Подготовительная программа по программированию на C/C++

Занятие №1

Валентина Глазкова

Основы работы с памятью в программах на языке С

- Указатели и одномерные массивы.
- Основы препроцессорной обработки.
- Классы памяти в языке С.
- Оформление процедурного кода. Антишаблоны.
- Реализация журналирования.
- Работа с аргументами командной строки.

Указатели и арифметика указателей. Тип ptrdiff_t

- Стандартные указатели типа T* как составной тип языка C и символический способ использования адресов можно условно считать «шестым классом памяти», важной особенностью которого является поддержка специфической арифметики.
- Пусть p, p2 — указатели типа T*, а n — значение целого типа (желательно — ptrdiff_t). Тогда:
  - p + n либо n + p — адрес, смещенный относительно p на n единиц хранения размера sizeof(T) в направлении увеличения адресов («вправо»);
  - p – n — адрес, смещенный относительно p на n единиц хранения размера sizeof(T) в направлении уменьшения адресов («влево»);
  - p++ либо ++p, p-- либо --p — аналогичны p + 1 и p – 1, соответственно;
  - p – p2 — разность содержащихся в указателях адресов, выраженная в единицах хранения и имеющая тип ptrdiff_t. Разность положительна при условии, что p расположен в пространстве адресов «правее» p2.

Одномерные массивы

- **Для одномерного массива `T a[N]` в языке C справедливо**:
  - массивы поддерживают полную и частичную инициализацию, в том числе с помощью выделенных инициализаторов;
  - в частично инициализированных массивах опущенные значения трактуются как нули;
  - элементы массивов размещаются в памяти непрерывно и занимают смежные адреса, для обхода которых может использоваться арифметика указателей;
  - `sizeof(a)` возвращает размер массива в байтах (не элементах!);
  - `sizeof(a[0])` возвращает размер элемента в байтах.
  - строки `char c[N]` конструктивно являются частными случаями массивов, при этом в корректных строках `c[sizeof(c) – 1] == '\0'`;
- **Принятая система обозначения массивов является лишь особым способом применения указателей**.

Одномерные массивы: пример

```
// с освобождением скобок
int a[] = {1, 2, 3};
// эквивалентно int a[3] = {1, 2, 3};


// с частичной неявной инициализацией
int b[5] = {1, 2, 3};
// эквивалентно:
// int b[5] = {1, 2, 3, 0, 0};


// с выделенными инициализаторами
int c[7] = {1, [5] = 10, 20, [1] = 2};
// эквивалентно:
// int c[7] = {1, 2, 0, 0, 0, 10, 20};
```

Одномерные массивы и указатели

- Пусть T a[N] — массив. Тогда:
  - имя массива является **константным указателем на 0-й элемент**:
    a == &a[0];
  - для любых типов и длин массивов **справедливо**:
    &a[i] == a + i и a[i] == *(a + i);
- С учетом этого эквивалентны прототипы:
  - int f (double [], int);
  - int f (double *, int);
- Передать массив в функцию можно так, как показано выше, или **как пару указателей: на 0-й и N-й элементы** (обращение к элементу a[N] без его разыменования допустимо):
  - int f (double *, double *);

Одномерные массивы: примеры

```
int findmax(int *arr, int count)
{
    int idx = 0;
    for (int i = 1; i < count; ++i) {
        if (arr[i] > arr[idx]) { idx = i; }
    }
    return idx;
}
```

```
int getmax2(const int *arr, int count)
{
    int prev_max = arr[0], curr_max = arr[1];
    if (arr[1] < arr[0]) { curr_max = arr[0]; prev_max = arr[1]; }
    for (int i = 2; i < count; i++) {
        if (arr[i] >= curr_max) { prev_max = curr_max; curr_max = arr[i]; }
    }
    return prev_max;
}
```

Указатели и строки: примеры (1/3)

```
char *strcat(char *str1, const char *str2)
{
        char *cp = str1;
        while (*cp) cp++;
        while (*cp++ = *str2++);
        return str1;
}
```

```
int strlen(const char *str)
{
        const char *eos = str;
        while (*eos++);
        return (int) (eos - str - 1);
}
```

Указатели и строки: примеры (2/3)

```
int strcmp(const char *str1, const char *str2)
{
        while(*str1==*str2 && *str1)
        {
                str1++; str2++;
        }
        return *str1-*str2;
}


char * strcpy(char * str1, const char * str2)

{

        char * d = str1;

        while ( *d++ = *str2++ );

        return str1;

}
```

Указатели и строки: примеры (3/3)

```
char * substr(char* str, char* sub)
{
        int i, j;
        for(i=j=0; str[i]!='\0'; i++)
        {
                while((str[i+j]!='\0')&&(sub[j]==str[i+j]))  j++;
                if(sub[j] == '\0') return &str[i];
                j = 0;
        }
        return NULL;
}
```

```
char *strchr(const char *str, int ch)
{
        while (*str && *str != (char) ch) str++;
        if (*str == (char) ch) return (char *) str;
        return NULL;
}
```

Основы препроцессорной обработки

- Препроцессор анализирует исходный код программы до компиляции, следуя предназначенным ему директивам.
- Директивы препроцессора:
  - имеют первым символом # («решетку»);
  - распространяют свое действие от точки вхождения до конца файла.
- Типичными директивами препроцессора являются:
  - #include — включает в текст файлы с исходным кодом;
  - #define — вводит в исходный код символические константы и макроопределения (обратная директива — #undef);
  - #if, #ifdef, #ifndef, #else, #elif, #endif — реализуют условное включение фрагментов исходного кода в текст, передаваемый компилятору;
  - #error — возбуждает ошибку времени компиляции;
  - #pragma once — контролирует, чтобы конкретный исходный файл при компиляции подключался строго один раз.

Основы препроцессорной обработки: пример использования #define

```
#define SIZE 100
#define PRINT_X printf("X:\t%7d\n", x)
#define CUBE(N) ((N) * (N) * (N))

int a[SIZE];
// эквивалентно: int a[100];
PRINT_X;
// эквивалентно: printf("X:\t%7d\n", x);
printf("%d\n", CUBE(SIZE));
// эквивалентно:
// printf("%d\n", ((100) * (100) * (100));
```

Основы препроцессорной обработки: пример условной компиляции и #error

```
// выбор генератора псевдослучайных чисел
#ifdef ANSI_C_LIKE
#define A 1103515245
#define C 12345
#else
#define A 22695477
#define C 1
#endif
/* выброс ошибки, если компилятор не является компилятором C++. __cplusplus - макрос, который задаётся компилятором (макрос объявлен, если работает компилятор C++, и не объявлен – если компилятор Си)*/
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

Основы препроцессорной обработки: пример использования #pragma

File «grandfather.h»

```
#pragma once
struct f { int member; };
```

File «father.h»

```
#include "grandfather.h"
```

File «child.c»

```
#include "grandfather.h"
#include "father.h"
```

Модели управления памятью и области видимости объектов данных

- Предлагаемые языком C модели управления объектами данных (переменными) закреплены в понятии класса памяти, которое охватывает:
  - **время жизни** — продолжительность хранения объекта в памяти;
  - **область видимости** — части исходного кода программы, из которых можно получить доступ к объекту по идентификатору;
  - **связывание** — части исходного кода, способные обращаться к объекту по его имени.
- Для языка C характерны три области видимости:
  - **блок** — фрагмент кода, ограниченный фигурными скобками (составной оператор), либо составной оператор и предшествующий заголовок функции или заголовок оператора for, while, do while и if;
  - **прототип функции**;
  - **файл**.

Связывание объектов данных

- Объекты данных, видимые в пределах блока и прототипа функции, связывания не имеют (замкнуты в областях, где определены).
- Для объектов, видимых в пределах файла (глобальных), язык предлагает **два варианта** связывания:
  - **внутреннее** — объект является «приватным» для файла и может использоваться лишь в нем (но любой функцией!);
  - **внешнее** — объект может использоваться в любой точке многофайловой программы.

Связывание объектов данных: пример

```
// область видимости: прототип функции
int foo(double *d, int n);


// область видимости: блок
for(int i = 0; i < n; i++) {
                int bar;
                // ...
}
// область видимости: файл, внутреннее связывание
static int count = 0;
// область видимости: файл, внешнее связывание
double accuracy = 0.001;
```

Время жизни объектов данных

- Объекты данных в программах на языке С имеют **статическую** или **автоматическую** продолжительность хранения:
  - время жизни статических объектов тождественно времени выполнения программы;
  - время жизни автоматических объектов в целом тождественно времени выполнения охватывающего их блока.
- **Статическими** являются, главным образом, объекты, видимые в пределах файла. Спецификатор static при таком объекте определяет только тип связывания, но не время жизни объекта.

Инициализация объектов данных

- Статические объекты неявно **инициализируются нулем** (0, '\0'), автоматические объекты неявно **вообще не инициализируются**.
- Для явной инициализации статических объектов должны использоваться константные выражения, вычислимые компилятором.
- Например:

```
static char   space    = 0x20; // верно
static size_t int_sz   = sizeof(int); // верно
static size_t int10_sz = 10 * int_sz; // неверно
```

Классы памяти в языке C

Класс памяти	Время жизни	Область видимости	Тип связывания	Точка определения
Автоматический	Автоматическое	Блок	Отсутствует	В пределах блока, опционально `auto`
Регистровый	Автоматическое	Блок	Отсутствует	В пределах блока, `register`
Статический, без связывания	Статическое	Блок	Отсутствует	В пределах блока, `static`
Статические, с внешним связыванием	Статическое	Файл	Внешнее	Вне функций
Статические, с внутренним связыванием	Статическое	Файл	Внутреннее	Вне функций, `static`

Автоматические и регистровые переменные: пример

```
// автоматические переменные
int foo(unsigned u)
{
   auto int bar = 42;
   // ...
}

// регистровые переменные
int get_total(register int n)
{
   // ...
   for(register int i = 0; i < n; i++)
   // ...
}
```

Размещение объектов данных на регистрах процессора

- Применение ключевого слова register для активно используемых переменных:
  - несет все риски «ручной оптимизации» кода;
  - относится к регистрам ЦП (в x86/x86-64: AX, EBX, RCX и т.д.), но не кэш-памяти ЦП 1-го или 2-го уровня;
  - является рекомендацией для компилятора, но не требованием к нему;
  - вполне может игнорироваться компилятором, который будет действовать «на свое усмотрение» (например, разместит переменную на регистре, потребность в котором возникнет позднее всего).
- **Операция взятия адреса переменной** со спецификатором register **недопустима** вне зависимости от того, размещена ли она фактически на регистре.

Статические объекты с внутренним связыванием и без связывания: пример

```
// без связывания:
// статические внутренние объекты функций
int callee(int n)
{
        static int counter = 0; // не часть функции!
        // …
}
// с внутренним связыванием:
// статические внутренние объекты файлов
static double epsilon = 0.001;
int foo(double accuracy)
{
        if(accuracy < epsilon)  // ...
}
```

Статические объекты с внешним связыванием: пример

```
// с внешним связыванием:
// статические внешние объекты ("внешняя память")
double     time;          // внешнее определение
long int   fib[100];      // внешнее определение
extern char space;        // внешнее описание
// (объект определен в другом файле)

int main(void)
{
        extern double time; // необязательное описание
        extern long int fib[]; // необязательное описание;
        // размер массива необязателен
        // ...
}
```

Классы памяти функций

- Применительно к невстраиваемым функциям различают **два класса памяти**:
  - **внешний** — выбирается компилятором по умолчанию и позволяет ссылаться на функцию (вызывать ее) из любой точки многофайловой программы;
  - **статический** — выбирается при наличии спецификатора static и позволяет изолировать функцию в том файле, где она определена.
- Например:

```
int one(void); // внешнее определение

// статическое определение

static int two(void);

// необязательное внешнее описание

extern int three(void);
```

Оформление процедурного кода

- **Выбор прототипов функций**
  - прототипы отражают поток данных (*англ.* dataflow)
  - использование const при передаче параметров
  - передача больших объектов через указатель
- **Выражение алгоритмов в терминах процедурного языка**
  - выбор конструкций ветвления и циклов
  - обработка частных случаев – дублирование кода
- **Группировка функций, отвечающих за один уровень функционала** – ввод, алгоритм, вывод на экран (в файл)
- **Стиль оформления** — отступы, пробелы, длина строк, имена и т.п. (пример правил: https://ejudge.ru/study/3sem/style.shtml)

Оформление процедурного кода: пример

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

//вызов функции по заданному имени и аргументу
double call_by_name(const char *name, int arg)
{
        static const char *names[] = {"sin", "cos", "tan", NULL};
        static double (*fp[])(double) = {sin, cos, tan};
        for (int i=0; names[i]!=NULL; i++)
          if (strcmp(names[i],name) == 0)
          {
                //вызов функции по i-му указателю в массиве fp
                return ((*fp[i])(arg));
          }
         return 0.0;
}
```

Оформление процедурного кода: антишаблоны (1/3)

- **Использование оператора безусловного перехода к метке** (goto) – только вперёд (например, для выхода из вложенных циклов и обработки ошибок)
- «**Загадочный» код** (*англ.* cryptic code) — выбор малоинформативных, часто однобуквенных идентификаторов, не сопровождаемых комментариями автора
- «**Жесткий» код** (*англ.* hard code) — запись конфигурационных параметров как строковых, логических и числовых литералов, рассеянных по исходному коду и затрудняющих настройку и сопровождение программной системы
- **Спагетти-код** (*англ.* spagehetti code) — несоблюдение правил выравнивания, расстановки пробельных символов и т.д., а также превышение порога сложности одной процедуры (функции); простой и удобной мерой такого порога сложности является высота экрана

Оформление процедурного кода: антишаблоны (2/3)

- **Магические числа** (*англ.* magic numbers) — определять как символические константы все числовые литералы за исключением, может быть, 0, 1 и -1
- **Применение функций как процедур** (*англ.* functions as procedures) — например, функция scanf языка C возвращает целочисленный результат и его необходимо использовать в своих программах
- **«Божественные» функции** (*англ.* God functions) —такие функции берут на себя — в разных сочетаниях — ввод данных, вычисления и вывод результатов на экран, диск, в поток и т.д. или иные задачи, каждая из который достойна оформления как самостоятельной функции

Оформление процедурного кода: антишаблоны (3/3)

- **Неиспользование переносимых типов** — прежде всего, ptrdiff_t
- «**Утечки» памяти** (*англ.* memory leaks)
- Внезапное **завершение процесса** вместо аварийного выхода из функции с возвратом кода ошибки
- Использование ветвлений **с условиями, статистически смещенными** не к истинному, а к ложному результату(способствуют образованию «пузырей» в конвейере микрокоманд и существенно снижают эффективность кэш-памяти инструкций микропроцессора)
- **Недостижимый код** (*англ.* unreachable code)

Реализация журналирования

- Одним из простейших и в вместе с тем эффективных инструментов отладки программы является **журналирование**
- **Журналирование** предполагает вывод в файл (или на экран) информации о событиях, возникающих в программе, и о промежуточном состоянии программы (например: факт вызова функции и данные о ее аргументах, содержимое переменных и т.п.)
- Отладочные сообщения должны находиться в ключевых узлах программы и позволять отследить ход ее выполнения
- Вывод на экран буферизируется построчно ('\n')
- В случае фатальной ошибки, приводящей к аварийному завершению программы (наиболее распространена ошибка сегментации - Segmentation Fault) отладочные сообщения, находящиеся в буферах, не будут выведены
- Часто бывает удобно иметь возможность оперативно отключать и включать отладочные выводы

Реализация журналирования: пример (1/2)

Файл debug.h

```
#ifndef DEBUG_H
#define DEBUG_H
#include <stdio.h>
#define PDEBUG(level, fmt, args,...)
#ifdef DEBUG
#undef PDEBUG
#define PDEBUG(level, fmt, args,...) \
   if(level <= DEBUG)
      printf("%s: %d: " fmt " \n", __FUNCTION__, __LINE__, ## args)
#endif
#endif
```

Реализация журналирования: пример (2/2)

```
#include <stdio.h>
#define DEBUG 10
#include "debug.h“
int main()
{
   int i = 0;
   while(i < 6)
   {
      PDEBUG(1, "i = %d", i);
      i++;
   }
}
```

```
$ ./debug

main: 10: i = 0
main: 10: i = 1
main: 10: i = 2
main: 10: i = 3
main: 10: i = 4
main: 10: i = 5
```

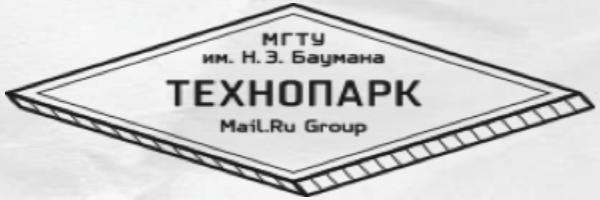
Работа с аргументами командной строки (1/2)

- Язык С имеет встроенные средства для получения аргументов команды непосредственно от командного процессора
- Функция main() может получать аргументы, с которыми запущена программа – аргументы командной строки. Для этого достаточно снабдить функцию main() набором параметров, которые обычно имеют имена argc и argv: main(int argc; char *argv[])
- **Параметр argc (ARGument Count)** получает от командного процессора информацию о количестве аргументов, набранных в командной строке, включая и имя самой команды (argc>= 1)

Работа с аргументами командной строки (2/2)

- **Параметр argv (ARGument Vector)** обычно определяется как массив указателей на строки, каждый из которых хранит адрес начала отдельного аргумента командной строки (ragv[0] – ссылается на имя самой команды)
- Пример – вывод на экран аргументов командной строки программы

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        for(int i=0; i<argc; i++)
            printf("%s ", argv[i]);
        return 0;
}
```



Валентина Глазкова

Спасибо за внимание!

О курсе

- Курс посвящён изучению и реализации основных принципов объектно-ориентированного и обобщённого программирования на языке C++
- **Цель курса** — формирование практических навыков и умений в области применения языков C и C++
- **Навыки** — работа с памятью, реализация основных структур данных на языке C, реализация средств инкапсуляции, полиморфизма и наследования в программах на языке C++, обработка исключительных ситуаций, базовые навыки обобщённого программирования, базовые навыки использования стандартной библиотеки C++

О курсе

- **Контроль знаний** — в рамках курса проводятся практические занятия, на каждом из которых оценивается выполнение домашних заданий; по окончании курса проводится экзамен
- **Состав курса** — 9 лекций, 11 практикумов
- Регламент:
  - Вопросы — общезначимые : в любое время (по поднятию руки!), индивидуальные: в перерыве или после занятия

Обзор курса (1/5)

- **Лекция №1. Основы работы с памятью в программах на языке С**
  - Основы препроцессорной обработки. Классы памяти в языке С. Указатели и арифметика указателей. Оформление процедурного кода. Антишаблоны. Реализация журналирования. Работа с аргументами командной строки.
- **Практикум №1. Решение задач на тему «Основы работы с памятью в программах на языке С».**
- **Лекция №2. Реализация структур данных на языке С.**
  - Одномерные массивы и строки. Многомерные массивы. Списки, стеки, очереди, деревья. Выделение динамической памяти.
- **Практикум №2,3. Решение задач на тему «Реализация структур данных на языке С»**

Обзор курса (2/5)

- **Лекция №3. Объектная модель языка C++**
  - Основные принципы объектно-ориентированной парадигмы. Определение и состав класса. Работа с членами класса и указателями на них. Дружественные классы и функции. Вложенные типы.
- **Практикум №4. Решение задач на тему «Объектная модель языка C++».**
- **Лекция №4. Специальные вопросы инкапсуляции.**
  - Инициализация, копирование, преобразование и уничтожение объектов. Праводопустимые выражения в C++11.
- **Практикум №5. Решение задач на тему «Специальные вопросы инкапсуляции».**

Обзор курса (3/5)

- **Лекция №5. Специальные вопросы наследования и полиморфизма. Класс как область видимости**
  - Раннее и позднее связывание. Перегрузка и перекрытие членов класса. Наследование как способ добавления новых членов класса. Наследование как способ изменения поведения класса. Виртуальные функции. Наследование и агрегирование. Класс как область видимости. Перегрузка функций. Перегрузка операций.
- **Практикум №6,7. Решение задач на тему «Специальные вопросы наследования и полиморфизма. Класс как область видимости. Перегрузка».**
- **Лекция №6. Объектно-ориентированное программирование.**
  - Применение и эффективная реализация принципов объектно-ориентированного программирования при реализации приложений на языке C++.
- **Практикум №8. Решение задач на тему «Реализация принципов объектно-ориентированной парадигмы в программах на языке C++».**

Обзор курса (3/5)

- **Лекция №7. Модульное программирование. Шаблоны классов и методов**
  - Пространства имён. Ортодоксальная каноническая форма класса. Разбиение программы на файлы и модули. Необходимость в обобщённом программировании. Шаблоны классов и методов. Параметры шаблонов. Специализация, конкретизация и перегрузка шаблонов.
- **Практикум №9. Решение задач на тему «Шаблоны классов и методов».**
- **Лекция №8. Обработка исключительных ситуаций.**
  - Понятие, поддержка и технология обработки исключительных ситуаций. Реализация журналирования.
- **Практикум №10. Решение задач на тему «Обработка исключительных ситуаций».**

Обзор курса (4/5)

- **Лекция №9. Стандартная библиотека шаблонов STL**
  - Контейнеры и итераторы. Примеры шаблонных классов-контейнеров (vector, list, map, set), сложность основных методов работы с ними. Основные алгоритмы библиотеки STL.
- **Практикум №11. Решение задач на тему «Стандартная библиотека шаблонов STL».**

Отчётность по курсу: основана на балльно-рейтинговой системе – каждый практикум + экзамен в конце курса оценивается по десятибалльной системе

Итоговая оценка: 0–84 неудовлетворительно, 85-99 удовлетворительно, 100-109 хорошо, 110–120 отлично

Обзор курса (5/5)

- **Оценка заданий практикума:**
  - 10 — задача сдана без замечаний
  - 9, 8 — задача сдана с несущественными замечаниями
  - 7, 6 — задача работает на всех тестах, но имеет существенные недостатки
  - 4, 5 — работают некоторые частные решения задачи
  - 1, 2, 3 — выполнен начальный уровень решения задачи

Рекомендуемая литература (1/2)

- Керниган Б., Ритчи Д. Язык программирования C. – Вильямс, 2009. – 292 с.
- Прата С. Язык программирования C. Лекции и упражнения. — Вильямс, 2013. — 960 с.
- Прата С. Язык программирования C++. Лекции и упражнения. — Вильямс, 2012. — 6 е изд. — 1248 с.
- Столяров А.В. Оформление программного кода: методическое пособие. – М.: МАКС Пресс, 2012. – 100 с. http://www.stolyarov.info/books/codestyle
- Столяров А.В. Введение в язык Си++. -М: МАКС Пресс, 2012. – 3-е изд. – 127 с. http://www.stolyarov.info/books/cppintro
- Макконнелл С. Совершенный код. Мастер-класс. – Русская редакция, 2012. – 896 с.
- Седжвик Р. Алгоритмы на C++. – Вильямс, 2011. – 1056 с.

Рекомендуемая литература (2/2)

- Шилдт Г. Полный справочник по C++. – Вильямс, 2007. – 800 с.
- Справка по языкам C/C++: http://ru.cppreference.com/w/, http://en.cppreference.com/w/
- Липпман С., Лажойе Ж. Язык программирования C++. Вводный курс. – Невский диалект, ДМК Пресс. – 1104 с.
- Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Вводный курс. – Вильямс, 2007. – 4-е изд. – 896 с.
- Страуструп Б. Программирование. Принципы и практика использования C++. – Вильямс, 2011. – 1248 с.
- Страуструп Б. Язык программирования C++. – Бином, 2011. – 1136 с.
- https://ejudge.ru/study/3sem/style.shtml

О себе

Валентина Глазкова

- Окончила с отличием факультет Вычислительной математики и кибернетики МГУ им. М.В.Ломоносова (2006)
- Кандидат физико-математических наук (2008, специальность 05.13.11, тема диссертации «Исследование и разработка методов построения программных средств классификации многотемных гипертекстовых документов»)
- Ассистент кафедры Автоматизации систем вычислительных комплексов факультета Вычислительной математики и кибернетики МГУ им. М.В. Ломоносова (с 2009 года по настоящее время)
- Автор более 20 научных статей и патента РФ на полезную модель «Система анализа и фильтрации интернет-трафика на основе методов классификации многотемных документов» (2009).