

**Пользовательские
типы данных:
Структуры
Объединения**

Языки программирования C/C++ поддерживает определяемые пользователем структуры – структурированный тип данных. Он является *сбором одного или более объектов (переменных, массивов, указателей, других структур и т.д.), которые для удобства работы с ними сгруппированы под одним именем.*

Структуры:

- облегчают написание и понимание программ.
- помогают сгруппировать данные, объединяемые каким-либо общим понятием.
- позволяют группу связанных между собой переменных использовать как множество отдельных элементов, а также как единое целое.

Структуры в C++ обладают практически теми же возможностями, что и классы, но чаще их применяют просто для логического объединения связанных между собой данных. В структуру, в противоположность массиву, можно объединять данные различных типов.

Как и массив, структура представляет собой совокупность данных, но отличается от него тем, что к ее элементам (компонентам) необходимо обращаться по имени и ее элементы могут быть различного типа. Структуры целесообразно использовать там, где необходимо объединить данные, относящиеся к одному объекту.

Определение структуры состоит из двух шагов:

- объявление шаблона структуры (задание нового типа данных, определенного пользователем);
- определение переменных типа объявленного шаблона.

Объявление шаблонов структур. Общий синтаксис объявления *шаблона структуры*:

```
struct имя_шаблона
{
    тип1 имя_переменной1;
    тип1 имя_переменной1;
    //другие члены данных;
};

struct DataBase
{
    char fam[20];
    char name[15];
    long TelNumber;
    char *Adress;
    double w;
};
```

Имена *шаблонов* должны быть *уникальными* в пределах их *области определения* для того, чтобы компилятор мог различать различные типы шаблонов. Задание шаблона осуществляется с помощью ключевого слова **struct**, за которым следует имя шаблона структуры и список элементов, заключенных в фигурные скобки.

Имена элементов в *одном шаблоне* также должны быть *уникальными*. Однако в разных шаблонах можно использовать одинаковые имена элементов.

Допускаются и другие варианты описания структурных переменных. Можно вообще *не задавать имя типа*, а описывать сразу переменные:

```
struct {char fam[30];  
int kurs;  
char grup[3];  
float stip;  
} studi, stud2, *pst;
```

В этом примере кроме двух переменных структурного типа объявлен указатель `pst` на такую структуру. В данном описании можно было сохранить имя структурного типа `student`.

Например, требуется обрабатывать информацию о расписании работы конференц-зала, и для каждого мероприятия надо знать время, тему, фамилию организатора и количество участников. Поскольку вся эта информация относится к одному событию, логично дать ему имя, чтобы впоследствии можно было к нему обращаться. Для этого описывается новый тип данных (обратите внимание на то, что после описания стоит точка с запятой):

```
struct Event {  
    int hour, min;  
    char theme[100], name[100];  
    int num;  
};
```

Имя этого типа данных – *Event*. Можно описать переменные этого типа точно так же, как переменные встроенных типов, например:

```
Event e1, e2[10]; // структура и массив структур
```

Для **инициализации структуры** значения ее элементов перечисляют в фигурных скобках в порядке их описания:

```
struct{
char fio[30];
int date, code;
double salary;
}worker = {"Страусенко", 31, 215, 3400.55};
```

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов):

```
struct complex{
float real, im;
} compl [2][3] = {
{{1. 1}, {1. 1}, {1. 1}}, // строка 1, то есть массив compl[0]
{{2. 2}, {2. 2}, {2. 2}} // строка 2. то есть массив compl[1]
};
```

Переменные структурного типа можно размещать и в динамической области памяти, для этого надо описать указатель на структуру и выделить под нее место:

```
Event *pe = new Event; // структура  
Event *pm = new Event[m]; // массив структур
```

Элементы структуры называются полями. Поля могут быть любого основного типа, массивом, указателем, объединением или структурой. Для обращения к полю используется операция выбора («точка» для переменной и `->` для указателя), например:

```
e1.hour = 12; e1.min = 30;  
strcpy(e2[0].theme, "Выращивание кактусов в условиях Крайне-  
го Севера", 99);  
pe->num = 30; // или (*pe).num = 30;  
pm[2].hour = 14; // или (*(pm + 2)).hour = 14;
```

Задание только шаблона *не влечет* резервирования памяти компилятором.

Шаблон представляет компилятору необходимую информацию об элементах структурной переменной для *резервирования места* в оперативной памяти и организации доступа к ней *при определении* структурной переменной и использовании отдельных элементов структурной переменной.

Среди членов данных структуры могут также присутствовать, кроме стандартных типов данных (**int**, **float**, **char** и т.д.), ранее определенные типы, например:

```
/* объявление шаблона структуры типа date */
```

```
    struct date
```

```
    { int day, month, year; };
```

```
/* шаблон структуры person */
```

```
    struct person
```

```
    {
```

```
        char fam[30], im[20], otch[15];
```

```
        float weight;
```

```
        int height;
```

```
        struct date birthday;
```

```
    };
```

Структура *date* имеет три поля типа **int**. Шаблон структуры *person* в качестве элемента включает поле *birthday*, которое, в свою очередь, имеет ранее объявленный тип данных: **struct date**. Этот элемент (*birthday*) содержит в себе все компоненты шаблона **struct date**.

Доступ к компонентам структуры. Доступ к полям осуществляется с помощью оператора «.» при непосредственной работе со структурой или «->» - при использовании указателей на структуру. Эти операторы называются *селекторами* членов класса. Общий синтаксис для доступа к компонентам структуры следующий:

```
имя_переменной_структуры.член_данных;  
имя_указателя->имя_поля;  
(*имя_указателя).имя_поля;
```

Пример:

Прямой доступ к элементам

- 1) date1[5].day=10;
- 2) date1[5].year=1991;
- 3) **strcpy**(book1.title, "Война и мир");

/ используя прямое обращение к элементу, присваиваем значение выбранной переменной. Текст помещается в переменную, используя функцию копирования – strcpy(); */*

- 4) stud[3].birthday.month=1;
- 5) stud[3].birthday.year=1980;

Доступ по указателю

1) `(date1+5)->day=10;`

2) `(stud+3)->birthday.month=1;`

// Используя доступ по указателю на структуру, присваиваем значение соответствующей переменной. Указатель можно использовать и так:

3) `*(date1+5).day=10;`

4) `*(stud+3).birthday.month=1;`

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

```
struct building //Создаем структуру!
{
    char *owner; //здесь будет храниться имя владельца
    char *city; //название города
    int amountRooms; //количество комнат
    float price; //цена
};
```

```
int main()
{
```

```
    setlocale (LC_ALL, "rus");
```

```
    building apartment1; //это объект структуры с типом данных, именем структуры, building
```

```
    apartment1.owner = "Денис"; //заполняем данные о владельце и т.д.
```

```
    apartment1.city = "Симферополь";
```

```
    apartment1.amountRooms = 5;
```

```
    apartment1.price = 150000;
```

```
    cout << "Владелец квартиры: " << apartment1.owner << endl;
```

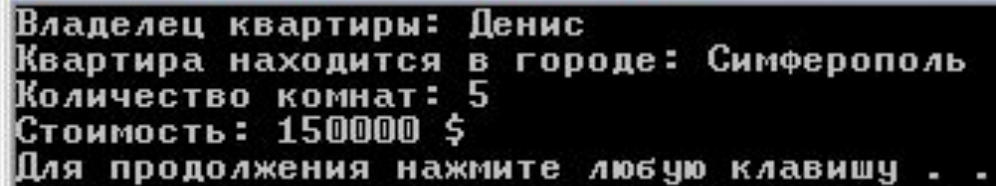
```
    cout << "Квартира находится в городе: " << apartment1.city << endl;
```

```
    cout << "Количество комнат: " << apartment1.amountRooms << endl;
```

```
    cout << "Стоимость: " << apartment1.price << " $" << endl;
```

```
    return 0;
```

```
}
```



```
Владелец квартиры: Денис
Квартира находится в городе: Симферополь
Количество комнат: 5
Стоимость: 150000 $
Для продолжения нажмите любую клавишу . .
```


Инициализация структур. При определении структурных переменных можно инициализировать их поля. Эта возможность подобна инициализации массива и следует тем же правилами:

имя_шаблона имя_переменной_структуры = {значение1, значение2, ...};

Компилятор присваивает *значение1* первой переменной в структуре, *значение2* – второй переменной структуры и т.д., и тут необходимо следовать некоторым правилам:

- присваиваемые значения должны совпадать по типу с соответствующими полями структуры;
- можно объявлять меньшее количество присваиваемых значений, чем количество полей.

Компилятор присвоит нули остальными полями структуры;

- список инициализации последовательно присваивает значения полям структуры, вложенных структур и массивов.

Пример:

```
struct date
```

```
{ int day,month,year; }d[5] = { {1,3,1980}, {5,1,1990}, {1,1,1983} };
```

Пример . Ввести сведения об N студентах. Определить фамилии студентов, получающих самую высокую стипендию.

```
#include <stdio.h>

#include <conio.h>
void main()
{
    const N=30; int i; float maxs;
    struct student {char fam[15];
                    int kurs;
                    char grup[3];
                    float stip;
                };
    student stud[N];
    clrscr();
    for(i=0;i<N;i++)
    { printf("%d-й студент",i);
      printf("\n"Фамилия:");scanf("%s",&stud[i].fam);
      printf("Курс:"); scanf("%d",&stud[i].kurs);
      printf("Группа:"); scanf("%s",&stud[i].grup);
      printf("Стипендия:"); scanf("%f",&stud[i].stip);
    }
    maxs=0;
    for(i=0;i<N;i++)
        if(stud[i].stip>maxs) maxs=stud[i].stip;
    printf("\n Студенты,получающие максимальную
    стипендию %f руб.",maxs);
    for(i=0; i<N; i++)
        if(stud[i].stip==maxs) printf("\n%s",stud[i].fam);
}
```

Рассмотрим пример, демонстрирующий сочетание массивов и структур

```
1  #include <iostream>
2  using namespace std;
3
4  struct PlayerInfo {
5      int skill_level;
6      string name;
7  };
8  using namespace std;
9
10 int main() {
11     // как и с обычными типами, вы можете объявить массив структур
12     PlayerInfo players[5];
13     for (int i = 0; i < 5; i++) {
14         cout << "Please enter the name for player : " << i << '\n';
15         // сперва получим доступ к элементу массива, используя
16         // обычный синтаксис для массивов, затем обратимся к полю структуры
17         // с помощью точки
18         cin >> players[ i ].name;
19         cout << "Please enter the skill level for " << players[ i ].name << '\n';
20         cin >> players[ i ].skill_level;
21     }
22     for (int i = 0; i < 5; ++i) {
23         cout << players[ i ].name << " is at skill level " << players[i].skill_level <<
24     }
25 }
```

1. Структура «Автосервис»: регистрационный номер автомобиля, марка, пробег, мастер, выполнивший ремонт, сумма ремонта.
2. Структура «Сотрудник»: фамилия, имя, отчество; должность; год рождения; заработная плата.
3. Структура «Государство»: название; столица; численность населения; занимаемая площадь.
4. Структура «Человек»: фамилия, имя, отчество; домашний адрес; номер телефона; возраст.
5. Структура «Читатель»: Фамилия И.О., номер читательского билета, название книги, срок возврата.
6. Структура «Школьник»: фамилия, имя, отчество; класс; номер телефона; оценки по предметам (математика, физика, русский язык, литература).
7. Структура «Студент»: фамилия, имя, отчество; домашний адрес; группа; рейтинг.
8. Структура «Покупатель»: фамилия, имя, отчество; домашний адрес; номер телефона; номер кредитной карточки.
9. Структура «Пациент»: фамилия, имя, отчество; домашний адрес; номер медицинской карты; номер страхового полиса.
10. Структура «Информация»: носитель; объем; название; автор.
11. Структура «Клиент банка»: Фамилия И.О., номер счета, сумма на счете, дата последнего изменения.
12. Структура «Склад»: наименование товара, цена, количество, процент торговой надбавки.
13. Структура «Авиарейсы»: номер рейса, пункт назначения, время вылета, дата вылета, стоимость билета.
14. Структура «Вокзал»: номер поезда, пункт назначения, дни следования, время прибытия, время стоянки.
15. Структура «Кинотеатр»: название кинофильма, сеанс, стоимость билета, количество зрителей.

ОБЪЕДИНЕН

Объединения очень похожи на структуры, однако способ, с помощью которого C++ хранит объединения, отличается от способа, с помощью которого C++ хранит структуры.

union — это пользовательский тип, в котором все члены используют одну область памяти.

Это означает, что в любой момент времени объединение *не может* содержать *больше одного объекта* из списка своих членов.

Независимо от количества членов объединения, оно использует лишь количество памяти, необходимое для хранения своего крупнейшего члена.

Объединение состоит из частей, называемых **элементами (членами)**.

Объединения могут быть полезны для экономии памяти при наличии множества объектов и/или ограниченном количестве памяти. Однако для их правильного использования требуется повышенное внимание, поскольку нужно всегда сохранять уверенность, что используется последний записанный элемент.

СИНТАКСИС

```
union [name] { member-list };
```

Параметры

name

Имя типа, присваиваемое объединению.

member-list

Члены, которые могут входить в объединение.См. заметки.

Объявление объединения

Объявление объединения начинается с ключевого слова **union** и содержит список членов, заключенный в фигурные скобки:

```
// declaring_a_union.cpp
union RecordType    // Declare a simple union type
{
    char  ch;
    int   i;
    long  l;
    float f;
    double d;
    int *int_ptr;
};
int main()
{
    RecordType t;
    t.i = 5; // t holds an int
    t.f = 7.25 // t now holds a float
}
```

Внутри программ **объединения** C++ очень похожи на **структуры**. Например, следующая структура определяет **объединение** с именем `distance`, содержащее два элемента:

```
union distance
{
    int miles;
    long meters;
};
```

Как и в случае со структурой, описание объединения не распределяет память. Вместо этого описание предоставляет шаблон для будущего объявления переменных.

Следующая программа иллюстрирует использование объединения `distance`. Сначала программа присваивает значение элементу `miles` и выводит это значение. Затем программа присваивает значение элементу `meters`. При этом значение элемента `miles` теряется:

```
#include <iostream.h>
void main(void)
{
    union distance
    {
        int miles;
        long meters;
    } walk;
    walk.miles = 5;
    cout << «Пройденное расстояние в милях » << walk.miles << endl;
    walk.meters = 10000;
    cout << «Пройденное расстояние в метрах » << walk.meters << endl;
}
```

*Программа обращается к элементам объединения с помощью **точки**, аналогичная запись использовалась при обращении к элементам структуры*

Есть возможность сделать объединение безымянным:

```
struct STRX {  
    int j;  
    union {  
        int i;  
        double a;  
    };  
} a;
```

При этом обращение упрощается:

```
a.i = 123;  
a.a = 4.5;
```

Использование объединений позволяет экономит память:

```
1 // Объединение
2 union A {
3     int a;
4     float b;
5     double c;
6 };
7
8 // Структура
9 struct B {
10    int a;
11    float b;
12    double c;
13 };
```

`sizeof(union A)` будет равен `sizeof(double)`, а

`sizeof(struct B)` будет равен `sizeof(int) + sizeof(float) + sizeof(double)`.

Объединение хранит значение только одного элемента в каждый момент времени

Объединение представляет собой структуру данных, которая, подобно структуре C++, позволяет вашим программам хранить связанные части информации внутри одной переменной. Однако в отличие от структуры объединение хранит значение только одного элемента в каждый момент времени. Другими словами, когда вы присваиваете значение элементу объединения, вы перезаписываете любое предыдущее присваивание.

Объединение определяет шаблон, с помощью которого ваши программы могут позднее объявлять переменные. Когда компилятор C++ встречает определение объединения, он распределяет количество памяти, достаточное для хранения только самого большого элемента объединения.