5

ZŁOŻONE ZMIENNE

Mylić się jest rzeczą ludzką,
ale żeby naprawdę coś spaprać
potrzeba komputera.
Edward Morgan Forster

Dzisiaj prawie żaden normalny program nie przechowuje swoich danych jedynie w prostych zmiennych - takich, jakimi zajmowaliśmy się do tej pory (tzw. **skalarnych**). Istnieje mnóstwo różnych sytuacji, w których są one po prostu niewystarczające, a konieczne stają się bardziej skomplikowane konstrukcje. Wspomnijmy choćby o mapach w grach strategicznych, tabelach w arkuszach kalkulacyjnych czy bazach danych adresowych - wszystkie te informacje mają zbyt złożoną naturę, aby dały się przedstawić przy pomocy pojedynczych zmiennych.

Szanujący się język programowania powinien więc udostępniać odpowiednie konstrukcje, służące do przechowywania takich nieelementarnych typów danych. Naturalnie, C++ posiada takowe mechanizmy - zapoznamy się z nimi w niniejszym rozdziale.

Tablice

Jeżeli nasz zestaw danych składa się z wielu drobnych elementów **tego samego rodzaju**, jego najbardziej naturalnym ekwiwalentem w programowaniu będzie **tablica**.

Tablica (ang. *array*) to zespół równorzędnych zmiennych, posiadających wspólną nazwę. Jego poszczególne elementy są rozróżnianie poprzez przypisane im liczby - tak zwane **indeksy**.

Każdy element tablicy jest więc zmienną należącą do tego samego typu. Nie ma tutaj żadnych ograniczeń: może to być liczba (w matematyce takie tablice nazywamy wektorami), łańcuch znaków (np. lista uczniów lub pracowników), pojedynczy znak, wartość logiczna czy jakikolwiek inny typ danych.

W szczególności, elementem tablicy może być także… inna tablica! Takimi podwójnie złożonymi przypadkami zajmiemy się nieco dalej.

Po tej garści ogólnej wiedzy wstępnej, czas na coś przyjemniejszego - czyli przykłady :)

Proste tablice

Zadeklarowanie tablicy przypomina analogiczną operację dla zwykłych (skalarnych) zmiennych. Może zatem wyglądać na przykład tak:

```
int aKilkaLiczb[5];
```

Jak zwykle, najpierw piszemy nazwę wybranego typu danych, a później oznaczenie samej zmiennej (w tym przypadku tablicy - to także jest zmienna). Nowością jest tu para nawiasów kwadratowych, umieszczona na końcu deklaracji. Wewnątrz niej wpisujemy **rozmiar** tablicy, czyli ilość elementów, jaką ma ona zawierać. U nas jest to `5`, a zatem z tylu właśnie liczb (każdej typu `int`) będzie składała się nasza świeżo zadeklarowana tablica.

Skoro żeśmy już wprowadzili nową zmienną, należałoby coś z nią uczynić - w końcu niewykorzystana zmienna to zmarnowana zmienna :) Nadajmy więc jakieś wartości jej kolejnym elementom:

```
aKilkaLiczb[0] = 1;
aKilkaLiczb[1] = 2;
aKilkaLiczb[2] = 3;
aKilkaLiczb[3] = 4;
aKlikaLiczb[4] = 5;
```

Tym razem także korzystamy z nawiasów kwadratowych. Teraz jednak używamy ich, aby uzyskać dostęp do **konkretnego elementu** tablicy, identyfikowanego przez **odpowiedni indeks**. Niewątpliwie bardzo przypomina to docieranie do określonego znaku w zmiennej tekstowej (typu `std::string`), aczkolwiek w przypadku tablic możemy mieć do czynienia z dowolnym rodzajem danych.
Analogia do łańcuchów znaków przejawia się w jeszcze jednym fakcie - są nim oczywiście indeksy kolejnych elementów tablicy. Identycznie jak przy napisach, liczymy je bowiem **od zera**; tutaj są to kolejno `0`, `1`, `2`, `3` i `4`. Na postawie tego przykładu możemy więc sformułować bardziej ogólną zasadę:

Tablica mieszcząca ***n*** **elementów** jest indeksowana wartościami **0**, **1**, **2**, **...**, ***n*** **- 2**, ***n*** **- 1**.

Z regułą tą wiąże się też bardzo ważne ostrzeżenie:

W tablicy *n*-elementowej **nie istnieje** element o indeksie równym *n*. Próba dostępu do niego jest bardzo częstym błędem, zwanym **przekroczeniem indeksów** (ang. *subscript out of bounds*).

Poniższa linijka kodu spowodowałaby zatem błąd podczas działania programu i jego awaryjne zakończenie:

```
aKilkaLiczb[5] = 6;  // BŁĄD!!!
```

Pamiętaj więc, byś zwracał baczną uwagę na indeksy tablic, którymi operujesz.

Przekroczenie indeksów to jeden z przedstawicieli licznej rodziny błędów, noszących wspólne miano „pomyłek o jedynkę”. Większość z nich dotyczy właśnie tablic, inne można popełnić choćby przy pracy z liczbami pseudolosowymi: najwredniejszym jest chyba warunek w rodzaju `rand() % 10 == 10`, który nigdy nie może być spełniony (pomyśl, dlaczego[53]!).

Krytyczne spojrzenie na zaprezentowany kilka akapitów wyżej kawałek kodu może prowadzić do wniosku, że idea tablic nie ma większego sensu. Przecież równie dobrze możnaby zadeklarować 5 zmiennych i zająć się każdą z nich osobno - podobnie jak czynimy to teraz z elementami tablicy:

[53] Reszta z dzielenia przez `10` może być z nazwy równa jedynie liczbom `0`, `1`, ..., `8`, `9`, zatem nigdy nie zrówna się z samą dziesiątką. Programista chciał tu zapewne uzyskać wartość z przedziału `<1; 10>`, ale nie dodał jedynki do wyrażenia - czyli pomylił się o nią :)

```
int nLiczba1, nLiczba2, nLiczba3, nLiczba4, nLiczba5;

nLiczba1 = 1;
nLiczba2 = 2;
// itd.
```

Takie rozumowanie jest pozornie słuszne… ale na szczęście, tylko pozornie! :D Użycie pięciu instrukcji - po jednej dla każdego elementu tablicy - nie było bowiem najlepszym rozwiązaniem. O wiele bardziej naturalnym jest odpowiednia pętla `for`:

```
for (int i = 0; i < 5; ++i)      // drugim warunkiem może być też i <= 4
   aKilkaLiczb[i] = i + 1;
```

Jej zalety są oczywiste: niezależnie od tego, czy nasza tablica składa się z pięciu, pięciuset czy pięciu tysięcy elementów, przytoczona pętla jest w każdym przypadku niemal identyczna!
Tajemnica tego faktu tkwi rzecz jasna w indeksowaniu tablicy licznikiem pętli, `i`. Przyjmuje on odpowiednie wartości (od zera do rozmiaru tablicy minus jeden), które pozwalają zająć się całością tablicy przy pomocy **jednej** tylko instrukcji!
Taki manewr nie byłby możliwy, gdybyśmy używali tutaj pięciu zmiennych, zastępujących tablice. Ich „indeksy" (będące *de facto* częścią nazw) musiałyby być bowiem stałymi wartościami, wpisanymi bezpośrednio do kodu. Nie dałoby się zatem skorzystać z pętli `for` w podobny sposób, jak to uczyniliśmy w przypadku tablic.

Inicjalizacja tablicy

Kiedy w tak szczegółowy i szczególny sposób zajmujemy się tablicami, łatwo możemy zapomnieć, iż w gruncie rzeczy są to takie same zmienne, jak każde inne. Owszem, składają się z wielu pojedynczych elementów („podzmiennych"), ale nie przeszkadza to w wykonywaniu nań większości znanych nam operacji. Jedną z nich jest inicjalizacja.

Dzięki niej możemy chociażby deklarować tablice będące stałymi.

Tablicę możemy zainicjalizować w bardzo prosty sposób, unikając przy tym wielokrotnych przypisań (po jednym dla każdego elementu):

```
int aKilkaLiczb[5] = { 1, 2, 3, 4, 5 };
```

Kolejne wartości wpisujemy w nawiasie klamrowym, oddzielając je przecinkami. Zostaną one umieszczone w następujących po sobie elementach tablicy, poczynając od początku. Tak więc `aKilkaLiczb[0]` będzie miał wartość `1`, `aKilkaLiczb[1]` - `2`, itd. Uzyskamy identyczny efekt, jak w przypadku poprzednich pięciu przypisań.

Interesującą nowością w inicjalizacji tablic jest możliwość **pominięcia** ich rozmiaru:

```
std::string aSystemyOperacyjne[] = {"Windows", "Linux", "BeOS", "QNX"};
```

W takiej sytuacji kompilator „**domyśli się**" prawidłowej wielkości tablicy na podstawie ilości elementów, jaką wpisaliśmy wewnątrz nawiasów klamrowych (w tzw. **inicjalizatorze**). Tutaj będą to oczywiście cztery napisy.

Inicjalizacja jest więc całkiem dobrym sposobem na wstępne ustawienie wartości kolejnych elementów tablicy - szczególnie wtedy, gdy nie jest ich zbyt wiele i nie są one ze sobą jakoś związane. Dla dużych tablic nie jest to jednak efektywna metoda; w takich wypadkach lepiej użyć odpowiedniej pętli `for`.

Przykład wykorzystania tablicy

Wiemy już, jak teoretycznie wygląda praca z tablicami w języku C++, zatem naturalną koleją rzeczy będzie teraz uważne przyglądnięcie się odpowiedniemu przykładowi. Ten (spory :)) kawałek kodu wygląda następująco:

```
// Lotto - użycie prostej tablicy liczb

const unsigned ILOSC_LICZB = 6;
const int MAKSYMALNA_LICZBA = 49;

void main()
{
   // deklaracja i wyzerowanie tablicy liczb
   unsigned aLiczby[ILOSC_LICZB];
   for (int i = 0; i < ILOSC_LICZB; ++i)
         aLiczby[i] = 0;

   // losowanie liczb
   srand (static_cast<int>(time(NULL)));
   for (int i = 0; i < ILOSC_LICZB; )
   {
         // wylosowanie liczby
         aLiczby[i] = rand() % MAKSYMALNA_LICZBA + 1;

         // sprawdzenie, czy się ona nie powtarza
         bool bPowtarzaSie = false;
         for (int j = 0; j < i; ++j)
         {
               if (aLiczby[j] == aLiczby[i])
               {
                     bPowtarzaSie = true;
                     break;
               }
         }

         // jeżeli się nie powtarza, przechodzimy do następnej liczby
         if (!bPowtarzaSie) ++i;
   }

   // wyświetlamy wylosowane liczby
   std::cout << "Wyniki losowania:" << std::endl;
   for (int i = 0; i < ILOSC_LICZB; ++i)
         std::cout << aLiczby[i] << " ";

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

Huh, trzeba przyznać, iż z pewnością nie należy on do elementarnych :) Nie jesteś już jednak zupełnym nowicjuszem w sztuce programowania, więc zrozumienie go nie przysporzy ci wielkich kłopotów. Na początek spróbuj zobaczyć tę przykładową aplikację w działaniu:

Screen 30. Wysyłanie kuponów jest od dzisiaj zbędne ;-)

Nie potrzeba przenikliwości Sherlocka Holmesa, by wydedukować, że program ten dokonuje losowania zestawu liczb według zasad znanej powszechnie gry loteryjnej. Te reguły są determinowane przez dwie stałe, zadeklarowane na samym początku kodu:

```
const unsigned ILOSC_LICZB = 6;
const int MAKSYMALNA_LICZBA = 49;
```

Ich nazwy są na tyle znaczące, iż dokumentują się same. Wprowadzenie takich stałych ma też inne wyraźne zalety, o których wielokrotnie już wspominaliśmy. Ewentualna zmiana zasad losowania będzie ograniczała się jedynie do modyfikacji tychże dwóch linijek, mimo że te kluczowe wartości są wielokrotnie używane w całym programie.

Najważniejszą zmienną w naszym kodzie jest oczywiście tablica, która przechowuje wylosowane liczby. Deklarujemy i inicjalizujemy ją zaraz na wstępie funkcji `main()`:

```
unsigned aLiczby[ILOSC_LICZB];
for (int i = 0; i < ILOSC_LICZB; ++i)
   aLiczby[i] = 0;
```

Posługując się tutaj pętlą `for`, ustawiamy wszystkie jej elementy na wartość `0`. Zero jest dla nas neutralne, gdyż losowane liczby będą przecież wyłącznie dodatnie.

Identyczny efekt (wyzerowanie tablicy) można uzyskać stosując funkcję `memset()`, której deklaracja jest zawarta w nagłówku *memory.h*. Użylibyśmy jej w następujący sposób:

```
memset (aLiczby, 0, sizeof(aLiczby));
```

Analogiczny skutek spowodowałaby także specjalna funkcja `ZeroMemory()` z *windows.h*:

```
ZeroMemory (aLiczby, sizeof(aLiczby));
```

Nie użyłem tych funkcji w kodzie przykładu, gdyż wyjaśnienie ich działania wymaga wiedzy o wskaźnikach na zmienne, której jeszcze nie posiadasz. Chwilowo jesteśmy więc zdani na swojską pętlę :)

Po wyzerowaniu tablicy przeznaczonej na generowane liczby możemy przystąpić do właściwej czynności programu, czyli ich losowania. Rozpoczynamy je od niezbędnego wywołania funkcji `srand()`:

```
srand (static_cast<int>(time(NULL)));
```

Po dopełnieniu tej drobnej formalności możemy już zająć się po kolei każdą wartością, którą chcemy uzyskać. Znowuż czynimy to poprzez odpowiednią pętlę `for`:

```
for (int i = 0; i < ILOSC_LICZB; )
{
   // ...
}
```

Jak zwykle, przebiega ona po wszystkich elementach tablicy `aLiczby`. Pewną niespodzianką może być tu nieobecność ostatniej części tej instrukcji, którą jest zazwyczaj inkrementacja licznika. Jej brak spowodowany jest koniecznością sprawdzania, czy wylosowana już liczba **nie powtarza** się wśród wcześniej wygenerowanych. Z tego też powodu program będzie niekiedy zmuszony do kilkakrotnego „obrotu” pętli przy tej samej wartości licznika i losowania za każdym razem nowej liczby, aż do skutku.

Rzeczone losowane przebiega tradycyjną i znaną nam dobrze drogą:

```
aLiczby[i] = rand() % MAKSYMALNA_LICZBA + 1;
```

Uzyskana w ten sposób wartość jest zapisywana w tablicy aLiczby pod i-tym indeksem, abyśmy mogli ją później łatwo wyświetlić. W powyższym wyrażeniu obecna jest także stała, zadeklarowana wcześniej na początku programu.
Wspominałem już parę razy, że konieczna jest kontrola otrzymanej tą metodą wartości pod kątem jej niepowtarzalności. Musimy po prostu sprawdzać, czy nie wystąpiła już ona przy poprzednich losowaniach. Jeżeli istotnie tak się stało, to z pewnością znajdziemy ją we wcześniej „przerobionej” części tablicy. Niezbędne poszukiwania realizuje kolejny fragment listingu:

```
bool bPowtarzaSie = false;
for (int j = 0; j < i; ++j)
{
   if (aLiczby[j] == aLiczby[i])
   {
         bPowtarzaSie = true;
         break;
   }
}

if (!bPowtarzaSie) ++i;
```

Wprowadzamy tu najpierw pomocniczą zmienną (flagę) logiczną, zainicjalizowaną wstępnie wartością false (fałsz). Będzie ona niosła informację o tym, czy faktycznie mamy do czynienia z duplikatem którejś z wcześniejszych liczb.
Aby się o tym przekonać, musimy dokonać ponownego przeglądnięcia części tablicy. Robimy to poprzez, a jakże, kolejną pętlę for :) Aczkolwiek tym razem interesują nas wszystkie elementy tablicy występujące przed tym aktualnym, o indeksie i. Jako warunek pętli wpisujemy więc j < i (j jest licznikiem nowej pętli).
Koncentrując się na niuansach zagnieżdżonej instrukcji for nie zapominajmy, że jej celem jest znalezienie ewentualnego bliźniaka wylosowanej kilka wierszy wcześniej liczby. Zadanie to wykonujemy poprzez odpowiednie porównanie:

```
if (aLiczby[j] == aLiczby[i])
```

aLiczby[i] (i-ty element tablicy aLiczby) reprezentuje oczywiście liczbę, której szukamy; jak wiemy doskonale, uzyskaliśmy ją w sławetnym losowaniu :D Natomiast aLiczby[j] (j-ta wartość w tablicy) przy każdym kolejnym przebiegu pętli oznacza jeden z przeszukiwanych elementów. Jeżeli zatem wśród nich rzeczywiście jest wygenerowana, „aktualna” liczba, niniejszy warunek instrukcji if z pewnością ją wykryje. Co powinniśmy zrobić w takiej sytuacji? Otóż nic skomplikowanego - mianowicie, ustawiamy naszą zmienną logiczną na wartość true (prawda), a potem przerywamy pętlę for:

```
bPowtarzaSie = true;
break;
```

Jej dalsze działanie nie ma bowiem najmniejszego sensu, gdyż jeden duplikat liczby w zupełności wystarcza nam do szczęścia :)
W tym momencie jesteśmy już w posiadaniu arcyważnej informacji, który mówi nam, czy wartość wylosowana na samym początku cyklu głównej pętli jest istotnie unikatowa, czy też konieczne będzie ponowne jej wygenerowanie. Ową wiadomość przydałoby się teraz wykorzystać - robimy to w zaskakująco prosty sposób:

```
if (!bPowtarzaSie) ++i;
```

Jak widać, właśnie tutaj trafiła brakująca inkrementacja licznika pętli, i. Zatem odbywa się ona wtedy, kiedy uzyskana na początku liczba losowa spełnia nasz warunek

niepowtarzalności. W innym przypadku licznik **zachowuje** swą aktualną wartość, więc wówczas będzie przeprowadzona kolejna próba wygenerowania unikalnej liczby. Stanie się to w następnym cyklu pętli.
Inaczej mówiąc, jedynie fałszywość zmiennej `bPowtarzaSie` uprawnia pętlę `for` do zajęcia się dalszymi elementami tablicy. Inna sytuacja zmuszą ją bowiem do wykonania kolejnego cyklu na **tej samej wartości licznika** `i`, a więc także na **tym samym elemencie tablicy** wynikowej. Czyni to aż do otrzymania pożądanego rezultatu, czyli liczby różnej od wszystkich poprzednich.

Być może nasunęła ci się wątpliwość, czy takie kontrolowanie wylosowanej liczby jest aby na pewno konieczne. Skoro prawidłowo zainicjowaliśmy generator wartości losowych (przy pomocy `srand()`), to przecież nie powinien on robić nam świństw, którymi z pewnością byłyby powtórzenia wylosowywanych liczb. Jeżeli nawet istnieje jakaś szansa na otrzymanie duplikatu, to jest ona zapewne znikomo mała…
Otóż nic bardziej błędnego! Sama **potencjalna możliwość** wyniknięcia takiej sytuacji jest wystarczającym powodem, żeby dodać do programu zabezpieczający przed nią kod. Przecież nie chcielibyśmy, aby przyszły użytkownik (niekoniecznie tego programu, ale naszych aplikacji w ogóle) otrzymał produkt, który raz działa dobrze, a raz nie!
Inna sprawa, że prawdopodobieństwo wylosowania powtarzających się liczb nie jest tu wcale takie małe. Możesz spróbować się o tym przekonać[54]…

Na finiszu całego programu mamy jeszcze wyświetlanie uzyskanego pieczołowicie wyniku. Robimy to naturalnie przy pomocy adekwatnego `for`'a, który tym razem jest o wiele mniej skomplikowany w porównaniu z poprzednim :)
Ostatnia instrukcja, `getch();`, nie wymaga już nawet żadnego komentarza. Na niej też kończy się wykonywanie naszej aplikacji, a my możemy również zakończyć tutaj jej omawianie. I odetchnąć z ulgą ;)

Uff! To wcale nie było takie łatwe, prawda? Wszystko dlatego, że postawiony problem także nie należał do trywialnych. Analiza algorytmu, służącego do jego rozwiązania, powinna jednak bardziej przybliżyć ci sposób konstruowania kodu, realizującego **konkretne** zadanie.

Mamy oto przejrzysty i, mam nadzieję, zrozumiały przykład na wykorzystanie tablic w programowaniu. Przyglądając mu się dokładnie, mogłeś dobrze poznać zastosowanie tandemu **tablica + pętla `for`** do wykonywania dosyć skomplikowanych czynności na złożonych danych. Jeszcze nie raz użyjemy tego mechanizmu, więc z pewnością będziesz miał szansę na jego doskonałe opanowanie :)

Więcej wymiarów

Dotychczasowym przedmiotem naszego zainteresowania były tablice **jednowymiarowe**, czyli takie, których poszczególne elementy są identyfikowane poprzez **jeden** indeks. Takie struktury nie zawsze są wystarczające. Pomyślmy na przykład o szachownicy, planszy do gry w statki czy mapach w grach strategicznych. Wszystkie te twory wymagają większej liczby wymiarów i nie dają się przedstawić w postaci zwykłej, ponumerowanej listy.

[54] Wyliczenie jest bardzo proste. Załóżmy, że losujemy n liczb, z których największa może być równa a. Wtedy pierwsze losowanie nie może rzecz jasna skutkować duplikatem. W drugim jest na to szansa równa $1/a$ (gdyż mamy już jedną liczbę), w trzecim - $2/a$ (bo mamy już dwie liczby), itd. Dla n liczb całościowe prawdopodobieństwo wynosi zatem $(1 + 2 + 3 + \ldots + n\text{-}1)/a$, czyli $n(n - 1)/2a$.
U nas $n = 6$, zaś $a = 49$, więc mamy $6(6 - 1)/(2*49) \approx 30{,}6\%$ szansy na otrzymanie zestawu liczb, w którym przynajmniej jedna się powtarza. Gdybyśmy nie umieścili kodu sprawdzającego, wtedy przeciętnie co czwarte uruchomienie programu dawałoby nieprawidłowe wyniki. Byłaby to ewidentna niedoróbka.

Naturalnie, tablice wielowymiarowe mogłyby być z powodzeniem symulowane poprzez ich jednowymiarowe odpowiedniki oraz formuły służące do przeliczania indeksów. Trudno jednak uznać to za wygodne rozwiązanie. Dlatego też C++ radzi sobie z tablicami wielowymiarowymi w znacznie prostszy i bardziej przyjazny sposób. Warto więc przyjrzeć się temu wielkiemu dobrodziejstwu ;)

Deklaracja i inicjalizacja

Domyślasz się może, iż aby zadeklarować tablicę wielowymiarową, należy podać więcej niż jedną liczbę określającą jej rozmiar. Rzeczywiście tak jest:

```
int aTablica[4][5];
```

Linijka powyższa tworzy nam dwuwymiarową tablicę o wymiarach `4` na `5`, zawierającą elementy typu `int`. Możemy ją sobie wyobrazić w sposób podobny do tego:

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

Schemat 8. Wyobrażenie tablicy dwuwymiarowej 4×5

Widać więc, że początkowa analogia do szachownicy była całkiem na miejscu :)

Nasza dziewicza tablica wymaga teraz nadania wstępnych wartości swoim elementom. Jak pamiętamy, przy korzystaniu z jej jednowymiarowych kuzynów intensywnie używaliśmy do tego odpowiednich pętli `for`. Nic nie stoi na przeszkodzie, aby podobnie postąpić i w tym przypadku:

```
for (int i = 0; i < 4; ++i)
   for (int j = 0; j < 5; ++j)
         aTablica[i][j] = i + j;
```

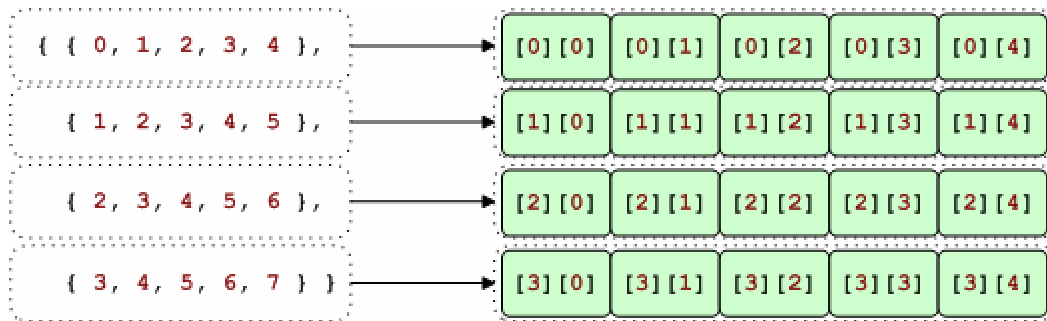
Teraz jednak mamy **dwa** wymiary tablicy, zatem musimy zastosować **dwie** zagnieżdżone pętle. Ta bardziej zewnętrzna przebiega nam po czterech kolejnych **wierszach** tablicy, natomiast wewnętrzna zajmuje się każdym z pięciu **elementów** wybranego wcześniej wiersza. Ostatecznie, przy każdym cyklu zagnieżdżonej pętli liczniki `i` oraz `j` mają odpowiednie wartości, abyśmy mogli za ich pomocą uzyskać dostęp do każdego z dwudziestu (`4 * 5`) elementów tablicy.

Znamy wszakże jeszcze inny środek, służący do wstępnego ustawiania zmiennych - chodzi oczywiście o inicjalizację. Zobaczyliśmy niedawno, że możliwe jest zaprzęgnięcie jej do pracy także przy tablicach jednowymiarowych. Czy będziemy mogli z niej skorzystać również teraz, gdy dodaliśmy do nich następne wymiary?...
Jak to zwykle w C++ bywa, odpowiedź jest pozytywna :) Inicjalizacja tablicy dwuwymiarowej wygląda bowiem następująco:

```
int aTablica[4][5] = { { 0, 1, 2, 3, 4 },
                       { 1, 2, 3, 4, 5 },
```

```
                             { 2, 3, 4, 5, 6 },
                             { 3, 4, 5, 6, 7 } };
```

Opiera się ona na tej samej zasadzie, co analogiczna operacja dla tablic jednowymiarowych: kolejne wartości oddzielamy przecinkami i umieszczamy w nawiasach klamrowych. Tutaj są to cztery **wiersze** naszej tabeli.
Jednak każdy z nich sam jest niejako odrębną tablicą! W taki też sposób go traktujemy: ostateczne, liczbowe wartości elementów podajemy albowiem wewnątrz **zagnieżdżonych** nawiasów klamrowych. Dla przejrzystości rozmieszczamy je w oddzielnych linijkach kodu, co sprawia, że całość łudząco przypomina wyobrażenie tablicy dwuwymiarowej jako prostokąta podzielonego na pola.



Schemat 9. Inicjalizacja tablicy dwuwymiarowej 4×5

Otrzymany efekt jest zresztą taki sam, jak ten osiągnięty przez dwie wcześniejsze, zagnieżdżone pętle.

Warto również wiedzieć, że inicjalizując tablicę wielowymiarową możemy pominąć wielkość pierwszego wymiaru:

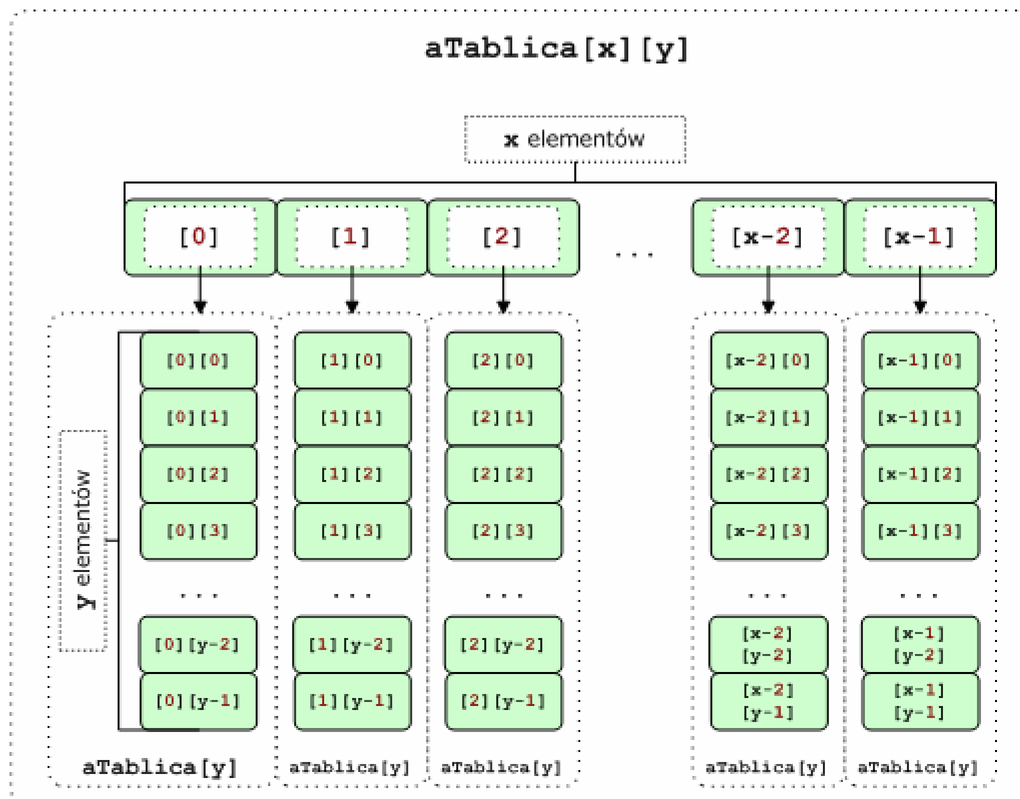
```
int aTablica[][5] = { { 0, 1, 2, 3, 4 },
                      { 1, 2, 3, 4, 5 },
                      { 2, 3, 4, 5, 6 },
                      { 3, 4, 5, 6, 7 } };
```

Zostanie on wtedy wywnioskowany z inicjalizatora.

Tablice w tablicy

Sposób obsługi tablic wielowymiarowych w C++ różni się zasadniczo od podobnych mechanizmów w wielu innych językach. Tutaj bowiem nie są one traktowane wyjątkowo, jako byty odrębne od swoich jednowymiarowych towarzyszy. Powoduje to, że w C++ dozwolone są pewne operacje, na które nie pozwala większość pozostałych języków programowania.

Dzieje się to za przyczyną dość ciekawego pomysłu potraktowania tablic wielowymiarowych jako zwykłych **tablic jednowymiarowych**, których elementami są… **inne tablice**! Brzmi to trochę topornie, ale w istocie nie jest takie trudne, jak być może wygląda :)
Najprostszy przykład tego faktu, z jakim mieliśmy już do czynienia, to konstrukcja dwuwymiarowa. Z punktu widzenia C++ jest ona jednowymiarową **tablicą swoich wierszy**; zwróciliśmy zresztą na to uwagę, dokonując jej inicjalizacji. Każdy z owych wierszy jest zaś także jednowymiarową tablicą, tym razem składającą się już ze zwykłych, skalarnych elementów.
Zjawisko to (oraz kilka innych ;D) nieźle obrazuje poniższy diagram:



Schemat 10. Przedstawienie tablicy dwuwymiarowej jako tablicy tablic

Uogólniając, możemy stwierdzić, iż:

Każda tablica *n*-wymiarowa składa się z odpowiedniej liczby tablic (*n*-1)-wymiarowych.

Przykładowo, dla trzech wymiarów będziemy mieli tablicę, składającą się z tablic dwuwymiarowych, które z kolei zbudowane są z jednowymiarowych, a te dopiero z pojedynczych skalarów. Nietrudne, prawda? ;)

Zadajesz sobie pewnie pytanie: cóż z tego? Czy ma to jakieś praktyczne znaczenie i zastosowanie w programowaniu?...
Pospieszam z odpowiedzią, brzmiącą jak zawsze „ależ oczywiście!” :)) Ujęcie tablic w takim stylu pozwala na ciekawą operację **wybrania jednego z wymiarów** i przypisania go do innej, pasującej tablicy. Wygląda to mniej więcej tak:

```
// zadeklarowanie tablicy trój- i dwuwymiarowej
int aTablica3D[2][2][2] = { { { 1, 2 },
                              { 2, 3 } },
                            { { 3, 4 },
                              { 4, 5 } } };
int aTablica2D[2][2];

// przypisanie drugiej "płaszczyzny" tablicy aTablica3D do aTablica2D
aTablica2D = aTablica3D[1];

// aTablica2D zawiera teraz liczby: { { 3, 4 }, { 4, 5 } }
```

Przykład ten ma w zasadzie charakter ciekawostki, lecz przyjrzenie mu się z pewnością nikomu nie zaszkodzi :D

Nieco praktyczniejsze byłoby odwołanie do części tablicy - tak, żeby możliwa była jej zmiana niezależnie od całości (np. przekazanie do funkcji). Takie działanie wymaga jednak poznania wskaźników, a to stanie się dopiero w rozdziale 8.

***

Poznaliśmy właśnie tablice jako sposób na tworzenie złożonych struktur, składających się z wielu elementów. Ułatwiają one (lub wręcz umożliwiają) posługiwanie się złożonymi danymi, jakich nie brak we współczesnych aplikacjach. Znajomość zasad wykorzystywania tablic z pewnością zatem zaprocentuje w przyszłości :)

Także w tym przypadku niezawodnym źródłem uzupełniających informacji jest MSDN.

Nowe typy danych

Wachlarz dostępnych w C++ typów wbudowanych jest, jak wiemy, niezwykle bogaty. W połączeniu z możliwością fuzji wielu pojedynczych zmiennych do postaci wygodnych w użyciu tablic, daje nam to szerokie pole do popisu przy konstruowaniu własnych sposobów na przechowywanie danych.

Nabyte już doświadczenie oraz tytuł niniejszego podrozdziału sugeruje jednak, iż nie jest to wcale kres potencjału używanego przez nas języka. Przeciwnie: C++ oferuje nam możliwość tworzenia swoich **własnych typów** zmiennych, odpowiadających bardziej konkretnym potrzebom niż zwykłe liczby czy napisy.
Nie chodzi tu wcale o znaną i prostą instrukcję `typedef`, która umie jedynie produkować nowe nazwy dla już istniejących typów. Mam bowiem na myśli znacznie potężniejsze narzędzia, udostępniające dużo większe możliwości w tym zakresie.

Czy znaczy to również, że są one trudne do opanowania? Według mnie siedzący tutaj diabeł wcale nie jest taki straszny, jakim go malują ;D Absolutnie więc nie ma się czego bać!

Wyliczania nadszedł czas

Pierwszym z owych narzędzi, z którymi się zapoznamy, będą **typy wyliczeniowe** (ang. *enumerated types*). Ujrzymy ich możliwe zastosowania oraz techniki użytkowania, a rozpoczniemy od przykładu z życia wziętego :)

Przydatność praktyczna

W praktyce często zdarza się sytuacja, kiedy chcemy ograniczyć możliwy zbiór wartości zmiennej do kilku(nastu/dziesięciu) ściśle ustalonych elementów. Jeżeli, przykładowo, tworzylibyśmy grę, w której pozwalamy graczowi jedynie na ruch w czterech kierunkach (góra, dół, lewo, prawo), z pewnością musielibyśmy przechowywać w jakiś sposób jego wybór. Służąca do tego zmienna przyjmowałaby więc jedną z czterech określonych wartości.
Jak możnaby osiągnąć taki efekt? Jednym z rozwiązań jest zastosowanie stałych, na przykład w taki sposób:

```
const int KIERUNEK_GORA = 1;
const int KIERUNEK_DOL = 2;
const int KIERUNEK_LEWO = 3;
const int KIERUNEK_PRAWO = 4;

int nKierunek;
```

```
nKierunek = PobierzWybranyPrzezGraczaKierunek();

switch (nKierunek)
{
   case KIERUNEK_GORA:     // porusz graczem w górę
   case KIERUNEK_DOL:      // porusz graczem w dół
   case KIERUNEK_LEWO:     // porusz graczem w lewo
   case KIERUNEK_PRAWO:    // porusz graczem w prawo
   default:                // a to co za kierunek? :)
}
```

Przy swoim obecnym stanie koderskiej wiedzy mógłbyś z powodzeniem użyć tego sposobu. Skoro jednak prezentujemy go w miejscu, z którego zaraz przejdziemy do omawiania nowych zagadnień, nie jest on pewnie zbyt dobry :)
Najpoważniejszym chyba mankamentem jest zupełna nieświadomość kompilatora co do specjalnego znaczenia zmiennej `nKierunek`. Traktuje ją więc identycznie, jak każdą inną liczbę całkowitą, pozwalając choćby na przypisanie podobne do tego:

```
nKierunek = 10;
```

Z punktu widzenia składni C++ jest ono całkowicie poprawne, ale dla nas byłby to niewątpliwy błąd. `10` nie oznacza bowiem żadnego z czterech ustalonych kierunków, więc wartość ta nie miałaby w naszym programie najmniejszego sensu!

Jak zatem podejść do tego problemu? Najlepszym wyjściem jest zdefiniowanie nowego typu danych, który będzie pozwalał na przechowywanie tylko kilku podanych wartości. Czynimy to w sposób następujący[55]:

```
enum DIRECTION { DIR_UP, DIR_DOWN, DIR_LEFT, DIR_RIGHT };
```

Tak oto stworzyliśmy typ wyliczeniowy zwany `DIRECTION`. Zmienne, które zadeklarujemy jako należące do tegoż typu, będą mogły przyjmować **jedynie** wartości **wpisane** przez nas w jego **definicji**. Są to `DIR_UP`, `DIR_DOWN`, `DIR_LEFT` i `DIR_RIGHT`, odpowiadające umówionym kierunkom. Pełnią one funkcję stałych - z tą różnicą, że nie musimy deklarować ich liczbowych wartości (gdyż i tak używać będziemy jedynie tych symbolicznych nazw).

Mamy więc nowy typ danych, wypadałoby zatem skorzystać z niego i zadeklarować jakąś zmienną:

```
DIRECTION Kierunek = PobierzWybranyPrzezGraczaKierunek();

switch (Kierunek)
{
   case DIR_UP:      // ...
   case DIR_DOWN:    // ...
   // itd.
}
```

Deklaracja zmiennej należącej do naszego własnego typu nie różni się w widoczny sposób od podobnego działania podejmowanego dla typów wbudowanych. Możemy również dokonać jej inicjalizacji, co też od razu czynimy.

[55] Nowe typy danych będę nazywał po angielsku, aby odróżnić je od zmiennych czy funkcji.

Kod ten będzie poprawny oczywiście tylko wtedy, gdy funkcja `PobierzWybranyPrzezGraczaKierunek()` będzie zwracała wartość będącą także typu `DIRECTION`.

Wszelkie wątpliwości powinna rozwiać instrukcja `switch`. Widać wyraźnie, że użyto jej w identyczny sposób jak wtedy, gdy korzystano jeszcze ze zwykłych stałych, deklarowanych oddzielnie.

Na czym więc polega różnica? Otóż tym razem niemożliwe jest przypisanie w rodzaju:

```
Kierunek = 20;
```

Kompilator nie pozwoli na nie, gdyż zmienna `Kierunek` podlega ograniczeniom swego typu `DIRECTION`. Określając go, ustaliliśmy, że może on reprezentować **wyłącznie** jedną z czterech podanych wartości, a `20` niewątpliwie nie jest którąś z nich :)
Tak więc teraz bezmyślny program kompilujący jest po naszej stronie i pomaga nam jak najwcześniej wyłapywać błędy związane z nieprawidłowymi wartościami niektórych zmiennych.

Definiowanie typu wyliczeniowego

Nie od rzeczy będzie teraz przyjrzenie się kawałkowi kodu, który wprowadza nam nowy typ wyliczeniowy. Oto i jego składnia:

```
enum nazwa_typu { stała_1 [ = wartość_1 ],
                  stała_2 [ = wartość_2 ],
                  stała_3 [ = wartość_3 ],
                  ...
                  stała_n [ = wartość_n ] };
```

Słowo kluczowe `enum` (ang. *enumerate* - wyliczać) pełni rolę informującą: mówi, zarówno nam, jak i kompilatorowi, iż mamy tu do czynienia z definicją typu wyliczeniowego. Nazwę, którą chcemy nadać owemu typowi, piszemy zaraz za tym słowem; przyjęło się, aby używać do tego wielkich liter alfabetu.

Potem następuje częsty element w kodzie C++, czyli nawiasy klamrowe. Wewnątrz nich umieszczamy tym razem **listę stałych** - dozwolonych wartości typu wyliczeniowego. Jedynie one będą dopuszczone przez kompilator do przechowywania przez zmienne należące do definiowanego typu.
Tutaj również zaleca się, tak jak w przypadku zwykłych stałych (tworzonych poprzez `const`), używanie wielkich liter. Dodatkowo, dobrze jest dodać do każdej nazwy odpowiedni przedrostek, powstały z nazwy typu, na przykład:

```
// przykładowy typ określający poziom trudności jakiejś gry
enum DIFFICULTY { DIF_EASY, DIF_MEDIUM, DIF_HARD };
```

Widać to było także w przykładowym typie `DIRECTION`.

Nie zapominajmy o średniku na końcu definicji typu wyliczeniowego!

Warto wiedzieć, że stałe, które wprowadzamy w definicji typu wyliczeniowego, reprezentują liczby całkowite i tak też są przez kompilator traktowane. Każdej z nich nadaje on kolejną wartość, poczynając zazwyczaj od zera.
Najczęściej nie przejmujemy się, jakie wartości odpowiadają poszczególnym stałym. Czasem jednak należy mieć to na uwadze - na przykład wtedy, gdy planujemy współpracę naszego typu z jakimiś zewnętrznymi bibliotekami. W takiej sytuacji możemy

wyraźnie określić, jakie liczby są reprezentowane przez nasze stałe. Robimy to, wpisując wartość po znaku = i nazwie stałej.
Przykładowo, w zaprezentowanym na początku typie `DIRECTION` moglibyśmy przypisać każdemu wariantowi kod liczbowy odpowiedniego klawisza strzałki:

```
enum DIRECTION { DIR_UP    = 38,
                 DIR_DOWN  = 40,
                 DIR_LEFT  = 37,
                 DIR_RIGHT = 39 };
```

Nie trzeba jednak wyraźnie określać wartości dla wszystkich stałych; możliwe jest ich sprecyzowanie tylko dla kilku. Dla pozostałych kompilator dobierze wtedy kolejne liczby, poczynając od tych narzuconych, tzn. zrobi coś takiego:

```
enum MYENUM { ME_ONE,                   // 0
              ME_TWO    = 12,           // 12
              ME_THREE,                 // 13
              ME_FOUR,                  // 14
              ME_FIVE   = 26,           // 26
              ME_SIX,                   // 27
              ME_SEVEN };               // 28
```

Zazwyczaj nie trzeba o tym pamiętać, bo lepiej jest albo całkowicie zostawić przydzielanie wartości w gestii kompilatora, albo samemu dobrać je dla wszystkich stałych i nie utrudniać sobie życia ;)

Użycie typu wyliczeniowego

Typy wyliczeniowe zalicza się do typów liczbowych, podobnie jak `int` czy `unsigned`. Mimo to **nie jest możliwe** bezpośrednie przypisanie do zmiennej takiego typu liczby zapisanej wprost. Kompilator nie przepuści więc instrukcji podobnej do tej:

```
enum DECISION { YES = 1, NO = 0, DONT_KNOW = -1 };
DECISION Decyzja = 0;
```

Zrobi tak nawet pomimo faktu, iż `0` odpowiada tutaj jednej ze stałych typu `DECISION`. C++ dba bowiem, aby typów `enum` używać zgodnie z ich przeznaczeniem, a nie jako zamienników dla zmiennych liczbowych. Powoduje to, że:

Do zmiennych wyliczeniowych możemy przypisywać **wyłącznie** odpowiadające im stałe. Niemożliwe jest nadanie im „zwykłych” wartości liczbowych.

Jeżeli jednak koniecznie potrzebujemy podobnego przypisania (bo np. odczytaliśmy liczbę z pliku lub uzyskaliśmy ją za pomocą jakiejś zewnętrznej funkcji), możemy salwować się rzutowaniem przy pomocy `static_cast`:

```
// zakładamy, że OdczytajWartosc() zwraca liczbę typu int lub podobną
Decyzja = static_cast<DECISION>(OdczytajWartosc());
```

Pamiętajmy aczkolwiek, żeby w zwykłych sytuacjach używać zdefiniowanych stałych. Inaczej całkowicie wypaczalibyśmy ideę typów wyliczeniowych.

Zastosowania

Ewentualni fani programów przykładowych mogą czuć się zawiedzeni, gdyż nie zaprezentuję żadnego krótkiego, kilkunastolinijkowego, dobitnego kodu obrazującego wykorzystanie typów wyliczeniowych w praktyce. Powód jest dość prosty: taki przykład miałby złożoność i celowość porównywalną do banalnych aplikacji dodających dwie liczby,

z którymi stykaliśmy się na początku kursu. Zamiast tego pomówmy lepiej o zastosowaniach opisywanych typów w konstruowaniu „normalnych”, przydatnych programów - także gier.

Do czego więc mogą przydać się typy wyliczeniowe? Tak naprawdę sposobów na ich konkretne użycie jest więcej niż ziaren piasku na pustyni; równie dobrze moglibyśmy, zadać pytanie w rodzaju „Jakie zastosowanie ma instrukcja `if`?” :) Wszystko bowiem zależy od postawionego problemu oraz samego programisty. Istnieje jednak co najmniej kilka ogólnych sytuacji, w których skorzystanie z typów wyliczeniowych jest wręcz naturalne:

- **Przechowywanie informacji o stanie jakiegoś obiektu czy zjawiska.** Przykładowo, jeżeli tworzymy grę przygodową, możemy wprowadzić nowy typ określający aktualnie wykonywaną przez gracza czynność: chodzenie, rozmowa, walka itd. Stosując przy tym instrukcję `switch` będziemy mogli w każdej klatce podejmować odpowiednie kroki sterujące konwersacją czy wymianą ciosów. Inny przykład to choćby odtwarzacz muzyczny. Wiadomo, że może on w danej chwili zajmować się odgrywaniem jakiegoś pliku, znajdować się w stanie pauzy czy też nie mieć wczytanego żadnego utworu i czekać na polecenia użytkownika. Te możliwe stany są dobrym materiałem na typ wyliczeniowy.

Wszystkie te i podobne sytuacje, z którymi można sobie radzić przy pomocy `enum`’ów, są przypadkami tzw. automatów o skończonej liczbie stanów (ang. *finite state machine*). Pojęcie to ma szczególne zastosowanie przy programowaniu sztucznej inteligencji, zatem jako (przyszły) programista gier będziesz się z nim czasem spotykał.

- **Ustawianie parametrów o ściśle określonym zbiorze wartości.** Był już tu przytaczany dobry przykład na wykorzystanie typów wyliczeniowych właśnie w tym celu. Jest to oczywiście kwestia poziomu trudności jakiejś gry; zapisanie wyboru użytkownika wydaje się najbardziej naturalne właśnie przy użyciu zmiennej wyliczeniowej.
  Dobrym reprezentantem tej grupy zastosowań może być również sposób wyrównywania akapitu w edytorach tekstu. Ustawienia: „do lewej”, do prawej”, „do środka” czy „wyjustowanie” są przecież świetnym materiałem na odpowiedni `enum`.
- **Przekazywanie jednoznacznych komunikatów w ramach aplikacji.** Nie tak dawno temu poznaliśmy typ `bool`, który może być używany między innymi do informowania o powodzeniu lub niepowodzeniu jakiejś operacji (zazwyczaj wykonywanej przez osobną funkcję). Taka czarno-biała informacja jest jednak mało użyteczna - w końcu jeżeli wystąpił jakiś błąd, to wypadałoby wiedzieć o nim coś więcej.
  Tutaj z pomocą przychodzą typy wyliczeniowe. Możemy bowiem zdefiniować sobie taki, który posłuży nam do identyfikowania ewentualnych błędów. Określając odpowiednie stałe dla braku pamięci, miejsca na dysku, nieistnienia pliku i innych czynników decydujących o niepowodzeniu pewnych działań, będziemy mogli je łatwo rozróżniać i raczyć użytkownika odpowiednimi komunikatami.

To tylko niektóre z licznych metod wykorzystywania typów wyliczeniowych w programowaniu. W miarę rozwoju swoich umiejętności sam odkryjesz dla nich mnóstwo specyficznych zastosowań i będziesz często z nich korzystał w pisanych kodach.

Upewnij się zatem, że dobrze rozumiesz, na czym one polegają i jak wygląda ich użycie w C++. To z pewnością sowicie zaprocentuje w przyszłości.
A kiedy uznasz, iż jesteś już gotowy, będziemy mogli przejść dalej :)

Kompleksowe typy

Tablice, opisane na początku tego rozdziału, nie są jedynym sposobem na modelowanie złożonych danych. Chociaż przydają się wtedy, gdy informacje mają jednorodną postać zestawu identycznych elementów, istnieje wiele sytuacji, w których potrzebne są inne rozwiązania…

Weźmy chociażby banalny, zdawałoby się, przykład książki adresowej. Na pierwszy rzut oka jest ona idealnym materiałem na prostą tablicę, której elementami byłyby jej kolejne pozycje - adresy.
Zauważmy jednak, że sama taka pojedyncza pozycja nie daje się sensownie przedstawić w postaci jednej zmiennej. Dane dotyczące jakiejś osoby obejmują przecież jej imię, nazwisko, ewentualnie pseudonim, adres e-mail, miejsce zamieszkania, telefon… Jest to przynajmniej kilka elementarnych informacji, z których każda wymagałaby oddzielnej zmiennej.

Podobnych przypadków jest w programowaniu mnóstwo i dlatego też dzisiejsze języki posiadają odpowiednie mechanizmy, pozwalające na wygodne przetwarzanie informacji o budowie hierarchicznej. Domyślasz się zapewne, że teraz właśnie rzucimy okiem na ofertę C++ w tym zakresie :)

Typy strukturalne i ich definiowanie

Wróćmy więc do naszego problemu książki adresowej, albo raczej listy kontaktów - najlepiej internetowych. Każda jej pozycja mogłaby się składać z takich oto trzech elementów:

- nicka tudzież imienia i nazwiska danej osoby
- jej adresu e-mail
- numeru identyfikacyjnego w jakimś komunikatorze internetowym

Na przechowywanie tychże informacji potrzebujemy zatem dwóch łańcuchów znaków (po jednym na nick i adres) oraz jednej liczby całkowitej. Znamy oczywiście odpowiadające tym rodzajom danych typy zmiennych w C++: są to rzecz jasna `std::string` oraz `int`. Możemy więc użyć ich do utworzenia nowego, **złożonego** typu, reprezentującego w całości pojedynczy kontakt:

```
struct CONTACT
{
   std::string strNick;
   std::string strEmail;
   int nNumerIM;
};
```

W ten właśnie sposób zdefiniowaliśmy **typ strukturalny**.

Typy strukturalne (zwane też w skrócie strukturami[56]) to zestawy kilku zmiennych, należących do innych typów, z których każda posiada swoją własną i **unikalną nazwę**. Owe „podzmienne” nazywamy **polami** struktury.

Nasz nowonarodzony typ strukturalny składa się zatem z trzech pól, zaś każde z nich przechowuje jedynie **elementarną** informację. Zestawione razem reprezentują jednak **złożoną** daną o jakiejś osobie.

---

[56] Zazwyczaj strukturami nazywamy już konkretne zmienne; u nas byłyby to więc rzeczywiste dane kontaktowe jakiejś osoby (czyli zmienne należące do zdefiniowanego właśnie typu `CONTACT`). Czasem jednak pojęć „typ strukturalny” i „struktura” używa się zamiennie, a ich szczegółowe znaczenie zależy od kontekstu.

Struktury w akcji

Nie zapominajmy, że zdefiniowane przed chwilą „coś" o nazwie `CONTACT` jest nowym typem, a więc możemy skorzystać z niego tak samo, jak z innych typów w języku C++ (wbudowanych lub poznanych niedawno `enum`'ów). Zadeklarujmy więc przy jego użyciu jakąś przykładową zmienną:

```
CONTACT Kontakt;
```

Logiczne byłoby teraz nadanie jej pewnej wartości… Pamiętamy jednak, że powyższy `Kontakt` to tak naprawdę trzy zmienne w jednym (coś jak szampon przeciwłupieżowy ;D). Niemożliwe jest zatem przypisanie mu zwykłej, „pojedynczej" wartości, właściwej typom skalarnym.
Możemy za to zająć się osobno każdym z jego pól. Są one znanymi nam bardzo dobrze tworami programistycznymi (napisem i liczbą), więc nie będziemy mieli z nimi najmniejszych kłopotów. Cóż zatem zrobić, aby się do nich dobrać?…

Skorzystamy ze specjalnego **operatora wyłuskania**, będącego zwykłą **kropką** (`.`). Pozwala on między innymi na uzyskanie dostępu do określonego pola w strukturze. Użycie go jest bardzo proste i dobrze widoczne na poniższym przykładzie:

```
// wypełnienie struktury danymi
Kontakt.strNick = "Hakier";
Kontakt.strEmail = "gigahaxxor@abc.pl";
Kontakt.nNumerIM = 192837465;
```

Postawienie kropki po nazwie struktury umożliwia nam niejako „wejście w jej głąb". W dobrych środowiskach programistycznych wyświetlana jest nawet lista wszystkich jej pól, jakby na potwierdzenie tego faktu oraz ułatwienie pisania dalszego kodu. Po kropce wprowadzamy więc nazwę pola, do którego chcemy się odwołać.
Wykonawszy ten prosty zabieg możemy zrobić ze wskazanym polem wszystko, co się nam żywnie podoba. W przykładzie powyżej czynimy doń zwykłe przypisanie wartości, lecz równie dobrze mogłoby to być jej odczytanie, użycie w wyrażeniu, przekazanie do funkcji, itp. Nie ma bowiem żadnej praktycznej różnicy w korzystaniu z pola struktury i ze zwykłej zmiennej tego samego typu - oczywiście poza faktem, iż to pierwsze jest tylko częścią większej całości.
Sądzę, że wszystko to powinno być dla ciebie w miarę jasne :)

Co uważniejsi czytelnicy (czyli pewnie zdecydowana większość ;D) być może zauważyli, iż nie jest to nasze pierwsze spotkanie z kropką w C++. Gdy zajmowaliśmy się dokładniej łańcuchami znaków, używaliśmy formułki `napis.length()` do pobrania długości tekstu. Czy znaczy to, że typ `std::string` również należy do strukturalnych?… Cóż, sprawa jest generalnie dosyć złożona, jednak częściowo wyjaśni się już w następnym rozdziale. Na razie wiedz, że cel użycia operatora wyłuskania był tam podobny do aktualnie omawianego (czyli „wejścia w środek" zmiennej), chociaż wtedy nie chodziło nam wcale o odczytanie wartości jakiegoś pola. Sugerują to zresztą nawiasy wieńczące wyrażenie… Pozwól jednak, abym chwilowo z braku czasu i miejsca nie zajmował się bliżej tym zagadnieniem. Jak już nadmieniłem, wrócimy do niego całkiem niedługo, zatem uzbrój się w cierpliwość :)

Spoglądając krytycznym okiem na trzy linijki kodu, które wykonują przypisania wartości do kolejnych pól struktury, możemy nabrać pewnych wątpliwości, czy aby składnia C++ jest rzeczywiście taka oszczędna, jaką się zdaje. Przecież wyraźnie widać, iż musieliśmy tutaj za w każdym wierszu wpisywać nieszczęsną nazwę struktury, czyli `Kontakt`! Nie dałoby się czegoś z tym zrobić?
Kilka języków, w tym np. Delphi i Visual Basic, posiada bloki **`with`**, które odciążają nieco

palce programisty i zezwalają na pisanie jedynie nazw pól struktur. Jakkolwiek jest to niewątpliwie wygodne, to czasem powoduje dość nieoczekiwane i niełatwe do wykrycia błędy logiczne. Wydaje się, że brak tego rodzaju instrukcji w C++ jest raczej rozsądnym skutkiem bilansu zysków i strat, co jednak nie przeszkadza mi osobiście uważać tego za pewien feler :D

Istnieje jeszcze jedna droga nadania początkowych wartości polom struktury, a jest nią naturalnie znana już szeroko inicjalizacja :) Ponieważ podobnie jak w przypadku tablic mamy tutaj do czynienia ze złożonymi zmiennymi, należy tedy posłużyć się odpowiednią formą inicjalizatora - taką, jak podana poniżej:

```
// inicjalizacja struktury
CONTACT Kontakt = { "MasterDisaster", "md1337@ajajaj.com.pl", 3141592 };
```

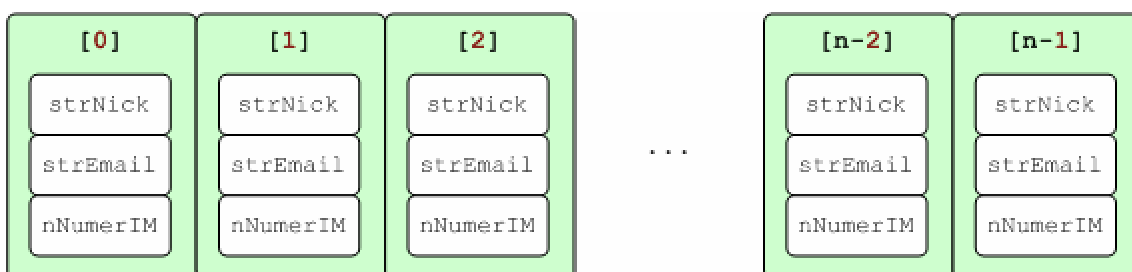
Używamy więc w znajomy sposób nawiasów klamrowych, umieszczając wewnątrz nich wyrażenia, które mają być przypisane kolejnym polom struktury. Należy przy tym pamiętać, by zachować **taki sam porządek pól**, jaki został określony w definicji typu strukturalnego. Inaczej możemy spodziewać się niespodziewanych błędów :)

Kolejność pól w definicji typu strukturalnego oraz w inicjalizacji należącej doń struktury musi być **identyczna**.

Uff, zdaje się, że w ferworze poznawania szczegółowych aspektów struktur zapomnieliśmy już całkiem o naszym pierwotnym zamyśle. Przypominam więc, iż było nim stworzenie elektronicznej wersji notesu z adresami, czyli po prostu listy internetowych kontaktów.
Nabyta wiedza nie pójdzie jednak na marne, gdyż teraz potrafimy już z łatwością wymyślić stosowne rozwiązanie pierwotnego problemu. Zasadniczą listą będzie po prostu odpowiednia **tablica struktur**:

```
const unsigned LICZBA_KONTAKTOW = 100;
CONTACT aKontakty[LICZBA_KONTAKTOW];
```

Jej elementami staną się dane poszczególnych osób zapisanych w naszej książce adresowej. Zestawione w jednowymiarową tablicę będą dokładnie tym, o co nam od początku chodziło :)



Schemat 11. Obrazowy model tablicy struktur

Metody obsługi takiej tablicy nie różnią się wiele od porównywalnych sposobów dla tablic składających się ze „zwykłych” zmiennych. Możemy więc łatwo napisać przykładową, prostą funkcję, która wyszukuje osobę o danym nicku:

```
int WyszukajKontakt(std::string strNick)
{
   // przebiegnięcie po całej tablicy kontaktów przy pomocy pętli for
   for (unsigned i = 0; i < LICZBA_KONTAKTOW; ++i)
         // porównywanie nicku każdej osoby z szukanym
```

```
            if (aKontakty[i].strNick == strNick)
                  // zwrócenie indeksu pasującej osoby
                  return i;

      // ewentualnie, jeśli nic nie znaleziono, zwracamy -1
      return -1;
}
```

Zwróćmy w niej szczególną uwagę na wyrażenie, poprzez które pobieramy pseudonimy kolejnych osób na naszej liście. Jest nim:

```
aKontakty[i].strNick
```

W zasadzie nie powinno być ono zaskoczeniem. Jak wiemy doskonale, `aKontakty[i]` zwraca nam `i`-ty element tablicy. U nas jest on strukturą, zatem dostanie się do jej konkretnego pola wymaga też użycia operatora wyłuskania. Czynimy to i uzyskujemy ostatecznie oczekiwany rezultat, który porównujemy z poszukiwanym nickiem.
W ten sposób przeglądamy naszą tablicę aż do momentu, gdy faktycznie znajdziemy poszukiwany kontakt. Wtedy też kończymy funkcję i oddajemy indeks znalezionego elementu jako jej wynik. W przypadku niepowodzenia zwracamy natomiast `-1`, która to liczba nie może być indeksem tablicy w C++.
Cała operacja wyszukiwania nie należy więc do szczególnie skomplikowanych :)

Odrobina formalizmu - nie zaszkodzi!

Przyszedł właśnie czas na uporządkowanie i usystematyzowanie posiadanych informacji o strukturach. Największym zainteresowaniem obdarzymy przeto reguły składniowe języka, towarzyszące ich wykorzystaniu.
Mimo tak groźnego wstępu nie opuszczaj niniejszego paragrafu, bo taka absencja z pewnością nie wyjdzie ci na dobre :)

Typ strukturalny definiujemy, używając słowa kluczowego `struct` (ang. *structure* - struktura). Składnia takiej definicji wygląda następująco:

```
struct nazwa_typu
{
      typ_pola_1 nazwa_pola_1;
      typ_pola_2 nazwa_pola_2;
      typ_pola_3 nazwa_pola_3;
      ...
      typ_pola_n nazwa_pola_n;
};
```

Kolejne wiersze wewnątrz niej łudząco przypominają deklaracje zmiennych i tak też można je traktować. Pola struktury są przecież zawartymi w niej „podzmiennymi”. Całość tej listy pól ujmujemy oczywiście w stosowne do C++ nawiasy klamrowe.

Pamiętajmy, aby za końcowym nawiasem koniecznie umieścić **średnik**. Pomimo zbliżonego wyglądu definicja typu strukturalnego nie jest przecież funkcją i dlatego nie można zapominać o tym dodatkowym znaku.

Przykład wykorzystania struktury

To prawda, że używanie struktur dotyczy najczęściej dość złożonych zbiorów danych. Tym bardziej wydawałoby się, iż trudno o jakiś nietrywialny przykład zastosowania tegoż mechanizmu językowego w prostym programie.
Jest to jednak tylko część prawdy. Struktury występują bowiem bardzo często zarówno w standardowej bibliotece C++, jak i w innych, często używanych kodach - Windows API

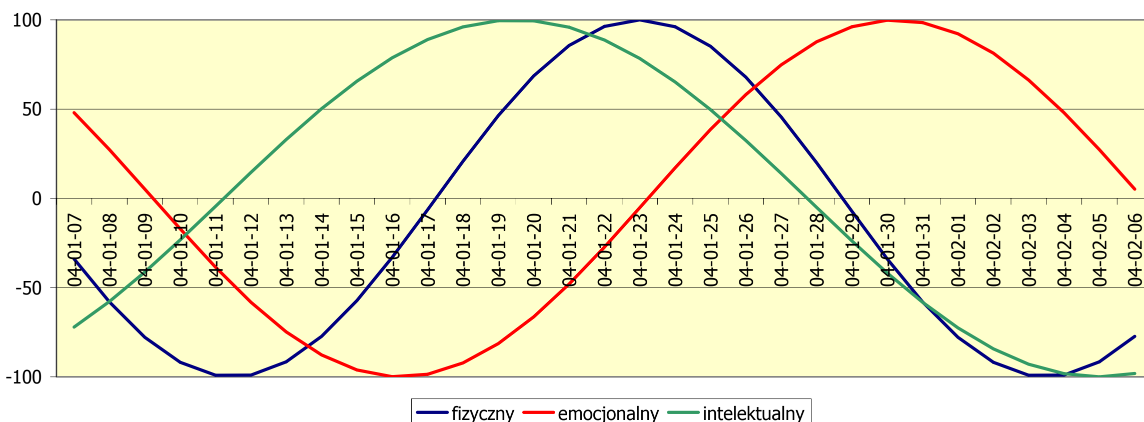
czy DirectX. Służą one nierzadko jako sposób na przekazywanie do i z funkcji dużej ilości wymaganych informacji. Zamiast kilkunastu parametrów lepiej przecież użyć jednego, kompleksowego, którym znacznie wygodniej jest operować.

My posłużymy się takim właśnie typem strukturalnym oraz kilkoma funkcjami pomocniczymi, aby zrealizować naszą prostą aplikację. Wszystkie te potrzebne elementy znajdziemy w pliku nagłówkowym c*time*, gdzie umieszczona jest także definicja typu `tm`:

```
struct tm
{
    int tm_sec;       // sekundy
    int tm_min;       // minuty
    int tm_hour;      // godziny
    int tm_mday;      // dzień miesiąca
    int tm_mon;       // miesiąc (0..11)
    int tm_year;      // rok (od 1900)
    int tm_wday;      // dzień tygodnia (0..6, gdzie 0 == niedziela)
    int tm_yday;      // dzień roku (0..365, gdzie 0 == 1 stycznia)
    int tm_isdst;     // czy jest aktywny czas letni?
};
```

Patrząc na nazwy jego pól oraz komentarze do nich, nietrudno uznać, iż typ ten ma za zadanie przechowywać datę i czas w formacie przyjaznym dla człowieka. To zaś prowadzi do wniosku, iż nasz program będzie wykonywał czynność związaną w jakiś sposób z upływem czasu. Istotnie tak jest, gdyż jego przeznaczeniem stanie się obliczanie biorytmu.

Biorytm to modny ostatnio zestaw parametrów, które określają aktualne możliwości psychofizyczne każdego człowieka. Według jego zwolenników, nasz potencjał fizyczny, emocjonalny i intelektualny waha się okresowo w cyklach o stałej długości, rozpoczynających się w chwili narodzin.



Wykres 1. Przykładowy biorytm autora tego tekstu :-)

Możliwe jest przy tym określenie liczbowej wartości każdego z trzech rodzajów biorytmu w danym dniu. Najczęściej przyjmuje się w tym celu przedział „procentowy", obejmujący liczby od -100 do +100.
Same obliczenia nie są szczególnie skomplikowane. Patrząc na wykres biorytmu, widzimy bowiem wyraźnie, iż ma on kształt trzech sinusoid, różniących się jedynie okresami. Wynoszą one tyle, ile długości trwania poszczególnych cykli biorytmu, a przedstawia je poniższa tabelka:

cykl	***długość***
fizyczny	23 dni

cykl	długość
emocjonalny	28 dni
intelektualny	33 dni

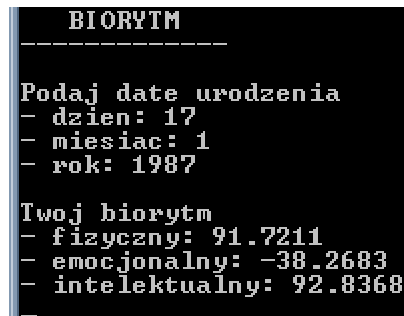
Tabela 10. Długości cykli biorytmu

Uzbrojeni w te informacje możemy już napisać program, który zajmie się liczeniem biorytmu. Oczywiście nie przedstawi on wyników w postaci wykresu (w końcu mamy do dyspozycji jedynie konsolę), ale pozwoli zapoznać się z nimi w postaci liczbowej, która także nas zadowala :)
Spójrzmy zatem na ten spory kawałek kodu:

```
// Biorhytm - pobieranie aktualnego czasu w postaci struktury
// i użycie go do obliczania biorytmu

// typ wyliczeniowy, określający rodzaj biorytmu
enum BIORHYTM {  BIO_PHYSICAL = 23,
                 BIO_EMOTIONAL = 28,
                 BIO_INTELECTUAL = 33 };

// pi :)
const double PI = 3.1415926538;

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

// funkcja wyliczająca dany rodzaj biorytmu
double Biorytm(double fDni, BIORHYTM Cykl)
{
   return 100 * sin((2 * PI / Cykl) * fDni);
}

// funkcja main()
void main()
{
   /* trzy struktury, przechowujące datę urodzenia delikwenta,
      aktualny czas oraz różnicę pomiędzy nimi                  */
   tm DataUrodzenia = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   tm AktualnyCzas = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   tm RoznicaCzasu = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };


   /* pytamy użytkownika o datę urodzenia */

   std::cout << "Podaj date urodzenia" << std::endl;

   // dzień
   std::cout << "- dzien: ";
   std::cin >> DataUrodzenia.tm_mday;

   // miesiąc - musimy odjąć 1, bo użytkownik poda go w systemie 1..12
   std::cout << "- miesiac: ";
   std::cin >> DataUrodzenia.tm_mon;
   DataUrodzenia.tm_mon--;

   // rok - tutaj natomiast musimy odjąć 1900
   std::cout << "- rok: ";
   std::cin >> DataUrodzenia.tm_year;
   DataUrodzenia.tm_year -= 1900;


   /* obliczamy liczbę przeżytych dni */
```

```
    // pobieramy aktualny czas w postaci struktury
    time_t Czas = time(NULL);
    AktualnyCzas = *localtime(&Czas);

    // obliczamy różnicę między nim a datą urodzenia
    RoznicaCzasu.tm_mday = AktualnyCzas.tm_mday - DataUrodzenia.tm_mday;
    RoznicaCzasu.tm_mon = AktualnyCzas.tm_mon - DataUrodzenia.tm_mon;
    RoznicaCzasu.tm_year = AktualnyCzas.tm_year - DataUrodzenia.tm_year;

    // przeliczamy to na dni
    double fPrzezyteDni = RoznicaCzasu.tm_year * 365.25
                         + RoznicaCzasu.tm_mon * 30.4375
                         + RoznicaCzasu.tm_mday;


     /* obliczamy biorytm i wyświelamy go */

    // otóż i on
    std::cout << std::endl;
    std::cout << "Twoj biorytm" << std::endl;
    std::cout << "- fizyczny: " << Biorytm(fPrzezyteDni, BIO_PHYSICAL)
              << std::endl;
    std::cout << "- emocjonalny: " << Biorytm(fPrzezyteDni,
                                              BIO_EMOTIONAL) << std::endl;
    std::cout << "- intelektualny: " << Biorytm(fPrzezyteDni,
                                            BIO_INTELECTUAL) << std::endl;

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

Jaki jest efekt tego, pokaźnych rozmiarów, listingu? Są nim trzy wartości określające dzisiejszy biorytm osoby o podanej dacie urodzenia:



Screen 31. Efekt działania aplikacji obliczającej biorytm

Za jego wyznaczenie odpowiada prosta funkcja `Biorytm()` wraz towarzyszącym jej typem wyliczeniowym, określającym rodzaj biorytmu:

```
enum BIORHYTM {  BIO_PHYSICAL = 23,
                 BIO_EMOTIONAL = 28,
                 BIO_INTELECTUAL = 33 };

double Biorytm(double fDni, BIORHYTM Cykl)
{
    return 100 * sin((2 * PI / Cykl) * fDni);
}
```

Godną uwagi sztuczką, jaką tu zastosowano, jest nadanie stałym typu `BIORHYTM` wartości, będących jednocześnie długościami odpowiednich cykli biorytmu. Dzięki temu funkcja zachowuje przyjazną postać wywołania, na przykład `Biorytm(liczba_dni, BIO_PHYSICAL)`, a jednocześnie unikamy instrukcji `switch` wewnątrz niej.

Sama formułka licząca opiera się na ogólnym wzorze sinusoidy, tj.:

$$f(x) = A\sin\left(\frac{2\pi}{T} \cdot x\right)$$

w którym *A* jest jej amplitudą, zaś *T* - okresem.
U nas okresem jest długość trwania poszczególnych cykli biorytmu, zaś amplituda 100 powoduje „rozciągnięcie” przedziału wartości do zwyczajowego <-100; +100>.

Stanowiąca większość kodu długa funkcja `main()` dzieli się na trzy części.

W pierwszej z nich pobieramy od użytkownika jego datę urodzenia i zapisujemy ją w strukturze o nazwie… `DataUrodzenia` :) Zauważmy, że używamy tutaj jej pól jako miejsca docelowego dla strumienia wejścia w identyczny sposób, jak to czyniliśmy dla pojedynczych zmiennych.
Po pobraniu musimy jeszcze odpowiednio zmodyfikować dane - tak, żeby spełniały wymagania podane w komentarzach przy definicji typu `tm` (chodzi tu o numerowanie miesięcy od zera oraz liczenie lat począwszy od roku 1900).

Kolejnym zadaniem jest obliczenie ilości dni, jaką dany osobnik przeżył już na tym świecie. W tym celu musimy najpierw pobrać aktualny czas, co też czynią dwie poniższe linijki:

```
time_t Czas = time(NULL);
AktualnyCzas = *localtime(&Czas);
```

W pierwszej z nich znana nam już funkcja `time()` uzyskuje czas w wewnętrznym formacie C++[57]. Dopiero zawarta w drugim wierszu funkcja `localtime()` konwertuje go na zdatną do wykorzystania strukturę, którą przypisujemy do zmiennej `AktualnyCzas`.

Troszkę udziwnioną postać tej funkcji musisz na razie niestety zignorować :)

Dalej obliczamy różnicę między oboma czasami (zapisanymi w `DataUrodzenia` i `AktualnyCzas`), odejmując od siebie liczby dni, miesięcy i lat. Otrzymany tą drogą wiek użytkownika musimy na koniec przeliczyć na pojedyncze dni, za co odpowiada wyrażenie:

```
double fPrzezyteDni =  RoznicaCzasu.tm_year * 365.25
                     + RoznicaCzasu.tm_mon * 30.4375
                     + RoznicaCzasu.tm_mday;
```

Zastosowane tu liczby `365.25` i `30.4375` są średnimi ilościami dni w roku oraz w miesiącu. Uwalniają nas one od konieczności osobnego uwzględniania lat przestępnych w przeprowadzanych obliczeniach.

Wreszcie, ostatnie wiersze kodu obliczają biorytm, wywołując trzykrotnie funkcję o tej nazwie, i prezentują wyniki w klarownej postaci w oknie konsoli.

[57] Jest to liczba sekund, które upłynęły od północy 1 stycznia 1970 roku.

Działanie programu kończy się zaś na tradycyjnym `getch()`, które oczekuje na przyciśnięcie dowolnego klawisza. Po tym fakcie następuje już definitywny i nieodwołalny koniec :D

Tak oto przekonaliśmy się, że struktury warto znać nawet wtedy, gdy nie planujemy tworzenia aplikacji manewrujących skomplikowanymi danymi. Nie zdziw się zatem, że w dalszym ciągu tego kursu będziesz je całkiem często spotykał.

Unie

Drugim, znacznie rzadziej spotykanym rodzajem złożonych typów są **unie**.

Są one w pewnym sensie podobne do struktur, gdyż ich definicje stanowią także listy poszczególnych pól:

```
union nazwa_typu
{
    typ_pola_1 nazwa_pola_1;
    typ_pola_2 nazwa_pola_2;
    typ_pola_3 nazwa_pola_3;
    ...
    typ_pola_n nazwa_pola_n;
};
```

Identycznie wyglądają również deklaracje zmiennych, należących do owych typów „unijnych", oraz odwołania do ich pól. Na czym więc polegają różnice?…

Przypomnijmy sobie, że struktura jest zestawem kilku odrębnych zmiennych, połączonych w jeden kompleks. Każde jego pole zachowuje się dokładnie tak, jakby było samodzielną zmienną, i posłusznie przechowuje przypisane mu wartości. Rozmiar struktury jest zaś co najmniej sumą rozmiarów wszystkich jej pól.
Unia opiera się na nieco innych zasadach. Zajmuje bowiem w pamięci jedynie tyle miejsca, żeby móc pomieścić swój **największy element**. Nie znaczy to wszak, iż w jakiś nadprzyrodzony sposób potrafi ona zmieścić w takim okrojonym obszarze wartości wszystkich pól. Przeciwnie, nawet nie próbuje tego robić. Zamiast tego obszary pamięci przeznaczone na wartości pól unii zwyczajnie **nakładają się** na siebie. Powoduje to, że:

W danej chwili tylko **jedno pole** unii zawiera poprawną wartość.

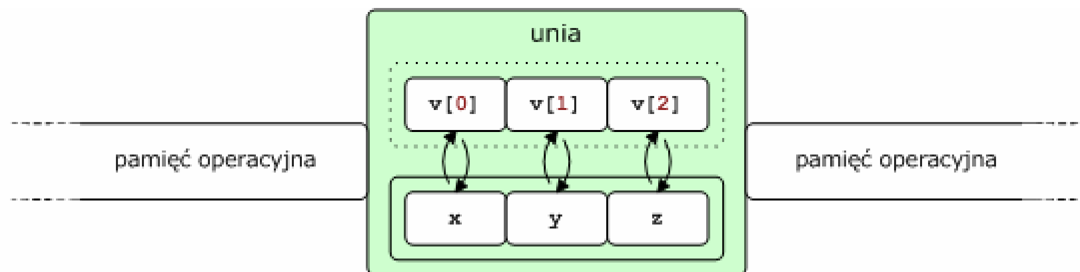
Do czego mogą się przydać takie dziwaczne twory? Cóż, ich zastosowania są dość swoiste, więc nieczęsto będziesz zmuszony do skorzystania z nich.
Jednym z przykładów może być jednak chęć zapewnienia kilku dróg dostępu do tych samych danych:

```
union VECTOR3
{
    // w postaci trójelementowej tablicy
    float v[3];

    // lub poprzez odpowiednie zmienne x, y, z
    struct
    {
            float x, y, z;
    };
};
```

W powyższej unii, która ma przechowywać trójwymiarowy wektor, możliwe są dwa sposoby na odwołanie się do jego współrzędnych: poprzez pola `x`, `y` oraz `z` lub indeksy odpowiedniej tablicy `v`. Oba są równoważne:

```
VECTOR3 vWektor;

// poniższe dwie linijki robią to samo
vWektor.x      = 1.0; vWektor.y    = 5.0; vWektor.z    = 0.0;
vWektor.v[0]   = 1.0; vWektor.v[1] = 5.0; vWektor.v[2] = 0.0;
```

Taka unię możemy więc sobie obrazowo przedstawić chociażby poprzez niniejszy rysunek:



Schemat 12. Model przechowywania unii w pamięci operacyjnej

Elementy tablicy `v` oraz pola `x`, `y`, `z` niejako „wymieniają” między sobą wartości. Oczywiście jest to tylko pozorna wymiana, gdyż tak naprawdę chodzi po prostu o odwoływanie się do **tego samego adresu w pamięci**, jednak **różnymi drogami**.

Wewnątrz naszej unii umieściliśmy tzw. anonimową strukturę (nieopatrzoną żadną nazwą). Musieliśmy to zrobić, bo jeżeli wpisalibyśmy `float x, y, z;` bezpośrednio do definicji unii, każde z tych pól byłoby zależne od pozostałych i tylko jedno z nich miałoby poprawną wartość. Struktura natomiast łączy je w integralną całość.

Można zauważyć, że struktury i unie są jakby odpowiednikiem operacji logicznych - koniunkcji i alternatywy - w odniesieniu do budowania złożonych typów danych. Struktura pełni jak gdyby funkcję operatora `&&` (pozwalając na niezależne istnienie wszystkim obejmowanym sobą zmiennym), zaś unia - operatora `||` (dopuszczając wyłącznie jedną daną). Zagnieżdżając frazy `struct` i `union` wewnątrz definicji kompleksowych typów możemy natomiast uzyskać bardziej skomplikowane kombinacje. Naturalnie, rodzi się pytanie „Po co?”, ale to już zupełnie inna kwestia ;)

Więcej informacji o uniach zainteresowani znajdą w MSDN.

***

Lektura kończącego się właśnie podrozdziału dała ci możliwość rozszerzania wachlarza standardowych typów C++ o takie, które mogą ci ułatwić tworzenie przyszłych aplikacji. Poznałeś więc typy wyliczeniowe, struktury oraz unie, uwalniając całkiem nowe możliwości programistyczne. Na pewno niejednokrotnie będziesz z nich korzystał.

Większy projekt

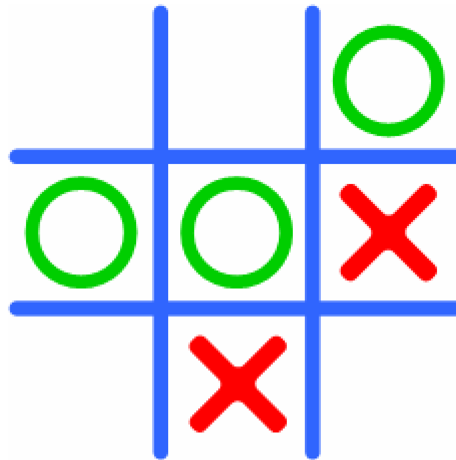
Doszedłszy do tego miejsca w lekturze niniejszego kursu posiadłeś już dosyć dużą wiedzę programistyczną. Pora zatem na wykorzystanie jej w praktyce: czas stworzyć jakąś

większą aplikację, a ponieważ docelowo mamy zajmować się programowaniem gier, więc będzie nią właśnie gra.

Nie możesz wprawdzie liczyć na oszałamiające efekty graficzne czy dźwiękowe, gdyż chwilowo potrafimy operować jedynie konsolą, lecz nie powinno cię to mimo wszystko zniechęcać. Środki tekstowe okażą się bowiem całkowicie wystarczające dla naszego skromnego projektu.

Projektowanie

Cóż więc chcemy napisać? Otóż będzie to produkcja oparta na wielce popularnej i lubianej grze w kółko i krzyżyk :) Zainteresujemy się jej najprostszym wariantem, w którym dwoje graczy stawia naprzemian kółka i krzyżyki na planszy o wymiarach 3×3. Celem każdego z nich jest utworzenie linii z trzech własnych symboli - poziomej, pionowej lub ukośnej.



Rysunek 2. Rozgrywka w kółko i krzyżyk

Nasza gra powinna pokazywać rzeczoną planszę w czasie rozgrywki, umożliwiać wykonywanie graczom kolejnych ruchów oraz sprawdzać, czy któryś z nich przypadkiem nie wygrał :)
I taki właśnie efekt będziemy chcieli osiągnąć, tworząc ten program w C++. Najpierw jednak, skoro już wiemy, **co** będziemy pisać, zastanówmy się, **jak** to napiszemy.

Struktury danych w aplikacji

Pierwszym zadaniem jest określenie **struktur danych**, wykorzystywanych przez program. Oznacza to ustalenie zmiennych, które przewidujemy w naszej aplikacji oraz danych, jakie mają one przechowywać. Ponieważ wiemy już niemal wszystko na temat sposobów organizowania informacji w C++, nasze instrumentarium w tym zakresie będzie bardzo szerokie. Zatem do dzieła!

Chyba najbardziej oczywistą potrzebą jest konieczność stworzenia jakiejś programowej reprezentacji planszy, na której toczy się rozgrywka. Patrząc na nią, nietrudno jest znaleźć odpowiednią drogę do tego celu: wręcz idealna wydaje się bowiem **tablica** dwuwymiarowa o rozmiarze 3×3.
Sama wielkość to jednak nie wszystko - należy także określić, **jakiego typu elementy** ma zawierać ta tablica. Aby to uczynić, pomyślmy, co się dzieje z planszą podczas rozgrywki. Na początku zawiera ona wyłącznie puste pola; potem kolejno pojawiają się w nich kółka lub krzyżyki… Czy już wiesz, jaki typ będzie właściwy?… Naturalnie, chodzi tu o odpowiedni **typ wyliczeniowy**, dopuszczający jedynie trzy możliwe wartości: pole puste, kółko lub krzyżyk. To było od początku oczywiste, prawda? :)

Ostatecznie plansza będzie wyglądać w ten sposób:

```
enum FIELD { FLD_EMPTY, FLD_CIRCLE, FLD_CROSS };
FIELD g_aPlansza[3][3] = { { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY } };
```

Inicjalizacja jest tu odzwierciedleniem faktu, iż na początku wszystkie jej pola są puste.

Plansza to jednakowoż nie wszystko. W naszej grze będzie się przecież coś dziać: gracze dokonywać będą swych kolejnych posunięć. Potrzebujemy więc zmiennych opisujących **przebieg** rozgrywki.
Ich wyodrębnienie nie jest już takie łatwe, aczkolwiek nie powinniśmy mieć z tym wielkich kłopotów. Musimy mianowicie pomyśleć o grze w kółko i krzyżyk jako o procesie przebiegającym **etapami**, według określonego schematu. To nas doprowadzi do pierwszej zmiennej, określającej aktualny **stan gry**:

```
enum GAMESTATE { GS_NOTSTARTED, // gra nie została rozpoczęta
                 GS_MOVE,       // gra rozpoczęta, gracze wykonują ruchy
                 GS_WON,        // gra skończona, wygrana któregoś gracza
                 GS_DRAW };     // gra skończona, remis
GAMESTATE g_StanGry = GS_NOTSTARTED;
```

Wyróżniliśmy tutaj cztery fazy:
- **początkowa** - właściwa gra jeszcze się nie rozpoczęła, czynione są pewne przygotowania (o których wspomnimy nieco dalej)
- **rozgrywka** - uczestniczący w niej gracze naprzemiennie wykonują ruchy. Jest to zasadnicza część całej gry i trwa najdłużej.
- **wygrana** - jeden z graczy zdołał ułożyć linię ze swoich symboli i wygrał partię
- **remis** - plansza została szczelnie zapełniona znakami zanim którykolwiek z graczy zdołał zwyciężyć

Czy to wystarczy? Nietrudno się domyśleć, że nie. Nie przewidzieliśmy bowiem żadnego sposobu na przechowywanie informacji o tym, **który z graczy** ma w danej chwili wykonać swój ruch. Czym prędzej zatem naprawimy swój błąd:

```
enum SIGN { SGN_CIRCLE, SGN_CROSS };
SIGN g_AktualnyGracz;
```

Zauważmy, iż nie posiadamy o graczach żadnych dodatkowych wiadomości ponad fakt, jakie znaki (kółko czy krzyżyk) stawiają oni na planszy. Informacja ta jest zatem jedynym kryterium, pozwalającym na ich odróżnienie - toteż skrzętnie z niej korzystamy, deklarując zmienną odpowiedniego typu wyliczeniowego.

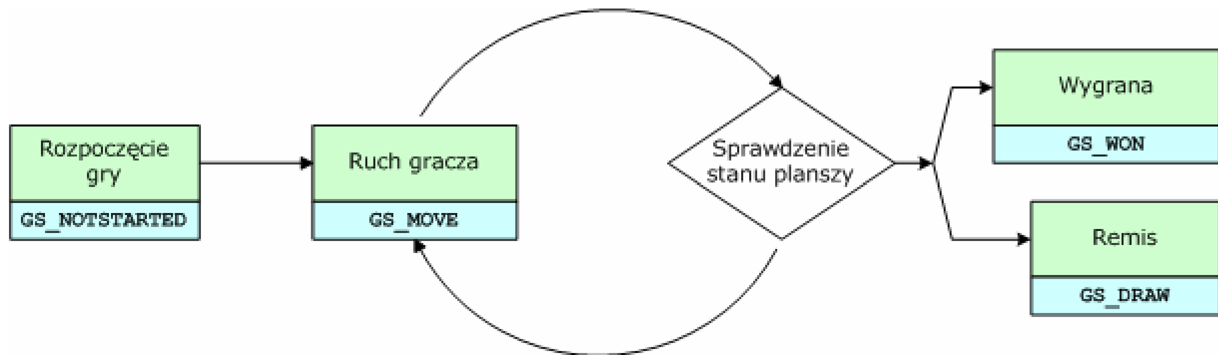
Zamodelowanie właściwych struktur danych kontrolujących przebieg gry to jedna z ważniejszych czynności przy jej projektowaniu. W naszym przypadku są one bardzo proste (jedynie dwie zmienne), jednak zazwyczaj przyjmują znacznie bardziej skomplikowaną formę. W swoim czasie zajmiemy się dokładniej tym zagadnieniem.

Zdaje się, że to już wszystkie zmienne, jakich będziemy potrzebować w naszym programie. Czas zatem zająć się jego drugą, równie ważną częścią, czyli kodem odpowiedzialnym za właściwe funkcjonowanie.

Działanie programu

Przed chwilą wprowadziliśmy sobie dwie zmienne, które będą nam pomocne w zaprogramowaniu przebiegu naszej gry od początku aż do końca. Teraz właśnie

zajmiemy się tymże „szlakiem” programu, czyli sposobem, w jaki będzie on działał i prowadził rozgrywkę. Możemy go zilustrować na diagramie podobnym do poniższego:

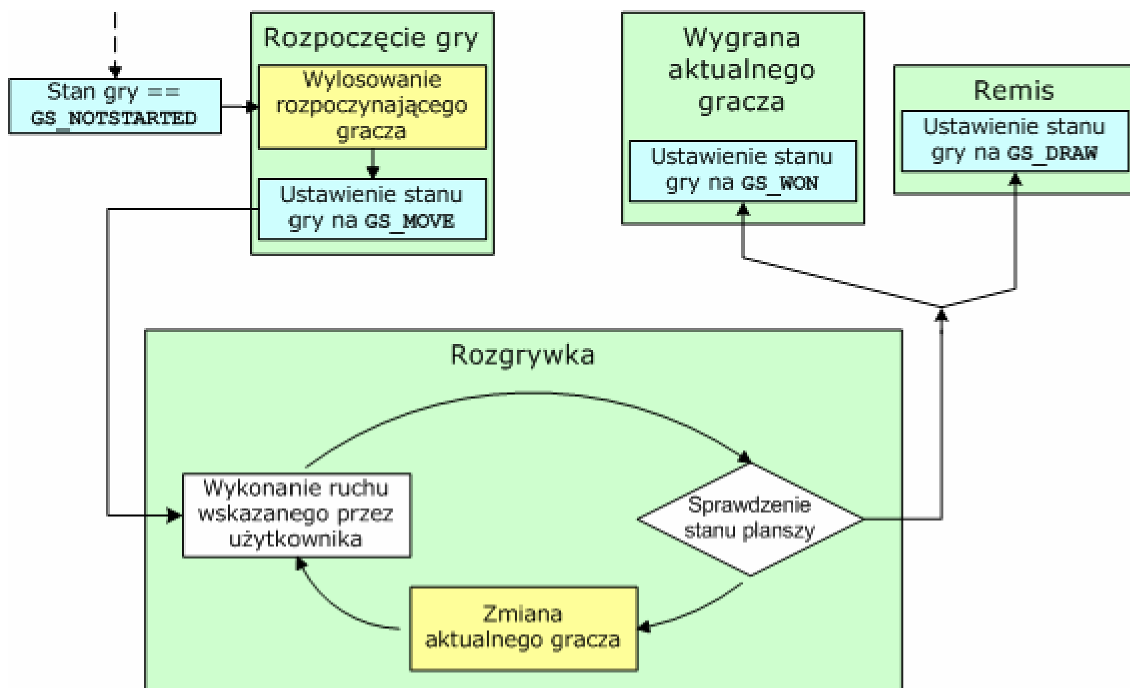


Schemat 13. Przebieg gry w kółko i krzyżyk

Widzimy na nim, w jaki sposób następuje **przejście** pomiędzy poszczególnymi **stanami** gry, a więc kiedy i jak ma się zmieniać wartość zmiennej `g_StanGry`. Na tej podstawie moglibyśmy też określić funkcje, które są konieczne do napisania oraz ogólne czynności, jakie powinny one wykonywać.

Powyższy rysunek jest uproszczonym diagramem przejść stanów. To jeden z wielu rodzajów schematów, jakie można wykonać podczas projektowania programu.

Potrzebujemy jednak bardziej szczegółowego opisu. Lepiej jest też wykonać go teraz, podczas projektowania aplikacji, niż przekładać do czasu faktycznego programowania. Przy okazji uściślania przebiegu programu postaramy się uwzględnić w nim także pominięte wcześniej, „drobne” szczegóły - jak choćby określenie aktualnego gracza i jego zmiana po każdym wykonanym ruchu.
Nasz nowy szkic może zatem wyglądać tak:



Schemat 14. Działanie programu do gry w kółko i krzyżyk

Można tutaj zauważyć czwórkę potencjalnych kandydatów na funkcje - są to sekwencje działań zawarte w zielonych polach. Faktycznie jednak dla dwóch ostatnich (wygranej oraz remisu) byłoby to pewnym nadużyciem, gdyż zawarte w nich operacje można z powodzeniem dołączyć do funkcji obsługującej rozgrywkę. Są to bowiem jedynie przypisania do zmiennej.

Ostatecznie mamy przewidziane dwie zasadnicze funkcje programu:

- rozpoczęcie gry, realizowane na początku. Jej zadaniem jest przygotowanie rozgrywki, czyli przede wszystkim wylosowanie gracza zaczynającego
- rozgrywka, a więc wykonywanie kolejnych ruchów przez graczy

Skoro wiemy już, jak nasza gra ma działać „od środka”, nie od rzeczy będzie zajęcie się metodą jej komunikacji z żywymi użytkownikami-graczami.

Interfejs użytkownika

Hmm, jaki interfejs?…

Zazwyczaj pojęcie to utożsamiamy z okienkami, przyciskami, pola tekstowymi, paskami przewijania i innymi zdobyczami graficznych systemów operacyjnych. Tymczasem termin ten ma bardziej szersze znaczenie:

Interfejs użytkownika to sposób, w jaki aplikacja prowadzi dialog z obsługującymi ją osobami. Obejmuje to zarówno pobieranie od nich danych wejściowych, jak i prezentację wyników pracy.

Niewątpliwie więc możemy czuć się uprawnieni, aby nazwać naszą skromną konsolę pełnowartościowym środkiem do realizacji interfejsu użytkownika! Pozwala ona przecież zarówno na uzyskiwanie informacji od osoby siedzącej za klawiaturą, jak i na wypisywanie przeznaczonych dla niej komunikatów programu.

Jak zatem mógłby wyglądać interfejs naszego programu?… Twoje dotychczasowe, bogate doświadczenie z aplikacjami konsolowymi powinny ułatwić ci odpowiedź na to pytanie. Informacja, którą prezentujemy użytkownikowi, to oczywiście aktualny stan planszy. Nie będzie ona wprawdzie miała postaci rysunkowej, jednakże zwykły tekst całkiem dobrze sprawdzi się w roli „grafiki”.
Po wyświetleniu bieżącego stanu rozgrywki można poprosić o gracza o wykonanie swojego ruchu. Gdybyśmy mogli obsłużyć myszkę, wtedy posunięcie byłoby po prostu kliknięciem, ale w tym wypadku musimy zadowolić się poleceniem wpisanym z klawiatury.

Ostatecznie wygląd naszego programu może być podobny do poniższego:

```
 -----
 |1XX|
 |406|
 |709|
 -----

Podaj numer pola, w ktorym
chcesz postawic kolko: _
```

Screen 32. Interfejs użytkownika naszej gry

Przy okazji zauważyć można jedno z rozwiązań problemu pt. „Jak umożliwić wykonywanie ruchów, posługując się jedynie klawiaturą?” Jest nim tutaj ponumerowanie kolejnych elementów tablicy-planszy liczbami od 1 do 9, a następnie prośba do gracza o podanie jednej z nich. To chyba najwygodniejsza forma gry, jaką potrafimy osiągnąć w tych niesprzyjających, tekstowych warunkach…

Metodami na przeliczanie pomiędzy zwyczajnymi, dwoma współrzędnymi tablicy oraz tą jedną „nibywspółrzędną” zajmiemy się podczas właściwego programowania.

***

Na tym możemy już zakończyć wstępne projektowanie naszego projektu :) Ustaliliśmy sposób jego działania, używane przezeń struktury danych, a nawet interfejs użytkownika. Wszystko to ułatwi nam pisanie kodu całej aplikacji, które to rozpoczniemy już za chwilę.

To był tylko skromny i bardzo nieformalny wstęp do dziedziny informatyki zwanej inżynierią oprogramowania. Zajmuje się ona projektowaniem wszelkiego rodzaju programów, poczynając każdy od pomysłu i prowadząc poprzez model, kod, testowanie i wreszcie użytkowanie. Jeżeli chciałbyś się dowiedzieć więcej na ten interesujący i przydatny temat, zapraszam do Materiału Pomocniczego C, *Podstawy inżynierii oprogramowania* (aczkolwiek zalecam najpierw skończenie tej części kursu).

Kodowanie

Nareszcie możemy uruchomić swoje ulubione środowisko programistyczne, wspierające ulubiony język programowania C++ i zacząć właściwe programowanie zaprojektowanej już gry. Uczyń to więc, stwórz w nim nowy projekt, nazywając go dowolnie[58], i czekaj na dalsze rozkazy ;D

Kilka modułów i własne nagłówki

Na początek utworzymy i dodamy do projektu wszystkie pliki, z jakich docelowo ma się składać. Zgadza się - **pliki**. Pisany przez nas program może okazać się całkiem duży, dlatego rozsądnie będzie podzielić jego kod pomiędzy kilka odrębnych modułów. Utrzymamy wtedy jego względny porządek oraz skrócimy czas kolejnych kompilacji.

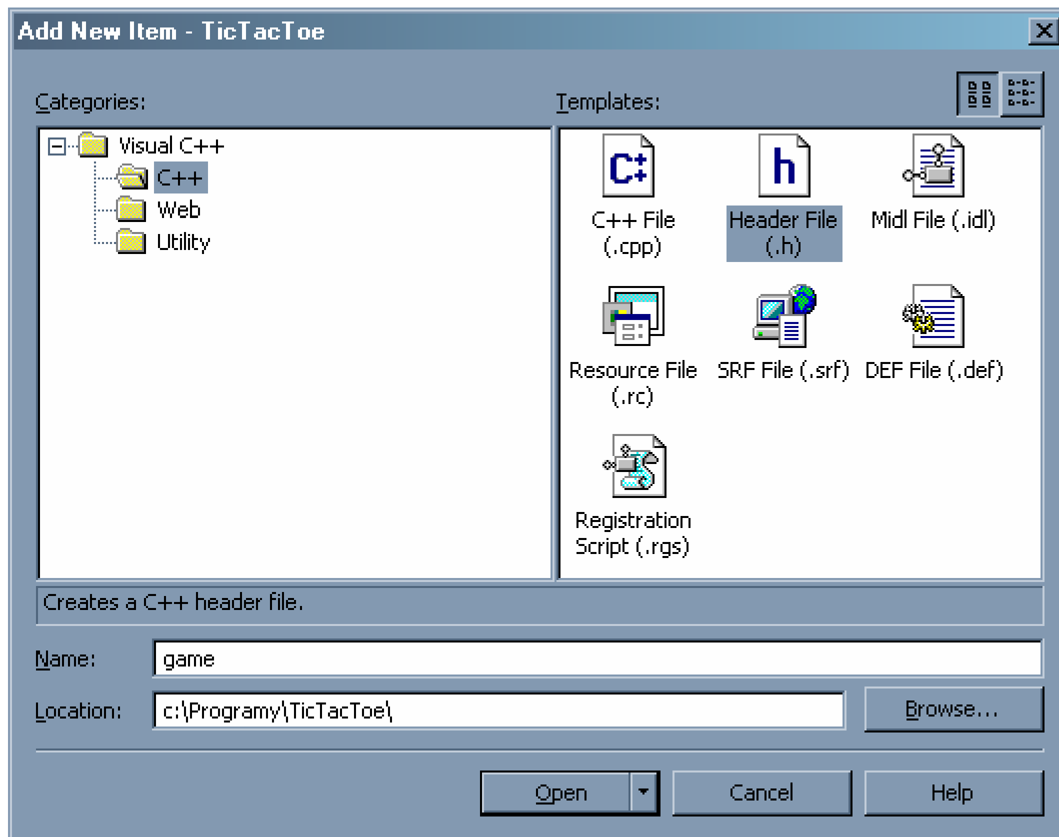
Zwyczajowo zaczniemy od pliku *main.cpp*, w którym umieścimy główną funkcję programu, `main()`. Chwilowo jednak nie wypełnimy ją żadną treścią:

```
void main()
{

}
```

Zamiast tego wprowadzimy do projektu jeszcze jeden moduł, w którym wpiszemy właściwy kod naszej gry. Przy pomocy opcji menu *Project|Add New Item* dodaj więc do aplikacji drugi już plik typu *C++ File (.cpp)* i nazwij go *game.cpp*. W tym module znajdą się wszystkie zasadnicze funkcje programu.

To jednak nie wszystko! Na deser zostawiłem bowiem pewną nowość, z którą nie mieliśmy okazji się do tej pory zetknąć. Stworzymy mianowicie swój **własny plik nagłówkowy**, idący w parze ze świeżo dodanym modułem *game.cpp*. Uczynimy to podobny sposób, co dotychczas - z tą różnicą, iż tym razem zmienimy typ dodawanego pliku na *Header File (.h)*.

[58] Kompletny kod całej aplikacji jest zawarty w przykładach do tego rozdziału i opatrzony nazwą `TicTacToe`.



Screen 33. Dodawanie pliku nagłówkowego do projektu

Po co nam taki własny nagłówek? W jakim celu w ogóle tworzyć nagłówki we własnych projektach?...
Na powyższe pytania istnieje dosyć prosta odpowiedź. Aby ją poznać przypomnijmy sobie, dlaczego dołączamy do naszych programów nagłówki w rodzaju *iostream* czy *conio.h*. Hmm?...
Tak jest - dzięki nim jesteśmy w stanie korzystać z takich dobrodziejstw języka C++ jak strumienie wejścia i wyjścia czy łańcuchy znaków. Generalizując, można powiedzieć, że nagłówki **udostępniają** pewien **kod** wszystkim modułom, które dołączą je przy pomocy dyrektywy `#include`.

Dotychczas nie zastanawialiśmy się zbytnio nad miejscem, w którym egzystuje kod wykorzystywany przez nas za pośrednictwem nagłówków. Faktycznie może on znajdować się „tuż obok" - w innym module tego samego projektu (i tak będzie u nas), lecz równie dobrze istnieć jedynie w skompilowanej postaci, na przykład biblioteki DLL.
W przypadku dodanego właśnie nagłówka *game.h* mamy jednak niczym nieskrępowany dostęp do odpowiadającego mu modułu *game.cpp*. Zdawałoby się zatem, że plik nagłówkowy jest tu całkowicie zbędny, a z kodu zawartego we wspomnianym module moglibyśmy z powodzeniem korzystać bezpośrednio.

Nic bardziej błędnego! Za użyciem pliku nagłówkowego przemawia wiele argumentów, a jednym z najważniejszych jest **zasada ograniczonego zaufania**. Według niej każda cząstka programu powinna posiadać dostęp jedynie do tych jego fragmentów, które są **niezbędne** do jej prawidłowego funkcjonowania.
U nas tą cząstką będzie funkcja `main()`, zawarta w module *main.cpp*. Nie napisaliśmy jej jeszcze, ale potrafimy już określić, czego będzie potrzebowała do swego poprawnego działania. Bez wątpienia będą dlań konieczne funkcje odpowiedzialne za wykonywanie posunięć wskazanych przez graczy czy też procedury wyświetlające aktualny stan rozgrywki. Sposób, w jaki te zadania są realizowane, nie ma jednak żadnego znaczenia!

Podobnie przecież nie jesteśmy zobligowani do wiedzy o szczegółach funkcjonowania strumieni konsoli, a mimo to stale z nich korzystamy.

Plik nagłówkowy pełni więc rolę swoistej zasłony, przykrywającej nieistotne detale implementacyjne, oraz klucza do tych zasobów programistycznych (typów, funkcji, zmiennych, itd.), którymi rzeczywiście chcemy się dzielić.

Dlaczego w zasadzie mamy się z podobną nieufnością odnosić do, bądź co bądź, samego siebie? Czy rzeczywiście w tym przypadku lepiej wiedzieć mniej niż więcej?…
Główną przyczyną, dla której zasadę ograniczonego zaufania uznaje się za powszechnie słuszną, jest fakt, iż wprowadza ona sporo porządku do każdego kodu. Chroni też przed wieloma błędami spowodowanymi np. nadaniem jakiejś zmiennej wartości spoza dopuszczalnego zakresu czy też wywołania funkcji w złym kontekście lub z nieprawidłowymi parametrami.

Nagłówki są też pewnego rodzaju „spisem treści” kodu źródłowego modułu czy biblioteki. Zawierają najczęściej deklaracje wszystkich typów oraz funkcji, więc mogą niekiedy służyć za prowizoryczną dokumentację[59] danego fragmentu programu, szczególnie przydatną w jego dalszym tworzeniu.
Z tego też powodu pliki nagłówkowe są najczęściej pierwszymi składnikami aplikacji, na których programista koncentruje swoją uwagę. Później stanowią one również podstawę do pisania właściwego kodu algorytmów.

My także zaczniemy kodowanie naszego programu od pliku *game.h*; gotowy nagłówek będzie nam potem doskonałą pomocą naukową :)

Treść pliku nagłówkowego

W nagłówku *game.h* umieścimy przeróżne deklaracje większości tworów programistycznych, wchodzących w skład naszej aplikacji. Będą to chociażby zmienne oraz funkcje.

Rozpoczniemy jednak od wpisania doń definicji trzech typów wyliczeniowych, które ustaliliśmy podczas projektowania programu. Chodzi naturalnie o `SIGN`, `FIELD` i `GAMESTATE`:

```
enum SIGN { SGN_CIRCLE, SGN_CROSS };
enum FIELD { FLD_EMPTY, FLD_CIRCLE, FLD_CROSS };
enum GAMESTATE { GS_NOTSTARTED, GS_MOVE, GS_WON, GS_DRAW };
```

Jest to powszechny zwyczaj w C++. Powyższe linijki moglibyśmy wszakże z równym powodzeniem umieścić wewnątrz modułu *game.cpp*. Wyodrębnienie ich w pliku nagłówkowym ma jednak swoje uzasadnienie: własne typy zmiennych są bowiem takimi zasobami, z których najczęściej korzysta większa część danego programu. Jako kod **współdzielony** (ang. *shared*) są więc idealnym kandydatem do umieszczenia w odpowiednim nagłówku.

W dalszej części pomyślimy już o konkretnych funkcjach, którym powierzymy zadanie kierowania naszą grą. Pamiętamy z fazy projektowania, iż przewidzieliśmy przynajmniej dwie takie funkcje: odpowiedzialną za rozpoczęcie gry oraz za przebieg rozgrywki, czyli wykonywanie ruchów i sprawdzanie ich skutku. Możemy jeszcze dołożyć do nich algorytm „rysujący” (jeśli można tak powiedzieć w odniesieniu do konsoli) aktualny stan planszy.

[59] Nie chodzi tu o podręcznik użytkownika programu, ale raczej o jego dokumentację techniczną, czyli opis działania aplikacji od strony programisty.

Teraz sprecyzujemy nieco nasze pojęcie o tych funkcjach. Do pliku nagłówkowego wpiszemy bowiem ich **prototypy**:

```
// prototypy funkcji
//-------------------

// rozpoczęcie gry
bool StartGry();

// wykonanie ruchu
bool Ruch(unsigned);

// rysowanie planszy
bool RysujPlansze();
```

Cóż to takiego? Prototypy, zwane też deklaracjami funkcji, są jakby ich nagłówkami oddzielonymi od bloku zasadniczego kodu (ciała). Mając prototyp funkcji, posiadamy informacje o jej **nazwie**, **typach parametrów** oraz **typie zwracanej wartości**. Są one wystarczające do jej wywołania, aczkolwiek nic nie mówią o faktycznych czynnościach, jakie dana funkcja wykonuje.

Prototyp (**deklaracja**) **funkcji** to wstępne określenie jej nagłówka. Stanowi on informację dla kompilatora i programisty o sposobie, w jaki funkcja może być wywołana.

Z punktu widzenia kodera dołączającego pliki nagłówkowe prototyp jest furtką do skarbca, przez którą można przejść jedynie z zawiązanymi oczami. Niesie wiedzę o tym, **co** prototypowana funkcja robi, natomiast nie daje żadnych wskazówek o sposobie, w **jaki** to czyni. Niemniej jest on nieodzowny, aby rzeczoną funkcję móc wywołać.

Warto wiedzieć, że dotychczas znana nam forma funkcji jest zarówno jej prototypem (deklaracją), jak i definicją (implementacją). Prezentuje bowiem pełnię wiadomości potrzebnych do jej wywołania, a poza tym zawiera wykonywalny kod funkcji.

Dla nas, przyszłych autorów zadeklarowanych właśnie funkcji, prototyp jest kolejną okazją do zastanowienia się nad kodem poszczególnych procedur programu. Precyzując ich parametry i zwracane wartości, budujemy więc solidne fundamenty pod ich niedalekie zaprogramowanie.

Dla formalności zerknijmy jeszcze na składnię prototypu funkcji:

```
typ_zwracanej_wartości/void nazwa_funkcji([typ_parametru [nazwa], ...]);
```

Oprócz uderzającego podobieństwa do jej nagłówka rzuca się w oczy również fakt, iż na etapie deklaracji nie jest konieczne podawanie nazw ewentualnych parametrów funkcji. Dla kompilatora w zupełności bowiem wystarczają ich typy.

Już któryś raz z kolei uczulam na kończący instrukcję **średnik**. Bez niego kompilator będzie oczekiwał bloku kodu funkcji, a przecież istotą prototypu jest jego niepodawanie.

Właściwy kod gry

Zastanowienie może budzić powód, dla którego żadna z trzech powyższych funkcji nie została zadeklarowana jako `void`. Przecież zgodnie z tym, co ustaliliśmy podczas projektowania wszystkie mają przede wszystkim wykonywać jakieś działania, a nie obliczać wartości.
To rzeczywiście prawda. Rezultat zwracany przez te funkcje ma jednak inną rolę - będzie informował o powodzeniu lub niepowodzeniu danej operacji. Typ `bool` zapewnia tutaj

najprostszą możliwą **obsługę ewentualnych błędów**. Warto o niej pomyśleć nawet wtedy, gdy pozornie nic złego nie może się zdarzyć. Wyrabiamy sobie w ten sposób dobre nawyki programistyczne, które zaprocentują w przyszłych, znacznie większych aplikacjach.

A co z parametrami tych funkcji, a dokładniej z jedynym argumentem procedury `Ruch()`? Wydaje mi się, iż łatwo jest dociec jego znaczenia: to bowiem elementarna wielkość, opisująca posunięcie zamierzone przez gracza. Jej sens został już zaprezentowany przy okazji projektu interfejsu użytkownika: chodzi po prostu o wprowadzony z klawiatury numer pola, na którym ma być postawione kółko lub krzyżyk.

Zaczynamy

Skoro wiemy już dokładnie, jak wyglądają wizytówki naszych funkcji oraz z grubsza znamy należyte algorytmy ich działania, napisanie odpowiedniego kodu powinno być po prostu dziecinną igraszką, prawda?… :) Dobre samopoczucie może się jednak okazać przedwczesne, gdyż na twoim obecnym poziomie zaawansowania zadanie to wcale nie należy do najłatwiejszych. Nie zostawię cię jednak bez pomocy!

Dla szczególnie ambitnych proponuję aczkolwiek samodzielne dokończenie całego programu, a następnie porównanie go z kodem dołączonym do kursu. Samodzielne rozwiązywanie problemów jest bowiem istotą i najlepszą drogą nauki programowania!
Podczas zmagania się z tym wyzwaniem możesz jednak (i zapewne będziesz musiał) korzystać z innych źródeł informacji na temat programowania w C++, na przykład MSDN. Wiadomościami, które niemal na pewno okażą ci się przydatne, są dokładne informacje o plikach nagłówkowych i związanej z nimi dyrektywie `#include` oraz słowie kluczowym `extern`. Poszukaj ich w razie napotkania nieprzewidzianych trudności…
Jeżeli poradzisz sobie z tym niezwykle trudnym zadaniem, będziesz mógł być z siebie niewypowiedzianie dumny :D Nagrodą będzie też cenne doświadczenie, którego nie zdobędziesz inną drogą!

Mamy więc zamiar pisać instrukcje stanowiące blok kodu funkcji, przeto powinniśmy umieścić je wewnątrz modułu, a nie pliku nagłówkowego. Dlatego też chwilowo porzucamy *game.h* i otwieramy nieskażony jeszcze żadnym znakiem plik *game.cpp*.
Nie znaczy to wszak, że nie będziemy naszego nagłówka w ogóle potrzebować. Przeciwnie, jest ona nam niezbędny - zawiera przecież definicje trzech typów wyliczeniowych, bez których nie zdołamy się obejść.
Powinniśmy zatem dołączyć go do naszego modułu przy pomocy poznanej jakiś czas temu i stosowanej nieustannie dyrektywy `#include`:

```
#include "game.h"
```

Zwróćmy uwagę, iż, inaczej niż to mamy w zwyczaju, ujęliśmy nazwę pliku nagłówkowego w **cudzysłowy** zamiast nawiasów ostrych. Jest to konieczne; w ten sposób należy zaznaczać nasze własne nagłówki, aby odróżnić je od „fabrycznych” (*iostream*, *cmath* itp.)

Nazwę dołączanego pliku nagłówkowego należy umieszczać w cudzysłowach (`""`), jeśli jest on w tym samym katalogu co moduł, do którego chcemy go dołączyć. Może być on także w jego pobliżu (nad- lub podkatalogu) - wtedy używa się względnej ścieżki do pliku (np. `"..\plik.h"`).

Dołączenie własnego nagłówka nie zwalnia nas jednak od wykonania tej samej czynności na dwóch innych tego typu plikach:

```
#include <iostream>
#include <ctime>
```

Są one konieczne do prawidłowego funkcjonowania kodu, który napiszemy za chwilę.

Deklarujemy zmienne

Włączając plik nagłówkowy *game.h* mamy do dyspozycji zdefiniowane w nim typy `SIGN`, `FIELD` i `GAMESTATE`. Logiczne będzie więc zadeklarowanie należących doń zmiennych `g_aPlansza`, `g_StanGry` i `g_AktualnyGracz`:

```
FIELD g_aPlansza[3][3] = { { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY } };
GAMESTATE g_StanGry = GS_NOTSTARTED;
SIGN g_AktualnyGracz;
```

Skorzystamy z nich niejednokrotnie w kodzie modułu *game.cpp*, zatem powyższe linijki należy umieścić poza wszelkimi funkcjami.

Funkcja `StartGry()`

Nie jest to trudne, skoro nie napisaliśmy jeszcze absolutnie żadnej funkcji :) Niezwłocznie więc zabieramy się do pracy. Rozpoczniemy od tej procedury, która najszybciej da o sobie znać w gotowym programie - czyli `StartGry()`.
Jak pamiętamy, jej rolą jest przede wszystkim wylosowanie gracza, który rozpocznie rozgrywkę. Wcześniej jednak przydałoby się, aby funkcja sprawdziła, czy jest wywoływana w odpowiednim momencie - gdy gra faktycznie się jeszcze nie zaczęła:

```
if (g_StanGry != GS_NOTSTARTED) return false;
```

Jeżeli warunek ten nie zostanie spełniony, funkcja zwróci wartość wskazującą na niepowodzenie swych działań.
Jakich działań? Nietrudno zapisać je w postaci kodu C++:

```
// losujemy gracza, który będzie zaczynał
srand (static_cast<unsigned>(time(NULL)));
g_AktualnyGracz = (rand() % 2 == 0 ? SGN_CIRCLE : SGN_CROSS);

// ustawiamy stan gry na ruch graczy
g_StanGry = GS_MOVE;
```

Losowanie liczby z przedziału `<0; 2)` jest nam czynnością na wskroś znajomą. W połączeniu z operatorem warunkowym `?:` pozwala na realizację pierwszego z celów funkcji. Drugi jest tak elementarny, że w ogóle nie wymaga komentarza. W końcu nie od dziś stykamy się z przypisaniem wartości do zmiennej :)
To już wszystko, co było przewidziane do zrobienia przez naszą funkcję `StartGry()`. W pełni usatysfakcjonowani możemy więc zakończyć ją zwróceniem informacji o pozytywnym rezultacie podjętych akcji:

```
return true;
```

Wywołujący otrzyma więc wiadomość o tym, że czynności zlecone funkcji zostały zakończone z sukcesem.

Funkcja `Ruch()`

Kolejną funkcją, na której spocznie nasz wzrok, jest `Ruch()`. Ma ona za zadanie umieścić w podanym polu znak aktualnego gracza (kółko lub krzyżyk) oraz sprawdzić stan planszy pod kątem ewentualnej wygranej któregoś z graczy lub remisu. Całkiem sporo do zrobienia, zatem do pracy, rodacy! ;D

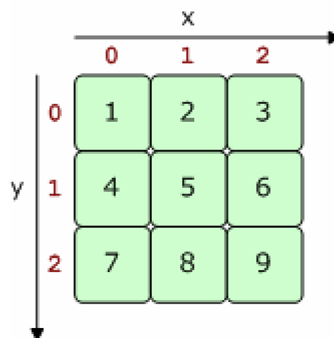
Pamiętamy oczywiście, że rzeczona funkcja ma przyjmować jeden parametr typu `unsigned`, więc jej szkielet wyglądać będzie następująco:

```
bool Ruch(unsigned uNumerPola)
{
   // ...
}
```

Na początku dokonamy tutaj podobnej co poprzednio kontroli ewentualnego błędu w postaci złego stanu gry. Dodamy jeszcze warunek sprawdzający, czy zadany numer pola zawiera się w przedziale `<1; 9>`. Całość wygląda następująco:

```
if (g_StanGry != GS_MOVE) return false;
if (!(uNumerPola >= 1 && uNumerPola <= 9)) return false;
```

Jeżeli punkt wykonania pokona obydwie te przeszkody, należałoby uczynić ruch, o który użytkownik (za pośrednictwem parametru `uNumerPola`) prosi. W tym celu konieczne jest przeliczenie, zamieniające pojedynczy numer pola (z zakresu od 1 do 9) na dwa indeksy naszej tablicy `g_aPlansza` (każdy z przedziału od 0 do 2). Pomocy może nam tu udzielić wizualny diagram, na przykład taki:



Schemat 15. Numerowanie pól planszy do gry w kółko i krzyżyk

Odpowiednie formułki, wyliczające współrzędną pionową (`uY`) i poziomą (`uX`) można napisać, wykorzystując dzielenie całkowitoliczbowe oraz resztę z niego:

```
unsigned uY = (uNumerPola - 1) / 3;
unsigned uX = (uNumerPola - 1) % 3;
```

Odjęcie jedynki jest spowodowane faktem, iż w C++ tablice są indeksowane od zera (poza tym jest to dobra okazja do przypomnienia tej ważnej kwestii :D).
Mając już obliczone oba indeksy, możemy spróbować postawić symbol aktualnego gracza w podanym polu. Uda się to jednak wyłącznie wtedy, gdy nikt nas tutaj nie uprzedził - a więc kiedy wskazane pole jest puste, co kontrolujemy dodatkowym testem:

```
if (g_aPlansza[uY][uX] == FLD_EMPTY)
   // wstaw znak aktualnego gracza w podanym polu
else
   return false;
```

Jeśli owa kontrola się powiedzie, musimy zrealizować zamierzenie i wstawić kółko lub krzyżyk - zależnie do tego, który gracz jest teraz uprawniony do ruchu - w żądanie miejsce. Informację o aktualnym graczu przechowuje rzecz jasna zmienna `g_AktualnyGracz`. Niemożliwe jest jednak jej zwykłe przypisanie w rodzaju:

```
g_aPlansza[uY][uX] = g_AktualnyGracz;
```

Wystąpiłby tu bowiem konflikt typów, gdyż `FIELD` i `SIGN` są typami wyliczeniowymi, nijak ze sobą niekompatybilnymi. Czyżbyśmy musieli zatem uciec się do topornej instrukcji `switch`?
Odpowiedź na szczęście brzmi nie. Inne, lepsze rozwiązanie polega na „dopasowaniu” do siebie stałych obu typów, reprezentujących kółko i krzyżyk. Niech będą one sobie równe; w tym celu zmodyfikujemy definicję `FIELD` (w pliku *game.h*):

```
enum FIELD {  FLD_EMPTY,
              FLD_CIRCLE   = SGN_CIRCLE,
              FLD_CROSS    = SGN_CROSS };
```

Po tym zabiegu cała operacja sprowadza się do zwykłego rzutowania:

```
g_aPlansza[uY][uX] = static_cast<FIELD>(g_AktualnyGracz);
```

Liczbowe wartości obu zmiennych będą się zgadzać, ale interpretacja każdej z nich będzie odmienna. Tak czy owak, osiągnęliśmy obrany cel, więc wszystko jest w porządku :)

Niedługo zresztą ponownie skorzystamy z tej prostej i efektywnej sztuczki.

Nasza funkcja wykonuje już połowę zadań, do których ją przeznaczyliśmy. Niestety, mniejszą połowę :D Oto bowiem mamy przed sobą znacznie poważniejsze wyzwanie niż kilka `if`’ów, a mianowicie zaprogramowanie algorytmu lustrującego planszę i stwierdzającego na jej podstawie ewentualną wygraną któregoś z graczy lub remis. Trzeba więc zakasać rękawy i wytężyć intelekt…

Zajmijmy się na razie wykrywaniem zwycięstw. Doskonale chyba wiemy, że do wygranej w naszej grze potrzebne jest graczowi utworzenie z własnych znaków linii poziomej, pionowej lub ukośnej, obejmującej trzy pola. Łącznie mamy więc osiem możliwych linii, a dla każdej po trzy pola opisane dwiema współrzędnymi. Daje nam to, bagatelka, 48 warunków do zakodowania, czyli 8 makabrycznych instrukcji `if` z sześcioczłonowymi (!) wyrażeniami logicznymi w każdej! Brr, brzmi to wręcz okropnie…
Jak to jednak nierzadko bywa, istnieje rozwiązanie alternatywne, które jest z reguły lepsze :) Tym razem jest nim użycie tablicy przeglądowej, w którą wpiszemy wszystkie wygrywające zestawy pól: **osiem** linii po **trzy** pola po **dwie** współrzędne daje nam ostatecznie taką oto, nieco zakręconą, stałą[60]:

```
const LINIE[][3][2] =  { { { 0,0 }, { 0,1 }, { 0,2 } }, // górna pozioma
                         { { 1,0 }, { 1,1 }, { 1,2 } },// środ. pozioma
                         { { 2,0 }, { 2,1 }, { 2,2 } },// dolna pozioma
                         { { 0,0 }, { 1,0 }, { 2,0 } }, // lewa pionowa
                         { { 0,1 }, { 1,1 }, { 2,1 } }, // środ. pionowa
                         { { 0,2 }, { 1,2 }, { 2,2 } }, // prawa pionowa
                         { { 0,0 }, { 1,1 }, { 2,2 } }, // p. backslashowa
                         { { 2,0 }, { 1,1 }, { 0,2 } } }; // p. slashowa
```

Przy jej deklarowaniu korzystaliśmy z faktu, iż w takich wypadkach pierwszy wymiar tablicy można pominąć, lecz równie poprawne byłoby wpisanie tam `8` *explicité*.
A zatem mamy już tablicę **przeglądową**… Przydałoby się więc jakoś ją **przeglądać** :)
Oprócz tego mamy jednak dodatkowy cel, czyli znalezienie linii wypełnionej tymi samymi znakami, nasze przeglądanie będzie wobec tego nieco skomplikowane i przedstawia się następująco:

[60] Brak nazwy typu w deklaracji zmiennej sprawia, iż będzie należeć ona do domyślnego typu `int`. Tutaj oznacza to, że elementy naszej tablicy będą liczbami całkowitymi.

```
FIELD Pole, ZgodnePole;
unsigned uLiczbaZgodnychPol;
for (int i = 0; i < 8; ++i)
{
   // i przebiega po kolejnych możliwych liniach (jest ich osiem)

   // zerujemy zmienne pomocnicze
   Pole = ZgodnePole = FLD_EMPTY;      // obie zmienne == FLD_EMPTY
   uLiczbaZgodnychPol = 0;

   for (int j = 0; j < 3; ++j)
   {
         // j przebiega po trzech polach w każdej linii

         // pobieramy rzeczone pole
         // to zdecydowanie najbardziej pogmatwane wyrażenie :)
         Pole = g_aPlansza[LINIE[i][j][0]][LINIE[i][j][1]];

         // jeśli sprawdzane pole różne od tego, które ma się zgadzać...
         if (Pole != ZgodnePole)
         {
               // to zmieniamy zgadzane pole na to aktualne
               ZgodnePole = Pole;
               uLiczbaZgodnychPol = 1;
         }
         else
               // jeśli natomiast oba pola się zgadzają, no to
               // inkrementujemy licznik takich zgodnych pól
               ++uLiczbaZgodnychPol;
   }

   // teraz sprawdzamy, czy udało nam się zgodzić linię
   if (uLiczbaZgodnychPol == 3 && ZgodnePole != FLD_EMPTY)
   {
         // jeżeli tak, no to ustawiamy stan gry na wygraną
         g_StanGry = GS_WON;

         // przerywamy pętlę i funkcję
         return true;
   }
}
```

„No nie” - powiesz pewnie - „Teraz to już przesadziłeś!” ;) Ja jednak upieram się, iż nie całkiem masz rację, a podany algorytm tylko wygląda strasznie, lecz w istocie jest bardzo prosty.
Na początek deklarujemy sobie trzy zmienne pomocnicze, które wydatnie przydadzą się w całej operacji. Szczególną rolę spełnia tu `uLiczbaZgodnychPol`; jej nazwa mówi wiele. Zmienna ta będzie przechowywała liczbę identycznych pól w aktualnie badanej linii - wartość równa 3 stanie się więc podstawą do stwierdzenia obecności wygrywającej kombinacji znaków.
Dalej przystępujemy do sprawdzania wszystkich ośmiu interesujących sytuacji, determinujących ewentualne zwycięstwo. Na scenę wkracza więc pętla `for`; na początku jej cyklu dokonujemy zerowania wartości zmiennych pomocniczych, aby potem… wpaść w kolejną pętlę :) Ta jednak będzie przeskakiwała po trzech polach każdej ze sprawdzanych linii:

```
for (int j = 0; j < 3; ++j)
{
   Pole = g_aPlansza[LINIE[i][j][0]][LINIE[i][j][1]];
```

```
      if (Pole != ZgodnePole)
      {
            ZgodnePole = Pole;
            uLiczbaZgodnychPol = 1;
      }
      else
            ++uLiczbaZgodnychPol;
   }
```

Koszmarnie wyglądająca pierwsza linijka bloku powyższej pętli nie będzie wydawać się aż tak straszne, jeśli uświadomimy sobie, iż `LINIE[i][j][0]` oraz `LINIE[i][j][1]` to odpowiednio: współrzędna pionowa oraz pozioma `j`-tego pola `i`-tej potencjalnie wygrywającej linii. Słusznie więc używamy ich jako indeksów tablicy `g_aPlansza`, pobierając stan pola do sprawdzenia.
Następująca dalej instrukcja warunkowa rozstrzyga, czy owe pole zgadza się z ewentualnymi poprzednimi - tzn. jeżeli na przykład poprzednio sprawdzane pole zawierało kółko, to aktualne także powinno mieścić ten symbol. W przypadku gdy warunek ten nie jest spełniony, sekwencja zgodnych pól „urywa się”, co oznacza w tym wypadku wyzerowanie licznika `uLiczbaZgodnychPol`. Sytuacja przeciwstawna - gdy badane pole jest już którymś z kolei kółkiem lub krzyżykiem - skutkuje naturalnie zwiększeniem tegoż licznika o jeden.
Po zakończeniu całej pętli (czyli wykonaniu trzech cykli, po jednym dla każdego pola) następuje kontrola otrzymanych rezultatów. Najważniejszym z nich jest wspomniany licznik `uLiczbaZgodnychPol`, którego wartość konfrontujemy z trójką. Jednocześnie sprawdzamy, czy „zgodzone” pole nie jest przypadkiem polem pustym, bo przecież z takiej zgodności nic nam nie wynika. Oba te testy wykonuje instrukcja:

```
if (uLiczbaZgodnychPol == 3 && ZgodnePole != FLD_EMPTY)
```

Spełnienie tego warunku daje pewność, iż mamy do czynienia z prawidłową sekwencją trzech kółek lub krzyżyków. Słusznie więc możemy wtedy przyznać palmę zwycięstwa aktualnemu graczowi i zakończyć całą funkcję:

```
g_StanGry = GS_WON;
return true;
```

W przeciwnym wypadku nasza główna pętla się zapętla w swym kolejnym cyklu i bada w nim kolejną ustaloną linię symboli - i tak aż do znalezienia pasującej kolumny, rzędu lub przekątnej albo wyczerpania się tablicy przeglądowej `LINIE`.

Uff?… Nie, to jeszcze nie wszystko! Nie zapominajmy przecież, że zwycięstwo nie jest jedynym możliwych rozstrzygnięciem rozgrywki. Drugim jest **remis** - zapełnienie wszystkich pól planszy symbolami graczy bez utworzenia żadnej wygrywającej linii.
Jak obsłużyć taką sytuację? Wbrew pozorom nie jest to wcale trudne, gdyż możemy wykorzystać do tego fakt, iż przebycie przez program poprzedniej, wariackiej pętli oznacza nieobecność na planszy żadnych ułożeń zapewniających zwycięstwo. Niejako „z miejsca” mamy więc spełniony pierwszy warunek konieczny do remisu.
Drugi natomiast - szczelne wypełnienie całej planszy - jest bardzo łatwy do sprawdzenia i wymagania jedynie zliczenia wszystkich niepustych jej pól:

```
unsigned uLiczbaZapelnionychPol = 0;

for (int i = 0; i < 3; ++i)
   for (int j = 0; j < 3; ++j)
         if (g_aPlansza[i][j] != FLD_EMPTY)
               ++uLiczbaZapelnionychPol;
```

Jeżeli jakimś dziwnym sposobem ilość ta wyniesie 9, znaczyć to będzie, że gra musi się zakończyć z powodu braku wolnych miejsc :) W takich okolicznościach wynikiem rozgrywki będzie tylko mało satysfakcjonujący remis:

```
if (uLiczbaZapelnionychPol == 3*3)
{
   g_StanGry = GS_DRAW;
   return true;
}
```

W taki oto sposób wykryliśmy i obsłużyliśmy obydwie sytuacje „wyjątkowe", kończące grę - zwycięstwo jednego z graczy lub remis. Pozostało nam jeszcze zajęcie się bardziej zwyczajnym rezultatem wykonania ruchu, kiedy to nie powoduje on żadnych dodatkowych efektów. Należy wtedy przekazać prawo do posunięcia drugiemu graczowi, co też czynimy:

```
g_AktualnyGracz = (g_AktualnyGracz == SGN_CIRCLE ?
                   SGN_CROSS : SGN_CIRCLE);
```

Przy pomocy operatora warunkowego zmieniamy po prostu znak aktualnego gracza na przeciwny (z kółka na krzyżyk i odwrotnie), osiągając zamierzony skutek.
Jest to jednocześnie ostatnia czynność funkcji `Ruch()`! Wreszcie, po długich bojach i bólach głowy ;) możemy ją zakończyć zwróceniem bezwarunkowo pozytywnego wyniku:

```
return true;
```

a następnie udać się po coś do jedzenia ;-)

Funkcja `RysujPlansze()`

Jako ostatnią napiszemy funkcję, której zadaniem będzie wyświetlenie na ekranie (czyli w oknie konsoli) bieżącego stanu gry:

Screen 34. Ekran gry w kółko i krzyżyk

Najważniejszą jego składową będzie naturalnie osławiona plansza, o zajęcie której toczą boje nasi dwaj gracze. Oprócz niej można jednak wyróżnić także kilka innych elementów. Wszystkie one będą „rysowane" przez funkcję `RysujPlansze()`. Niezwłocznie więc rozpocznijmy jej implementację!

Tradycyjnie już pierwsze linijki są szukaniem dziury w całym, czyli potencjalnego błędu. Tym razem usterką będzie wywołanie kodowanej właśnie funkcji przez rozpoczęciem właściwego pojedynku, gdyż w tej sytuacji nie ma w zasadzie nic do pokazania. Logiczną konsekwencją jest wtedy przerwanie funkcji:

```
if (g_StanGry == GS_NOTSTARTED)  return false;
```

Jako że jednak wierzymy w rozsądek programisty wywołującego pisaną teraz funkcję (czyli *nomen-omen* w swój własny :D), przejdźmy raczej do kodowania jej właściwej części „rysującej".
Od czego zaczniemy? Odpowiedź nie jest szczególnie trudna; co ciekawe, w przypadku każdej innej gry i jej odpowiedniej funkcji byłaby taka sama. Rozpoczniemy bowiem od wyczyszczenia całego ekranu (czyli konsoli) - tak, aby mieć wolny obszar działania. Dokonamy tego poprzez polecenie systemowe CLS, które wywołamy funkcją C++ o nazwie `system()`:

```
system ("cls");
```

Mając oczyszczone przedpole przystępujemy do zasadniczego rysowania. Ze względu na specyfikę tekstowej konsoli zmuszeni jesteśmy do zapełniania jej wierszami, od góry do dołu. Nie powinno nam to jednak zbytnio przeszkadzać.
Na samej górze umieścimy tytuł naszej gry, stały i niezmienny. Kod odpowiedzialny za tę czynność przedstawia się więc raczej trywialnie:

```
std::cout << "   KOLKO I KRZYZYK   " << std::endl;
std::cout << "---------------------" << std::endl;
std::cout << std::endl;
```

Żądnych wrażeń pocieszam jednak, iż dalej będzie już ciekawiej :) Oto mianowicie przystępujemy do prezentacji planszy w postaci tekstowej - z zaznaczonymi kółkami i krzyżykami postawionymi przez graczy oraz numerami wolnych pól. Operację tą przeprowadzamy w sposób następujący:

```
std::cout << "        -----" << std::endl;
for (int i = 0; i < 3; ++i)
{
   // lewa część ramki
   std::cout << "        |";

   // wiersz
   for (int j = 0; j < 3; ++j)
   {
         if (g_aPlansza[i][j] == FLD_EMPTY)
               // numer pola
               std::cout << i * 3 + j + 1;
         else
               // tutaj wyświetlamy kółko lub krzyżyk... ale jak? :)
   }

   // prawa część ramki
   std::cout << "|" << std::endl;
}
std::cout << "        -----" << std::endl;
std::cout << std::endl;
```

Cały kod to oczywiście znowu dwie zagnieżdżone pętle `for` - stały element pracy z dwuwymiarową tablicą. Zewnętrzna przebiega po poszczególnych wierszach planszy, zaś wewnętrzna po jej pojedynczych polach.
Wyświetlenie takiego pola oznacza pokazanie albo jego numerku (jeżeli jest puste), albo dużej litery O lub X, symulującej wstawione weń kółko lub krzyżyk. Numerek wyliczamy poprzez prostą formułkę `i * 3 + j + 1` (dodanie jedynki to znowuż kwestia indeksów liczonych od zera), w której `i` jest numerem wiersza, zaś `j` - kolumny. Cóż jednak zrobić z drugim przypadkiem - zajętym polem? Musimy przecież rozróżnić kółka i krzyżyki…
Można oczywiście skorzystać z instrukcji `if` lub operatora `?:`, jednak już raz zastosowaliśmy lepsze rozwiązanie. Dopasujmy mianowicie stałe typu `FIELD` (każdy

element tablicy g_aPlansza należy przecież do tego typu) do znaków 'O' i 'X'. Przypatrzmy się najpierw definicji rzeczonego typu:

```
enum FIELD {  FLD_EMPTY,
              FLD_CIRCLE   = SGN_CIRCLE,
              FLD_CROSS    = SGN_CROSS };
```

Widać nim skutek pierwszego zastosowania sztuczki, z której chcemy znowu skorzystać. Dotyczy on zresztą interesujących nas stałych FLD_CIRCLE i FLD_CROSS, równych odpowiednio SGN_CIRCLE i SGN_CROSS. Czy to oznacza, iż z triku nici?
Bynajmniej nie. Nie możemy wprawdzie bezpośrednio zmienić wartości interesujących nas stałych, ale możliwe jest „sięgnięcie do źródeł" i zmodyfikowanie SGN_CIRCLE oraz SGN_CROSS, zadeklarowanych w typie SIGN:

```
enum SIGN { SGN_CIRCLE = 'O', SGN_CROSS = 'X' };
```

Tą drogą, pośrednio, zmienimy też wartości stałych FLD_CIRCLE i FLD_CROSS, przypisując im kody ANSI wielkich liter „O" i „X". Teraz już możemy skorzystać z rzutowania na typ char, by wyświetlić niepuste pole planszy:

```
std::cout << static_cast<char>(g_aPlansza[i][j]);
```

Kod rysujący obszar rozgrywki jest tym samym skończony.

Pozostał nam jedynie komunikat o stanie gry, wyświetlany najniżej. Zależnie od bieżących warunków (wartości zmiennej g_StanGry) może on przyjmować formę prośby o wpisanie kolejnego ruchu lub też zwyczajnej informacji o wygranej lub remisie:

```
switch (g_StanGry)
{
   case GS_MOVE:
         // prośba o następny ruch
         std::cout << "Podaj numer pola, w ktorym" << std::endl;
         std::cout << "chcesz postawic ";
         std::cout << (g_AktualnyGracz == SGN_CIRCLE ?
                       "kolko" : "krzyzyk") << ": ";
         break;
   case GS_WON:
         // informacja o wygranej
         std::cout << "Wygral gracz stawiajacy ";
         std::cout << (g_AktualnyGracz == SGN_CIRCLE ?
                       "kolka" : "krzyzyki") << "!";
         break;
   case GS_DRAW:
         // informacja o remisie
         std::cout << "Remis!";
         break;
}
```

Analizy powyższego kodu możesz z łatwością dokonać samodzielnie[61].

Na tymże elemencie „scenografii" kończymy naszą funkcję RysujPlansze(), wieńcząc ją oczywiście zwyczajowym oddaniem wartości true:

[61] A jakże! Już coraz rzadziej będę omawiał podobnie elementarne kody źródłowe, będące prostym wykorzystaniem doskonale ci znanych konstrukcji języka C++. Jeżeli solennie przykładałeś się do nauki, nie powinno być to dla ciebie żadną niedogodnością, zaś w zamian pozwoli na dogłębne zajęcie się nowymi zagadnieniami bez koncentrowania większej uwagi na banałach.

```
return true;
```

Możemy na koniec zauważyć, iż pisząc tą funkcję uporaliśmy się jednocześnie z elementem programu o nazwie „interfejs użytkownika” :D

Funkcja `main()`, czyli składamy program

Być może trudno w to uwierzyć, ale mamy za sobą zaprogramowanie wszystkich funkcji sterujących przebiegiem gry! Zanim jednak będziemy mogli cieszyć się działającym programem musimy wypełnić kodem główną funkcję aplikacji, od której zacznie się jej wykonywanie - `main()`.

W tym celu zostawmy już wymęczony moduł *game.cpp* i wróćmy do *main.cpp*, w którym czeka nietknięty szkielet funkcji `main()`. Poprzedzimy go najpierw dyrektywami dołączenia niezbędnych nagłówków - także naszego własnego, *game.h*:

```
#include <iostream>
#include <conio.h>
#include "game.h"
```

Własne pliki nagłówkowe najlepiej umieszczać na końcu szeregu instrukcji `#include`, dołączając je po tych pochodzących od kompilatora.

Teraz już możemy zająć się treścią najważniejszej funkcji w naszym programie. Zaczniemy od następującego wywołania:

```
StartGry();
```

Spowoduje ono rozpoczęcie rozgrywki - jak pamiętamy, oznacza to między innymi wylosowanie gracza, któremu przypadnie pierwszy ruch, oraz ustawienie stanu gry na `GS_MOVE`.
Od tego momentu zaczyna się więc zabawa, a nam przypada obowiązek jej prawidłowego poprowadzenia. Wywiążemy się z niego w nieznany dotąd sposób - użyjemy **pętli nieskończonej**:

```
for (;;)
{
   // ...
}
```

Konstrukcja ta wcale nie jest taka dziwna, a w grach spotyka się ją bardzo często. Istota pętli nieskończonej jest częściowo zawarta w jej nazwie, a po części można ją wydedukować ze składni. Mianowicie, nie posiada ona żadnego warunku zakończenia[62], więc w zasadzie wykonywałaby się do końca świata i o jeden dzień dłużej ;) Aby tego uniknąć, należy gdzieś wewnątrz jej bloku umieścić instrukcję `break;`, która spowoduje przerwanie tego zaklętego kręgu. Uczynimy to, kodując kolejne instrukcje w tejże pętli. Najpierw funkcja `RysujPlansze()` wyświetli nam aktualny stan rozgrywki:

```
RysujPlansze();
```

Pokaże więc tytuł gry, planszę oraz dolny komunikat - komunikat, który przez większość czasu będzie prośbą o kolejny ruch. By sprawdzić, czy tak jest w istocie, porównamy zmienną opisującą stan gry z wartością `GS_MOVE`:

[62] Zwanego też czasem warunkiem terminalnym.

```
if (g_StanGry == GS_MOVE)
{
   unsigned uNumerPola;
   std::cin >> uNumerPola;
   Ruch (uNumerPola);
}
```

Pozytywny wynik wspomnianego testu słusznie skłania nas do użycia strumienia wejścia i pobrania od użytkownika numeru pola, w które chce wstawić swoje kółko lub krzyżyk. Przekazujemy go potem do funkcji `Ruch()`, serca naszej gry.
Następujące po sobie posunięcia graczy, czyli kolejne cykle pętli, doprowadzą w końcu do rozstrzygnięcia rozgrywki - czyjejś wygranej albo obustronnego remisu. I to jest właśnie warunek, na który czekamy:

```
else if (g_StanGry == GS_WON || g_StanGry == GS_DRAW)
   break;
```

Przerywamy wtedy pętlę, zostawiając na ekranie końcowy stan planszy oraz odpowiedni komunikat. Aby użytkownicy mieli szansę go zobaczyć, stosujemy rzecz jasna funkcję `getch()`:

```
getch();
```

Po odebraniu wciśnięcia dowolnego klawisza program może się już ze spokojem zamknąć ;)

Uroki kompilacji

Fanfary! Zdaje się, że właśnie zakończyliśmy kodowanie naszego wielkiego projektu! Nareszcie zatem możemy przeprowadzić jego kompilację i uzyskać gotowy do uruchomienia plik wykonywalny.
Zróbmy więc to! Uruchom Visual Studio (jeżeli je przypadkiem zamknąłeś), otwórz swój projekt, zamknij drzwi i okna, wyprowadź zwierzęta domowe, włącz automatyczną sekretarkę i wciśnij klawisz F7 (lub wybierz pozycję menu *Build*|*Build Solution*)…

***

Co się stało? Wygląda na to, że nie wszystko udało się tak dobrze, jak tego oczekiwaliśmy. Zamiast działającej aplikacji kompilator uraczył nas czterema błędami:

```
c:\Programy\TicTacToe\main.cpp(20) : error C2065: 'g_StanGry' : undeclared identifier
c:\Programy\TicTacToe\main.cpp(20) : error C2677: binary '==' : no global operator found which takes type 'GAMESTATE' (or there is no acceptable conversion)
c:\Programy\TicTacToe\main.cpp(28) : error C2677: binary '==' : no global operator found which takes type 'GAMESTATE' (or there is no acceptable conversion)
c:\Programy\TicTacToe\main.cpp(28) : error C2677: binary '==' : no global operator found which takes type 'GAMESTATE' (or there is no acceptable conversion)
```

Wszystkie one dotyczą tego samego, ale najwięcej mówi nam pierwszy z nich. Dwukrotnie kliknięcie na dotyczący go komunikat przeniesie nas bowiem do linijki:

```
if (g_StanGry == GS_MOVE)
```

Występuje w niej nazwa zmiennej `g_StanGry`, która, sądząc po owym komunikacie, jest tutaj uznawana za **niezadeklarowaną**…
Ale dlaczego?! Przecież z pewnością umieściliśmy jej deklarację w kodzie programu. Co więcej, stale korzystaliśmy z tejże zmiennej w funkcjach `StartGry()`, `Ruch()` i

`RysujPlansze()`, do których kompilator nie ma najmniejszych zastrzeżeń. Czyżby więc tutaj dopadła go nagła amnezja?

Wyjaśnienie tego, jak by się wydawało, dość dziwnego zjawiska jest jednak w miarę logiczne. Otóż `g_StanGry` została zadeklarowana wewnątrz modułu *game.cpp*, więc jej zasięg ogranicza się **jedynie** do tegoż modułu. Funkcja `main()`, znajdująca się w pliku *main.cpp*, jest poza tym zakresem, zatem dla niej rzeczona zmienna po prostu **nie istnieje**. Nic dziwnego, iż kompilator staje się wobec nieznanej nazwy `g_StanGry` zupełnie bezradny.
Nasuwa się oczywiście pytanie: jak zaradzić temu problemowi? Co zrobić, aby nasza zmienna była dostępna wewnątrz funkcji `main()`?... Chyba najszybciej pomyśleć można o przeniesieniu jej deklaracji w obszar **wspólny** dla obu modułów *game.cpp* oraz *main.cpp*. Takim współdzielonym terenem jest naturalnie plik nagłówkowy *game.h*. Czy należy więc umieścić tam deklarację `GAMESTATE g_StanGry = GS_NOTSTARTED;`?
Niestety, nie jest to poprawne. Musimy bowiem wiedzieć, że zmienna **nie może** rezydować wewnątrz nagłówka! Jej prawidłowe zdefiniowanie powinno być zawsze umieszczone w module kodu. W przeciwnym razie każdy moduł, który dołączy plik nagłówkowy z definicją zmiennej, stworzy swoją **własną kopię** tejże! U nas znaczyłoby to, że zarówno *main.cpp*, jak i *game.cpp* posiadają zmienne o nazwach `g_StanGry`, ale są one od siebie całkowicie **niezależne** i „nie wiedzą o sobie nawzajem”!

Definicja musi zatem pozostać na swoim miejscu, ale plik nagłówkowy niewątpliwie nam się przyda. Mianowicie, wpiszemy doń następującą linijkę:

```
extern GAMESTATE g_StanGry;
```

Jest to tak zwana **deklaracja zapowiadająca** zmiennej. Jej zadaniem jest poinformowanie kompilatora, że **gdzieś** w programie[63] istnieje zmienna o podanej nazwie i typie. Deklaracja ta **nie tworzy** żadnego nowego bytu ani nie rezerwuje dlań miejsca w pamięci operacyjnej, lecz jedynie **zapowiada** (stąd nazwa), iż czynność ta zostanie wykonana. Obietnica ta może być spełniona podczas kompilacji lub (tak jak u nas) dopiero w czasie linkowania.
Z praktycznego punktu widzenia deklaracja `extern` (ang. *external* - zewnętrzny) pełni bardzo podobną rolę, co prototyp funkcji. Podaje bowiem jedynie minimum informacji, potrzebnych do skorzystania z deklarowanego tworu bez marudzenia kompilatora, a jednocześnie odkłada jego właściwą definicję w inne miejsce i/lub czas.

Deklaracja zapowiadająca (ang. *forward declaration*) to częściowe określenie jakiegoś programistycznego bytu. Nie definiuje dokładnie wszystkich jego aspektów, ale wystarcza do skorzystania z niego wewnątrz zakresu umieszczenia deklaracji.
Przykładem może być prototyp funkcji czy użycie słowa `extern` dla zmiennej.

Umieszczenie powyższej deklaracji w pliku nagłówkowym *game.h* udostępnia zatem zmienną `g_StanGry` wszystkim modułom, które dołączą wspomniany nagłówek. Tym samym jest już ona znana także funkcji `main()`, więc ponowna kompilacja powinna przebiec bez żadnych problemów.

Czujny czytelnik zauważył pewnie, że dość swobodnie operuję terminami „deklaracja” oraz „definicja”, używając ich zamiennie. Niektórzy puryści każą jednak je rozróżniać. Według nich jedynie to, co nazwaliśmy przed momentem „deklaracja zapowiadającą”, można nazwać krótko „deklaracją”. „Definicją” ma być za to dokładne sprecyzowanie cech danego obiektu, oraz, przede wszystkim, przygotowanie dla niego miejsca w pamięci operacyjnej.

[63] Mówiąc ściśle: gdzieś poza bieżącym zakresem.

Zgodnie z taką terminologią instrukcje w rodzaju `int nX;` czy `float fY;` miałyby być „definicjami zmiennych”, natomiast `extern int nX;` oraz `extern float fY;` - „deklaracjami”. Osobiście twierdzę, że jest to jeden z najjaskrawszych przykładów szukania dziury w całym i prób niezmiernego gmatwania programistycznego słownika. Czy ktokolwiek przecież mówi o „definicjach zmiennych”? Pojęcie to brzmi tym bardziej sztucznie, że owe „definicje” nie przynoszą żadnych dodatkowych informacji w stosunku do „deklaracji”, a składniowo są od nich nawet krótsze!
Jak więc w takiej sytuacji nie nazwać spierania się o nazewnictwo zwyczajnym malkontenctwem? :)

Uruchamiamy aplikację

To niemalże niewiarygodne, jednak stało się faktem! Zakończyliśmy w końcu programowanie naszej gry! Wreszcie możesz więc użyć klawisza F5, by cieszyć tym oto wspaniałym widokiem:

Screen 35. Gra w „Kółko i krzyżyk” w akcji

A po kilkunastominutowym, zasłużonym relaksie przy własnoręcznie napisanej grze przejdź do dalszej części tekstu :)

Wnioski

Stworzyłeś właśnie (przy drobnej pomocy :D) swój pierwszy w miarę poważny program, w dodatku to, co lubimy najbardziej - czyli grę. Zdobyte przy tej okazji doświadczenie jest znacznie cenniejsze od najlepszego nawet, lecz tylko teoretycznego wykładu.

Warto więc podsumować naszą pracę, a przy okazji odpowiedzieć na pewne ogólne pytania, które być może przyszły ci na myśl podczas realizacji tego projektu.

Dziwaczne projektowanie

Tworzenie naszej gry rozpoczęliśmy od jej dokładnego zaprojektowania. Miało ono na celu wykreowanie komputerowego modelu znanej od dziesięcioleci gry dwuosobowej i zaadaptowanie go do potrzeb kodowania w C++.
W tym celu podzieliliśmy sobie zadanie na trzy części:
- określenie struktur danych wykorzystywanych przez aplikację
- sprecyzowanie wykonywanych przez nią czynności
- stworzenie interfejsu użytkownika

Aby zrealizować pierwsze dwie, musieliśmy przyjąć dość dziwną i raczej nienaturalną drogę rozumowania. Należało bowiem zapomnieć o takich „namacalnych” obiektach jak plansza, gracz czy rozgrywka. Zamiast tego mówiliśmy o pewnych **danych**, na których program miał wykonywać jakieś **operacje**.
Te dwa światy - statycznych informacji oraz dynamicznych działań - rozdzieliły nam owe „naturalne” obiekty związane z grą i kazały oddzielnie zajmować się ich cechami (jak np. symbole graczy) oraz realizowanymi przezeń czynnościami (np. wykonanie ruchu).

Podejście to, zwane **programowaniem strukturalnym**, mogło być dla ciebie trudne do zrozumienia i sztuczne. Nie martw się tym, gdyż podobnie uważa większość współczesnych koderów! Czy to znaczy, że programowanie jest udręką?
Domyślasz się pewnie, że wszystko co niedawno uczyniliśmy, dałoby się zrobić bardziej naturalnie i intuicyjne. Masz w tym całkowitą rację! Już w następnym rozdziale poznamy znacznie wygodniejszą i przyjaźniejszą technikę programowania, który zbliży kodowanie do ludzkiego sposobu myślenia.

Dość skomplikowane algorytmy

Kiedy już uporaliśmy się z projektowaniem, przyszedł czas na uruchomienie naszego ulubionego środowiska programistycznego i wpisanie kodu tworzonej aplikacji.

Jakkolwiek większość użytych przy tym konstrukcji języka C++ była ci znana od dawna, a duża część pozostałej mniejszości wprowadzona w tym rozdziale, sam kod nie należał z pewnością do elementarnych. Różnica między poprzednimi, przykładowymi programami była znacząca i widoczna niemal przez cały czas.
Na czym ona polegała? Po prostu język programowania przestał tu być **celem**, a stał się **środkiem**. Już nie tylko prężył swe „muskuły" i prezentował szeroki wachlarz możliwości. Stał się w pokornym sługą, który spełniał nasze wymagania w imię wyższego dążenia, którym było napisanie działającej i sensownej aplikacji.

Oczywiste jest więc, iż zaczęliśmy wymagać więcej także od siebie. Pisane algorytmy nie były już trywialnymi przepisami, wyważającymi otwarte drzwi. Wyżyny w tym względzie osiągnęliśmy chyba przy sprawdzaniu stanu planszy w poszukiwaniu ewentualnych sekwencji wygrywających. Zadanie to było swoiste i unikalne dla naszego kodu, dlatego też wymagało nieszablonowych rozwiązań. Takich, z jakimi będziesz się często spotykał.

Organizacja kodu

Ostatnia uwaga dotyczy porządku, jaki wprowadziliśmy w nasz kod źródłowy. Zamiast pojedynczego modułu zastosowaliśmy dwa i zintegrowaliśmy je przy pomocy własnego pliku nagłówkowego.

Nie obyło się rzecz jasna bez drobnych problemów, ale ogólnie zrobiliśmy to w całkowicie poprawny i efektywny sposób. Nie można też zapominać o tym, że jednocześnie poznaliśmy kolejny skrawek informacji na temat programowania w C++, tym razem dotyczący dyrektywy `#include`, prototypów funkcji oraz modyfikatora `extern`.

Drogi samodzielny programisto - ty, który dokończyłeś kod gry od momentu, w którym rozstaliśmy się nagłówkiem *game.h*, bez zaglądania do dalszej części tekstu!
Jeżeli udało ci się dokonać tego z zachowaniem założonej funkcjonalności programu oraz podziału kodu na trzy odrębne pliki, to naprawdę chylę czoła :) Znaczy to, że jesteś wręcz idealnym kandydatem na świetnego programistę, gdyż sam potrafiłeś rozwiązać postawiony przed tobą szereg problemów oraz znalazłeś brakujące ci informacje w odpowiednich źródłach. Gratulacje!
Aby jednak uniknąć ewentualnych kłopotów ze zrozumieniem dalszej części kursu, doradzam powrót do opuszczonego fragmentu tekstu i przeczytanie chociaż tych urywków, które dostarczają wspomnianych nowych informacji z zakresu języka C++.

Podsumowanie

Dotarliśmy (wreszcie!) do końca tego rozdziału. Nabyłeś w nim bardzo dużo wiadomości na temat modelowania złożonych struktur danych w C++.

Zaczęliśmy od prezentacji tablic, czyli zestawów określonej liczby tych samych elementów, opatrzonych wspólną nazwą. Poznaliśmy sposoby ich deklaracji oraz użycia w programie, a także możliwe zastosowania.
Dalej zajęliśmy się definiowaniem nowych, własnych typów danych. Wśród nich były typy wyliczeniowe, dopuszczające jedynie kilka możliwych wartości, oraz agregaty w rodzaju struktur, zamykające kilka pojedynczych informacji w jedną całość. Zetknęliśmy się przy tym z wieloma przykładami ich zastosowania w programowaniu.
Wreszcie, na ukoronowanie tego i kilku poprzednich rozdziałów stworzyliśmy całkiem spory i całkiem skomplikowany program, będący w dodatku grą! Mieliśmy niepowtarzalną okazję na zastosowanie zdobytych ostatnimi czasy umiejętności w praktyce.

Kolejny rozdział przyniesie nam natomiast zupełnie nowe spojrzenie na programowanie w C++.

Pytania i zadania

Jako że mamy za już sobą sporo wyczerpującego kodowania, nie zadam zbyt wielu programów do samodzielnego napisania. Nie uciekniesz jednak od pytań sprawdzających wiedzę! :)

Pytania

1. Co to jest tablica? Jak deklarujemy tablice?
2. W jaki sposób używamy pętli `for` oraz tablic?
3. Jak C++ obsługuje tablice wielowymiarowe? Czym są one w istocie?
4. Czym są i do czego służą typy wyliczeniowe? Dlaczego są lepsze od zwykłych stałych?
5. Jak definiujemy typy strukturalne?
6. Jaką drogą można dostać się do pojedynczych pól struktury?

Ćwiczenia

1. Napisz program, który pozwoli użytkownikowi na wprowadzenie dowolnej ilości liczb (ilość tą będzie podawał na początku) i obliczenie ich średniej arytmetycznej. Podawane liczby przechowuj w 100-elementowej tablicy (wykorzystasz zeń tylko część).
   (**Trudne**) Możesz też zrobić tak, by program nie pytał o ilość liczb, lecz prosił o kolejne aż do wpisania innych znaków.
   (**Bardzo trudne**) Czy można jakoś zapobiec marnotrawstwu pamięci, związanemu z tak dużą, lecz używaną tylko częściowo tablicą? Jak?
2. Stwórz aplikację, która będzie pokazywała liczbę dni do końca bieżącego roku. Wykorzystaj w niej strukturę `tm` i funkcję `localtime()` w taki sam sposób, jak w przykładzie `Biorhytm`.
3. (**Trudne**) W naszej grze w kółko i krzyżyk jest ukryta pewna usterka. Objawia się wtedy, gdy gracz wpisze coś innego niż liczbę jako numer pola. Spróbuj naprawić ten błąd; niech program reaguje tak samo, jak na wartość spoza przedziału <1; 9>.
   Wskazówka: zadanie jest podobne do trudniejszego wariantu ćwiczenia 1.
4. (**Bardzo trudne**) Ulepsz napisaną grę. Niech rozmiar planszy nie będzie zawsze wynosił 3×3, lecz mógł być zdefiniowany jako stała w pliku *game.h*.
   Wskazówka: ponieważ plansza pozostanie kwadratem, warunkiem zwycięstwa będzie nadal ułożenie linii poziomej, pionowej lub ukośnej z własnych symboli. Modyfikacji musi jednak ulec algorytm sprawdzania planszy (ten straszny :D) oraz sposób numerowania i rysowania pól.