Operatory

- C++ pozwala na następujący skrócony zapis: `x += 2` oznacza `x = x + 2`. Podobnie działają operatory `-=`, `*=`, `/=`, `%=`. Są jeszcze inne, podobne skróty dotyczące operacji na poszczególnych bitach, o których za moment powiemy.
- Mamy operatory *logiczne* (których wynikiem jest wartość typu `bool` `true` lub `false`). To są operatory `<` , `<=` , `>` , `>=` , `==` , `!=`. Działają one zgodnie z intuicją, potrzebują dwóch argumentów, które jakoś można porównać. Jedyna uwaga jest taka, żeby nie mylić podstawienia = z porównaniem ==. To częsty błąd, wspominaliśmy już o tym. Oczywiście `!=` oznacza „jest różne od”
- Podobnie, zupełnie zgodnie z intuicją działają operatory sumy i iloczynu logicznego: `&&` , `||`. Operatory te wymagają dwóch wartości typu `bool` (najczęściej są to wyniki działania innych operatorów logicznych), i odpowiadają logicznemu **i** oraz **lub**.

Operatory

- C++ bardzo liberalnie podchodzi do typu wyrażeń, które mogą reprezentować wartość logiczną. Wszystko, co w najbardziej ogólny sposób można zinterpretować jako różne od zera jest interpretowane jako prawda. Na przykład

```
char znak = 'A';
if ( znak ) ... ;
```

W tym przypadku ... będzie wykonane. Po prostu zmienna `znak` ma przypisaną wartość ASCII, i dla `A` wynosi ona 65 = 0x41, więc jest różna od zera. Gdybyśmy podstawili

```
char znak = '\0';
if ( znak ) ... ;
```

to ... nie będzie wykonane, bo znak \0 to właśnie znak o kodzie ASCII 0. Ten niedrukowalny znak jest używany do oznaczenia końca stringów (napisów). Warto o tym pamiętać.

Operatory

- Spójniki logiczne **i** oraz **lub** można oczywiście łączyć. Na przykład

```
if ((znak=='T'||znak=='t') && dalej) ...;
```

W tym przypadku ... będzie wykonane, jeżeli zmienna `znak` będzie zawierała małe lub duże T, oraz zmienna `dalej` będzie miała jakąkolwiek wartość różną od 0. Zwróćmy uwagę na dodatkowy nawias wokół ||. Operatory mają swoje priorytety, i && ma wyższy priorytet niż ||. W takich sytuacjach zawsze warto jest stosować nawiasy, nawet jeżeli z priorytetów wynika, że operacje zostaną wykonane we właściwej kolejności. Jeśli chodzi o priorytety łatwo się pomylić.

Operatory

- W ramach jednego priorytetu operacje wykonywane są od lewej do prawej. Warto pamiętać, że C++ sprawdza prawdziwość wyrażenia tylko do momentu, kiedy może ją stwierdzić. To znaczy, że niektóre wyrażenia bardzo na prawo mogą nie zostać w ogóle wyliczone. To jest subtelność, o której warto pamiętać. Na przykład, jeżeli mamy dwie funkcje, powiedzmy `pierwsza()` i `druga()`, zwracające wartości typu `bool` i gdzieś wyrażenie

```
if ( pierwsza() && druga() ) ... ;
```

to program najpierw wywoła funkcję `pierwsza()` i sprawdzi co zwróciła. Jeżeli zwróciła `false`, to druga funkcja nie zostanie już wywołana.

Operatory

- operator `!` oznacza logiczne zaprzeczenie. Przyjmuje jeden parametr z prawej strony i zwraca `true` lub `false`. Na przykład, jeżeli `k` jest typu `int` to pętla

  ```
  while ( !k ) ... ;
  ```

  będzie wykonywana jeżeli `k` jest równe 0.
- Operatory *bitowe* operują na zmiennych typu całkowitego. Te operatory to `&` , `|` , `^` , `~` , `<<` , `>>`. Dwa ostatnie operatory pojawiają się często w przykładach w operacjach we/wy, ale to jest zupełnie inna rola. W tej chwili powiemy o ich podstawowej funkcji.
- Operatory te przyjmują po dwa parametry, za wyjątkiem `~`, który przyjmuje jeden parametr.

Operatory

- Działanie tych operatorów jest zgodne z intuicją. Parametry są zapisywane w postaci binarnej, a następnie dany operator stosowany jest do kolejnych bitów z osobna. Na przykład, `5 & 10` jest równe 0, bo 5 = 0b101, natomiast 10 = 0b1010. Porównując kolejne bity, idąc od prawej, czyli od bitów najmniej znaczących widzimy, że wszystkie się różnią. Jeżeli przy porównaniu brakuje nam bitów z lewej strony - najbardziej znaczących - oczywiście uzupełniamy zerami. Z drugiej strony `5 | 10` jest równe 15, bo na każdej pozycji (od zeorwego do piątego bita) jest co najmniej jedna 1.
- Operator ~ odwraca każdy bit. Na przykład ~50 = -51. Znowu, łatwo to zauważyć:

```
 50 = 0b 0000 0000 0011 0010
~50 = 0b 1111 1111 1100 1101 = -51
```

Ostatnia równość jest jasna, jeżeli pamiętamy, jak kodowane są liczby ujemne.

Operatory

- Operator ^ przeprowadza na każdej pozycji bitowej operację logiczną **exclusive or**, to znaczy rezultat jest 1 jeżeli odpowiednie bity są równe i 0 w przeciwnym wypadku. Na przykład `12 ^ 21 = 25`. Łatwo to sprawdzić:

```
      12 = 0b 0000 0000 0000 1100
      21 = 0b 0000 0000 0001 0101
12 ^ 21 = 0b 0000 0000 0001 1001 = 25
```

- pozostałe 2 operatory bitowe to przesunięcia. >> to przesunięcie wszystkich bitów w prawo. Z lewej strony dopisywane są zera, bity, które wypadną „z prawego końca" po prostu giną. Lewy parametr operatora to liczba do „przewinięcia", a prawy parametr to ilość pozycji przesunięcia. Na przykład `12 >> 2 = 3`. Łatwo to zauważyć patrząc się na binarną reprezentację 12 powyżej.

Operatory

- << działa przeciwnie, to znaczy przesuwa w lewo. Z prawej strony dopisywane są zera, bity z lewej strony po prostu znikają. Na przykład `12 << 3 = 96`. Przy okazji zauważmy, że dla liczb dodatnich przesunięcie w prawo to dzielenie przez 2 w odpowiedniej potędze (i odrzucenie ew. części ułamkowej), natomiast przesunięcie w lewo tp mnożenie przez 2 w odpowiedniej potędze, z pominięciem ew. „przelewu". Operatorów przesunięcia nie powinno się stosować do liczb ujemnych, wyniki mogą być różne, w zależności od systemu operacyjnego.
- Podobnie jak dla dwuparametrowych operatorów logicznych mamy zapis skrótowy:

```
i &= k oznacza i = i & k
i |= k oznacza i = i | k
i ^= k oznacza i = i ^ k
i <<= k oznacza i = i << k
i >>= k oznacza i = i >> k
```

Operatory

- Operatory bitowe są przydatne w bardzo specyficznych, ale jednocześnie ważnych sytuacjach. Zdarza się, że w programie musimy pamiętać stan różnego rodzaju procesów czy urządzeń. Wygodnie jest wtedy utworzyć zmienną, której poszczególne bity sygnalizują dany stan. Takie bity często nazywane są flagami. Wiele urządzeń jest sterowanych przez swoje rejestry. Zmienne w takich rejestrach nie mają konkretnego znaczenia jako liczby, natomiast poszczególne bity mają znaczenie. Taki rejestr często nazywa się rejestrem flagowym. W przyszłości zobaczymy program napisany w C++ sterujący małym procesorem wbudowanym w urządzenie powszechnego użytku. W takich przypadkach właśnie zastosowanie mają operatory bitowe.

Operatory

- Na przykład wyobraźmy sobie, że zmienna `portA` typu `int` reprezentuje stan wyjścia jakiegoś procesora. Jeżeli chcemy ustawić stan wysoki na którejś z 16 linii, musimy wpisać w odpowiednie miejsce zmiennej odpowiedni bit. Chcemy mieć możliwość sterowania konkretną linią, bez zmiany pozostałych.

```
int portA = 0; // początkowo linie portu są
               //w stanie niskim

const int out3 = (1 << 3); // out3 reprezentuje
        // linię 3. Jest to pojedynczy bit
        // na odpowiedniej pozycji

portA |= out3; // ustawiony zostaje właściwy bit,
               // pozostałe nie ruszone

portA &= ~out3; // skasowany zostaje odpowiedni
                //bit, pozostałe nie ruszone
```

Operatory

- Może się pojawić pytanie, co z innymi operatorami. Oczywiście są jeszcze inne operatory zdefiniowane w C++, które będą się pojawiać w przyszłości, i wtedy je omówimy. Ale jest wiele operatorów, których w C++ nie ma. Na przykład zwykłe podnoszenie do potęgi. To nie jest słabość C++, tylko świadoma decyzja. Wiele operatorów jest przeniesiona do bibliotek. Potrzebujemy takiego operatora - dołączamy odpowiednią bibliotekę. Na przykład podnoszenie do potęgi realizowane jest przez funkcję `pow()`, zawartą w bibliotece `cmath`
- Operatory mają ściśle określone priorytety, to znaczy niektóre są stosowane przed innymi. Nie będziemy tego szczegółowo tutaj pisać, ale trzeba o tym pamiętać i w razie potrzeby sprawdzać. Nigdy nie zaszkodzi zastosować nawiasy (które mają najwyższy priorytet).

Projekt 1.

- Podczas sprawdzania projektów (w mojej grupie laboratoryjnej) nasunęły mi się następujące uwagi.
- Bardzo często kod fatalnie wygląda graficznie. To się może początkowo wydawać nieważne, ale warto jest przyzwyczaić się do pilnowania zasad.
- Jeżeli chcecie Państwo przekonać się o wartości porządnego pisania, proszę poprosić kogoś o pokazanie ich kodu i proszę go przeanalizować. Jest wielka różnica w czytaniu porządnie sformatowanego kodu i kiepsko sformatowanego.

Projekt 1.

- Pamiętajmy, poszczególne bloki funkcjonalne oddzielamy pustą linią, zawsze jedną, a nie dwoma, trzema.
- Kolejne „zagłębienia" kodu (pętle, instrukcje warunkowe itp.) wcinajmy. Codeblocks pomaga w tym, i nie należy z tym walczyć.
- Pamiętajmy o komentarzach. Powinny być krótkie ale dobrze objaśniające. Na przykład, to jest źle:

```
int portA = 0; // zmienna portA
```

  a to jest dobrze:

```
int portA = 0; // stan wyjść portu A
```

- Zmienne powinny być deklarowane w odpowiednich miejscach. Niektóre zmienne są ważne i pełnią rolę w całym programie, deklarujemy je na samym początku, wszystkie. Inne mają znaczenie lokalne, tylko w jakimś bloku, i takie deklarujmy tam, gdzie są potrzebne.

Projekt 1.

- Miejsce deklaracji zmiennej wpływ na sam program wynikowy, może mieć wpływ na wielkość programu w pamięci, i szybkość jego wykonania. Dla programistów C++ to jest ważne, w tym języku pisze się z reguły oprogramowanie systemowe, gdzie wydajność jest ważna. Bardzo dobrze jest mieć świadomość tych zagadnień, i deklaracje zmiennych mieć przemyślane. Najgorsze, co można zrobić, to deklarować zmienne w sposób przypadkowy.
- W wielu przypadkach stosowane były narzędzia o których nie mówiliśmy. Proszę tego nie robić. Celem projektów jest opanowanie poznanych narzędzi. Nie było jeszcze mowy o funkcjach, więc nie stosujmy funkcji. Oczywiście aż się prosi o zastosowanie funkcji, ale nie róbmy tego jeszcze na razie. Przemyślmy przepływ programu, i rozwiążmy to pętlami i instrukcjami warunkowymi. Już w następnym projekcie będziemy mogli rozwinąć skrzydła w dziedzinie funkcji.

Projekt 1.

- W wielu przypadkach stosowane były rozmaite biblioteki. Nie róbmy tego bez przemyślenia. W ogóle nie stosujmy bibliotek, o których nie było mowy. Niektóre biblioteki były specyficzne, i tego już w ogóle nie powinniśmy robić na tym kursie.
- Na przykład biblioteka `windows.h`, która jest specyficzna dla Windowsa, i w ogóle jej nie ma w Linuksie. Nie jest częścią standardu języka C++. Może się zdarzyć, że będziemy pisać program bezpośrednio na procesor na którym w ogóle nie ma systemu operacyjnego. Język C++ właśnie do takich rzeczy się dobrze nadaje. Zostańmy przy tym.
- W niektórych przypadkach zamiast biblioteki `iostream` pojawiała się biblioteka `stdio.h` albo nawet `conio.h`. Zdarzały się przypadki, że biblioteki `iostream` i `stdio.h` były używane jednocześnie.

Projekt 1.

- Biblioteka `conio.h` jest całkowicie przestarzała, i kompilatory mogą jej nie posiadać, w szczególności pod Linuksem. Nie ma żadnego powodu, żeby ją stosować. To wygląda tak, jakby ktoś ściągnął skądś szkielet programu, i tam właśnie taka biblioteka była. Wszystkie funkcje występujące w `conio.h` są też dostępne w `iostream`, i nie ma **żadnego** powodu, żeby stosować egzotyczną, przestarzałą i zawodną bibliotekę do prostych programów. Obawiam się, że w przyszłości będę za takie rzeczy obcinał punkty.
- Biblioteki `stdio.h` i `iostream` są podobne, i żadna z nich nie jest uważana za lepszą od drugiej. W pierwszej występują funkcje takie jak `printf()` i `scanf()`, natomiast w drugiej `cin >>` oraz `cout <<` (oczywiście też wiele innych).

Projekt 1.

- Biblioteki te służą do komunikacji we/wy, i `stdio.h` jest wersją z klasycznego C, natomiast `iostream` z bieżącego C++. Jeżeli jesteśmy doświadczonym programistą C i mamy głęboko zakorzenione przyzwyczajenia, to jak najbardziej stosujmy `stdio.h`. Właśnie dla takich ludzi ta biblioteka została pozostawiona w standardzie C++ (w swojej uwspółcześnionej wersji `cstdio`). Ale jeżeli uczymy się C++ od podstaw, używajmy współczesnej biblioteki `iostream`, chociażby dlatego, żeby ktoś nie pomyślał, że jesteśmy *starej daty*. Całkowitym błędem jest stosowanie obu bibliotek jednocześnie.
- Biblioteki występują pod różnymi nazwami, `stdio`, `stdio.h`, `cstdio`. Bądźmy świadomi tych różnic, i stosujmy nazwy konsekwentnie. Dla biblioteki `stdio` prawidłową nazwą pliku nagłówkowego jest `stdio.h`. Jednak większość kompilatorów wybaczy nam, jeżeli pominiemy `.h`. To jest ta sama biblioteka.

Projekt 1.

- Natomiast `cstdio` jest już nową wersją tej samej biblioteki. Podstawową różnicą jest to, że nazwy zadeklarowane w `cstdio` są w przestrzeni nazw `std` a nie w przestrzeni globalnej. To jest pewna subtelność, do której w przyszłości wrócimy. Jak się można domyślać chodzi o to żeby w rozbudowanych programach unikać konfliktu nazw zmiennych czy innych obiektów z nazwami już istniejącymi. W C++ rozdzielono tak zwane przestrzenie nazw. Trzeba w programie powiedzieć, w jakiej przestrzeni nazw jesteśmy, i w tej przestrzeni jest tylko ograniczona ilość zastrzeżonych nazw. Tak zwana globalna przestrzeń nazw to nazwy zastrzeżone zawsze i wszędzie. Nie powinno się tej przestrzeni „zaśmiecać”.

Projekt 1.

- Kiedy wprowadzono standard języka C++, jako następcy C, pojawiło się właśnie pojęcie przestrzeni nazw. Wszystkie stare biblioteki języka C pozostały aktualne, ale uzupełniono je o przestrzenie nazw. W ten sposób zachowano zgodność ze starymi programami, dodając bieżącą funkcjonalność dla programistów którzy chcą obecnie pisać programy przy użyciu tradycyjnych bibliotek. Podsumowując, jeżeli bardzo chcemy korzystać z dawnych dobrych funkcji `printf()` i `scanf()` to jednak powinniśmy dołączać bibliotekę `cstdio`. Trzeba być świadomym tego co się robi, taki jest nasz cel. Nie mieszajmy tych plików nagłówkowych, i nie dołączajmy `stdio` ani `stdio.h`
- Jeżeli dopiero zaczynamy naukę programowania w C++ to do komunikacji we/wy stosujmy współczesną bibliotekę `iostream`
- Przykładowy program Projektu 1.

Zadanie domowe 3 (do oddania 19.04.)

Napisz program który prosi o podanie trzech liczb naturalnych, a następnie wypisze największą, najmniejszą, średnią arytmetyczną i geometryczną (pierwiastek iloczynu). Każda tych wartości powinna być obliczana przy pomocy odpowiednio zdefiniowanej funkcji. Rozdziel definicję i deklarację funkcji. Dopracuj menu programu, czyli czy użytkownik chce kontynuować, czy zakończyć.
Program (podobnie jak projekty) proszę przesyłać prowadzącym ćwiczenia