejszą użyteczność praktyczną.
Takie też założenie przyjąłem w tym kursie. Nie chcę sobie jednak robić autoreklamy twierdząc, że jest on „inny niż wszystkie” pozostałe; mam nawet nadzieję, że to określenie jest całkowitą nieprawdą i że istnieje jeszcze mnóstwo innych publikacji, których autorzy skupili się głównie na *nauczaniu programowania*, a nie na *opisywaniu języków programowania*.

Zatem zgodnie z powyższą tezą kwestie programowania obiektowego, jako niezwykle ważne same w sobie, wprowadziłem tak wcześnie jak to tylko było możliwe - nie przywiązując wagi to faktu, czy są one właściwe jeszcze językowi C, czy może już C++. Bardziej liczyła się bowiem ich rzeczywista przydatność.
Na tej samej zasadzie opieram się także teraz, gdy przyszedł czas na szczegółowe omówienie wskaźników. To również ważne zagadnienie, którego geneza nie wydaje się wcale tak bardzo istotna. Najważniejsze, iż są one częścią języka C++, w dodatku jedną z kluczowych - chociaż może nie najprostszych. Umiejętność właściwego posługiwania się wskaźnikami oraz pamięcią operacyjną jest więc niebagatelna dla programisty C++. Opanowaniu przez ciebie tej umiejętności został poświęcony cały niniejszy rozdział. Możesz więc do woli z niego korzystać :)

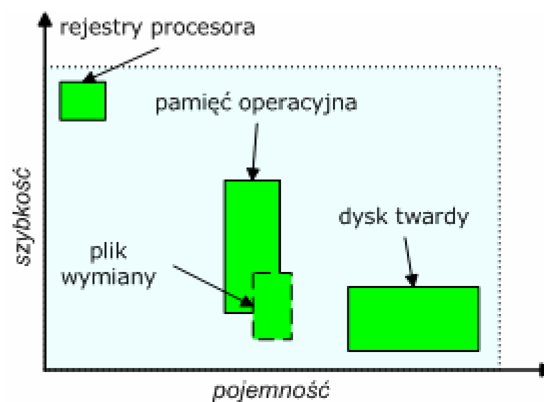
Ku pamięci

Wskaźniki są ściśle związane z pamięcią komputera - a więc miejscem, w którym przechowuje on dane. Przydatne będzie zatem przypomnienie sobie (a może dopiero poznanie?) kilku podstawowych informacji na ten temat.

Rodzaje pamięci

Można wyróżnić wiele rodzajów pamięci, jakimi dysponuje pecet, kierując się różnymi przesłankami. Najczęściej stosuje się kryteria **szybkości** i **pojemności**; są one ważne nie tylko dla nas, programistów, ale praktycznie dla każdego użytkownika komputera.

Nietrudno przy tym zauważyć, że są one ze sobą wzajemnie powiązane: im większa jest szybkość danego typu pamięci, tym mniej danych można w niej przechowywać, i na odwrót. Nie ma niestety pamięci zarówno wydajnej, jak i pojemnej - zawsze potrzebny jest jakiś kompromis.
Zjawisko to obrazuje poniższy wykres:



Wykres 2. Szybkość oraz pojemność kilku typów pamięci komputera

Zostały na nim umieszczone wszystkie rodzaje pamięci komputera, jakimi się zaraz dokładnie przyjrzymy.

Rejestry procesora

Procesor jest jednostką obliczeniową w komputerze. Nieszczególnie zatem kojarzy się z przechowywaniem danych w jakiejś formie pamięci. A jednak posiada on własne jej zasoby, które są kluczowe dla prawidłowego funkcjonowania całego systemu. Nazywamy je **rejestrami**.

Każdy rejestr ma postać pojedynczej komórki pamięci, zaś ich liczba zależy głównie od modelu procesora (generacji). Wielkość rejestru jest natomiast potocznie znana jako „bitowość" procesora: najpopularniejsze obecnie jednostki 32-bitowe mają więc rejestry o wielkości 32 bitów, czyli 4 bajtów.

> Ten sam rozmiar mają też w C++ zmienne typu `int`, i nie jest to bynajmniej przypadek :)

Większość rejestrów ma ściśle określone znaczenie i zadania do wykonania. Nie są one więc przeznaczone do reprezentowania dowolnych danych, które by się weń zmieściły. Zamiast tego pełnią różne ważne funkcje w obrębie całego systemu.
Ze względu na wykonywane przez siebie role, wśród rejestrów procesora możemy wyróżnić:

- cztery **rejestry uniwersalne** (EAX, EBX, ECX i EDX[92]). Przy ich pomocy procesor wykonuje operacje arytmetyczne (dodawanie, odejmowanie, mnożenie i dzielenie). Niektóre wspomagają też wykonywanie programów, np. EAX jest używany do zwracania wyników funkcji, zaś ECX jako licznik w pętlach. Rejestry uniwersalne mają więc największe znaczenie dla programistów (głównie asemblera), gdyż często są wykorzystywane na potrzeby ich aplikacji. Z pozostałych natomiast korzysta prawie wyłącznie sam procesor.

Każdy z rejestrów uniwersalnych zawiera w sobie mniejsze, 16-bitowe, a te z kolei po dwa rejestry ośmiobitowe. Mogą one być modyfikowane niezależnie do innych, ale trzeba oczywiście pamiętać, że zmiana kilku bitów pociąga za sobą pewną zmianę całej wartości.

- **rejestry segmentowe** pomagają organizować pamięć operacyjną. Dzięki nim procesor „wie", w której części RAMu znajduje się kod aktualnie działającego programu, jego dane itp.
- **rejestry wskaźnikowe** pokazują na ważne obszary pamięci, jak choćby aktualnie wykonywana instrukcja programu.
- dwa **rejestry indeksowe** są używane przy kopiowaniu jednego fragmentu pamięci do drugiego.

Ten podstawowy zestaw może być oczywiście uzupełniony o inne rejestry, jednak powyższe są absolutnie niezbędne do pracy procesora.

Najważniejszą cechą wszystkich rejestrów jest **błyskawiczny czas dostępu**. Ponieważ ulokowane są w samym procesorze, skorzystanie z nich nie zmusza do odbycia „wycieczki" wgłąb pamięci operacyjnej i dlatego odbywa się wręcz ekspresowo. Jest to w zasadzie najszybszy rodzaj pamięci, jakim dysponuje komputer.
Ceną za tę szybkość jest oczywiście znikoma objętość rejestrów - na pewno nie można w nich przechowywać złożonych danych. Co więcej, ich panem i władcą jest tylko i wyłącznie sam procesor, zatem nigdy nie można mieć pewności, czy zapisane w nich informacje nie zostaną zastąpione innymi. Trzeba też pamiętać, że nieumiejętne manipulowanie innymi rejestrami niż uniwersalne może doprowadzić nawet do zawieszenia komputera; na tak niskim poziomie nie ma już bowiem żadnych komunikatów o błędach…

Zmienne przechowywane w rejestrach

Możemy jednak odnieść pewne korzyści z istnienia rejestrów procesora i sprawić, by zaczęły działać po naszej stronie. Jako niezwykle szybkie porcje pamięci są idealne do przechowywania małych, ale często i intensywnie używanych zmiennych.

Na dodatek nie musimy wcale martwić się o to, w którym dokładnie rejestrze możemy w danej chwili zapisać dane oraz czy pozostaną one tam nienaruszone. Czynności te można bowiem zlecić kompilatorowi: wystarczy jedynie użyć słowa kluczowego `register` - na przykład:

```
register int nZmiennaRejestrowa;
```

Gdy opatrzymy deklarację zmiennej tym modyfikatorem, to będzie ona w miarę możliwości przechowywana w którymś z rejestrów uniwersalnych procesora. Powinno to rzecz jasna przyspieszyć działanie całego programu.

[92] Wszystkie nazwy rejestrów odnoszą się do procesorów 32-bitowych.

Dostęp do rejestrów

Rejestry procesora, jako związane ściśle ze sprzętem, są rzeczą **niskopoziomową**. C++ jest zaś językiem wysokiego poziomu i szczyci się niezależnością od platformy sprzętowej.

Powoduje to, iż nie posiada on żadnych specjalnych mechanizmów, pozwalających odczytać lub zapisywać dane do rejestrów procesora. Zdecydowała o tym nie tylko przenośność, ale i bezpieczeństwo - „mieszanie” w tak zaawansowanych obszarach systemu może bowiem przynieść sporo szkody.

Jedynym sposobem na uzyskanie dostępu do rejestrów jest skorzystanie z wstawek asemblerowych, ujmowanych w bloki `__asm`. Można o nich przeczytać w MSDN; używając ich trzeba jednak mieć świadomość, w co się pakujemy :)

Pamięć operacyjna

Do sensownego funkcjonowania komputera potrzebne jest miejsce, w którym mógłby on składować kod wykonywanych przez siebie programów (obejmuje to także system operacyjny) oraz przetwarzane przez nie dane. Jest to stosunkowo spora ilość informacji, więc wymaga znacznie więcej miejsca niż to oferują rejestry procesora. Każdy komputer posiada więc osobną **pamięć operacyjną**, przeznaczoną na ten właśnie cel. Nazywamy ją często angielskim skrótem RAM (ang. *random access memory* - pamięć o dostępie bezpośrednim).

Skąd się bierze pamięć operacyjna?

Pamięć tego rodzaju utożsamiamy zwykle z jedną lub kilkoma elektronicznymi układami scalonymi (tzw. kośćmi), włożonymi w odpowiednie miejsca płyty głównej peceta.

Fotografia 2. Kilka kości RAM typu DIMM
(zdjęcie pochodzi z serwisu Tom’s Hardware Guide)

Rzeczywiście jest to najważniejsza część tej pamięci (sama zwana jest czasem **pamięcią fizyczną**), ale na pewno nie jedyna. Obecnie wiele podzespołów komputerowych posiada własne zasoby pamięci operacyjnej, przystosowane do wykonywania bardziej specyficznych zadań.
W szczególności dotyczy to kart graficznych i dźwiękowych, zoptymalizowanych do pracy z właściwymi im typami danych. Ilość pamięci, w jaką są wyposażane, systematycznie rośnie.

Pamięć wirtualna

Istnieje jeszcze jedno, przebogate źródło dodatkowej pamięci operacyjnej: jest nim dysk twardy komputera, a ściślej jego część zwana **plikiem wymiany** (ang. *swap file*) lub **plikiem stronnicowania** (ang. *paging file*).

Obszar ten służy systemowi operacyjnemu do „udawania", iż ma pokaźnie więcej pamięci niż posiada w rzeczywistości. Właśnie dlatego taką symulowaną pamięć nazywamy **wirtualną**.
Podobny zabieg jest niewątpliwie konieczny w środowisku wielozadaniowym, gdzie naraz może być uruchomionych wiele programów. Chociaż w danej chwili pracujemy tylko z jednym, to pozostałe mogą nadal działać w tle - nawet wówczas, gdy łączna ilość potrzebnej im pamięci znacznie przekracza fizyczne możliwości komputera.

Ceną za ponadplanowe miejsce jest naturalnie wydajność. Dysk twardy charakteryzuje się dłuższym czasem dostępu niż układy RAM, zatem wykorzystanie go jako pamięci operacyjnej musi pociągnąć za sobą spowolnienie działania systemu. Dzieje się jednak tylko wtedy, gdy uruchamiamy wiele aplikacji naraz.

Mechanizm pamięci wirtualnej, jako niemal niezbędny do działania każdego nowoczesnego systemu operacyjnego, funkcjonuje zazwyczaj bardzo dobrze. Można jednak poprawić jego osiągi, odpowiednio ustawiając pewne opcje pliku wymiany. Przede wszystkim warto umieścić go na nieużywanej zwykle partycji (Linux tworzy nawet sam odpowiednią partycję) i ustalić stały rozmiar na mniej więcej dwukrotność ilości posiadanej pamięci fizycznej.

Pamięć trwała

Przydatność komputerów nie wykraczałaby wiele poza zastosowania kalkulatorów, gdyby swego czasu nie wynaleziono sposobu na trwałe zachowywanie informacji między kolejnymi uruchomieniami maszyny. Tak narodziły się dyskietki, dyski twarde, zapisywalne płyty CD, przenośne nośniki „długopisowe" i inne media, służące do długotrwałego magazynowania danych.
Spośród nich na najwięcej uwagi zasługują dyski twarde, jako że obecnie są niezbędnym elementem każdego komputera. Zwane są czasem **pamięcią trwałą** (z wyjaśnionych wyżej względów) albo **masową** (z powodu ich dużej pojemności).

Możliwość zapisania dużego zbioru informacji jest aczkolwiek okupiona ślamazarnością działania. Odczytywanie i zapisywanie danych na dyskach magnetycznych trwa bowiem zdecydowanie dłużej niż odwołanie do komórki pamięci operacyjnej. Ich wykorzystanie ogranicza się więc z reguły do jednorazowego wczytywania dużych zestawów danych (na przykład całych plików) do pamięci operacyjnej, poczynienia dowolnej ilości zmian oraz powtórnego, trwałego zapisania. Wszelkie operacje np. na otwartych dokumentach są więc w zasadzie dokonywane na ich kopiach, rezydujących wewnątrz pamięci operacyjnej.

Nie zajmowaliśmy się jeszcze odczytem i zapisem informacji z plików na dysku przy pomocy kodu C++. Nie martw się jednak, gdyż ostatecznie poznamy nawet więcej niż jeden sposób na dokonanie tego. Pierwszy zdarzy się przy okazji omawiania strumieni, będących częścią Biblioteki Standardowej C++.

Organizacja pamięci operacyjnej

Spośród wszystkich trzech rodzajów pamięci, dla nas w kontekście wskaźników najważniejsza będzie pamięć operacyjna. Poznamy teraz jej budowę widzianą z koderskiego punktu widzenia.

Adresowanie pamięci

Wygodnie jest wyobrażać sobie pamięć operacyjną jako coś w rodzaju wielkiej tablicy bajtów. W takiej strukturze każdy element (zmiemy go **komórką**) powinien dać się jednoznacznie identyfikować poprzez swój indeks. I tutaj rzeczywiście tak jest - numer danego bajta w pamięci nazywamy jego **adresem**.
W ten sposób dochodzimy też do pojęcia wskaźnika:

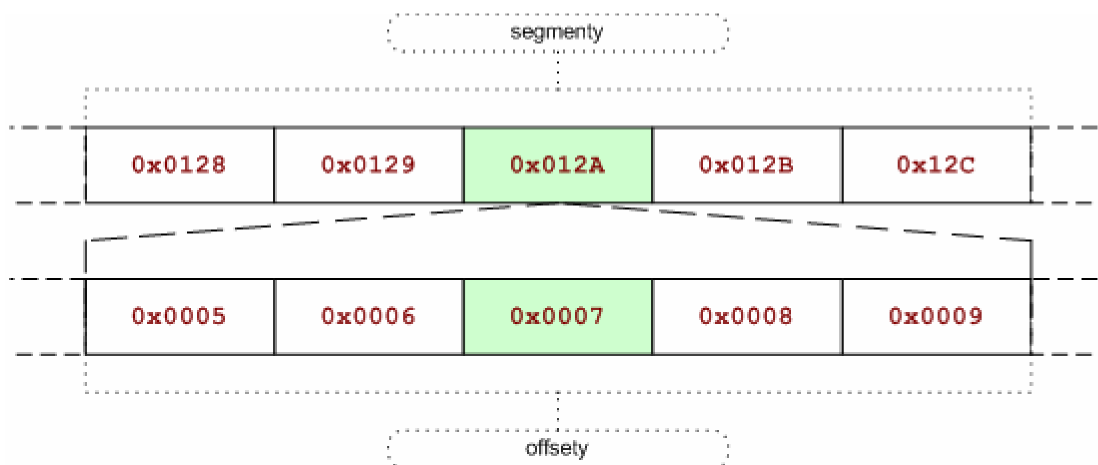
Wskaźnik (ang. *pointer*) jest adresem pojedynczej komórki pamięci operacyjnej.

Jest to więc w istocie liczba, interpretowana jako unikalny indeks danego miejsca w pamięci. Specjalne znaczenie ma tu jedynie wartość zero, interpretowana jako **wskaźnik pusty** (ang. *null pointer*), czyli nieodnoszący się do żadnej konkretnej komórki pamięci.
Wskaźniki służą więc jako łączą do określonych miejsc w pamięci operacyjnej; poprzez nie możemy **odwoływać się** do tychże miejsc. Będziemy również potrafili **pobierać wskaźniki** na zmienne oraz funkcje, zdefiniowane we własnych aplikacjach, i wykonywać przy ich pomocy różne wspaniałe rzeczy :)

Zanim jednak zajmiemy się bliżej samymi wskaźnikami w języku C++, poświęćmy nieco uwagi na to, w jaki sposób systemy operacyjne zajmują się organizacją i systematyzacją pamięci operacyjnej - czyli jej adresowaniem. Pomoże nam to lepiej zrozumieć działanie wskaźników.

Epoka niewygodnych segmentów

Dawno, dawno temu (co oznacza przełom lat 80. i 90. ubiegłego stulecia) większość programistów nie mogła być zbytnio zadowolona z metod, jakich musieli używać, by obsługiwać większe ilości pamięci operacyjnej. Była ona bowiem podzielona na tzw. **segmenty**, każdy o wielkości 64 kilobajtów.
Aby zidentyfikować konkretną komórkę należało więc podać aż dwie opisujące jej liczby: oczywiście numer segmentu, a także **offset**, czyli konkretny już indeks w ramach danego segmentu.



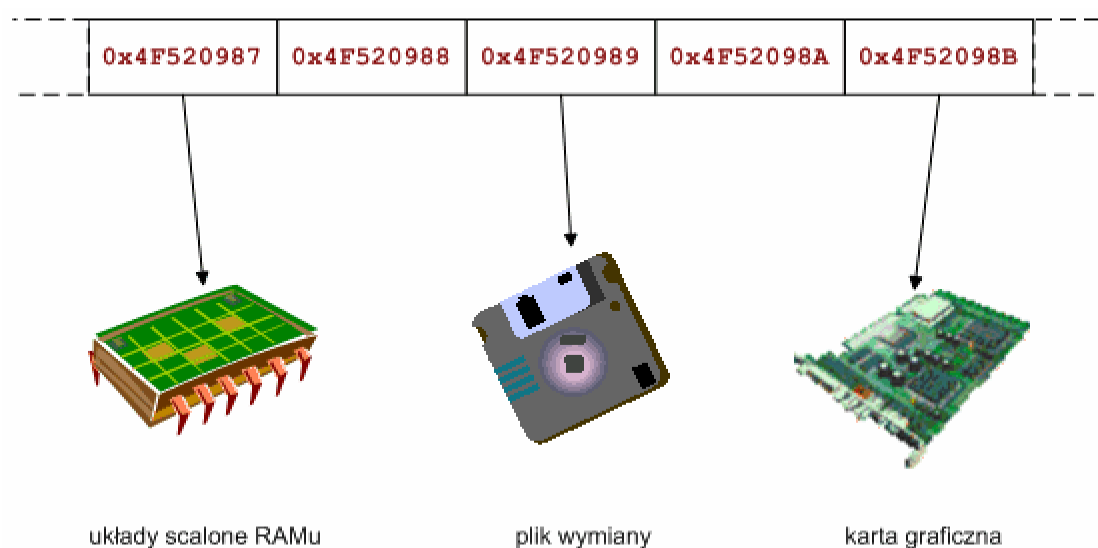
Schemat 31. Segmentowe adresowanie pamięci. Adres zaznaczonej komórki zapisywano zwykle jako 012A:0007, a więc oddzielając dwukropkiem numer segmentu i offset (oba zapisane w systemie szesnastkowym). Do ich przechowywania potrzebne były dwie liczby 16-bitowe.

Może nie wydaje się to wielką niedogodnością, ale naprawdę było nią. Przede wszystkim niemożliwe było operowanie na danych o rozmiarze większym niż owe 64 kB (a więc chociażby na długich napisach). Chodzi też o fakt, iż to programista musiał martwić się o rozmieszczenie kodu oraz danych pisanego programu w pamięci operacyjnej. Czas pokazał, że obowiązek ten z powodzeniem można przerzucić na kompilator - co zresztą wkrótce stało się możliwe.

Płaski model pamięci

Dzisiejsze systemy operacyjne mają znacznie wygodniejszy sposób organizacji pamięci RAM. Jest nim właśnie ów **płaski model** (ang. *flat memory model*), likwidujący wiele mankamentów swego segmentowego poprzednika.

32-bitowe procesory pozwalają mianowicie, by **cała pamięć** była **jednym segmentem**. Taki segment może mieć rozmiar nawet 4 gigabajtów, więc z łatwością zmieszczą się w nim wszystkie fizyczne i wirtualne zasoby RAMu.
To jednakże nie wszystko. Otóż płaski model umożliwia zgrupowanie wszystkich dostępnych rodzajów pamięci operacyjnej (kości RAM, plik wymiany, pamięć karty graficznej, itp.) w jeden ciągły obszar, zwany **przestrzenią adresową**. Programista nie musi się przy tym martwić, do jakiego szczególnego typu pamięci odnosi się dany wskaźnik! Na poziomie języka programowania znikają bowiem wszelkie praktyczne różnice między nimi: oto mamy jeden, wielki segment **całej pamięci operacyjnej** i basta!



Schemat 32. Idea płaskiego modelu pamięci. Adresy składają się tu tylko z offsetów, przechowywanych jako liczby 32-bitowe. Mogą one odnosić się do jakiegokolwiek rzeczywistego rodzaju pamięci, na przykład do takich jak na ilustracji.

W Windows dodatkowo każdy proces (program) posiada swoją własną przestrzeń adresową, niedostępną dla innych. Wymiana danych może więc zachodzi jedynie poprzez dedykowane do tego mechanizmy. Będziemy o nich mówić, gdy już przejdziemy do programowania aplikacji okienkowych.

Przy takim modelu pamięci porównanie jej do ogromnej, jednowymiarowej tablicy staje się najzupełniej słuszne. Wskaźniki można sobie wtedy całkiem dobrze wyobrażać jako indeksy tej tablicy.

Stos i sterta

Na koniec wspomnimy sobie o dwóch ważnych dla programistów rejonach pamięci operacyjnych, a więc właśnie o stosie oraz stercie.

Czym jest stos?

Stos (ang. *stack*) jest obszarem pamięci, który zostaje automatycznie przydzielony do wykorzystania dla programu.

Na stosie egzystują wszystkie zmienne zadeklarowane jawnie w kodzie (szczególne te lokalne w funkcjach), jest on także używany do przekazywania parametrów do funkcji.

Faktycznie więc można by w ogóle nie wiedzieć o jego istnieniu. Czasem jednak objawia się ono w dość nieprzyjemny sposób: poprzez błąd **przepełnienia stosu** (ang. *stack overflow*). Występuje on zwykle wtedy, gdy nastąpi zbyt wiele wywołań funkcji.

O stercie

Reszta pamięci operacyjnej nosi oryginalną nazwę **sterty**.

Sterta (ang. *heap*) to cała pamięć dostępna dla programu i mogąca być mu przydzielona do wykorzystania.

Czytając oba opisy (stosu i sterty) pewnie trudno jest wychwycić między nimi jakieś różnice, jednak w rzeczywistości są one całkiem spore.

Przede wszystkim, rozmiar stosu jest ustalany raz na zawsze podczas kompilacji programu i nie zmienia się w trakcie jego działania. Wszelkie dane, jako są na nim przechowywane, muszą więc mieć **stały rozmiar** - jak na przykład skalarne zmienne, struktury czy też statyczne tablice.
Kontrolą pamięci sterty zajmuje się natomiast sam programista i dlatego może przyznać swojej aplikacji odpowiednią jej ilość w danej chwili, **podczas działania programu**. Jest to bardzo dobre rozwiązanie, kiedy konieczne jest przetwarzanie zbiorów informacji o **zmiennym rozmiarze**.

Terminy 'stos' i 'sterta' mają w programowaniu jeszcze jedno znaczenie. Tak mianowicie nazywają się dwie często wykorzystywane struktury danych. Omówimy je przy okazji poznawania Biblioteki Standardowej C++.

***

Na tym zakończymy ten krótki wykład o samej pamięci operacyjnej. Część tych wiadomości była niektórym pewnie doskonale znana, ale chyba każdy miał okazję dowiedzieć się czegoś nowego :)

Wiedza ta będzie nam teraz szczególnie przydatna, gdyż rozpoczynamy wreszcie zasadniczą część tego rozdziału, czyli omówienie wskaźników w języku C++: najpierw na zmienne, a potem wskaźników na funkcje.

Wskaźniki na zmienne

Trudno zliczyć, ile razy stosowaliśmy zmienne w swoich programach. Takie statystyki nie mają zresztą zbytniego sensu - programowanie bez użycia zmiennych jest przecież tym samym, co prowadzenie samochodu bez korzystania z kierownicy ;D

Wiele razy przypominałem też, że zmienne rezydują w pamięci operacyjnej. Mechanizm wskaźników na nie jest więc zupełnie logiczną konsekwencją tego zjawiska. W tym podrozdziale zajmiemy się właśnie takimi wskaźnikami.

Używanie wskaźników na zmienne

Wskaźnik jest przede wszystkim liczbą - adresem w pamięci, i w takiej też postaci istnieje w programie. Język C++ ma ponadto ścisłe wymagania dotyczące kontroli typów i z tego powodu każdy wskaźnik musi mieć dodatkowo określony **typ**, na jaki wskazuje. Innymi

słowy, kompilator musi znać odpowiedź na pytanie: „Jakiego rodzaju jest zmienna, na którą pokazuje dany wskaźnik?”. Dzięki temu potrafi zachowywać kontrolę nad typami danych w podobny sposób, w jaki czyni to w stosunku do zwykłych zmiennych.

Obejmuje to także rzutowanie między wskaźnikami, o którym też sobie powiemy.

Wiedząc o tym, spójrzmy teraz na ten elementarny przykład deklaracji oraz użycia wskaźnika:

```
// deklaracja zmiennej typu int oraz wskaźnika na zmienne tego typu
int nZmienna = 10;
int* pnWskaznik;     // nasz wskaźnik na zmienne typu int

// przypisanie adresu zmiennej do naszego wskaźnika i użycie go do
// wyświetlenia jej wartości w konsoli
pnWskaznik = &nZmienna;    // pnWskaznik odnosi się teraz do nZmienna
std::cout << *pnWskaznik;  // otrzymamy 10, czyli wartość zmiennej
```

Dobra wiadomość jest taka, iż mimo prostoty ilustruje on większość zagadnień związanych ze wskaźnikami na zmiennej. Nieco gorszą jest pewnie to, że owa prostota może dla niektórych nie być wcale taka prosta :) Naturalnie, wyjaśnimy sobie po kolei, co dzieje się w powyższym kodzie (chociaż komentarze mówią już całkiem sporo).

Oczywiście najpierw mamy deklarację zmiennej (z inicjalizacją), lecz nas interesuje bardziej sposób zadeklarowania wskaźnika, czyli:

```
int* pnWskaznik;
```

Poprzez dodanie gwiazdki (`*`) do nazwy typu `int` informujemy kompilator, że oto nie ma już do czynienia ze zwykłą zmienną liczbową, ale ze wskaźnikiem przeznaczonym do przechowywania **adresu** takiej zmiennej. `pWskaznik` jest więc **wskaźnikiem na zmienne typu `int`**, lub, krócej, **wskaźnikiem na (typ) `int`**.

A zatem mamy już zmienną, mamy i wskaźnik. Przydałoby się zmusić je teraz do współpracy: niech `pWskaznik` zacznie odnosić się do naszej zmiennej! Aby tak było, musimy **pobrać jej adres** i przypisać go do wskaźnika - o tak:

```
pnWskaznik = &nZmienna;
```

Zastosowany tutaj operator `&` służy właśnie w tym celu - do **uzyskania adresu** miejsca w pamięci, gdzie egzystuje zmienna. Potem rzecz jasna zostaje on zapisany w `pnWskaznik`; odtąd wskazuje on więc na zmienną `nZmienna`.

Na koniec widzimy jeszcze, że za pośrednictwem wskaźnika możemy dostać się do zmiennej i użyć jej w ten sam sposób, jaki znaliśmy dotychczas, choćby do wypisania jej wartości w oknie konsoli:

```
std::cout << *pnWskaznik;
```

Jak z pewnością przypuszczasz, operator `*` nie dokonuje tutaj mnożenia, lecz **podejmuje wartość** zmiennej, z którą połączony został `pnWskaznik`; nazywamy to **dereferencją wskaźnika**. W jej wyniku otrzymujemy na ekranie liczbę, którą oryginalnie przypisaliśmy do zmiennej `nZmienna`. Bez zastosowania wspomnianego operatora zobaczylibyśmy **wartość wskaźnika** (a więc adres komórki w pamięci), nie zaś **wartość zmiennej**, na którą on pokazuje. To oczywiście wielka różnica.

Zaprezentowana próbka kodu faktycznie realizuje zatem zadanie wyświetlenia wartości zmiennej nZmienna w iście okrężny sposób. Zamiast bezpośredniego przesłania jej do strumienia wyjścia posługujemy się w tym celu dodatkowym pośrednikiem w postaci wskaźnika.
Samo w sobie może to budzić wątpliwości co do sensowności korzystania ze wskaźników. Pomyślmy jednak, że mając wskaźnik możemy umożliwić dostęp do danej zmiennej z jakiegokolwiek miejsca programu - na przykład z funkcji, do której przekażemy go jako parametr (w końcu to tylko liczba!). Potrafimy wtedy zaprogramować każdą czynność (algorytm) i zapewnić jej wykonanie w stosunku do **dowolnej ilości zmiennych**, pisząc odpowiedni kod **tylko raz**.

Więcej przekonania do wskaźników na zmiennej nabierzesz wówczas, gdy poznasz je bliżej - i temu właśnie zadaniu poświęcimy teraz uwagę.

Deklaracje wskaźników

Stwierdziliśmy, że wskaźniki mogą z powodzeniem odnosić się do zmiennych - albo ogólnie mówiąc, do danych w programie. Czynią to poprzez przechowywanie numeru odpowiedniej komórki w pamięci, a zatem pewnej **wartości**. Sprawia to, że wskaźniki są w rzeczy samej także zmiennymi.

Wskaźniki w C++ to zmienne należące do specjalnych **typów wskaźnikowych**.

Taki typ łatwo poznać po obecności przynajmniej jednej gwiazdki w jego nazwie. Jest nim więc choćby int* - typ zmiennej pWskaznik z poprzedniego przykładu. Zawiera on jednocześnie informację, na jaki rodzaj danych będzie nasz wskaźnik pokazywał - tutaj jest to int. Typ wskaźnikowy jest więc **typem pochodnym**, zdefiniowanym na podstawie jednego z już wcześniej istniejących.

To definiowanie może się odbywać *ad hoc*, podczas deklarowania konkretnej zmiennej (wskaźnika) - tak było w naszym przykładzie i tak też postępuje się najczęściej. Dozwolone (i przydatne) jest aczkolwiek stworzenie aliasów na typy wskaźnikowe poprzez instrukcję typedef; standardowe nagłówki systemu Windows zawierają na przykład wiele takich nazw.

Deklarowanie wskaźników jest zatem niczym innym, jak tylko wprowadzeniem do kodu nowych zmiennych - tyle tylko, iż mają one swoiste przeznaczenie, inne niż reszta ich licznych współbraci. Czynność ich deklarowania, a także same typy wskaźnikowe zasługują przeto na szersze omówienie.

Nieodżałowany spór o gwiazdkę

Dowiedzieliśmy się już, że pisząc gwiazdkę po nazwie jakiegoś typu, uzyskujemy odpowiedni wskaźnik na ten typ. Potem możemy użyć go, deklarując właściwy wskaźnik; co więcej, możliwe jest uczynienie tego aż na cztery sposoby:

```
int* pnWskaznik;
int *pnWskaznik;
int*pnWskaznik;
int * pnWskaznik;
```

Widać więc, że owa gwiazdka „nie trzyma się” kurczowo nazwy typu (tutaj int) i może nawet oddzielać go od nazwy deklarowanej zmiennej, bez potrzeby użycia w tym celu spacji.

Wydawałoby się, że taka swoboda składniowa powinna tylko cieszyć. W rzeczywistości jednak powoduje najczęściej trudności w rozumieniu kodu napisanego przez innych,

jeżeli używają oni innego sposobu deklarowania wskaźników niż „nasz”. Dlatego też wielokrotnie próbowano ustalić jakiś jeden, słuszny wariant w tej materii… i w zasadzie nigdy się to nie udało!

Podobnie rzecz ma się także z umieszczaniem nawiasów klamrowych po instrukcjach `if`, `else` oraz nagłówkach pętli.

Jeśli więc chodzi o dwa ostatnie sposoby, to generalnie prawie nikt nich nie używa i raczej nie jest to niespodzianką. Nieużywanie spacji czyni instrukcję mało czytelną, zaś ich obecność po obu stronach znaku `*` nieodparcie przywodzi na myśl mnożenie, a nie deklarację zmiennej.
Co do dwóch pierwszych metod, to w kwestii ich używania panuje niczym niezmącona dowolność… Poważnie! W kodach, jakie spotkasz, na pewno będziesz miał okazję zobaczyć obie te składnie. Argumenty stojące za ich wykorzystaniem są niemal tak samo silne w przypadku każdej z nich - tak przynajmniej twierdzą ich zwolennicy.

Temu problemowi poświęcony jest nawet fragment FAQ autora języka C++.

Zauważyłeś być może, iż w tym kursie używam pierwszej konwencji i będę się tego konsekwentnie trzymał. Nie chcę jednak nikomu jej narzucać; najlepiej będzie, jeśli sam wypracujesz sobie odpowiadający ci zwyczaj i, co najważniejsze, będziesz go konsekwentnie przestrzegał. Nie ma bowiem nic gorszego niż niespójny kod.

Z opisywanym problemem wiąże się jeszcze jeden dylemat, powstający gdy chcemy zadeklarować kilka zmiennych - na przykład tak:

```
int* a, b;
```

Czy w ten sposób otrzymamy dwa wskaźniki (zmienne typu `int*`)?… Pozostawiam to zainteresowanym do samodzielnego sprawdzenia[93]. Odpowiedź nie jest taka oczywista, jak by się to wydawało na pierwszy rzut oka, zatem stosowanie takiej konstrukcji pogarsza czytelność kodu i może być przyczyną błędów. Czuje się więc w obowiązku przestrzec przed nią:

Nie próbuj deklarować kilku wskaźników w jednej instrukcji, oddzielając je przecinkami.

Trzeba niestety przyznać, że język C++ zawiera w sobie jeszcze kilka podobnych niejasności. Będę zwracał na nie uwagę w odpowiednim czasie i miejscu.

Wskaźniki do stałych

Wskaźniki mają w C++ pewną, dość oryginalną cechę. Mianowicie, nierzadko aplikuje się do nich modyfikator `const`, a mimo to cały czas możemy je nazywać zmiennymi. Dodatkowo, ów modyfikator może być doń zastosowany aż na dwa różne sposoby.

Pierwszy z nich zakłada poprzedzenie nim **całej deklaracji** wskaźnika, co wygląda mniej więcej tak:

```
const int* pnWskaznik;
```

`const`, jak wiemy, zmienia nam zmienną w **stałą**. Tutaj mamy jednak do czynienia ze wskaźnikiem na zmienną, zatem działanie modyfikatora powoduje jego zmianę we… **wskaźnik na stałą** :)

[93] Można skorzystać z podanego wcześniej linka do FAQa.

Wskaźnik na stałą (ang. *pointer to constant*) pokazuje na wartość, która może być poprzez ten wskaźnik **jedynie odczytywana**.

Przypatrzmy się, jak wskaźnik na stałą może być wykorzystany w przykładowym kodzie:

```
// deklaracja zmiennej i wskaźnika do stałej
float fZmienna = 3.141592;
const float* pfWskaznik;

// związanie zmiennej ze wskaźnikiem
pfWskaznik = &fZmienna;

// pokazanie wartości zmiennej poprzez wskaźnik
std::cout << *pfWskaznik;
```

Przykład ten jest podobny do poprzedniego: za pośrednictwem wskaźnika **odczytujemy** tu wartość zmiennej. Dozwolne jest zatem, aby ów wskaźnik był wskaźnikiem na stałą - jako taki więc go deklarujemy:

```
const float* pfWskaznik;
```

Rożnica, jaką czyni modyfikator `const`, ujawni się przy próbie zapisania wartości do zmiennej, na którą pokazuje wskaźnik:

```
*pfWskaznik = 1.0;   // BŁĄD! pfWskaznik pokazuje na stałą wartość
```

Kompilator nie pozwoli na to. Decydując się na zadeklarowanie wskaźnika na stałą (tutaj typu `const float*`) uznaliśmy bowiem, że będziemy tylko odczytywać wartość, do której się on odnosi. Zapisywanie jest oczywiście pogwałceniem tej zasady.

Powyższa linijka byłaby rzecz jasna poprawna, gdyby `pfWskaznik` był zwykłym wskaźnikiem typu `float*`.

Jeżeli wskaźnik **na stałą** jest dodatkowo wskaźnikiem **na obiekt**, to na jego rzecz możliwe jest wywołanie jedynie **stałych metod**. Nie modyfikują one bowiem pól obiektu.

Wskaźnik na stałą umożliwia więc zabezpieczenie przed niepożądaną modyfikacją wartości, na którą wskazuje. Z tego wzgledu jest dosyć często wykorzystywany w praktyce, chociażby przy przekazywaniu parametrów do funkcji.

Stałe wskaźniki

Druga możliwość użycia `const` powoduje nieco inny efekt. Odmienne jest wówczas także umiejscowienie modyfikatora w deklaracji wskaźnika:

```
float* const pfWskaznik;
```

Takie ustawienie powoduje mianowicie zadeklarowanie **stałego wskaźnika** zamiast wskaźnika na stałą.

Stały wskaźnik (ang. *const(ant) pointer*) jest **nieruchomy**, na zawsze przywiązany do **jednego adresu** pamięci.

Ten jeden jedyny i niezmienny adres możemy określić **tylko podczas inicjalizacji** wskaźnika:

```
float fA;
```

```
float* const pfWskaznik = &fA;
```

Wszelkie późniejsze próby związania wskaźnika z inną komórką pamięci (czyli inną zmienną) skończą się niepowodzeniem:

```
float fB;
pfWskaznik = &fB;     // BŁĄD! pfWskaznik jest stałym wskaźnikiem
```

Zadeklarowanie stałego wskaźnika jest bowiem umową z kompilatorem, na mocy której zobowiązujemy się **nie zmieniać adresu**, do którego tenże wskaźnik pokazuje.

Pole zastosowań stałych wskaźników jest, przyznam szczerze, raczej wąskie. Mimo to mieliśmy już okazję korzystać z tego rodzaju wskaźników - i to niejednokrotnie. Gdzie? Otóż stałym wskaźnikiem jest `this`, który, jak pamiętamy, pokazuje wewnątrz metod klasy na aktualny jej obiekt. Nie ogranicza on w żaden sposób dostępu do tego obiektu, jednak nie pozwala na zmianę samego **wskazania**; jest więc **trwale związany** z tym obiektem.
Typem wskaźnika `this` wewnątrz metod klasy *`klasa`* jest więc *`klasa*`* `const`.

W przypadku stałych metod wskaźnik `this` nie pozwala także na modyfikację pól obiektu, a zatem wskazuje na stałą. Jego typem jest wtedy `const` *`klasa*`* `const`, czyli mikst obu rodzajów „stałości” wskaźnika.

Podsumowanie deklaracji wskaźników

Na sam koniec tematu deklarowania wskaźników tradycyjnie podam trochę wskazówek dotyczacych składni oraz stosowalności praktycznej.

Składnie deklaracji wskaźnika możemy, opierając się na przykładach z poprzednich paragrafów, przedstawić następująco:

```
[const] typ* [const] wskaźnik;
```

Możliwość występowania lub niewystępowania modyfikatora `const` w aż dwóch miejscach deklaracji pozwala stwierdzić, że z każdego *`typu`* możemy wyprowadzić łącznie nawet **cztery** odpowiednie typy wskaźnikowe. Ich charakterystykę przedstawia poniższa tabelka:

typ wskaźnikowy	***nazwa***	***dostęp do pamięci***	***zmiana adresu***
`typ*`	wskaźnik (zwykły)	odczyt i zapis	dozwolona
`const` *`typ*`*	wskaźnik do stałej	wyłącznie odczyt	dozwolona
`typ*` `const`	stały wskaźnik	odczyt i zapis	niedozwolona
`const` *`typ*`* `const`	stały wskaźnik do stałej	wyłącznie odczyt	niedozwolona

Tabela 12. Zestawienie typów wskaźnikowych

Czy jest jakiś prosty sposób na zapamiętanie, która deklaracja odpowiada jakiemu rodzajowi wskaźników? No cóż, może nie jest to banalne, ale w pewien sposób zawsze można sobie pomóc. Przede wszystkim patrzmy na frazę bezpośrednio **za modyfikatorem** `const`.
Dla stałych wskaźników (przypominam, że to te, które zawsze wskazują na to samo miejsce w pamięci) deklaracja wygląda tak:

```
typ* const wskaźnik;
```

Bezpośrednio po słowie `const` mamy więc nazwę *`wskaźnika`*, co razem daje `const` *`wskaźnik`*. W wolnym tłumaczeniu znaczy to oczywiście ‘stały wskaźnik’ :)

W przypadku wskaźników na stałe forma deklaracji przedstawia się następująco:

```
const typ* wskaźnik;
```

Używamy tu `const` w ten sam sposób, w jaki ze zmiennych czynimy stałe. W tym przypadku mamy rzecz jasna do czynienia ze 'wskaźnikiem na zmienną', a ponieważ `const` przemienia nam 'zmienną' w 'stałą', więc ostatecznie otrzymujemy 'wskaźnik na stałą'. Potwierdzenia tego możemy szukać w tabelce.

Niezbędne operatory

Na wszelkich zmiennych można w C++ wykonywać jakieś operacje i wskaźniki nie są w tym względnie żadnym wyjątkiem. Posiadają nawet własne instrumentarium specjalnych operatorów, dokonujących na nich pewnych szczególnych działań. To na nich właśnie skupimy się teraz.

Pobieranie adresu i dereferencja

Wskaźnik powinien na coś wskazywać - to znaczy przechowywać adres jakieś komórki w pamięci. Taki adres można uzyskać na wiele sposobów, w zależności od tego, jakie znaczenie ma owa komórka w programie. Dla zmiennych właściwą metodą jest użycie **operatora pobierania adresu**, oznaczanego znakiem `&` (ampersandem).
Popatrzmy na niniejszy przykład:

```
// zadeklarowanie zmiennej oraz odpowiedniego wskaźnika
unsigned uZmienna;
unsigned* puWskaznik;

// pobranie adresu zmiennej i zapisanie go we wskaźniku
puWskaznik = &uZmienna;
```

Wyrażenie `&uZmienna` reprezentuje tutaj wartość liczbową, będącą **adresem miejsca w pamięci**, w którym rezyduje zmienna `uZmienna`. Typem tej zmiennej jest `unsigned`; wyrażenie `&uZmienna` jest natomiast przynależne typowi wskaźnikowemu `unsigned*`. Przypisujemy go więc zmiennej tego typu, czyli wskaźnikowi `puWskaznik`. Odtąd odnosi się on do naszej zmiennej liczbowej i może być użyty w celu odwołania się do niej.
Prezentowany tu operator `&` jest więc **unarny** - żądą tylko jednego argumentu: obiektu, którego adres ma uzyskać. Zwraca go w wyniku, zaś typem tego rezultatu jest odpowiedni typ wskaźnikowy - zobaczyliśmy to zresztą na powyższym przykładzie.

> Przypominam, że adres zmiennej możemy przypisać jedynie do **niestałego** („ruchomego”) wskaźnika.

Mając wskaźnik, chciałoby się odwołać do komórki w pamięci, czyli zmiennej, na którą on wskazuje. Potrzebujemy zatem operatora, który dokona czynności odwrotnej niż operator `&`, a więc wydobędzie zmienną spod adresu przechowywanego przez wskaźnik. Dokonuje tego **operator dereferencji**, symbolem którego jest `*` (asterisk albo po prostu gwiazdka). Czynność przez niego wykonywaną nazywamy więc **dereferencją** wskaźnika. Wystarczy spojrzeć na poniższy kod, a wszystko stanie się jasne:

```
// zapisanie wartości w komórce pamięci, na którą pokazuje wskaźnik
*puWskaznik = 123;

// odczytanie i wyświetlenie tej wartości
std::cout << "Wartosc zmiennej uZmienna: " << *puWskaznik;
```

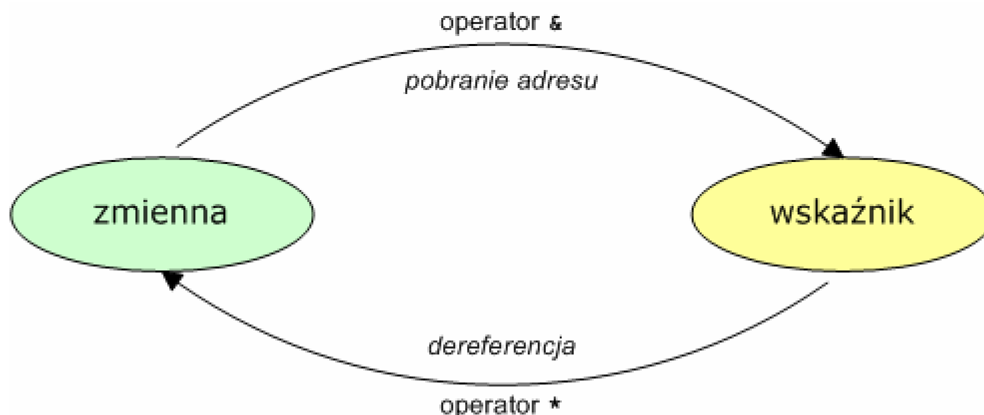
Widzimy, że operator ten jest także **unarny**, co w oczywisty sposób różni go od operatora mnożenia, który w C++ jest przecież reprezentowany przez ten sam znak.

Argumentem operatora jest naturalnie **wskaźnik**, przechowujący adres miejsca w pamięci, do którego chcemy się dostać. W wyniku działania tego operatora otrzymujemy możliwość odczytania oraz ewentualnie zapisania tam jakiejś wartości.
Typ tej wartości musi się jednak zgadzać z typem wskaźnika: jeżeli u nas był to `unsigned*`, to po dereferencji zostanie typ `unsigned`, akceptujący tylko dodatnie liczby całkowite. Podobnie z wyrażenia `*puWskaznik` możemy skorzystać jedynie tam, gdzie dozwolone są tego rodzaju wartości.

Wyrażenie `*pWskaznik` jest tu tak zwaną **l-wartością** (ang. *l-value*). Nazwa bierze się stąd, iż taka wartość może występować po lewej (ang. *left*) stronie operatora przypisania. Typowymi l-wartościami są więc zmienne, a w ogólności są to wszystkie wyrażenia, za którymi kryją się konkretne miejsca w pamięci operacyjnej i które nie zostały opatrzone modyfikatorem `const`.
Dla odróżnienia, **r-wartość** (ang. *r-value*) jest dopuszczalna tylko po prawej (ang. *right*) stronie operatora przypisania. Ta grupa obejmuje oczywiście wszystkie l-wartości, a także liczby, znaki i ich łańcuchy (tzw. stałe dosłowne) oraz wyniki obliczeń z użyciem wszelkiego rodzaju operatorów (wykorzystujących tymczasowe obiekty).

Pamiętajmy, że zapisanie danych do komórki pokazywanej przez wskaźnik jest możliwe tylko wtedy, gdy **nie jest** on wskaźnikiem **do stałej**.

Natura operatorów `&` i `*` sprawia, że najlepiej rozpatrywać je łącznie. Powiedzieliśmy sobie nawet, że ich funkcjonowanie jest sobie wzajemnie **przeciwstawne**. Ilustruje to dobrze poniższy diagram:



Schemat 33. Działanie operatorów: pobrania adresu i dereferencji

Warto również wiedzieć, że pobranie adresu zmiennej oraz dereferencja wskaźnika są możliwe **zawsze**, niezależnie od typu tejże zmiennej czy też wskaźnika. Dopiero inne związane z tym operacje, takie jak zachowanie adresu w zmiennej wskaźnikowej lub zapisanie wartości w miejscu, do którego odwołuje się wskaźnik, może napotykać ograniczenia związane z typami zmiennej i/lub stosowanego wskaźnika.

Wyłuskiwanie składników

Trzeci operator wskaźnikowy jest nam już znany od wprowadzenia OOPu. **Operator wyłuskania** `->` (strzałka) służy do wybierania składników obiektu, na który wskazuje wskaźnik. Pod pojęciem 'obiektu' kryje się tu zarówno instancja klasy, jak i typu strukturalnego lub unii.

Ponieważ znamy już doskonale tę konstrukcję, na prostym przykładzie prześledzimy jedynie związek tego operatora z omówionymi przed chwilą `&` i `*`.
Załóżmy więc, że mamy taką oto klasę:

```
class CFoo
{
   public:
         int Metoda() const { return 1; }
};
```

Tworząc dynamicznie jej instancję przy użyciu wskaźnika, możemy wywołać składowe metody:

```
// stworzenie obiektu
CFoo* pFoo = new CFoo;

// wywołanie metody
std::cout << pFoo->Metoda();
```

`pFoo` jest tu wskaźnikiem, takim samym jak te, z których korzystaliśmy dotąd; wskazuje na typ złożony - obiekt. Wykorzystując operator `->` potrafimy dostać się do tego obiektu i wywołać jego metodę, co też niejednokrotnie czyniliśmy w przeszłości.
Zwróćmy jednakowoż uwagę, że ten sam efekt osiągnęlibyśmy dokonując dereferencji naszego wskaźnika i stosując drugi z operatorów wyłuskania - kropkę:

```
// inna metoda wywołania metody Metoda() ;D
(*pFoo).Metoda();

// zniszczenie obiektu
delete pFoo;
```

Nawiasy pozwalają nie przejmować się tym, który z operatorów: `*` czy `.` ma wyższy priorytet. Ich wykorzystywanie jest więc zawsze wskazane, o czym zresztą nie raz wspominam :)

Analogicznie, można instancjować obiekt poprzez zmienną obiektową i mimo to używać operatora `->` celem dostępu do jego składowych:

```
// zmienna obiektowa
CFoo Foo;

// obie poniższe linijki robią to samo
std::cout << Foo.Metoda();
std::cout << (&Foo)->Metoda();
```

Tym razem bowiem pobieramy adres obiektu, czyli wskaźnik na niego, i aplikujemy doń wskaźnikowy operator wyłuskania `->`.

Widzimy zatem wyraźnie, że oba operatory wyłuskania mają charakter mocno umowny i teoretycznie mogą być stosowane zamiennie. W praktyce jednak korzysta się zawsze z kropki dla zmiennych obiektowych oraz strzałki dla wskaźników, i to z bardzo prostego powodu: wymuszenie zaakceptowania drugiego z operatorów wiąże się przecież z **dodatkową czynnością** pobrania adresu albo dereferencji. Łącznie zatem używamy wtedy **dwóch operatorów** zamiast jednego, a to z pewnością może odbić się na wydajności kodu.

Konwersje typów wskaźnikowych

Dwa poznane operatory nie wyczerpują rzecz jasna asortymentu operacji, jakich możemy dokonywać na wskaźnikach. Dosyć często zachodzi bowiem potrzeba przypisywania wskaźników, których typy są w większym lub mniejszym stopniu niezgodne - podobnie

zresztą jak to czasem bywa dla zwykłych zmiennych. W takich przypadkach z pomocą przychodzą nam różne metody **konwersji** typów wskaźnikowych, jakie oferuje C++.

Matka wszystkich wskaźników

Przypomnijmy sobie definicję wskaźnika, jaką podaliśmy na początku rozdziału. Otóż jest to przede wszystkim adres jakiejś komórki (miejsca) w pamięci. Przy jej płaskim modelu sprowadza się to do pojedynczej liczby bez znaku.

Na przechowywanie takiej liczby wystarczyłby więc tylko jeden typ zmiennej liczbowej! C++ oferuje jednak możliwość definiowania własnych typów wskaźnikowych w oparciu o już istniejące, inne typy. Cel takiego postępowania jest chyba oczywisty: tylko znając typ wskaźnika możemy dokonać jego dereferencji i uzyskać zmienną, na którą on wskazuje. Informacja o docelowym typie wskazywanych danych jest więc niezbędna do ich użytkowania.

Możliwe jest aczkolwiek zadeklarowanie **ogólnego wskaźnika** (ang. *void pointer* lub *pointer to void*), któremu nie są przypisane żadne informacje o typie. Taki wskaźnik jest więc jedynie adresem samym w sobie, bez dodatkowych wiadomości o rodzaju danych, jakie się pod tym adresem znajdują.
Aby zadeklarować taki wskaźnik, zamiast nazwy typu wpisujemy mu `void`:

```
void* pWskaznik;      // wskaźnik, który może pokazywać na wszystko
```

Ustalamy tą drogą, iż nasz wskaźnik nie będzie związany z żadnym konkretnym typem zmiennych. Nic nie wiadomo zatem o komórkach pamięci, do których się on odnosi - mogą one zawierać **dowolne dane**.
Brak informacji o typie upośledza jednak podstawowe właściwości wskaźnika. Nie mogąc określić rodzaju danych, na które pokazuje wskaźnik, kompilator nie może pozwolić na dostęp do nich. Powoduje to, że:

Niedozwolone jest dokonanie dereferencji ogólnego wskaźnika typu `void*`.

Cóż bowiem otrzymalibyśmy w jej wyniku? Jakiego typu byłoby wyrażenie `*pWskaznik`? `void`?... Nie jest to przecież żaden konkretny typ danych. Słusznie więc dereferencja wskaźnika typu `void*` jest niemożliwa.

Ułomność takich wskaźników nie jest zbytnią zachętą do ich stosowania. Czym więc zasłużyły sobie na tytuł paragrafu im poświęconego?...
Otóż mają one jedną szczególną i przydatną cechę, związaną z brakiem wiadomości o typie. Mianowicie:

Wskaźnik typu `void*` może przechowywać **dowolny adres** z pamięci operacyjnej.

Możliwe jest zatem przypisanie mu wartości każdego innego wskaźnika (z wyjątkiem wskaźników na stałe). Poprawny jest na przykład taki oto kod:

```
int nZmienna;
void* pWskaznik = &nZmienna;      // &nZmienna jest zasadniczo typu int*
```

Fakt, że wskaźnik typu `void*` to tylko sam adres, bez dodatkowych informacji o typie, przeznaczonych dla kompilatora, sprawia, że owe informacje są **tracone** w momencie przypisania. Wskazywanym w pamięci danym nie dzieje się naturalnie żadna krzywda, jedynie my tracimy możliwość odwoływania się do nich poprzez dereferencję.

Czy przypadkiem czegoś nam to nie przypomina?... W miarę podobna sytuacja miała przecież okazję zainstnieć przy okazji programowania obiektowego i polimorfizmu.

Wskaźnik do obiektu klasu pochodnej mogliśmy bowiem przypisać do wskaźnika na obiekt klasy bazowej i używać go potem tak samo, jak każdego innego wskaźnika na obiekt tej klasy.
Tutaj typ void* jest czymś rodzaju „typu bazowego” dla wszystkich innych typów wskaźnikowych. Możliwe jest zatem przypisywanie ich wskaźników zmiennym typu void*. Wówczas tracimy wprawdzie wiedzę o pierwotnym typie wskaźnika, ale zachowujemy to, co najważniejsze: **adres** przechowywany przez wskaźnik

Przywracanie do stanu używalności

Cały problem z ogólnymi wskaźnikami polega na tym, że przy ich pomocy nie możemy w zasadzie zrobić niczego konkretnego. Dereferencja nie wchodzi w grę z powodu niedostatecznych informacji o typie danych, na które wskaźnik pokazuje. Żeby móc z tych danych skorzystać, musimy więc przekazać kompilatorowi niezbędne informacje o ich typie. Dokonujemy tego poprzez rzutowanie.

Operacja rzutowania wskaźnika typu void* na inny typ wskaźnikowy jest przede wszystkim zabiegiem formalnym. Zarówno przed nią, jak i po niej, mamy bowiem do czynienia z adresem **tej samej komórki** w pamięci. Jej zawartość jest jednak inaczej interpretowana.
Dokonanie takiego rzutowania nie jest trudne - wystarczy posłużyć się standardowym operatorem static_cast:

```
// zmienna oraz ogólny wskaźnik, do której zapiszemy jej adres
int nZmienna = 17011987;
void* pVoid = &nZmienna;

// ponowne wykorzystanie owego adresu we wskaźniku na typ unsigned
// stosujemy rzutowanie, aby przypisać mu wskaźnik typu void*
unsigned* puLiczba = static_cast<unsigned*>(pVoid);

// wyświetlenie wartości pokazywanej przez wskaźnik
std::cout << *puLiczba;    // wynikiem jest wartość zmiennej nZmienna
```

W powyższym przykładzie wskaźnik typu int* zostaje najpierw zredukowany do void*, by potem poprzez rzutowanie zostać zinterpretowany jako unsigned*. Cały czas pokazuje on oczywiście na to samo miejsce w pamięci, tyle że w toku programu jest ono traktowane na różne sposoby.

Między palcami kompilatora

Chwileczkę! Przecież tą drogą możemy zwyczajnie oszukać kompilator i sprawić, że zacznie on traktować jakiś typ danych jako zupełnie inny, nawet całkowicie niezwiązany z tym oryginalnym!

Istotnie - za pośrednictwem wskaźnika typu void* możliwe jest **dosłownie zinterpretowanie** ciągu bitów jako dowolnego typu zmiennych. Dzieje się tak dlatego, że podczas rzutowania nie jest dokonywane żadne sprawdzenie faktycznej poprawności typów. static_cast nie działa tak jak dynamic_cast i **nie kontroluje** sensowności oraz celowości rzutowania.

> Zakres stosowalności dynamic_cast jest zaś, jak pamiętamy, ograniczony tylko do typów polimorficznych. Skalarne typy podstawowe z pewnościa nimi nie są, dlatego nie możemy do nich używać tego typu rzutowania.

Potencjalnie więc dostajemy do ręki brzytwę, którą można się nieźle pokaleczyć. W określonych sytuacjach potrzebne jest jednak takie dosłowne potraktowanie pewnego

rodzaju danych jako zupełnego innego. Pośrednictwo typu void* w niskopoziomowych konwersjach między wskaźnikami staje się wtedy kłopotliwe.
Z tego powodu (a także z potrzeby całkowitego zastąpienia rzutowania w stylu C) wprowadzono do C++ kolejny **operator rzutowania** - reinterpret_cast. Potrafi on rzutować dowolny typ wskaźnikowy na dowolny inny typ wskaźnikowy i nie tylko. Konwersje przy użyciu tego operatora prawie zawsze **nie są** więc **bezpieczne** i powinny być stosowane wyłącznie wtedy, gdy zależy nam na **mechanicznej zmianie** (bit po bicie) jednego typu danych w inny.
Jeżeli chodzi o przykłady, to chyba jedynym bezpiecznym zastosowaniem reinterpret_cast jest zapisanie adresu pamięci ze wskaźnika do zwykłej zmiennej liczbowej:

```
int* pnWskaznik;
unsigned uAdres = reinterpret_cast<unsigned>(pnWskaznik);
```

W innych przypadkach stosowanie tego operatora powinno być wyjątkowo ostrożne i oszczędne.

Kompletnych informacji o reinterpret_cast dostarcza oczywiście MSDN. Jest tam także ciekawy artykuł, wyjaśniający dogłębnie różnice między tym operatorem, a zwykłym rzutowaniem static_cast.

Istnieje jeszcze jeden, czwarty operator rzutowania const_cast. Jego zastosowanie jest bardzo wąskie i ogranicza się do usuwania modyfikatora const z opatrzonych nim typów danych. Można więc użyć go, aby zmienić stały wskaźnik lub wskaźnik do stałej w zwykły.
Bliższe informacje na temat tego operatora można naturalnie znaleźć we wiadomym źródle :)

Wskaźniki i tablice

Tradycyjnie wskaźników używa się do operacji na tablicach. Celowo piszę tu 'tradycyjnie', gdyż prawie wszystkie te operacje można wykonać także bez użycia wskaźników, więc korzystanie z nich w C++ nie jest tak popularne jak w jego generacyjnym poprzedniku. Ponieważ jednak czasem będziemy zmuszeni korzystać z kodu wywodzącego się z czasów C (na przykład z Windows API), wiedza o zastosowaniu wskaźników w stosunku do tablic może być przydatna. Obejmuje ona także zagadnienia łańcuchów znaków w stylu C, którym poświęcimy osobny paragraf.

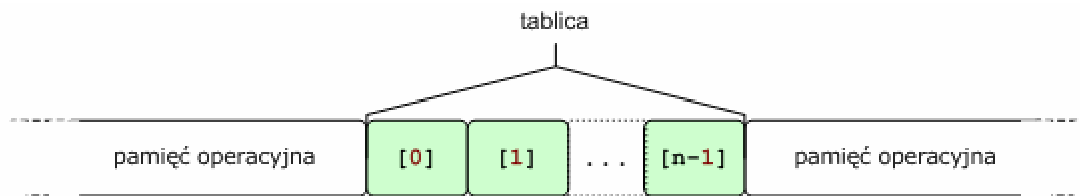
Już słyszę głosy oburzenia: „Przecież miałeś zajmować się nauczaniem C++, a nie wywlekaniem jego różnic w stosunku do swego poprzednika!”. Rzeczywiście, to prawda. Wskaźniki są to dziedziną języka, która najczęściej zmusza nas do podróży w przeszłość. Wbrew pozorom nie jest to jednak przeszłość zbyt odległa, skoro z powodzeniem wpływa na teraźniejszość. Z właściwości wskaźników i tablic będziesz bowiem korzystał znacznie częściej niż sporadycznie.

Tablice jednowymiarowe w pamięci

Swego czasu powiedzieliśmy sobie, że tablice są zespołem wielu zmiennych opatrzonych tą samą nazwą i identyfikowanych poprzez indeksy. Symbolicznie przedstawialiśmy na diagramach tablice jednowymiarowe jako równy rząd prostokątów, wyobrażających kolejne elementy.
To nie był wcale przypadek. Tablice takie mają bowiem ważną cechę:

Kolejne elementy tablicy jednowymiarowej są ułożone **obok siebie**, w **ciągłym obszarze** pamięci.

Nie są więc porozrzucane po całej dostępnej pamięci (czyli pofragmentowane), ale grzecznie zgrupowane w jeden pakiet.



Schemat 34. Ułożenie tablicy jednowymiarowej w pamięci operacyjnej

Dzięki temu kompilator nie musi sobie przechowywać adresów każdego z elementów tablicy, aby programista mógł się do nich odwoływać. Wystarczy tylko jeden: adres **początku tablicy**, jej zerowego elementu.
W kodzie można go łatwo uzyskać w ten sposób:

```
// tablica i wskaźnik
int aTablica[5];
int* pnTablica;

// pobranie wskaźnika na zerowy element tablicy
pnTablica = &aTablica[0];
```

Napisałem, że jest to także adres początku samej tablicy, czyli w gruncie rzeczy wartość kluczowa dla całego agregatu. Dlatego reprezentuje go również **nazwa** tablicy:

```
// inny sposób pobrania wskaźnika na zerowy element (początek) tablicy
pnTablica = aTablica;
```

Wynika stąd, iż:

Nazwa tablicy jest także stałym **wskaźnikiem** do jej **zerowego elementu** (**początku**).

Stałym - bo jego adres jest nadany raz na zawsze przez kompilator i nie może być zmieniany w programie.

Wskaźnik w ruchu

Posiadając wskaźnik do jednego z elementów tablicy, możemy z łatwością dostać się do pozostałych - wykorzystując fakt, iż tablica jest ciągłym obszarem pamięci. Można mianowicie odpowiednio przesunąć nasz wskaźnik, np.:

```
pnTablica += 3;
```

Po tej operacji będzie on pokazywał na **3 elementy dalej** niż dotychczas. Ponieważ na początku wskazywał na początek tablicy (zerowy element), więc teraz zacznie odnosić się do jej trzeciego elementu.
To ciekawe zjawisko. Wskaźnik jest przecież adresem, liczbą, zatem dodanie do niego jakiejś liczby powinno skutkować odpowiednim zwiększeniem przechowywanej wartości. Ponieważ kolejne adresy w pamięci są numerami bajtów, więc `pnTablica` powinien, zdawałoby się, przechowywać adres trzeciego **bajta**, licząc od początku tablicy.
Tak jednak nie jest, gdyż kompilator podczas dokonywania arytmetyki na wskaźnikach korzysta także z informacji o ich **typie**. „Skoki” spowodowane dodawaniem liczb całkowitych następują w odstępach bajtowych równych **wielokrotnościom rozmiaru**

zmiennej, na jaką wskazuje wskaźnik. W naszym przypadku pnTablica przesuwa się więc o 3*sizeof(int) bajtów, a nie o 3 bajty!

Obecnie wskazuje zatem na trzeci element tablicy aTablica. Dokonując dereferencji wskaźnika, możemy odwołać się do tego elementu:

```
// obie poniższe linijki są równoważne
*pnTablica = 0;
aTablica[3] = 0;
```

Wreszcie, dozwolony jest także trzeci sposób:

```
*(aTablica + 3) = 0;
```

Używamy w nim wskaźnikowych właściwości nazwy tablicy. Wyrażenie aTablica + 3 odnosi się zatem do jej trzeciego elementu. Jego dereferencja pozwala przypisać temu elementowi jakąś wartość.

Wydało się więc, że do *i*-tego elementu *tablicy* można odwołać się na dwa różne sposoby:

```
*(tablica + i)
tablica[i]
```

W praktyce kompilator sam stosuje tylko pierwszy. Wprowadzenie drugiego miało oczywiście głęboki sens: jest on zwyczajnie prostszy, nie tylko w zapisie, ale i w zrozumieniu. Nie wymaga też żadnej wiedzy o wskaźnikach, a ponadto daje większą elastyczność przy definiowaniu własnych typów danych.

Nie należy jednak zapominać, że oba sposoby są tak samo podatne na błąd przekroczenia indeksów, który występuje, gdy i wykracza poza przedział <0; *rozmiar_tablicy* - 1>.

Tablice wielowymiarowe w pamięci

Dla tablic wielowymiarowych sprawa ich rozmieszczenia w pamięci jest nieco bardziej skomplikowana. W przeciwieństwe do pamięci nie mają one bowiem struktury liniowej, zatem kompilator ją jakoś **symulować** (czyli **linearyzować** tablicę).
Nie jest to specjalnie trudna czynność, ale praktyczny sens jej omawiania jest raczej wątpliwy. Z tego względu mało kto stosuje wskaźniki do pracy z wielowymiarowymi tablicami, zaś my nie będziemy tutaj żadnym wyjątkiem od reguły :)

Zainteresowanym mogę wyjaśnić, że wymiary tablicy są układane w pamięci według kolejności ich zadeklarowania w kodzie, od lewej do prawej. Posuwając się wzdłuż takiej zlinearyzowanej tablicy najszybciej zmienia się więc ostatni indeks, wolniej przedostatni, i tak dalej.
Formułka matematyczna służąca do obliczania wskaźnika na element wielowymiarowej tablicy jest natomiast podana w MSDN.

Łańcuchy znaków w stylu C

Kiedy już omawiamy wskaźniki w odniesieniu do tablic, nadarza się niepowtarzalna okazja, aby zapoznać się także z łańcuchami znaków w języku C - poprzedniku C++.

Po co? Otóż jak dotąd jest to najczęściej wykorzystywana forma wymiany tekstu między aplikacjami oraz bibliotekami. Do koronnych przykładów należy choćby Windows API, której obsługi przecież będziemy się w przyszłości uczyć.

Od razu spotka nas tutaj pewna niespodzianka. O ile bowiem C++ posiada wygodny typ `std::string`, służący do przechowywania napisów, to C w ogóle takiego typu nie posiada! Zwyczajnie nie istnieje żaden specjalny typ danych, służący reprezentacji tekstu.
Zamiast niego stosowanie jest inne podejście do problemu. Napis jest to **ciąg znaków**, a więc uporządkowany zbiór kodów ANSI, opisujących te znaki. Dla pojedynczego znaku istnieje zaś typ `char`, zatem ich ciąg może być przedstawiany jako odpowiednia **tablica**.

Łańcuch znaków w stylu C to jednowymiarowa **tablica** elementów typu `char`.

Różni się ona jednak on innych tablic. Są one przeznaczone głównie do pracy nad ich pojedynczymi elementami, natomiast łańcuch znaków jest częściej przetwarzany w całości, niż znak po znaku.
Sprawia to, że dozwolone są na przykład takie (w gruncie rzeczy trywialne!) operacje:

```
char szNapis[256] = "To jest jakiś tekst";
```

Manipulujemy w nich **więcej niż jednym** elementem tablicy naraz.
Zauważmy jeszcze, że przypisywany ciąg jest krótszy niż rozmiar tablicy (`256`). Aby zaznaczyć, gdzie się on kończy, kompilator dodaje zawsze jeszcze jeden, specjalny znak o kodzie 0, na samym końcu napisu. Z powodu tej właściwości łańcuchy znaków w stylu C są często nazywane **napisami zakończonymi zerem** (ang. *null-terminated strings*).

Dlaczego jednak ten sposób postępowania z tekstem jest zły (został przecież zastąpiony przez typ `std::string`)?...
Pierwszą przyczyną są problemy ze **zmienną długością** napisów. Tekst jest kłopotliwym rodzajem danych, który może zajmować bardzo różną ilość pamięci, zależnie od liczby znaków. Rozsądnym rozwiązaniem jest oczywiście przydzielanie mu dokładnie tylu bajtów, ilu wymaga; do tego potrzebujemy jednak mechanizmów zarządzania pamięcią w czasie działania programu (poznamy je zresztą w tym rozdziale). Można też statycznie rezerwować więcej miejsca, niż to jest potrzebne - tak zrobiłem choćby w poprzednim skrawku przykładowego kodu. Wada tego rozwiązania jest oczywista: spora część pamięci zwyczajnie się marnuje.
Drugą niedogodnością są utrudnienia w dokonywaniu najprostszych w zasadzie operacji na tak potraktowanych napisach. Chodzi tu na przykład o konkatenację; wiedząc, jak proste jest to dla napisów typu `std::string`, pewnie bez wahania napisalibyśmy coś w tym rodzaju:

```
char szImie[] = "Max";
char szNazwisko[] = "Planck";

char szImieINazwisko[] = szImie + " " + szNazwisko;      // BŁĄD!
```

Visual C++ zareagowałby zaś takim oto błędem:

```
error C2110: '+': cannot add two pointers
```

Miałby w nim całkowitą słuszność. Rzeczywiście, próbujemy tutaj dodać do siebie dwa wskaźniki, co jest niedozwolone i pozbawione sensu. Gdzie są jednak te wskaźniki?...
To przede wszystkim `szImie` i `szNazwisko` - jako nazwy tablic są przecież wskaźnikami do swych zerowych elementów. Również spacja `" "` jest przez kompilator traktowana jako wskaźnik, podobnie zresztą jak wszystkie napisy wpisane w kodzie *explicité*.

Porównywanie takich napisów poprzez operator `==` jest więc **niepoprawne**!

Łączenie napisów w stulu C jest naturalnie możliwe, wymaga jednak użycia specjalnych funkcji w rodzaju `strcat()`. Inne funkcje są przeznaczone choćby do przypisywania napisów (`str[n]cpy()`) czy pobierania ich długości (`strlen()`). Nietrudno się domyśleć, że korzystanie z nich nie należy do rzeczy przyjemnych :)

Na całe szczęście ominie nas ta „rozkosz”. Standardowy typ `std::string` zawiera bowiem wszystko, co jest niezbędne do programowej obsługi łańcuchów znaków. Co więcej, zapewnia on także kompatybilnośc z dawnymi rozwiązaniami.
Metoda `c_str()` (skrót od *C string*), bo o nią tutaj chodzi, zwraca wskaźnik typu `const char*`, którego można użyć wszędzie tam, gdzie wymagany jest napis w stylu C. Nie musimy przy tym martwić się o późniejsze zwolnienie zajmowanej przez nasz tekst pamięci - zadba oto sama Biblioteka Standardowa.
Przykładem wykorzystania tego rozwiązania może być wyświetlenie okna komunikatu przy pomocy funkcji `MessageBox()` z Windows API:

```
#include <string>
#include <windows.h>

std::string strKomunikat = "Przykładowy komunikat";
strKomunikat += ".";

MessageBox (NULL, strKomunikat.c_str(), "Komunikat", MB_OK);
```

O samej funkcji `MessageBox()` powiemy sobie wszystko, gdy już przejdziemy do programowania aplikacji okienkowych. Powyższy kod zadziała jednak także w programie konsolowym.

Drugi oraz trzeci parametr tej funkcji powinien być łańcuchem znaków w stylu C. Możemy więc skorzystać z metody `c_str()` dla zmiennej `strKomunikat`, by uczynić zadość temu wymaganiu. W sumie więc nie przeszkadza ono zupełnie w normalnym korzystaniu z dobrodziejstw standardowego typu `std::string`.

Przekazywanie wskaźników do funkcji

Jedną z ważniejszych płaszczyzn zastosowań wskaźników jest usprawnienie korzystania z funkcji. Wskaźniki umożliwiają osiągnięcie kilku niespotykanych dotąd możliwości i optymalizacji.

Dane otrzymywane poprzez parametry

Wskaźnik jest odwołaniem do zmiennej („kluczem” do niej), które ma jedną zasadniczą zaletę: może mianowicie być przekazywane gdziekolwiek i nadal zachowywać swoją podstawową rolę. Niezależnie od tego, w którym miejscu programu użyjemy wskaźnika, będzie on nadal wskazywał na ten sam adres w pamięci, czyli na tą samą zmienną.

Jeżeli więc przekażemy wskaźnik do funkcji, wtedy będzie ona mogła operować na jego docelowej komórce pamięci. W ten sposób możemy na przykład sprawić, aby funkcja zwracała **więcej niż jedną wartość** w wyniku swego działania.
Spójrzmy na prosty przykład takiego zachowania:

```
// funkcja oblicza całkowity iloraz dwóch liczb oraz jego resztę
int Podziel(int nDzielna, int nDzielnik, int* const pnReszta)
{
   // zapisujemy resztę w miejscu pamięci, na które pokazuje wskaźnik
   *pnReszta = nDzielna % nDzielnik;

   // zwracamy iloraz
   return nDzielna / nDzielnik;
```

```
}
```

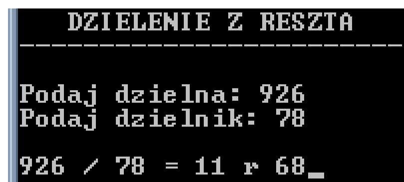
Ta prosta funkcja dzielenia całkowitego zwraca dwa rezultaty. Pierwszy to zasadniczy iloraz - jest on oddawany w tradycyjny sposób poprzez `return`. Natomiast reszta z dzielenia jest przekazywana poprzez stały wskaźnik `pReszta`, który funkcja otrzymuje jako parametr. Dokonuje jego dereferencji i zapisuje żądaną wartość w miejscu, na które on wskazuje.
Jeżeli pamiętamy o tym, skorzystanie z powyższej funkcji jest raczej proste i przedstawia się mniej więcej tak:

```
// Division - dzielenie przy użyciu wskaźnika przekazywanego do funkcji

void main()
{
    // (pominiemy pobranie dzielnej i dzielnika od użytkownika)

    // obliczenie rezultatu
    int nIloraz, nReszta;
    nIloraz = Podziel(nDzielna, nDzielnik, &nReszta);

    // wyświetlenie rezultatu
    std::cout << std::endl;
    std::cout << nDzielna << " / " <<nDzielnik << " = "
              << nIloraz << " r " << nReszta;
    getch();
}
```

Jako trzeci parametr w wywołaniu funkcji `Podziel()`:

```
nIloraz = Podziel(nDzielna, nDzielnik, &nReszta);
```

przekazujemy adres zmiennej (uzyskany oczywiście poprzez operator `&`). W niej też znajdziemy potem żądaną resztę i wyświetlimy ją w oknie konsoli:



Screen 42. Dwie wartości zwracane przez jedną funkcję

W podobny sposób działa wiele funkcji z Windows API czy DirectX. Zaletą tego rozwiązania jest także możliwość oddzielenia zasadniczego wyniku funkcji (zwracanego przez wskaźnik) od ewentualnej informacji o błędzie czy też sukcesie jego uzyskania (przekazywanego w tradycyjny sposób).

Oczywiście nic nie stoi na przeszkodzie, aby tą drogą zwracać więcej niż jeden „dodatkowy" rezultat funkcji. Jeśli jednak ich liczba jest znaczna, lepiej złączyć je w strukturę niż deklarować po kilkanaście parametrów w nagłówku funkcji.

Zapobiegamy niepotrzebnemu kopiowaniu

Oprócz otrzymywania kilku wyników z jednej funkcji, zastosowanie wskaźników może mieć też podłoże optymalizacyjne. Pomyślmy, że taki wskaźnik to zawsze jest tylko zwykła liczba całkowita, zajmująca zaledwie 4 bajty w pamięci. Jednocześnie jednak może ona odnosić się do bardzo wielkich obiektów.

Kiedy zaś wywołujemy funkcję z parametrami, wówczas kompilator dokonuje ich całościowego **kopiowania** - tak, że w ciele funkcji mamy do czynienia z **duplikatami** rzeczywistych parametrów aktualnych funkcji. Mówiliśmy zresztą we właściwym czasie, iż parametry pełnią w funkcji rolę **dodatkowych zmiennych lokalnych**.
Aby to zilustrować, weźmy taką oto banalną funkcję:

```
int Dodaj(int nA, int nB)
{
   nA += nB;
   return nA;
}
```

Jak widać, dokonujemy w niej modyfikacji jednego z parametrów. Kiedy jednak wywołamy niniejszą funkcję w sposób podobny do tego:

```
int nLiczba1 = 1, nLiczba2 = 2;
std::cout << Dodaj(nLiczba1, nLiczba2);
std::cout << nLiczba1;     // nadal nLiczba1 == 1 !
```

zobaczymy, że podana jej zmienna pozostaje **nietknięta**. Funkcja otrzymała bowiem tylko jej wartość, która została w tym celu **skopiowana**.

Trzeba jednak przyznać, że większość funkcji z założenia nie modyfikuje swoich parametrów, a jedynie odczytuje z nich wartości. W takim przypadku jest im więc „wszystko jedno”, czy odwołują się do faktycznie istniejących zmiennych, czy też do ich kopii, istniejących tylko podczas działania funkcji.
Jednak nam, programistom, nie jest wszystko jedno. Stworzenie kopii zmiennych wymaga bowiem **dodatkowego czasu** - na przydzielenie odpowiedniej ilości pamięci i zapisanie w niej pożądanej wartości. Naturalnie, w przypadku typów liczbowych jest to pomijalnie mały interwał, ale dla większych obiektów (chociażby łańcuchów znaków) może stać się znaczący. A przecież wcale nie musi tak być!

Możliwe jest zlikwidowanie konieczności tworzenia duplikatów zmiennych dla wywoływanych funkcji: wystarczy tylko zamiast wartości przekazywać **odwołania** do nich, czyli… wskaźniki! Skopiowanie czterech bajtów będzie na pewno znacznie szybsze niż przemieszczanie ilości danych liczonej na przykład w dziesiątkach kilobajtów.
Zobaczmy więc, jak można przyspieszyć działanie funkcji operujących na dużych obiektach. Posłużę się tu przykładem na wyszukiwanie pozycji jednego ciągu znaków wewnątrz innego:

```
#include <string>

// funkcja przeszukuje drugi napis w poszukiwaniu pierwszego;
// gdy go znajdzie, zwraca indeks pierwszego pasującego znaku,
// w przeciwnym wypadku wartość -1
int Wyszukaj  (const std::string* pstrSzukany,
               const std::string* pstrPrzeszukiwany)
{
   // przeszukujemy nasz napis
   for (unsigned i = 0;
        i <= pstrPrzeszukiwany->length() - pstrSzukany->length(); ++i)
   {
         // porównujemy kolejne wycinki napisu (o odpowiedniej długości)
         // z poszukiwanym łańcuchem. Metoda std::string::substr() służy
         // do pobierania wycinka napisu
         if (pstrPrzeszukiwany->substr(i, pstrSzukany->length())
             == *pstrSzukany)
               // jeżeli wycinek zgadza się, to zwracamy jego indeks
               return i;
```

```
    }

    // w razie niepowodzenia zwracamy -1
    return -1;
}
```

Przeszukiwany tekst może być bardzo długi - edytory pozwalają na przykład na poszukiwanie wybranej frazy wewnątrz całego dokumentu, liczącego nieraz wiele kilobajtów. Nie jest to jednak problemem: dzięki temu, że funkcja operuje na nim poprzez wskaźnik, pozostaje on cały czas „na swoim miejscu” w pamięci i **nie jest kopiowany**. Zysk na wydajność aplikacji może być wtedy znaczny.

W zamian jednakże doświadczamy pewnej niedogodności, związanej ze składnią działań na wskaźnikach. Aby odwołać się do przekazanego napisu, musimy każdorazowo dokonywać jego dereferencji; także wywoływanie metod wymaga innego operatora niż kropka, do której przyzwyczailiśmy się, operując na napisach.
Ale i na to jest rada. Na koniec podrozdziału poznamy bowiem referencje, które zachowują cechy wskaźników przy jednoczesnym umożliwieniu stosowania zwykłej składni, właściwej zmiennym.

Dynamiczna alokacja pamięci

Kto wie, czy nie najważniejszym polem do popisu dla wskaźników jest zawłaszczanie nowej pamięci w trakcie działania programu. Mechanizm ten daje nieosiągalną inaczej elastyczność aplikacji i pozwala manipulować danymi o **zmiennej wielkości**. Bez niego wszystkie programy miałyby z góry narzucone limity na ilość przetwarzanych informacji, których nijak nie można by przekroczyć.
Koniecznie więc musimy przyjrzeć się temu zjawisku.

Przydzielanie pamięci dla zmiennych

Wszystkie zmienne deklarowane w kodzie mają **statycznie przydzieloną** pamięć o stałym rozmiarze. Rezydują one w obszarze pamięci zwanym **stosem**, który również ma niezmienną wielkość. Stosując wyłącznie takie zmienne, nie możemy więc przetwarzać danych cechujących się dużą rozpiętością zajmowanego miejsca w pamięci.

Oprócz stosu istnieje wszak także **sterta**. Jest to reszta pamięci operacyjnej, niewykorzystana przez program w momencie jego uruchomienia, ale stanowiąca rezerwę na przyszłość. Aplikacja może zeń czerpać potrzebną w danej chwili ilość pamięci (nazywamy to **alokacją**), wypełniać własnymi danymi i pracować na nich, a po zakończeniu roboty zwyczajnie oddać ją z powrotem (**zwolnić**) do wspólnej puli.
Najważniejsze, że o ilości niezbędnego miejsca można zdecydować **w trakcie działania programu**, np. obliczyć ją na podstawie liczb pobranych od użytkownika czy też z jakiegokolwiek innego źródła. Nie jesteśmy więc skazani na stały rozmiar stosu, lecz możemy **dynamicznie przydzielać** sobie ze sterty tyle pamięci, ile akurat potrzebujemy. Zbiory informacji o niestałej wielkości stają się wtedy możliwe do opanowania.

Alokacja przy pomocy `new`

Całe to dobrodziejstwo jest ściśle związane z wskaźnikami, gdyż to właśnie za ich pomocą uzyskujemy nową pamięć, odwołujemy się do niej i wreszcie zwalniamy ją po skończonej pracy.

Wszystkie te czynności prześledzimy na prostym przykładzie. Weźmy więc sobie zwyczajny wskaźnik na typ `int`:

```
int* pnLiczba;
```

Chwilowo nie pokazuje on na żadne sensowne dane. Moglibyśmy oczywiście złączyć go z jakąś zmienną zadeklarowaną w kodzie (poprzez operator `&`), lecz nie o to nam teraz chodzi. Chcemy sobie sami takową zmienną **stworzyć** - używamy do tego operatora `new` ('nowy') oraz nazwy typu tworzonej zmiennej:

```
pnLiczba = new int;
```

Wynikiem działania tego operatora jest **adres**, pod którym widnieje w pamięci nasza świeżo stworzona, nowiutka zmienna. Umieszczamy go zatem w przygotowanym wskaźniku - odtąd będzie on służył nam do manipulowania wykreowaną zmienną.

Cóż takiego różni ją innych, deklarowanych w kodzie? Ano całkiem sporo rzeczy:

- nie ma ona **nazwy**, poprzez którą moglibyśmy się do niej odwływać. Wszelka „komunikacja" z nią musi zatem odbywać się za pośrednictwem wskaźnika, w którym zapisaliśmy adres zmiennej.
- **czasu istnienia** zmiennej nie kontroluje kompilator, ale sam programista. Inaczej mówiąc, nasza zmienna istnieje aż do momentu jej zwolnienia (poprzez operator `delete`, który omówimy za chwilę). Wynika stąd również, że dla takiej zmiennej nie ma sensu pojęcie zasięgu.
- **początkowa wartość** zmiennej jest przypadkowa. Zależy bowiem od tego, co poprzednio znajdowało się w tym miejscu pamięci, które teraz system operacyjny oddał do dyspozycji naszego programu.

Poza tymi aspektami, możemy na tak stworzonej zmiennej wykonywać te same operacje, co na wszystkich innych zmiennych tego typu. Dereferując pokazujący nań wskaźnik, otrzymujemy pełen dostęp do niej:

```
*pnLiczba = 100;
*pnLiczba += rand();
std::cout << *pnLiczba;
// itp.
```

Oczywiście nasze możliwości nie ograniczają się tylko do typów liczbowych czy podstawowych. Przeciwnie, za pomocą `new` możemy alokować pamięć dla dowolnych rodzajów zmiennych - także tych definiowanych przez nas samych.
Widzimy więc, że to bardzo potężne narzędzie.

Zwalnianie pamięci przy pomocy `delete`

Z każdej potęgi trzeba jednak korzystać z rozwagą. W przypadku dynamicznej alokacji zasada BHP brzmi:

Zawsze zwalniaj zaalokowaną przez siebie pamięć.

Służy do tego odrębny operator `delete` ('usuń'). Użycie go jest nadzwyczaj łatwe: wystarczy jedynie podać mu wskaźnik na przydzielony obszar pamięci, a on posłusznie posprząta po nim i zwróci go do dyspozycji systemu operacyjnego, a więc i wszystkich pozostałych programów.

Bez zwolnienia pamięci operacyjnej następuje jej **wyciek** (ang. *memory leak*). Zaalokowana, a niezwolniona pamięć nie jest już bowiem dostępna dla innych aplikacji.

Po skończeniu pracy z naszą dynamicznie stworzoną zmienną musimy ją zatem usunąć. Wygląda to następująco:

```
delete pnLiczba;
```

Należy mieć świadomość, że `delete` niczego nie modyfikuje w samym wskaźniku, zatem nadal pokazuje on na ten sam obszar pamięci. Teraz jednak nasz program nie jest już jego właścicielem, dlatego też aby uniknąć omyłkowego odwołania się do nieswojego rejonu pamięci, wypadałoby wyzerować nasz wskaźnik:

```
pnLiczba = NULL;
```

Wartość `NULL` to po prostu **zero**, zaś zerowy adres nie istnieje. `pnLiczba` staje się więc **wskaźnikiem pustym**, niepokazującym na żadną konkretną komórkę pamięci. Gdybyśmy teraz (omyłkowo) spróbowali ponownie zastosować wobec niego operator `delete`, wtedy instrukcja ta zostałaby po prostu zignorowana. Jeżeli jednak wskaźnik nadal pokazywałby na **już zwolniony** obszar pamięci, wówczas bez wątpienia wystąpiłby **błąd ochrony pamięci** (ang. *access violation*).

Zatem pamiętaj, aby dla bezpieczeństwa **zerować wskaźnik** po **zwolnieniu dynamicznej zmiennej**, na którą on wskazywał.

Nowe jest lepsze

Jeżeli czytają to jakieś osoby znające język C (w co wątpie, ale wyjątki zawsze się zdarzają :D), to pewnie nie darowałyby mi, gdybym nie wspomniał o sposobach na alokację i zwalnianie pamięci w tym języku. Chcą zapewne wiedzieć, dlaczego powinny o nich **zapomnieć** (a powinny!) i stosować wyłącznie `new` oraz `delete`.

Otóż w C mieliśmy dwie funkcje, `malloc()` i `free()`, służące odpowiednio do przydzielania obszaru pamięci o żądanej wielkości oraz do jego późniejszego zwalniania. Radziły sobie z tym zadaniem całkiem dobrze i mogłyby w zasadzie nadal sobie z nim radzić. W C++ doszły jednak nowe zadania związane z dynamiczną alokacją pamięci operacyjnej.
Chodzi tu naturalnie o kwestię klas z programowania obiektowego i związanymi z nimi **konstruktorami** i **destruktorami**. Kiedy używamy `new` i `delete` do tworzenia i niszczenia obiektów, w poprawny sposób wywołują one te specjalne metody. Funkcje znane z C **nie robią tego**; nie ma w tym jednak niczego dziwnego, bo w ich macierzystym języku w ogóle nie istniało pojęcie klasy czy obiektu, nie mówiąc już o metodach uruchamianych podczas ich tworzenia i niszczenia.

„Nowy" sposób alokacji ma jeszcze jedną zaletę. Otóż `malloc()` zwraca w wyniku wskaźnik ogólny, typu `void*`, zamiast wskaźnika na określony typ danych. Aby przypisać go do wybranej zmiennej wskaźnikowej, należało użyć rzutowania.
Przy korzystaniu z `new` nie jest to konieczne. Za pomocą tego operatora od razu uzyskujemy właściwy typ wskaźnika i nie musimy stosować żadnych konwersji.

Dynamiczne tablice

Alokacja pamięci dla pojedynczej zmiennej jest wprawdzie poprawna i klarowna, ale raczej mało efektowna. Trudno wówczas powiedzieć, że faktycznie operujemy na zbiorze danych o niejednostajnej wielkości, skoro owa niestałość objawia się jedynie… obecnością lub nieobecnością jednej zmiennej!

O wiele bardziej interesują są **dynamiczne tablice** - takie, których rozmiar jest ustalany w czasie działania aplikacji. Mogą one przechowywać różną ilość elementów, więc nadają się do mnóstwa wspaniałych celów :)
Zobaczymy teraz, jak obsługiwać takie tablice.

Tablice jednowymiarowe

Najprościej sprawa wygląda z takimi tablicami, których elementy są indeksowane jedną liczbą, czyli po prostu z tablicami jednowymiarowymi. Popatrzmy zatem, jak odbywa się ich alokacja i zwalnianie.

Tradycyjnie już zaczynamy od odpowiedniego wskaźnika. Jego typ będzie determinował rodzaj danych, jakie możemy przechowywać w naszej tablicy:

```
float* pfTablica;
```

Alokacja pamięci dla niej także przebiega w dziwnie znajomy sposób. Jedyną różnicą w stosunku do poprzedniego paragrafu jest oczywista konieczność podania **wielkości tablicy**:

```
pfTablica = new float [1024];
```

Podajemy ją w nawiasach klamrowych, za nazwą typu pojedynczego elementu. Z powodu obecności tych nawiasów, występujący tutaj operator jest często określony jako `new[]`. Ma to szczególny sens, jeżeli porównamy go z operatorem zwalniania tablicy, który zobaczymy za momencik.

Zważmy jeszcze, że rozmiar naszej tablicy jest dosyć spory. Być może wobec dzisiejszych pojemności RAMu brzmi to zabawnie, ale zawsze przecież istnieje potencjalna możliwość, że zabraknie dla nas tego życiodajnego zasobu, jakim jest pamięć operacyjna. I na takie sytuacje powinniśmy być przygotowani - tym bardziej, że poczynienie odpowiednich kroków nie jest trudne.
W przypadku **braku pamięci** operator `new` zwróci nam **pusty wskaźnik**; jak pamiętamy, nie odnosi się on do żadnej komórki, więc może być użyty jako wartość kontrolna (spotkaliśmy się już z tym przy okazji rzutowania `dynamic_cast`). Wypadałoby zatem sprawdzić, czy nie natrafiliśmy na taką nieprzyjemną sytuację i zareagować na nią odpowiednio:

```
if (pfTablica == NULL)      // może być też if (!pfTablica)
   std::cout << "Niestety, zabraklo pamieci!";
```

Możemy zmienić to zachowanie i sprawić, żeby w razie niepowodzenia alokacji pamięci była wywoływana nasza własna funkcja. Po szczegóły możesz zajrzeć do opisu funkcji `set_new_handler()` w MSDN.

Jeżeli jednak wszystko poszło dobrze - a tak chyba będzie najczęściej :) - możemy używać naszej tablicy **w identyczny sposób**, jak tych alokowanych statycznie. Powiedzmy, że wypełnimy ją treścią przy pomocy następującej pętli:

```
for (unsigned i = 0; i < 1024; ++i)
   pfTablica[i] = i * 0.01;
```

Widać, że dostęp do poszczególnych elementów odbywa się tutaj tak samo, jak dla tablic o stałym rozmiarze. A właściwie, żeby być ścisłym, to raczej tablice o stałym rozmiarze zachowują się podobnie, gdyż w obu przypadkach mamy do czynienia z jednym i tym samym mechanizmem - wskaźnikami.
Należy jeszcze pamiętać, aby **zachować gdzieś rozmiar** alokowanej tablicy, żeby móc na przykład przetwarzać ją przy pomocy pętli `for`, podobnej do powyższej.

Na koniec trzeba oczywiście zwolnić pamięć, która przeznaczyliśmy na tablicę. Za jej usunięcie odpowiada operator `delete[]`:

```
delete[] pfTablica;
```

Musimy koniecznie uważać, aby nie pomylić go z podobnym operatorem `delete`. Tamten służy do zwalniania **wyłącznie pojedynczych zmiennych**, zaś jedynie niniejszy może być użyty do usunięcia tablicy. Nierespektowanie tej reguły może prowadzić do bardzo nieprzyjemnych błędów!

Zatem do **zwalniania tablic** korzystaj tylko z **operatora `delete[]`**!

Łatwo zapamiętać tę zasadę, jeżeli przypomnimy sobie, iż do alokowania tablicy posłużyła nam instrukcja `new[]`. Jej usunięcie musi więc również odbywać się przy pomocy operatora z nawiasami kwadratowymi.

Opakowanie w klasę

Jeśli często korzystamy z dynamicznych tablic, warto stworzyć dlań odpowiednią klasę, która ułatwi nam to zadanie. Nie jest to specjalnie trudne.
My stworzymy tutaj przykładową klasę jednowymiarowej tablicy elementów typu `int`.

Zacznijmy może od jej prywatnych pól. Oprócz oczywistego wskaźnika na wewnętrzną tablicę klasa powinna być wyposażona także w zmienną, w której **zapamiętamy rozmiar** utworzonej tablicy. Uwolnimy wtedy użytkownika od konieczności zapisywania jej we własnym zakresie.
Metody muszą zapewnić dostęp do elementów tablicy, a więc **pobieranie wartości** o określonym indeksie oraz **zapisywanie nowych liczb** w określonych elementach tablicy. Przy okazji możemy też **kontrolować indeksy** i zapobiegać ich przekroczeniu, co znowu zapewni nam dozgonną wdzięczność programisty-klienta naszej klasy ;)

Definicja takiej tablicy może więc przedstawiać się następująco:

```
class CIntArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         int* m_pnTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         CIntArray()                      // domyślny
               { m_uRozmiar = DOMYSLNY_ROZMIAR;
                 m_pnTablica = new int [m_uRozmiar]; }
         CIntArray(unsigned uRozmiar)  // z podaniem rozmiaru tablicy
               { m_uRozmiar = uRozmiar;
                 m_pnTablica = new int [m_uRozmiar]; }

         // destruktor
         ~CIntArray()      { delete[] m_pnTablica; }

         //-------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         int Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pnTablica[uIndeks];
                 else                        return 0;                  }
         bool Ustaw(unsigned uIndeks, int nWartosc)
```

```
                  { if (uIndeks >= m_uRozmiar) return false;
                    m_pnTablica[uIndeks] = uWartosc;
                    return true;                                }

            // inne
            unsigned Rozmiar() const      { return m_uRozmiar; }
};
```

Są w niej wszystkie detale, o jakich wspomniałem wcześniej.
Dwa konstruktory mają na celu zaalokowanie pamięci na naszą tablicę; jeden z nich jest domyślny i ustawia określoną z góry wielkość (wpisaną jako stała `DOMYSLNY_ROZMIAR`), drugi zaś pozwala podać ją jako parametr. Destruktor natomiast dba o zwolnienie tak przydzielonej pamięci. W tego typu klasach metoda ta jest więc szczególnie przydatna.
Pozostałe funkcje składowe zapewniają intuicyjny dostęp do elementów tablicy, zabezpieczając przy okazji przed błędem przekroczenia indeksów. W takiej sytuacji `Pobierz()` zwraca wartość zero, zaś `Ustaw()` - `false`, informując o zainstniałym niepowodzeniu.

Skorzystanie z tej gotowej klasy nie jest chyba trudne, gdyż jej definicja niemal dokumentuje się sama. Popatrzmy aczkolwiek na następujący przykład:

```
#include <cstdlib>
#include <ctime>

srand (static_cast<unsigned>(time(NULL)));
CIntArray aTablica(rand());

for (unsigned i = 0; i < aTablica.Rozmiar(); ++i)
   aTablica.Ustaw (i, rand());
```

Jak widać, generujemy w nim losową ilość losowych liczb :) Nieodmiennie też używamy do tego pętli `for`, nieodzownej przy pracy z tablicami.

Zdefiniowana przed momentem klasa jest więc całkiem przydatna, posiada jednak trzy zasadnicze wady:

- raz ustalony rozmiar tablicy nie może już ulegać zmianie. Jego modyfikacja wymaga stworzenia nowej tablicy
- dostęp do poszczególnych elementów odbywa się za pomocą mało wygodnych metod zamiast zwyczajowych nawiasów kwadratowych
- typem przechowywanych elementów może być jedynie `int`

Na dwa ostatnie mankamenty znajdziemy radę, gdy już nauczymy się przeciążać operatory oraz korzystać z szablonów klas w języku C++.
Niemożność zmiany rozmiaru tablicy możemy jednak usunąć już teraz. Dodajmy więc jeszcze jedną metodę za to odpowiedzialną:

```
class CIntArray
{
   // (resztę wycięto)

   public:
         bool ZmienRozmiar(unsigned);
};
```

Wykona ona alokację nowego obszaru pamięci i przekopiuje do niego już istniejącą część tablicy. Następne zwolni ją, zaś cała klasa będzie odtąd operowała na nowym fragmencie pamięci.
Brzmi to dosyć tajemniczo, ale w gruncie rzeczy jest bardzo proste:

```
#include <memory.h>

bool CIntArray::ZmienRozmiar(unsigned uNowyRozmiar)
{
   // sprawdzamy, czy nowy rozmiar jest większy od starego
   if (!(uNowyRozmiar > m_uRozmiar)) return false;

   // alokujemy nową tablicę
   int* pnNowaTablica = new int [uNowyRozmiar];

   // kopiujemy doń starą tablicę i zwalniamy ją
   memcpy (pnNowaTablica, m_pnTablica, m_uRozmiar * sizeof(int));
   delete[] m_pnTablica;

   // "podczepiamy" nową tablicę do klasy i zapamiętujemy jej rozmiar
   m_pnTablica = pnNowaTablica;
   m_uRozmiar = uNowyRozmiar;

   // zwracamy pozytywny rezultat
   return true;
}
```

Wyjaśnienia wymaga chyba tylko funkcja `memcpy()`. Oto jej prototyp (zawarty w nagłówku *memory.h*, który dołączamy):

```
void* memcpy(void* dest, const void* src, size_t count);
```

Zgodnie z nazwą (ang. *memory copy* - kopiuj pamięć), funkcja ta służy do kopiowania danych z jednego obszaru pamięci do drugiego. Podajemy jej miejsce docelowe i źródłowe kopiowania oraz **ilość bajtów**, jaka ma być powielona.
Właśnie ze względu na bajtowe wymagania funkcji `memcpy()` używamy operatora `sizeof`, by pobrać wielkość typu `int` i pomnożyć go przez rozmiar (liczbę elementów) naszej tablicy. W ten sposób otrzymamy wielkość zajmowanego przez nią rejonu pamięci w bajtach i możemy go przekazać jako trzeci parametr dla funkcji kopiującej.

Pełna dokumentacja funkcji `memcpy()` jest oczywiście dostępna w MSDN.

Po rozszerzeniu nowa tablica będzie zawierała wszystkie elementy pochodzące ze starej oraz nowy obszar, możliwy do natychmiastowego wykorzystania.

Tablice wielowymiarowe

Uelastycznienie wielkości jest w C++ możliwe także dla tablic o większej liczbie wymiarów. Jak to zwykle w tym języku bywa, wszystko odbywa się analogicznie i intuicyjnie :D

Przypomnijmy, że tablice wielowymiarowe to takie tablice, których elementami są… inne tablice. Wiedząc zaś, iż mechanizm tablic jest w C++ zarządzany poprzez wskaźniki, dochodzimy do wniosku, że:

Dynamiczna tablica *n*-wymiarowa składa się ze wskaźników do tablic (*n-1*)-wymiarowych.

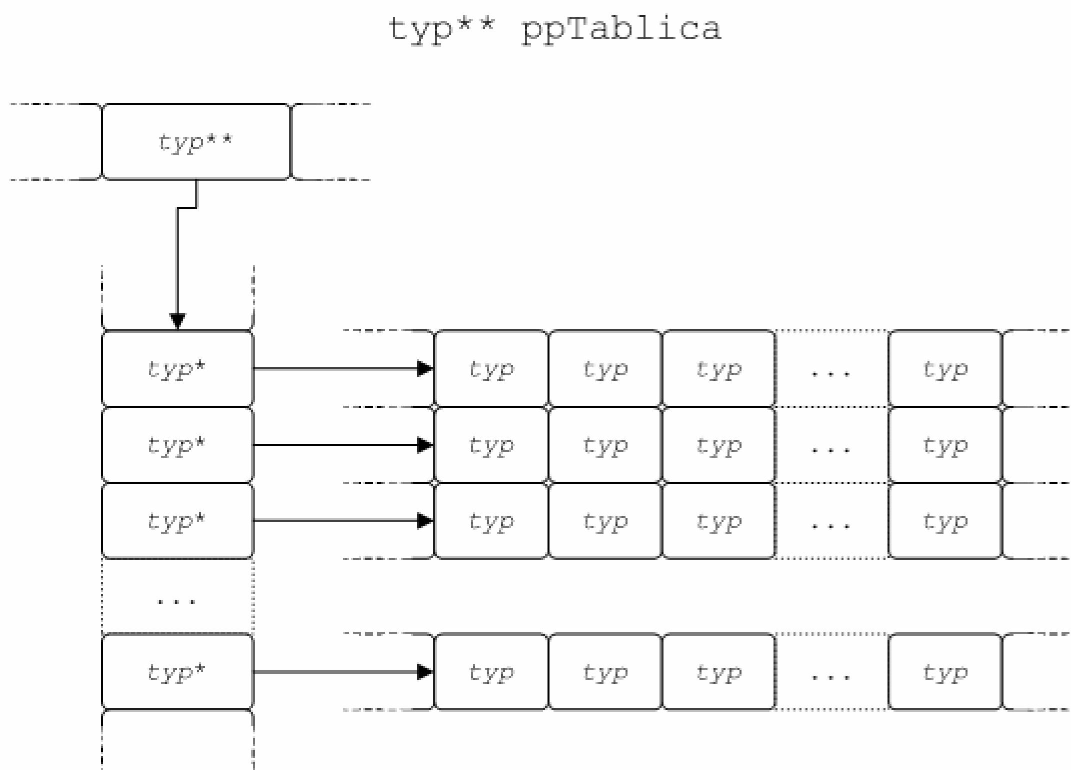
Dla przykładu, tablica o dwóch wymiarach jest tak naprawdę jednowymiarowym wektorem wskaźników, z których każdy pokazuje dopiero na jednowymiarową tablicę właściwych elementów.

Aby więc obsługiwać taką tablicę, musimy użyć dość osobliwej konstrukcji programistycznej - **wskaźnika na wskaźnik**. Nie jest to jednak takie dziwne. Wskaźnik to przecież też zmienna, a więc rezyduje pod jakimś adresem w pamięci. Ten adres może być przechowywany przez kolejny wskaźnik.
Deklaracja czegoś takiego nie jest trudna:

```
int** ppnTablica;
```

Wystarczy dodać po prostu kolejną gwiazdkę do nazwy typu, na który ostatecznie pokazuje nasz wskaźnik.

Jak taki wskaźnik ma się do dynamicznych, dwuwymiarowych tablic?... Ilustrując nim opis podany wcześniej, otrzymamy schemat podobny do tego:



Schemat 35. Dynamiczna tablica dwuwymiarowa jest tablicą wskaźników do tablic jednowymiarowych

Skoro więc wiemy już, do czego zmierzamy, pora osiągnąć cel.
Alokacja dwywumiarowej tablicy musi odbywać się dwuetapowo: najpierw przygotowujemy pamięć pod tablicę **wskaźników do jej wierszy**. Potem natomiast przydzielamy pamięć każdemu z tych wierszy - tak, że w sumie otrzymujemy tyle elementów, ile chcieliśmy.
Po przełożeniu na kod C++ algorytm wygląda w ten sposób:

```
// Alokacja tablicy 3 na 4

// najpierw tworzymy tablicę wskaźników do kolejnych wierszy
ppnTablica = new int* [3];

// następnie alokujemy te wiersze
for (unsigned i = 0; i < 3; ++i)
   ppnTablica[i] = new int [4];
```

Przeanalizuj go dokładnie. Zwróć uwagę szczególnie na linijkę:

```
ppnTablica[i] = new int [4];
```

Za pomocą wyrażenia `ppnTablica[i]` odwołujemy się tu do **`i`-tego wiersza** naszej tablicy - a ściślej mówiąc, do **wskaźnika** na niego. Przydzielamy mu następnie adres zaalokowanego fragmentu pamięci, który będzie pełnił rolę owego wiersza. Robimy tak po kolei ze wszystkimi wierszami tablicy.

Użytkowanie tak stworzonej tablicy dwuwymiarowej nie powinno nastręczać trudności. Odbywa się ono bowiem identycznie, jak w przypadku statycznych macierzy. Najczęstszą konstrukcją jest tu znowu zagnieżdżona pętla `for`:

```
for (unsigned i = 0; i < 3; ++i)
   for (unsigned j = 0; j < 4; ++j)
         ppnTablica[i][j] = i - j;
```

Co zaś ze zwalnianiem tablicy? Otóż przeprowadzamy je w sposób dokładnie **przeciwny** do jej alokacji. Zaczynamy od uwolnienia poszczególnych wierszy, a następnie pozbywamy się także samej tablicy wskaźników do nich.
Wygląda to mniej więcej tak:

```
// zwalniamy wiersze
for (unsigned i = 0; i < 3; ++i)
   delete[] ppnTablica[i];

// zwalniamy tablicę wskaźników do nich
delete[] ppnTablica;
```

Przedstawioną tu kolejność należy zawsze **bezwględnie zachowywać**. Gdybyśmy bowiem najpierw pozbyli się wskaźników do wierszy tablicy, wtedy nijak nie moglibyśmy zwolnić samych wierszy! Usuwanie tablicy „od tyłu" chroni zaś przed taką ewentualnością.

Znając technikę alokacji tablicy dwuwymiarowej, możemy łatwo rozszerzyć ją na większą liczbę wymiarów. Popatrzmy tylko na kod odpowiedni dla trójwymiarowej tablicy:

```
/* Dynamiczna tablica trójwymiarowa, 5 na 6 na 7 elementów */

// wskaźnik do niej ("trzeciego stopnia"!)
int*** p3nTablica;


/* alokacja */

// tworzymy tablicę wskaźników do 5 kolejnych "płaszczyzn" tablicy
p3nTablica = new int** [5];

// przydzielamy dla nich pamięć
for (unsigned i = 0; i < 5; ++i)
{
   // alokujemy tablicę na wskaźniki do wierszy
   p3nTablica[i] = new int* [6];

   // wreszcie, dla przydzielamy pamięć dla właściwych elementów
   for (unsigned j = 0; j < 6; ++j)
         p3nTablica[i][j] = new int [7];
}
```

```
/* użycie */

// wypełniamy tabelkę jakąś treścią
for (unsigned i = 0; i < 5; ++i)
   for (unsigned j = 0; j < 6; ++j)
         for (unsigned k = 0; k < 7; ++k)
               p3nTablica[i][j][k] = i + j + k;


/* zwolnienie */

// zwalniamy kolejne "płaszczyzny"
for (unsigned i = 0; i < 5; ++i)
{
   // zaczynamy jednak od zwolnienia wierszy
   for (unsigned j = 0; j < 6; ++j)
         delete[] p3nTablica[i][j];

   // usuwamy "płaszczyznę"
   delete[] p3nTablica[i];
}

// na koniec pozbywamy się wskaźników do "płaszczyzn"
delete[] p3nTablica;
```

Widać niestety, że z każdym kolejnym wymiarem kod odpowiedzialny za alokację oraz zwalnianie tablicy staje się coraz bardziej skomplikowany. Na szczęście jednak dynamiczne tablice o większej liczbie wymiarów są bardzo rzadko wykorzystywane w praktyce.

Referencje

Naocznie przekonałeś się, że domena zastosowań wskaźników jest niezwykle szeroka. Jeżeli nawet nie dałyby w danym programie jakichś niespotykanych możliwości, to na pewno za ich pomocą można poczynić spore optymalizacje w kodzie i przyspieszyć jego działanie.

Za poprawę wydajności trzeba jednak zapłacić wygodą: odwoływanie się do obiektów poprzez wskaźniki wymaga bowiem ich dereferencji. Wprowadza ona nieco zamieszania do kodu i wymaga poświęcenia mu większej uwagi. Cóż, zawsze coś za coś, prawda?... Otóż nieprawda :) Twórcy C++ wyposażyli bowiem swój język w mechanizm **referencji**, który łączy zalety wskaźników z normalną składnią zmiennych. Zatem i wilk jest syty, i owca cała.

Referencje (ang. *references*) to zmienne wskazujące na adresy miejsc w pamięci, ale pozwalające używać zwyczajnej składni przy odwoływaniu się do tychże miejsc.

Można je traktować jako pewien szczególny rodzaj wskaźników, ale stworzony dla czystej wygody programisty i poprawy wyglądu pisanego przezeń kodu. Referencje są aczkolwiek niezbędne przy przeciążaniu operatorów (o tym powiemy sobie niedługo), jednak swoje zastosowania mogą znaleźć niemal wszędzie.
Przy takiej rekomendacji trudno nie oprzeć się chęci ich poznania, nieprawdaż? ;) Tym właśnie zagadnieniem zajmiemy się więc teraz.

Typy referencyjne

Podobnie jak wskaźniki wprowadziły nam pojęcie typów wskaźnikowych, tak i referencje dodają do naszego słownika analogiczny termin **typów referencyjnych**.
W przeciwieństwie jednak do wskaźników, dla każdego normalnego typu istnieją **jedynie dwa** odpowiadające mu typy referencyjne. Dlaczego tak jest, dowiesz się za chwilę. Na razie przypatrzmy się deklaracjom przykładowych referencji.

Deklarowanie referencji

Referencje odnoszą się do zmiennych, zatem najpierw przydałoby się jakąś zmienną posiadać. Niech będzie to coś w tym rodzaju:

```
short nZmienna;
```

Odpowiednia referencja, wskazująca na tę zmienną, będzia natomiast zadeklarowana w ten oto sposób:

```
short& nReferencja = nZmienna;
```

Kończący nazwę typu znak `&` jest wyróżnikiem, który mówi nam i kompilatorowi, że mamy do czynienia właśnie z referencją. Inicjalizujemy ją od razu tak, ażeby wskazywała na naszą zmienną `nZmienna`. Zauważmy, że nie używamy do tego **żadnego dodatkowego operatora**!

Posługując się referencją możliwe jest teraz zwyczajne odwoływanie się do zmiennej, do której się ona odnosi. Wygląda to więc bardzo zachęcająco - na przykład:

```
nReferencja = 1;      // przypisanie wartości zmiennej nZmienna
std::cout << nReferencja;  // wyświetlenie wartości zmiennej nZmienna
```

Wszystkie operacje, jakie tu wykonujemy, odbywają się **na zmiennej `nZmienna`**, chociaż wygląda, jakby to `nReferencja` była jej celem. Ona jednak tylko w nich **pośredniczy**, tak samo jak czynią to wskaźniki. Referencja **nie wymaga** jednak skorzystania z operatora `*` (zwanego *notabene* operatorem **dereferencji**) celem dostania się do miejsca pamięci, na które sama wskazuje. Ten właśnie fakt (między innymi) różni ją od wskaźnika.

Prawo stałości referencji

Najdziwniej wygląda pewnie linijka z przypisaniem wartości. Mimo że po lewej stronie znaku `=` stoi zmienna `nReferencja`, to jednak nową wartość otrzyma nie ona, lecz `nZmienna`, na którą tamta pokazuje. Takie są po prostu uroki referencji i trzeba do nich przywyknąć.
No dobrze, ale jak w takim razie **zmienić adres** pamięci, na który pokazuje nasza referencja?… Powiedzmy, że zadeklarujemy sobie drugą zmienną:

```
short nInnaZmienna;
```

Chcemy mianowicie, żeby odtąd `nReferencja` pokazywała właśnie na nią (a nie na `nZmienna`). Jak (czy?) można to uczynić?…

Niestety, odpowiedź brzmi: nijak. Raz ustalona referencja **nie może** być bowiem „doczepiona” do innej zmiennej, lecz do końca pozostaje związana wyłącznie z tą pierwszą. A zatem:

W C++ występują **wyłącznie stałe referencje**. Po **koniecznej** inicjalizacji nie mogą już być zmieniane.

To jest właśnie powód, dla którego istnieją tylko dwa warianty typów referencyjnych. O ile więc w przypadku wskaźników atrybut `const` mógł występować (lub nie) w dwóch różnych miejscach deklaracji, o tyle dla referencji jego drugi występ jest niejako **domyślny**. Nie istnieje zatem żadna „niestała referencja”.
Przypisanie zmiennej do referencji może więc się odbywać **tylko podczas jej inicjalizacji**. Jak widzieliśmy, dzieje się to prawie tak samo, jak przy stałych wskaźnikach - naturalnie z wyłączeniem braku operatora `&`, np.:

```
float fLiczba;
float& fRef = fLiczba;
```

Czy fakt ten jest jakąś niezmiernie istotną wadą referencji? Śmiem twierdzić, że ani trochę! Tak naprawdę prawie nigdy nie używa się mechanizmu referencji w odniesieniu do zwykłych zmiennych. Ich prawdziwa użyteczność ujawnia się bowiem dopiero w połączeniu z funkcjami.
Zobaczmy więc, dlaczego są wówczas takie wspaniałe ;D

Referencje i funkcje

Chyba jedynym miejscem, gdzie rzeczywiście używa się referencji, są nagłówki funkcji (prototypy). Dotyczy to zarówno parametrów, jak i wartości przez te funkcje zwracanych. Referencje dają bowiem całkiem znaczące optymalizacje w szybkości działania kodu, i to w zasadzie za darmo. Nie wymagają żadnego dodatkowego wysiłku poza ich użyciem w miejsce zwykłych typów.
Brzmi to bardzo kusząco, zatem zobaczmy te wyśmienite rozwiązania w akcji.

Parametry przekazywane przez referencje

Już przy okazji wskaźników zauważyliśmy, że wykorzystanie ich jako parametrów funkcji może przyspieszyć działanie programu. Zamiast całych obiektów funkcja otrzymuje wtedy odwołania do nich, zaś poprzez nie może odnosić się do faktycznych obiektów. Na potrzeby funkcji kopiowane są więc tylko 4 bajty odwołania, a nie czasem wiele kilobajtów właściwego obiektu!
Przy tej samej okazji narzekaliśmy jednak, że zastosowanie wskaźników wymaga przeformatowania składni całego kodu, w którym należy dodać konieczne dereferencje i zmienić operatory wyłuskania. To niewielki, ale jednak dolegliwy kłopot.

I oto nagle pojawia się cudowne rozwiązanie :) Referencje, bo o nich rzecz jasna mówimy, są także odwołaniami do obiektów, ale możliwe jest stosowanie wobec nich zwyczajnej składni, bez uciążliwości związanych ze wskaźnikami. Czyniąc je parametrami funkcji, powinniśmy więc upiec dwie pieczenie na jednym ogniu, poprawiając zarówno osiągi programu, jak i własne samopoczucie :D
Spójrzmy zatem jeszcze raz na funkcję `Wyszukaj()`, z którą spotkaliśmy się już przy wskaźnikach. Tym razem jej parametry będą jednak referencjami. Oto jak wpłynie to na wygląd kodu:

```
#include <string>

int Wyszukaj  (const std::string& strSzukany,
               const std::string& strPrzeszukiwany)
{
   // przeszukujemy nasz napis
   for (unsigned i = 0;
         i <= strPrzeszukiwany.length() - strSzukany.length(); ++i)
   {
```

```
            // porównujemy kolejne wycinki napisu
            if (strPrzeszukiwany.substr(i, strSzukany.length())
                == strSzukany)
                  // jeżeli wycinek zgadza się, to zwracamy jego indeks
                  return i;
      }

      // w razie niepowodzenia zwracamy -1
      return -1;
   }
```

Obecnie nie widać tu najmniejszych oznak silenia się na jakąkolwiek optymalizację, a mimo jest ona taka sama jak w wersji wskaźnikowej. Powodem jest forma nagłówka funkcji:

```
int Wyszukaj  (const std::string& strSzukany,
               const std::string& strPrzeszukiwany)
```

Oba jej parametry są tutaj **referencjami do stałych** napisów, a więc nie są kopiowane w inne miejsca pamięci wyłącznie na potrzeby funkcji. A jednak, chociaż faktycznie funkcja otrzymuje tylko ich adresy, możemy operować na tych parametrach zupełnie tak samo, jakbyśmy dostali całe obiekty poprzez ich wartości. Mamy więc zarówno wygodną składnię, jak i dobrą wydajność tak napisanej funkcji.

Zatrzymajmy się jeszcze przez chwilę przy modyfikatorach `const` w obu parametrach funkcji. Obydwa napisy nie w jej ciele w żaden sposób zmieniane (bo i nie powinny), zatem logiczne jest zadeklarowanie ich jako referencji do stałych. W praktyce tylko takie referencje stosuje się jako parametry funkcji; jeżeli bowiem należy zwrócić jakąś wartość poprzez parametr, wtedy lepiej dla zaznaczenia tego faktu użyć odpowiedniego wskaźnika.

Zwracanie referencji

Na podobnej zasadzie, na jakiej funkcje mogą pobierać referencje poprzez swoje parametry, mogą też je zwracać na zewnątrz. Uzasadnienie dla tego zjawiska jest również takie samo, czyli zaoszczędzenie niepotrzebnego kopiowania wartości.

Najprotszym przykładem może być ciekawe rozwiązanie problemu metod dostępowych - tak jak poniżej:

```
class CFoo
{
   private:
         unsigned m_uPole;
   public:
         unsigned& Pole()  { return m_uPole; }
};
```

Ponieważ metoda `Pole()` zwraca referencję, możemy używać jej niemal tak samo, jak zwyczajnej zmiennej:

```
CFoo Foo;
Foo.Pole() = 10;
std::cout << Foo.Pole();
```

Oczywiście kwestia, czy takie rozwiązanie jest w danym przypadku pożądane, jest mocno indywidualna. Zawsze należy rozważyć, czy nie lepiej zastosować tradycyjnego wariantu metod dostępowych - szczególnie, jeżeli chcemy zachowywać kontrolę nad wartościami przypisywanymi polom.

Z praktycznego punktu widzenia zwracanie referencji nie jest więc zbytnio przydatną możliwością. Wspominam jednak o niej, gdyż stanie się ona niezbędna przy okazji przeładowywania operatorów - zagadnienia, którym zajmiemy się w jednym z przyszłych rozdziałów.

***

Tym drobnym wybiegnięciem w przyszłość zakończymy nasze spotkania ze wskaźnikami na zmienne. Jeżeli miałeś jakiekolwiek wątpliwości co do użyteczności tego elementu języka C++, to chyba do tego momentu zostały one całkiem rozwiane. Najlepiej jednak przekonasz się o przydatności mechanizmów wskaźników i referencji, kiedy sam będziesz miał okazję korzystać z nich w swoich własnych aplikacjach. Przypuszczam także, że owe okazje nie będą wcale odosobnionymi przypadkami, ale stałą praktyką programistyczną.

Oprócz wskaźników na zmienne język C++ oferuje również inną ciekawą konstrukcję, jaką są wskaźniki na funkcje. Nie od rzeczy będzie więc zapoznanie się z nimi, co też pilnie uczynimy.

Wskaźniki do funkcji

Myśląc o tym, co jest przechowywane w pamięci operacyjnej, zwykle wyobrażamy sobie różne dane programu: zmienne, tablice, struktury itp. One stanowią informacje reprezentowane w komórkach pamięci, na których aplikacja wykonuje swoje działania. Cała pamięć operacyjna jest więc usiana danymi każdego z aktualnie pracujących programów.

Hmm… Czy aby na pewno o czymś nie zapomnieliśmy? A co z samymi programami?! Kod aplikacji jest przecież pewną porcją binarnych danych, zatem i ona musi się gdzieś podziać. Przez większość czasu egzystuje wprawdzie na dysku twardym w postaci pliku (zwykle o rozszerzeniu EXE), ale dla potrzeb wykonywania kodu jest to z pewnością zbyt wolne medium. Gdyby system operacyjny co rusz sięgał do pliku w czasie działania programu, wtedy na pewno wszelkie czynności ciągnęłyby się niczym toffi i przyprawiały zniecierpliwionego użytkownika o białą gorączkę. Co więc zrobić z tym fantem?… Rozsądnym wyjściem jest umieszczenie w pamięci operacyjnej **także kodu** działającej aplikacji. Dostęp do nich jest wówczas wystarczająco szybki, aby programy mogły działać w normalnym tempie i bez przeszkód wykonywać swoje zadania. Pamięć RAM jest przecież stosunkowo wydajna, wielokrotnie bardziej niż nawet najszybsze dyski twarde.

Tak więc podczas uruchamiania programu jego kod jest umieszczany wewnątrz pamięci operacyjnej. Każdy podprogram, każda funkcja, a nawet każda instrukcja otrzymują wtedy swój unikalny **adres**, zupełnie jak zmienne. Maszynowy kod binarny jest bowiem także swoistego rodzaju danymi. Z tych danych korzysta system operacyjny (głównie poprzez procesor), wykonując kolejne instrukcje aplikacji. Wiedza o tym, jaka komenda ma być za chwilę uruchomiona, jest przechowywana właśnie w postaci jej adresu - czyli po prostu **wskaźnika**.
Nam zwykle nie jest potrzebna aż tak dokładna lokalizacja jakiegoś wycinka kodu w naszej aplikacji, szczególnie jeżeli programujemy w języku wysokiego poziomu, którym jest z pewnością C++. Trudno jednak pogardzić możliwością uzyskania **adresu funkcji** w programie, jeśli przy pomocy tegoż adresu (oraz kilku dodatkowych informacji, o czym za chwilę) można ową funkcję swobodnie wywoływać. C++ oferuje więc mechanizm **wskaźników do funkcji**, który udostępnia taki właśnie potencjał.

Wskaźnik do funkcji (ang. *pointer to function*) to w C++ zmienna, która przechowuje **adres**, pod jakim istnieje w pamięci operacyjnej dana **funkcja**.

Wiem, że początkowo może być ci trudno uświadomić sobie, w jaki sposób kod programu jest reprezentowany w pamięci i jak wobec tego działają wskaźniki na funkcje. Dokładnie wyjaśnienie tego faktu wykracza daleko poza ramy tego rozdziału, kursu czy nawet programowania w C++ jako takiego (oraz, przyznam szczerze, częściowo także mojej wiedzy :D). Dotyka to już bowiem niskopoziomowych aspektów działania aplikacji.
Niemniej postaram się przystępnie wyjaśnić przynajmniej te zagadnienia, które będą nam potrzebne do sprawnego posługiwania się wskaźnikami do funkcji. Zanim to się stanie, możesz myśleć o nich jako o swoistych łączach do funkcji, podobnych w swych założeniach do skrótów, jakie w systemie Windows można tworzyć w odniesieniu do aplikacji. Tutaj natomiast mamy do czynienia z pewnego rodzaju „skrótami" do pojedynczych funkcji; przy ich pomocy możemy je bowiem wywoływać niemal w ten sam sposób, jak to czynimy bezpośrednio.

Omawianie wskaźników do funkcji zaczniemy nieco od tyłu, czyli od bytów na które one wskazują - a więc od funkcji właśnie. Przypomnimy sobie, cóż takiego charakteryzuje funkcję oraz powiemy sobie, jakie jej cechy będą szczególne istotne w kontekście wskaźników.
Potem rzecz jasna zajmiemy się używaniem wskaźników do funkcji w naszych własnych programach, poczynając od deklaracji aż po wywoływanie funkcji za ich pośrednictwem.
Na koniec uświadomimy sobie także kilka zastosowań tej ciekawej konstrukcji programistycznej.

Cechy charakterystyczne funkcji

Różnego rodzaju funkcji - czy to własnych, czy też wbudowanych w język - używaliśmy dotąd tak często i w takich ilościach, że chyba nikt nie ma najmniejszych wątpliwości, czym one są, do czego służą i jaka jest ich rola w programowaniu.
Teraz więc przypomnimy sobie jedynie te własności funkcji, które będą dla nas istotne przy omawianiu wskaźników. Nie omieszkamy także poznać jeszcze jednego aspektu funkcji, o którym nie mieliśmy okazji dotychczas mówić. Wszystko to pomoże nam zrozumieć koncepcję i stosowanie wskaźników na funkcje.

Trzeba tu zaznaczyć, że w tym momencie absolutnie nie chodzi nam o to, jakie instrukcje mogą zawierać funkcje. Przeciwnie, nasza uwaga będzie skoncentrowana wyłącznie na **prototypie funkcji**, jej „wizytówce". Dla wskaźników na funkcje pełni on bowiem podobne posługi, jak typ danych dla wskaźników na zmienne. Sama zawartość bloku funkcji, podobnie jak wartość zmiennej, jest już zupełnie jednak inną („wewnętrzną") sprawą.
Na początek przyjrzyjmy się składni prototypu (deklaracji) funkcji. Wydaje się, że jest ona doskonale nam znana, jednak tutaj przedstawimy jej pełną wersję:

```
zwracany_typ [konwencja_wywołania] nazwa_funkcji([parametry]);
```

Każdemu z jej elementów przypatrzymy się natomiast w osobnym paragrafie.

Typ wartości zwracanej przez funkcję

Wiele języków programowania rozróżnia dwa rodzaje podprogramów. I tak procedury mają za zadanie wykonanie jakichś czynności, zaś funkcje są przeznaczone do obliczania pewnych wartości. Dla obu tych rodzajów istnieją zwykle odmienne rozwiązania składniowe, na przykład inne słowa kluczowe.

W C++ jest nieco inaczej: tutaj zawsze mamy do czynienia z funkcjami, gdyż bezwględnie konieczne jest określenie typu wartości, zwracanej przez nie. Naturalnie może być nim każdy typ, który mógłby również wystepować w deklaracji zmiennej: od typów wbudowanych, poprzez wskaźniki, referencje, aż do definiowanych przez użytkownika typów wyliczeniowych, klas czy struktur (lecz nie tablic).

Specjalną rolę pełni tutaj typ `void` ('pustka'), który jest synonimem 'niczego'. Nie można wprawdzie stworzyć zmiennych należących do tego typu, jednak możliwe jest uczynienie go typem zwracanym przez funkcję. Taka funkcją będzie zatem „zwracać nic”, czyli po prostu nic nie zwracać; można ją więc nazwać procedurą.

Instrukcja czy wyrażenie

Od kwestii, czy funkcja zwraca jakąś wartość czy nie, zależy to, jak możemy nazwać jej wywołanie: **instrukcją** lub też **wyrażeniem**. Różnica pomiędzy tymi dwoma elementami języka programowania jest dość oczywista: instrukcja to polecenie wykonania jakichś działań, zaś wyrażenie - obliczenia pewnej wartości; wartość ta jest potem reprezentowana przez owo wyrażenie.
C++ po raz kolejny raczy nas tu niespodzianką. Otóż w tym języku **niemal wszystko jest wyrażeniem** - nawet taka wybitnie „instrukcyjna” działalność jak choćby przypisanie. Rzadko jednak używamy jej w takim charakterze, zaś o wiele częściej jako zwykłą instrukcję i jest to wówczas całkowicie poprawne.

Wyrażenie może być w programowaniu użyte jako instrukcja, natomiast instrukcja nie może być użyta jako wyrażenie.

Dla wyrażenia występującego w roli instrukcji jest wprawdzie obliczana jego wartość, ale nie zostaje potem do niczego wykorzystana. To raczej typowa sytuacja i chociaż może brzmi niepokojąco, większość kompilatorów nigdy o niej nie ostrzega i trudno poczytywać to za ich beztroskę.

Pedantyczni programiści stosują jednak niecodzienny zabieg rzutowania na typ `void` dla wartości zwróconej przez funkcję użytą w charakterze instrukcji. Nie jest to rzecz jasna konieczne, ale niektórzy twierdzą, iż można w ten sposób unikać nieporozumień.

Przeciwny przypadek: kiedy staramy się umieścić wywołanie procedury (niezwracającej żadnej wartości) wewnątrz normalnego wyrażenia, jest już w oczywisty sposób nie do przyjęcia. Takie wywołanie nie reprezentuje bowiem żadnej wartości, która mogłaby być użyta w obliczeniach. Można to również interpretować jako niezgodność typów, ponieważ `void` jako typ pusty jest niekompatybilny z żadnym innym typem danych.

Widzimy zatem, że kwestia zwracania lub niezwracania przez funkcję wartości oraz jej rodzaju jest nierzadko bardzo ważna.

Konwencja wywołania

Trochę trudno w to uwierzyć, ale podanie (zdawałoby się) wszystkiego, co można powiedzieć o danej funkcji: jej parametrów, wartości przezeń zwracanej, nawet nazwy - nie wystarczy kompilatorowi do jej poprawnego wywołania. Będzie on aczkolwiek wiedział, co musi zrobić, ale nikt mu nie powie, **jak** ma to zrobić.

Cóż to znaczy?… Celem wyjaśnienia porównajmy całą sytuację do telefonowania. Gdy mianowicie chcemy zadzwonić pod konkretny numer telefonu, mamy wiele możliwych dróg uczynienia tego. Możemy zwyczajnie pójść do drugiego pokoju, podnieść słuchawkę stacjonarnego aparatu i wystukać odpowiedni numer. Możemy też siegnąc po telefon komórkowy i użyć go, wybierając na przykład właściwą pozycję z jego książki adresowej. Teoretycznie możemy też wybrać się do najbliższej budki telefonicznej i skorzystać z zainstalowanego tam aparatu. Wreszcie, możliwe jest wykorzystanie modemu umieszczonego w komputerze i odpowiedniego oprogramowania albo też dowolnej formy dostępu do globalnej sieci oraz protokołu VoIP (*Voice over Internet Protocol*). Technicznych możliwości mamy więc mnóstwo i zazwyczaj wybieramy tę, która jest nam w aktualnej chwili najwygodniejsza. Zwykle też osoba po drugiej stronie linii nie odczuwa przy tym żadnej różnicy.

Podobnie rzecz ma się z wywoływaniem funkcji. Znając jej miejsce docelowe (adres funkcji w pamięci) oraz ewentualne dane do przekazania jej w parametrach, możliwe jest zastosowanie kilku dróg osiągnięcia celu. Nazywamy je **konwencjami wywołania** funkcji.

Konwencja wywołania (ang. *calling convention*) to określony sposób wywoływania funkcji, precyzujący przede wszystkim kolejność przekazywania jej parametrów.

Dziwisz się zapewne, dlaczego dopiero teraz mówimy o tym aspekcie funkcji, skoro jasno widać, iż jest on nieodzowny dla ich działania. Przyczyna jest prosta. Wszystkie funkcje, jakie samodzielnie wpiszemy do kodu i dla których nie określimy konwencji wywołania, posiadają domyślny jej wariant, właściwy dla języka C++. Jeżeli zaś chodzi o funkcje biblioteczne, to ich prototypy zawarte w plikach naglówkowych zawierają informacje o używanej konwencji. Pamiętajmy, że korzysta z nich głównie sam kompilator, gdyż w C++ wywołanie funkcji wygląda **składniowo zawsze tak samo**, niezależnie od jej konwencji. Jeżeli jednak używamy funkcji do innych celów niż tylko prostego przywoływania (a więc stosujemy choćby wskaźniki na funkcje), wtedy wiedza o konwencjach wywołania staje się potrzebna także i dla nas.

O czym mówi konwencja wywołania?

Jak już wspomniałem, konwencja wywołania determinuje głównie **przekazywanie parametrów** aktualnych dla funkcji, by mogła ona używać ich w swoim kodzie. Obejmuje to **miejsce w pamięci**, w którym są one tymczasowo przechowywane oraz **porządek**, w jakim są w tym miejscu kolejno umieszczane.

Podstawowym rejonem pamięci operacyjnej, używanym jako pośrednik w wywołaniach funkcji, jest **stos**. Dostęp do tego obszaru odbywa się w dość osobliwy sposób, który znajdują zresztą odzwierciedlenie w jego nazwie. Stos charakteryzuje się bowiem tym, że gdy położymy na nim po kolei kilka elementów, wtedy mamy bezpośredni dostęp jedynie do tego **ostatniego**, położonego najpóźniej (i najwyżej). Jeżeli zaś chcemy dostać się do obiektu znajdującego się na samym dole, wówczas musimy zdjąć po kolei wszystkie pozostałe elementy, umieszczone na stosie później. Czynimy to więc w **odwrotnej kolejności** niż następowało ich odkładanie na stos.

Dobrą przykładem stosu może być hałda książek, piętrząca się na twoim biurku ;D

Jeśli zatem wywołujący funkcję (ang. *caller*) umieści na stosie jej parametry w pewnym porządku (co zresztą czyni), to sama funkcja (ang. *callee* - wywoływana albo *routine* - podprogram) musi je pozyskać w kolejności odwrotnej, aby je właściwie zinterpretować. Obie strony korzystają przy tym z informacji o konwencji wywołania, lecz w opisach „katalogowych” poszczególnych konwencji podaje się wyłącznie **porządek stosowany przez wywołującego**, a więc tylko **kolejność odkładania** parametrów na stos. Kolejność ich podejmowania z niego jest przecież dokładnie odwrotna.
Nie myśl jednak, że kompilatory dokonują jakichś złożonych permutacji parametrów funkcji podczas ich wywoływania. Tak naprawdę istnieją jedynie dwa porządki, które mogą być kanwą dla konwencji i stosować się dla każdej funkcji bez wyjątku.
Można mianowicie podawać parametry wedle ich deklaracji w prototypie funkcji, czyli od lewej do prawej strony. Wówczas to wywołujący jest w uprzywilejowanej pozycji, gdyż używa bardziej naturalnej kolejności; sama funkcja musi użyć odwrotnej. Drugi wariant to odkładanie parametrów na stos w odwrotnej kolejności niż w deklaracji funkcji; wtedy to funkcja jest w wygodniejszej sytuacji.

Oprócz stosu do przekazywania parametrów można też używać **rejestrów procesora**, a dokładniej jego czterech rejestrów uniwersalnych. Im więcej parametrów zostanie tam umieszczonych, tym szybsze powinno być (przynajmniej w teorii) wywołanie funkcji.

Typowe konwencje wywołania

Gdyby każdy programista ustalał własne konwencje wywołania funkcji (co jest teoretycznie możliwe), to oczywiście natychmiast powstałby totalny rozgardiasz w tej materii. Konieczność uwzględniania upodobań innych koderów byłaby z pewnością niezwykle frustrująca.

Za sprawą języków wysokiego poziomu nie ma na szczęście aż tak wielkich problemów z konwencjami wywołania. Jedynie korzystając z kodu napisanego w innym języku trzeba je uwzględniać. W zasadzie więc zdarza się to dość często, ale w praktyce cały wysiłek włożony w zgodność z konwencjami ogranicza się **co najwyżej** do dodania odpowiedniego słowa kluczowego do prototypu funkcji - w miejsce, które oznaczyłem w jego składni jako `konwencja_wywołania`. Często nawet i to nie jest konieczne, jako że prototypy funkcji oferowanych przez przeróżne biblioteki są umieszczane w ich plikach nagłówkowych, zaś zadanie programisty-użytkownika ogranicza się jedynie do włączenia tychże nagłówków do własnego kodu.

Kompilator wykonuje zatem sporą część pracy za nas. Warto jednak przynajmniej znać te najczęściej wykorzystywane konwencje wywołania, a nie jest ich wcale aż tak dużo. Poniższa lista przedstawia je wszystkie:

- *cdecl* - skrót od *C declaration* ('deklaracja C').. Zgodnie z nazwą jest to domyślna konwencja wywołania w językach C i C++. W Visual C++ można ją jednak jawnie określić poprzez słowo kluczowe `__cdecl`. Parametry są w tej konwencji przekazywane na stos w kolejności od prawej do lewej, czyli odwrotnie niż są zapisane w deklaracji funkcji
- *stdcall* - skrót od *Standard Call* ('standardowe wywołanie'). Jest to konwencja zbliżona do *cdecl*, posługuje się na przykład tym samym porządkiem odkładania parametrów na stos. To jednocześnie niepisany standard przy pisaniu kodu, który w skompilowanej formie będzie używany przez innych. Korzysta z niego więc chociażby system Windows w swych funkcjach API.
  W Visual C++ konwencji tej odpowiada słowo `__stdcall`
- *fastcall* ('szybkie wywołanie') jest, jak nazwa wskazuje, zorientowany na szybkość działania. Dlatego też w miarę możliwości używa rejestrów procesora do przekazywania parametrów funkcji.
  Visual C++ obsługuję tą konwencję poprzez słówko `__fastcall`
- *pascal* budzi słuszne skojarzenia z popularnym ongiś językiem programowania. Konwencja ta była w nim wtedy intensywnie wykorzystywana, lecz dzisiaj jest już przestarzała i coraz mniej kompilatorów (wszelkich języków) wspiera ją
- *thiscall* to specjalna konwencja wywoływania metod obiektów w języku C++. Funkcje wywoływane z jej użyciem otrzymują dodatkowy parametr, będący wskaźnikiem na obiekt danej klasy[94]. Nie występuje on na liście parametrów w deklaracji metody, ale jest dostępny poprzez słowo kluczowe `this`. Oprócz tej szczególnej właściwości *thiscall* jest identyczna z *stdcall*.
  Ze względu na specyficzny cel istnienia tej konwencji, nie ma możliwości zadeklarowania zwykłej funkcji, która by jej używała. W Visual C++ nie odpowiada jej więc żadne słowo kluczowe

A zatem dotychczas (nieświadomie!) używaliśmy tylko dwóch konwencji: *cdecl* dla zwykłych funkcji oraz *thiscall* dla metod obiektów. Kiedy zaczniemy naukę programowania aplikacji dla Windows, wtedy ten wachlarz zostanie poszerzony. W każdym przypadku składnia wywołania funkcji w C++ będzie jednak identyczna.

---

[94] Jest on umieszczany w jednym z rejestrów procesora.

Nazwa funkcji

To zadziwiające, że chyba najważniejsza dla programisty cecha funkcji, czyli jej **nazwa**, jest niemal zupełnie nieistotna dla działającej aplikacji!… Jak już bowiem mówiłem, „widzi” ona swoje funkcje wyłącznie poprzez ich adresy w pamięci i przy pomocy tych adresów ewentualnie wywołuje owe funkcje.
Można dywagować, czy to dowód na całkowity brak skrzyżowania między drogami człowieka i maszyny, ale fakt pozostaje faktem, zaś jego przyczyna jest prozaicznie pragmatyczna. Chodzi tu po prostu o wydajność: skoro funkcje programu są podczas jego uruchamiania umieszczane w pamięci operacyjnej (można ładnie powiedzieć: **mapowane**), to dlaczego system operacyjny nie miałby używać wygenerowanych przy okazji adresów, by w razie potrzeby rzeczone funkcje wywoływać? To przecież proste i szybkie rozwiązanie, naturalne dla komputera i niewymagające żadnego wysiłku ze strony programisty. A zatem jest ono po prostu dobre :)

Rozterki kompilatora i linkera

Jedynie w czasie kompilacji kodu nazwy funkcji mają jakieś znaczenie. Kompilator musi bowiem zapewnić ich **unikalność** w skali całego projektu, tj. wszystkich jego modułów. Nie jest to wcale proste, jeżeli przypomnimy sobie o funkcjach przeciążanych, które z założenia mają te same nazwy. Poza tym funkcje o tej samej nazwie mogą też występować w różnych zakresach: jedna może być na przykład metodą jakiejś klasy, zaś druga zwyczajną funkcją globalną.

Kompilator rozwiązuje te problemy, stosując tak zwane **dekorowanie nazw**. Wykorzystuje po prostu dodatkowe informacje o funkcji (jej prototyp oraz zakres, w którym została zadeklarowana), by wygenerować jej niepowtarzalną, wewnętrzną nazwę. Zawiera ona wiele różnych dziwnych znaków w rodzaju `@`, `^`, `!` czy `_`, dlatego właśnie jest określana jako **nazwa dekorowana**.
Wywołania z użyciem takich nazw są umieszczane w skompilowanych modułach. Dzięki temu linker może bez przeszkód połączyć je wszystkie w jeden plik wykonywalny całego programu.

Parametry funkcji

Ogromna większość funkcji nie może obyć się bez dodatkowych danych, przekazywanych im przy wywoływaniu. Pierwsze strukturalne języki programowania nie oferowały żadnego wspomagania w tym zakresie i skazywały na korzystanie wyłącznie ze zmiennych globalnych. Bardziej nowoczesne produkty pozwalają jednak na deklarację parametrów funkcji, co też niejednokrotnie czynimy w praktyce.

Aby wywołać funkcję z parametrami, kompilator musi znać ich **liczbę** oraz **typ** każdego z nich. Informacje te podajemy w prototypie funkcji, zaś w jej kodzie zwykle nadajemy także **nazwy** poszczególnym parametrom, by móc z nich później korzystać.
Parametry pełnią rolę zmiennych lokalnych w bloku funkcji - z tą jednak różnicą, że ich początkowe wartości pochodzą **z zewnątrz**, od kodu wywołującego funkcję. Na tym wszakże kończą się wszelkie odstępstwa, ponieważ parametrów możemy używać identycznie, jak gdyby było one zwykłymi zmiennymi odpowiednich typów. Po zakończeniu wykonywania funkcji są one niszczone, nie pozostawiając żadnego śladu po ewentualnych operacjach, które mogły być na nich dokonywane kodzie funkcji.
Wnioskujemy stąd, że:

Parametry funkcji są w C++ przekazywane **przez wartości**.

Reguła ta dotyczy **wszystkich typów parametrów**, mimo że w przypadku wskaźników oraz referencji jest ona pozornie łamania. To jednak tylko złudzenie. W rzeczywistości także i tutaj do funkcji są przekazywane **wyłącznie wartości** - tyle tylko, że owymi

wartościami są tu adresy odpowiednich komórek w pamięci. Za ich pośrednictwem możemy więc uzyskać dostęp do rzeczonych komórek, zawierających na przykład jakieś zmienne. Gdy dodatkowo korzystamy z referencji, wtedy nie wymaga to nawet specjalnej składni. Trzeba być jednak świadomym, że zjawiska te dotyczą samej natury wskaźników czy też referencji, **nie zaś parametrów** funkcji! Dla nich bowiem zawsze obowiązuje przytoczona wyżej zasada przekazywania poprzez wartość.

Używanie wskaźników do funkcji

Przypomnieliśmy sobie i uzupełniliśmy wszystkie niezbędne wiadomości funkcjach, konieczne do poznania i stosowania wskaźników na nie. Teraz więc możemy już przejść do właściwej części tematu.

Typy wskaźników do funkcji

Jakkolwiek wskaźniki są przede wszystkim adresami miejsc w pamięci operacyjnej, niemal wszystkie języki programowania oraz ich kompilatory wprowadzają pewne dodatkowe informacje, związane ze wskaźnikami. Chodzi tu głównie o **typ wskaźnika**. W przypadku wskaźników na zmienne był on pochodną typu zmiennej, na którą dany wskaźnik pokazywał. Podobne pojęcie istnieje także dla wskaźników do funkcji - w tym wypadku możemy więc mówić o **typie funkcji**, na które wskazuje określony wskaźnik.

Własności wyróżniające funkcję

Co jednak mamy rozumieć pod pojęciem „typ funkcji”? W jaki sposób funkcja może w ogóle być zakwalifikowana do jakiegoś rodzaju?…

W odpowiedzi może nam znowu pomóc analogia do zmiennych. Otóż typ zmiennej określamy w momencie jej deklaracji - jest nim w zasadzie cała ta deklaracja **z wyłączeniem nazwy**. Określa ona wszystkie cechy deklarowanej zmiennej, ze szczególnym uwzględnieniem rodzaju informacji, jakie będzie ona przechowywać. Typu funkcji możemy zatem również szukać w jej deklaracji, czyli prototypie. Kiedy bowiem wyłączymy z niego nazwę funkcji, wtedy pozostałe składniki wyznaczą nam jej typ. Będą to więc kolejno:

- typ wartości zwracanej przez funkcję
- konwencja wywołania funkcji
- parametry, które funkcja przyjmuje

Wraz z adresem danej funkcji stanowi to wystarczający zbiór informacji dla kompilatora, na podstawie których może on daną funkcję wywołać.

Typ wskaźnika do funkcji

Posiadając wyliczone wyżej wiadomości na temat funkcji, możemy już bez problemu zadeklarować właściwy wskaźnik na nią. Typ tego wskaźnika będzie więc **oparty** na typie funkcji - to samo zjawisko miało miejsce także dla zmiennych.

> **Typ wskaźnika na funkcję** określa typ zwracanej wartości, konwencję wywołania oraz listę parametrów funkcji, na które wskaźnik może pokazywać i które mogą być za jego pośrednictwem wywoływane.

Wiedząc to, możemy przystąpić do poznania sposobu oraz składni, poprzez które język C++ realizuje mechanizm wskaźników do funkcji.

Wskaźniki do funkcji w C++

Deklarując wskaźnik do funkcji, musimy podać jego typ, czyli te trzy cechy funkcji, o których już kilka razy mówiłem. Jednocześnie kompilator powinien wiedzieć, że ma do czynienia ze wskaźnikiem, a nie z funkcją jako taką. Oba te wymagania skutkują specjalną składnią deklaracji wskaźników na funkcje w C++.

Zacznijmy zatem od najprostszego przykładu. Oto deklaracja wskaźnika do funkcji, która nie przyjmuje żadnych parametrów i nie zwraca też żadnego rezultatu[95]:

```
void (*pfnWskaznik)();
```

Jesteśmy teraz władni użyć tego wskaźnika i wywołać za jego pośrednictwem funkcję o odpowiednim nagłówku (czyli nic niebiorącą oraz nic niezwracającą). Może to wyglądać chociażby tak:

```
#include <iostream>

// funkcja, którą będziemy wywoływać
void Funkcja()
{
   std::cout << "Zostalam wywolana!";
}

void main()
{
   // deklaracja wskaźnika na powyższą funkcję
   void (*pfnWskaznik)();

   // przypisanie funkcji do wskaźnika
   pfnWskaznik = &Funkcja;

   // wywołanie funkcji poprzez ten wskaźnik
   (*pfnWskaznik)();
}
```

Ponownie, tak samo jak w przypadku wskaźników na zmienne, moglibyśmy wywołać naszą funkcję bezpośrednio. Pamiętasz jednakże o korzyściach, jakie daje wykorzystanie wskaźników - większość z nich dotyczy także wskaźników do funkcji. Ich użycie jest więc często bardzo przydatne.

Omówmy zatem po kolei wszystkie aspekty wykorzystania wskaźników do funkcji w C++.

Od funkcji do wskaźnika na nią

Deklaracja wskaźnika do funkcji jest w C++ dość nietypową czynnością. Nie przypomina bowiem znanej nam doskonale deklaracji w postaci:

```
typ_zmiennej nazwa_zmiennej;
```

Zamiast tego nazwa wskaźnika jest niejako **wtrącona** w typ funkcji, co w pierwszej chwili może być nieco mylące. Łatwo jednak można zrozumieć taką formę deklaracji, jeżeli porównamy ją z prototypem funkcji, np.:

```
float Funkcja(int);
```

[95] Posiada też domyślną w C++ konwencję wywołania, czyli *cdecl*. Później zobaczymy przykłady wskaźników do funkcji, wykorzystujących inne konwencje.

Otóż odpowiadający mu wskaźnik, który mógłby pokazywać na zadeklarowaną wyżej funkcję `Funkcja()`, zostanie wprowadzony do kodu w ten sposób:

```
float (*pfnWskaznik)(int);
```

Nietrudno zauważyć różnicę: zamiast nazwy funkcji, czyli `Funkcja`, mamy tutaj frazę `(*pfnWskaznik)`, gdzie `pfnWskaznik` jest oczywiście nazwą zadeklarowanego właśnie wskaźnika. Może on pokazywać na funkcje przyjmujące jeden parametr typu `int` oraz zwracające wynik w postaci liczby typu `float`.

Ogólnie zatem, dla **każdej funkcji** o tak wyglądającym prototypie:

```
zwracany_typ nazwa_funkcji([parametry]);
```

deklaracja odpowiadającego jej wskaźnika jest bardzo podobna:

```
zwracany_typ (*nazwa_wskaźnika)([parametry]);
```

Ogranicza się więc do niemal mechanicznej zmiany ściśle określonego fragmentu kodu.

Deklaracja wskaźnika na funkcję o domyślnej konwencji wywołania wygląda tak, jak jej prototyp, w którym *`nazwa_funkcji`* została zastąpiona przez `(*nazwa_wskaźnika)`.

Ta prosta zasada sprawdza się w 99 procentach przypadków i będziesz z niej stale korzystał we wszystkich programach wykorzystujących mechanizm wskaźników do funkcji.

Trzeba jeszcze podkreślić znaczenie nawiasów w deklaracji wskaźników do funkcji. Mają one tutaj niebagatelną rolę składniową, gdyż ich brak całkowicie zmienia sens całej deklaracji. Gdybyśmy więc opuścili je:

```
void *pfnWskaznik();        // a co to jest?
```

cała instrukcja zostałaby zinterpretowana jako:

```
void* pfnWskaznik();        // to prototyp funkcji, a nie wskaźnik na nią!
```

i zamiast wskaźnika do funkcji otrzymalibyśmy funkcję zwracającą wskaźnik. Jest to oczywiście całkowicie niezgodne z naszą intencją.

Pamiętaj zatem o poprawnym umieszczaniu nawiasów w deklaracjach wskaźników do funkcji.

Specjalna konwencja

Opisanego powyżej sposobu tworzenia deklaracji nie można niestety użyć do wskaźników do funkcji, które stosują inną konwencję wywołania niż domyślna (czyli *cdecl*) i zawierają odpowiednie słowo kluczowe w swoim naglówku czy też prototypie. W Visual C++ tymi słowami są `__cdecl`, `__stdcall` oraz `__fastcall`.
Przykład funkcji podpadającej pod te warunki może być następujący:

```
float __fastcall Dodaj(float fA, float fB)   { return fA + fB; }
```

Dodatkowe słowo między `zwracanym_typem` oraz `nazwą_funkcji` całkowicie psuje nam schemat deklaracji wskaźników. Wynik jego zastosowania zostałby bowiem odrzucony przez kompilator:

```
float __fastcall (*pfnWskaznik)(float, float);     // BŁĄD!
```

Dzieje się tak, ponieważ gdy widzi on najpierw nazwę typu (`float`), a potem specyfikator konwencji wywołania (`__fastcall`), bezdyskusyjne interpretuje całą linijkę jako deklarację funkcji. Następującą potem niespodziewaną sekwencję `(*pfnWskaznik)` traktuje więc jako błąd składniowy.

By go uniknąć, musimy **rozciągnąć nawiasy**, w których umieszczamy nazwę wskaźnika do funkcji i „wziąć pod ich skrzydła" także określenie konwencji wywołania. Dzięki temu kompilator napotka otwierający nawias zaraz po nazwie zwracanego typu (`float`) i zinterpretuje całość jako deklarację wskaźnika do funkcji. Wygląda ona tak:

```
float (__fastcall *pfnWskaznik)(float, float);     // OK
```

Ten, zdawałoby się, szczegół może niekiedy stanąć ością w gardle w czasie kompilacji programu. Wypadałoby więc o nim pamiętać.

Składnia deklaracji wskaźnika do funkcji

Obecnie możemy już zobaczyć ogólną postać deklaracji wskaźnika do funkcji. Jeżeli uważnie przestudiowałeś poprzednie akapity, to nie będzie on dla ciebie żadną niespodzianką. Przedstawia się zaś następująco:

```
zwracany_typ ([konwencja_wywołania] *nazwa_wskaźnika)([parametry]);
```

Pasujący do niego prototyp funkcji wygląda natomiast w ten sposób:

```
zwracany_typ [konwencja_wywołania] nazwa_funkcji([parametry]);
```

Z obu tych wzorców widać, że deklaracja wskaźnika do funkcji na podstawie jej prototypu oznacza wykonanie jedynie trzech prostych kroków:

- zamiany *nazwy_funkcji* na *nazwę_wskaźnika*
- dodania * (gwiazdki) przed *nazwą_wskaźnika*
- ujęcia w parę nawiasów ewentualną *konwencję_wywołania* oraz *nazwę_wskaźnika*

Nie jest to więc tak trudna operacja, jak się niekiedy powszechnie sądzi.

Wskaźniki do funkcji w akcji

Zadeklarowanie wskaźnika to naturalnie tylko początek jego wykorzystania w programie. Aby był on użyteczny, powinniśmy przypisać mu adres jakiejś funkcji i skorzystać z niego celem wywołania tejże funkcji. Przypatrzmy się bliżej obu tym czynnościom.

W tym celu zdefiniujmy sobie następującą funkcję:

```
int PobierzLiczbe()
{
   int nLiczba;

   std::cout << "Podaj liczbe: ";
   std::cin >> nLiczba;

   return nLiczba;
}
```

Właściwy wskaźnik, mogący pokazywać na tę funkcję, deklarujemy w ten oto (teraz już, mam nadzieję, oczywisty) sposób:

```
int (*pfnWskaznik)();
```

Jak każdy wskaźnik, zaraz po zadeklarowaniu nie pokazuje on na nic konkretnego - w tym przypadku na żadną konkretną funkcję. Musimy dopiero przypisać mu adres naszej przygotowanej funkcji `PobierzLiczbe()`. Czynimy to więc w następującej zaraz linijce kodu:

```
pfnWskaznik = &PobierzLiczbe;
```

Zwróćmy uwagę, że nazwa funkcji `PobierzLiczbe()` występuje tutaj bez, wydawałoby się - nieodłącznych, nawiasów okrągłych. Ich pojawienie się oznaczałoby bowiem **wywołanie** tej funkcji, a my przecież tego nie chcemy (przynajmniej na razie). Pragniemy tylko **pobrać jej adres w pamięci**, by móc jednocześnie przypisać go do swojego wskaźnika. Wykorzystujemy do tego znany już operator `&`.

Ale… niespodzianka! Ów operator tak naprawdę **nie jest konieczny**. Ten sam efekt osiągniemy również i bez niego:

```
pfnWskaznik = PobierzLiczbe;
```

Po prostu już sam brak nawiasów okrągłych `()`, wyróżniających wywołanie funkcji, jest wystarczająca wskazówką mówiącą kompilatorowi, iż chcemy pobrać adres funkcji o danej nazwie, nie zaś - wywoływać ją. Dodatkowy operator, chociaż dozwolony, nie jest więc niezbędny - wystarczy sama **nazwa funkcji**.
Czy nie mamy w związku z tym uczucia *deja vu*? Identyczną sytuację mieliśmy przecież przy tablicach i wskaźnikach na nie. A zatem zasada, którą tam poznaliśmy, w poprawionej formie stosuje się również do funkcji:

Nazwa funkcji jest także **wskaźnikiem** do niej.

Nie musimy więc korzystać z operatora `&`, by pobrać adres funkcji.

W tym miejscu mamy już wskaźnik `pfnWskaznik` pokazujący na naszą funkcję `PobierzLiczbe()`. Ostatnim aktem będzie wywołanie jej za pośrednictwem tegoż wskaźnika, co czynimy poniższym wierszem kodu:

```
std::cout << (*pfnWskaznik)();
```

Liczbę otrzymaną z funkcji wypisujemy na ekranie, ale najpierw wywołujemy samą funkcję, korzystając między innymi z następnego znajomego operatora - dereferencji, czyli `*`.
Po raz kolejny jednak nie jest to niezbędne! Wywołanie funkcji przy pomocy wskaźnika można z równym powodzeniem zapisać też w takiej formie:

```
std::cout << pfnWskaznik();
```

Jest to druga konsekwencja faktu, iż funkcja jest reprezentowana w kodzie poprzez swój wskaźnik. Taki sam fenomen obserwowaliśmy i dla tablic.

Przykład wykorzystania wskaźników do funkcji

Wskaźniki do funkcji umożliwiają wykonywanie ogólnych operacji przy użyciu funkcji, których implementacja nie musi być im znana. Ważne jest, aby miały one nagłówek zgodny z typem wskaźnika.

Prawie podręcznikowym przykładem może być tu poszukiwanie miejsc zerowych funkcji matematycznej. Procedura takiego poszukiwania jest zawsze identyczna, również same

funkcje mają nieodmiennie tę samą charakterystykę (pobierają liczbę rzeczywistą i taką też liczbę zwracają w wyniku). Możemy więc zaimplementować odpowiedni algorytm (tutaj jest to algorytm **bisekcji**[96]) w sposób **ogólny** - posługując się wskaźnikami do funkcji.

Przykładowy program wykorzystujący tę technikę może przedstawiać się następująco:

```
// Zeros - szukanie miejsc zerowych funkcji

// granica toleracji
const double EPSILON = 0.0001;

// rozpietość badanego przedziału
const double PRZEDZIAL = 100;

// współczynniki funkcji f(x) = k * log_a(x - p) + q
double g_fK, g_fA, g_fP, g_fQ;

//----------------------------------------------------------------------

// badana funkcja
double f(double x)  { return g_fK * (log(x - g_fP) / log(g_fA)) + g_fQ; }

// algorytm szukający miejsca zerowego danej funkcji w danym przedziale
bool SzukajMiejscaZerowego(double fX1, double fX2,      // przedział
                           double (*pfnF)(double),      // funkcja
                           double* pfZero)              // wynik
{
   // najpierw badamy końce podanego przedziału
   if (fabs(pfnF(fX1)) < EPSILON)
   {
         *pfZero = fX1;
         return true;
   }
   else if (fabs(pfnF(fX2)) < EPSILON)
   {
         *pfZero = fX2;
         return true;
   }

   // dalej sprawdzamy, czy funkcja na końcach obu przedziałów
   // przyjmuje wartości różnych znaków
   // jeżeli nie, to nie ma miejsc zerowych
   if ((pfnF(fX1)) * (pfnF(fX2)) > 0) return false;

   // następnie dzielimy przedział na pół i sprawdzamy, czy w ten sposób
   // nie otrzymaliśmy pierwiastka
   double fXp = (fX1 + fX2) / 2;
   if (fabs(pfnF(fXp)) < EPSILON)
   {
         *pfZero = fXp;
         return true;
   }

   // jeśli otrzymany przedział jest wystarczająco mały, to rozwiązaniem
   // jest jego punkt środkowy
   if (fabs(fX2 - fX1) < EPSILON)
```

[96] Oprócz niego popularna jest również metoda Newtona, ale wymaga ona znajomości również pierwszej pochodnej funkcji.

```
    {
          *pfZero = fXp;
          return true;
    }

    // jezeli nadal nic z tego, to wybieramy tę połówkę przedziału,
    // w której zmienia się znak funkcji
    if ((pfnF(fX1)) * (pfnF(fXp)) < 0)
          fX2 = fXp;
    else
          fX1 = fXp;

    // przeszukujemy ten przedział tym samym algorytmem
    return SzukajMiejscaZerowego(fX1, fX2, pfnF, pfZero);
}

//----------------------------------------------------------------------

// funkcja main()
void main()
{
    // (pomijam pobranie współczynników k, a, p i q dla funkcji)


    /* znalezienie i wyświetlenie miejsca zerowego */

    // zmienna na owo miejsce
    double fZero;

    // szukamy miejsca i je wyświetlamy
    std::cout << std::endl;
    if (SzukajMiejscaZerowego(g_fP > -PRZEDZIAL ? g_fP : -PRZEDZIAL,
                              PRZEDZIAL, f, &fZero))
          std::cout << "f(x) = 0  <=>  x = " << fZero << std::endl;
    else
          std::cout << "Nie znaleziono miejsca zerowego." << std::endl;

    // czekamy na dowolny klawisz
    getch();
}
```

Aplikacja ta wyszukuje miejsca zerowe funkcji określonej wzorem:

$$f(x) = k \log_a (x - p) + q$$

Najpierw zadaje więc użytkownikowi pytania co do wartości współczynników k, a, p i q w tym równaniu, a następnie pogrążą się w obliczeniach, by ostatecznie wyświetlić wynik.

Niniejszy program jest przykładem zastosowania wskaźników na funkcje, a nie rozwiązywania równań. Jeśli chcemy wyliczyć miejsce zerowe powyżej funkcji, to znacznie lepiej będzie po prostu przekształcić ją, wyznaczając x:

$$x = \exp_a\left(-\frac{q}{k}\right) + p$$

```
   POSZUKIWANIE MIEJSC ZEROWYCH
---------------------------------

Program poszukuje miejsca zerowego funkcji
o wzorze f(x) = k * log_a(x - p) + q
w przedziale <-100; 100>

Podaj wspolczynnik k: 14
Podaj wspolczynnik a: 3
Podaj wspolczynnik p: 7
Podaj wspolczynnik q: -6

f(x) = 0  <=>  x = 8.60133
_
```

Screen 43. Program poszukujący miejsc zerowych funkcji

Oczywiście w niniejszym programie najbardziej interesująca będzie dla nas funkcja `SzukajMiejscaZerowego()` - głównie dlatego, że wykorzystany w niej został mechanizm wskaźników na funkcje. Ewentualnie możesz też zainteresować się samym algorytmem; jego działanie całkiem dobrze opisują obfite komentarze :)

Gdzie jest więc ów sławetny wskaźnik do funkcji?… Znaleźć go możemy w nagłówku `SzukajMiejscaZerowego()`:

```
bool SzukajMiejscaZerowego(double fX1, double fX2,
                           double (*pfnF)(double),
                           double* pfZero)
```

To nie pomyłka - wskaźnik do funkcji (biorącej jeden parametr `double` i zwracającej także typ `double`) jest tutaj **argumentem innej funkcji**. Nie ma ku temu żadnych przeciwwskazań, może poza dość dziwnym wyglądem nagłówka takiej funkcji. W naszym przypadku, gdzie funkcja jest swego rodzaju „danymi”, na ktorych wykonujemy operacje (szukanie miejsca zerowego), takie zastosowanie wskaźnika do funkcji jest jak najbardziej uzasadnione.
Pierwsze dwa parametry funkcji poszukującej są natomiast liczbami określającymi przedział poszukiwań pierwiastka. Ostatni parametr to z kolei wskaźnik na zmienną typu `double`, poprzez którą zwrócony zostanie ewentualny wynik. Ewentualny, gdyż o powodzeniu lub niepowodzeniu zadania informuje „regularny” rezultat funkcji, będący typu `bool`.

Naszą funkcję szukającą wywołujemy w programie w następujący sposób:

```
double fZero;
if (SzukajMiejscaZerowego(g_fP > -PRZEDZIAL ? g_fP : -PRZEDZIAL,
                          PRZEDZIAL, f, &fZero))
   std::cout << "f(x) = 0  <=>  x = " << fZero << std::endl;
else
   std::cout << "Nie znaleziono miejsca zerowego." << std::endl;
```

Przekazujemy jej tutaj aż dwa wskaźniki jako ostatnie parametry. Trzeci to, jak wiemy, wskaźnik na funkcję - w tej roli występuje tutaj adres funkcji `f()`, którą badamy w poszukiwaniu miejsc zerowych. Aby przekazać jej adres, piszemy po prostu jej nazwę bez nawiasów okrągłych - tak jak się tego nauczyliśmy niedawno.
Czwarty parametr to z kolei zwykły wskaźnik na zmienną typu `double` i do tej roli wystawiamy adres specjalnie przygotowanej zmiennej. Po zakończonej powodzeniem operacji poszukiwania wyświetlamy jej wartość poprzez strumień wyjścia.
Jeżeli zaś chodzi o dwa pierwsze parametry, to określają one obszar poszukiwań, wyznaczony głównie poprzez stałą `PRZEDZIAL`. Dolna granica musi być dodatkowo

„przycięta” z dziedziną funkcji - stąd też operator warunkowy `?:` i porównanie granicy przedziału ze współczynnikiem *p*.

Powiedzmy sobie jeszcze wyraźnie, jaka jest praktyczna korzyść z zastosowania wskaźników do funkcji w tym programie, bo może nie jest ona zbytnio widoczna. Otóż mając wpisany algorytm poszukiwań miejsca zerowego w **ogólnej wersji**, działający na wskaźnikach do funkcji zamiast bezpośrednio na funkcjach, możemy stosować go do tylu różnych funkcji, ile tylko sobie zażyczymy. Nie wymaga to więcej wysiłku niż jedynie zdefiniowania nowej funkcji do zbadania i przekazania wskaźnika do niej jako parametru do `SzukajMiejscaZerowego()`. Uzyskujemy w ten sposób większą elastyczność programu.

Zastosowania

Poprawa elastyczności nie jest jednak jedynym, ani nawet najważniejszym zastosowaniem wskaźników do funkcji. Tak naprawdę stosuje się je głównie w technice programistycznej znanej jako **funkcje zwrotne** (ang. *callback functions*).

Dość powiedzieć, że opierają się na niej wszystkie nowoczesne systemy operacyjne, z Windows na czele. Umożliwia ona bowiem informowanie programów o zdarzeniach zachodzących w systemie (wywołanych na przykład przez użytkownika, jak kliknięcie myszką) i odpowiedniego reagowania na nie. Obecnie jest to najczęstsza forma pisania aplikacji, zwana **programowaniem sterowanym zdarzeniami**. Kiedy rozpoczniemy tworzenie aplikacji dla Windows, także będziemy z niej nieustannie korzystać.

***

I tak zakończyliśmy nasze spotkanie ze wskaźnikami do funkcji. Nie są one może tak często wykorzystywane i przydatne jak wskaźniki na zmienne, ale, jak mogłeś przeczytać, jeszcze wiele razy usłyszysz o nich i wykorzystasz je w przyszłości. Warto więc było dobrze poznać ich składnię (fakt, jest nieco zagmatwana) oraz sposoby użycia.

Podsumowanie

Wskaźniki są często uważane za jedną z natrudniejszych koncepcji programistycznych w ogóle. Wielu całkiem dobrych koderów ma niekiedy większe lub mniejsze kłopoty w ich stosowaniu.
Celowo nie wspomniałem o tych opiniach, abyś mógł najpierw samodzielnie przekonać się o tym, czy zagadnienie to jest faktycznie takie skomplikowane. Dołożyłem przy tym wszelkich starań, by uczynić je chociaż trochę prostszym do zrozumienia. Jednocześnie chciałem jednak, aby zawarty tu opis wskaźników był jak najbardziej dokładny i szczegółowy. Wiem, że pogodzenie tych dwóch dążeń jest prawie niemożliwe, ale mam nadzieję, że wypracowałem w tym rozdziale w miarę rozsądny kompromis.

Zacząłem więc od przedstawienia garści przydatnych informacji na temat samej pamięci operacyjnej komputera. Podejrzewam, że większość czytelników nawet i bez tego była wystarczająco obeznana z tematem, ale przypomnień i uzupełnień nigdy dość :) Przy okazji wprowadziliśmy sobie samo pojęcie wskaźnika.
Dalej zajęliśmy się wskaźnikami na zmienne, ich deklarowaniem i wykorzystaniem: do wspomagania pracy z tablicami, przekazywania parametrów do funkcji czy wreszcie dynamicznej alokacji pamięci. Poznaliśmy też referencje.
Podrozdział o wskaźnikach na funkcje składał się natomiast z poszerzenia wiadomości o samych funkcjach oraz wyczerpującego opisu stosowania wskaźników na nie.

Nieniejszy rozdział jest jednocześnie ostatnim z części 1, stanowiącej podstawowy kurs C++. Po nim przejdziemy (wreszcie ;D) do bardziej zaawansowanych zagadnień języka, Biblioteki Standardowej, a później Windows API i DirectX, a wreszcie do programowania gier.
A zatem pierwszy duży krok już za nami, lecz nadal szykujemy się do wielkiego skoku :)

Pytania i zadania

Tradycji musi stać się zadość: oto świeża porcja pytań dotyczących treści tego rozdziału oraz ćwiczeń do samodzielnego rozwiązania.

Pytania

1. Jakie są trzy rodzaje pamięci wykorzystywanej przez komputer?
2. Na czym polega płaski model adresowania pamięci operacyjnej?
3. Czym jest wskaźnik?
4. Co to jest stos i sterta?
5. W jaki sposób deklarujemy w C++ wskaźniki na zmienne?
6. Jak działają operatory pobrania adresu i dereferencji?
7. Czym rózni się wskaźnik typu `void*` od innych?
8. Dlaczego łańcuchy znaków w stylu C nazywamy napisami zakończonymi zerem?
9. Dlaczego używanie wskaźników lub referencji jako parametrów funkcji może poprawić wydajność programu?
10. W jaki sposób dynamicznie alokujemy zmienne, a w jaki tablice?
11. Co to jest wyciek pamięci?
12. Czym różnią się referencje od wskaźników na zmienne?
13. Jakie podstawowe konwencje wywoływania funkcji są obecnie w użyciu?
14. (**Trudne**) Czy funkcja może nie używać żadnej konwencji wywołania?
15. Jakie są trzy cechy wyznaczające typ funkcji i jednocześnie typ wskaźnika na nią?
16. Jak zadeklarować wskaźnik do funkcji o znanym prototypie?

Ćwiczenia

1. Przejrzyj przykładowe kody z poprzednich rozdziałów i znajdź instrukcje, wykorzystujące wskaźniki lub operatory wskaźnikowe.
2. Zmodyfikuj nieco metodę `ZmienRozmiar()` klasy `CIntArray`. Niech pozwala ona także na zmniejszenie rozmiaru tablicy.
3. Spróbuj napisać podobną klasę dla tablicy dwuwymiarowej.
   (**Trudne**) Niech przechowuje ona elementy w ciągłym obszarze pamięci - tak, jak robi to kompilator ze statycznymi tablicami dwuwymiarowymi.
4. Zadeklaruj wskaźnik do funkcji:
   1) pobierającej jeden parametr typu `int` i zwracającej wynik typu `float`
   2) biorącej dwa parametry typu `double` i zwracającej łańcuch `std::string`
   3) pobierającej trzy parametry: jeden typu `int`, drugi typu `__int64`, a trzeci typu `std::string` i zwracającej wskaźnik na typ `int`
   4) (**Trudniejsze**) przyjmującej jako parametr pięcioelementową tablicę liczb typu `unsigned` i nic niezwracającą
   5) (**Trudne**) zwracającej wartość typu `float` i przyjmującej jako parametr wskaźnik do funkcji biorącej dwa parametry typu `int` i nic niezwracającej
   6) (**Trudne**) pobierającej tablicę pięcioelementową typu `short` i zwracającej jedną liczbę typu `int`
   7) (**Bardzo trudne**) biorącej dwa parametry: jeden typu `char`, a drugi typu `int`, i zwracającej tablicę 10 elementów typu `double`
   Wskazówka: to nie tylko trudne, ale i podchwytliwe :)

8) (**Ekstremalne**) przyjmującej jeden parametr typu `std::string` oraz zwracającej w wyniku wskaźnik do funkcji przyjmującej dwa parametry typu `float` i zwracającej wynik typu `bool`

5. Określ typy parametrów oraz typ wartości zwracanej przez funkcje, na które może pokazywać wskaźnik o takiej deklaracji:
   a) `int (*pfnWskaznik)(int);`
   b) `float* (*pfnWskaznik)(const std::string&);`
   c) `bool (*pfnWskaznik)(void* const, int**, char);`
   d) `const unsigned* const (*pfnWskaznik)(void);`
   e) (**Trudne**) `void (*pfnWskaznik)(int (*)(bool), const char*);`
   f) (**Trudne**) `int (*pfnWskaznik)(char[5], tm&);`
   g) (**Bardzo trudne**) `float (*pfnWskaznik(short, long, bool))(int, int);`