Лабораторная работа №11

Структуры и битовые поля.

Цель работы: Изучение теоретических сведений и получение практических навыков по работе со структурами, объединениями и битовыми полями.

Теоретические сведения

Структура в языке Си представляет собой набор переменных, в общем случае различных типов, которые для удобства работы с ними объединены под одним именем. Структуры позволяют организовать сложные данные, позволяя объединять связанные между собой переменные в группу, оперируя ими как единым целым. Объявление структуры начинается с ключевого слова `struct`:

```
struct structure tag
{
   member definition;
   member definition;
   ...
   member definition;
} list of variables;
```

и содержит список объявлений переменных, заключенных в фигурные скобки. За ключевым словом `struct` может следовать имя, называемое *тегом структуры* (от англ. *tag* - ярлык). Тег позволяет именовать структуру данного вида и в дальнейшем служит ее кратким обозначением. Перечисленные в фигурных скобках переменные являются элементами (*members* - члены структуры). Имена как элементов, так и тегов могут совпадать с именами переменных в главной функции `main`(). Кроме того, одинаковые имена элементов могут встречаться в разных структурах. Объявление структуры определяет новый пользовательский тип данных, поэтому за закрывающей фигурной скобкой, может следовать список переменных.

Примером структуры может выступать зачетная книжка студента. Она содержит такие сведения о студенте, как его полное имя, номер зачетной книжки, группу, названия предметов и оценки по этим предметам и т.д. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонент: фамилия, имя и отчество. Другим примером структуры может быть комплексное число – точка в комплексной плоскости есть пара координат: Re и Im:

Объявление структуры:

```
struct Complex
{
      float Re;
      float Im;
};
```

Объявление структуры со структурными переменными:

```
struct Complex
{
      float Re;
      float Im;
} a, b, c;
```

```
int main()
{
  struct Complex a, b, c;
  return 0;
}
```

Объявление структуры, которое не содержит объявления списка структурных переменных, не резервирует памяти (см. пример объявления слева). В данном случае объявление структуры просто описывает некоторый шаблон. Также при определении структурных переменных в теле функции `main()` можно пользоваться структурным тегом:

```
  struct Complex a, b, c;
```

Основными действиями, которые можно выполнять со структурами, являются: копирование, передача их функциям в качестве аргументов, а возврат структур из функции в качестве результатов.

Инициализация переменных структурного типа

Для структурных переменных также допускается инициализация при их объявлении. Для инициализации структурных переменных существует несколько способов. Во-первых, структурные переменные можно инициализировать при объявлении:

```
struct Complex a={1.0,3.2}, b;
```

Также можно инициализировать структуры посредством использования оператора точка (.), обращаясь к конкретным элементам структуры:

```
      b.Im=10.0;
      b.Re= 1.0;
```

Таким образом, точка (.) – это оператор доступа к элементу структуры, которая соединяет имя структуры и имя элемента.

Структуры и функции

Чтобы понять, как взаимодействуют структуры и функции, напишем несколько функций, описывающих действия над комплексными числами. Существует, по крайней мере, три возможности передачи структурных переменных функциям: передавать элементы структуры по отдельности (как обычные переменные), передавать структурные переменные и передавать указатели на структурные переменные.

Первая функция, *Arg*, рассчитывающая аргумент комплексного числа, получает в качестве аргумента структурную переменную и возвращает действительное число.

```
/* Arg: вычисляет аргумент комплексного числа */
double Arg(struct Complex num)
{
    if (num.Re==0 && num.Im<0)
        return –M_PI_2;
    else if (num.Re==0 && num.Im>0)
        return M_PI_2;
    else
        return atan(num.Im/num.Re);
}
```

Отметим, что для корректной работы данной функции необходимо подключить математическую библиотеку math.h. Использованная в функции константа M_PI_2 объявляет константу $\pi/2$, также зарезервированную в библиотеке math.h.

Следующая функция позволяет вычислить сумму комплексных чисел:

```
struct Complex Sum(struct Complex a, struct Complex b)
{
    struct Complex tmp;
    tmp.Re=a.Re+b.Re;
    tmp.Im=a.Im+b.Im;
    return tmp;

}
```

В качестве возвращаемого функцией значения также выступает структурная переменная типа `struct Complex`.

Указатели на структуры

При работе с большими структурами зачастую эффективнее передать в функцию указатель на структуру, чем всю структурную переменную целиком. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Объявление

```
struct Complex *pointer;
```

создает объект `pointer` – указатель на структуру типа `struct Complex`. Если `pointer` указывает на структуру `Complex`, то `*pointer` – это сама структурная переменная, а `(*pointer).Re` и `(*pointer).Im` – ее элементы. Используя указатель `pointer`, мы могли бы написать:

```
struct Complex z={10.0, 1.0}, *pointer;
pointer = &z;
printf("z: (%f,%f)\n", (*pointer).Re, (*pointer).Im);
```

Скобки в `(*pointer).Re` обязательны, поскольку приоритет оператора точка (.) выше, чем приоритет *. Выражение `*pointer.Re` будет проинтерпретировано как `*(pointer.Re)`, что неверно, поскольку `pointer.Re` не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к ее элементам была придумана более короткая форма записи (–>). Если `pointer` – указатель на структуру, то

```
pointer –>элемент-структуры;
```

ее отдельный элемент. Поэтому `printf` можно переписать в виде

```
printf("z: (%f,%f)\n", pointer -> Re, pointer -> Im);
```

Операторы (.) и (**->**) выполняются слева направо. Операторы доступа к элементам структуры (.) и (**->**) вместе с операторами вызова функции () и индексации массива **[]** обладают самыми высокими приоритетами в иерархии приоритетов и выполняются раньше любых других операторов.

Объявление типов данных typedef

Язык Си предоставляет возможность вводить сокращенные обозначения для пользовательских типов данных с помощью ключевого слова `typedef`. Например, объявление

```
typedef struct Complex Complex;
```

делает имя Complex синонимом struct Complex. С этого момента тип Complex можно применять в объявлениях, точно так же, как тип struct Complex:

```
Complex x, y;
```

Заметим, что объявляемый в typedef тип стоит на месте имени переменной в обычном объявлении, а не сразу за словом typedef. Следует подчеркнуть, что объявление typedef не создает объявления нового типа, оно лишь сообщает новое имя уже существующему типу. Никакого нового смысла эти новые имена не несут, они объявляют переменные в точности с теми же свойствами, как если бы те были объявлены напрямую без переименования типа.

Для применения typedef существуют две важные причины. Во-первых, правильная параметризация программы, как правило, связанная с проблемой переносимости. Если с помощью typedef объявить типы данных, которые являются машинно-зависимыми, то при переносе программы на другую машину потребуется внести изменения только в определения typedef. Одна из распространенных ситуаций - использование typedef-имен для варьирования целыми величинами. Для каждой конкретной машины это предполагает соответствующие установки short, int или long, которые делаются аналогично установкам стандартных типов. Вторая причина, побуждающая к применению typedef, - улучшение читаемости программы.

Пример Си-программы

```
#include <stdio.h>
#include <math.h>
struct Complex
{
  float Re;
  float Im;
};

typedef struct Complex Complex;

double Arg(Complex num)
{
  double tmp;
  if (num.Re==0 && num.Im<0)
      tmp=-M_PI_2;
```

```
  else if (num.Re==0 && num.Im>0)
      tmp=M_PI_2;
  else
    tmp=atan(num.Im/num.Re);
  return tmp;
}

Complex Sum(Complex a, Complex b)
{
  Complex tmp;
  tmp.Re=a.Re+b.Re;
  tmp.Im=a.Im+b.Im;
  return tmp;
}

int main()
{
  Complex a, b, c = {1.0,2.0},z,*pointer;
  printf("\nc(%f,%f)\n",c.Re,c.Im);
  printf("\nArg(%f+i%f)=%f\n",c.Re,c.Im,Arg(c));
  printf("\n Input real part of complex number a:");
  scanf("%f",&a.Re);
  printf("\n Input imaginary part of complex number a:");
  scanf("%f",&a.Im);
  printf("\n Input real part of complex number b:");
  scanf("%f",&b.Re);
  printf("\n Input imaginary part of complex number b:");
  scanf("%f",&b.Im);
  z=Sum(a,b);
  printf("\nSum (a,b)=%f+%fi\n",z.Re,z.Im);
  pointer = &z;
  printf("z: (%f,%f)\n", (*pointer).Re, (*pointer).Im);
  printf("z: (%f,%f)\n", pointer -> Re, pointer -> Im);
  return 0;
}
```

Битовые поля

В языке Си мы можем определить размер (в битах) для элементов структуры. Идея заключается в эффективном использовании памяти, когда мы знаем наверняка, что значение поля или группы полей никогда не превзойдет определенного предела (лежит в строго определенной области значений).

Рассмотрим в качестве примера определение структуры `Date`, которая позволяет задавать день, месяц и год в качестве элементов:

```
#include <stdio.h>
// Date representation
struct Date{
   unsigned int dd;
   unsigned int mm;
   unsigned int yy;
};

int main()
{
   printf("\n Size of Date %ld bytes\n",sizeof(struct Date));
   struct Date date={31,12,2016};
   printf("\n Date is %d/%d/%d\n",date.dd,date.mm,date.yy);
   return 0;
}
```

В результате работы программы можно определить, что размер памяти выделяемой под структуру будет составлять 12 байт (поскольку тип unsigned int занимает 4 байта)[1] . Данную структуру можно попытаться сделать более оптимальной, поскольку мы знаем, что величина числа в каждом месяце не превышает 31, а значение месяца не превышает 12. В результате представленное выше объявление структуры Date можно записать в виде:

```
struct Date{
   // dd has value between 1 and 35, thus it does not exceed 5 bits
   unsigned int dd: 5;
   // value of mm ranges from 1 to 12 while 4 bit is enough to store it
   unsigned int mm: 4;
   unsigned int yy;
};
```

[1] Результат зависит от архитектуры памяти конкретной ЭВМ.

В результате такого усовершенствования программы, размер выделяемой под структуру памяти сократится до 8 байт.

Обратите внимание на важные моменты, которые стоит помнить при работе с битовыми полями:

1. Специальные неименованные битовые поля при объявлении структуры используются для выравнивания поля по границе бита.

```
struct Date{
   unsigned int dd: 5;
   unsigned int: 0;
   unsigned int mm: 4;
   unsigned int yy;
};
```

2. Нельзя использовать указатели на элементы битовых полей, поскольку размещение элементов в памяти начинается не на границе байта.
3. Значения битовых полей будут интерпретироваться в зависимости от архитектуры компилятора.

```
#include <stdio.h>
struct test
{
   unsigned int x: 2;
   unsigned int y: 2;
   unsigned int z: 2;
};
int main()
{
   struct test t={5,5,5};
   printf("%d\t%d\t%d", t.x,t.y,t.z);
   return 0;
}
```

4. Элементы битовых полей не могут быть определены как статические (с использованием ключевого слова `static`).
5. Массивы не могут выступать в качестве элементов битовых полей. Попытка создания следующего битового поля:

```
struct test{
   unsigned int A[10]: 5;
```

```
};
```

приведет к следующему сообщению об ошибке от компилятора: bit-field ‘A’ has an invalid type.

Контрольные вопросы

1. Что такое структура? В чем заключается отличие структуры от массива?
2. Каким образом происходит объявление структуры? Приведите пример объявления структуры.
3. Каким образом можно объявить структурную переменную? Приведите примеры.
4. Для чего нужен тег структуры? Можно ли создавать структуры без тега? Приведите примеры.
5. Каким образом можно инициализировать структурные переменные? Приведите примеры.
6. Для каких целей используется оператор точка при работе со структурами?
7. Как передать структурную переменную в функцию?
8. Может ли структура быть возвращаемым функцией значением? Приведите примеры.
9. Как работать с указателями на структурные переменные? Приведите примеры.

10*. Для каких целей используется ключевое слово `typedef`?

11*. Что такое битовое поле? Какова связь между битовыми полями и структурами?

12*. Какие ограничения существуют при работе с битовыми полями?