# 2

# Z CZEGO SKŁADA SIĘ PROGRAM?

*Każdy działający program jest przestarzały.*
pierwsze prawo Murphy'ego o oprogramowaniu

Gdy mamy już przyswojoną niezbędną dawkę teoretycznej i pseudopraktycznej wiedzy, możemy przejść od słów do czynów :)

W aktualnym rozdziale zapoznamy się z podstawami języka C++, które pozwolą nam opanować umiejętność tworzenia aplikacji w tym języku. Napiszemy więc swój pierwszy program (drugi, trzeci i czwarty zresztą też :D), zaznajomimy się z podstawowymi pojęciami używanymi w programowaniu i zdobędziemy garść niezbędnej wiedzy :)

## C++, pierwsze starcie

Zanim w ogóle zaczniemy programować, musisz zaopatrzyć się w odpowiedni kompilator C++ - wspominałem o tym pod koniec poprzedniego rozdziału, zalecając jednocześnie używanie Visual C++. Dlatego też opisy poleceń IDE czy screeny będą dotyczyły właśnie tego narzędzia. Nie uniemożliwia to oczywiście używania innego środowiska, lecz w takim wypadku będziesz w większym stopniu zdany na siebie. Ale przecież lubisz wyzwania, prawda? ;)

### *Bliskie spotkanie z kompilatorem*

Pierwsze wyzwanie, jakim jest instalacja środowiska, powinieneś mieć już za sobą, więc pozwolę sobie optymistycznie założyć, iż faktycznie tak jest :) Swoją drogą, instalowanie programów jest częścią niezbędnego zestawu umiejętności, które trzeba posiąść, by mienić się (średnio)zaawansowanym użytkownikiem komputera. Zaś kandydat na przyszłego programistę powinien niewątpliwie posiadać pewien (w domyśle – raczej większy niż mniejszy) poziom obeznania w kwestii obsługi peceta. W ostateczności można jednak sięgnąć do odpowiednich pomocy naukowych :D

Środowisko programistyczne będzie twoim podstawowym narzędziem pracy, więc musisz je dobrze poznać. Nie jest ono zbyt skomplikowane w obsłudze – z pewnością nie zawiera takiego natłoku nikomu niepotrzebnych funkcji jak chociażby popularne pakiety biurowe :) Niemniej, z pewnością przyda ci kilka słów wprowadzenia.

W użyciu jest parę wersji Visual C++. W tym tutorialu będę opierał się na pakiecie Visual Studio 7 .NET (Visual C++ jest jego częścią), który różni się nieco od wcześniejszej, do niedawna bardzo popularnej, wersji 6. Dlatego też w miarę możliwości będę starał się wskazywać na najważniejsze różnice między tymi dwoma edycjami IDE.

„Goły” kompilator jest tylko maszynką zamieniającą kod C++ na kod maszynowy, działającą na zasadzie „ty mi podajesz plik ze swoim kodem, a ja go kompiluję”. Gdy uświadomimy sobie, że przeciętny program składa się z kilku(nastu) plików kodu źródłowego, z których każdy należałoby kompilować oddzielnie i wreszcie linkować je w jeden plik wykonywalny, docenimy zawarte w środowiskach programistycznych mechanizmy **zarządzania projektami**.

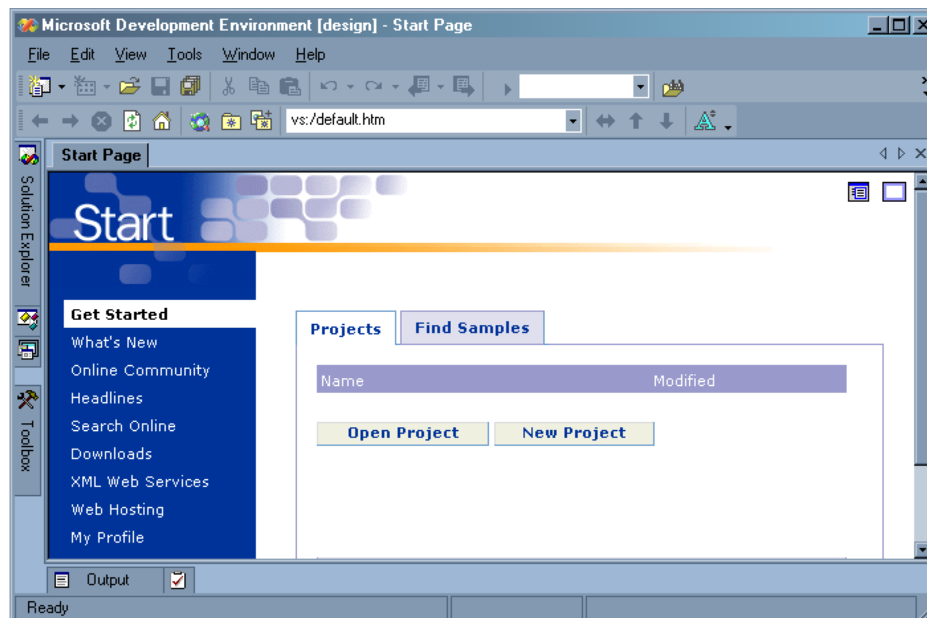
**Projekt** w środowiskach programistycznych to zbiór modułów kodu źródłowego i innych plików, które po kompilacji i linkowaniu stają się pojedynczym plikiem EXE, czyli programem.

Do najważniejszych zalet projektu należy bardzo łatwa kompilacja – wystarczy wydać jedno polecenie (na przykład wybrać opcję z menu), a projekt zostanie automatycznie skompilowany i zlinkowany. Zważywszy, iż tak nie tak dawno temu czynność ta wymagała wpisania kilkunastu długich poleceń lub napisania oddzielnego skryptu, widzimy tutaj duży postęp :)

Kilka projektów można pogrupować w tzw. rozwiązania[9] (ang. *solutions*). Przykładowo, jeżeli tworzysz grę, do której dołączysz edytor etapu, to zasadnicza gra oraz edytor będą oddzielnymi projektami, ale rozsądnie będzie zorganizować je w jedno rozwiązanie.

***

Teraz, gdy wiemy już sporo na temat sposobu działania naszego środowiska oraz przyczyn, dlaczego ułatwia nam ono życie, przydałoby się je uruchomić – uczyń więc to niezwłocznie. Powinieneś zobaczyć na przykład taki widok:



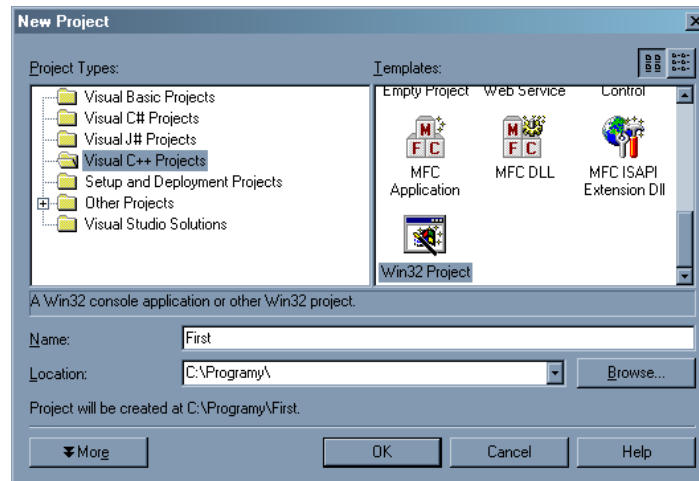

**Screen 12. Okno początkowe środowiska Visual Studio .NET**

Cóż, czas więc coś napisać - skoro mamy nauczyć się programowania, pisanie programów jest przecież koniecznością :D
Na podstawie tego, co wcześniej napisałem o projektach, nietrudno się domyśleć, iż rozpoczęcie pracy nad aplikacją oznacza właśnie stworzenie nowego projektu. Robi się to bardzo prosto: najbardziej elementarna metoda to po prostu kliknięcie w przycisk *New*

[9] W Visual C++ 6 były to obszary robocze (ang. *workspaces*)

*Project* widoczny na ekranie startowym; można również użyć polecenia *File|New|Project* z menu.

Twoim oczom ukaże się wtedy okno zatytułowane, a jakże, *New Project*[10] :) Możesz w nim wybrać typ projektu – my zdecydujemy się oczywiście na *Visual C++* oraz *Win32 Project*, czyli aplikację Windows.




**Screen 13. Opcje nowego projektu**

Nadaj swojemu projektowi jakąś dobrą nazwę (chociażby taką, jak na screenie), wybierz dla niego katalog i kliknij OK.

Najlepiej jeżeli utworzysz sobie specjalny folder na programy, które napiszesz podczas tego kursu. Pamiętaj, porządek jest bardzo ważny :)

Po krótkiej chwili ujrzysz następne okno – kreator :) Obsesja Microsoftu na ich punkcie jest powszechnie znana, więc nie bądź zdziwiony widząc kolejny przejaw ich radosnej twórczości ;) Tenże egzemplarz służy dokładnemu dopasowaniu parametrów projektu do osobistych życzeń. Najbardziej interesująca jest dla nas strona zatytułowana *Application Settings* – przełącz się zatem do niej.

# Rodzaje aplikacji

Skoncentrujemy się przede wszystkim na opcji *Application Type*, a z kilku dopuszczalnych wariantów weźmiemy pod lupę dwa:

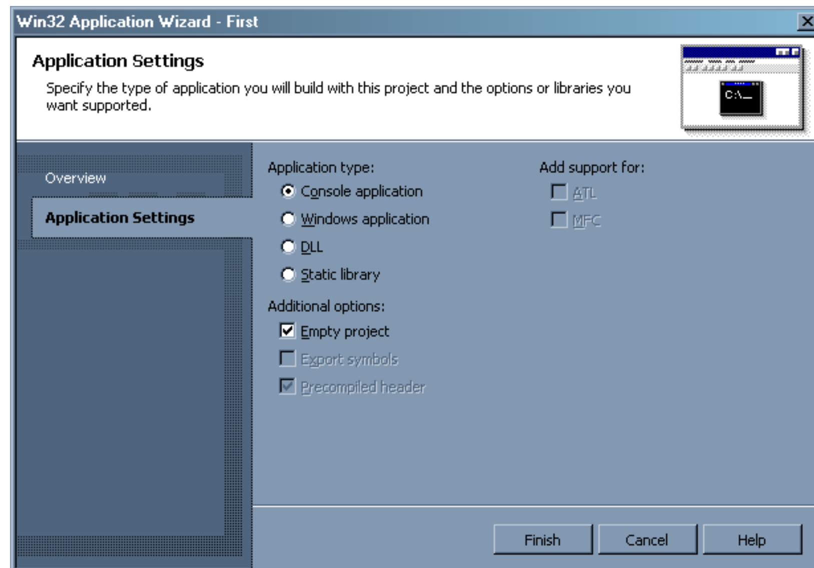
- *Windows application* to zgodnie z nazwą aplikacja okienkowa. Składa się z jednego lub kilku okien, zawierających przyciski, pola tekstowe, wyboru itp. – czyli wszystko to, z czym stykamy się w Windows nieustannie.
- *Console application* jest programem innego typu: do komunikacji z użytkownikiem używa tekstu wypisywanego w **konsoli** – stąd nazwa. Dzisiaj może wydawać się to archaizmem, jednak aplikacje konsolowe są szeroko wykorzystywane przez doświadczonych użytkowników systemów operacyjnych. Szczególnie dotyczy to tych z rodziny Unixa, ale w Windows także mogą być bardzo przydatne.

Programy konsolowe nie są tak efektowne jak ich okienkowi bracia, posiadają za to bardzo ważną dla początkującego programisty cechę – są proste :) Najprostsza aplikacja tego typu to kwestia kilku linijek kodu, podczas gdy program okienkowy wymaga ich

[10] Analogiczne okno w Visual C++ 6 wyglądało zupełnie inaczej, jednak ma podobne opcje

kilkudziesięciu. Idee działania takiego programu są również trochę bardziej skomplikowane.

Z tych właśnie powodów zajmiemy się na razie wyłącznie aplikacjami konsolowymi – pozwolą nam w miarę łatwo nauczyć się samego języka C++ (co jest przecież naszym aktualnym priorytetem), bez zagłębiania się w skomplikowane meandry programowania Windows.




**Screen 14. Ustawienia aplikacji**

Wybierz więc pozycję *Console application* na liście *Application type*. Dodatkowo zaznacz też opcję *Empty project* – spowoduje to utworzenie pustego projektu, a oto nam aktualnie chodzi.

# *Pierwszy program*

Gdy wreszcie ustalimy i zatwierdzimy wszystkie opcje projektu, możemy przystąpić do właściwej części tworzenia programu, czyli kodowania.
Aby dodać do naszego projektu pusty plik z kodem źródłowym, wybierz pozycję menu *Project|Add New Item*. W okienku, które się pojawi, w polu *Templates* zaznacz ikonę *C++ File (.cpp)*, a jako nazwę wpisz po prostu *main*. W ten sposób utworzysz plik *main.cpp*, który wypełnimy kodem naszego programu.

Plik ten zostanie od razu otwarty, więc możesz bez zwłoki wpisać doń taki oto kod:

```
// First - pierwszy program w C++

#include <iostream>
#include <conio.h>

void main()
{
   std::cout << "Hurra! Napisalem pierwszy program w C++!" << std::endl;
   getch();
}
```

Tak jest, to wszystko – te kilka linijek kodu składają się na cały nasz program. Pewnie niezbyt wielka to teraz pociecha, bo ów kod jest dla ciebie zapewne „trochę” niejasny, ale spokojnie – powoli wszystko sobie wyjaśnimy :)

Na razie wciśnij klawisz F7 (lub wybierz z menu *Build|Build Solution*), by skompilować i zlinkować aplikację. Jak widzisz, jest to proces całkowicie automatyczny i, jeżeli tylko kod jest poprawny, nie wymaga żadnych działań z twojej strony.
W końcu, wciśnij F5 (lub wybierz *Debug|Start*) i podziwiaj konsolę z wyświetlonym entuzjastycznym komunikatem :D (A gdy się nim nacieszysz, naciśnij dowolny klawisz, by zakończyć program.)

# Kod programu

Naturalnie, teraz przyjrzymy się bliżej naszemu elementarnemu projektowi, przy okazji odkrywając najważniejsze aspekty programowania w C++.

## *Komentarze*

Pierwsza linijka:

```
// First – pierwszy program w C++
```

to **komentarz**, czyli dowolny opis słowny. Jest on całkowicie ignorowany przez kompilator, natomiast może być pomocny dla piszącego i czytającego kod.
Komentarze piszemy w celu wyjaśnienia pewnych fragmentów kodu programu, oddzielenia jednej jego części od drugiej, oznaczania funkcji i modułów itp. Odpowiednia ilość komentarzy ułatwia zrozumienie kodu, więc stosuj je często :)

W C++ komentarze zaczynamy od `//` (dwóch slashy):

```
// To jest komentarz
```

lub umieszczamy je między `/*` i `*/`, na przykład:

```
/* Ten komentarz może być bardzo długi
   i składać się z kilku linijek.     */
```

W moim komentarzu do programu umieściłem jego tytuł[11] oraz krótki opis; będę tę zasadę stosował na początku każdego przykładowego kodu. Oczywiście, ty możesz komentować swoje źródła wedle własnych upodobań, do czego cię gorąco zachęcam :D

## *Funkcja* `main()`

Kiedy uruchamiamy nasz program, zaczyna on wykonywać kod zawarty w funkcji `main()`. Od niej więc rozpoczyna się działanie aplikacji – a nawet więcej: na niej też to działanie się kończy. Zatem program (konsolowy) to przede wszystkim kod zawarty w funkcji `main()` – determinuje on bezpośrednio jego zachowanie.

W przypadku rozważanej aplikacji funkcja ta nie jest zbyt obszerna, niemniej zawiera wszystkie niezbędne elementy.
Najważniejszym z nich jest **nagłówek**, który u nas prezentuje się następująco:

```
void main()
```

[11] Takim samym tytułem jest oznaczony gotowy program przykładowy dołączony do kursu

Występujące na początku słowo kluczowe `void` mówi kompilatorowi, że nasz program nie będzie informował systemu operacyjnego o wyniku swojego działania. Niektóre programy robią to poprzez zwracanie liczby – zazwyczaj zera w przypadku braku błędów i innych wartości, gdy wystąpiły jakieś problemy. Nam to jest jednak zupełnie niepotrzebne – w końcu nie wykonujemy żadnych złożonych operacji, zatem nie istnieje możliwość jakiekolwiek niepowodzenia[12].

Gdybyśmy jednak chcieli uczynić systemowi zadość, to powinniśmy zmienić nagłówek na `int main()` i na końcu funkcji dopisać `return 0;` - a więc poinformować system o sukcesie operacji. Jak jednak przekonaliśmy się wcześniej, nie jest to niezbędne.

Po nagłówku występuje nawias otwierający `{`. Jego główna rola to informowanie kompilatora, że „tutaj coś się zaczyna”. Wraz z nawiasem zamykającym `}` tworzy on **blok kodu** – na przykład funkcję. Z takimi parami nawiasów będziesz się stale spotykał; mają one znaczenie także dla programisty, gdyż porządkują kod i czynią go bardziej czytelnym.

Przyjęte jest, iż następne linijki po nawiasie otwierającym `{`, aż do zamykającego `}`, powinny być przesunięte w prawo za pomocą wcięć (uzyskiwanych spacjami lub klawiszem TAB). Poprawia to oczywiście przejrzystość kodu, lecz pamiętanie o tej zasadzie podczas pisania może być uciążliwe. Na szczęście dzisiejsze środowiska programistyczne są na tyle sprytne, że same dbają o właściwe umieszczanie owych wcięć. Nie musisz więc zawracać sobie głowy takimi błahostkami – grunt, żeby wiedzieć, komu należy być wdzięcznym ;))

Takim oto sposobem zapoznaliśmy się ze strukturą funkcji `main()`, będącej główną częścią programu konsolowego w C++. Teraz czas zająć się jej zawartością i dowiedzieć się, jak i dlaczego nasz program działa :)

## *Pisanie tekstu w konsoli*

Mieliśmy okazję się przekonać, że nasz program pokazuje nam komunikat podczas działania. Nietrudno dociec, iż odpowiada za to ta linijka:

```
std::cout << "Hurra! Napisalem pierwszy program w C++!" << std::endl;
```

`std::cout` oznacza tak zwany **strumień wyjścia**. Jego zadaniem jest wyświetlanie na ekranie konsoli wszystkiego, co doń wyślemy – a wysyłać możemy oczywiście tekst. Korzystanie z tego strumienia umożliwia zatem pokazywanie nam w oknie konsoli wszelkiego rodzaju komunikatów i innych informacji. Będziemy go używać bardzo często, dlatego musisz koniecznie zaznajomić się ze sposobem wysyłania doń tekstu.
A jest to wbrew pozorom bardzo proste, nawet intuicyjne. Spójrz tylko na omawianą linijkę – nasz komunikat jest otoczony czymś w rodzaju strzałek wskazujących `std::cout`, czyli właśnie strumień wyjścia. Wpisując je (za pomocą znaku mniejszości), robimy dokładnie to, o co nam chodzi: wysyłamy nasz tekst **do** strumienia wyjścia.
Drugim elementem, który tam trafia, jest `std::endl`. Oznacza on ni mniej, ni więcej, jak tylko koniec linijki i przejście do następnej. W przypadku, gdy wyświetlamy tylko jedną linię tekstu nie ma takiej potrzeby, ale przy większej ich liczbie jest to niezbędne.

Występujący przez nazwami `cout` i `endl` przedrostek `std::` oznacza tzw. **przestrzeń nazw**. Taka przestrzeń to nic innego, jak zbiór symboli, któremu nadajemy nazwę.

[12] Podobny optymizm jest zazwyczaj grubą przesadą i możemy sobie na niego pozwolić tylko w najprostszych programach, takich jak niniejszy :)

Niniejsze dwa należą do przestrzeni `std`, gdyż są częścią Biblioteki Standardowej języka C++ (wszystkie jej elementy należą do tej przestrzeni). Możemy uwolnić się od konieczności pisania przedrostka przestrzeni nazw `std`, jeżeli przed funkcją `main()` umieścimy deklarację `using namespace std;`. Wtedy moglibyśmy używać krótszych nazw `cout` i `endl`.

Konkludując: strumień wyjścia pozwala nam na wyświetlanie tekstu w konsoli, zaś używamy go poprzez `std::cout` oraz „strzałki” `<<`.

***

Druga linijka funkcji `main()` jest bardzo krótka:

```
getch();
```

Podobnie krótko powiem więc, że odpowiada ona za oczekiwanie programu na naciśnięcie dowolnego klawisza. Traktując rzecz ściślej, `getch()` jest funkcją podobnie jak `main()`, jednakże do związanego z tym faktem zagadnienia przejdziemy nieco później. Na razie zapamiętaj, iż jest to jeden ze sposobów na wstrzymanie działania programu do momentu wciśnięcia przez użytkownika dowolnego klawisza na klawiaturze.

## *Dołączanie plików nagłówkowych*

Pozostały nam jeszcze dwie pierwsze linijki programu:

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

które wcale nie są tak straszne, jak wyglądają na pierwszy rzut oka :)
Przede wszystkim zauważmy, że zaczynają się one od znaku `#`, czym niewątpliwie różnią się od innych instrukcji języka C++. Są to bowiem specjalne polecenia wykonywane jeszcze przed kompilacją - tak zwane **dyrektywy**. Przekazują one różne informacje i komendy, pozwalają więc sterować przebiegiem kompilacji programu.

Jedną z tych dyrektyw jest właśnie `#include`. Jak sama nazwa wskazuje, służy ona do **dołączania** – przy jej pomocy włączamy do naszego programu **pliki nagłówkowe** Ale czym one są i dlaczego ich potrzebujemy?
By się o tym przekonać, zapomnijmy na chwilę o programowaniu i wczujmy się w rolę zwykłego użytkownika komputera. Gdy instaluje on nową grę, zazwyczaj musi również zainstalować DirectX, jeżeli jeszcze go nie ma. To całkowicie naturalne, gdyż większość gier **korzysta** z tej biblioteki, więc wymaga jej do działania. Równie oczywisty jest także fakt, że do używania edytora tekstu czy przeglądarki WWW ów pakiet nie jest potrzebny – te programy po prostu z niego nie korzystają.
Nasz program nie korzysta ani z DirectX, ani nawet ze standardowych funkcji Windows (bo nie jest aplikacją okienkową). Wykorzystuje natomiast konsolę i dlatego potrzebuje odpowiednich mechanizmów do jej obsługi - zapewniają je właśnie pliki nagłówkowe.

Pliki nagłówkowe umożliwiają korzystanie z pewnych funkcji, technik, bibliotek itp. wszystkim programom, które dołączają je do swojego kodu źródłowego.

W naszym przypadku dyrektywa `#include` ma za zadanie włączenie do kodu plików *iostream* i *conio.h*. Pierwszy z nich pozwala nam pisać w oknie konsoli za pomocą `std::cout`, drugi zaś wywołać funkcję `getch()`, która czeka na dowolny klawisz.

Nie znaczy to jednak, że każdy plik nagłówkowy odpowiada tylko za jedną instrukcję. Jest wręcz odwrotnie, na przykład wszystkie funkcje systemu Windows (a jest ich kilka tysięcy) wymagają dołączenia tylko jednego pliku!

Konieczność dołączania plików nagłówkowych (zwanych w skrócie nagłówkami) może ci się wydawać celowym utrudnianiem życia programiście. Ma to jednak głęboki sens, gdyż zmniejsza rozmiary programów. Dlaczego kompilator miałby powiększać plik EXE zwykłej aplikacji konsolowej o nazwy (i nie tylko nazwy) wszystkich funkcji Windows czy DirectX, skoro i tak nie będzie ona z nich korzystać? Mechanizm plików nagłówkowych pozwala temu zapobiec i tą drogą korzystnie wpłynąć na objętość programów.

***

Tym zagadnieniem zakończyliśmy omawianie naszego programu - możemy sobie pogratulować :) Nie był on wprawdzie ani wielki, ani szczególnie imponujący, lecz początki zawsze są skromne. Nie spoczywajmy zatem na laurach i kontynuujmy…

# Procedury i funkcje

Pierwszy napisany przez nas program składał się wyłącznie z jednej funkcji `main()`. W praktyce takie sytuacje w ogóle się nie zdarzają, a kod aplikacji zawiera najczęściej bardzo wiele procedur i funkcji. Poznamy zatem dogłębnie istotę tych konstrukcji, by móc pisać prawdziwe programy.

Procedura lub funkcja to fragment kodu, który jest wpisywany raz, ale może być wykonywany wielokrotnie. Realizuje on najczęściej jakąś pojedynczą czynność przy użyciu ustalonego przez programistę algorytmu. Jak wiemy, działanie wielu algorytmów składa się na pracę całego programu, możemy więc powiedzieć, że procedury i funkcje są **podprogramami**, których cząstkowa praca przyczynia się do funkcjonowania programu jako całości.

Gdy mamy już (a mamy? :D) pełną jasność, czym są podprogramy i rozumiemy ich rolę, wyjaśnijmy sobie różnicę między procedurą a funkcją.

**Procedura** to wydzielony fragment kodu programu, którego zadaniem jest wykonywanie jakiejś czynności.

**Funkcja** zawiera kod, którego celem jest obliczenie i zwrócenie jakiejś wartości[13].

Procedura może przeprowadzać działania takie jak odczytywanie czy zapisywanie pliku, wypisywanie tekstu czy rysowanie na ekranie. Funkcja natomiast może policzyć ilość wszystkich znaków 'a' w danym pliku czy też wyznaczyć najmniejszą liczbę spośród wielu podanych.
W praktyce (i w języku C++) różnica między procedurą a funkcją jest dosyć subtelna, dlatego często obie te konstrukcje nazywa się dla uproszczenia funkcjami. A ponieważ lubimy wszelką prostotę, też będziemy tak czynić :)

## *Własne funkcje*

Na początek dokonamy prostej modyfikacji programu napisanego wcześniej. Jego kod będzie teraz wyglądał tak:

[13] Oczywiście może ona przy tym wykonywać pewne dodatkowe operacje

```
// Functions – przykład własnych funkcji

#include <iostream>
#include <conio.h>

void PokazTekst()
{
   std::cout << "Umiem juz pisac wlasne funkcje! :)" << std::endl;
}

void main()
{
   PokazTekst();
   getch();
}
```

Po kompilacji i uruchomieniu programu nie widać większych zmian – nadal pokazuje on komunikat w oknie konsoli (oczywiście o innej treści, ale to raczej średnio ważne :)). Jednak wyraźnie widać, że kod uległ poważnym zmianom. Na czym one polegają?
Otóż wydzieliliśmy fragment odpowiedzialny za wypisywanie tekstu do osobnej funkcji o nazwie `PokazTekst()`. Jest ona teraz wywoływana przez naszą główną funkcję, `main()`. Zmianie uległ więc sposób działania programu – rozpoczyna się od funkcji `main()`, ale od razu „przeskakuje" do `PokazTekst()`. Po jej zakończeniu ponownie wykonywana jest funkcja `main()`, która z kolei wywołuje funkcję `getch()`. Czeka ona na wciśnięcie dowolnego klawisza i gdy to nastąpi, wraca do `main()`, której jedynym zadaniem jest teraz zakończenie programu.

# *Tryb śledzenia*

Przekonajmy się, czy to faktycznie prawda! W tym celu uruchomimy nasz program w specjalnym **trybie krokowym**. Aby to uczynić wystarczy wybrać z menu opcję *Debug|Step Into* lub *Debug|Step Over*, ewentualnie wcisnąć F11 lub F10.
Zamiast konsoli z wypisanym komunikatem widzimy nadal okno kodu z żółtą strzałką, wskazującą nawias otwierający funkcję `main()`. Jest to aktualny **punkt wykonania** (ang. *execution point*) programu, czyli bieżąco wykonywana instrukcja kodu – tutaj początek funkcji `main()`.
Wciśnięcie F10 lub F11 spowoduje „wejście" w ową funkcję i sprawi, że strzałka spocznie teraz na linijce

```
PokazTekst();
```

Jest to wywołanie naszej własnej funkcji `PokazTekst()`. Chcemy dokładnie prześledzić jej przebieg, dlatego wciśniemy F11 (lub skorzystamy z menu *Debug|Step Into*), by skierować się do jej wnętrza. Użycie klawisza F10 (albo *Debug|Step Over*) spowodowałoby ominięcie owej funkcji i przejście od razu do następnej linijki w `main()`. Oczywiście mija się to z naszym zamysłem i dlatego skorzystamy z F11.
Punkt wykonania osiadł obecnie na początku funkcji `PokazTekst()`, więc korzystając z któregoś z dwóch używanych ostatnio klawiszy możemy umieścić go w jej kodzie. Dokładniej, w pierwszej linijce

```
std::cout << "Umiem juz pisac wlasne funkcje! :)" << std::endl;
```

Jak wiemy, wypisuje ona tekst do okna konsoli. W tym momencie użyj Alt+Tab lub jakiegoś innego windowsowego sposobu, by przełączyć się do niego. Przekonasz się (czarno na czarnym ;)), iż jest całkowicie puste. Wróć więc do okna kodu, wciśnij

F10/F11 i ponownie spójrz na konsolę. Zobaczysz naocznie, iż `std::cout` faktycznie wypisuje nam to, co chcemy :D
Po kolejnych dwóch uderzeniach w jeden z klawiszy powrócimy do funkcji `main()`, a strzałka zwana punktem wykonania ustawi się na

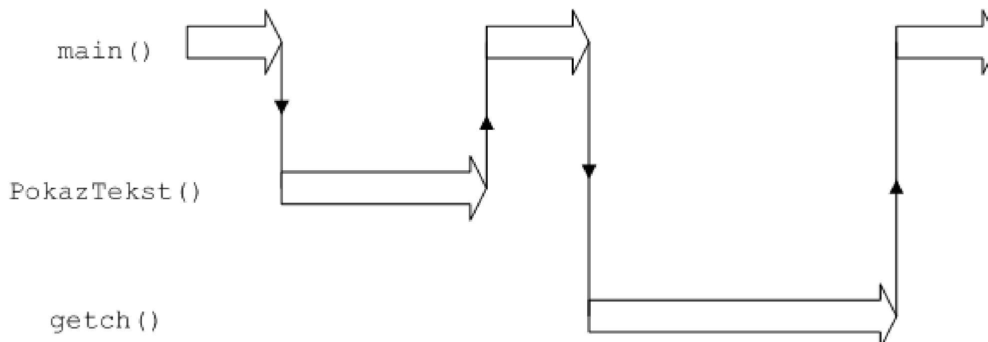
```
getch();
```

Moglibyśmy teraz użyć F11 i zobaczyć kod źródłowy funkcji `getch()`, ale to ma raczej niewielki sens. Poza tym, darowanemu koniowi (jakim są z pewnościa tego rodzaju funkcje!) nie patrzy się w zęby :) Posłużymy się przeto klawiszem F10 i pozwolimy funkcji wykonać się bez przeszkód.
Zaraz zaraz… Gdzie podziała się strzałka? Czyżby coś poszło nie tak?… Spokojnie, wszystko jest w jak najlepszym porządku. Pamiętamy, że funkcja `getch()` oczekuje na przyciśnięcie dowolnego klawisza, więc teraz wraz z nią czeka na to cała aplikacja. Aby nie przedłużać nadto wyczekiwania, przełącz się do okna konsoli i uczyń mu zadość :)
I tak oto dotarliśmy do epilogu – punkt wykonania jest teraz na końcu funkcji `main()`. Tu kończy się kod napisany przez nas, na program składa się jednak także dodatkowy kod, dodany przez kompilator. Oszczędzimy go sobie i wciśnięciem F5 (tudzież *Debug*|*Continue*) pozwolimy przebiec po nim sprintem i w konsekwencji zakończyć program.

Tą oto drogą zapoznaliśmy się z bardzo ważnym narzędziem pracy programisty, czyli trybem krokowym, pracą krokową lub trybem śledzenia[14]. Widzieliśmy, że pozwala on dokładnie przestudiować działanie programu od początku do końca, poprzez wszystkie jego instrukcje. Z tego powodu jest nieocenionym pomocnikiem przy szukaniu i usuwaniu błędów w kodzie.

# *Przebieg programu*

Konkluzją naszej przygody z funkcjami i pracą krokową będzie diagram, obrazujący działanie programu od początku do końca:




**Schemat 2. Sposób działania przykładowego programu**

Czarne linie ze strzałkami oznaczają wywoływanie i powrót z funkcji, zaś duże białe – ich wykonywanie. Program zaczyna się u z lewej strony schematu, a kończy po prawej; zauważmy też, że w obu tych miejscach wykonywaną funkcją jest `main()`. Prawdą jest zatem fakt, iż to ona jest główną częścią aplikacji konsolowej.
Jeśli opis słowny nie był dla ciebie nie do końca zrozumiały, to ten schemat powinien wyjaśnić wszystkie wątpliwości :)

[14] Inne nazwy to także przechodzenie krok po kroku czy śledzenie

# Zmienne i stałe

Umiemy już wypisywać tekst w konsoli i tworzyć własne funkcje. Niestety, nasze programy są na razie zupełnie bierne, jeżeli chodzi o kontakt z użytkownikiem. Nie ma on przy nich nic do roboty oprócz przeczytania komunikatu i wciśnięcia dowolnego klawisza. Najwyższy czas to zmienić. Napiszmy więc program, który będzie porozumiewał się z użytkownikiem. Może on wyglądać na przykład tak:

```
// Input - użycie zmiennych i strumienia wejścia

#include <string>
#include <iostream>
#include <conio.h>

void main()
{
   std::string strImie;

   std::cout << "Podaj swoje imie: ";
   std::cin >> strImie;
   std::cout << "Twoje imie to " << strImie << "." << std::endl;

   getch();
}
```

Po kompilacji i uruchomieniu widać już wyraźny postęp w dziedzinie form komunikacji :) Nasza aplikacja oczekuje na wpisanie imienia użytkownika i potwierdzenie klawiszem ENTER, a następnie chwali się dopiero co zdobytą informacją.

Patrząc w kod programu widzimy kilka nowych elementów, zatem nie będzie niespodzianką, jeżeli teraz przystąpię do ich omawiania :D

## *Zmienne i ich typy*

Na początek zauważmy, że program **pobiera** od nas pewne **dane** i wykonuje na nich operacje. Są to działania dość trywialne (jak wyświetlenie rzeczonych danych w niezmienionej postaci), jednak wymagają przechowania przez jakiś czas uzyskanej porcji informacji.
W językach programowania służą do tego zmienne.

**Zmienna** (ang. *variable*) to miejsce w pamięci operacyjnej, przechowujące pojedynczą wartość określonego typu. Każda zmienna ma nazwę, dzięki której można się do niej odwoływać.

Przed pierwszym użyciem zmienną należy **zadeklarować**, czyli po prostu poinformować kompilator, że pod taką a taką nazwą kryje się zmienna danego typu. Może to wyglądać choćby tak:

```
std::string strImie;
```

W ten sposób zadeklarowaliśmy w naszym programie zmienną typu `std::string` o nazwie `strImie`. W deklaracji piszemy więc najpierw typ zmiennej, a potem jej nazwę. Nazwa zmiennej może zawierać liczby, litery oraz znak podkreślenia w dowolnej kolejności. Nie można jedynie zaczynać jej od liczby. W nazwie zmiennej nie jest także dozwolony znak spacji.

Zasady te dotyczą wszystkich nazw w C++ i, jak sądzę, nie szczególnie trudne do przestrzegania.
Z brakiem spacji można sobie poradzić używając w jej miejsce podkreślenia (`jakas_zmienna`) lub rozpoczynać każdy wyraz z wielkiej litery (`JakasZmienna`).

W jednej linijce możemy ponadto zadeklarować kilka zmiennych, oddzielając ich nazwy przecinkami. Wszystkie będą wtedy przynależne do tego samego typu.

**Typ** określa nam rodzaj informacji, jakie można przechowywać w naszej zmiennej. Mogą to być liczby całkowite, rzeczywiste, tekst (czyli łańcuchy znaków, ang. *strings*), i tak dalej. Możemy także sami tworzyć własne typy zmiennych, czym zresztą niedługo się zajmiemy. Na razie jednak powinniśmy zapoznać się z dość szerokim wachlarzem typów standardowych, które to obrazuje niniejsza tabelka:

| ***nazwa typu*** | ***opis*** |
|---|---|
| `int` | liczba całkowita (dodatnia lub ujemna) |
| `float` | liczba rzeczywista (z częścią ułamkową) |
| `bool` | wartość logiczna (prawda lub fałsz) |
| `char` | pojedynczy znak |
| `std::string` | łańcuch znaków (tekst) |

**Tabela 1. Podstawowe typy zmiennych w C++**

Używając tych typów możemy zadeklarować takie zmienne jak:

```
int nLiczba;              // liczba całkowita, np. 45 czy 12 + 89
float fLiczba;            // liczba rzeczywista (1.4, 3.14, 1.0e-8 itp.)
std::string strNapis;     // dowolny tekst
```

Być może zauważyłeś, że na początku każdej nazwy widnieje tu przedrostek, np. `str` czy `n`. Jest to tak zwana notacja węgierska; pozwala ona m.in. rozróżnić typ zmiennej na podstawie nazwy. Zapis ten stał się bardzo popularny, szczególnie wśród programistów języka C++ - spora ich część uważa, że znacznie poprawia on czytelność kodu.
Szerszy opis notacji węgierskiej możesz znaleźć w Dodatku A.

# *Strumień wejścia*

Cóż by nam jednak było po zmiennych, jeśli nie mieliśmy skąd wziąć dla nich danych?...

Prostych sposobem uzyskania ich jest prośba do użytkownika o wpisanie odpowiednich informacji z klawiatury. Tak też czynimy w aktualnie analizowanym programie – odpowiada za to kod:

```
std::cin >> strImie;
```

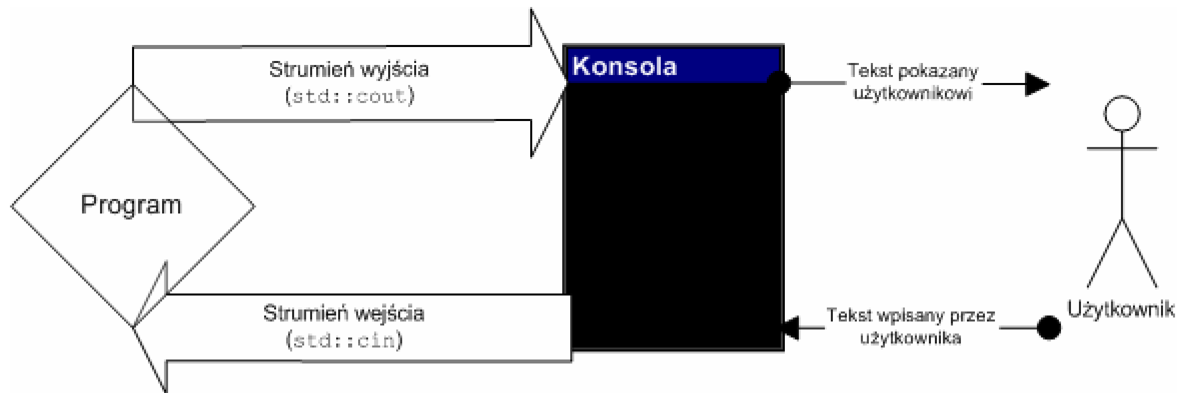
Wygląda on podobnie do tego, który jest odpowiedzialny za wypisywanie tekstu w konsoli. Wykonuje jednak czynność dokładnie odwrotną: pozwala na wprowadzenie sekwencji znaków i zapisuje ją do zmiennej `strImie`.
`std::cin` symbolizuje **strumień wejścia**, który zadaniem jest właśnie pobieranie wpisanego przez użytkownika tekstu. Następnie kieruje go (co obrazują „strzałki” `>>`) do wskazanej przez nas zmiennej.

Zauważmy, że w naszej aplikacji kursor pojawia się w tej samej linijce, co komunikat „Podaj swoje imię”. Nietrudno domyśleć się, dlaczego – nie umieściliśmy po nim `std::endl`, wobec czego nie jest wykonywane przejście do następnego wiersza.

Jednocześnie znaczy to, iż strumień wejścia zawsze pokazuje kursor tam, gdzie skończyliśmy pisanie – warto o tym pamiętać.

***

Strumienie wejścia i wyjścia stanowią razem nierozłączną parę mechanizmów, które umożliwiają nam pełną swobodę komunikacji z użytkownikiem w aplikacjach konsolowych.




**Schemat 3. Komunikacja między programem konsolowym i użytkownikiem**

# Stałe

Stałe są w swoim przeznaczeniu bardzo podobne do zmiennych - tyle tylko że są… niezmienne :)) Używamy ich, aby nadać znaczące nazwy jakimś niezmieniającym się wartościom w programie.

**Stała** to niezmienna wartość, której nadano nazwę celem łatwego jej odróżnienia od innych, często podobnych wartości, w kodzie programu.

Jej deklaracja, na przykład taka:

```
const int STALA = 10;
```

przypomina nieco sposób deklarowania zmiennych – należy także podać typ oraz nazwę. Słówko `const` (ang. *constant* – stała) mówi jednak kompilatorowi, że ma do czynienia ze stałą, dlatego oczekuje również podania jej wartości. Wpisujemy ją po znaku równości `=`.

W większości przypadków stałych używamy do identyfikowania liczb - zazwyczaj takich, które występują w kodzie wiele razy i mają po kilka znaczeń w zależności od kontekstu. Pozwala to uniknąć pomyłek i poprawia czytelność programu.

Stałe mają też tę zaletę, że ich wartości możemy określać za pomocą innych stałych, na przykład:

```
const int NETTO = 2000;
const int PODATEK = 22;
const int BRUTTO = NETTO + NETTO * PODATEK / 100;
```

Jeżeli kiedyś zmieni się jedna z tych wartości, to będziemy musieli dokonać zmiany tylko w jednym miejscu kodu – bez względu na to, ile razy użyliśmy danej stałej w naszym programie. I to jest piękne :)

Inne przykłady stałych:

```
const int DNI_W_TYGODNIU = 7;          // :-)
const float PI = 3.141592653589793;    // w końcu to też stała!
const int MAX_POZIOM = 50;             // np. w grze RPG
```

# Operatory arytmetyczne

Przyznajmy szczerze: nasze dotychczasowe aplikacje nie wykonywały żadnych sensownych zadań – bo czy można nimi nazwać wypisywanie ciągle tego samego tekstu? Z pewnością nie. Czy to się szybko zmieni? Niczego nie obiecuję, jednak z czasem powinno być w tym względzie coraz lepiej :D

Znajomość operatorów arytmetycznych z pewnością poprawi ten stan rzeczy – w końcu od dawien dawna podstawowym przeznaczeniem wszelkich programów komputerowych jest właśnie **liczenie**.

## *Umiemy liczyć!*

Tradycyjnie już zaczniemy od przykładowego programu:

```
// Arithmetic - proste działania matematyczne

#include <iostream>
#include <conio.h>

void main()
{
   int nLiczba1;
   std::cout << "Podaj pierwsza liczbe: ";
   std::cin >> nLiczba1;

   int nLiczba2;
   std::cout << "Podaj druga liczbe: ";
   std::cin >> nLiczba2;

   int nWynik = nLiczba1 + nLiczba2;
   std::cout << nLiczba1 << " + " << nLiczba2 << " = " << nWynik;
   getch();
}
```

Po uruchomieniu skompilowanej aplikacji przekonasz się, iż jest to prosty… kalkulator :) Prosi on najpierw o dwie liczby całkowite i zwraca później wynik ich dodawania. Nie jest to może imponujące, ale z pewnością bardzo pożyteczne ;)

Zajrzyjmy teraz w kod programu. Początkowa część funkcji `main()`:

```
int nLiczba1;
std::cout << "Podaj pierwsza liczbe: ";
std::cin >> nLiczba1;
```

odpowiada za uzyskanie od użytkownika pierwszej z liczb. Mamy tu deklarację zmiennej, w której zapiszemy ową liczbę, wyświetlenie prośby przy pomocy strumienia wyjścia oraz pobranie wartości za pomocą strumienia wejścia.
Kolejne trzy linijki są bardzo podobne do powyższych, gdyż ich zadanie jest niemal identyczne – chodzi oczywiście o zdobycie drugiej liczby naszej sumy. Nie ma więc potrzeby dokładnego ich omawiania.

Ważny jest za to następny wiersz:

```
int nWynik = nLiczba1 + nLiczba2;
```

Jest to deklaracja zmiennej `nWynik`, połączona z przypisaniem do niej sumy dwóch liczb uzyskanych poprzednio. Taką czynność (natychmiastowe nadanie wartości deklarowanej zmiennej) nazywamy **inicjalizacją**. Oczywiście można by zrobić to w dwóch instrukcjach, ale tak jest ładniej, prościej i efektywniej :)

Znak `=` nie wskazuje tu absolutnie na równość dwóch wyrażeń – jest to bowiem **operator przypisania**, którego używamy do ustawiania wartości zmiennych.

Ostatnie dwie linijki nie wymagają zbyt wiele komentarza – jest to po prostu wyświetlenie obliczonego wyniku i przywołanie znanej już skądinąd funkcji `getch()`, która oczekuje na dowolny klawisz.

## *Rodzaje operatorów arytmetycznych*

Znak `+`, którego użyliśmy w napisanym przed chwilą programie, jest jednym z kilkunastu **operatorów** języka C++.

**Operator** to jeden lub kilka znaków (zazwyczaj niebędących literami), które mają specjalne znaczenie w języku programowania.

Operatory dzielimy na kilka grup; jedną z nich są właśnie operatory arytmetyczne, które służą do wykonywania prostych działań na liczbach. Odpowiadają one podstawowym operacjom matematycznym, dlatego ich poznanie nie powinno nastręczać ci problemów. Przedstawia je ta oto tabelka:

| *operator* | *opis* |
|---|---|
| + | dodawanie |
| - | odejmowanie |
| * | mnożenie |
| / | dzielenie |
| % | reszta z dzielenia |

**Tabela 2. Operatory arytmetyczne w C++**

Pierwsze trzy pozycje są na tyle jasne i oczywiste, że darujemy sobie ich opis :)
Przyjrzyjmy się za to bliżej operatorom związanym z dzieleniem.

Operator `/` działa na dwa sposoby w zależności od tego, jakiego typu liczby dzielimy. Rozróżnia on bowiem dzielenie **całkowite**, kiedy interesuje nas jedynie wynik bez części po przecinku, oraz **rzeczywiste**, gdy życzymy sobie uzyskać dokładny iloraz. Rzecz jasna, w takich przypadkach jak `25 / 5`, `33 / 3` czy `221 / 13` wynik będzie zawsze liczbą całkowitą. Gdy jednak mamy do czynienia z liczbami niepodzielnymi przez siebie, sytuacja nie wygląda już tak prosto.
Kiedy zatem mamy do czynienia z którymś z typów dzielenia? Zasada jest bardzo prosta – jeśli obie dzielone liczby są całkowite, wynik również będzie liczbą całkowitą; jeżeli natomiast choć jedna jest rzeczywista, wtedy otrzymamy iloraz wraz z częścią ułamkową.
No dobrze, wynika stąd, że takie przykładowe działanie

```
float fWynik = 11.5 / 2.5;
```

da nam prawidłowy wynik `4.6`. Co jednak zrobić, gdy dzielimy dwie niepodzielne liczby całkowite i chcemy uzyskać dokładny rezultat?... Musimy po prostu obie liczby zapisać

jako rzeczywiste, a więc wraz z częścią ułamkową – choćby była równa zeru, przykładowo:

```
float fWynik = 8.0 / 5.0;
```

Uzyskamy w ten sposób prawidłowy wynik `1.6`.

A co z tym dziwnym „procentem”, czyli operatorem `%`? Związany jest on ściśle z dzieleniem całkowitym, mianowicie oblicza nam **resztę** z dzielenia jednej liczby przez drugą. Dobrą ilustracją działania tego operatora mogą być… zakupy :) Powiedzmy, że wybraliśmy się do sklepu z siedmioma złotymi w garści celem nabycia drogą kupna jakiegoś towaru, który kosztuje 3 złote za sztukę i jest możliwy do sprzedaży jedynie w całości. W takiej sytuacji dzieląc (całkowicie!) 7 przez 3 otrzymamy ilość sztuk, które możemy kupić. Zaś

```
int nReszta = 7 % 3;
```

będzie kwotą, która pozostanie nam po dokonaniu transakcji – czyli jedną złotówką. Czyż to nie banalne? ;)

## *Priorytety operatorów*

Proste obliczenia, takie jak powyższe, rzadko występują w prawdziwych programach. Najczęściej łączymy kilka działań w jedno wyrażenie i wtedy może pojawić się problem **pierwszeństwa** (**priorytetu**) operatorów, czyli po prostu kolejności wykonywania działań.
W C++ jest ona na szczęście identyczna z tą znaną nam z lekcji matematyki. Najpierw więc wykonywane jest mnożenie i dzielenie, a potem dodawanie i odejmowanie. Możemy ułożyć obrazującą ten fakt tabelkę:

| ***priorytet*** | ***operator(y)*** |
|---|---|
| 1 | `*`, `/`, `%` |
| 2 | `+`, `-` |

**Tabela 3. Priorytety operatorów arytmetycznych w C++**

Najlepiej jednak nie polegać na tej własności operatorów i używać nawiasów w przypadku jakichkolwiek wątpliwości.

Nawiasy chronią przed trudnymi do wykrycia błędami związanymi z pierwszeństwem operatorów, dlatego stosuj je w przypadku **każdej** wątpliwości co do kolejności działań.

***

W taki oto sposób zapoznaliśmy się właśnie z operatorami arytmetycznymi.

## *Tajemnicze znaki*

Twórcy języka C++ mieli chyba na uwadze oszczędność palców i klawiatur programistów, uczynili więc jego składnię wyjątkowo zwartą i dodali kilka mechanizmów skracających zapis kodu. Z jednym z nich, bardzo często wykorzystywanym, zapoznamy się za chwilę.

Otóż instrukcje w rodzaju

```
nZmienna = nZmienna + nInnaZmienna;
nX = nX * 10;
```

```
i = i + 1;
j = j - 1;
```

mogą być, przy użyciu tej techniki, napisane nieco krócej. Zanim ją poznamy, zauważmy, iż we wszystkich przedstawionych przykładach po obu stronach znaku = znajdują się **te same zmienne**. Instrukcje powyższe nie są więc przypisywaniem zmiennej nowej wartości, ale modyfikacją już przechowywanej liczby.
Korzystając z tego faktu, pierwsze dwie linijki możemy zapisać jako

```
nZmienna += nInnaZmienna;
nX *= 10;
```

Jak widzimy, operator + przeszedł w +=, zaś * w *=. Podobna „sztuczka" możliwa jest także dla trzech pozostałych znaków działań[15]. Sposób ten nie tylko czyni kod krótszym, ale także przyspiesza jego wykonywanie (pomyśl, dlaczego!).
Jeżeli chodzi o następne wiersze, to oczywiście dadzą się one zapisać w postaci

```
i += 1;
j -= 1;
```

Można je jednak skrócić (i przyspieszyć) nawet bardziej. Dodawanie i odejmowanie jedynki są bowiem na tyle częstymi czynnościami, że dorobiły się własnych operatorów ++ i –– (tzw. **inkrementacji** i **dekrementacji**), których używamy tak:

```
i++;
j--;
```

lub[16] tak:

```
++i;
--j;
```

Na pierwszy rzut oka wygląda to nieco dziwnie, ale gdy zaczniesz stosować tę technikę w praktyce, szybko docenisz jej wygodę.

# Podsumowanie

Bohatersko brnąc przez kolejne akapity dotarliśmy wreszcie do końca tego rozdziału :))
Przyswoiliśmy sobie przy okazji spory kawałek koderskiej wiedzy.

Rozpoczęliśmy od bliskiego spotkania z IDE Visual Studio, następnie napisaliśmy swój pierwszy program. Po zapoznaniu się z działaniem strumienia wyjścia, przeszliśmy do funkcji (przy okazji poznając uroki trybu śledzenia), a potem wreszcie do zmiennych i strumienia wejścia. Gdy już dowiedzieliśmy się, czym one są, okrasiliśmy wszystko drobną porcją informacji na temat operatorów arytmetycznych. Smacznego! ;)

## *Pytania i zadania*

Od niestrawności uchronią cię odpowiedzi na poniższe pytania i wykonanie ćwiczeń :)

---

[15] A także dla niektórych innych rodzajów operatorów, które poznamy później
[16] Istnieje różnica między tymi dwoma formami zapisu, ale na razie nie jest ona dla nas istotna… co nie znaczy, że nie będzie :)

## Pytania

1. Dzięki jakim elementom języka C++ możemy wypisywać tekst w konsoli i zwracać się do użytkownika?
2. Jaka jest rola funkcji w kodzie programu?
3. Czym są stałe i zmienne?
4. Wymień poznane operatory arytmetyczne.

## Ćwiczenia

1. Napisz program wyświetlający w konsoli trzy linijki tekstu i oczekujący na dowolny klawisz po każdej z nich.
2. Zmień program napisany przy okazji poznawania zmiennych (ten, który pytał o imię) tak, aby zadawał również pytanie o nazwisko i wyświetlał te dwie informacje razem (w rodzaju „Nazywasz się Jan Kowalski”).
3. Napisz aplikację obliczającą iloczyn trzech podanych liczb.
4. (**Trudne**) Poczytaj, na przykład w MSDN, o deklarowaniu stałych za pomocą dyrektywy `#define`. Zastanów się, jakie niebezpieczeństwo błędów może być z tym związane.
   *Wskazówka*: chodzi o priorytety operatorów.