1

KRÓTKO O PROGRAMOWANIU

Programy nie spadają z nieba,
najpierw tym niebem potrząść trzeba.
gemGreg

Rozpoczynamy zatem nasz kurs programowania gier. Zanim jednak napiszesz swojego własnego Quake'a, Warcrafta czy też inny wielki przebój, musisz nauczyć się tworzenia programów (gry to przecież też programy, prawda?) – czyli programowania.

Jeszcze niedawno czynność ta była traktowana na poły mistycznie: oto bowiem programista (czytaj jajogłowy) wpisuje jakieś dziwne ciągi liter i numerków, a potem w niemal magiczny sposób zamienia je w edytor tekstu, kalkulator czy wreszcie grę.
Obecnie obraz ten nie przystaje już tak bardzo do rzeczywistości, a tworzenie programów jest prostsze niż jeszcze kilkanaście lat temu. Nadal jednak wiele zależy od umiejętności samego kodera oraz jego doświadczenia, a zyskiwanie tychże jest kwestią długiej pracy i realizacji wielu projektów.
Nagrodą za ten wysiłek jest możliwość urzeczywistnienia dowolnego praktycznie pomysłu i wielka satysfakcja.

Czas więc przyjrzeć się, jak powstają programy.

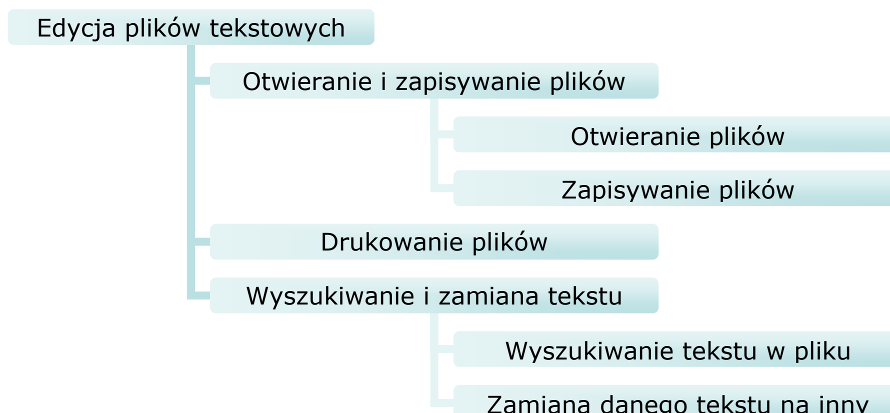
Krok za krokiem

Większość aplikacji została stworzona do realizacji jednego, konkretnego, choć obszernego zadania. Przykładowo, Notatnik potrafi edytować pliki tekstowe, Winamp – odtwarzać muzykę, a Paint tworzyć rysunki.

Screen 6. Głównym zadaniem Winampa jest odtwarzanie plików muzycznych

Możemy więc powiedzieć, że **główną funkcją** każdego z tych programów będzie odpowiednio edycja plików tekstowych, odtwarzanie muzyki czy tworzenie rysunków. Funkcję tę można jednak podzielić na mniejsze, bardziej szczegółowe. I tak Notatnik potrafi otwierać i zapisywać pliki, drukować je i wyszukiwać w nich tekst. Winamp zaś pozwala nie tylko odtwarzać utwory, ale też układać z nich playlisty.

Idąc dalej, możemy dotrzeć do następnych, coraz bardziej szczegółowych funkcji danego programu. Przypominają one więc coś w rodzaju drzewka, które pozwala nam niejako „rozłożyć daną aplikację na części”.



Schemat 1. Podział programu Notatnik na funkcje składowe

Zastanawiasz się pewnie, na jak drobne części możemy w ten sposób dzielić programy. Innymi słowy, czy dojdziemy wreszcie do takiego elementu, który nie da się rozdzielić na mniejsze. Spieszę z odpowiedzią, iż oczywiście tak – w przypadku Notatnika byliśmy zresztą bardzo blisko.

Czynność zatytułowana *Otwieranie plików* wydaje się być już jasno określona. Kiedy wybieramy z menu Plik programu pozycję Otwórz, Notatnik robi kilka rzeczy: najpierw pokazuje nam okno wyboru pliku. Gdy już zdecydujemy się na jakiś, pyta nas, czy chcemy zachować zmiany w już otwartym dokumencie (jeżeli jakiekolwiek zmiany rzeczywiście poczyniliśmy). W przypadku, gdy je zapiszemy w innym pliku lub odrzucimy, program przystąpi do odczytania zawartości żądanego przez nas dokumentu i wyświetli go na ekranie. Proste, prawda? :)
Przedstawiona powyżej charakterystyka czynności otwierania pliku posiada kilka znaczących cech:

- określa dokładnie kolejne kroki wykonywane przez program
- wskazuje różne możliwe warianty sytuacji i dla każdego z nich przewiduje odpowiednią reakcję

Pozwalają one nazwać niniejszy opis **algorytmem**.

Algorytm to jednoznacznie określony sposób, w jaki program komputerowy realizuje jakąś elementarną czynność.[5]

Jest to bardzo ważne pojęcie. Myśl o algorytmie jako o czymś w rodzaju przepisu albo instrukcji, która „mówi” aplikacji, co ma zrobić gdy napotka taką czy inną sytuację. Dzięki swoim algorytmom programy wiedzą co zrobić po naciśnięciu przycisku myszki, jak zapisać, otworzyć czy wydrukować plik, jak wyświetlić poprawnie stronę WWW, jak odtworzyć utwór w formacie MP3, jak rozpakować archiwum ZIP i ogólnie – jak wykonywać zadania, do których zostały stworzone.

Jeśli nie podoba ci się, iż cały czas mówimy o programach użytkowych zamiast o grach, to wiedz, że gry także działają w oparciu o algorytmy. Najczęściej są one nawet znacznie

[5] Nie jest to ścisła matematyczna definicja algorytmu, ale na potrzeby programistyczne nadaje się bardzo dobrze :)

bardziej skomplikowane od tych występujących w używanych na co dzień aplikacjach. Czyż nie łatwiej narysować prostą tabelkę z liczbami niż skomplikowaną scenę trójwymiarową? :)

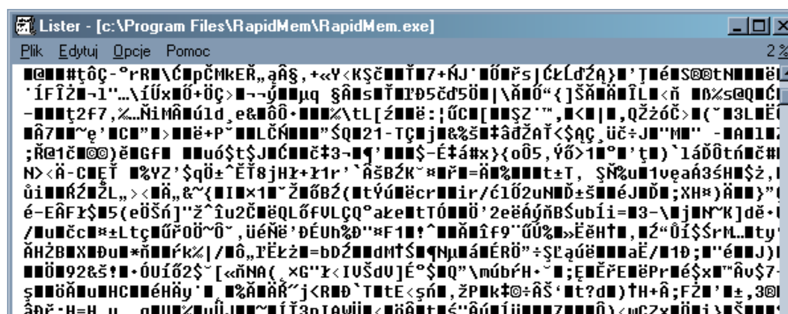
Z tego właśnie powodu wymyślanie algorytmów jest ważną częścią pracy twórcy programów, czyli programisty. Właśnie tą drogą koder określa sposób działania („zachowanie”) pisanego programu.

Podsumujmy: w każdej aplikacji możemy wskazać wykonywane przez nią czynności, które z kolei składają się z mniejszych etapów, a te jeszcze z mniejszych itd. Zadania te realizowane są poprzez algorytmy, czyli przepisy określone przez programistów – twórców programów.

Jak rozmawiamy z komputerem?

Wiemy już, że programy działają dzięki temu, że programiści konstruują dla nich odpowiednie algorytmy. Poznaliśmy nawet prosty algorytm, który być może jest stosowany jest przez program Notatnik do otwierania plików tekstowych.

Zauważ jednak, że jest on napisany w języku naturalnym – to znaczy takim, jakim posługują się ludzie. Chociaż jest doskonale zrozumiały dla wszystkich, to ma jedną niezaprzeczalną wadę: nie rozumie go komputer! Dla bezmyślnej maszyny jest on po prostu zbyt niejednoznaczny i niejasny.
Z drugiej strony, już istniejące programy są przecież doskonale „zrozumiałe” dla komputera i działają bez żadnych problemów. Jak to możliwe? Otóż pecet też posługuje się pewnego rodzaju językiem. Chcąc zobaczyć próbkę jego talentu lingwistycznego, wystarczy podejrzeć zawartość dowolnego pliku EXE. Co zobaczymy? Ciąg bezsensownych, chyba nawet losowych liter, liczb i innych znaków. On ma jednak sens, tyle że nam bardzo trudno poznać go w tej postaci. Po prostu język komputera jest dla równowagi zupełnie niezrozumiały dla nas, ludzi :)



Screen 7. Tak wygląda plik EXE :-)

Jak poradzić sobie z tym, zdawałoby się nierozwiązalnym, problemem? Jak radzą sobie wszyscy twórcy oprogramowania, skoro budując swoje programy muszą przecież „rozmawiać” z komputerem?
Ponieważ nie możemy peceta nauczyć naszego własnego języka i jednocześnie sami nie potrafimy porozumieć się z nim w jego „mowie”, musimy zastosować rozwiązanie kompromisowe. Na początek uściślimy więc i przejrzyście zorganizujemy nasz opis algorytmów. W przypadku otwierania plików w Notatniku może to wyglądać na przykład tak:

```
Algorytm Plik -> Otwórz
   Pokaż okno wyboru plików
   Jeżeli użytkownik kliknął Anuluj, To Przerwij
```

```
    Jeżeli poczyniono zmiany w aktualnym dokumencie, To
          Wyświetl komunikat "Czy zachować zmiany w aktualnym
dokumencie?" z przyciskami Tak, Nie, Anuluj
          Sprawdź decyzję użytkownika
                Decyzja Tak: wywołaj polecenie Plik -> Zapisz
                Decyzja Anuluj: Przerwij

    Odczytaj wybrany plik
    Wyświetl zawartość pliku
Koniec Algorytmu
```

Jak widać, sprecyzowaliśmy tu kolejne kroki wykonywane przez program – tak aby „wiedział", co należy po kolei zrobić. Fragmenty zaczynające się od `Jeżeli` i `Sprawdź` pozwalają odpowiednio reagować na różne sytuacje, takie jak zmiana decyzji użytkownika i wciśnięcie przycisku Anuluj.
Czy to wystarczy, by komputer wykonał to, co mu każemy? Otóż nie bardzo… Chociaż wprowadziliśmy już nieco porządku, nadal używamy języka naturalnego – jedynie struktura zapisu jest bardziej ścisła. Notacja taka, zwana **pseudokodem**, przydaje się jednak bardzo do przedstawiania algorytmów w czytelnej postaci. Jest znacznie bardziej przejrzysta oraz wygodniejsza niż opis w formie zwykłych zdań, które musiałyby być najczęściej wielokrotnie złożone i niezbyt poprawne gramatycznie. Dlatego też, kiedy będziesz wymyślał własne algorytmy, staraj się używać pseudokodu do zapisywania ich ogólnego działania.

No dobrze, wygląda to całkiem nieźle, jednak nadal nie potrafimy się porozumieć z tym mało inteligentnym stworem, jakim jest nasz komputer. Wszystko dlatego, iż nie wie on, w jaki sposób przetworzyć nasz algorytm, napisany w powstałym *ad hoc* języku, do postaci zrozumiałych dla niego „krzaczków", które widziałeś wcześniej.
Dla rozwiązania tego problemu stworzono sztuczne języki o dokładnie określonej składni i znaczeniu, które dzięki odpowiednim narzędziom mogą być zamieniane na kod binarny, czyli formę zrozumiałą dla komputera. Nazywamy je **językami programowania** i to właśnie one służą do tworzenia programów komputerowych. Wiesz już zatem, czego najpierw musisz się nauczyć :)

Język programowania to forma zapisu instrukcji dla komputera i programów komputerowych, pośrednia między językiem naturalnym a kodem maszynowym.

Program zapisany w języku programowania jest, podobnie jak nasz algorytm w pseudokodzie, zwykłym tekstem. Podobieństwo tkwi również w fakcie, że sam taki tekst nie wystarczy, aby napisaną aplikację uruchomić – najpierw należy ją zamienić w plik wykonywalny (w systemie Windows są to pliki z rozszerzeniem EXE). Czynność ta jest dokonywana w dwóch etapach.
Podczas pierwszego, zwanego **kompilacją**, program nazywany **kompilatorem** zamienia instrukcje języka programowania (czyli kod źródłowy, który, jak już mówiliśmy, jest po prostu tekstem) w kod maszynowy (binarny). Zazwyczaj na każdy plik z kodem źródłowym (zwany **modułem**) przypada jeden plik z kodem maszynowym.

Kompilator – program zamieniający kod źródłowy, napisany w jednym z języków programowania, na kod maszynowy w postaci oddzielnych modułów.

Drugi etap to **linkowanie** (zwane też konsolidacją lub po prostu łączeniem). Jest to budowanie gotowego pliku EXE ze skompilowanych wcześniej modułów. Oprócz nich mogą tam zostać włączone także inne dane, np. ikony czy kursory. Czyni to program zwany **linkerem**.

Linker łączy skompilowane moduły kodu i inne pliki w jeden plik wykonywalny, czyli program (w przypadku Windows – plik EXE).

Tak oto zdjęliśmy nimb magii z procesu tworzenia programu ;D

Skoro kompilacja i linkowanie są przeprowadzane automatycznie, a programista musi jedynie wydać polecenie rozpoczęcia tego procesu, to dlaczego nie pójść dalej – niech komputer na bieżąco tłumaczy sobie program na swój kod maszynowy. Rzeczywiście, jest to możliwe – powstało nawet kilka języków programowania działających w ten sposób (tak zwanych języków interpretowanych, przykładem jest choćby PHP, służący do tworzenia stron internetowych). Jednakże ogromna większość programów jest nadal tworzona w „tradycyjny" sposób.
Dlaczego? Cóż – jeżeli w programowaniu nie wiadomo, o co chodzi, to na pewno chodzi o wydajność[6] ;)) Kompilacja i linkowanie trwa po prostu długo, od kilkudziesięciu sekund w przypadku niewielkich programów, do nawet kilkudziesięciu minut przy dużych. Lepiej zrobić to raz i używać szybkiej, gotowej aplikacji niż nie robić w ogóle i czekać dwie minuty na rozwinięcie menu podręcznego :DD

Zatem czas na konkluzję i usystematyzowanie zdobytej wiedzy. Programy piszemy w językach programowania, które są niejako formą komunikacji z komputerem i wydawania mu poleceń. Są one następnie poddawane procesom kompilacji i konsolidacji, które zamieniają zapis tekstowy w binarny kod maszynowy. W wyniku tych czynności powstaje gotowy plik wykonywalny, który pozwala uruchomić program.

Języki programowania

Przegląd najważniejszych języków programowania

Obecnie istnieje bardzo, bardzo wiele języków programowania. Niektóre przeznaczono do konkretnych zastosowań, na przykład sieci neuronowych, inne zaś są narzędziami ogólnego przeznaczenia. Zazwyczaj większe korzyści zajmuje znajomość tych drugich, dlatego nimi właśnie się zajmiemy.
Od razu muszę zaznaczyć, że mimo to nie ma czegoś takiego jak język, który będzie dobry do wszystkiego. Spośród języków „ogólnych" niektóre są nastawione na szybkość, inne na rozmiar kodu, jeszcze inne na przejrzystość itp. Jednym słowem, panuje totalny rozgardiasz ;)

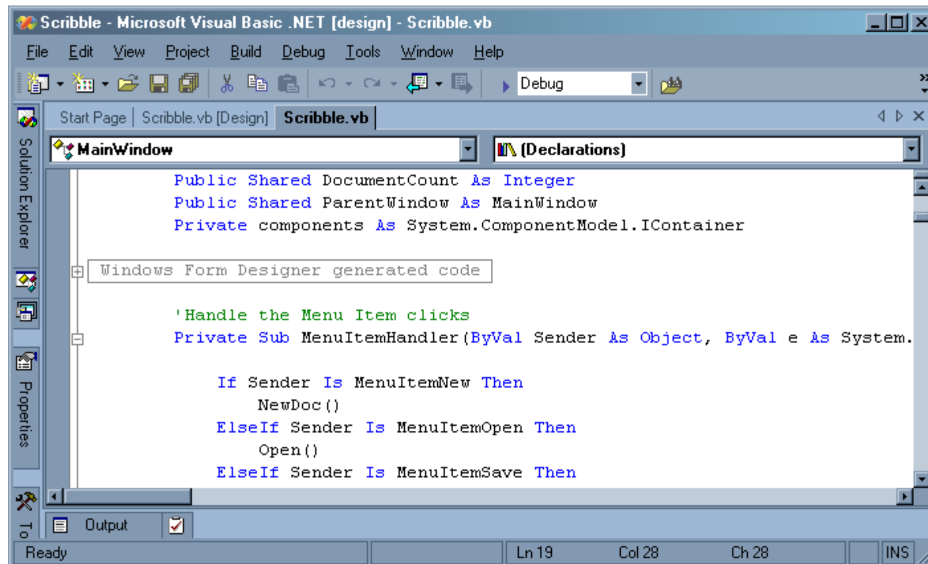
Należy koniecznie odróżniać języki programowania od innych języków używanych w informatyce. Na przykład HTML jest językiem opisu, gdyż za jego pomocą definiujemy jedynie wygląd stron WWW (wszelkie interaktywne akcje to już domena JavaScriptu). Inny rodzaj to języki zapytań w rodzaju SQL, służące do pobierania danych z różnych źródeł (na przykład baz danych).
Niepoprawne jest więc (popularne skądinąd) stwierdzenie „programować w HTML".

Przyjrzyjmy się więc najważniejszym używanym obecnie językom programowania:

1. **Visual Basic**
   Jest to następca popularnego swego czasu języka BASIC. Zgodnie z nazwą (*basic* znaczy prosty), był on przede wszystkim łatwy do nauki. Visual Basic pozwala na tworzenie programów dla środowiska Windows w sposób wizualny, tzn. poprzez konstruowanie okien z takich elementów jak przyciski czy pola tekstowe.
   Język ten posiada dosyć spore możliwości, jednak ma również jedną, za to bardzo

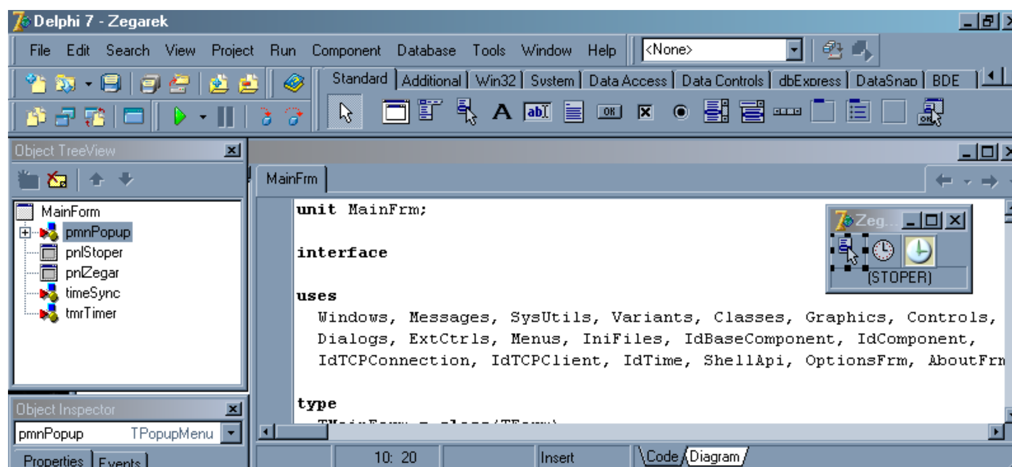
[6] Niektórzy powiedzą, że o niezawodność, ale to już kwestia osobistych priorytetów :)

poważną wadę. Programy w nim napisane nie są kompilowane w całości do kodu maszynowego, ale **interpretowane** podczas działania. Z tego powodu są znacznie wolniejsze od tych kompilowanych całkowicie.
Obecnie Visual Basic jest jednym z języków, który umożliwia tworzenie aplikacji pod lansowaną przez Microsoft platformę .NET, więc pewnie jeszcze o nim usłyszymy :)



Screen 8. Kod przykładowego projektu w Visual Basicu

2. **Object Pascal (Delphi)**
   Delphi z kolei wywodzi się od popularnego języka Pascal. Podobnie jak VB jest łatwy do nauczenia, jednakże oferuje znacznie większe możliwości zarówno jako język programowania, jak i narzędzie do tworzenia aplikacji. Jest całkowicie kompilowany, więc działa tak szybko, jak to tylko możliwe. Posiada również możliwość wizualnego konstruowania okien. Dzięki temu jest to obecnie chyba najlepsze środowisko do budowania programów użytkowych.

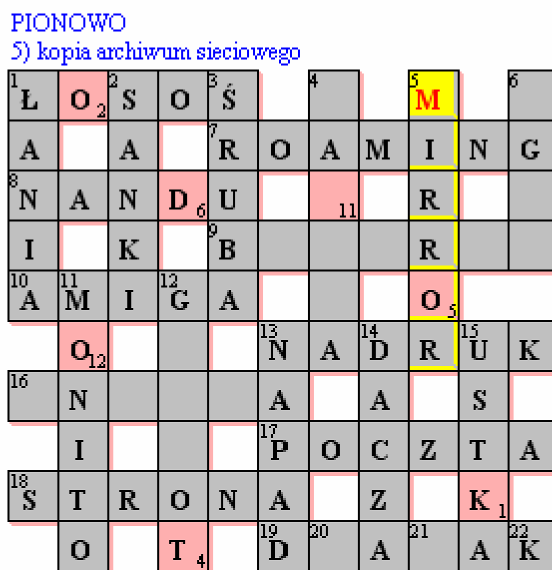


Screen 9.Tworzenie aplikacji w Delphi

3. **C++**
   C++ jest teraz chyba najpopularniejszym językiem do zastosowań wszelakich. Powstało do niego bardzo wiele kompilatorów pod różne systemy operacyjne i dlatego jest uważany za najbardziej przenośny. Istnieje jednak druga strona medalu – mnogość tych narzędzi prowadzi do niewielkiego rozgardiaszu i pewnych

trudności w wyborze któregoś z nich. Na szczęście sam język został w 1997 roku ostatecznie ustandaryzowany.
O C++ nie mówi się zwykle, że jest łatwy – być może ze względu na dosyć skondensowaną składnię (na przykład odpowiednikiem pascalowych słów `begin` i `end` są po prostu nawiasy klamrowe `{` i `}`). To jednak dosyć powierzchowne przekonanie, a sam język jest spójny i logiczny.
Jeżeli chodzi o możliwości, to w przypadku C++ są one bardzo duże – w sumie można powiedzieć, że nieco większe niż Delphi. Jest on też chyba najbardziej elastyczny – niejako dopasowuje się do preferencji programisty.

4. **Java**
Ostatnimi czasy Java stała się niemal częścią kultury masowej – wystarczy choćby wspomnieć o telefonach komórkowych i przeznaczonych doń aplikacjach. Ilustruje to dobrze główny cel Javy, a mianowicie przenośność – i to nie kodu, lecz skompilowanych programów! Osiągnięto to poprzez kompilację do tzw. *bytecode*, który jest wykonywany w ramach specjalnej maszyny wirtualnej. W ten sposób, program w Javie może być uruchamiany na każdej platformie, do której istnieje maszyna wirtualna Javy – a istnieje prawie na wszystkich, od Windowsa przez Linux, OS/2, QNX, BeOS, palmtopy czy wreszcie nawet telefony komórkowe. Z tego właśnie powodu Java jest wykorzystywana do pisania niewielkich programów umieszczanych na stronach WWW, tak zwanych **apletów**.
Ceną za tą przenośność jest rzecz jasna szybkość – *bytecode* Javy działa znacznie wolniej niż zwykły kod maszynowy, w dodatku jest strasznie pamięciożerny. Ponieważ jednak zastosowaniem tego języka nie są wielkie i wymagające aplikacje, lecz proste programy, nie jest to aż tak wielki mankament.
Składniowo Java bardzo przypomina C++.



Screen 10. Krzyżówka w formie apletu Javy

5. **PHP**
PHP (skrót od *Hypertext Preprocessor*) jest językiem używanym przede wszystkim w zastosowaniach internetowych, dokładniej na stronach WWW. Pozwala dodać im znacznie większą funkcjonalność niż ta oferowana przez zwykły HTML. Obecnie miliony serwisów wykorzystuje PHP – dużą rolę w tym sukcesie ma zapewne jego licencja, oparta na zasadach Open Source (czyli brak ograniczeń w rozprowadzaniu i modyfikacji).
Możliwości PHP są całkiem duże, nie można tego jednak powiedzieć o szybkości – jest to język interpretowany. Jednakże w przypadku głównego zastosowania PHP, czyli obsłudze serwisów internetowych, nie ma ona większego znaczenia – czas

wczytywania strony WWW to przecież w większości czas przesyłania gotowego kodu HTML od serwera do odbiorcy.
Jeżeli chodzi o składnię, to trochę przypomina ona C++. Kod PHP można jednak swobodnie przeplatać znacznikami HTML.
Z punktu widzenia programisty gier język ten jest w zasadzie zupełnie bezużyteczny (chyba że kiedyś sam będziesz wykonywał oficjalną stronę internetową swojej wielkiej produkcji ;D), wspominam o nim jednak ze względu na bardzo szerokie grono użytkowników, co czyni go jednym z ważniejszych języków programowania.

Screen 11. Popularny skrypt forów dyskusyjnych, phpBB, także działa w oparciu o PHP

To oczywiście nie wszystkie języki – jak już pisałem, jest ich całe mnóstwo. Jednakże w ogromnej większości przypadków główną różnicą między nimi jest składnia, a więc sprawa mało istotna (szczególnie, jeżeli dysponuje się dobrą dokumentacją :D). Z tego powodu poznanie jednego z nich bardzo ułatwia naukę następnych – po prostu im więcej języków już znasz, tym łatwiej uczysz się następnych :)

Brzemienna w skutkach decyzja

Musimy zatem zdecydować, którego języka będziemy się uczyć, aby zrealizować nasz nadrzędny cel, czyli poznanie tajników programowania gier. Sprecyzujmy więc wymagania wobec owego języka:

- programy w nim napisane muszą być szybkie – w takim wypadku możemy wziąć pod uwagę jedynie języki całkowicie **kompilowane** do kodu maszynowego
- musi dobrze współpracować z różnorodnymi bibliotekami graficznymi, na przykład DirectX
- powinien posiadać duże możliwości i zapewniać gotowe, często używane rozwiązania
- nie zaszkodzi też, gdy będzie w miarę prosty i przejrzysty :)

Jeżeli uwzględnimy wszystkie te warunki, to spośród całej mnogości języków programowania (w tym kilku przedstawionych wcześniej) zostają nam aż… dwa – Delphi oraz C++.

Przyglądając się bliżej Delphi, możemy zauważyć, iż jest on przeznaczony przede wszystkim do programowania aplikacji użytkowych, które pozostają przecież poza kręgiem naszego obecnego zainteresowania :) Na plus można jednak zaliczyć prostotę i przejrzystość języka oraz jego bardzo dużą wydajność. Również możliwości Delphi są całkiem spore.
Z kolei C++ zdaje się być bardziej uniwersalny. Dobrze rozumie się z ważnymi dla nas bibliotekami graficznymi, jest także bardzo szybki i posiada duże możliwości. Składnia z kolei jest raczej „ekonomiczna” i być może nieco bardziej skomplikowana.

Czyżbyśmy mieli zatem remis, a prawda leżała (jak zwykle) pośrodku? :) Otóż niezupełnie – nie uwzględniliśmy bowiem ważnego czynnika, jakim jest **popularność** danego języka. Jeżeli jest on szeroko znany i używany (do programowania gier), to z pewnością istnieje o nim więcej przydatnych źródeł informacji, z których mógłbyś korzystać.
Z tego właśnie powodu Delphi jest gorszym wyborem, ponieważ ogromna większość dokumentacji, artykułów, kursów itp. dotyczy języka C++. Wystarczy chociażby wspomnieć, iż Microsoft nie dostarcza narzędzi pozwalających na wykorzystanie DirectX w Delphi – są one tworzone przez niezależne zespoły[7] i ich używanie wymaga pewnego doświadczenia.

A więc – C++! Język ten wydaje się najlepszym wyborem, jeżeli chodzi o programowanie gier komputerowych. A skoro mamy już tą ważną decyzję za sobą, została nam jeszcze tylko pewna drobnostka – trzeba się tego języka nauczyć :))

Kwestia kompilatora

Jak już wspominałem kilkakrotnie, C++ jest bardzo przenośnym językiem, umożliwiającym tworzenie aplikacji na różnych platformach sprzętowych i programowych. Z tegoż powodu istnieje do niego całe mnóstwo kompilatorów.
Ale kompilator to tylko program do zamiany kodu C++ na kod maszynowy – w dodatku działa on zwykle w trybie wiersza poleceń, a więc nie jest zbyt wygodny w użyciu. Dlatego równie ważne jest **środowisko programistyczne**, które umożliwiałoby wygodne pisanie kodu, zarządzanie całymi projektami i ułatwiałoby kompilację.

Środowisko programistyczne (ang. *integrated development environment* – w skrócie IDE) to pakiet aplikacji ułatwiających tworzenie programów w danym języku programowania. Umożliwia najczęściej organizowanie plików z kodem w projekty, łatwą kompilację, czasem też wizualne tworzenie okien dialogowych.
Popularnie, środowisko programistyczne nazywa się po prostu kompilatorem (gdyż jest jego główną częścią).

Przykłady takich środowisk zaprezentowałem na screenach przy okazji przeglądu języków programowania. Nietrudno się domyśleć, iż dla C++ również przewidziano takie narzędzia. W przypadku środowiska Windows, które rzecz jasna interesuje nas najbardziej, mamy ich kilka:

1. **Bloodshed Dev-C++**
   Pakiet ten ma niewątpliwą zaletę – jest darmowy do wszelakich zastosowań, także komercyjnych. Niestety zdaje się, że na tym jego zalety się kończą :) Posiada wprawdzie całkiem wygodne IDE, ale nie może się równać z profesjonalnymi narzędziami: nie posiada na przykład możliwości edycji zasobów (ikon, kursorów itd.)
   Można go znaleźć na stronie producenta.
2. **Borland C++Builder**
   Z wyglądu bardzo przypomina Delphi – oczywiście poza zastosowanym językiem programowania, którym jest C++. Niemniej, tak samo jak swój kuzyn jest on przeznaczony głównie do tworzenia aplikacji użytkowych, więc nie odpowiadałby nam zbytnio :)
3. **Microsoft Visual C++**
   Ponieważ jest to produkt firmy Microsoft, znakomicie integruje się z innym produktem tej firmy, czyli DirectX – wobec czego dla nas, (przyszłych)

[7] Najbardziej znanym jest JEDI

programistów gier, wypada bardzo korzystnie. Nic dziwnego zatem, że używają go nawet profesjonalni twórcy.

Tak jest, dobrze myślisz – zalecam Visual C++ :) Warto naśladować najlepszych, a skoro ogromna większość komercyjnych gier powstaje przy użyciu tego narzędzia (i to nie tylko w połączeniu z DirectX), musi to chyba znaczyć, że faktycznie jest dobre[8].

Jeżeli upierasz się przy innym środowisku, to pamiętaj, że przedstawione przeze mnie opisy niektórych poleceń i opcji mogą nie odpowiadać twojemu IDE. W większości nie dotyczy to jednak samego języka C++, którego składnię i możliwości zachowują wszystkie kompilatory. W razie jakichkolwiek kłopotów możesz zawsze odwołać się do dokumentacji :)

Podsumowanie

Uff, to już koniec tego rozdziału :) Zaczęliśmy go od dokładnego zlustrowania Notatnika i podzieleniu go na drobne części – aż doszliśmy do algorytmów. Dowiedzieliśmy się, iż to głównie one składają się na gotowy program i że zadaniem programisty jest właśnie wymyślanie algorytmów.

Następnie rozwiązaliśmy problem wzajemnego niezrozumienia człowieka i komputera, dzięki czemu w przyszłości będziemy mogli tworzyć własne programy. Poznaliśmy służące do tego narzędzia, czyli języki programowania.

Wreszcie, podjęliśmy (OK, ja podjąłem :D) ważne decyzje, które wytyczają nam kierunek dalszej nauki – a więc wybór języka C++ oraz środowiska Visual C++.

Następny rozdział jest wcale nie mniej znaczący, a może nawet ważniejszy. Napiszesz bowiem swój pierwszy program :)

Pytania i zadania

Cóż, prace domowe są nieuniknione :) Odpowiedzenie na poniższe pytania i wykonanie ćwiczeń pozwoli ci lepiej zrozumieć i zapamiętać informacje z tego rozdziału.

Pytania

1. Dlaczego języki interpretowane są wolniejsze od kompilowanych?

Ćwiczenia

1. Wybierz dowolny program i spróbuj nazwać jego główną funkcję. Postaraj się też wyróżnić te bardziej szczegółowe.
2. Zapisz w postaci pseudokodu algorytm… parzenia herbaty :D Pamiętaj o uwzględnieniu takich sytuacji jak: pełny/pusty czajnik, brak zapałek lub zapalniczki czy brak herbaty

---

[8] Wiem, że dla niektórych pojęcia „dobry produkt” i „Microsoft” wzajemnie się wykluczają, ale akurat w tym przypadku wcale tak nie jest :)