TEORETYCZNE MINIMUM

R. Masełek

29 maja 2020

Spis treści

1 Wyrażenia logiczne i instrukcje warunkowe

Często potrzebujemy wykonać instrukcje w kodzie, tylko jeśli zachodzi jakaś relacja między zmiennymi. Używamy wtedy operatorów: $>, <, >=, <=, ==, != .$ Pełne wyrażenie postaci

```
        x >= y
```

jest wyrażeniem logicznym i w myśl logiki dwuwartościowej ma wartość prawda (w C++ true) lub fałsz (w C++ false). Możemy to wykorzystać tworząc instrukcję warunkową z pomocą słów kluczowych if, else, else if.

Przykład:

```
int n;
std::cin >> n;
if ( n > 0 )
        std::cout << "n jest dodatnie" << std::endl;
else if (n < 0 )
        std::cout << "n jest ujemne" << std::endl;
else
{
        std::cout << "n jest zero" << std::endl;
}
```

Powyżej deklarujemy zmienną typu całkowitego int o nazwie n. Następnie wczytujemy z konsoli jej wartość. Instrukcja warunkowa if sprawdza czy n jest dodatnie, jeśli tak to wyświetla komunikat, jeśli nie to kolejna instrukcja else if sprawdza czy n jest ujemne. Jeśli n jest ujemne to wyświetlony jest komunikat, jeśli nie jest to wykonywany jest blok kodu po else. Zauważmy, że nawiasy klamrowe są opcjonalne jeśli po if/else występuje pojedyncza linia kodu. Jeśli linii jest więcej to należy użyć nawiasów klamrowych. Ich nieużycie niekoniecznie zostanie wychwycone przez kompilator i może prowadzić do błędów w działaniu programu (tzw. bugów).

Wartości logiczne możemy zapisywać do zmiennych typu bool, np.

```
        bool war = 100 % 9 == 1;
```

Powyższe wyrażenie tworzy zmienną logiczną o nazwie war i przypisuje jej true lub false w zależności od wartości wyrażenia po prawej stronie. Po prawej stronie używamy operatora modulo

Możemy stosować operatory logiczne: koniunkcja: && alternatywa: —— negacja: ! Wyrażenia możemy łączyć za pomocą operatorów i nawiasów. Np. można sprawdzić czy liczba n jest w przedziale 1-100 lub nie jest podzielne przez 17:

```
if ( ( n > 0 && n <= 100 ) ||  ( n % 17 != 0  ) )
{
        Jakis kod.
}
```

2 Funkcja

Funkcja jest definicją instrukcji, która na podstawie danych wejściowych (argumentów) dostarcza w miejscu jej użycia (wywołania) wartość określonego typu. Funkcje są więc sposobem na realizację tego samego czy podobnego zadania wielokrotnie, bez potrzeby wielokrotnego powtarzania w naszym kodzie tych samych sekwencji instrukcji.

Definicja funkcji:

```
typ funkcja( typ1 arg1, typ2 arg2, typ3 arg3 )
{
           Tutaj wpisujemy instrukcje
           Funkcja musi konczyc sie wywolaniem instrukcji
           return x;
           gdzie x jest wynikiem dzialania funkcji
```

```
            i jest typu typ. (wyjatkiem jest typ void)
}
```

Argumenty funkcji służą do przesyłania danych do funkcji.

WAŻNE! Nazwy argumentów (w przykładzie są to arg1, arg2,...) są widoczne w funkcji. Jeżeli stworzymy wewnątrz funkcji zmienną lokalną to nie będzie ona widoczna poza funkcją.

Wywołanie funkcji:

Funkcję wywołujemy pisząc jej nazwę, po niej nawiasy okrągłe a w nich podajemy argumenty (przez zmienną, stałą, literał itd.). Na przykład funkcja int silnia(int n) ... Może zostać wywołana poprzez:

```
        silnia(5);
```

Ale również można np. użyć zmiennej:

```
        int x = 5;
        silnia(x);
```

Prawie zawsze chcemy przechwycić rezultat działania funkcji. Robimy to przypisując wywołanie funkcji do zmiennej:

```
        int wynik;
        wynik = silnia(5);
```

WAŻNE Funkcja powinna być zdefiniowana przed jej wywołaniem.

Funkcja main

Każdy program C++ musi posiadać funkcję main. Uruchamiając skompilowany program wykonujemy instrukcje z ciała funkcji main od góry do dołu aż do instrukcji return. Oznacza to, że funkcja main jest centralnym punktem w każdym programie i wszystkie instrukcje muszą być albo bezpośrednio w niej zapisane, lub zapisane w funkcjach/metodach wywoływanych z funkcji main. Dopuszczalne jest zagnieżdżanie wywołań, tzn. funkcja main może wołać inną funkcję, która woła kolejną, a to jeszcze jedną itd. Zasadniczo, funkcja main może być używana bez argumentów, albo z dwoma:

```
int main() //sygnatura funkcji main bez argumentow
int main(int argc, char* argv[]) //sygnatura funkcji main z argumentami
```

Druga postać funkcji main umożliwia nam przekazywanie argumentów do programu z linii poleceń. Argument *argc* jest typu int i określa liczbę argumentów programu z linii poleceń, natomiast argv jest wskaźnikiem do tablicy znaków (można myśleć o nim jak o tablicy, w której każdym elementem jest tablica-ciąg znaków) i zawiera wartości tych parametrów (jako ciągi znaków).

WAŻNE: Zerowym argumentem jest nazwa programu!

Rozważmy przykład:

```
#include <iostream>

int main(int argc, char* argv[])
{
        std::cout << "Nazwa: " << argv[0] << std::endl
        << " Liczba arg: " << argc << std::endl
        << " Argument+1: " << atof(argv[1])+1 << std::endl;

        return 0;
}
```

Wynik działania programu z podanym argumentem:
MacBook-Pro-Rafal:zaj4 rafalmaselek$ ./main 3.14
Nazwa: ./main
Liczba arg: 2
Argument: 4.14

WAŻNE: Dostęp do argumentów wywołania programu uzyskujemy poprzez operator dostępu tablicy []. Ponieważ argumenty są wczytywane jako ciagi znaków, jeżeli argumentem jest liczba to przed wykonaniem jakichkolwiek działań musimy dokonać konwersji na typ liczbowy. Najłatwiej to zrobić korzystając z wbudowanych funkcji *atoi, atof, atol*, które zamieniają ciąg znaków na typy *int double long* odpowiednio.

Funkcje rekurencyjne:

Funkcje rekurencyjne to takie, które wywołują same siebie. Np. funkcja licząca sumę liczb całkowitych dodatnich aż do n (są prostsze metody, to tylko przykład na rekurencję).

```
int suma(int n)
{
        if (n == 1)
                return 1;
        else
                return n+suma(n-1);
}
```

Powyższa funkcja zwróci 1, jeżeli pytamy o sumę liczb całkowitych dodatnich od 1 do 1 (oczywisty wynik). Jeżeli natomiast pytamy o n¿1 to funkcja doda n do wyniku działania siebie samej dla n-1.

WAŻNE Funkcja rekurencyjna musi mieć warunek, który pozwoli na zakończenie rekurencji.

Referencje jako argumenty funkcji

Kiedy wywołujemy jakąś funkcję i w wywołaniu podajemy jej argumenty, to przed wykonaniem jakichkolwiek operacji wewnątrz funkcji, tworzona jest kopia argumentów. Np. jeżeli funkcja przyjmuje jako argument inta, to wywołując ją z użyciem zmiennej typu int, zostanie stworzona kopia wartości tej zmiennej i na niej będą wykonywane wszelkie operacje [1]. Pociąga to trzy ważne konsekwencje:

1. Z wnętrza funkcji nie ma możliwości zmiany wartości zmiennych użytych do wywołania funkcji, gdyż funkcja ma dostęp jedynie do kopii ich wartości a nie samych zmiennych.
2. Zużycie pamięci jest rośnie, gdyż tworzone są kopie wszystkich argumentów. Może to być istotne w przypadku dużych obiektów.
3. Kopiowanie zajmuje dodatkowy czas, który może być zauważalny dla dużych obiektów.

Dlatego w C++ istnieją typy referencyjne. Typ referencyjny należy rozumieć jako adres do zmiennej w pamięci. Typy referencyjne deklaruje się tak samo jak zwykłe, z tym że następuje po nich znak & (ampersand). Przykłady:

```
int a = 5;
int& x;
x = a;
```

Zmienna x jest referencją typu int, tzn. wskazuje na położenie zmiennej typu int. Przypisanie w trzeciej linii sprawia, że x staje się inną nazwą (aliasem) zmiennej a. Jeżeli teraz zmienimy wartość a, to wartość x również się zmieni.

Przeciążanie funkcji

Bardzo często chcemy mieć funkcję, która działa dla kilku typów danych, np. funkcję *potęga* chcielibysmy używać zarówno dla liczb typu unsigned, jak i int, long, double, float itd. Jednym ze sposobów aby to osiągnąć jest **prze-**

[1] Są wyjątki. Np. dla tablic w C++ zachodzi automatyczna konwersja na wskaźnik.

ciążanie funckji. Przeciążanie polega po prostu na napisaniu kilku wersji funkcji dla różnych typów argumentów i wartości zwracanych. Na przykład jeżeli chcemy napisać funkcję *potęga*:

```
int potega(int x, unsigned n)
{
...
}

double potega(double x, unsigned n)
{
...
}
```

WAŻNE: Funkcje nie mogą różnić się jedynie typem zwracanego argumentu.

Szablony funkcji

Jeżeli chcemy napisać funkcję operującą na dwóch-trzech typach argumentów to przeciążanie jest dobrym wyborem. Jeżeli jednak chcemy napisać bardzo ogólną funkcję, która ma działać dla wielu typów, lepiej jest użyć **szablonów funkcji**. Szablony to przepis na podstawie którego kompilator sam stworzy wersje funkcji dla różnych typów. Składnia definicji szablonu funkcji wygląda nastepująco:

```
template <typename T, typename U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

W powyższym przykładzie zdefiniowaliśmy funkcję GetMin, która przyjmuje dwa argumenty, które mogą być różnych typów (T i U), a następnie zwraca mniejszą z nich typu tego samego co pierwszy argument. Definicja zaczyna się od słowa kluczowego *template* po którym następuje lista używanych typów. Listę otwiera znak ¡ i kończy ¿. Typy określamy słowem kluczowym *typename* po którym podajemy alias, który od tej pory będzie oznaczał dany typ. Poźniej mamy standardową definicję funkcji, z tym, że typ zwracany i typy argumentów są typami szablonu. Tak zdefiniowana funkcja będzie działać dla każdego typu, pomiędzy którymi można używać operatora ¡.

3 Pętla

Pętle pozwalają wykonywać instrukcje wielokrotnie, bez potrzeby kopiowania kodu.

Pętla for

Używamy, gdy wiemy z góry ile razy chcemy wykonać dane instrukcje. Składnia wygląda tak:

```
for( int ii=0; ii<34; ii++)
{
        Tu wpisujemy instrukcje.
}
```

Po słowie kluczowym for piszemy nawiasy okrągłe. Następnie definiujemy zmienną pomocniczą typu int i przypisujemy jej wartość 0. Stawiamy średnik. Podajemy warunek logiczny, który będzie sprawdzany przed wykonaniem pętli (przed każdą kolejną iteracją). Gdy warunek nie będzie spełniony, program opuści pętlę.

Pętla for-each

Często gdy iterujemy po tablicy czy kolekcji to nie interesuje nas indeks elementu w danej iteracji a jedynie sam element. Nowsze standardy C++ oferują bardzo wygodną funkcjonalność tworzenia takich odchudzonych pętli for, zwanych pętlami for-each:

```
double wartosci[5] = {3.4, 5.1, 0.0, -4.2, 9.9};
for( double w : wartosci)
{
        std::cout << w << " ";
}
```

W powyższym przykładzie mamy tablicę pięciu wartości typu double, które chcemy wypisać. Używamy do tego pętli for-each. Składnie jest następująca: piszemy słowo kluczowe for, po nim nawiasy okrągłe. W nawiasach definiujemy nową zmienną, takiego samego typu jak elementy tablicy/kolekcji po której chcemy iterować. W przykładzie jest to "double w". Następnie piszemy dwukropek, a po nim nazwę tablicy/kolekcji (etykietę). Wewnątrz pętli mamy dostęp do zmiennej w, która w każdej iteracji ma inną wartość odpowiadającą kolejnym elmentom tablicy. Zatem output jaki dostaniemy to będzie "3.4 5.1 0.0 -4.2 9.9".

Pętla while

Używamy, gdy nie wiemy z góry ile razy trzeba wykonać instrukcje. Składnia:

```
while( warunek)
{
        Tu wpisujemy instrukcje.
}
```

warunek jest tutaj wyrażeniem logicznym, które jest sprawdzane przed wykonaniem pętli. WAŻNE Trzeba zawsze zadbać, żeby pętla while w którymś momencie się skończyła, inaczej będzie się wykonywać w nieskończoność! Można np. zdefiniować przed pętlą zmienną typu bool, przypisać do nie wartość true, w pętli w którymś momencie ustawić wartość zmiennej na false, wtedy następna iteracja nie wykona się.

Pętla do while

Podobna do pętli while. Używamy jej gdy chcemy, żeby instrukcje wewnątrz pętli wykonały się przynajmniej raz. Składnia:

```
do
{
Tu wpisujemy instrukcje.
}
while ( warunek) ;
```

WAŻNE Warunek jest sprawdzany po wykonaniu instrukcji!

4 Tablice

Tablice pozwalają przechowywać wiele wartości danego typu. Np. jeżeli chcemy przechowywać 4 liczb naturalnych, możemy zdefiniować sobie tablicę:

```
int tab[4] = {1, 0, -1, 2};
```

Nie musimy ich od razu wypełniać. Dostęp do elementów tablicy możemy uzyskać poprzez użycie nazwy, po niej nawiasów kwadratowych, w których podajemy indeks elementu. Np. jeśli chcemy wypełnić tablicę intów liczbami od 9 do 0 i wypisać to możemy napisać:

```
        int tab[10];
        for ( int ii=9; ii >=0; ii--)
        {
                tab[9 - ii] = ii;
        }
        for (int jj = 0; jj < 10; jj++)
        {
```

```
                std::cout << tab[jj] << " ";
        }
```

Możemy tworzyć również tablice wielowymiarowe, np.:

```
int x[2][3][4] =
 {
   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
   { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };
```

Dostep do danych mamy podobnie jak w tablicach jednowymiarowych poprzez podanie indeksu w nawiasach kwadratowych, tylko tym razem podajemy indeksów tyle, ile wynosi wymiar tablicy, a każdy z indeksów w nowej parze nawiasów kwadratowych.

WAŻNE Elementy tablic indeksujemy od 0.

Jak zrobić int z char? Musimy pamiętać, że znak siódemki '7' i całkowita liczba 7 to dla komputera zupełnie różne rzeczy. Można jednak łatwo przekonwertować '7' na 7. Wystarczy wiedzieć, że każdy znak ASCII ma swój numer i możemy rzutować ten znak na typ int za pomocą konstrukcji:

```
char znak = '7';
int war = (int) znak;
```

W ten sposób dostaniemy wartość liczbową, ale wcale nie 7! To dlatego, że znak '7' nie jest siódmym znakiem w tabeli ASCII. Ale nie musimy wiedzieć, który jest, wystarczy wiedzieć, że znaki cyfr są ułożone po kolei. Ponieważ dla typu char zdefiniowany jest operator odejmowania, możemy napisać:

```
(int) ( '7' - '0' )
```

5 Wskaźniki

5.1 Wskaźniki do zmiennych

Wskaźniki są centralnym elementem w języku C++, który nastręcza wielu trudności początkującym programistom.

Dla każdego typu, zarówno wbudowanego jak i stworzonego przez użytkownika, istnieje stowarzyszony typ wskaźnikowy. Możemy utworzyć zmienną takiego typu. W przeciwieństwie do zwykłej zmiennej, która zawiera wartość (liczbę, znak, tablicę, obiekt itd.), wartością zmiennej typu wskaźnikowego jest **adres innej zmiennej**, która jest stowarzyszonego typu. I tak np. wskaźnik na double zawiera adres zmiennej, która ma wartość double. Wskaźnik na obiekt typu Klasa1, zawiera adres do obiektu tego typu. Wynikają stąd następujące własności wskaźników:

- Wskaźnik na dany typ nie jest tym samym co zmienna danego typu, nie można więc wykonywać na nim operacji zdefiniowanych dla danego typu.
- Wskaźnik wskazuje adres zmiennej w pamięci, korzytając z tego można odczytać lub zmodyfikować tą wartość.
- Ponieważ wskaźnik zawiera tylko adres, a nie wartość, wydajnie jest przekazywać z funkcji do funkcji wskaźniki do obiektów zamiast samych obiektów (pamiętaj, że zmienna przekazywana do funkcji jest kopiowana, mniej pamięci i czasu zajmuje skopiowanie adresu do dużego obiektu niż samego obiektu.).

Praca ze wskaźnikami w C++ związana jest z operatorem gwiazdki "*", który pełni kilka ról.
Zacznijmy od zdefiniowania wskaźnika typu int:

```
int x = 4;
int* y = &x;
```

W pierwszej linii powyżej stworzyliśmy i zainicjalizowaliśmy zmienną typu int. W drugiej linii stworzyliśmy zmienną typu wskaźnikowego wskazującego na zmienną typu int oraz przypisaliśmy coś do niej. Widzimy, że wskaźnik tworzymy dodając gwiazdkę po nazwie zwykłego typu. To co przypisujemy po prawej stronie to adres zmiennej x, uzyskujemy go poprzez podanie nazwy zmiennej poprzedzonej znakiem ampersand &.

Jak odczytać lub zmodyfikować wartość wskazywaną przez wskaźnik? Trzeba ponownie użyć operatora gwiazdki (nazywanego w tym kontekście operatorem wyłuskania):

```
int odczytane = *y;
*y = 8;
```

W pierwszej linii powyższego kodu tworzymy nową zmienną int i przypisujemy do niej wartość wskazywaną przez zmienną typu wskaźnikowego y. Stawiając gwiazdkę przed nazwą zmiennej zaznaczamy, że nie chodzi nam o jej wartość, tylko wartość zmiennej, którą wskazuje. W drugiej linijce zmieniamy wartość zmiennej wskazywanej przez wskaźnik. Ponownie używamy operatora gwiazdki przed nazwą zmiennej.

Można tworzyć wskaźniki do wskaźników, w takim przypadku dajemy wiecej gwiazdek:

```
int x = 4;
int* y = &x;
int** z = &y;
**z = 10;
```

5.2 Arytmetyka wskaźników

Na wskaźnikach możemy wykonywać pewne operacje, przydatne zwłaszcza przy tablicach i wektorach. Wskaźniki możemy inkrementować (zwiększać) lub dekrementować (zmniejszać), dzięki czemu możemy przechodzić od jednego elementu tablicy/wektora do następnego. Składnia jest taka sama jak dla lcizbowych typów całkowitych, np.:

```
double * tab = new double[4]; //tworzymy tablice na stosie
//teraz wskznik tab wskazuje na pierwszy element
std::cout << tab[3]; //wypisz czwarty element
std::cout << *(tab+3); //wypisz czwarty element
```

Możemy również zrealizować pętlę for po elementach wektora, bez podawania jawnie jego długości z wykorzystaniem tzw. iteratora, który jest wskaźnikiem:

```
std::vector<int> v;

//robimy cos z wektorem, np. uzupelniamy

for ( std::vector<int>::iterator iter = v.begin(); iter != v.end(); iter++)
{
        std::cout << *iter << " ";
}
```

W powyższym przykładzie zmienną wewnętrzną pętli for jest tzw. iterator, będący tak na prawdę wskaźnikiem na obiekty typu int wewnątrz wektora. Inicjalizujemy go adresem pierwszego elementu wektora, używając metody składowej **begin()**. Pętla będzie się wykonywać tak długo, jak wskaźnik nie wskaże na **v.end()** oznaczające koenic wektora (nie ostatnią wartość tylko "dalej"). Iterator zmieniamy tak jak zmienną int w zwykłej pętli for, za pomocą operatora ++. Aby uzyskać wartość wskazywaną w danej iteracji przez iterator, używamy operatora wyłuskania.

5.3 Wskaźnik pusty

Często przydatne jest używanie wskaźnika nie zawierającego adresu żadnej zmiennej, zwanego wskaźnikiem null. Aby uzyskać taki wskaźnik, tworzymy zmienną wskaźnikową (dowolnego typu) i przypisujemy do niej wyrażenie **nullptr**:

```
char* wsk = nullptr;
```

Wskaźniki puste przydają się np. gdy mamy funkcję, zwracającą wskaźnik. Kiedy dostajemy wynik działania takiej funkcji to nie powinniśmy odczytywać wskazywanej zmiennej, jeżeli nie mamy pewności, że adres jest faktycznie zapisany do wskaźnika. Możemy zatem najpierw sprawdzić czy wskaźnik ma wartość nullptr, a jeśli nie ma, to próbować odczytać wartość wskazywanej zmiennej.

5.4 Wskaźniki funkcyjne

Funkcje w C++ również są obiektami i można do niech tworzyć wskaźniki. Rozpatrzmy przykłady:

```
int (*fun)(int);
double (*fun[3])(double);
```

W pierwszej linijce deklarujemy wskaźnik funkcyjny o nazwie fun. Funkcja na którą wskazuje, przyjmuje jeden argument typu int oraz zwraca typ int. W drugiej linijce deklarujemy zmienną fun, która jest 3 elementową tablicą wskaźników funkcyjnych na funkcje przyjmujące jeden argument typu double i zwracające double.

Pouczający jest poniższy przykład, w którym wykorzystujemy wskaźnik funkcyjny na funkcję przyjmującą jeden argument typu double oraz zwraca double. Następnie przypisujemy do tego wskaźnika różne funckje i wywołujemy je. Wynik działania wypisujemy na cout. Na przykładzie widać, jak ten sam efekt można uzyskać przy użyciu różnej składni.

```
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 4*atan(1.);

double nasza(double);

int main() {
                double (*f)(double);

                f = sin;
                cout << "sin(PI/2) =  " << (*f)(PI/2) << endl;

                f = &cos;
                cout << "  cos(PI) = "  << f(PI)      << endl;

                f = nasza;
                cout << " nasza(3) =  " << f(3)       << endl;
        }

double nasza(double x) {
        return x*x;
}
```

5.5 Tablice a wskaźniki

C++ nie dopuszcza tworzenia tablic referencji, ale można tworzyć tablice wskaźników. Składnia jest taka sama jak dla zwykłych typów.

```
double* tab[10];
```

Bardzo ważną rzeczą jest fakt, że tablice w C++ są nierozerwalnie związane ze wskaźnikami. W praktyce zmienną tablicową można utożsamiać ze wskaźnikiem na pierwszy element tej tablicy. W szczególności poprawne jest poniższe przypisanie:

```
        double* tab[10];
        double ** wsk;
        wsk = tab;
```

Ponieważ tablica typu T i wskaźnik typu T* na pierwszy element tablicy to zazwyczaj to samo[2], więc możemy używać

[2] **Zazwyczaj** dlatego, że kiedy przekazujemy tablicę do funkcji jako argument, to zawsze zachodzi niejawna konwersja z typu tablicowego na typ wskaźnikowy. Jeżeli natomiast utworzymy sobie w danym zakresie zmienną lokalną tablicową to ona nie jest traktowana jako wskaźnik, chyba, że jawnie dokonamy konwersji.

arytmetyki wskaźników dla tablic. Np. poniższy kod inicjalizuje tablicę 5 liczb typu int, a następnie wypisuje je za pomocą wskaźników i pętli do while:

```
#include <iostream>

int main()
{
        int tab[5] = {1, 2, 4, 7, 999};
        int* wsk = tab; //ta konwersja jest potrzebna
        do
        {
                std::cout << *wsk << " ";
                wsk++;
        }
        while(wsk != &tab[0]+5);
        return 0;
}
```

Powyższy przykład jest bardzo pouczający. Po pierwsze widzimy, że aby używać arytmetyki wskaźników musimy zrzutować zmienną tablicową na typ wskaźnikowy. Taka konwersja zachodzi zawsze, gdy podajemy tablicę jako argument do funkcji. Następnie w pętli wypisujemy wartość liczbową wskazywaną przez wskaźnik wsk za pomocą operatora wyłuskania (*). Po wyłuskaniu inkrementujemy zmienną wskaźnikową, jest to równoznaczne ze zmianą wskazywania wskaźnika na następny element w tablicy. Warunkiem wykonania pętli jest to, żeby wskaźnik wskazywał na coś innego niż

```
&tab[0]+5
```

które jest niczym innym jak adresem w pamięci odpowiadającym elementowi tablicy na pozycji z indeksem 5 [3]. Ale ten element nie istnieje! Ostatni element ma indeks 4, więc wskaźnik wskazuje na coś na co nie powinien. To nie jest problem tak długo, jak nie próbujemy odczytać jego wartości. Właśnie dlatego używamy tutaj pętli do while a nie while, co gwarantuje nam, że nie nastąpi czytanie z zabronionego obszaru pamięci, które skończyłoby się w najlepszym wypadku błędnym działaniem programu, a w większości przypadków jego crashem.

6 Dynamiczna alokacja pamięci

6.1 Funkcje

Niedogodnością w korzystaniu z funkcji jest to, że jeśli utworzymy w nich jakąś zmienną, to ta zmienna zostanie usunięta po zakończeniu funkcji. O ile możemy zwrócić jakąś wartość, to tylko jedną i w tym przypadku nastąpi kopiowanie jej wartości, co dla dużych obiektów jest kłopotliwe. Możemy posłużyć się referencjami, ale wtedy musimy utworzyć zmienne poza funkcją co jest niewygodne. Jednym z rozwiązań tego problemu jest dynamiczna alokacja pamięci. Rozważmy przykład.

```
int* pi = new int;
delete pi;
```

W pierwszej linii stworzyliśmy zmienną wskaźnikową pi i przypisaliśmy do niej wynik działania instrukcji **new int**. Operator **new** alokuje pamięć na zmienną typu takiego jak podany zaraz po nim i zwraca wskaźnik na tą zmienna. Pamięć ta alokowana jest na stosie. Od praktycznej strony oznacza to, że ta **pamięć nie będzie zwolniona, póki sami tego nie zrobimy albo program nie zakończy pracy**. W drugiej linijce używamy operaotra **delete** aby zwolnić pamięć po tej zmiennej. Rozpatrzmy bardziej rozbudowany przykład:

```
int* f(){
        int* x = new int(7);
        return x;
```

[3] Widzimy, że moglibyśmy napisać na początku programu int* wsk = &tab [0]osiągając ten sam efekt co wcześniej. Widzimy również, że tab[ii]jest tożsame z *(&tab [0]+ii)

```
}

int main(){
        int* pi = f();
        delete pi;
        return 0;
}
```

W tym przykładzie ponownie alokujemy pamięć na stosie, ale tym razem wewnątrz funkcji (od razu inicjalizujemy wartością 7). Funkcja zwraca wskaźnik do zmiennej, która istnieje nawet po zakończeniu działania funkcji.

Uwaga: Pamięć zaalokowaną zawsze należy zwalniać, inaczej prowadzi to do wycieków pamięci i zwolnienia działania komputera!

O ile powyższy przykład jest trywialny i mało użyteczny, o tyle w przypadku typów zdefiniowanych przez użytkownika (klas) wskaźniki stają się bardzo przydatne.

6.2 Tablice

Główną niedogodnością związaną z tworzeniem nowych tablic było to, że z góry musieliśmy znać ich rozmiar. W szczególności nie mogliśmy stworzyć tablicy, której rozmiar byłby zmienną. Rozwiązaniem tego problemu jest **dynamiczna alokacja pamięci**. Popatrzmy na poniższy przykład:

```
int* a = nullptr;   // Pointer do int, inicjalizujemy nullptr.
int n;              // Rozmiar naszej tablicy
cin >> n;           // Wczytaj rozmiar
a = new int[n];  // Zaalokuj nowa pamiec na tablice n intow i zapisz wskaznik do a.
for (int i=0; i<n; i++) {
    a[i] = 0;     // Incijalizuj wszystkie elementy zerami.
}
. . .  // Uzywaj jak zwyklej tablicy
delete [] a;  // Kiedy skonczysz, zwolnij pamiec.
a = nullptr;      // Ustaw na nullptr, zeby zapobiec przypadkowemu ponownemu uzyciu.
```

W powyższym przykładzie odczytano ze standardowego wejścia wielkość tablicy, następnia utworzono taką tablice na stosie i wypełniono zerami. Na koniec zwolniono pamięć wykorzystując operator **delete**[]. Do tablic jest inny operator zwalniania pamięci!

Uwaga: Pamięć zaalokowaną na zmienną zwalniamy operatorem **delete**, a zaalokowaną na tablicę przy użyciu **delete**[]

6.3 Tablice wielowymiarowe

Tablice wielowymiarowe w C++ definiuje się jako tablice wskaźników na wskaźniki. Często wykorzystuje się przy tym dynamiczną alokację pamięci. Np. żeby stworzyć dwuwymiarową tablicę reprezentującą macierz 4 x 4 liczb typu double możemy użyć poniższego kodu:

```
double** v = new double*[4];
for(int row=0; row<4; row++)
{
        double* rowv = new double[4];
        for(int col=0; col<4; col++)
        {
                rowv[col] = 7+4*row+col; //albo cos innego
        }
        v[row] = rowv;
}
```

Wiszimy, że najpierw tworzymy wskaźnik na tablicę wskaźników typu double, następnie każdą z wewnętrznych tablic tworzymy oddzielnie na stosie, wpisujemy odpowiednie wartości i przypisujemy do odpowiednich wskaźników w tablicy obejmującej. Dostęp uzyskujemy przez operator nawiasów kwadratowych, używając go podwójnie, np.

```
std::cout << v[2][3];
```

Oczywiście w powyższy sposób można stworzyć również tablicę o większej liczbie wymiarów.

Trzeba pamiętać, żeby poprawnie zwolnić pamięć po tablicy wielowymiarowej. Zwalnianie zaczynamy od najgłębszego poziomu i przechodzimy stopniowo wyżej. Przykład

```
for(int jj=0; jj<4; jj++)
        {
                //do tablic jest operator delete[] a nie delete !!!
                delete[] v[jj]; //usuwamy wewnetrzne tablice
        }
delete[] v; //usuwamy nadrzedna tablic
```

Gdybyśmy wywołali tylko

```
delete[] v;
```

To usunęlibyśmy tablicę wskaźników na tablice, ale same tablice, które są wskazywane przez te wskaźniki istniałyby dalej. Nie dałoby się ich już usunąć, gdyż wskaźniki do nich zostały usunięte. Pamięć zajęta przez te tablice byłaby zajęta aż do zakończenia działania programu, byłby to tzw. *wyciek pamięci*. Wyobraźmy sobie program, który tworzy duże macierze, np 1e6 na 1e6, i coś z nimi robi, po czym powtarza procedurę dla innych danych tworząc nowe macierze. Jeżeli za każdym razem będzie alokowana nowa pamięć, a stara nie będzie zwalniana, to po pewnym czasie pamięć się po prostu skończy i komputer nam się zawiesi!

7 Klasy i obiekty

Klasy są niezwykle istotnym elementem języka C++ i wielu innych współczesnych języków programowania. Najprościej ujmując, klasa to kontener, w którym możemy pogrupować dane oraz funkcje. Ponadto możemy określić różne reguły dostępu do nich, oraz wykorzystać tzw. dziedziczenie do usprawnienia procesu pisania kodu.

7.1 Definicja

Składnia klasy w C++ jest następująca:

```
class osoba : czlowiek {
    // cialo klasy
};
```

Zaczynamy od słowa kluczowego "class" po którym podajemy nazwę klasy (tutaj "osoba"). Następnie możemy, choć nie musimy, zdefiniować dziedziczenie (o którym później). Robimy to pisząc po identyfikatorze (nazwie) klasy dwukropek a po nim identyfikator klasy po której nasza nowa klasa ma dziedziczyć. W podanym przykładzie klasa "osoba" dziedziczy po klasie "czlowiek". Następnie w nawiasach klamrowych podajemy ciało klasy, podobnie jak robiliśmy to dla funkcji. Po klamrze zamykającej stawiamy średnik. Definicję klasy umieszczamy poza funkcjami.

7.2 Pola

W klasach możemy przechowywać dane. Używamy do tego "pól". Pola są jakby zmiennymi wewnątrz klasy. Definiujemy je tak samo jak zmienne:

```
class Mebel {
    string nazwa;

    int ilosc_nog;
```

```
    float waga;

    int szerokosc;
    int wysokosc;
    int dlugosc;
};
```

W podanym przykładzie zdefiniowalismy klasę mebel, oraz jej pola określające parametry rzeczywistego obiektu (mebla). Deklaracja pola odbywa się poprzez podanie jego typu i nazwy. Nie przypisujemy polom wartości przy definicji, używamy do tego konstruktorów, o czym później. Widzimy, że klasy pozwalają nadać naszym danym logiczną i semantyczną strukturę.

7.3 Metody

Wewnątrz klas możemy definiować funkcje, nazywane "metodami". Deklarujemy je w następujący sposób:

```
class Osoba {
    string imie;
    string nazwisko;

    int wiek;
    int wzrost;

    void WypiszImie();
};
```

W powyższym przykładzie zadeklarowaliśmy jedną metodę o nazwie "WypiszImie". Widzimy, że deklaracja zaczyna się od napisania typu zwracanego, później nazwy funkcji, a na końcu podajemy w nawiasach okrągłych parametry metody (metoda w przykładzie ich nie przyjmuje). Zawsze w klasie musimy zadeklarować metody, możemy ale nie musimy ich tam definiować. Rozpatrzmy kolejny przykład:

```
class Kwadrat
{
        double a;
        double Pole(){return a*a;};
};

class Trojkat
{
        double p;
        double h;
        double Pole();
};

double Trojkat::Pole()
{
        return p * h/2.0;
};
```

W tym przykładzie mamy dwa style definiowania metod. W klasie Kwadrat definiujemy metodę pole bezpośrednio w ciele klasy. Natomiast dla klasy Trojkat jedynie deklarujemy metodę w ciele klasy, ale definiujemy ją poniżej. Do tego celu uzywamy operatora przestrzeni nazw, czyli podwójnego dwukropka. Wyrażenie "double Trojkat::Pole()" mówi, że chcemy zdefiniować funkcję zwracającą double, nie przyjmującej żadnego argumentu, która nazywa się "Pole" i jest zdefiniowana wewnątrz przestrzeni nazw "Trojkat". Przestrzeń nazw jest tworzona automatycznie dla każdej klasy. Przykładem przestrzeni nazw jest "std", w której znajdują się m. in. "cin" oraz "cout". Użycie przestrzeni nazw pozwala wykorzystywać takie same nazwy funkcji/metod i zmiennych/pól bez pomyłki i ograniczeń.

7.4 Sepcyfikatory dostępu

Klasy pozwalają nie tylko grupować dane, ale również zarządzaniem dostępem do nich. Odbywa się to poprzez użycie słów kluczowych "public, protected, provate". Oto krótki opis:

- public – pola/metody są dostępne z każdego miejsca programu, w którym są widoczne, tzn. jeżeli można utworzyć obiekt klasy to ma się do nich dostep.
- protected – pola/metody dostępne są jedynie z poziomu klas pochodnych i zaprzyjaźnionych
- private – pola/metody są dostępne jedynie z poziomu klasy i klas zaprzyjaźnionych.

```
class Mebel {

    string nazwa;

        protected:
    int ilosc_nog;
    float waga;

        public:
    int szerokosc;
    int wysokosc;
    int dlugosc;
};
```

W tym przykładzie pole nazwa będzie "private", ponieważ "private" jest domyślnym specyfikatorem dostępu. Pola "ilosc_nog" oraz "waga" będą "protected", ponieważ są poprzedzone słowem kluczowym "protected" po którym występuje dwukropek. Oznacza to, że wszystkie pola i metody jakie pojawią się po tym specyfikatorze, aż do pojawienia się innego specyfikatora, będą takie jak specyfikator. Następnym specyfikatorem jest "public", który anuluje wcześniejszy specyfikator (tutaj "protected") i sprawia, że pola "szerokosc", "wysokosc" i "dlugosc" będą publicznie dostepne.

Po co używać specyfikatorów dostępu? Uniemożliwiają przypadkową modyfikację składowych klasy. Jeżeli fragment kodu, który nie powinien mieć dostępu do składowych próbuje ten dostęp otrzymać, to kompilator zgłosi błąd.

7.5 Tworzenie obiektów

Obiektem nazywamy instancję klasy. Co przez to rozumieć? Klasa określa nam typ, pola i metody. Obiekt natomiast jest realizacją klasy, pola obiektu zawierają konkretne wartości. Obiekty tworzymy w nastepujący sposób:

```
Mebel meb;
```

Zatem najpierw podajemy typ (nazwę klasy), potem nazwę zmiennej (obiektu).Czasami tworzenie obiektów wygląda trochę inaczej, przykład dla kalsy o mało oryginalnej nazwie "Klasa":

```
Klasa k(2.3, 5.8, "tekst");
```

Dodaliśmy nawiasy okrągłe a w nich wartości. Nawiasy biorą się stąd, że w momencie tworzenia obiektu wołana jest specjalna metoda zwana konstruktorem. Konstruktor może być bezargumentowy, jak w pierwszym przykładzie, lub posiadać argumenty, które podajemy w nawiasach okrągłych tak samo jak dla funkcji.

Obiekty często tworzy się dynamicznie na stosie:

```
Mebel* meb = new Mebel;
meb->dlugosc = 145;
```

W powyższym przykladzie stworzyliśmy obeikt typu Mebel na stosie, oraz zmienną wskaźnikową meb i przypisaliśmy do niej adres nowoutworzonego obiektu. Następnie użyliśmy operatora strzałki "-¿", żeby zmienić wartość publicznego pola w tym obiekcie. Pamięć dynamicznie alokowaną należy zawsze zwalniać z użyciem operatora delete.

7.6 Konstruktory

Szczególnym rodzajem metod są konstruktory. Konstruktory są to metody wywoływane w momencie tworzenia obiektu danego typu. Używamy ich do przygotowania obiektów do użytku, np. przypisując domyślne wartości pól, również z użyciem dynamicznej alokacji pamięci. Konstruktory definiuje się tak jak inne metody, z tym, że nie pisze się ich typu zwracanego (nawet void się nie pisze) oraz muszą nazywać się dokładnie tak jak klasa.

Spójrzmy na poniższy przykład:

```
class Student
{
public:
        char name[100];
        int id;

        //konstruktor domyslny
        Student()
        {
                strcpy(name, "");
                id = -1;
        };
        // konstruktor kopiujacy
        Student(const Student& s)
        {
                strcpy(name, s.name);
                id = s.id;
        };
        //inny konstruktor
        Student(const char* name, int id=0)
        {
                strcpy(this->name, name);
                this->id = id;
        };

};
```

W przykładzie tworzymy klasę o nazwie "Student". Wszystkie składowe klasy są publiczne.

Następnie definiujemy konstruktor. Konstruktor, który nie przyjmuje żadnych argumentów, albo ma argumenty ale wszystkie mają domyślne wartości, nazywamy konstruktorem domyślnym. Każda klasa posiada konstruktor domyślny, nawet jeśli go nie zdefiniujemy. Definiując własny konstruktor domyślny nadpisujemy konstruktor, który zostałby utworzony przez kompilator. W tym konstruktorze przypisujemy domyślne wartości polom klasy.

Następny jest konstruktor kopiujący. Konstruktory kopiujące są to konstruktory wywoływane za każdym razem, gdy kopiowany jest obiekt. Konstruktor kopiujący przyjmuje jeden argument, stałą referencję na obiekt tej samej klasy w której jest konstruktor. Ponieważ kopiowanie odbywa się często, np. przy dodawaniu obiektór do std::vector albo przy przekazywaniu do funkcji, to zazwyczaj chcemy mieć konstruktor kopiujący. Podobnie jak konstruktor domyślny, kompilator również stworzy samodzielnie konstruktor kopiujący jeśli go nie zdefiniujemy. Do prostych klas taki konstruktor jest wystarczający, ale jeżeli przy kopiowaniu chcemy zrobić cokolwiek ponad proste przypisanie do pól wartości pól z obiektu kopiowanego, to musimy napisać własny konstruktor kopiujący. W przykładzie używamy funkcji strcpy do skopiowania łańcucha znaków. Ponieważ łańcuch znaków to tak na pawdę tablica, więc zwykłe przypisanie rodzjau

```
name = s.name;
```

nie zadziała. Podobnie musimy uważać, gdy wśród pól naszej klasy są tablice innych typów. Szczególnie należy pamiętać, gdy alokujemy w klasie pamięć na stosie. Np. często polami klas są wskaźniki na wektory, które tworzymy na stosie. Wtedy konstruktor kopiujący musi utworzyć nowy wektor na stosie i przepisać do niego dane ze starego. Możnaby przepisać tylko wskaźnik, ale zazwyczaj to zły pomysł, bo jeżeli inny obiekt usunie w międzyczasie wektor,

to inne obiekty nie będą tego wiedziały i może nastąpić błąd pamięci. Na koniec zauważmy, że dostęp do pól uzyskujemy przez operator kropki, jak dla $std :: vector$. To dlatego, że wektory też są obiektami.

Ostatnia metoda to również konstruktor, typ razem żadne szczególny tylko wymyślony przez nas. Konstruktor ten ma dwa argumenty, pierwszy to stały wskaźnik na char, musimy użyć tego typu jeżeli chcemy przekazać łańcuch znaków. Drugi argument to int. Zauważmy, że w definicji mamy przypisanie do tego argumentu:

```
int id = 0
```

Taka składnie oznacza, że argument id jest opcjonalny. Możemy stworzyć klasę student na dwa sposoby:

```
Student ania("Anna Grodzka");
Student jacek("Jacek Krawczyk", 4);
```

W obydwu przypadkach zostanie wywołany ten sam konstruktor. W przypadku Anny, wartość pola zostanie ustawiona na 0. Zauważmy, że w tym konstruktorze użyliśmy dziwnego obiektu o nazwie **this**. "this" to specjalne słowo oznaczające wskaźnik na obiekt w którym wywoływana jest instrukcja. Ten wskaźnik jest automatycznie tworzona i przekazywany implicite do wszystkich metod klasy. Pozwala nam odwołać się do jej składowych nawet, gdy nazwy pól i metod zostaną przykryte. Tak się dzieje w naszym przykładzie. Zauważmy, że nasze pola nazywają się name i id, a argumenty konstruktora nazywają się tak samo. Z tego powodu nazwy pól zostaną przykryte przez nazwy argumentów. Jeżeli napiszemy name, to będzie to oznaczać argument a nie pole. Żeby dostać się do pola używamy wskaźnika this: $this- > name$. W tym przypadku użycie wskaźnika this było konieczne. W pozostałych konstruktorach nie było, ale również tam moglibyśmy go użyć.

Definiowanie kilku konstruktorów to przykład przeciążania metod. Konstruktory muszą być publiczne, zebyśmy mogli tworzyć obiekty klasy!

Listy inicjalizacyjne

Bardzo często piszemy konstruktory, których jednym zadaniem jest przepisanie wartości argumentów do odpowiednich pól. Można to zrobić w standardowy sposób, albo z wykorzystaniem specjalnej składni zwanej listą inicjalizacyjną. Wygląda to tak, że po sygnaturze konstruktora a przed nawiasami kwadratowymi piszemy dwukropek, po nim nazwy pól a za nimi w nawaisach kwadratowych nazwy argumentów. Poszczególne pola oddzielamy przecinkami. Przykład:

```
class Punkt
{
        public:
        double x;
        double y;
        double z;
        Punkt(double a=0.0, double b=0.0, double c=0.0) : x(a), y(b), z(c) {}
};
```

W przykładzie korzystamy z listy inicjalizacyjnej. Dodatkowo ustawiamy domyślne wartości pól. Na końcu są puste klamerki, możemy jednak tam coś napisać jeśli chcemy – nie ma wymogu, żeby były puste.

Destruktory

Destruktory są przeciwieństwem konstruktorów, wykonują się przed usunięciem obiektu z pamięci. Używamy ich najczęściej do zwolnienia pamięci na stosie, albo jeżeli chcemy coś zrobić z danymi przed usunięciem obiektu. Destruktory nie mają typu zwracanego, ich nazwa musi być taka sama jak nazwa klasy, ale poprzedzona znakiem tyldy. Rozważmy przykład:

```
class Student
{
public:
        int id;
        std::vector<double>* marks;
        Student(int id=0)
```

```
        {
                this->id = id;
                this->marks = new std::vector<double>;
        }
        void addMark(double m)
        {
                (this->marks)->push_back(m);
        }
        ~Student()
        {
                delete this->marks;
        }

};
```

W przykładzie definiujemy klasę Student, która posiada dwa pola. Pierwsze to id, a drugie to wskaźnik na wektor z ocenami. W konstruktorze ustawiamy id, tworzymy nowy wektor na stosie i przypisujemy jego adres do wskaźnika. Mamy dalej metodę addMark(double), która dodaje liczbę do wektora. Ostatni jest destruktor, w którym zwalniamy pamięc po wektorze. Jeżeli byśmy tego nie zrobili, to po usunięciu obiektu wektor z ocenami nadal istniałby w pamięci aż do zakończenia programu. Nie moglibyśmy w żaden sposób dostać się do tych danych ani ich usunąć. Byłby to tzw. wyciek pamięci, poważny błąd programistyczny, który może skutkować spowolnieniem a nawet zawieszeniem się programu.

Jak się pewnie domyślasz, wszystkie klasy posiadają domyślny destruktor, który nie zwalnia pamięci alokowanej na stosie. Jest to celowe, pamięc alokowana na stosie to pamięć, za którą pełną odpowiedzialność ponosi programista.

7.7 Metody set i get

Rzadko zdaża się, żebyśmy nie chcieli modyfikować danych wewnątrz klasy. Ale żeby zmodyfikować wartości pól czy wywołać metodę np. w funkcji main, składowe są publiczne. Publiczne składowe są wygodne, ale niosą ze sobą duże ryzyko, gdyż publiczne pola mogą być zmodyfikowane w dowolnej części kodu. Może to prowadzić do trudnych do wyłapania błędów. Dlatego przyjęło się używać metod get/set. Ich istnienie nie wynika z samego języka, jest powszechny zwyczaj, jedna z tzw. praktyk dobrego programowania.

Idea jest następująca: deklarujemy pola składowe klasy z ograniczonym dostępem, np. private, i definiujemy publicznie dostępne metody set i get, które kolejno ustawiają i zwracają wartość pola. W ten sposób istnieje tylko jeden sposób na zmodyfikowanie i uzyskanie danych z klasy. Znacznie ułatwia to debugowanie kodu.

Metoda typu get zwraca wartość jednego z wybranych pól. Zatem typ zwracany jest zazwyczaj tożsamy z typem pola. Metody typu get zazwyczaj nie przyjmują argumentów, chociaż nic nie stoi na przeszkodzie by było inaczej. Przykład wykorzystujący klasę Student z poprzedniej sekcji:

```
int getId(){return this->id;}
```

Metoda typu set ustawia wartość pola. Zazwyczaj nie zwraca żadnej wartości i przyjmuje argumenty pozwalające na ustawienie wartości pola, zazwyczaj tylko jeden:

```
void setId(int id){this->id = id;}
```

7.8 Dziedziczenie – podstawy

Wyobraźmy sobie, że piszemy grę cRPG, w której gracz walczy z potworami. Naturalne jest wykorzystać klasy do zaprogramowania potworów. W klasach będziemy mogli pogrupować różne cechy potworów: ilość ich punktów życia, siłę ataku, odporność na czary itp. Niektóre potowry będą miały specjalne ataki, inne nie. Pojawia się tutaj problem specjalizacji, potrzebujemy wielu klas, które są do siebie podobne ale nieidentyczne. Nie chcemy pisać kodu dla każdego potwora z osobna, z drugiej strony, gdybyśmy napisali jedną klasę ze wszystkimi możliwymi opcjami to byłaby nieczytelna i w wielu przypadkach zbędna. Rozwiązaniem jest dziedziczenie.

Idea dziedziczenia jest bardzo prosta: piszemy klasę A definiując jej pola i metody. Następnie definiujemy klasę B, która ma wszystko to co klasa A, oraz kilka innych rzeczy. Zamiast kopiować ręcznie zawartość klasy A do klasy B,

możemy powiedzieć kompilatorowi, że klasa B ma wszystko to co klasa A. Właśnie to nazywamy dziedziczeniem. Klasę A nazywamy *klasą bazową*, a klasę B *klasą pochodną*.

Składniowo, wystarczy definiując klasę B, po jej nazwie napisać dwukropek, dalej specyfikator dostępu i nazwę klasy z której ma dziedziczyć. Rozważmy przykład:

```
class Figura
{
public:
        double pole;
        double obwod;
        int liczbaKatow;
};

class Kwadrat : public Figura
{
public:
        double a;
        Kwadrat()
        {
                this->liczbaKatow = 4;
        }
        Kwadrat(double a)
        {
                this->liczbaKatow = 4;
                this->a = a;
                this->obwod = 4*a;
                this->pole = a*a;
        }
};
```

W przykładzie zdefiniowaliśmy klasę Figura oraz dziedziczącą po niej klasę Kwadrat. Widzimy, że Kwadrat ma dostęp do pól, które nie zostały w nim zdefiniowane, ale zdefiniowane w klasie z której dziedziczy.

Użyliśmy tutaj specyfikatora dostępu "public". Rodzaj użytego specyfikatora determinuje jakie będą specyfikatory dostępu pól w klasie pochodnej.

- public – jeżeli dziedziczenie jest publiczne, to publiczne składowe klasy bazowej stają się publicznymi składowymi klasy pochodnej, podobnie składowe protected pozostają protected. Składowe private nie są dziedziczone.
- protected – zarówno składowe publiczne jak i protected z klasy bazowej stają się wszystkie protected w klasie pochodnej. Składowe private nie są dziedziczone.
- private– zarówno składowe publiczne jak i protected z klasy bazowej stają się wszystkie private w klasie pochodnej. Składowe private klasy bazowej nie są dziedziczone.

Pamiętajmy, że możliwe jest nadpisywanie dziedziczonych składowych. Jeżeli w klasie Kwadrat zdefiniowalibyśmy pole/metodę o tej samej sygnaturze co składowa odziedziczona z klasy bazowej, to byśmy ją przesłonili. Na szczęście można wciąż się do niej dostać używając odpowiedniej przestrzeni nazw, np.

```
Figura::liczbaKatow = 4;
```

7.9 Konstruktory w klasach pochodnych

Wiemy już jak pisać klasy pochodne, ale należy powiedzieć trochę więcej o konstruktorach w klasach pochodnych. Rozważmy przykład:

```
#include <iostream>
class A
{
public:
```

```
        double poleA;
        A(){poleA = 0;}
        A(double a) : poleA(a)
        {
                std::cout << "Konstruktor A" << std::endl;
        }
};

class B : public A
{
public:
        double poleB;
        B()
        {
                std::cout << "Konstruktor B" << std::endl;
        }
};

int main()
{
        B litera;
        return 0;
}
```

Okazuje się, że jeżeli skompilujemy i uruchomimy powyższy program, to zostanie wyświetlone:

Konstruktor A// Konstruktor B

Zatem konstruktor klasy bazowej jest wywoływany **implicite** w konstruktorze klasy pochodnej. Co więcej, jest wywoływany przed wywołaniem konstruktora klasy pochodnej (co jest całkiem logiczne). Pytanie jakie się nasuwa jest następujące: Który konstruktor klasy A się wywoła? Jeżeli nie sprecyzujemy to będzie to konstruktor domyślny.

Warto o tym pamiętać. Szczególnie jeżeli chcemy używać list inicjalizacyjnych w klasach pochodnych. Jeżeli chcielibyśmy napisać nowy konstruktor dla klasy B, taki, który przyjmie dwie wartości typu double do ustawienia pól poleA i poleB, to musimy to zrobić w nastepujący sposób:

```
#include <iostream>
class A
{
public:
        double poleA;
        A(){poleA = 0;}
        A(double a) : poleA(a)
        {
                std::cout << "Konstruktor A" << std::endl;
        }
};

class B : public A
{
public:
        double poleB;
        B()
        {
                std::cout << "Konstruktor B" << std::endl;
        }
        B(double a, double b) : A(a), poleB(b){
        std::cout << "Konstruktor B" << std::endl;}
};
```

```
int main()
{
        B litera(1.5, 4.2);
        return 0;
}
```

A więc w liście inicjalizacyjnej wywołujemy odpowiedni konstruktor klasy bazowej.

7.10 Klasy abstrakcyjne

Klasy są użyteczne, ponieważ pozwalają odzwierciedlić struktury istniejące w rzeczywistości. Za pomocą dziedziczenia możemy uwzględnić również relacje pomiędzy obiektami. Dziedziczenie stosuje się zazwyczaj, gdy chcemy rozszerzyć możliwości klasy, albo stworzyć hierarchię/specjalizować.

Przykładem rozszerzenia możliwości może być klasa Pracownik, reprezentująca ogólne informacje o pracowniku w jakiejść firmie. Możemy się spodziewać, że będzie zawierać informacje osobowe, dział, staż pracy itd. Możemy rozszerzyć pracownika pisząc klasę Kierownik, która będzie dziedziczyć po pracowniku, a ponadto będzie posiadać dodatkowe pola, np. wektor podwładnych Pracowników, oraz metody, np. metodę do dawania podwyżki.

Przykładem budowania hierarchii mogą być klasy reprezentujące cząstki elementarne w Modelu Standardowym. Możemy zacząć od ogólnej klasy Particle, dziedziczących po niej klas Fermion i Boson, dalej klas Scalar, Vector, Lepton itd. W ten sposób oddamy w programie fizyczną klasyfikację cząstek.

O ile w pierwszym przykładzie klasa Pracownik może reprezentować rzeczywiste obiekty, czyli osoby zatrudnione na szeregowych stanowiskach, o tyle w drugi przykładzie klasa Boson nie reprezentuje żadnego fizycznego obiektu. Bozonem nazywamy cząstkę o spinie całkowitym. Jeżeli stworzymy klasy dziedziczące po Boson, wyczerpujące wszystkie interesujące nas możliwości, np. klasy Scalar(spin=0), Vector(spin=1), Graviton(spin=2) ... to tworzenie "czystego" obiektu typu Boson nie ma sensu. Język C++ pozwala określić klasę jako **abstrakcyjną**, tzn. taką, która służy wyłącznie do tego by z niej dziedziczyć. Nie można utworzyć obiektu klasy abstrakcyjnej.

Klasę abstrakcyjną pisze się tak samo jak każdą inną, z tym, że należy w szczególny sposób zdefiniować jej destruktor. Należy zrobić z niego "metodę całkowicie statyczną". Dokładne znaczenie tego terminu zostanie wyjaśnione później. Na ten moment wystarczy wiedzieć jak to zrobić. Przykład:

```
class A
{
public:
        double poleA;
        A() : poleA(0){}
        A(double a) : poleA(a) {}
        virtual ~A() = 0;
};

A::~A(){}

class B : public A
{
        // jakis kod
}
```

W przykładzie stworzyliśmy całkiem zwyczajną klasę A, ale jej destruktor jest szczególny. Po pierwsze, przed destruktorem wstawiliśmy słowo kluczowe "virtual". Po drugie "przypisaliśmy" destruktor do 0. Język wymaga, żeby taki destruktor posiadał swoje ciało. Dlatego poza klasą A zdefiniowaliśmy go jako pustą metodę. Tak napisana klasa jest już abstrakcyjna, nie można utworzyć jej obiektu, ale można użyć jej do napisania klasy B, która będzie dziedziczyć po A.

7.11 Funkcje zaprzyjaźnione

Czasami chcemy, żeby jakaś globalna funkcja miała dostęp do prywatnych pól klasy, którą właśnie definiujemy. Możemy to osiągnąć stosując słowo kluczowe "friend". W tym celu w ciele klasy piszemy pełną sygnaturę funkcji poprzedzoną słowem kluczowym "friend", np.:

```
class Kwadrat
{
private:
        int a;
public:
        Kwadrat(int x=0) : a(x){}
 friend int pole(Kwadrat &k);

};

int pole(Kwadrat &k){return k.a*k.a;}
```

Chociaż funkcja pole nie jest metodą klasy Kwadrat, jest całkowicie niezależną funkcją globalną, to jednak ma dostęp do składowej a klasy Kwadrat.

7.12 Składowe statyczne

Składowe klasy możemy zadeklarować jako statyczne, stawiając przed nimi na samym początku słowo kluczowe "static". Osobno omówimy pola i metody, gdyż mają trochę inne zastosowanie.

Pola statyczne są wspólne dla wszystkich obiektów klasy, to znaczy, z każdego obiektu mamy do nich dostęp i jeżeli w jednym z obiektów zmienimy wartość pola statycznego to wszystkie obiekty danej klasy będą widzieć tą zmianę. Przydaje się to w niektórych sytuacjach, np. do liczenia ile obiektów danej klasy zostało utworzonych. W tym celu tworzymy pole statyczne typu int, nazwyamy np. counter, i w konstruktirze zwiększamy jego wartość o jeden. Spójrz na poniższy przykład:

```
class Klasa
{
public:
        static int counter;
        Klasa(){counter++;}
        ~Klasa(){counter--;}

};

int Klasa::counter = 0;
```

W powyższym przykładzie zadeklarowaliśmy statyczne pole typu int o nazwie "counter". Widzimy, że słowo kluczowe "static" należy umieścić przed nazwą typu. Następnie w konstruktorze zwiększamy wartość pola o jeden, a w destruktorze zmniejszamy o jeden. W ten sposób zmienna counter będzie przechowywać informację o liczbie aktywnych obiektów w programie. Gdybyśmy nie zmniejszali jej wartości w destruktorze, to pokazywałaby ile obiektów w sumie utworzono. Pola statyczne są domyślnie inicjalizowane zerem w momencie utworzenia pierwszego obiektu danej klasy. Można jednak ręcznie je zainicjalizować, odbywa się to w następujacy sposób:

```
int Klasa::counter = 0;
```

Musimy podać przestrzeń nazw w której istnieje pole (czyli po prostu nazwę klasy i podwójny dwukropek.

Słowo kluczowe static może również znaleźć się przed typem zwracanym metody. Wtedy taką metodę można wywołać bez tworzenia obiektu klasy, bezpośrednio z użyciem operatora ::. Np. rozbudujmy wcześniejszy przykład:

```
class Klasa
{
public:
```

```
        static int counter;
        static bool isSomething()
        {
                if(counter > 0)
                        return true;
                else
                        return false:
        }
        Klasa(){counter++;}
        ~Klasa(){counter--;}

};

int Klasa::counter = 0;
```

Zdefiniowaliśmy metodę statyczną *isSomething*(), która zwraca true jeżeli istnieje jakiś aktywny obiekt klasy Klasa, lub false jeżeli żaden nie istnieje. Co ważne, taką metodę możemy wywołać w oderwaniu od konkretnego obiektu, tzn. pisząc w kodzie (np. w main):

```
Klasa::isSomething()
```

Gdybyśmy nie mogli tego zrobić, to metoda nie miałaby sensu. Musilibyśmy utworzyć obiekt typu Klasa i poprzez operator kropki/strzałki wywołać metodę.

Z racji swoich właściwości metody statyczne nie mają dostępu do niestatycznych metod i pól klasy, ponieważ te pola (i pośrednio mające do nich metody) mogą mieć różne wartości w różnych obiektach.

7.13 Polimorfizm i metody wirtualne

Polimorfizm to ogólna nazwa na mechanizmy pozwalające używać zmiennych, wartości i podprogramów na różne sposoby. Pozwala na uwolnienie się od definiowania osobnego kodu dla każdego z typów. Przykładem polimorfizmu już poznanym jest przeciążanie funkcji/operatorów.

Polimorfizm w klasach opiera się na dziedziczeniu i metodach wirtualnych. Bardzo często używamy dziedziczenia, żeby specjalizować obiekty, np. klasa opisująca cząstkę skalarną dziedziczy po klasie Bozon, podobnie jak klasa opisująca cząstkę wektorową itd. W tym przykładzie tworzymy obiekty klas Skalar i Wektor, a więc dwóch różnych typów. Chcielibyśmy móc traktować je wspólnie, jako specjalizacje tej samej klasy bazowej Bozon. Okazuje się, że da się to zrobić, pod warunkiem, że zadeklarujemy naszą klasę bazową jako polimorficzną. W praktyce odbywa się to poprzez zrobienie przynajmniej jednej z metod w klasie bazowej wirtualną. Odbywa się to poprzez dodanie słowa kluczowego "virtual" przed sygnaturą metody.

Metody wirtualne różnią się od zwykłych tym, że mogą rozpoznawać typ obiektu na rzecz którego mają być wywołane. Wyobraźmy sobie, że mamy klasę bazową Bozon i klasy pochodne Skalar i Wektor. W klasie Bozon definiujemy funkcję o nazwie foo. W klasach pochodnych definiujemy funkcje o identycznej nazwie i sygnaturze, co powoduje przesłonięcie metody foo odziedziczonej z Bozon. Jeżeli zrobimy tą metodę wirtualną, przez dopisanie słowa "virtual" w klasie Bozon, to będziemy mogli korzystać z niej w sposób pozwalający zapomnieć czy dany obiekt jest klasy Skalar czy Wektor.

Opisany sposób jest następujący:

1. Tworzymy wskaźnik na typ bazowy, u nas Bozon.
2. Do tego wskaźnika można przypisać adres obiektu typu Bozon, trywialne.
3. ALE można do niego całkiem poprawnie przypisać adres obiektu typu Skalar lub Wektor.
4. Kiedy wywołamy przez wskaźnik metodę foo(), to wybór wersji tej metody, czy ma być z klasy Bozon czy Wektor czy Skalar, dokona się automatycznie poprzez mechanizm metod wirtualnych.
5. Dzięki temu, możemy pisać kod, w którym operujemy na wskaźnikach klas bazowych, a który działa w pożądany sposób dla typów pochodnych.

Właśnie traktowanie obiektów różnego typu wskazywanych przez wskaźnik tak samo – jako specjalizacji klasy bazowej – jest polimorfizmem. Otwiera to zupełnie nowe możliwości w C++, gdyż wskaźników możemy używać w kolekcjach (np. *std :: vector*), w argumentach funkcji, jako pól w klasach, jak typu zwracanego przez funkcje itd. Używając polimorfizmu możemy napisać kod działający poprawnie dla wszystkich/wybranych klas pochodnych i tylko dla nich. Możemy nawet napisać program, w którym typ tworzonej klasey będzie zależał od interakcji z użytkownikiem.

Poniżej implementacja z innym przykładem:

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

class base {
public:
        virtual void print()
        {
                cout << "print base class" << endl;
        }

        void show()
        {
                cout << "show base class" << endl;
        }
};

class derived : public base {
public:
        void print()
        {
                cout << "print derived class" << endl;
        }

        void show()
        {
                cout << "show derived class" << endl;
        }
};

int main()
{
        base* bptr;
        derived d;
        bptr = &d;

        // virtual function, binded at runtime
        bptr->print();

        // Non-virtual function, binded at compile time
        bptr->show();
}
```

W przykładzie zdefiniowaliśmy dwie klasy: bazową i pochodną, a w nich dwie metody: print() i show(). Pierwsza z nich jest wirtualna, druga nie. W funkcji main tworzymy wskaźnik na typ bazowy i przypisujemy do niego adres obiektu klasy pochodnej, następnie wywołujemy obie funkcje. Outputem jest:

print derived class

show base class

Widzimy zatem, że wirtualna funkcja wie jaki jest prawdziwy typ obiektu we wskaźniku (który jest ustalany w trakcie wykonywania programu) i wywoływana jest właściwa wersja. Natomiast dla zwykłej metody to co zostanie wywołane zależy od typu wskaźnika, który znany jest już etapie kompilacji.

8 Przeciążanie operatorów

Operatorami są symbole (nie słowa) mające specjalne znaczenie w języku C++. Przykładem operatora jest "+" służący do dodawania dwóch liczb typu int. Kiedy piszemy 2+2 to tak na prawdę wywoływana jest specjalna funkcja podobna do

```
int operator+(int a, int b)
```

Tak samo jak zwykłe funkcje, możemy przeciążać operatory, dzięki czemu możliwe staje się ich używanie dla zdefiniowanych przez nas typów (klas).

Operatory dzielimy na dwie zasadnicze grupy: operatory jedno- i dwuargumentowe. Dodawanie jest przykładem operatora dwuargumentowego. Operatorem jednoargumentowym jest np. operator dereferencji (*). Operatory mogą istnieć w dwóch wersjach jednocześnie, np. operator "-" może być zarówno dwuargumentowy i reprezentować odejmowanie, jak i jednoelementowy i reprezentować zmianę liczby na przeciwną.

Rozważmy klasę Complex reprezentującą liczbę zespoloną.

```
class Complex
{
private:
        double x;
        double y;
public:
        void setX(double a){x=a;}
        void setY(double a){y=a;}
        double getX() const {return x;}
        double getY() const {return y;}
        Complex(double a=0, double b=0) : x(a),y(b){}
        friend std::ostream& operator<<(std::ostream &out, const Complex& c);
        Complex operator+(const Complex& z) const
        {
                return Complex(x+z.getX(), y+z.getY());
        }
};

std::ostream& operator<<(std::ostream &os, const Complex& c)
{
        os << c.getX() << " +i(" << c.getY() << ")";
        return os;
};
```

Widzimy dwie interesujące rzeczy w implementacji tej klasy. Pierwszą jest linia

```
friend std::ostream& operator<<(std::ostream &out, const Complex& c);
```

Jest to nic innego jak deklaracja operatora strumieniowego, wyjściowego, takiego samego jak używamy do wypisywania rzeczy na std::cout. Zwróćmy uwagę, że jest to operator dwuargumentowy. Typem pierwszego argumentu oraz typem zwracanym jest std::ostream. Słowo kluczowe "friend" oznacza, że ten operator zdefiniowany jako funkcja globalna, będzie miał dostęp do prywantych składowych klasy Complex.

Drugą interesującą rzeczą jest linijka

```
Complex operator+(const Complex& z) const
{
        return Complex(x+z.getX(), y+z.getY());
}
```

W tej linijce zdefiniowaliśmy operator "+". Jest to inna forma definicji, tym razem operator nie jest funkcją globalną, lecz metodą składową klasy, co automatycznie zapewnia dostęp do prywatnych składowych. Operator "+" w przykładzie jest operatorem dwuargumentowym. Pierwszy argument (ten po lewj stronie dodawania) jest przekazywany **implicite** do metody poprzez znany nam już wskaźnik this. Jeżeli nie ma przesłonięcia nazw, to możemy używać identyfikatorów pól z pominięciem wskaźnika this, tak jak robimy to w przykładzie. Na koniec zwrócmy jeszcze uwagę na słowo "const" stojące na końcu sygnatury metody. Postawienie słowa "const" w tym miejscu oznacza, że metoda nie może zmienić wartości pól obiektu na rzecz którego zostanie wywołana (tego wskazywanego przez this). Użycie słowa "const" przy podawaniu typu argumentu zapobiega modyfikacji drugiego z argumentów.

Przeciążanie operatorów pozwala znacząco uprościć i przyspieszyć korzystanie ze zdefiniowanych przez nas obiektów. Należy jednak zachować ostrożność i przy definiowaniu operatorów kierować się zawsze zasadą, że wynik i sposób działania operatora dla konkretnego typu powinien być oczywisty dla użytkownika nie znającego implementacji. O ile dodawanie dwóch liczb zespolonych jest jasne, o tyle porównywanie ich za pomocą operatora ¿ już nie.

8.1 Funktory

Specjalnym operatorem jest operator wywołania (), który może być przeciążony jedynie jako metoda. Operator ten może przyjmować dowolną liczbę argumentów, podanych w nawiasach okrągłych. Składnia:

```
TYP operator()(TYP1 ARG1, TYP2 ARG2, ...)
```

Użycie tego operatora odbywa się poprzez podanie identyfikatora zmiennej po którym stoją nawiasy okrągłe, np. dla klasy Klasa z operatorem wywołania przyjmującym jedną liczbę typu double wywołanie wygląda nastepująco:

```
Klasa v;
v(3.14);
```

Ponieważ takie wywołanie pozwala traktować obiekt jak funkcję, obiekty ze zdefiniowanym operatorem wywołania nazywamy *funktorami*.

9 Kompilowanie kilku plików

Tylko najprostsze programy piszemy w pojedynczym pliku. Jeżeli pliki są zbyt duże, to debugowanie i rozwój programu stają się bardzo trudne[4]. Dlatego pisząc większe programy rozbija je się na kilka plików.

W C++ są dwa rodzaje plików, pliki zwykłe (rozszerzenie .cpp, .cxx) oraz pliki nagłówkowe (.h, .hpp). Zazwyczaj jest tak, że w plikach nagłówkowych piszemy klasy, deklaracje funkcji/metod, stałe globalne, natomiast w plikach .cpp piszemy definicje funkcji/metod. Rozważmy jako przykład klasę Wektor, która ma reprezentować wektor w 3D. Wówczas plik nagłówkowy mógłby wyglądać tak:

```
#ifndef WEKTOR_HPP
#def WEKTOR_HPP

class Wektor
{
        double x;
        double y;
        double z;
        double mod;
public:
        Wektor(): x(0), y(0), z(0), mod(0){}
```

[4]Fanatycy Fortran, niech przepadnie, krzywdzą ludzkość pisząc programy mające tysiące linii kodu w pojedynczym pliku.

```
        Wektor(double a, double b, double c);
        void printWektor();
};

#endif
```

Zaczęliśmy od dyrektyw preprocesora, które mają zagwarantować, że dany plik nagłówkowy zostanie dołączony do programu tylko raz. Następnie mamy klasę Wektor z jej polami i metodami. Widzimy, że konstruktor domyślny jest w niej zdefiniowany, a pozostałe metody jedynie zadeklarowane. Ma to sens, konstruktor domyślny jest trywialny i jego kod mieści się w jednej linijce. Pozostałe metody umieścimy w osobnym pliku .cpp.

Tworzymy zatem plik, np. wektor.cpp, i definiujemy tam brakujące metody:

```
#include <cmath>
#include <iostream>
#include "wektor.hpp"

Wektor::Wektor(double a, double b, double c)
{
        x=a;
        y=b;
        z=c;
        mod = sqrt(a*a+b*b+c*c);
}

void Wektor::printWektor()
{
        std::cout << x << " " << y << " "
                << z << " " << mod << std::endl;
}
```

Nasz plik w tym przypadku nie zawiera funkcji main. Funkcja main znajdzie się jeszcze w innym pliku. wektor.cpp zaczyna się standardowo od dyrekty #include. Ostatnia z nich ma za zadanie wczytać napisany przez nas plik nagłówkowy. Przy wczytywaniu własnych plików nagłówkowych stosujemy cudzysłów i wpisujemy pełną nazwę z rozszerzeniem. Następnie mamy definicje metod. Aby kompilator wiedział, że chodzi nam o metody wewnątrz klasy wektor, a nie globalne funkcje, używamy operatora przestrzeni nazw "::".

Jeżeli deklaracja metody wygląda tak:

```
class KLASA
{
        public:
        TYP METODA(T1 ARG1, T2 ARG2);
}
```

To jej definicja poza klasą (w tym samym pliku co klasa lub w innym) wygląda

```
TYP KLASA::METODA(T1 ARG1, T2 ARG2)
{
        //...
}
```

Zatem widzimy, że identyfikator (nazwę) klasy musimy napisać po typie zwracanym, a przed nazwą metody. Między nazwą klasy a nazwą metody stawiamy dwa dwukropki "::".

Ok, napisaliśmy nasz program i mamy następujące pliki: main.cpp wektor.cpp wektor.hpp. Skompilujemy je w dwóch krokach:

1. Kompilujemy ze wszystkimi flagami, przed plikami źródłowymi stawiamy flagę "-c", następnie wymieniamy nazwy wszystkich plików, zaczynając od pliku z funkcją main:

```
g++ -Wall -std=c++17 -c main.cpp wektor.cpp wektor.hpp
```

Skutkiem tej komendy będzie utworzenie 2 plików obiektowych: main.o oraz wektor.o.

2. Linkujemy pliki obiektowe do pojedynczego pliku wykonywalnego. Wołamy kompilator z flagą "-o", po której podajemy nazwę pliku wykonywalnego jaki zostanie utworzony (w Linuksie takie pliki zwyczajowo nie mają rozszerzenia) a następnie wszystkie pliki obiektowe.

```
g++ -o program main.o wektor.o
```

Skutkiem działania będzie plik wykonywalny o nazwie "program".

10 Biblioteka standardowa

Biblioteka standardowa to zestaw funkcji i klas, która jest częścią standardu C++ i zawiera wiele użytecznych narzędzi. Poniżej omówimy tylko kilka z nich.

Liczby pseudolosowe

Tradycyjne komputery nie potrafią generować prawdziwie losowych liczb. Potrafią natomiast generować rekurencyjne ciągi liczb, które wyglądają losowo, chociaż są w pełni zdeterminowane przez rodzaj ciągu i element początkowy tzw. ziarno. Jeżeli uda nam się dobrać ziarno w sposób przypadkowy, to ciąg liczb pseudolosowych będzie z dobrym przybliżeniem losowy. Funkcje i klasy, których potrzebujemy są zdefiniowane w pliku nagłówkowym *¡random¿*, przyda się również plik nagłówkowy *¡chrono¿*. Aby wygenerować liczby pseudolosowe należy:

1. Załączyć pliki nagłówkowe $< random >$ i $< chrono >$
2. Stworzyć zmienną typu unsigned, która będzie zawierać ziarno (ang. seed).
3. Wpisujemy do ziarna wynik działania funkcji metody z nagłówka $< chrono >$, która zwraca pewną liczbę, będącą czasem jaki upłynął od 1. stycznia 1970 roku. Ponieważ zegar jest bardzo dokładny, za każdym wywołaniem ziarno będzie inne i program wygeneruje inne liczby. Wywołanie funkcji:

```
std::chrono::steady_clock::now().time_since_epoch().count()
```

4. Mówimy programowi, której zmiennej ma użyć jako ziarna do generowania liczb pseudolosowych. Odbywa się to poprzez stworzenie obiektu o nazwie e:

```
std::default_random_engine e (seed);
```

Od tej pory będziemy używać obiektu e typu $std :: default\_random\_engine$ do generowania liczb pseudolosowych.

5. Tworzymy obiekt związany z rozkładem, który nas interesuje. Np. dla rozkładu normalnego o średniej 5 i odchyleniu 2:

```
std::normal_distribution<double> distN(5,2);
```

Stworzyliśmy obiekt o nazwie distN. Teraz możemy użyć go wraz z obiektem e tak, jakbyśmy wołali funkcję:

```
double x = distN(e);
```

Powyższy kod generuje liczbę pseudolosową z rozkładu normalnego o średniej 5 i odchyleniu standardowym 2, a następnie zapisuje ją do zmiennej x. Jeżeli chcemy wygenerować kolejną liczbę, to piszemy distN(e) ponownie.

Pełny program dla rozkładu normalnego:

```
#include <iostream>
#include <random>
#include <chrono>
```

```
int main()
{
        unsigned seed = std::chrono::steady_clock::now().time_since_epoch().count();
        std::default_random_engine e (seed);
        std::normal_distribution<double> distN(5, 2);
        std::cout << distN(e) <<  endl;
        return 0;
}
```

Inne rozkłady niż normalny generuje się z użyciem odpowiednich obiektów. Pełna lista dostepna jest na `https://en.cppreference.com/w/cpp/numeric/random`
Wybrane rozkłady:

- Rozkład normalny o średniej a i odchyleniu standardowym b:

```
std::normal_distribution<double> dist(a, b);
```

- Rozkład jednorodny na przedziale [a, b):

```
std::uniform_real_distribution<> dist(a, b);
```

- Rozkład jednorodny na przedziale [a, b] dla liczb całkowitych:

```
std::uniform_int_distribution<> dist(a, b);
```

WAŻNE: Istnieje więcej metod generowania liczb pseudolosowych w C++ (inne biblioteki). Powyższa procedura jest skomplikowana, ale jest to standardowy sposób generowania liczb pseudolosowych w standardzie C++11, który aktualnie jest dominujący.

Kolekcje

Kolekcje to kontenery do przechowywania danych, posiadające wiele użytecznych narzędzi.

10.0.1 Vector

Najczęściej używaną kolekcją jest *vector* zdefiniowany w pliku nagłówkowym $< vector >$. Można myśleć o vectorze jak o tablicy, która może mieć zmienny rozmiar, tzn. takiej do której możemy dodawać i usuwać elementy. Przykład:

```
#include <iostream>
#include <vector>

int main()
{
    // Create a vector containing integers
    std::vector<int> v = {7, 5, 16, 8};

    // Add two more integers to vector
    v.push_back(25);
    v.push_back(13);

    // Iterate and print values of vector
        for(unsigned i = 0; i < v.size(); i++)
        {
                std::cout << v[i] << " ";
        }
}
```

W powyższym przykładzie tworzony jest vector dla zmiennych typu int – typ podajemy w < ... >. W przykładzie vector jest od razu inicjalizowany 4 liczbami. Następnie używana jest metoda **push_back()** (vector jest klasą, dlatego używamy składni: zmienna.funkcja(arg), o klasach będziemy się jeszcze uczyć). Aby wypisać elementy używamy pętli for, z tym że vector w przeciwienstwie do zwykłej tablicy ma informację o swoim rozmiarze poprzez metodę **size()**, której używamy jako górnego limitu dla pętli. Dostęp do danych mamy poprzez operator [], tak samo jak dla tablic.

Czasem przydaje się również metoda **data()**, która pozwala uzyskać dane z vectora w postaci zwykłej tablicy.

Nic nie stoi na przeszkodzie, aby typem vectora był vector, np.

```
std::vector< std::vector<int> > v;
```

w ten sposób możemy tworzyć kolekcję wielowymiarowe

Inną użyteczną cechą wektorów jest możliwość iterowania po ich elementach za pomocą wskaźników. Używa się do tego metod *begin()* oraz **end()**, które zwracają odpowiednio wskaźniki na pierwszy element wektora oraz wskaźnik "za" ostatnim elementem. Praktyczny przykład wypisania elementów wektora w ten sposób:

```
std::vector<int> v;

//robimy cos z wektorem, np. uzupelniamy

for ( std::vector<int>::iterator iter = v.begin(); iter != v.end(); iter++)
{
        //operator wyluskania bo iter jest wskaznikiem!
        std::cout << *iter << " ";
}
```

10.0.2 Set

std :: *set* jest zdefiniowany w pliku < *set* > i jest posortowaną kolekcją zawierającą unikalne elementy danego typu. Set tworzymy w następujący sposób:

```
std::set<T> s;
```

gdzie T typem elementów kolekcji, tak samo jak dla wektora mamy $std :: vector < T >$.

Nowe elementy dodaje się za pomocą metody *insert*:

```
std::set<double> s;
s.insert(3.1415);
```

Można też utworzyć set z istniejącej kolekcji za pomocą iteratorów, np. z wykorzystaniem istniejącego wektora:

```
std::vector<double> v = {3., 12.3, 6.7, 0.0};
std::set<double> s(v.begin(), v.end());
```

Widzimy, że odbywa się to poprzez podanie iteratora początkowego i końcowego. Taka konstrukcja umożliwia konwersję tylko pożądanego fragmentu wektora na *std* :: *set*.

Najważniejszą właściwością *std* :: *set* jest to, że jego elementy są unikalne, tzn. jeśli dodamy 5 razy taki sam element, to *std* :: *set* będzie przechowywał tylko jedną jego kopię. Jest to przydatne, gdy chcemy szybko pozbyć się powtórzeń z danych. Np. jeśli mamy dane w wektorze i chcemy usunąć powtórzenia to po prostu tworzymy *std* :: *set* z tego wektora.

Po kolekcji *std* :: *set* można iterować z pomocą metod begin() oraz end(), np. wypisanie zawartości kolekcji s realizuje się tak:

```
for(auto it = s.begin(); it != s.end(); ++it)
        std::cout << *it << " ";
```

10.0.3 Map

Kolejną ważną kolekcją jest $std :: map$, zdefiniowana w pliku $< map >$. Ta kolekcja umożliwia nam łączenie danych w pary klucz-wartość, gdzie klucz zazwyczaj dobiera się tak, żeby ułatwić dostęp do danych. Mapę definiuje się następująco:

```
std::map< T1, T2> m;
```

gdzie T1 jest typem klucza, a T2 typem wartości. Dane możemy dodać do mapy stosując następującą składnię:

```
m[KLUCZ] = WARTOSC;
```

Rozjaśni się to, jeżeli spojrzymy na przykład, w którym przypisujemy klucze-numery poszczególnym dniom tygodnia:

```
std::map<int, std::string> m;
m[1] = std::string("poniedzialek");
m[2] = std::string("wtorek");
m[3] = std::string("sroda");
```

W ten sposób możemy np. wykonywać operacje na liczbach by policzyć jaki dzień tygodnia będzie za 3 lata i 45 dni, a następnie użyć wyniku, żeby dostać się do napisu z nazwą dnia tygodnia. Np. jeżeli nasze obliczenia dadzą wynik 4, to wyświetlimy nazwę dnia tygodnia poprzez:

```
std::cout << m[4] << std::endl;
```

Po mapie możemy iterować w podobny sposób jak po wektorze i zbiorze, ale tym razem iterator pokazuje na parę klucz-wartość, więc musimy użyć pól "first" oraz "second", żeby dostać się do odpowiednio klucza i wartości, np. jeżeli chcemy wypisać wszystkie pary klucz-wartość w nowych liniach to piszemy

```
for(auto it=m.begin(); it!=m.end(); ++it)
        std::cout << (it->first) << " " << (it->second) << std::endl;
```

Nic nie stoi na przeszkodzie, aby wartości były kolekcjami. Możemy np. przechowywać w wektorach kolejne serie danych, wymyślić dla nich opisowe klucze i stworzyć odpowiednią mapę. Taka mapa mogłaby być typu $std :: map < std :: string, std :: vector < double >>$. Gdybyśmy chcieli wypisać wszystkie pary klucz-wartość to musilibyśmy uzyć zagnieżdżonych pętli for,np.:

```
//petla po mapie
for(auto it = dane.begin(); it!=dane.end(); ++it)
        {
                std::cout << it ->first << " : ";
                //petla po kolekcji bedacej wartoscia
                for(auto it2 = (it->second).begin(); it2!=(it->second).end(); ++it2)
                {
                        std::cout << *it2 << " ";
                }
                std::cout << std::endl;
        }
```

11 Inne

Słowo kluczowe auto

Rozważmy następujący przykład

```
char znak = 'f';
```

Mamy tu definicję zmiennej typu char z przypisaniem wartości. Zauważmy, że bez trudu moglibyśmy odgadnąć typ zmiennej na podstawie tego, co jest do niej przypisywane. Zatem informacja o typie zmiennej jest w tym przypadku zbędna. Skoro my mogliśmy odgadnąć typ zmiennej to tym bardziej może to zrobić kompilator. W tym celu stworzono słowo kluczowe **auto**. Używamy go zamiast nazwy typu i jest to informacja dla kompilatora, że powinien sam zgadnąć co to za typ. Przykład:

```
auto znak = 'f';
```

Typ każdej zmiennej musi być znany na etapie kompilacji. Jeżeli kompilator nie będzie mógł wydedukować typu to zgłosi błąd. W praktyce nie warto używać auto dla prostych typów jak w przykładzie powyżej. Opłaca się to stosować do nazw typów, których nie pamiętamy, albo nie chcemy pisać ze względu na ich długość. Np.

```
auto v = new std::vector<double>;
auto f = cos;
```

W przykładzie użyliśmy auto dla typu $std :: vector < double > *$ oraz dla typu $double(*f)(double)$. "cos" to oczywiście nazwa funkcji z pliku nagłówkowego cmath.