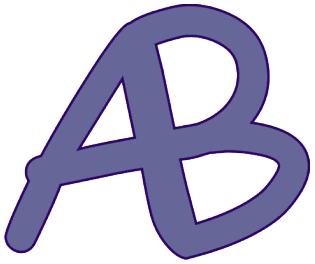

Wykład: 12

Struktury, unie, pola bitowe

Struktury

Struktury

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;
};

osoba pracownik1, pracownik2;
```

Opisujemy typ strukturalny o nazwie „osoba”

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:

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

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

```
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

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

Funkcja może zwracać zmienną typu strukturalnego.

```
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

Struktury jako wartość funkcji

```
5   struct rgb{
6       int kol[3];
7   };
```

Technikę tę można wykorzystać do zwracania przez funkcję tablic

```
9   rgb kolor()
10  {
11      rgb pom;
12      pom.kol[0]=255;
13      pom.kol[1]=255;
14      pom.kol[2]=255;
15      return pom;
16  }
```

```
18  int main()
19  {
20      rgb wynik;
21      wynik=kolor();
22      cout<<wynik.kol[0] << wynik.kol[1] << wynik.kol[2];
23      return 0;
24  }
```

Unie

Unie

Unia typem definiowanym przez użytkownika.

Od struktur różni ją to że swoje składniki zapisuje w tym samym (współdzielonym) obszarze pamięci.

Oznacza to, że w danej chwili, unia może przechowywać wartość wyłącznie jednej ze swoich zmiennych składowych

```
union PrzykladowaUnia
{
    int liczba_calkowita;
    char znak;
    double liczba_rzeczywista;
};
```

Unie

Jeżeli unia zawiera np. obiekt typu int i double, gdy aktualnie korzystamy z double (8B) to po odczytaniu wartości int(4B) bez uprzedniego zapisu do niej pokażą nam się zwykłe śmieci.

- jednocześnie używamy tylko jednego obiektu.

Unie

```
union nazwa{
   typ pierwszy_element;
   typ drugi_element;
   ...
   typ n_ty_element;
};


int main()
{
   //tworzenie unii
   nazwa unia;

   //odwoływanie się do elementów
   unia.pierwszy_element = 0;

   return 0;
}
```

Unie

```
1  #include<iostream>
2  #include<cstdlib>
3  using namespace std;
4
5  union liczba{
6          int calkowita;
7          long long dluga;
8          double rzeczywista;
9  };
10
11 int main()
12 {
13     liczba a, b, c, d;
14     cout<<"Unia zajmuje "<<sizeof(liczba)
15     <<" bajtów"<<endl;
16     cout<<"Podaj trzy liczby całkowite: ";
17     cin>>a.calkowita>>b.calkowita>>c.calkowita;
18     d.rzeczywista = double(a.calkowita+b.calkowita+c.calkowita)/3.0;
19     cout<<"Średnia wczytanych liczb wynosi: "<<d.rzeczywista<<endl;
20     return 0;
21 }
```

Unie

Unie mogą być składowymi innych obiektów takich jak struktur czy klas

```
union liczba{
   int calkowita;
   double rzeczywista;
};

struct samochod{
   char marka[20];
   char model[20];
   int rocznik;
   liczba pojemnosc;
};
```

```
cout<<"Podaj pojemnosc: ";
  cin>>renault.pojemnosc.rzeczywista;
```

Pola bitowe

Pola bitowe

Oprócz zwykłych pól w strukturach możemy zastosować pola bitowe.

Pole bitowe to wydzielenie pewnej stałej liczby bitów na daną zmienną.
Np.: zmienna typu char zajmuje w pamięci 1 bajt = 8 bitów. Możemy ją okroić lub rozszerzyć o porcję bitów dostosowaną do potrzeb programu.

Pamięć jaką będzie zajmować takie pole będzie zawsze krotnością bajtów danego typu zmiennej. Np. jeśli stworzymy pole typu int na 4 bity, to i tak zostanie przydzielona pamięć na cały typ int czyli 4 bajty = 32 bity, a używać będziemy mogli tylko tych czterech bitów.

Więc gdzie tu oszczędność?

Gdy stworzymy na przykład pięć pól bitowych typu int, i każde będzie zajmowało 6 bitów, to w pamięci zostanie zarezerwowany obszar na jeden int, czyli 32 bity, ponieważ $5*6 = 30 \leq 32$.

Pola bitowe

Pola te tworzymy zgodnie z zasadą:

typ nazwa : [ilość bitów];

```
int pole_bitowe : 4; //pole bitowe na czterech bitach
```

Pola bitowe

```
1  #include<iostream>
2  #include<cstdlib>
3  using namespace std;
4
5  struct pola_bitowe{
6    int a:4;
7    int b:4;
8  };
9
10 int main()
11 {
12   pola_bitowe x;
13   cout<<"Struktura zajmuje "<<sizeof(x)<<" bajty pamieci\n";
14   //cztery bajty, tyle ile jeden int
15   x.a = 7;
16   //maksymalna wartosć dodatnia jaka możemy nadać dla typu int na 4 bitach
17   cout<<x.a<<endl;
18   x.a--;
19   //pola bitowe możemy inkrementować i dekrementować tak jak zmienne typu int
20   cout<<x.a<<endl;
21   return 0;
22 }
```

Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Gręboszj. : ***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.
- Kerninghan B. W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

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