ПЕРЕЛІКИ, СТРУКТУРИ ТА ОБ'ЄДНАННЯ

Переліки

Перелічуваний тип (enumerated type) – *тип даних, множиною значень якого є обмежений список ідентифікаторів – констант цього типу.*

```
enum [ім'я_перелічуваного_типу]
{ список_елементів_переліку }
 [список_змінних_переліків];
```

```
enum Color { RED, YELLOW, GREEN };
```

Змінні перелічуваних типів

```
enum Color { RED, YELLOW, GREEN } x, y;
```

```
enum Color { RED, YELLOW, GREEN };
Color x, y;
```

При використанні переліків зустрічаються наступні проблеми:

1) Відображення значення елемента переліку в літерний рядок, який містить ім'я цього елемента, тобто RED → "RED".
2) Ітерація по елементах переліку та контроль виходу за межі: скільки б ми не додавали елементів до переліку, завжди є константа, яка на одиницю перевищує значення останнього елемента

Структури

*Структури – це об'єднані дані **різних типів**, які характеризують певну сутність та за цим змістом пов'язані між собою*

Структурний тип – складається із набору елементів, які називаються полями.

```
struct [ім'я_структурного_типу]
{ список_полів } [список_змінних];
```

```
enum Kurs { I = 1, II, III, IV, V, VI };
enum Spec { PC, KI, KN, IT, IK };
struct Student
{
      string surname;
      unsigned birYear;
      Kurs kurs;
      Spec spec;
};
```

Оголошення змінних структур

```
struct Student
{
      string surname;
      unsigned birYear;
      int kurs;
} st1, st2;
```

```
struct Student
{
      string surname;
      unsigned birYear;
      int kurs;
};
Student st1, st2;
```

```
Student s3[25], * ps;
```

Вкладені структури

```
struct Student
 {
        string surname;
        unsigned birYear;
        int kurs;
        struct adresStud
        {
               string city;
               int number;
        } a;

 };

 Student st1;
 Student::adresStud adr;
```

не допускається рекурсія при визначенні структур

```
struct Student
{
      Student st1;// помилка

};
```

Але

```
struct Student
{
      Student*  st1; ;// дозволено

};
```

Розподіл пам'яті для структур

В пам'яті для структури виділяється неперервна область, розмір якої більший або дорівнює сумі розмірів всіх полів: є нюанс, який називається вирівнюванням – розмір області пам'яті, виділеної для структури, може бути більшим за суму розмірів полів цієї структури.

Ініціалізація структури

Ініціалізація структури виконується аналогічно до ініціалізації масиву:

```
struct Student
{
        string surname;
        unsigned birYear;
        int kurs;
};
Student st1 = {"Petrenko", 2003, 4};
```

Якщо якесь поле залишилося не заповненим, то воно автоматично заповнюється значенням

- *0 – для цілих типів,*
- *NULL – для вказівників,*
- *'\0' (нуль-символ) – для літерних рядків*

Операції над структурами

1) присвоєння полю структури значення відповідного типу;

2) присвоєння всій структурі значення того самого структурного типу;

3) отримання адресу структури за допомогою операції &;

4) отримання доступу до будь-якого поля структури;

5) визначачення розміру структури та її полів за допомогою операції sizeof().

До елементів структури можна звертатися

- за допомогою імені структури

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

struct Student
 {
      string surname;
      unsigned birYear;
      int kurs;
 };
int main()
{
 Student st1;
 st1.surname = "Petrenko";
 st1.birYear = 2003;
 st1.kurs = 4;
 return 0;
}
```

за допомогою вказівника на структуру

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

struct Student
 {
      string surname;
      unsigned birYear;
      int kurs;
 };
int main()
{
 Student* p = new Student;
 (*p).surname = "Petrenko";
 (*p).birYear = 2003;
 (*p).kurs = 4;
 return 0;
}
```

Імена структурних змінних можна використовувати в якості операндів операції присвоєння. При цьому обидві структурні змінні (перед та після знаку присвоєння =) мають бути оголошені за допомогою одного і того самого структурного типу:

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

struct Student
 {
      string surname;
      unsigned birYear;
      int kurs;
 };
int main()
{
 Student st1 ={"Petrenko",2003, 1};
 Student st2=st1;
 st2.kurs = 2;
 return 0;
}
```

Передавання структур у функції

1) всю структуру можна передати у функцію:

```
void f1(Student s)
{
 cout << s.surname << " "
      << s.birYear << " "
      << s.kurs << " ";
      }
```

2) можна передати вказівник на структуру:

```
void f2(Student *s)
{
 cout << (*s).surname << " "
      << (*s).birYear << " "
      << (*s).kurs << " ";
      }
```

3) можна передавати елементи структури:

```
void f3(string s, unsigned bYear, int k)
{
 cout << s << " "
      << bYear << " "
      << k << endl;
}
```

4) можна повертати структури як результат функції:

```
Student f4(Student s)
{
 return s;
}
```

5) можна описати параметри-посилання на структуру

```
void f5(Student& s)
{
 // ...
}
```

Об'єднання

```
union [ім'я_об'єднуваного_типу]
{ список_полів } [список_змінних_об'єднань];
```

Команда визначення об'єднуваного типу – це команда визначення даних, яка містить блок визначення даних (елементів об'єднання).

```
union Pay
 {
      int total;
      double tax;
 };
```

```
union Pay

{
      int total;
      double tax;
} p1,p2;
```

```
union Pay
 {
      int total;
      double tax;
 };
Pay p1, p2;
```

```
Pay p3[5], *p4;
```

Ініціалізація об'єднання

Ініціалізація об'єднання виконується аналогічно до ініціалізації масиву чи структури, проте в команді ініціалізації можна надати значення лише першому полю об'єднання:

```
union Pay
 {
      int total;
      double tax;
 };
 Pay p1 = { 23};

 cout << p1.total << endl;
 cout << p1.tax << endl;
```

Над об'єднаннями можна виконувати ті самі операції, що і над структурами. Об'єднання передаються у / із функції аналогічно структурам.

Визначення типів

Команда `typedef` визначає синонім типу. Її загальний вигляд:

```
typedef визначення_типу нове_ім'я_типу;
```

де

`визначення_типу` – будь-яке допустиме в C++ визначення типу;

`нове_ім'я_типу` – синонім (нове ім’я), що надається цьому типу.

Наприклад:

`int a[10];` – оголошує змінну a як масив із 10 елементів цілого типу.

Використаємо команду визначення синоніму типу typedef

```
typedef int A[10];
A a;
```

– визначає тип A як сукупність масивів із 10 елементів цілого типу та оголошує змінну цього типу.

Еквівалентність типів

Існує кілька схем для визначення, чи еквівалентні типи двох об'єктів. Найчастіше використовуються схеми, що називаються *структурна еквівалентність типів та іменна еквівалентність типів* (в англомовній термінології structural type system та nominative type system відповідно).

```
struct s1 { int a; };
struct s2 { int a; };
```

– визначено два різні типи. Змінні цих типів – несумісні:

```
s1 x;
s2 y = x; // помилка
```

Структурні типи – відрізняються від простих, тому наступне присвоєння – помилкове:

```
s1 x;
int i = x; // помилка: невідповідність типів
```

Команда `typedef` реалізує схему іменної еквівалентності типів – дозволяє задати нове ім'я типу, не визначаючи новий тип: в оголошенні, яке починається ключовим словом `typedef`, описується не змінна, а вводиться нове ім'я для типу

Дякую за увагу

Лектор:
кандидат фіз.-мат. наук, доцент
Шаклеіна Ірина
iryna.o.shakleina@lpnu.ua
кафедра ІСМ, ІКНІ