Zmienne

- Zmienna przechowuje dane. Program pamięta nazwę zmiennej (przetłumaczoną na swój wewnętrzny *wskaźnik*). Wie więc, gdzie zmiennej w pamięci szukać. Program też wie, jakiego typu jest zmienna, więc wie ile bajtów spod danego adresu odczytać, i jak je potraktować. Pisząc program musimy poinformować kompilator, że potrzebna jest zmienna, jaką ma nazwę i jakiego jest typu. Taka zmienna staje się częścią programu.
- W przyszłości poznamy zmienne, które nie mają nazwy. Odnosi się do nich bezpośrednio przy pomocy ich adresu w pamięci (wskaźnika). Takie zmienne można tworzyć w czasie wykonywania programu, i nie są częścią samego programu. Programista pisząc program nie musi więc z góry wiedzieć, ile taich zmiennych w czasie wykonania programu będzie potrzebnych.

Zmienne

- C++ jest językiem w którym każda zmienna musi być przed użyciem zadeklarowana. C++ ściśle też sprawdza, czy zmienna jest wykorzystywana zgodnie ze swoim typem. Inne języki programowania mogą być po tym względem bardziej wyrozumiałe. Taką filozofię reprezentuje na przykład PHP, który jest stosowany do pisania stron internetowych, czy VBA, który jest dostępny w aplikacjach Microsoft Office. Filozofia C++ jest taka, żeby jak najwcześniej wykrywać błędy w programie. Jeżeli użyta jest nazwa, która wcześniej nie była nigdzie zadeklarowana, albo do zmiennej jednego typu podstawiana jest wartość innego typu, to przypuszczalnie jest to błąd programisty, i zostanie wychwycony już na wstępnym etapie kompilacji.

Zmienne

- Jest pewna ilość typów, które są *wbudowane*, i można z nich korzystać, ale C++ pozwala na tworzenie własnych typów. Tworzenie własnych typów, często bardzo rozbudowanych, jest podstawową funkcjonalnością, i będziemy takie typy tworzyli w przyszłości.
- Na przykład, program implementujący bazę danych o pracownikach może zdefiniować typ zmiennej o nazwie `person`, w której będzie przechowywane imię, nazwisko, PESEL itp. Zmienna może przechowywać nie tylko dane, ale także funkcje związane z danymi. Takie zmienne nazywamy *obiektami*, a związane z nimi typy *klasami*

Typy całkowite

- Typy `int`, `long`, `long long`, `short` to typy służące do przechowywania zmiennych całkowitych. Różnią się ilością zajmowanej pamięci, a więc także największą możliwą do przechowywania wartością. Te typy nie mają twardo zdefiniowanej ilości zajmowanej pamięci, w zależności od systemu operacyjnego mogą zajmować różną ilość bajtów, a więc także mieć różną „pojemność". W systemie Windows `short` to 2 bajty, `int` to 4 bajty, `long` to także 4 bajty, a `long long` to 8 bajtów. Jak łatwo policzyć największą liczbą całkowitą, którą można przechować w zmiennej typu `int` to $2^{32} - 1$ czyli 4294967294. Trzeba o tym pamiętać.

Typy całkowite

- Przestrzeganie pojemności zmiennych należy do programisty. Jeżeli w programie zwiększymy jakąś zmienną poza jej pojemność, nastąpi zwykły „przelew”, podobnie jak w liczniku samochodowym. Liczby które powinny być wielkie, staną się nagle małe albo, co gorsza, ujemne. Ani kompilator w czasie kompilacji, ani program w czasie wykonania nie ostrzeże nas o możliwym przepełnieniu zmiennej
- Każdy z powyższych typów może wystąpić w wersji `signed` (ze znakiem), lub `unsigned` (bez znaku). Domyślną wersją jest `signed`.
- Zakres dopuszczalnych wartości dla `signed int` to w takim razie $-2^{31} \ldots 2^{31} - 1$. Liczby ze znakiem zapisywane są w ten sposób, że ujemne „mapowane” są na dodatnie z zakresu $2^{31} \ldots 2^{32} - 1$ przez operację $x \mapsto 2^{32} + x$ (dla liczb 4 bajtowych). Widać więc, że zwiększanie liczby typu `signed int` poza jej zakres w końcu zmieni ją w ujemna.

Typy całkowite

- Przykłady definicji (definicja może obejmować inicjalizację):

```
int a, c = -3;
unsigned long long ludnosc_swiata = 7238347826;
```

- Typy całkowitoliczbowe mają wielkość zależną od systemu operacyjnego. To jest pozostałość po dawnych czasach, kiedy komputery miały generalnie malo pamięci, i trzeba było się wysilać, żeby nią oszczędnie gospodarować. Podobnie z prędkością obliczeń. Mniej bajtów to szybsze obliczenia. Obecnie nie jest to już takie ważne, i w nowych specyfikacjach standardu C++ zdefiniowane są typy całkowitoliczbowe o konkretnej długości. Jeżeli chcemy z takich korzystać, wystarczy dołączyć odpowiednią bibliotekę. Dla tych nowych typów wykształcił się standard nazw, polegający na tym, że nazwa typu kończy się dołączonym „_t". Na przykład, typ `uint32_t` to zmienna 4 bajtowa bez znaku. Ten typ zdefiniowany jest w bibliotece `cstdint`.

Typ znakowy

- Typ znakowe: `char`. To zmienna zajmująca 1 bajt pamięci, służy do przechowywania znaku. Znak zapisywany jest jako jego kod ASCII. Zmienna `char` przechowuje więc nie tylko znaki drukowalne, takie jak litery, ale także niedrukowalne takie jak tabulacje. Do znaków wrócimy za chwilę.
- Zwróćmy uwagę, że dla polskich znaków takich jak ą, ę nie ma jednego standardowego kodu ASCII. Na przykład, niezależnie jak zdefiniujemy kodowanie znaków w opcjach Codeblocks, w konsoli zawsze zobaczymy „krzaczki” (w Windowsach).
- Konsola w Windowsach stosuje bowiem archaiczny standard kodowania, którego nie wspiera już nawet Codeblocks. Nie jest to wina programu. On wysyła do konsoli znaki jako liczby w dobrej wierze, a to konsola (przypuszczalnie dla zachowania kompatybilności) interpretuje je dziwnie. Zapis polskich znaków w systemie Linux, podobnie jak w Internecie wypracował już uniwersalny standard, tak zwany UTF-8. Wrócimy jeszcze do tego tematu.

Typ znakowy

- Typ `char` występuje również w wersji `signed` i `unsigned` (domyślnie `signed`). Warto o tym pamiętać, szczególnie, kiedy będziemy *rzutowali* zmienną typu `char` na `int`. Polski znak stanie się dziwaczną liczbą ujemną.
- Przykład 1.
- W powyższym przykładzie występują stałe dosłowne typu znakowego, np. 'T'. Zajmiemy się takimi stałymi na następnym wykładzie. Występuje też funkcja `sqrt()` (pierwiastek kwadratowy), która wymaga dołączenia biblioteki `cmath`.
- Typ `bool` przechowuje wartości `true` lub `false`. Pod wieloma względami zachowuje się jak typ całkowity (rozmiaru 1 bajta). Wartości `true` odpowiada 1 a wartości `false` 0.

Typy zmiennoprzecinkowe

- Typy zmiennoprzecinkowe to `float`, `double` i `long double`. Ich wielkość w Windowsach to odpowiednio 4, 8 i 12 bajtów. Typy te mają różny zakres możliwych wartości i różną dokładność. Typowo stosuje się typ `double`
- Typ void. To jest typ pusty. W zasadzie stosowany jest jedynie w sytuacji, gdy funkcja nie zwraca żadnej wartości. W języku C++ deklarując funkcję musimy określić typ zmiennej zwracanej (czyli tej, która jest wynikiem działania funkcji). Jeżeli funkcja nic nie zwraca, nie mielibyśmy co napisać. Piszemy wtedy typ `void`. Podkreśla to, że brak typu zmiennej zwracanej nie jest błędem, tylko jest świadomy. Co prawda typ ten jest pusty, i nie można zdefiniować zmiennej typu `void`, ale `sizeof(void)` zwraca 1 bajt. Co to znaczy zobaczymy w przyszłości.

Typ wyliczeniowy

- Typ wyliczeniowy `enum`. To jest typ, w którym jawnie wypisujemy wszystkie możliwe wartości, które zmienna może przyjąć

```
enum dni_robocze {pon, wt, sr, czw, pt};
dni_robocze seminarium = czw;
cout << seminarium;
```

  Program wypisze: `czw`. Zauważmy, że składnia wygląda tak, jakbyśmy zdefiniowali własny specyficzny typ `dni_robocze`

- Instrukcja `typedef` nadaje nową nazwę istniejącemu typowi zmiennej

```
typedef int cena;
cena cena_piwa, cena_soku;
```

  Zmienne `cena_piwa, cena_soku` są typu cena, który jest identyczny z typem `int`. Taka konstrukcja jest przydatna w wielu sytuacjach

Typy pochodne

- Typy pochodne:
  - `[]` - tablice, np. `int a[16]` - tablica 16 `int`-ów
  - `*` - wskaźniki, np. `int *a` - adres zmiennej `int`
  - `()` - funkcja zwracająca dany typ, np. `int main()`
  - `&` - referencja (adres) obiektu danego typu, np.

```
double *ptr, prom;
ptr = &prom;
```

`prom` jest zmienną typu `double`. `ptr` jest zmienną typu wskaźnik (adres) do zmiennej typu `double`. Następnie do tej zmiennej podstawiony zostaje adres istniejącej już zmiennej `prom`

Nazwy

- Nazwa (identyfikator) zmiennej może składać się z liter, cyfr i podkreśleń. Nie może się zaczynać od cyfry. Musi być unikalna. Oczywiście, nie może pokrywać się z żadnym ze słów kluczowych C++. Nie możemy oczywiście zadeklarować zmiennej o nazwie `int` czy `main`. Takich słów zastrzeżonych jest niewiele, kilkadziesiąt.
- Dodatkowe zastrzeżone nazwy zgromadzone są w tak zwanych przestrzeniach nazw. Jeżeli używamy np. przestrzeni nazw `std`, to dochodzą nazwy takie jak `cin`. Nazwa zmiennej obowiązuje od momentu deklaracji. Dopiero od tego miejsca w programie można się do zmiennej odwoływać.
- Wymyślając nazwy zmiennych czy funkcji warto trzymać się pewnych zasad. Nazwa powinna coś mówić o zawartości. Na przykład `stawkaVAT` jest lepszą nazwą niż `u1`.

Nazwy

- Jeżeli chcielibyśmy użyć nazwy składającej się z wielu słów, to możemy poszczególne słowa zaczynać od dużej litery, albo rozdzielać podkreśleniami. Na przyład `nazwiskoPracownika` albo nazwisko_pracownika. Tradycyjnie pierwsza litera jest mała. Nazwy własnych typów uzupełniamy sufiksem `_t`, a nazwy klas zaczynamy od dużego T
- Przypomnijmy pojęcie deklaracji. Deklaracja informuje kompilator, że obiekt o określonym typie i określonej nazwie występuje w programie. To jest tylko podstawowa informacja.
- Przypomnijmy pojęcie definicji. Definicja robi wszystko to, co deklaracja, ale dodatkowo tworzy daną zmienną w pamięci. Zawiera więc w sobie deklarację.

Nazwy

- Od momentu deklaracji zmiennej możemy posługiwać się jej nazwą. Od momentu definicji zmienna istnieje. Dla zmiennych o których mówiliśmy nie stosuje się oddzielnej deklaracji przed definicją. Instrukcja

  ```
  double prom;
  ```

  jest definicją, i nie wymaga wcześniejszej deklaracji
- Oddzielne deklaracje zmiennych stosujemy tylko w jednym przypadku. Jeżeli nasz program składa się z wielu plików, i w którymś pliku chcemy odwoływać się do zmiennej, która jest zdefiniowana w innym. Pamiętajmy, że kompilator kompiluje każdy plik osobno. Musimy jakoś poinformować kompilator o już istniejącej gdzie indziej zmiennej. A więc w jednym pliku umieszczamy definicję

  ```
  double prom;
  ```

Nazwy

a w innym pliku tylko deklarację, ze słowem kluczowym `extern`:

```
extern double prom;
```

- Gdybyśmy pominęli słowo kluczowe `extern`, to w drugim pliku mielibyśmy zupełnie inną zmienną o tej samej nazwie. Nie byłoby konfliktu nazw, bo *zasięg* każdej z nazw jest ograniczony do co najwyżej pliku.
- Inaczej jest z funkcjami. Deklarację musimy umieścić na początku każdego pliku, w którym z funkcji korzystamy. Deklaracja informuje o tym, jak dana funkcja się nazywa, ile i jakiego typu przyjmuje parametrów (zmiennych wejściowych) oraz jakiego typu zmienną zwraca. Sama definicja funkcji, czyli jej kompletna treść, z reguły znajduje się w innym pliku.

Nazwy

- Każda zadeklarowana nazwa ma swój zasięg. Jeżeli zmienna zadeklarowana jest wewnątrz funkcji, to zasięgiem jest ta funkcja. Poza nią możemy zadeklarować zupełnie inną zmienną o tej samej nazwie, i nie będzie konfliktu. Zmienne zadeklarowane poza jakąkolwiek funkcją (można tak, nawet poza funkcją `main()`) mają nazwy o zasięgu całego pliku. Takie zmienne nazywamy globalnymi. Zmienne zadeklarowane wewnątrz bloku zamkniętego klamrami {....} mają zasięg tego bloku. Zasięgi mogą być zagnieżdżone. Wtedy nazwa najgłębiej zagnieżdżona *przesłania* identyczne nazwy na zewnątrz.
- Przykład 2.
- Zauważmy konstrukcję `::i`. Operator `::` jest operatorem przestrzeni nazw. W tym przypadku mówi: weź zmienną globalną a nie lokalną o tej nazwie

Modyfikatory

- Etykiety mają zawsze zakres funkcji w której się pojawiają. Do etykiet można się odnosić także zanim się pojawią. Nie można skoczyć instrukcją `goto` pomiędzy funkcjami.
- Deklaracje zmiennych mogą posiadać tak zwane modyfikatory. Widzieliśmy już modyfikator `extern`. Modyfikator `const` oznacza zmienną, której wartości nie można zmieniać. Trzeba ją zainicjalizować od razu w ramach definicji, potem nie można już zmieniać.

```
double pi = 3.14;
const double const_pi = 3.14;
```

- Obie zmienne są tego samego typu, i początkowo mają tą samą wartość. Jednak `pi` można nadpisać inną wartością, natomiast próba podstawienia czegokolwiek za `const_pi` spowoduje błąd kompilacji.

Modyfikatory

- Typowym zastosowaniem jest sytuacja, gdzie w programie występuje np. stawka VAT. Nie chcemy jej wpisywać na stałe, bo w przypadku zmiany musielibyśmy wyszukiwać wszystkie miejsca gdzie była zastosowana. Ale też nie chcemy, żeby była zwykłą zmienną, żeby gdzieś przypadkowo jej nie nadpisać.
- Modyfikator `register` odnosi się do tak zwanych zmiennych *automatycznych*, czyli zmiennych, które nie są tworzone w momencie uruchamiania programu, ale są tworzone w miarę potrzeby. Typowy przykład, to zmienne lokalne, definiowane w ramach jakiegoś bloku, albo zmienne występujące w funkcjach.
- Taka zmienna nie ma adresu. Modyfikator `register` powoduje, że zmienna nie jest tworzona w pamięci (z reguły na stosie), ale w specjalnych rejestrach mikroprocesora. Takie zmienne mogą być bardzo szybkie w odczycie i zapisie. Kompilator może jednak zignorować ten modyfikator, jeżeli wszystkie rejestry są zajęte

Modyfikatory

- Modyfikator `volatile` uprzedza kompilator, że wartość zmiennej może się zmienić w momencie, którego kompilator może nie zauważyć (np. wewnątrz obsługi przerwania). Kompilatory z reguły optymalizują kod. Jeżeli kompilator widzi, że zmienna nigdy nie zmienia wartości, to może zastąpić ją stałą wartością. Modyfikator `volatile` zapobiega temu