Структури й об'єднання

Тема 10

ЗМІСТ

ЛІТЕРАТУРА:

1. Бублик В.В. Об’єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.
2. Вступ до програмування мовою С++. Організація обчислень : навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с.
3. Зубенко В.В., Омельчук Л.Л. Програмування. Поглиблений курс. – К.:Видавничо-поліграфічний центр “Київський університет”, 2011. - 623 с.
4. Страуструп Бьярне. Программирование: принципы и практика с использованием С++, 2-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильяме", 2016. - 1328 с.
5. Прата С. Язык программирования С++. Лекции и упражнения. Учебник. -СПб. ООО «ДиаСофтЮП», 2003. 1104 с.
6. Шилдт Г. С++: базовый курс, 3-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2010. – 624 с.
7. Stroustrup, Bjarne. The C++ programming language. — Fourth edition. – Addison-Wesley, 2013. – 1361 pp.

Структури

Структура – це іменована упорядкована група логічно зв'язаних змінних, що зберігаються в одному місці.

Змінні, що складають структуру, називаються її членами (елементами, полями).

```
struct inv_type // оголошення структури
(структурного типу)
{
  char item[40]; // найменування товару (40 байт)
  double cost;   // вартість (8 байт)
  double retail; // роздрібна ціна (8 байт)
  int on_hand;   // кількість, що є в наявності (4
байти)
  int lead_time; // число днів до поповнення
запасів (4 байти)
};
```

Структури

```
inv_type inv_var;

// визначення екземпляра структури

struct inv_type {
  char item[40]; // найменування товару
  double cost;   // вартість
  double retail; // роздрібна ціна
  int on_hand;   // кількість, що є в
наявності
  int lead_time; // число днів до
поповнення запасів
} inv_varA, inv_varB, inv_varC;
```

Якщо для програми досить тільки однієї структурної змінної, у її визначення необов'язково включати ім'я структурного типу:

```
struct {
  char item[40]; // найменування товару
  double cost;   // вартість
  double retail; // роздрібна ціна
  int on_hand;   // кількість, що є в наявності
   int lead_time; // число днів до поповнення
запасів
} temp;
```

Цей фрагмент коду визначає одну структурну змінну **temp** відповідно до оголошеного перед нею безіменного структурного типу.

Загальний формат визначення структури виглядає так:

```
struct ім'я_типу_структури {
   тип ім'я_елемента1;
   тип ім'я_елемента2;
   тип ім'я_елемента3;
   . . .
   тип ім'я_елементаN;
} структурні_змінні;
```

Доступ до членів структури

До окремих членів структури доступ здійснюється за допомогою оператора "крапка".

```
inv_var.cost = 10.39;
```

Загальний формат доступу записується так.

ім'я_структурної_змінної.ім'я_члена

Щоб вивести значення поля cost на екран, необхідно виконати наступну інструкцію.

```
cout << inv_var.cost;
```

Доступ до членів структури

Аналогічним способом можна використовувати символьний масив inv_var.item у виклику функції gets().

```
gets(inv_var.item);
```

Тут функції gets() буде переданий символьний покажчик на початок області пам'яті, відведеної елементу item.

Якщо потрібно одержати доступ до окремих елементів масиву inv_var.item, використовується індексація. Наприклад, за допомогою цього коду можна посимвольно вивести на екран вміст масиву inv_var.item.

```
int t;
for(t=0; inv_var.itern[t]; t++)
  cout << inv_var.item[t];
```

Масиви структур

Структури можуть бути елементами масивів. Наприклад, щоб визначити 100-елементний масив структур типу inv_type, досить записати наступне:

```
inv_type invtry[100];
```

Щоб одержати доступ до конкретної (наприклад, третьої) структури в масиві структур, необхідно індексувати ім'я масиву.

```
cout << invtry[2].on_hand;
```

Передача структур функціям

При передачі структури у функцію як аргумента використовується механізм передачі параметрів **за значенням**, тобто, вміст структури-аргумента просто копіюється у структуру-параметр.

```
#include <iostream>
using namespace std;
struct sample {          // Оголошуємо тип структури.
  int a;
  char ch;
};
void f1(sample parm);
int main() {
  struct sample arg; // Визначаємо змінну arg типу sample.
  arg.a = 1000;
  arg.ch = 'x';
  f1(arg);
  return 0;
}
void f1(sample parm) {
  cout << parm.a << " " << parm.ch << "\n";
}
```

Аргумент **arg** у функції main() і параметр **parm** у функції f1() **мають однаковий тип**.

Присвоювання структур

Вміст однієї структури можна присвоїти іншій, якщо обидві ці структури мають однаковий тип.

Програма надрукує
До присвоювання.
svar1:10 10
svar2:20 20
Після присвоювання.
svar1:10 10
svar2:10 10

```
#include <iostream>
using namespace std;
struct stype {
  int a, b;
};
int main() {
  stype svar1, svar2;
  svar1.a = svar1.b = 10;   svar2.a = svar2.b = 20;
  cout << “До присвоювання.\n";
  cout << "svar1: " << svar1.a << ' ' << svar1.b <<'\n';
  cout << "svar2: " << svar2.a << ' ' << svar2.b <<"\n\n";
  svar2 = svar1; // присвоювання структур
  cout << “Після присвоювання.\n”;
  cout << "svar1: " << svar1.a << ' ' << svar1.b << '\n';
  cout << "svar2: " << svar2.a << ' ' << svar2.b;
  return 0;
}
```

В C++ кожний новий опис структури визначає новий тип.

Навіть якщо дві структури фізично однакові, але мають різні імена типів, компілятор буде вважати їх різними (різнотипними) й не дозволить присвоїти значення однієї з них іншій.

Наступний фрагмент коду некоректний і тому не скомпілюється.

```
struct stype1 {
  int a, b;
};

struct stype2 {
  int a, b;
};

stype1 svar1;
stype2 svar2;
svar2 = svar1;
// Помилка через невідповідність типів.
```

Незважаючи на те що структури stype1 і stype2 фізично однакові, з погляду компілятора вони є різними типами.

Використання покажчиків на структури й оператора "стрілка"

Покажчик на структуру оголошується так само, як покажчик на змінну будь-якого іншого типу, тобто за допомогою символу «*», поставленого перед ім'ям структурної змінної:

```
inv_type *inv_pointer;
```

Щоб знайти адресу структурної змінної, треба перед її ім'ям розмістити оператор «&».

```
struct bal {
  float balance;
  char name[80];
} person;
bal *p; // Оголошуємо покажчик на структуру.
```

Тоді при виконанні інструкції

```
p = &person;
```

у покажчик `p` буде поміщена адреса структурної змінної `person`.

Використання покажчиків на структури й оператора "стрілка"

До членів структури можна одержати доступ за допомогою покажчика на цю структуру. Але в цьому випадку використовується не оператор "крапка", а оператор `->` "стрілка". Наприклад, при виконанні цієї інструкції одержуємо доступ до поля `balance` через покажчик `p`:

```
p->balance
```

Покажчик на структуру можна використовувати як параметр функції. (Передача покажчика завжди відбувається швидше, ніж передача самої структури.)

Посилання на структури

```
#include <iostream>
using namespace std;
struct mystruct {
  int a; int b;
};
mystruct &f(mystruct &var);
int main() {
  mystruct x, y;
  x.a = 10; x.b = 20;
  cout << "Вихідні значення полів x.a та x.b: ";
  cout << x.a << ' ' << x.b << '\n';
  y = f (x);
  cout << "Модифіковані значення полів x.a та x.b: ";
  cout << x.a << ' ' << x.b << '\n';
  cout << "Модифіковані значення полів y.a та y.b: ";
  cout << y.a << ' ' << y.b << '\n';
  return 0;
}
mystruct &f(mystruct &var) {
  var.a = var.a * var.a;
  var.b = var.b / var.b;
  return var;
}
```

```
Вихідні значення полів x.a та x.b: 10 20
Модифіковані значення полів x.a та x.b: 100 1
Модифіковані значення полів y.a та y.b: 100 1
```

Використання в якості членів структур масивів і структур

```
struct stype {
  int nums[10][10]; //масив 10x10
  float b;
} var;
```

Звертаємось до елемента масиву nums з індексами 3, 7 у структурі `var` типу `stype`:

```
var.nums[3][7]
```

Якщо деяка структура є членом іншої структури, вона називається **вкладеною структурою**.

```
struct addr {
  char name[40];
  char street[40];
  char city[40];
  char zip[10];
}
struct emp {
  addr address;
  float wage;
```

Використання в якості членів структур масивів і структур

Членом структури також може бути покажчик на цю ж структуру.

Наприклад, у наступній структурі змінна `sptr` оголошується як покажчик на структуру типу `mystruct`, тобто на оголошену тут структуру.

```
struct mystruct {
  int a;
  char str[80];
  mystruct *sptr; // покажчик на
//структуру mystruct
};
```

Структури, що містять покажчики на самих себе, часто використовуються при створенні таких структур даних, як зв'язні списки, «дерева» тощо.

Порівняння С- і С++-структур

С++-структури - нащадки С-структур. Отже, будь-яка С-структура також є і дійсною С++-структурою

Існують важливі відмінності:

1) С++-структури мають деякі унікальні атрибути, які дозволяють їм підтримувати об’єктно-орієнтоване програмування.
2) **Оголошуючи структуру в С++, ми визначаємо ім’я нового типу**, яке можна використовувати для визначення змінних, значень, що повертаються функціями тощо. Проте **у мові С ім'я структури називається її тегом (або дескриптором). А тег сам по собі не є ім'ям типу.**

Порівняння C- і C++-структур

```
struct C_struct {                    // це фрагмент C-коду
  int a;
  int b;
}
struct C_struct svar;      // визначення змінної C_struct
```

Але **визначення структурної змінної `svar` також починається із ключового слова `struct`**.

На відміну від C++ у мові C після визначення структури для повного задання типу даних все одно потрібно використовувати ключове слово `struct` разом з тегом цієї структури (у даному випадку з ідентифікатором `C_struct`).

Проте C++ приймає C-орієнтовані оголошення. Тобто, попередній фрагмент C-коду коректно скомпілюється як частина будь-якої C++-програми.

Бітові поля структур

C++ передбачений вбудований спосіб доступу до конкретного розряду байта.

Побітовий доступ забезпечується використанням бітових полів.

Бітові поля можуть бути корисними в різних ситуаціях:

1) Якщо мало пам'яті, можна зберігати кілька булевих (логічних) значень в одному байті.
2) Інтерфейси деяких пристроїв передають інформацію, закодовану саме в бітах.
3) Існують підпрограми кодування, яким потрібний доступ до окремих бітів у рамках байта.

Бітове поле - це спеціальний тип члена структури, що визначає свій розмір (довжину) у бітах.

Бітові поля структур

```
struct ім'я_типу_структури {
  тип ім'я1 : довжина;
  тип ім'я2 : довжина;
  . . .
  тип имя : довжина;
};
```

Тут **елемент тип означає тип бітового поля**, а **елемент довжина - кількість бітів у цьому полі**. Бітове поле повинно бути оголошене як значення цілочисельного типу або перерахування.

Бітові поля довжиною 1 біт оголошуються як значення типу без знака (unsigned), оскільки єдиний біт не може мати знакового розряду

Порт станів послідовного адаптера зв'язку може повертати байт стану, з таким призначенням його окремих бітів:

0 - Зміна в лінії установки у вихідний стан
1 - Зміна в лінії готовності даних
2 - Виявлено задній фронт
3 - Зміна в лінії прийому даних
4 - Встановлення у вихідне положення
5 - Дані готові
6 - Телефонний сигнал виклику
7 - Сигнал прийнято

Інформацію, що міститься в байті стану, можна представити структурою з бітовими полями.

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge: 1;
  unsigned delta_rec: 1;
  unsigned cts: 1;
  unsigned dsr: 1;
  unsigned ring: 1;
  unsigned rec_line: 1;
} status;
```

Щоб визначити, коли можна відправити або одержати дані, використовується код:

```
status = get_port_status();
if(status.cts)
 cout<<"Установка у вихідний стан";
if(status.dsr)
 cout << "Дані готові";
```

Наступна інструкція очищає бітове поле ring:

```
status.ring = 0;
```

Якщо загальний доступ до структури здійснюється через покажчик, необхідно використовувати оператор "->".

Варто мати на увазі, що зовсім необов'язково присвоювати ім'я кожному бітовому полю. Це дозволяє звертатися тільки до потрібних бітів, "обходячи" інші. Наприклад, якщо нас цікавлять тільки біти cts і dsr:

```
struct status_type {
  unsigned:4; //4 біта не іменовані
  unsigned cts:1;
  unsigned dsr:1; //біти далі не описуються
```

У структурі можна змішувати "звичайні" члени і члени з бітовими полями:

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off: 1;    // працює чи ні
  unsigned hourly: 1;
// погодинна оплата або оклад
  unsigned deductions: 3;
// утримання податку (8 варіантів)
};
```

Ця структура визначає запис по кожному службовцю, у якому використовується тільки один байт для зберігання трьох елементів інформації: статус службовця, характер оплати його праці (погодинна оплата або твердий оклад) і податкова ставка. Без використання бітових полів для зберігання цієї інформації довелося б зайняти три байти.

Використання бітових полів має певні обмеження:

- Програміст не може одержати адресу бітового поля або посилання на нього.
- Бітові поля не можна зберігати в масивах.
- Їх не можна оголошувати статичними.
- При переході від одного комп'ютера до іншого неможливо знати напевно порядок проходження бітових полів: справа наліво або зліва направо.

Це означає, що будь-яка програма, у якій використовуються бітові поля, може страждати певною залежністю від марки комп'ютера.

Можливі й інші обмеження, пов'язані з особливостями реалізації компілятора C++, тому має сенс прояснити це питання у відповідній документації.

Об'єднання

Об'єднання складається з декількох змінних, які займають одну область пам'яті.

Об'єднання забезпечує можливість інтерпретації однієї й тієї ж конфігурації бітів двома (або більше) різними способами.

Оголошення об'єднання подібне оголошенню структури.

```
union utype {
   short int i;
   char ch;
};
```

Змінну об’єднання можна визначити, розмістивши її ім'я наприкінці оголошення або інструкцією:

```
utype u_var;
```

Об'єднання

Доступ до елемента об'єднання здійснюється так як для для структур: оператори «**.**» і «**->**»

```
u_var.i = 0; // обнулимо пам’ять під u_var
u_var.ch = 'A’;
// присвоїть букву A елементу ch об'єднання u_var
cout << u_var.i;
//надрукує 65 - ASCII-код латинської букви A
```

У наступному прикладі функції передається покажчик на об'єднання u_var

```
void func1 (utype *un){
 un->i = 10; // присвоїть число 10 члену i об'єднання u_var
}
{
  // . . .
  func1(&u_var);
// Передаємо func1() адресу об'єднання u_var
  // . . .
}
```

```
#include <iostream>                          // Перестановка двох байтів
using namespace std;
void disp_binary(unsigned u);
union swap_bytes {
   short int num;
   char ch[2];
};

int main() {
   swap_bytes sb;
   char temp;
   sb.num = 15; //двійковий код: 00000000 00001111
   disp_binary(sb.ch[1]); cout << " "; disp_binary(sb.ch[0]);
   cout << "\n";
   temp=sb.ch[0]; sb.ch[0] = sb.ch[1]; sb.ch[1] = temp; // Обмін байтів
   disp_binary(sb.ch[1]); cout << " "; disp_binary(sb.ch[0]);
   return 0;
}

void disp_binary(unsigned u) { // Відображення бітів, що складають байт
   register int t;
   for(t=128; t>0; t=t/2)
      if(u & t) cout << "1 ";  else cout << "0 ";
}
```

```
00000000 00001111
00001111 00000000
```

Важливо!

Оскільки **об'єднання** припускає, що кілька різнотипних змінних займають одну й ту ж саму область пам'яті, ця структура **надає можливість зберігати інформацію, яка** (залежно від ситуації) **може інтерпретуватись по різному.**

Об'єднання забезпечують низькорівневу підтримку принципів поліморфізму.

Об'єднання забезпечує єдиний інтерфейс для кількох різних типів даних, втілюючи в такий спосіб концепцію "один інтерфейс - безліч методів" у своїй найпростішій формі.

Анонімні об'єднання

Анонімні об'єднання дозволяють визначати змінні, які займають спільну область пам'яті.

Анонімне об'єднання не має імені типу, і тому змінну такого об'єднання визначити неможливо.

Анонімне об'єднання повідомляє компілятор про те, що його члени займають одну і ту ж саму область пам'яті.

Звертання до самих змінних об'єднання відбувається безпосередньо, без використання оператора "крапка".

Анонімні об'єднання

```
#include <iostream>
using namespace std;
int main()
{
  // Це анонімне об'єднання.
  union {
    short int count;
    char ch[2];
  };
  // Безпосереднє звертання до членів анонімного
об'єднання.
  ch[0] = 'X';
  ch[1] = 'Y';
  cout << “У вигляді символів: “ << ch[0] << ch[1] << '\n';
  cout << “У вигляді цілого значення: " << count << '\n';
  return 0;
}
```

Ця програма відображає такий результат.

```
У вигляді символів: XY
У вигляді цілого значення: 22872
```

Використання оператора sizeof

- Іноді компілятор заповнює структуру або об'єднання так, щоб вирівняти їх на межу парного слова або абзацу. (Абзац містить 16 байт.)
- **Якщо в програмі потрібно визначити розмір (у байтах) структури або об'єднання, треба скористатись оператором sizeof**.
- Не намагайтеся вручну рахувати ромір складної змінної додаванням розмірів окремих членів.
- Через заповнення або інші апаратно-залежні фактори розмір структури або об'єднання може виявитися більшим суми розмірів окремих їхніх членів.

Використання оператора sizeof

Об'єднання завжди буде займати область пам'яті, достатню (не меншу) для зберігання його найбільшого члена. Розглянемо приклад.

```
union x {
  char ch;
  int i;
  double f;
} u_var;
```

Використання оператора sizeof

При виконанні оператора **sizeof u_var** одержимо результат 8 (за умови, що double-значення займає 8 байт).

Під час виконання програми не має значення, що реально буде зберігатися в змінній u_var; тут важливий розмір найбільшої змінної, що входить до складу об'єднання, оскільки об'єднання повинно мати розмір найбільшого його елемента.

ЗАПИТАННЯ?