Сложные типы данных в простых примерах

Массивы и строки

Предположим нам необходимо написать программу выводящую на экран сумму двух векторов. Напомним, что суммой двух векторов является сумма их одноименных координат, т.е (x1, y1) + (x2, y2) = (x1 + x2, y1 + y2). Реализуем эти вычисления в программе

```
#include <stdio.h>

//Функция считающая и печатающая на экран сумму координат векторов.
void print_sum(int x1, int y1, int x2, int y2)
{
    printf("result = (%d, %d) \n", (x1 + x2), (y1 + y2));
}

int main()
{
    int x1 = 2, y1 = 3, x2 = 1, y2 = 0;

    print_sum(x1, y1, x2, y2);

    return 0;
}
```

Данная программа отлично решает поставленную задачу, однако её масштабируемость вызывает вопросы. Так как вектора существуют не только в двумерном пространстве, то хотя бы для трёхмерных векторов пришлось бы увеличивать количество аргументов функции, увеличивать количество параметров вызова, вводить дополнительные переменные. Отвязываясь, от геометрического смысла векторов, мы знаем, что нет никаких ограничений для размерности вектора. Поэтому хорошо бы иметь сущность, позволяющую хранить множество однотипных данных.

Такой сущностью являются массивы. Массивы представляют собой строго фиксированное количество данных одного типа последовательно расположенных в памяти.

Наша программа с использованием массивов будет выглядеть так

```
#include <stdio.h>

//int first [] - обозначает передачу массива в функцию, как параметра.
void print_sum (int first [], int second [])
{
    printf("result = (%d, %d) \n", first[0] + second[0], first[1] + second[1]);
}

int main()
{
    int a[2], b[2]; //Объявление 2 массивов, состоящих из 2 целых чисел.
    a[0] = 1;      //Определение значения нулевого элемента (первого по порядку)
    a[1] = 2;

    b[0] = 2;
    b[1] = 1;

    print_sum(a, b); //Вызов функции печати. Аргументы – объявленные массивы.
    return 0;
}
```

В общем случае объявление массива выглядит так ТИП ИМЯ [РАЗМЕР].

Получать доступ к элементу массива, как для чтения так и для модификации мы можем обращаясь по его индексу в квадратных скобках (нумерация начинается с нуля).

ИМЯ [ИНДЕКС] .

Индекс не должен превышать размер массива или быть равным размерности. Выход за пределы массива приводит в лучшем случае к неопределенному поведению, но как правило, к краху программы.

Так же допустим следующий вариант определения значений массива при помощи инициализации (будем считать инициализацией объявление и определение переменной в одном выражении)

```
    int a[2] = { 1, 2};
    int b[2] = { 2, 1};
```

Модифицируем нашу программу таким образом, чтобы она могла работать с массивами любой заранее известной длины.

```
#include <stdio.h>

#define N 5 //Определяем значение N как число 5

void print_sum (int first [], int second [])
{
    printf("result = (");
    for(int i = 0; i < N; i++)
        printf("%d, ", first[i] + second[i]); //Считываем и выводим в цикле.

    printf(")\n");
}

int main()
{
    int a[N], b[N];

    printf("Enter %d elements.\n ", N);

    //Ввод данных в цикле.
    for(int i = 0; i < N; i++)
        scanf("%d", &a[i]);

    printf("Enter %d elements.\n ", N);
    for(int i = 0; i < N; i++)
        scanf("%d", &b[i]);

    print_sum(a, b);

    return 0;
}
```

#define N 5 — Означает, что на стадии работы препроцессора все встречания в коде отдельного символа N будут заменены на число 5.

Таким образом при помощи циклов мы можем обрабатывать массивы любой заранее известной длины.

Строки. Краткое введение.

Строки в Си представляют собой массив объектов типа char, заканчивающийся символом '\0' или значением (0).

На этом простом примере можно убедиться в необходимости завершающего символа.

```
#include <stdio.h>
#define N 5

int main()
{
    char str[N];

    for(int i = 0; i < N; i++)
    //for(int i = 0; i < N - 1; i++)
    {
        scanf("%c", &str[i]);
    }
    //str[N - 1] = '\0';

    printf("%s\n", str); //%s – формат для вывода строки.

    return 0;
}
```

Расскоментировав условие цикла `for(int i = 0; i < N - 1; i++)` и определение последнего элемента массива `str[N - 1] = '\0';` , мы получим корректный вывод на экран.

Строки так же можно инициализировать в программном коде значениями, заключенными в двойные кавычки, так называемыми строковыми литерами. Обратите внимание, такая инициализация возможна только при объявлении строки.

```
#include <stdio.h>
#define N 5

int main()
{
    char str[N] = "abcd";
    //char str[N] = "abcdefg"; //В строку попадет только N-1 символов

    //str = "wxyz"; //ОШИБКА! Объявленной строке нельзя присвоить строковый
литерал.

    printf("%s\n", str);

    return 0;
}
```

Структуры. Введение.

Предположим нам необходимо написать программу выводящую информацию о неком ученике, имеющим имя и возраст. Воспользовавшись имеющимися знаниями для написания такой программы, мы получим такой код.

```
#include <stdio.h>
#define N 100

void print_student_info(char name[], char surname[], int age)
{
    printf("name = %s\nsurname=%s\nage=%d\n",name, surname, age);
}

int main()
{
    char name[N] = "Ivan";
    char surname[N] = "Ivanov";
    int age = 24;

    print_student_info(name, surname, age);
}
```

Очевидно, что когда ученик один, проблем не возникает, но если нам придётся оперировать информацией о нескольких учениках количество переменных будет стремительно возрастать. Тем более мы можем отметить, что имя фамилия и возраст описывают одну сущность — ученик и, было бы намного удобнее оперировать этой информацией как единым объектом. Иными словами, объединить в одну сущность данные различного типа. Такое средство есть и называется структура. Рассмотрим модифицированный пример.

```
#include <stdio.h>
#include <string.h>

#define N 100

//Объявление структуры.
//1. Ключевое слово struct
//2. Имя структуры.
//3. Фигурные скобки.

struct student {
    char name[N]; //4. В скобках объявляем объекты структуры (как будто
переменные).
    char surname[N];
    int age;
}; //5. После фигурных скобок точка с запятой.

//На вход функции передаем объект структуры.
//К данным структуры можно обращаться через точку
void print_info(struct student st)
{
    printf("name=%s\nsurname=%s\nage=%d", st.name, st.surname, st.age);
}


int main()
{
    char name[N] = "Ivan";
    struct student ivan;      //Создаем объект структуры

    //Присваивать строковый литерал мы не можем, напишем цикл для инициализации.
    int i = 0;    //Счётчик символов.
    while(name[i] != '\0') //Пока не встретим сивол завершения строки.
    {
        ivan.name[i] = name[i]; //Копируем символы.
        i++; //Увеличиваем счётчик.
```

```
    }
    ivan.name[i] = '\0'; //Добавляем завешающий символ '\0'

    //Инициализации при помощи стадартной функции.
    //Подробности можно почитать man strcpy
    strcpy(ivan.surname, "Ivanov");
    ivan.age = 24;
    print_info(ivan); //Передача объекта в качестве параметра вызова.

    //Ещё один вариант инициализации объектов структур при помощи значений в
фигурных скобках.
    struct student petr = {.name = "Petr", .surname = "Petrov" , .age = 10};
    print_info(petr);
}
```

Допустим, мы хотим написать функцию поздравляющую ученика с днем рождения и увеличивающую его возраст. Например:

```
void happy_birthday( struct student stud)
{
    printf("Happy birthday\n");
    stud.age = stud.age + 1;
}
```

К сожалению, если мы попробуем использовать такую функцию, она работать не будет. Возраст увеличен не будет. Это происходит потому, что в функцию передаётся копия объекта структуры. Как сделать так, чтобы изменения коснулись исходного объекта?

Указатели.

В данном случае на помощь приходят указатели. Каждый байт в компьютере имеет свой адрес. Программа в процессе выполнения занимает определенную область адресов. Узнать адрес переменной можно при помощи операции &.

```
#include <stdio.h>

int main()
{
    int a = 3;
    printf("address of a = 0x%llx", &a); //Компилятор выдаст тут предупредение.

    return 0;
}
```

Запуская программу мы будем видеть, что переменная «a» всегда занимает различные адреса.

Указатель по своей сути — это переменная, хранящая адрес обеъекта в памяти. Объявляется как ТИП * ИМЯ; (Указатель на переменную типа ТИП).

```
#include <stdio.h>

int main()
{
    int a = 3;
    int *a_ptr = &a;
    printf("address of a = 0x%llx", a_ptr); //Компилятор выдаст предупредение.

    return 0;
}
```

Одно из частых применений указателей — это передача переменной в функцию по адресу, что позволяет изменять исходное значение переменной в функции (как разт о, что у нас не

получилось со структурами).

```
#include <stdio.h>

//В качестве аргумента функция принимает указатель на int.
void increment(int * input)
{
    // Операция *Имя_указателя – есть получение значения переменной по адресу, храняющемуся в указателе.
    *(input) = *(input) + 1;
}

int main()
{
    int a = 3;
    int *a_ptr = &a;

    printf("a = %d", a);
    increment(&a);
    printf("a = %d", a);

    return 0;
}
```

Воспользуемся полученными знаниями для модификации кода функции, поздравляющей студента с днём рождения.

```
//В качестве аргумента теперь указатель на переменную stud
void happy_birthday( struct student * stud)
{
    printf("Happy birthday\n");

/*
Важно отметить, что stud теперь не объект структуры, а указатель на объект структуры. По этой причине обращение к полям структуры через «.» невозможно. В случае указателей применяется операция «->».
*/
    stud->age = stud->age + 1;
}
```

Теперь эта функция работает правильно.

Небольшое замечание по поводу массивов.

Массивы в функции всегда передаются по указателю на первый элемент.

Иными словами в функцию

```
void foo(int a[]) { … }
```

При вызове будет передан указатель на первый элемент массива.

```
int a[10] = { 0 }; //Инициализация массива нулями.
foo(a); //В foo будет передан указатель хранящий адрес первого элемента (&a[0]).
```

В этом смысле запись в объявлении аргументов функции `int a[]` эквивалентна `int * a`.