Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

03 декабря 2014

Процедурные функции ввода-вывода

Состоят из функций и специализированных структур. Входят в стандартную библиотеку Си.

```
int printf(const char *, ...);
int sprintf(char *, const char *);
int fprintf(FILE *, const char *);

int scanf(const char *, ...);
int sscanf(const char *, const char *);
int fscanf(FILE *, const char *, ...);

int wprintf(const wchar_t *, ...);
int wscanf(const wchar_t *, ...);
```

Форматированный вывод

Функция printf формирует результирующую строку и выводит ее на экран. Принимает один обязательный параметр, который может быть либо текстом, либо специально сформированными спецификаторами.
Спецификаторы представляют из себя строку, содержащую символы %, которые указывают в строке позиции для подстановки не обязательных параметров.

```
#include <cstdio> //stdio.h

printf("Hello World");
```

Спецификаторы

```
%c //символ
%d //знаковое число
%f //float по умолчанию печатает 6 знаков после
    запятой
%e //double
%x //шестнадцатеричное число
%p //значение адреса
%s //печать строки (до завершающего нулевого
    символа)
%[flags][width][.percision][{h | l | ll }] type

int dig = 5;
float f = 0.3333;
const char * str = "Hello";
printf("digit=%d float=%f str=%s\n", dig, f, str);
```

Модификаторы

Модификаторы вывода: [-n] //задает количество знакомест под число, если число занимет больше позиций то усечения не происходит! Минус - выравнивание по левому краю, Плюс - по правому.
В случае, если число вещественное, то можно задать количество знакомест под целую и дробную часть [-n.m]. n — количество знакомест(целая часть + вещественная), включая символ точки.

```
printf("%6d\n%6d\n", 10, 10000);
printf("%-6d\n%-6d\n", 10, 10000);

printf("%f\n%f\n", 133.1, 3333.3333);
printf("%7.2f\n%7.2f\n", 133.1, 3333.3333);
```

Формирование строки в памяти

```
char buffer[256];

char name[] = "Petr";
int age = 20;

sprintf(buffer, "My␣name␣is␣%s,␣my␣age␣is␣%d\n",
    name, age); //опасно
snprintf(buffer, 256, "My␣name␣is␣%s,␣my␣age␣is␣%d\
    n", name, age); //не запишет более, чем 256
    байт в buffer
printf("%s", buffer);
```

Вывод в файл

Используя функцию fprintf, можно сделать вывод данных в файл, причем синтаксис использования функции не будет отличаться от функций printf и sprintf.

```
FILE *f = fopen("my.txt", "w");
if(f)
{
  int age;
  std::cin >> age;
  const char *str = "Vasia";
  fprintf(f, "My name is %s and my age: %d\n", str,
      age);
  fclose(f); //обязательно, иначе операционная
     система заблокирует дескриптор файла
}
```

FILE — специализированный указатель на структуру типа FILE. Этак структура предоставляется стандартной библиотекой.

Форматированный ввод

Осуществляется, со стандартного ввода stdin, при помощи функции scanf. Пользователь задает типы вводимых значений через спецификаторы по аналогии с printf. В функцию необходимо передать адреса переменных, т.к. функция scanf должна иметь возможность задать значения передаваемых в параметрах.

```
int d, q;
float fl;
char str[256];
scanf("%d␣%f␣%s␣%d", &d, &fl, str, &q);

scanf("%d␣%*f␣%s", &d, str); //означает, что
    параметр с типом float, не будет записан в
    переменную, т.к. указан спецификатор *
```

Дополнительно существует оператор *, который позволяет опускать вводимый параметр.

Защита при форматированном вводе

```
char str[5];
scanf("%s", str); //плохо! т.к. пользователь может
    ввсти больше, чем 4 символа.

scanf("%4s", str); //позволит ввести пользователю
    максимум 4 символа, плюс 1 байт памяти на
    нулевой символ.
fflush(stdin); //важно, если функция scanf не
    считала оставшиеся символы, то они будут
    оставлены в специальном буфере до следующего
    считывания. Поэтому буффер необходимо очистить.
```

Чтение из файла

```
FILE *f = fopen("myfile.txt", "r");
if(f)
{
   int x;
   char buffer[256];
   fscanf(f, "My␣name␣is␣%s␣and␣my␣age:␣%d\n",
       buffer, &x); //строка My name is считывается,
        но пропускается
   fclose(f);
}
```

Считывание из строки в память

```
char temp[64];
int number;
const char *str = "String is TEST and number is
    1000";
sscanf(str, "String is %s and number is %d", temp
    , &number);
```

Пользовательские типы данных

Основным типом для реализации пользовательских типов данных являются структуры. Зачем нужны структуры?

```
int d1, m1, y1;
int d2, m2, y2;
```

Набор данных описывает одну группу, хотя наборы между собой не связаны.

Структуры

Позволяют:

- Хранить совокупность характеристик, как единое целое.
- Манипулировать этой совокупностью, как единым целым.
- Получать возможность обращаться к характеристикам по отдельности.

Структуры

Включают:

- Существующие, ранее определенные типы (например, базовые типы).
- Произвольное количество данных, т.е. полей структуры.
- Поля содержащие: базовый тип, указатель, массив, структуру.

Объявление структуры

Это описание компилятору внутреннего устройства нового агрегатного типа данных, содержащее:

- Количество полей.
- Типы полей.
- Порядок расположения полей.

Объявление ДОЛЖНО располагаться в заголовочном файле, чтобы остальные файлы в программе могли им пользоваться.

```
struct имя_пользовательского_типа {
        список_полей_структуры (типы и имена полей);
}; //<--- обязательная ;
```

Синтаксис объявления

```
//date.h
struct MyType {
    int day;
    int month;
    int year;
    char info[20];
};

//main.cpp
#include "date.h"

int main()
{
    struct MyType t1; //Требуется в ANSI C
    MyType t2; //ключевое слово struct можно
        опустить в C++ и C99, а также в различных
        модификациях компиляторов.
}
```

Инициализация структурных переменных

```
#include "date.h"

namespace { MyType t; }
namespace QQQ { MyType t; }

int main()
{
    MyType t3;
    static MyType t2;
}
```

Подчиняется темже правилам, что и инициализация обычных переменных и массивов.

Обращение к полям структуры

```
#include "date.h"

int main()
{
    MyType t3;
    t3.day = 3;
    t3.month = 12;
    t3.year = 2014;
    t3.info = //FIXME
}
```

Ввод структур

```
#include "date.h"

int main()
{
    MyType element;
    scanf("%d␣%d␣%d␣%19s", &element.day,
                           &element.month,
                           &element.year,
                            element.info);

    MyType arr[5]; //FIXME для массива
}
```

typedef для структур

```
typedef struct { int day; /* ... */ } Date;

Date d1;
```

Можно использовать в ANSI C, чтобы убрать слово struct при использовании структур.

Явная инициализация

```
#include "date.h"

int main()
{
    MyType d1 = {10, 10, 2014, "Inform" };
    MyType d1 = {10, 10, 2014};
    MyType d4 = {0};
    d4 = {10, 11}; //???

    MyType arr[] = {
            {10, 10, 2014},
            {11, 11, 2015, "New"}
               };
}
```

Операции со структурами

```
#include "date.h"

int main()
{
    MyType d1 = {10, 10, 2014, "Inform" };
    MyType d2 = d1;
    MyType d3;

    d3 = d2;
}
```

Динамическая память

```
#include "date.h"

int main()
{
    MyType *pD1 = new MyType;

    (*pD1).day = 10;
    (*pD1).month = 10;
    (*pD1).info = "Hello"; //FIXME

    pdD1->year = 2000;

    delete pD1;

}
```

Динамическая память

```
#include "date.h"

int main()
{
  MyType *pD1 = new MyType[n]; //const int n = 10;

  for(int i = 0; i < n; i++)
  {
    printf("%d␣%d␣%d␣%s\n", pD1[i].day, pD1[i].
        month,
                  pD1[i].year, pD1[i].info);
  }

  delete [] pD1;

}
```

Динамический массив указателей

```
#include "date.h"

int main()
{
  int n; cin >>n;
  MyType *pD1 = new MyType* [n];

  for(int i = 0; i < n; i++)
      pD1[i] = new MyType;

  for(int i = 0; i < n; i++)
  {
      printf("%d %d %d %s\n", pD1[i]->day, pD1[i]->
          month,
                  pD1[i]->year, pD1[i]->info);
  }
  for(int i = 0; i < n; i++)
      delete pD1[i];
  delete [] pD1;
```

Упаковка полей структуры

Компилятор гарантированно отводит под структуру количество байтов больше или равное сумме размеров всех полей структуры.
Компилятор выделяет память в том порядке в котором поля объявлены в структуре.

```
struct A
{
    char a;
};

struct B
{
    char a;
    int b;
};

sizeof(A); //???
sizeof(B); //???
```

Упаковка полей структуры

```
struct A
{
    char a;
    int n;
    bool b;
    double d;
};

struct B
{
    int n;
    char a;
    bool b;
    double d;
};

sizeof(A); //???
sizeof(B); //???
```

Структуры и функции

При передаче структуры в качестве параметра по значению — тратится время на копирование полей структуры и занимается дополнительная память в стеке.

```
//date.cpp
#include "date.h"

void Print(MyType s) { //ПЛОХО, т.к. лишние
    копирования
     printf("%2d␣%2d␣%4d␣%s\n", s.day, s.month,
                    s.year, s.info);
}
void Print(const MyType *s) { //OK
     printf("%2d␣%2d␣%4d␣%s\n", s->day, s->month,
                    s->year, s->info);
}
void Print2(const MyType &s) { //OK
     printf("%2d␣%2d␣%4d␣%s\n", s.day, s.month,
                    s.year, s.info);
}
```

Структуры и функции

При передаче структуры в качестве параметра по значению — тратится время на копирование полей структуры и занимается дополнительная память в стеке.

```
#include "date.h"

int main()
{
    MyType t;
    Print(t); //???
    Print(&t); //???
    Print2(t); //???
}
```

Возвращение значения

```
MyType Ret() //что будет если MyType &
{
    MyType temp;
    return temp;
}

int main()
{
    MyType t;
    t = Ret(); //???

    MyType t2 = Ret(); //???

}
```

Поля структуры

```
struct A
{
    int a;
    char c[100];
};

struct B
{
    A a;
    int *q;
    A *z;
    B *next;
};
```

Экономия памяти и битовые поля

```
enum WD {MONDAY, TUESDAY, ...};

struct Date
{
    unsigned char day; //требуется 5bits -> 8bits
    unsigned char month; //требуется 4bits -> 8bits
    unsigned short year; //требуется 12bits -> 16
        bits
    WD wd; //требуется 3bits -> 32bits
};

sizeof(Date);
```

Битовые поля

Битовое поле — отдельное поле структуры или класса.

- Для каждой переменной отводится только требуемое количество битов.
- Несколько переменных упаковывают в одну физическую переменную подходящей длины.
- Компилятор сам отводит нужное количество битов под переменные и осуществляет сдвиги для чтения и записи значений.
- Целый тип переменной может быть signed или unsigned, char, short, int, long, enum, bool.

Пример битовых полей

```
enum WD {MONDAY, TUESDAY, ...};

struct Date
{
    unsigned char day : 5;
    unsigned char month : 4;
    unsigned short year : 12;
    WD wd : 3;
};

int main()
{
    Date d;
    d.month = 6;
    d.wd = MONDAY;
}
```

Ограничения битовых полей

- Для битовых полей невозможно получить адрес.
- Невозможно объявить ссылку и получить для нее адрес.
- Оператор sizeof не работает с битовыми полями.

Объединения (union)

Позволяют по разному интерпретировать одну и туже переменную (область памяти).

```
union имя_пользовательского_типа
{
    список_полей;
};

union A
{
    int a;
    char ar[4];
};
```

Использование объединений

```
union A
{
    int a;
    char ar[4];
};

int main()
{
    union A a; //Или A a; //для C++
    a.a = 0x12345678;
    std::cout << a.ar[0] << "␣" << a.ar[1] << std::
        endl;
    std::cout << sizeof(A);
    A b = { 0x00000111 }; //Инициализация

}
```