Об’єднання в мові програмування С.

Об’єднання можна означити як визначений користувачем тип даних, який є колекцією різних змінних різних типів даних в одному участку пам’яті. Також коректно означити об’єднання як “багато елементів, серед яких лише один елемент містить значення в певний момент часу”.

Об’єднання можна порівняти зі структурами, проте на відміну від них, вони використовують одне місце в пам’яті.

Розглянемо на прикладі:

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

struct abc  {
    int a;
    char b;
};

int main() {
    struct abc A;
    A.a = 10;
    A.b = 'b';
    cout << &A.a << endl;
    cout << (void *) &A.b << endl;
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTE

0x7fffffffd990
0x7fffffffd994
```

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

union abc  {
    int a;
    char b;
};

int main() {
    union abc A;
    A.a = 10;
    A.b = 'b';
    cout << &A.a << endl;
    cout << (void *) &A.b << endl;
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTE

0x7fffffffd994
0x7fffffffd994
```

Код вище демонструє визначену мною структуру, яка складається з кількох змінних: змінної **a** типу **int** (ціле число) та **b** типу **character** (символ).

Можна побачити також, що адреси цих змінних різні.

Об’єднання визначаються схожим із структурою чином, єдина відмінність у використанні іншого ключового слова - **union** замість **struct.** І якщо перевірити адреси змінних об’єднання, ми помітимо, що їхні адреси тепер збігаються.

Визначення розміру об’єднання

Розмір об’єднання визначається як розмір найбільшої змінної у ньому.

Розмір int - 4 байти, char - 1 байт, float - 4 байти та double - 8 байтів. Так як найбільше пам’яті займає саме double, тому і під наше об’єднання виділено саме таку частину пам’яті, що можна побачити на прикладі.

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

union abc  {
    int a;     // size = 4
    char b;    // size = 1
    float c;   // size = 4
    double d;  // size = 8
};

int main() {
    union abc A;
    cout << "Розмір об'єднання: " << sizeof(A) << endl;
    cout << endl;
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

Розмір об'єднання: 8
```

```cpp
int main() {
    union abc *ptr;
    union abc A;
    A.a = 90;
    ptr = &A;
    cout << "The value of a is: " << ptr->a << endl;
    cout << endl;
    return 0;

    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

The value of a is: 90
```

Доступ до змінних об’єднання.

Отримати доступ до змінної об’єднання можна за допомогою вказівника, використовуючи оператор “стрілка” (->).

Для чого потрібні об’єднання?

Знову розглянемо на прикладі. Припустимо, в нас є магазини, в яких продаються книги та сорочки.

Власник магазинів хоче зберігати записи двох вищезазначених товарів разом із відповідною інформацією. Наприклад, книги містять назву, автора, кількість сторінок, ціну, а сорочки містять колір, дизайн, розмір і ціну. Властивість 'ціна' є спільною для обох елементів.

```cpp
struct store
{
    double price;
    char *title;
    char *author;
    int number_pages;
    int color;
    int size;
    char *design;
};
```

Ця структура містить всі відомості, які власник хоче зберігати. Також вона цілком робоча, але ціна (price) відноситься до обох типів предметів, а всі інші характеристики є індивідуальними.

Наступним чином ми отримуємо доступ до змінних цієї структури:

```cpp
int main()
{
    struct store book;
    book.title = "C programming";
    book.author = "Paulo Cohelo";
    book.number_pages = 190;
    book.price = 205;
    cout << "Розмір: " << sizeof(book) << " байт" << endl;
    return 0;
}
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER: VARIABLES

Розмір: 48 байт
```

У коді вище я створюю змінну типу **store** і маю доступ до значень змінних назва, автор, кількість сторінок та ціна, але ця змінна не володіє змінними розмір, колір та дизайн. Очевидно, що це витрата пам’яті, адже структура займає 48 байт.

Таку втрату пам’яті можна вирішити саме скориставшись об’єднанням.

За рахунок використання **union** для виділення обралась найбільша пам’ять, яку займає змінна. Вивід наведеної нижче програми становить 32 байти.

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

struct store {
    double price;

    union  {
        struct {
        char *title;
        char *author;
        int number_pages;
        } book;

        struct {
        int color;
        int size;
        char *design;
        } shirt;
    } item;
};

int main() {
    struct store s;
    s.item.book.title = "C programming";
    s.item.book.author = "John";
    s.item.book.number_pages = 189;
    cout << "Розмір (з використанням об'єднання) " << sizeof(s) << " байти" << endl;
    cout << endl;
    return 0;
}
```

```
PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER: VARIABLES

Розмір (з використанням об'єднання) 32 байти
```

Псевдоконтейнер variant у мові програмування C++

std::variant додали у C++17 задля підтримки типів-сум та типів-добутків. Він додає більше функціональності об’єднанню і є його безпечнішою версією.

Тип-суми та тип-добутки

Тип визначає набір значень і допустимі операції над цими значеннями.

Типи зазвичай класифікуються в різні **класи типів**, такі як перелічувані (*enumerable)*, порівнювані (*comparable)* тощо. Типи, що нас цікавлять, це складені типи, які мають тип-суми та тип-добутки.

Тип-суми (sum types), такі як T1+T2, складаються з суми усіх значень T1 з усіма значеннями T2. Тип-суми часто називають об’єднаннями (**unions**). Якщо альтернативи позначені (або названі), ми називаємо їх тегованими об’єднаннями (**tagged unions).** Загалом ми можемо мати суми будь-якого розміру: T1+T2+ … +Tn.

Тип-добутки (product types), такі як T1×T2 складаються з усіх пар (x, y), де x - це змінна типу T1, а y - типу T2. Зазвичай типи-добутки можуть мати будь-який розмір і називаються кортежами (**tuples)**. Якщо компоненти позначені (або названі), ми називаємо їх записами **(records)**.

std::variant як безпечніше об’єднання

Тип-сумами є об’єднання та теговані об’єднання. У C++ ми маємо тип класу union, який представляє об’єднання, і шаблон класу std::variant, який представляє теговане об’єднання.

Проблеми об’єднання

Об’єднання за своєю суттю схильні до помилок через такі основні причини:

1) Об’єднання прості і не можуть повернути поточний тип.

Наприклад:

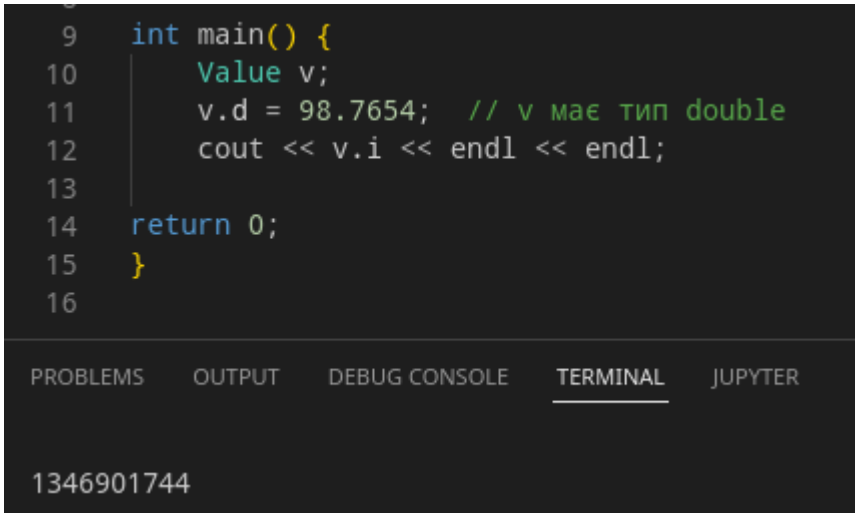
Тут визначене об'єднання з типом int та double. Але якщо користувач отримує доступ до int, це призводить до невизначеної поведінки, проте повідомлення про помилку не відображається.

```cpp
union Value{
    int i;
    double d;
};

int main() {
    Value v;
    v.d = 98.7654;  // v має тип double

return 0;
}

```



v містить значення типу double, але ми читаємо його як int (1346901744, яке є цілочисельним представленням бітового шаблону для 98,7654)

2) В об’єднаннях відсутній незалежний виклик конструктора і деструктора, коли змінюється тип.

Може бути визначений користувачем конструктор і деструктор.

Переваги std::variant

Визначення поточного типу, що зберігається у std::variant

std::variant є покращеним об’єднанням, його вважають “класом-обгорткою” для об’єднання з додатковою функціональністю.

std::variant забезпечує спостерігачів (поведінкові патерни): index() та valueless_by_exception()

index() повертає індекс типу, який активний зараз. Наприклад, екземпляр типу variant<int, string, double> може мати int з індексом 0, string з індексом 1 і double з індексом 2, і лише один з них буде активним у певний час.

valueless_by_exception() повертає false якщо у варіанті міститься значення. Якщо під час зміни типу варіанту виникає якийсь виняток, тоді ця функція корисна для визначення того, чи не матиме варіант значення.

Варіант має конструктор і деструктори

Друга причина переваги std::variant над union те, що псевдоконтейнер надає конструктор і деструктор. Конструктор надає різні способи побудови варіантного об’єкта. Деструктор дозволяє викликати відповідний деструктор для типу, який зараз активний.

Пам’ять

Об'єднання економлять пам'ять, оскільки дозволяють використовувати одну частину пам'яті для різних типів об'єктів у різний час. Отже, їх можна використовувати для збереження пам’яті, коли у нас є кілька об’єктів, які ніколи не використовуються одночасно.

std::variant використовує пам'ять, подібно до об'єднання - він займає максимальний розмір основних типів. Однак std::variant займе трохи додаткового місця для всіх типобезпечних функцій, які ми надаємо. Але додаткова пам’ять витрачена, об’єктивно, з користю.

Ще одна особливість в тому, що std::variant динамічно не виділяє пам’ять для значень. Будь-який екземпляр std::variant у будь-який момент часу або містить значення одного зі своїх альтернативних типів, або не містить значення взагалі. Значення, що міститься, розміщується в пам’яті варіантного об’єкта й не використовуватиме додаткову пам’ять, наприклад динамічну пам’ять, для виділення значення. Значення, що міститься, буде відповідно вирівняно для всіх типів.

Використання

Створення

Щоб створити variant який містить int або string. Змінна int початково має значення 0.

std::variant<int, std::string> var;

ми можемо задати такі значення, як

```
std::variant<int, double> var{1};  // значення типу int
var = 2.34;  // значення типу double
```

Крім того, серед інших є конструктори копіювання та конструктор переміщення.

```cpp
#include <variant>
#include <string>
#include <iostream>

using namespace std;


int main() {

    variant<string, int> var{"str"};
    variant<string, int> u = var;
    variant<string, int> v = variant<string, int>("str");

    return 0;
}
```

Ще деякі важливі функції

index()

Вже згадувана функція index() повертає індекс типу, активного в даний момент.

наприклад:

```cpp
int main() {

    variant<int, string, double> var = "str";
    cout << var.index() << endl;
    var = 123;
    cout << var.index() << endl << endl;

    return 0;
}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

1
0
```

valueless_by_exception()

Ще одна згадування функція valueless_by_exception() повертає false, якщо змінна має значення.

наприклад:

```cpp
#include <variant>
#include <string>
#include <iostream>

using namespace std;

struct S {
    operator int() { throw 42; }
};

int main() {

variant<float, int> v{12.0f};
cout << boolalpha << v.valueless_by_exception() << endl; // false
try{
    v.emplace<1>(S()); // v може не мати значення
}
catch(...){
}
cout << v.valueless_by_exception() << endl << endl;  // true якщо не містить значення

    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

false
false
```

visit()

visit() allows us to access the contents of a variant. Він приймає “відвідувача”, який є поліморфною функцією.

std::variant<int, std::string> v{78};
visit([](auto&& elem){std::cout << elem << endl; }, v);

У прикладі використана поліморфна функція лямбда.

holds_alternative()

holds_alternative() дозволяє визначити, чи екземпляр std::variant містить значення альтернативних типів.

приклад:

```cpp
#include <variant>
#include <string>
#include <iostream>

using namespace std;

int main() {
variant<int, string> var = "abc";
    cout << boolalpha
        << holds_alternative<int>(var) << endl  // false
        << holds_alternative<string>(var) << endl << endl;  // true


    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
false
true
```

get<T>()

get<T> надає доступ до значення залежно від індексу певного типу.

Приклад доступу до значення за допомогою індексу та за допомогою типу:

```cpp
int main() {
variant<int, float> v{99};

// доступ за індексом
cout << (get<0>(v)) << endl;  // 99

// доступ за типом
cout << (get<int>(v)) << endl << endl;  // 99

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
99
99
```

Приклад

Ще один загальний приклад використання псевдоконтейнеру variant, функцій, які він надає та особливостей його поведінки:

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

int main()
{
    cout << boolalpha;                              // виводить "true" або "false" замість 1 та 0

    variant<int, string> var;

    cout << holds_alternative<int>(var) << endl;    // true: містить int
    cout << (var.index() == 0) << endl;             // true: перший (нульовий) тип (int) активний
    cout << get<int>(var) << endl;                  // 0: var початково ініціалізується як int 0


    variant<int, string> var2{"hello"};             // визначаємо string як "hello"
    cout << (var2.index() == 1) << endl;            // true

    if (holds_alternative<string>(var2)) {          // true
        cout << get<string>(var2) << endl << endl;  // "hello"
    }

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
true
true
0
true
hello
```

Заключення:

Як ми побачили, об’єкт std::variant зберігає та керує часом життя значення змінної, зберігає альтернативні типи. Це дозволяє нам визначити поточний тип змінної, використовуючи функції, а також дозволяє конструювати та знищувати об’єкт. Таким чином, це додає більше функціональності об’єднанню (union) і, отже, є його безпечнішою версією.