

Wykład 1: Powtórzeniowy

Wskaźniki i zmienne dynamiczne
Dynamiczne alokowanie struktur



Wskaźniki

dla przypomnienia



Wskaźnik na zmienną danego typu to zmienna, która przechowuje adres zmiennej danego typu.

- ✓ Zmienne statyczne są niczym innym jak tylko obszarami pamięci operacyjnej RAM przyznanymi do przechowywania danych.
- ✓ Odwołując się do zmiennej poprzez jej nazwę odwołujemy się do przydzielonej jej pamięci.
- ✓ **Wartością wskaźnika** jest adres pamięci RAM, gdzie znajduje się taka zmienna.
- ✓ Adres zmiennej przechowujemy w **zmiennej wskaźnikowej**.

Definiowanie wskaźników

Zmienne wskaźnikowe dzielą się na różne typy – przeznaczone do przechowywania adresów różnych typów danych.

typ_wskazywanego_obiektu * nazwa wskaźnika;

Np.:

```
int *wsk_na_int;  
char * wsk_na_znak;  
float  * wsk_na_float;
```

Pojęcie wskaźnika

Aby uzyskać adres zmiennej statycznej, który można przechowywać w zmiennej wskaźnikowej posłużyć się można operatorem **&**

```
int *wskaznik;  
int zmienna = 10;
```

Zdefiniowanie wskaźnika na int oraz zmiennej typu int

Przekazanie adresu „zmiennej” do wskaźnika

```
wskaznik = &zmienna;
```

Posługiwanie się wskaźnikami

- | | |
|--------------------------------|---------------------------------------------------|
| <code>int *x;</code> | - definicja wskaźnika do obiektów typu int |
| <code>int st = 100;</code> | - definicja obiektu typu int z liczbą 100 |
| <code>x = &st;</code> | - ustawienie wskaźnika na obiekt st |
| <code>cout << *x;</code> | - wypisanie wartości obiektu wskazywanego przez x |
| <code>cin >> *x;</code> | - zapisanie wartości do wskaźnika |

```
int *x;           // definicja wskaźnika
                  // do obiektów typu int

int st = 100;     // definicja obiektu typu int
                  // zainicjalizowanego liczbą 100

x = &st;          // ustawienie wskaźnika na obiekt st

cout << *x;       // wypisanie wartości obiektu
                  // wskazywanego przez x

cin >> *x;        // to samo, co cin >> st;
```



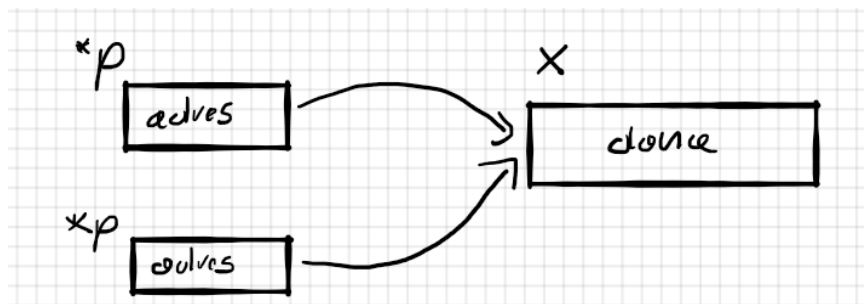
```
1  #include <iostream>
2  using namespace std ;
3
4  int main()
5  {
6      int zmienna = 8 , drugi = 4 ;
7      int *wskaznik ;
8
9      wskaznik = &zmienna ;
10
11     // prosty wypis na ekran ;
12     cout << "zmienna = " << zmienna
13          << "\n a odczytana przez wskaznik = "
14          << *wskaznik << endl ;
15
16     zmienna = 10 ;
17     cout << "zmienna = " << zmienna
18          << "\n a odczytana przez wskaznik = "
19          << *wskaznik << endl ;
20
21     *wskaznik = 200 ;
22     cout << "zmienna = " << zmienna
23          << "\n a odczytana przez wskaznik = "
24          << *wskaznik << endl ;
25
26     wskaznik = &drugi ;
27     cout << "zmienna = " << zmienna
28          << "\n a odczytana przez wskaznik = "
29          << *wskaznik << endl ;
30 }
```

Posługiwanie się wskaźnikami

Wskaźniki jako argumenty funkcji - przekazując wskaźniki jako argumenty funkcji sprawiamy, że z wnętrza funkcji mamy pełny dostęp do zmiennych przekazanych jako argumenty (możemy je modyfikować).

Efekt jest podobny jak przy przekazywaniu argumentów przez referencję.

```
1  #include<iostream>
2  using namespace std;
3  void zamien(int *x, int *y)
4  {
5      int pom = *x;
6      *x = *y;
7      *y = pom;
8  }
9  int main()
10 {
11     int a, b;
12     int *p1=&a, *p2=&b;
13     cin>>a>>b;
14     zamien(p1,p2); //przekazujemy adresy zmiennych
15     cout<<a<<" "<<b; //wartości zmiennych zostały zamienione
16     return 0;
17 }
```





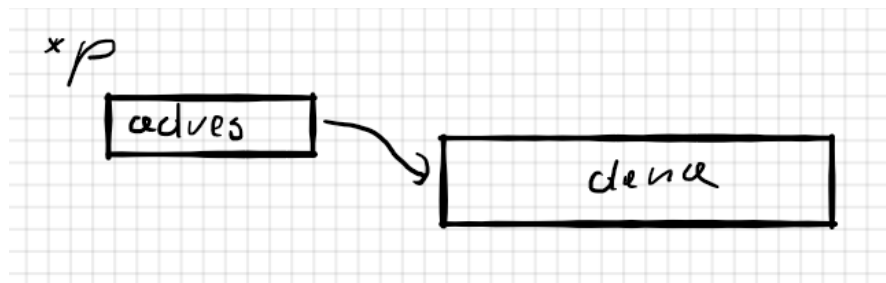
Zmienne dynamiczne

Zmienne dynamiczne

Wskaźniki można zastosować do **dynamicznej alokacji zmiennych** – czyli rezerwacji w pamięci obszarów do przechowywania zmiennych w trakcie działania programu.

Tak stworzona zmienna nie ma nazwy, lecz tylko adres.

Adres ten przechowywany jest w statycznej zmiennej wskaźnikowej (lub bardziej skomplikowanej strukturze danych takiej jak lista lub drzewo binarne).



Zmienne dynamiczne



Do dynamicznej alokacji zmiennych służy operator **new**

```
int *wsk;  
wsk = new int;
```

Lub krócej:

```
int *wsk = new int;
```

Operator **new** na podstawie typu zmiennej (lub też typu i rozmiaru tablicy) przydzieli odpowiednią ilość pamięci.

Jeśli przydział pamięci powiódł się, to wartość zmiennej „**wsk**” będzie różna od zera.

Jeśli wartość wskaźnika będzie równa 0, to pamięć nie została przydzielona. Wartość 0 bardzo często jest zastępowana stałą NULL (zalecane).

Sytuacje w których pamięć nie może zostać przydzielona:

1. rozmiar bloku pamięci, który chcesz zarezerwować jest zbyt duży;
2. system nie posiada więcej zasobów pamięci i w związku z tym nie może jej przydzielić.

Zmienne dynamiczne



Do dynamicznej alokacji zmiennych służy operator **new**

```
int *wsk;  
wsk = new int;
```

Lub krócej:

```
int *wsk = new int;
```

Do usunięcia z pamięci zmiennej dynamicznej służy operator **delete**

```
delete wsk;
```

W języku **C** do przydzielania i zwalniania pamięci służyły głównie **funkcje malloc()** i **free()** w **C++** zostały one zastąpione **operatorami new i delete**.

Zmienne dynamiczne



Za pomocą operatora **delete** kasuje się tylko obiekty stworzone operatorem **new**

Próba skasowania czegokolwiek innego jest błędem.

Uwaga: nie należy dwukrotnie kasować obiektu.

Wyjątkiem jest zastosowanie operatora delete w stosunku do wskaźnika pokazującego na **adres zerowy (NULL)** – ponieważ żaden obiekt nie może mieć adresu 0 - taka konstrukcja nie powoduje błędu.

Zmienne dynamiczne



Często stosowana konstrukcja zabezpieczająca przed podwójnym kasowaniem obiektów:

```
int *wskaznik = new int;  
// .....  
delete wskaznik;  
wskaznik = NULL;  
// .....  
delete wskaznik; // nie spowoduje błędu
```



Cechy obiektów dynamicznych.

- Obiekty utworzone dynamicznie istnieją od momentu, gdy je utworzymy operatorem **new** do momentu, gdy je skasujemy operatorem **delete**.
- Obiekt utworzony dynamicznie nie ma nazwy. Można nim operować tylko za pomocą wskaźników.
- Obiektów takich nie obowiązują zwykłe zasady o zakresie ważności (zasady mówiące w których miejscach programu są widzialne, a w których niewidzialne pomimo, że istnieją). Jeśli tylko jest w danym momencie dostępny choćby jeden wskaźnik, który na taki obiekt pokazuje, to mamy do tego obiektu dostęp.
- Obiekty dynamicznie nie są inicjalizowane zerami (po utworzeniu zawierają przypadkowe wartości).

Tablice dynamiczne



Dynamiczna alokacja tablic:

```
int *wsk;
```

```
wsk = new int [1000];
```

Lub krócej:

```
int *wsk = new int [1000];
```

Rozmiar tablicy nie musi być stałą. Wystarczy, że jego wartość będzie znana w momencie alokacji tablicy (niekoniecznie w momencie kompilacji programu)

Tablice dynamiczne



Do usunięcia z pamięci dynamicznej tablicy służy konstrukcja:

```
delete [] wsk;
```

Jeśli przydzieliliśmy pamięć określając ilość elementów tablicy to musimy poinformować operator **delete** o tym, że wskaźnik wskazywał na tablicę – inaczej usuniemy tylko jej pierwszy element.

Aby to zrobić dopisujemy zaraz za operatorem nawiasy kwadratowe []

Nie podajemy w nich rozmiaru tablicy, ponieważ operator sam ustala rozmiar bloku jaki został przydzielony.

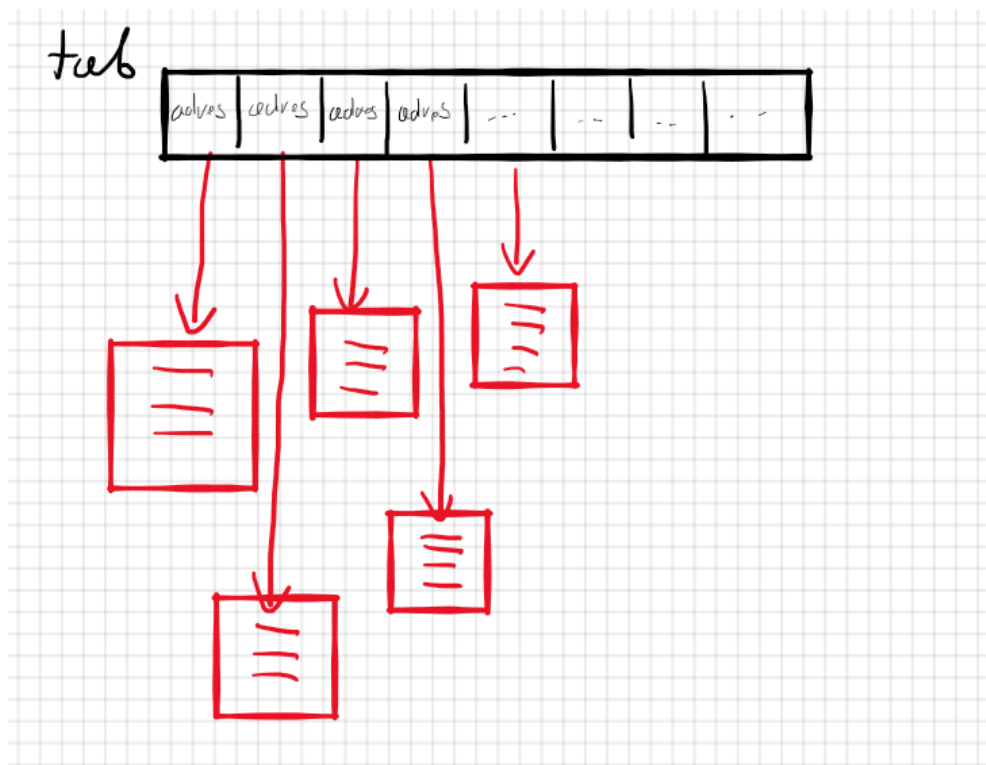
Tablice dynamiczne

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "ile lementow ma miec tablica? ";
8      int ile;
9      cin >> ile;
10
11     int *tab = new int[ile];
12     int *p=tab;
13     for (int i=0; i<ile;i++) *p++=i;
14     p=tab;
15     for (int i=0; i<ile;i++)
16         cout << tab[i]<<" "<<endl;
17     delete [] tab;
18     return 0;
19 }
```

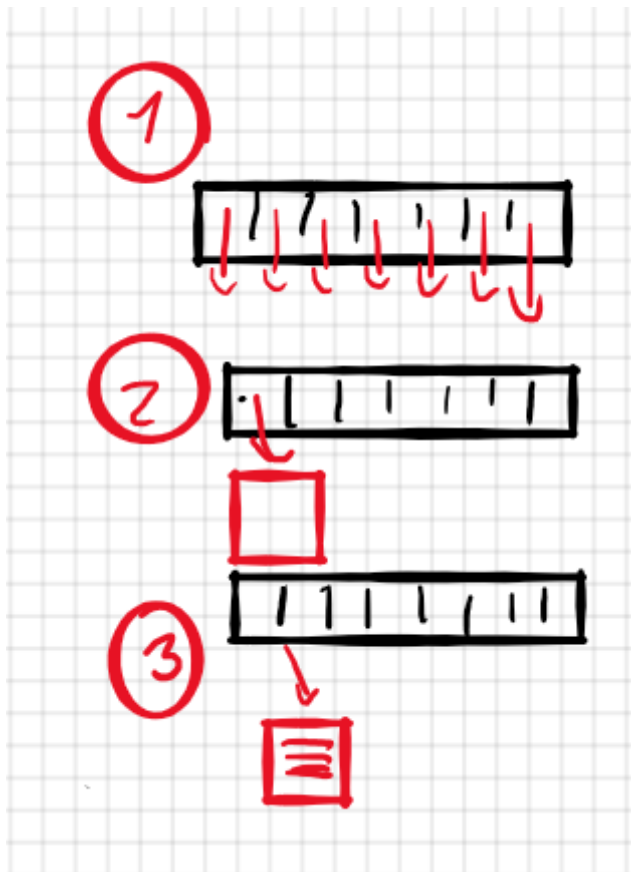
Tablice wskaźników

W tablicy przechowywać można wszystkie rodzaje zmiennych prostych, w tym także wskaźniki adresami różnych miejsc w pamięci.

```
int *tabl_wsk[1000];
```



Tablice wskaźników



```
int *tab[100];
```

```
tab[0] = new int;
```

```
*tab[0] = 10;
```

Tablice wskaźników



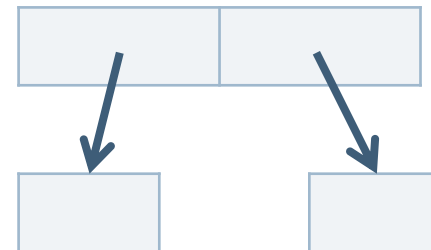
```
int    *tabl_wsk[1000];
```

```
int    *tabl_wsk[1000];  
    //statyczna tablica zawierająca wskaźniki  
for (int i=0; i<1000; i++)  
    tabl_wsk[i] = new int;  
    // tworzenie zmiennych dynamicznych  
    // i zapamiętanie wskaźników do nich w tablicy  
cout << *tabl_wsk[0];  
    // wypisanie zmiennej dynamicznej  
    // wskazywanej przez pole z tablicy
```

Tablice wskaźników do zmiennych dynamicznych

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int *tablica[10]; //STSTYCZNA tablia wskaxników
8      int temp;
9      for (int i=0; i<10;i++) //tworzymy zmienne dynamiczne
10     { //wskazniki do nich zapamietujemy w tablicy
11         cout<<"Tab["<<i<<"]=";
12         cin >> temp;
13         tablica[i]=new int;
14         *tablica[i]=temp;
15     }
16     for (int i=0;i<10;i++) //wypisujemy zawartość
17     { //zmiennych podczepionych pod tablice
18         cout << *tablica[i] <<" ";
19     }
20     for (int i=0;i<10;i++) //usuwanie elementów
21     { //podczepionych pod tablice
22         delete tablica[i];
23     }
24     return 0;
25 }
```

tablica





Czyszczenie bufora strumienia wejściowego

Czyli drobna, ale przydatna wskazówka,
jeżeli chcemy wczytywać naprzemiennie
liczby za pomocą `cin` i łańcuchy za
pomocą `getline()`

Bufor strumienia cin

```
int main()
{
    string s;
    int x;
    cout << "Podaj tekst: ";
    getline(cin, s);
    cout << "Podaj liczbe: ";
    cin>>x;
    cout<< "Wczytano: "<<s<< " i "<<x;
    return 0;
}
```

W tej wersji program działa poprawnie.

Po zamianie kolejności poleceń cin i getline program działa błędnie.

```
int main()
{
    string s;
    int x;
    cout << "Podaj liczbe: ";
    cin>>x;
    cout << "Podaj tekst: ";
    getline(cin, s);
    cout<< "Wczytano: "<<x<< " i "<<s;
    return 0;
}
```

błąd

Po wczytaniu liczby, w buforze strumienia wejściowego cin zostaje znak końca wiersza. Jest on przechwytywany przez getline() – czyli do getline() „wrzucany” jest od razu enter – użytkownik nie ma szans nic wpisać. Zmienna s zawiera więc łańcuch pusty.

Bufor strumienia cin

```
int main()
{
    string s;
    int x;
    cout << "Podaj liczbe: ";
    cin >> x;

    cin.clear();
    cin.ignore(1000, '\n' );

    cout << "Podaj tekst: ";
    getline(cin, s);
    cout << "Wczytano: " << x << " i " << s;
    return 0;
}
```

cin.clear powoduje usunięcie flagi błędu ale w buforze wejściowym nadal jest znak końca wiersza. Flaga błędu pojawi się gdy podamy nieprawidłowy typ danych (np. zapiszemy łańcuch znaków w zmiennej int)

cin.ignore() – spowoduje zignorowanie znaków w buforze.

- pierwszy parametr – liczba znaków do usunięcia (maksymalna)
- drugi parametr – na jakim znaku kończymy usuwanie np. '\n' znak końca wiersza

Bufor strumienia cin

Druga - prosta, ale nieco mniej elegancka metoda – po wyczyszczeniu flagi błędów wczytujemy zawartość bufora (np. znak końca wiersza) do tymczasowej zmiennej (w tym przykładzie „kosz”).

Zawartość tej zmiennej nas nie interesuje, ale bufor strumienia wejścia zostanie w ten sposób wyczyszczony.

```
int main()
{
    string s, kosz;
    int x;
    cout << "Podaj liczbe: ";
    cin>>x;
    cin.clear();
    getline(cin, kosz);
    cout << "Podaj tekst: ";
    getline(cin, s);
    cout<< "Wczytano: "<<x<< " i "<<s;
    return 0;
}
```

Bufor strumienia cin

Przykład: wykorzystanie powyższej metody do kontroli poprawności danych.

```
int main()
{
    string s, kosz="";
    int x;
    cout << "Podaj liczbe: ";
    cin>>x;
    cin.clear();
    getline(cin, kosz);
    cout << "Podaj tekst: ";
    getline(cin, s);
    cout<< "Wczytano: "<<x<< " i "<<s<<endl;
    if (kosz!="") cout<< "Dana: "<<kosz<< " nie jest liczba typu int";
    return 0;
}
```



Struktury

Dla przypomnienia

Struktury to złożone typy danych pozwalające przechowywać dane różnego typu w jednym obiekcie.

- ✓ Za pomocą struktur możliwe jest grupowanie wielu zmiennych o różnych typach.
- ✓ Za pomocą struktur można w prosty sposób organizować zbiory danych, bez konieczności korzystania z tablic.

Struktura nazywana jest też **rekordem** (szczególnie w odniesieniu do baz danych).

Deklaracja struktury w C++

Struktury tworzymy słowem kluczowym **struct**.

1. podajemy nazwę typu,
2. w nawiasie klamrowym definiujemy elementy składowe

```
struct nazwa{  
    typ nazwa_elementu;  
    typ nazwa_drugiego_elementu;  
    typ nazwa_trzeciego_elementu;  
    // ...  
};
```

Uwaga!

Tak opisana struktura nie jest jeszcze egzemplarzem zmiennej a dopiero definicją nowego typu zmiennej złożonej.

Deklaracja struktury w C++

Przykład – struktura zawierająca rekord prostej bazy danych:

```
struct osoba {  
    string imie;  
    string nazwisko;  
    int wiek;  
};
```

Opisujemy typ strukturalny o nazwie „osoba”

```
osoba pracownik1, pracownik2;
```

Druga metoda:

```
struct osoba {  
    string imie;  
    string nazwisko;  
    int wiek;  
} pracownik1, pracownik2;
```

Definiujemy dwie zmienne opisanego wyżej typu o nazwach „pracownik1” i „pracownik2”

Inicjalizacja struktury

Struktury można inicjalizować już w chwili ich tworzenia.

```
struct osoba {  
    string imie;  
    string nazwisko;  
    int wiek;  
};  
osoba ktos = {"Jan", "Kowalski", 10};
```

Lub też krócej:

```
struct osoba {  
    string imie;  
    string nazwisko;  
    int wiek;  
} ktos = {"Jan", "Kowalski", 10};
```


Zapis i odczyt danych struktury

Zapis do pól struktury:

```
ktos.imie="Jan";  
ktos.nazwisko="Kowalski";  
ktos.wiek=40;
```

```
struct osoba {  
    string imie;  
    string nazwisko;  
    int wiek;  
} ktos, ktos_inny;
```

```
cout << "Podaj imie: ";  
getline(cin, ktos_inny.imie);
```

```
cout << "Podaj nazwisko: ";  
cin >> ktos_inny.nazwisko; //UWAGA! problem
```

```
cout << "Podaj wiek: ";  
getline(cin, ktos_inny.wiek);
```

Zapis i odczyt danych struktury

Odczyt z pól struktury:

```
cout << ktos.imie << endl;  
cout << ktos.nazwisko << endl;  
cout << ktos.wiek;
```

```
struct osoba {  
    string imie;  
    string nazwisko;  
    int wiek;  
} ktos, ktos_inny;
```

Struktury globalne i lokalne

- ✓ Struktura stworzona przed funkcją main() będzie strukturą globalną, (każdy podprogram będzie mógł z niej korzystać).
- ✓ Struktura stworzona wewnątrz jakiegoś bloku, będzie lokalną i widoczna tylko w tym miejscu.

Globalna

```
6 struct osoba {  
7     string imie;  
8     string nazwisko;  
9     int wiek;  
10 } ktos;  
11  
12 int main()  
13 {  
14     return 0;  
15 }
```

Lokalna

```
7 int main()  
8 {  
9     struct osoba {  
10         string imie;  
11         string nazwisko;  
12         int wiek;  
13     } ktos;  
14     return 0;  
15 }
```

Tablice struktur

Tablicę struktur tworzymy i dowołujemy się do niej w ten sam sposób co do zwykłych tablic prostych zmiennych.

```
nazwa_struktury nazwa_tablicy [liczba_elementów];
```

Tablice możemy tworzyć też bezpośrednio po deklaracji i definicji struktury:

```
struct punkty{  
    int x, y;  
    char nazwa;  
} tab[1000];
```

Tablice struktur

Przykład:

```
1  #include <iostream>
2  #include <string.h>
3  #include <cstdlib>
4  using namespace std;
5  struct osoba {
6      string imie;
7      string nazwisko;
8      int wiek;
9  };
10 int main()
11 {
12     osoba pracownicy[4];
13     for (int i = 0; i<4; i++)
14     {
15         cout << "Podaj imie " << i+1 << " pracownika" << endl;
16         cin >> pracownicy[i].imie;
17     }
18     for (int i = 0; i<4; i++)
19     {
20         cout << pracownicy[i].imie << endl;
21     }
22     return 0;
23 }
24
```

Zagnieżdżenie struktur

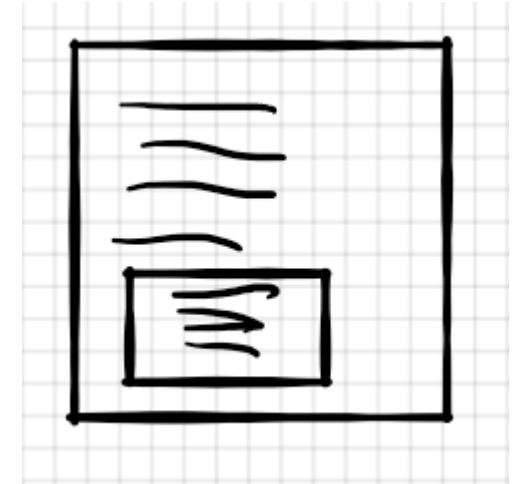
Zagnieżdżanie struktur polega na deklarowaniu pól jednej struktury jako typ strukturalny innej struktury.

- ✓ Struktury można zagnieżdżać wielokrotnie
- ✓ Wiele typów strukturalnych używać można jednocześnie jako pól jednej struktury,

```
5 struct adres{  
6     string miejscowosc;  
7     string ulica;  
8     int nr_domu;  
9 };  
10  
11 struct student{  
12     string imie;  
13     string nazwisko;  
14     adres dom;  
15 };
```

Zagnieżdżenie struktur

- ✓ Do pól zagnieżdżonych struktur odwołujemy wykorzystując wielokrotnie operator "."



```
19 student kowalski;  
20 |  
21 cin>>kowalski.imie;  
22 cin>>kowalski.nazwisko;  
23 cin>>kowalski.dom.miejscowosc;  
24 cin>>kowalski.dom.nr_domu;  
25  
26 cout<<kowalski.imie<<endl;  
27 cout<<kowalski.nazwisko<<endl;  
28 cout<<kowalski.dom.miejscowosc<<endl;  
29 cout<<kowalski.dom.nr_domu<<endl;
```

Struktury jako wartość funkcji

Funkcja może zwracać zmienną typu strukturalnego.

```
5 struct rgb{  
6     int r;  
7     int g;  
8     int b;  
9 };
```

```
11 rgb kolor()  
12 {  
13     rgb pom;  
14     pom.r=255;  
15     pom.g=0;  
16     pom.b=0;  
17     return pom;  
18 }
```

```
20 int main()  
21 {  
22     rgb wynik;  
23     wynik=kolor();  
24     cout<<wynik.r << wynik.g << wynik.b;  
25     return 0;  
26 }
```

Możemy więc zapakować do niej kilka zmiennych typu prostego



Dynamiczne alokowanie struktur

Dynamiczna alokacja struktur

```
struct osoba
{
    string imie;
    string nazwisko;
    int wiek;
};

int main()
{
    osoba *ktos = new osoba;
    ktos->imie = "Jan";
    ktos->nazwisko = "Kowalski";
    ktos->wiek = 40;

    cout<<ktos->imie<<" " <<ktos->nazwisko
         <<" (" <<ktos->wiek<<") ";
    return 0;
}
```

Obsługując strukturę stworzoną
statycznie używamy operatora „.”

`ktos.nazwisko = „Jan”;`

Obsługując strukturę stworzoną
dynamicznie (za pomocą **new**)
używamy operatora „->”

`ktos->nazwisko = „Jan”;`

```
struct osoba
```

```
{
```

```
    string imie;
```

```
    string nazwisko;
```

```
    int wiek;
```

```
};
```

```
int main()
```

```
{
```

```
    osoba * tab[4]; //tablica wskaźników
```

```
                //do struktur typu "osoba"
```

```
    for (int i=0; i<4; i++) //wczytanie danych
```

```
    {
```

```
        tab[i] = new osoba;
```

```
        cin>>tab[i]->imie;
```

```
        cin>>tab[i]->nazwisko;
```

```
        cin>>tab[i]->wiek;
```

```
    }
```

```
    for (int i=0; i<4; i++) //wypisanie tablicy
```

```
    {
```

```
        cout<<tab[i]->imie<<" "<<tab[i]->nazwisko
```

```
        <<" ("<<tab[i]->wiek<<" )";
```

```
    }
```

```
    for (int i=0; i<4; i++)
```

```
    {
```

```
        //usunięcie struktur dynamicznych
```

```
        delete tab[i];
```

```
    }
```

```
    return 0;
```

```
}
```

Przykład: obsługa tablicy
struktur utworzonych
dynamicznie



Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Grębosz J. : ***Symfonia C++, Programowanie w języku C++ orientowane obiektowo***, Wydawnictwo Edition 2000.
- Jakubczyk K.: *Turbo Pascal i Borland C++ Przykłady*, Helion.

Warto zajrzeć także do:

- Sokół R. : ***Microsoft Visual Studio 2012 Programowanie w Ci C++***, Helion.
- Kernighan B. W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

- Grębosz J. : ***Pasja C++***, Wydawnictwo Edition 2000.
- Meyers S.: ***język C++ bardziej efektywnie***, Wydawnictwo Naukowo Techniczne