AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Podstawy C++, część 2

Szymon Buchaniec

http://home.agh.edu.pl/~buchanie/

Akademia Górniczo-Hutnicza, Katedra Podstawowych Problemów Enegetyki

19 grudnia 2018

Zmienne

Przypomnienie i wstęp do tablic oraz struktur

Do przechowywania pewnych wartości, pozwalających opisać dowolny problem, używamy zmiennych o określonych typach. Na razie ograniczamy się do pojedynczych wartości i niepowiązanych zmiennych. Oznacza to, że aby użyć większego stopnia abstrakcji w programie, potrzebujemy sami narzucić pewne waunki, np.: aby opisać koło na dwuwymiarowej płaszczyźnie potrzebujemy 3 zmiennych: double x, y, R;
Dla programu oraz innych osób mających dostęp do naszego kodu, a także dla nas 2 tygodnie po napisaniu programu zmienne te nie są w żaden sposób powiązane. Załóżmy, że chcielibyśmy opisać samochód. Składa się z silnika (opisanego kilkoma zmienymi), kół, wyposażenia... w dodatku chcielibyśmy mieć 50 takich samochodów w programie. Powyższy sposób jest całkowicie niepraktyczny.

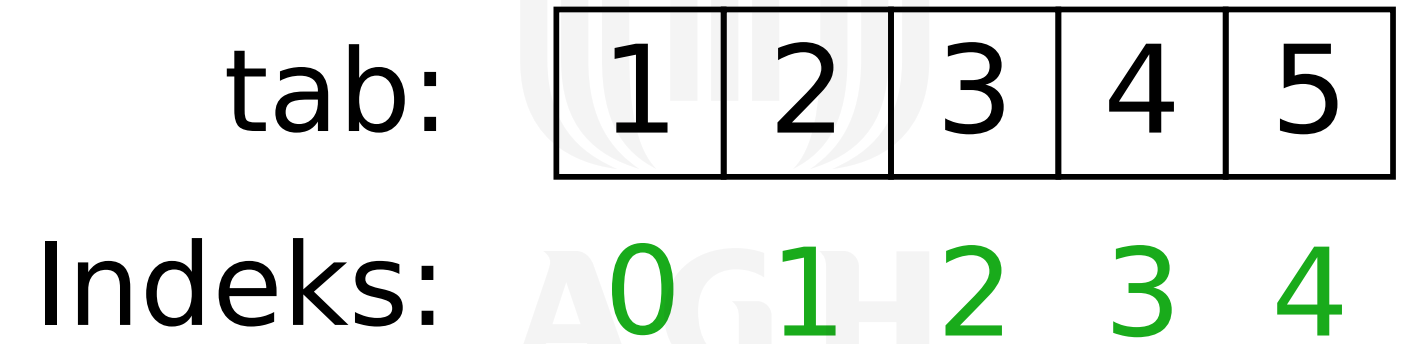
Tablice

Na razie zajmiemy się kwestią ilości podobnych danych. Powiedzmy, że chcemy w programie operować na 100 elementach, np. posiadamy 100 czujników neutrin i chcemy zliczać ilość neutrin dla każdego czujnika. Albo chcemy wygenerować punkty na osi liczbowej, dla których później będziemy obliczać wartość funkcji. Albo nawet punkty na płaszczyźnie lub w przestrzeni. Tworzenie 100 zmiennych o 100 nazwach byłoby co najmniej nieadekwatne i niepraktyczne. Do zadeklarowania dowolnej (ograniczonej pamięcią) ilości zmiennych używamy struktur danych. Podstawową strukturą danych jest **tablica**. Jest to struktura, która przechowuje dane, a do każdego elementu tablicy przyporządkowany jest jednoznacznie **indeks**.

Deklaracja tablicy: typ nazwa_tablicy[rozmiar];. Rozmiar oznacza ilość elementów w tablicy.

Tablice

int tab[5] = { 1, 2, 3, 4, 5 }; \\pierwszy sposób inicjalizacji.
Indeksy tablicy numerowane są od 0 do $n-1$, gdzie n to liczba elementów.



Tablice

Aby odwołać się do elementu tablicy, potrzebujemy nazwy tablicy oraz indeksu, np. tab[2]; Elementów używamy identycznie jak zmiennych, np.

- tab[0] = 1;
- tab[3] = tab[2] + tab[1]*a;

W przypadku indeksacji możemy wykorzystać zmienne typu całkowitego. Musimy tylko pamiętać, aby nie wyjść poza zakres.

```
int i;
```

- tab[i] = tab[i-1] + tab[i*2] + c;

Tablice

Rozmiar podawany podczas inicjalizacji nie powinien być zmienną (może być stałą, znaną podczas kompilacji programu). To znaczy, że możemy:

- const int N = 5; bool tablica[N];

Z tego powodu (stały rozmiar znany na etapie kompilacji) tablice tego typu nazwyamy **statycznymi**. Innym typem tablic, których rozmiar nie musi być znany na etapie kompilacji, a dopiero w trakcie działania programu, są tablice **dynamiczne**. Ze względu na ilość zajęć nie przerobimy tego materiału, ale za to poznamy coś bardziej zaawansowanego, to znaczy klasę vector;

Pętle

Tablice wydają się być bardzo przydatnymi narzędzami do efektywnego zarządzania dużą ilością danych, jednak porzebujemy jeszcze sposobu na ich obsługę. Wyobraźmy sobie, że chcemy wykonać taką samą operację na wszystkich elementach tablicy, np. dodać do wszystkich elementów 1. Naturalną jest potrzeba istnienia narzędzia, które potrafi wykonywać pewną czynność wielokrotnie. Tymi narzędziami są pętle **for** oraz **while**. Różnica między nimi polega tylko na sposobie zapisu - zawsze można zamienić jedną na drugą, lecz w zależności od przypadku jedna z nich może być bardziej wygodna. Jeżeli operujemy na tablicach, używamy pętli **for**.

Pętla for

```
for (init ; condition ; update)
{
  //code
}
```

- init - wykonane tylko raz zaraz przed wykonaniem petli
- condition - wyrażenie logiczne - warunek kontynuowania pętli, sprawdzany przed każdym wykonaniem iteracji
- update - wyrażenie wykonywane po każdej iteracji

Pętla for

Najczęściej: *init* służy nam do inicjalizacji zmiennej, która będzie zmieniana w każdej iteracji (np. int i $=$ 0;), *condition* sprawdza czy zmienna mieści się w odpowiednim zakresie (np. i $<$ 10;), *update* zmienia wartość tej zmiennej (np. i++). Przykład pętli wypisującej kolejno liczby od 9 do 0:

```
for (int i = 9 ; i >=0 ; --i)
{
  std::cout<<i<<" ";
}
```

Pętla for

Tablice

Aby przejść po wszystkich elementach tablicy o rozmiarze N i przypisać do nich zero (zakładamy int tab[10];):

```
for (int i = 0 ; i < N ; ++i)
{
  tab[i] = 0;
}
```

Pętla while

Pętla while przydatna jest w momencie gdy czekamy na jakiś warunek, nie znając dokładnie liczby iteracji, np. czekamy aż użytkownik nie poda poprawnie hasła lub nie pomyli się 3 razy. Pętla while jest dokładnie tym samym co pętla for w postaci **for (; condition ;)**. Składnia pętli while

```
while (condition)
{
  //code
}
```

Instrukcja warunkowa if

Jednym z podstawowych narzędzi w programowaniu jest sterowanie ścieżką programu w trakcie jego wykonania. Np. chcemy liczyć pierwiastki równania kwadratowego tylko gdy delta jest nieujemna. Albo chcemy policzyć ile wartości z tablicy spełnia podany warunek.

if

```
if (condition)
{
  //gdy warunek prawdziwy
}
else if (2nd_condition) //opcjonalne
{
  //gdy pierwszy warunek falszywy
  //a drugi prawdziwy
  //ilosc else if dowolna
}
else //opcjonalne
{
  //gdy wyszstkie wczesniej falszywe
}
```

if

Chcemy policzyć ilość liczb większych od 10 w tablicy:

```
int count = 0;
for (int i = 0; i < N; i++)
{
  if (tab[i] > 10)
  {
    count++;
  }
}
```

Struktury

Struktury są kolejną podstawową strukturą danych. Grupują one zmienne dowolnego typu (także tablice) w celu nadania abstrakcji do naszego programu. Są typami definiowanymi przez użytkownika, możemy np. utworzyć strukturę reprezentującą punkt w 2D, posiadający dwie zmienne typu rzeczywistego - x oraz y. Struktury traktujemy tak samo jak inne typy - możemy tworzyć zmienne o typie danej struktury, tablice struktur czy też jako składowe innych struktur.

Struktury

Deklaracja struktury:

```
struct name
{
   type1 name1; //pola
   type2 name2; //
   ...
}; //wazny srednik
```

Przykład struktury Point oraz zmiennej jej typu:

```
struct Point
{
   double x;
   double y;
};
//definicja zmiennej
// o typie Point
Point punkt1;
```

Struktury

Aby odwołać się do pola struktury używamy kropki (.).

- punkt1.x = 5 + punkt1.y;

Przykład tablicy struktur:

- Point pointTab[100];

Oraz odwołania się do pola x w 10 elemencie tablicy:

- pointTab[9].x = 11;

Klasa vector

Wstęp

Tablice statyczne mogą nie być wystarczające, może być tak, że rozmiar tablicy zależy od przebiegu programu lub jest zmienny w jego trakcie. Moglibyśmy deklarować tablice statyczne o pewnym wystarczającym rozmiarze, jednak nigdy nie ma pewności czy to wystarczy, trzeba dobrze pamiętać jaki jest aktualny rozmiar tablicy (możliwe błędy) a także może to powodować niepotrzebne wykorzystanie dużych obszarów pamięci. Dodatkowo, ze względów technicznych, tablice statyczne mają dodatkowe ograniczenie na maksymalny rozmiar. W celu ominięcia tych niedogodności pwstały tablice dynamiczne, jednak ich obsługa jest niskopoziomowa, wymaga dobrych zdolnosci programistycznyh, a mimo to jest schematyczna. Z tego powodu powstały kontenery - typy pozwalające na przechowywanie wielu wartości w różny sposób, będące uniwersalne i łatwe w obsłudze.

Klasa vector

Jednym z kontenerów w bibliotece standardowej jest vector, który jest bardzo dobrze zaimplementowaną wersją tablicy dynamicznej. Jest zdefiniowany w bibliotece vector. Jego wykorzystanie jest bardzo podobne do tablicy. Przykłady deklaracji:

```
std::vector<double> vec;  //rozmiar 0
std::vector<int> vec2(10);//rozmiar 10
vec2[0] = 1; //dostep do pierwszego elementu
vec2.size(); //zwraca aktualny rozmiar tablicy
vec.push_back(3.1); //dodaje element na koniec
      //tablicy zwiekszajac rozmiar o 1
```

Klasa vector

Przechodzenie po elementach typu vector jest łatwiejsze niż w przypadku tablic. Dwa przykłady robiące to samo:

```
for (int i = 0; i < vec.size(); i++)
{
  vec[i] = 0;
}

//standard c++11
for (auto& x: vec)
{
  x = 0;
}
```