1. Указатели, void*, указатели на функции (на примере функции qsort)

План ответа:
1 понятие «указатель»;
2 void*, особенности операций с ним;
3 приведение указателей разных типов к void* и обратно;
4 определение указателя на функцию;
5 присваивание значения указателю на функцию;
6 вызов функции по указателю;
7 использование указателей на функции.

1. Указатель – это объект, содержащий адрес объекта или функции, либо выражение, обозначающее адрес объекта или функции.

2. void* – бестиповой указатель, указывает на произвольное место в памяти.
Особенности
- Указателя типа void* нельзя разыменовывать.
- К указателям типа void* не применима адресная арифметика.

Основная миссия – упрощение преобразований указателей, универсализация структур и формальных параметров функций.
Тип указателя void* используется, если тип объекта неизвестен.
Позволяет передавать в функцию указатель на объект любого типа; полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов.

3. В языке С допускается присваивание указателя типа void* указателю любого другого типа (и наоборот) без явного преобразования типа указателя.

```
double d = 5.0;
double *pd = &d;
void *pv = pd;
pd = pv;
```

4. Объявление указателя на функцию

```
double trapezium(double a, double b, int n, double (*func)(double));
```

5. присваивание значения указателю на функцию

```
result = integrate(0, 3.14, 25, sin);
```

6. Вызов функции по указателю

```
y = (*func)(x); // y = func(x);
```

7. Использование указателей
Например, функция
void qsort(void* base, size_t memb, size_t size, int (*compar)(const void*, const void*)) из заголовочного файла <stdlib.h> имеет в качестве формального параметра указатель на функцию, сравнивающие некие 2 значения, задаваемые ссылками на них, без ограничения на типы параметров.

2. Динамические одномерные массивы.

План ответа:
1 Функции для выделения и освобождения памяти (malloc, calloc, realloc, free). Порядок работы и особенности использования этих функций.
2 Способы возврата динамического массива из функции
3 Типичные ошибки при работе с динамической памятью (утечка памяти, «дикий» указатель, двойное освобождение).

1 Для выделения памяти необходимо вызвать одну из трех функций, объявленных в заголовочном файле stdlib.h:
– malloc (выделяет блок памяти и не инициализирует его);
– calloc (выделяет блок памяти и заполнят его нулями);
– realloc (перевыделяет предварительно выделенные блок памяти).

Особенности:
Указанные функции не создают переменную, они лишь выделяют область памяти. В качестве результата функции возвращают адрес расположения этой области в памяти компьютера, т.е. указатель.
Поскольку ни одна из этих функций не знает данные какого типа будут располагаться в выделенном блоке все они возвращают указатель на void.
В случае если запрашиваемый блок памяти выделить не удалось, любая из этих функций вернет значение NULL.
После использования блока памяти он должен быть освобожден. Сделать это можно с помощью функции free.

Результат вызова функций malloc, calloc или realloc, когда запрашиваемый размер блока равен 0, зависит от реализации:
– вернется нулевой указатель;
– вернется «нормальный» указатель, но его нельзя использоваться для разыменования.
Поэтому перед вызовом этих функций нужно убедиться, что запрашиваемый размер блока не равен нулю.

malloc
void* malloc(size_t size);
Функция malloc выделяет блок памяти указанного размера size. Величина size указывается в байтах.
Выделенный блок памяти не инициализируется (т.е. содержит «мусор»).
Для вычисления размера требуемой области памяти необходимо использовать операцию sizeof.
a = (int*) malloc(n * sizeof(int));
Преимущества явного приведения типа:
– компиляции с помощью c++ компилятора;
– у функции malloc до стандарта ANSI C был другой прототип (char* malloc(size_t size));
– дополнительная «проверка» аргументов разработчиком.

Недостатки явного приведения типа:
– начиная с ANSI C приведение не нужно;
– может скрыть ошибку, если забыли подключить stdlib.h;
– в случае изменения типа указателя придется менять и тип в приведении.

calloc
void* calloc(size_t nmemb, size_t size);
Функция calloc выделяет блок памяти для массива из nmemb элементов, каждый из которых имеет размер size байт.
Выделенная область памяти инициализируется таким образом, чтобы каждый бит имел значение 0.

realloc
realloc(void* ptr, size_t bytes)
перевыделяет память под объект ptr. Выделяют следующие случаи:
ptr==NULL && bytes>0. Происходит обычное выделение памяти, как при malloc(bytes).
ptr!=NULL && bytes==0. Происходит освобождение памяти, как при free(ptr).
ptr!=NULL && bytes!=0. В худшем случае выделяется bytes байтов, копируются имеющиеся значения байтов из старой области памяти в новую, освобождается старая память.
В лучшем случае, когда соседние справа байты свободны в достаточном количестве или bytes не больше текущего размера выделенной области, перемещений не происходит.

Особенность использования: отслеживать, если realloc возвращает NULL (результат записать в отдельный буфер).

```
            void *ptmp = realloc(pbuf, 2 * n);
            if (ptmp)
                  pbuf = ptmp;
            else
                  // обработка ошибочной ситуации
```

free
void free(void *ptr);
Функция free освобождает (делает возможным повторное использование) ранее выделенный блок памяти, на который указывает ptr.
Если значением ptr является нулевой указатель, ничего не происходит.
Если указатель ptr указывает на блок памяти, который не был получен с помощью одной из функций malloc, calloc или realloc, поведение функции free не определено.

2 Способы возврата динамического массива из функции

```
double* get_array_1(int *n)
{
   *n = 0;    // ? значение изменяется, даже если была ошибка

   // определить кол-во элементов (м.б., это сложный процесс)
   int nmemb = 5;

   // выделить память
   double *p = NULL;
   if (nmemb)
   {
      p = malloc(nmemb * sizeof(double));
      if (p)
         *n = nmemb;
   }

   return p;
}

int get_array_2(double **data, int *n)
{
   int rc = 0;
   *n = 0;          // ? эти значения меняются, даже если
   *data = NULL;    // ? была ошибка

   // определить кол-во элементов (м.б., это сложный процесс)
   int nmemb = 5;

   // выделить память
   if (!rc && nmemb)
   {
      *data = malloc(nmemb * sizeof(double));
      if (*data)
         *n = nmemb;
      else
         rc = -1; // ошибка выделения памяти
   }
   return rc;
}
```

3 Типичные ошибки

Утечки памяти (memory leak)

Разыменование «битого» указателя(invalid pointer)

Двойное освобождение памяти(double free)

3. Указатели и многомерные статические массивы.

План ответа:

1 концепция многомерного массива как «массива массивов»;
2 определение многомерных массивов;
3 инициализация многомерных массивов;
4 «слои», составляющие многомерные массивы;
5 обработка многомерных массивов с помощью указателей;
6 передача многомерных массивов в функцию;
7 const и многомерные массивы.

1-2

Отличие многомерного массива от одномерного состоит в том, что в одномерном массиве положение элемента определяется одним индексом, а в многомерном – несколькими. Примером многомерного массива является матрица.
Общая форма объявления многомерного массива
тип имя[размерность_1][размерность_2]...[размерность_m];
Количество размерностей массива практически не ограничено.

```
int a[3][2];
```

Компилятор Си располагает строки матрицы “a” в памяти одну за другой вплотную друг к другу.

3 Инициализация многомерных массивов

```
int a[3][3] = {
      {1, 2, 3},
      {4, 5}
};

int d[][2] = { {1, 2} };

int e[][] = {
      {1, 2},
      {4, 5}
}; // error: array type has incomplete element type

int b[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// warning: missing braces around initializer [-Wmissing-braces]

// c99
int c[2][2] = {[0][0] = 1, [1][1] = 1};
```

4 Компоненты многомерного массива int a[2][3][5];

a – массив из двух элементов типа “int [3][5]”

```
int (*p)[3][5] = a;
```

a[i] – массив из трех элементов типа “int [5]” (i из [0, 1])

```
int (*q)[5] = a[i];
```

a[i][j]–массив из пяти элементов типа “int” (i из [0, 1], j из [0,1,2])

```
int *r = a[i][j];

a[i][j][k] – элемент типа “int” (i из [0, 1], j из [0, 1, 2],
k из [0, 1, 2, 3, 4])
int s = a[i][j][k];
```

5 Обработка многомерных массивов с помощью указателей

```
как одномерный.
      int a[N][M];
      int *p;
      for (p = &a[0][0]; p <= &a[N-1][M-1]; p++)
            *p = 0;

Обработка строки матрицы (обнуление i-ой строки)
      // указатель на начало i-ой строки
      int *p = &a[i][0];
      // обнуление i-ой строки
      for (p = a[i]; p < a[i] + M; p++)
            *p = 0;
a[i] можно передать любой функции, обрабатывающей одномерный массив

Обработка столбца матрицы (обнуление j-го столбца)
      // указатель на строку (строка – это массив из M элементов)
      int (*q)[M]; //!!Без скобок получится массив из M указателей.
      Выражение q++ смещает указатель на следующую строку
      Выражение (*q)[j] возвращает значение в j-ом столбце строки, на
      которую указывает q.
      for (q = a; q < a + N; q++)
      (*q)[j] = 0;
```

6 Передача многомерных массивов в функцию

```
Пусть определена матрица
int a[N][M];
Для ее обработки могут быть использованы функции со следующими
прототипами:
      void f(int a[N][M], int n, int m);
      void f(int a[][M], int n, int m);
      void f(int (*a)[M], int n, int m);

Неверные прототипы:
      void f(int a[][], int n, int m);
      // не указана вторая размерность => компилятор не сможет
      // выполнить обращение по индексу
      void f(int *a[M], int n, int m);
      void f(int **a, int n, int m);
      // неверный тип – массив указателей
```

7 const и многомерные массивы.

Согласно C99 T (*p)[N] не преобразуется неявно в T const (*p)[N].
Способы борьбы
- не использовать const;
- использовать явное преобразование типа

4. Массивы переменной длины (c99), их преимущества и недостатки, особенности использования.

В C99 внутри функции или блока можно задавать размер массива с помощью выражений, содержащих переменные.

```
{
      int n;
      ...
      int a[n];
}
```

Особенности использования

– Длина такого массива вычисляется во время выполнения программы, а не во время компиляции.

– Память под элементы массива выделяется на стеке.

– Массивы переменной длины могут быть многомерными.

– Адресная арифметика справедлива для массивов переменной длины.

- sizeof для такого массива также выполняется во время выполнения

Преимущества

– Массивы переменной длины облегчают описание заголовков функций, которые обрабатывают массивы.

Недостатки

– Массивы переменного размера нельзя инициализировать при определении.

- переменную длину могут иметь только локальные массивы (т.е. видимые в блоке или в прототипе)

- массивы переменной длины за время "своей жизни" не меняют своих размеров. На самом деле массив переменной длины создается с другим размером каждый раз, когда встречается его объявление.

5. Динамические многомерные массивы.

План ответа:

Способы выделения памяти для динамических матриц: идеи, реализации, анализ преимуществ и недостатков.

способы представления:

1. одномерный массив

   Как одномерный массив; представление как статического массива. Выделение и освобождение памяти тривиально.

   Достоинства

   - Простота выделения и освобождения памяти
   - Минимум занимаемого места
   - Возможность использовать как одномерный массив

   Недостатки

   - Необходимость всё время обращаться к элементам по сложному индексу
   - Средства для контроля работы с памятью не могут отследить за пределы строки

```
double *data;
int n = 3, m = 2;

data = malloc(n * m * sizeof(double));
if (data)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            // Обращение к элементу i, j
            data[i*m+j] = 0.0;

    free(data);
}
```

2. указатели на строки

Объявляется массив указателей на массивы-строки, элементы которых имеют некий тип, и т.д.

Память под массив строк и массивы-строки выделяется и освобождается отдельно.

Алгоритм выделения памяти

Вход: количество строк (n) и количество столбцов (m)
Выход: указатель на массив строк матрицы (p)

Выделить память под массив указателей (p)
Обработать ошибку выделения памяти
В цикле по количеству строк матрицы (0 <= i < n)
- Выделить память под i-ую строку матрицы (q)
- Обработать ошибку выделения памяти
- p[i]=q

Алгоритм освобождения памяти

Вход: указатель на массив строк матрицы (p) и количество строк (n)

В цикле по количеству строк матрицы (0 <= i < n)
- Освободить память из-под i-ой строки матрицы
Освободить память из-под массива указателей (p)

Достоинства

Возможность обмена строки через обмен указателей.
СРПР может отследить выход за пределы строки.

Недостатки

Сложность выделения и освобождения памяти.
Память под матрицу "не лежит" одним куском.

```
void free_matrix_rows(double **data, int n)
{
    for (int i = 0; i < n; i++)
       // free можно передать NULL
       free(data[i]);

    free(data);
}
```

```
double** allocate_matrix_rows(int n, int m)
{
    double **data = calloc(n, sizeof(double*));
    if (!data)
        return NULL;

    for (int i = 0; i < n; i++)
    {
        data[i] = malloc(m * sizeof(double));
      if (!data[i])
      {
          free_matrix_rows(data, n);
          return NULL;
      }
    }
    return data;
}
```

3. одним дампом

Выделяется единый дамп памяти, первая часть которого идет на массивы указателей на строки, вторая содержит значения.

Алгоритм выделения памяти

Вход: количество строк (n) и количество столбцов (m)

Выход: указатель на массив строк матрицы (p)

Выделить память под массив указателей на строки и элементы матрицы (p)

Обработать ошибку выделения памяти

В цикле по количеству строк матрицы (0 <= i < n)

- Вычислить адрес i-ой строки матрицы (q)
- p[i]=q

Достоинства

Простота выделения и освобождения памяти.

Возможность использовать как одномерный массив.

Перестановка строк через обмен указателей.

Недостатки

Сложность начальной инициализации.

СРПР не может отследить выход за пределы строки.

```
double** allocate_matrix_solid(int n, int m)
{
    double **data = malloc(n * sizeof(double*) +
                           n * m * sizeof(double));
    if (!data)
        return NULL;

    for (int i = 0; i < n; i++)
        data[i] = (double*)((char*) data +
                            n * sizeof(double*) +
                            i * m * sizeof(double));
    return data;
}
```

6. Строки.

План ответа:
1 понятия «строка» и «строковый литерал»;
2 определение переменной-строки, инициализация строк;
3 ввод/вывод строк (scanf, gets, fgets, printf, puts);
4 функции стандартной библиотеки для работы со строками (strcpy, strlen, strcmp и др.);
5 идиомы обработки строк.

1 Строка – это последовательность символов, заканчивающаяся и включающая первый нулевой символ
Строковый литерал – последовательность символов, заключенных в двойные кавычки.
Строковый литерал рассматривается компилятором как массив элементов типа char.
Когда компилятор встречает строковый литерал из n символов, он выделяет n+1 байт памяти,
которые заполняет символами строкового литерала и завершает нулевым символом.

```
char str[] = “String for test”;
```

Строка – массив символов, для доступа к элементу строки может использоваться операция индексации
Массив, который содержит строковый литерал, существует в течение всего времени выполнения программы.
В стандарте сказано, что поведение программы не определено при попытке изменить строковый литерал.
Обычно строковые литералы хранятся в read only секции.

2 Определение переменной-строки, которая может содержать до 80 символов обычно выглядит следующим образом:

```
#define STR_LEN 80
...
char str[STR_LEN+1];
```

Инициализация строковых переменных

```
char str_1[] = {‘J’,‘u’,‘n’, ‘e’, ‘\0’};
char str_2[] = “June”;
char str_3[5] = “June”;

char str_4[3] = “June”;
// error: initializer-string for array of chars is too long

char str_5[4] = “June”;
// str_5 не строка!
```

3 Стандартные функции ввода-вывода строк:

Ввод

```
scanf(“%s”,str), fscanf(f,”%s”,str);
// символы-разделители строк – в том числе пробелы и табуляция.
Считывание идёт в заранее выделенный буфер.
Программист сам отслеживает ограничения на длину.

gets(buf), fgets(buf, n_max, f);
// считывает строку вплоть до символа перевода строки или конца
файла (записывая и его, а после него – терминирующий нуль).
```

Вывод

```
printf, fprintf  // вывод строки без её перевода
puts, fputs   // вывод с переводом строки
```

4 функции стандартной библиотеке

strcpy

```
char* strcpy(char *s1, const char *s2);
Вместо функции strcpy безопаснее использовать функцию strncpy
char* strncpy(char *s1, const char *s2, size_t count);
Пример
      strncpy(dst, src, sizeof(dst) - 1);
      dst[sizeof(dst) - 1] = '\0';
```

strlen

```
size_t strlen(const char *s);
```

strcmp

```
int strcmp(const char *s1, const char *s2);
      значение < 0, если s1 меньше s2
      0, если s1 равна s2
      значение > 0, если s1 больше s2
Строки сравниваются в лексикографическом порядке.
Функция strcmp сравнивает символы, сравнивая значения кодов,
которые представляют эти символы.

int strncmp(const char *s1, const char *s2, size_t count);
```

другие

```
char* strdup(const char *s); // НЕ c99
char* strndup(const char *s, size_t count); // НЕ c99
int sprintf(char *s, const char *format, ...);
int snprintf(char *s, size_t num, const char *format, ...); // c99
char* strtok(char *string, const char *delim);  //выполняет поиск
                                                    лексем в строк
```

```
char *strchr (const char *str, int ch);   //выполняет поиск
          первого вхождения символа symbol в строку string.


Перевод числа в строку
      #include <stdlib.h>
      // Семейство функций (atoi, atof, atoll)
      long int atol(const char* str);
      // Семейство функций (strtoul, strtoll, ...)
      long int strtol(const char* string, char** endptr, int
                    basis);
```

5 Особенности обработки строк:

Выход за пределы выделенного буфера строки – крайне нежелательный эффект, не отслеживается компилятором и ложится на плечи программиста.

Терминирующий нуль при преобразованиях строк необходимо сдвигать, а также обеспечивать его нахождение в буфере

Пробег по строке с помощью указателей также должен быть аккуратным, использование вне пределов буфера чревато последствиями.

7. Область видимости, время жизни и связывание.

План ответа:

1 понятия «область видимости», «время жизни» и «связывание» в языке Си;
2 классификация этих понятий;
3 правила перекрытия областей видимости;
4 размещение «объектов» в памяти в зависимости от времени жизни;
5 влияние связывания на объектный и/или исполняемый файл.

1 Время жизни – это интервал времени выполнения программы, в течение которого программный “объект” существует.

Область видимости (scope) имени – это часть текста программы, в пределах которой имя может быть использовано.

Связывание (linkadge) определяет область программы (функция, файл, вся программа целиком), в которой «программный объект» может быть доступен другим функциям программы.

2 В языке Си выделяют следующие области видимости
блок - последовательность объявлений, определений и операторов, заключенная в фигурные скобки.

виды блоков:

составной оператор
определение функции

Переменная, определенная внутри блока, имеет область видимости в пределах блока.
Формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.

файл

Область видимости в пределах файла имеют имена, описанные за пределами какой бы то ни было функции.

Переменная с областью видимости в пределах файла видна на протяжении от точки ее описания и до конца файла, содержащего это определение.

Имя функции всегда имеет файловую область видимости.

функция

Метки - это единственные идентификаторы, область действия которых - функция.

Метки видны из любого места функции, в которой они описаны.

В пределах функции имена меток должны быть уникальными.

прототип функции

Область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций.

Область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа.

```
int f(int, double); // ок
int f(int i, double i); // ошибка компиляции
int print_matrix(int n, int m, double a[n][m]);
```

В языке Си время жизни «программного объекта» делится на три категории

глобальное (по стандарту - статическое (англ. static))

он существует на протяжении выполнения всей программы.

локальное (по стандарту - автоматическое (англ. automatic))

обладают «программные объекты», область видимости которых ограничена блоком.

Такие объекты создаются при каждом входе в блок, где они определяются.

Они уничтожаются при выходе из этого «родительского» блока.

динамическое (по стандарту - выделенное (англ. allocated)).

Время жизни «выделенных» объектов длится с момента выделения памяти и заканчивается в момент ее освобождения.

Связывание:

внешнее (external);

Имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.

внутреннее (internal);

Имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.

никакое (none).

Имена без связывания принадлежат одной функции и не могут разделяться вообще.

3 Правила:

Переменные, определённые в некотором блоке, будут видны во всех блоках, вложенных в данный.

Имя переменной во вложенном блоке, совпадающее с именем переменной, определенной в блоке-предке, легально и закрывает видимость одноименной переменной-предка.

4 Время жизни, область видимости и связывание переменной зависят от места ее определения.
По умолчанию

```
int i; // глобальная переменная
      Глобальное время жизни
      Файловая область видимости
      Внешнее связывание

{
int i; } // локальная переменная
      Локальное время жизни
      Видимость в блоке
      Отсутствие связывания
```

5 Содержание объектного файла – в сущности две вещи:

- код, соответствующий определению функции в С файле
- данные, соответствующие определению глобальных переменных в С файле (для инициализированных глобальных переменных начальное значение переменной тоже должно быть сохранено в объектном файле).

При встрече объявления компилятор оставляет пустые места. Пустое место (ссылка) имеет имя, но значение соответствующее этому имени пока не известно. Далее компоновщик производит исполняемый файл, присвоив каждой ссылке на символ подходящее определение,

8. Журналирование.

План ответа:
1 Назначение
2 идеи реализации

1 Журналирование – процесс записи информации о происходящих с каким-то объектом (или в рамках какого-то процесса) событиях в журнал (например, в файл).
Этот процесс часто называют также аудитом.

2 Идеи реализации

Файловую переменную для журнала определяют глобальной и объявляют во всех файлах реализации проекта.

Это позволяет вызывать функции записи в файл для журналирования отовсюду.

```
// log.c
#include <stdio.h>
FILE *flog;
int log_init(const char* name)
{
	flog = fopen(name, "w");
	if(!flog)
		return 1;
	return 0;
}
void log_close(void)
{
	fclose(flog);
}

// log.h
#ifndef __LOG__H__
#define __LOG__H__
extern FILE *flog;
int log_init(const char*name);
void log_close(void);
#endif // __LOG__H__
```

Для записи в журнал можно написать собственные функции, где переменная-указатель на файл с журналом может быть как глобальной, так и статической переменной

```
// log.c
#include <stdio.h>
static FILE *flog;
int log_init(const char* name)
{
      flog = fopen(name, "w");
      if(!flog)   return 1;
      return 0;
}
FILE* log_get(void)
{
      return flog;
}
void log_close(void)
{
      fclose(flog);
}

// log.h
#ifndef __LOG__H__
#define __LOG__H__
#include <stdio.h>
int log_init(const char* name);
FILE* log_get(void);
void log_close(void);
#endif // __LOG__H__
```

Реализация с помощью функций с переменным числом параметров

```
// log.c
#include <stdio.h>
#include <stdarg.h>
static FILE* flog;
int log_init(const char*name)
{
      flog = fopen(name, "w");
      if(!flog)
      return 1;
      return 0;
}
void log_message(const char* format, ...)
{
      va_list args;
      va_start(args, format);
      vfprintf(flog, format, args);
      va_end(args);
}
```

```
void log_close(void)
{
      fclose(flog);
}

// log.h
#ifndef __LOG__H__
#define __LOG__H__
#include <stdio.h>
int log_init(const char* name);
void log_message(const char* format, ...);
void log_close(void);
#endif // __LOG__H__
```

9. Классы памяти.

auto

класс автоматических переменных, создаваемых при вызове функции и удаляемых после выхода из нее.
Применим только к переменным, определенным в блоке.

локальное время жизни,
видимость в пределах блока,
не имеет связывания.

register

компилятору предъявляется пожелание о том, чтобы переменная использовала какой-нибудь регистр процессора для хранения.
Как правило, компилятор игнорирует это резервное слово и способен сам решать, какой переменной можно выделять регистр процессора.
Используется только для переменных, определенных в блоке.
Задает:

- локальное время жизни,
- видимость в блоке
- отсутствие связывания.

К переменным с классом памяти register нельзя применять операцию получения адреса &.

static

класс статических переменных, создается при первом вызове функции, а удаляется только при завершении работы программы.
Компилятор хранит значения таких переменных от одного вызова функции до другого.

Для переменной вне какого-либо блока:

- static изменяет связывание этой переменной на внутреннее,
- глобальное время жизни,
- область видимости в пределах файла

Для переменной в блоке,

- static изменяет время жизни с автоматического на глобальное,
- область видимости в пределах блока,
- отсутствие связывания.

```
void f(void)
{
      static int j;
      ...
}
```

Такая переменная сохраняет свое значение после выхода из блока.
Инициализируется только один раз.
Если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.

extern

класс внешних переменных, память под них не выделяется. Это означает объявление переменной, которая может быть объявлена или в текущем, или в некотором внешнем файле.

extern int i;
Объявлений (extern int number;) может быть сколько угодно.
Определение (int number;) должно быть только одно.
Объявления и определение должны быть одинакового типа.
НО extern int number = 5; // определение

По умолчанию считается:

если переменная объявлена в теле функции или в блоке без спецификатора класса памяти, по умолчанию он равен auto;

если переменная объявлена вне всех функций, она считается внешней и может быть использована в любом смежном файле, в т. ч. и текущем;

все функции внешние.

ФУНКЦИИ:

К функциям могут применяться классы памяти static и extern.

extern int f(int i);

f имеет внешнее связывание. Может вызываться из других файлов.

static int g(int i);

g имеет внутренне связывание. Из других файлов вызываться не может.

int h(int i);

h имеет внешнее связывание (по умолчанию). Может вызываться из других файлов.

Аргументы функций по умолчанию имеют класс памяти auto.

Единственный другой класс памяти, который может использоваться с параметрами функций, - register.

10. Стек и куча.

План ответа:
1 автоматическая память: использование и реализация;
2 использование аппаратного стека (вызов функции, возврат управления из функции, передача параметров, локальные переменные, кадр стека);
3 ошибки при использовании автоматической памяти;
4 динамическая память: использование и реализация;
5 идеи реализации функций динамического выделения и освобождения памяти

1 Автоматическая память используется для хранения локальных переменных

+ Память под локальные переменные выделяет и освобождает компилятор. - Время жизни локальной переменной "ограничено" блоком, в котором она определена.
– Размер размещаемых в автоматической памяти объектов должен быть известен на этапе компиляции. – Размер автоматической памяти в большинстве случаев ограничен.

2 Аппаратный стек используется для:
1. вызова функции
    call name
        поместить в стек адрес команды, следующей за командой call
    передать управление по адресу метки name

2. возврата из функции
    ret
        извлечь из стека адрес возврата address
        передать управление на адрес address

3. передачи параметров в функцию
    соглашение о вызове:
        расположение входных данных;
        порядок передачи параметров;
        какая из сторон очищает стек;
    cdecl
        аргументы передаются через стек, справа налево;
        очистку стека производит вызывающая сторона
        результат функции возвращается через регистр EAX, но …

4. выделения и освобождения памяти под локальные переменные

Стековый кадр (фрейм) - механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека. В стековом кадре размещаются:

- значения фактических аргументов функции;
- адрес возврата;
- локальные переменные;
- иные данные, связанные с вызовом функции.

+ Удобство и простота использования.
– Передача данных через память без необходимости замедляет выполнение программы.
– Стековый кадр перемещает данные приложения с критическими данными - указателями, значениями регистров и адресами возврата.

3 Ошибки:

- возврат указателя на локальную переменную
- переполнение буфера

4-5 ??? Динамическая память

Куча представляет собой непрерывную область памяти, поделенную на занятые и свободные области (блоки) различного размера.
Информация о свободных и занятых областях кучи обычно храниться в списках различных форматов.
При запуске процесса ОС выделяет память для размещения кучи.
Алгоритм работы malloc
просматривает список занятых/свободных областей памяти, размещенных в куче, в поисках свободной области подходящего размера;
если область имеет точно такой размер, как запрашивается, добавляет найденную область в список занятых областей и возвращает указатель на начало области памяти;
если область имеет больший размер, она делится на части, одна из которых будет занята (выделена), а другая останется в списке свободных областей;
если область не удается найти, у ОС запрашивается очередной большой фрагмент памяти, который подключается к списку, и процесс поиска свободной области продолжается;
если по тем или иным причинам выделить память не удалось, сообщает об ошибке (например, malloc возвращает NULL).
Алгоритм работы free
просматривает список занятых/свободных областей памяти, размещенных в куче, в поисках указанной области;
удаляет из списка найденную область (или помечает область как свободную);
если освобожденная область вплотную граничит со свободной областью с какой-либо из двух сторон, то она сливается с ней в единую область большего размера.

11. Функции с переменным числом параметров.

План ответа:

Идея реализации

использование стандартной библиотеки

1 Фактические параметры передаются через стек в порядке, обратном их записи в вызове функции. Это позволяет сравнительно легко передавать переменное число параметров в вызываемую функцию.

Функции с переменным числом параметров объявляются так:

тип_рез-та имя(<непустой список первых параметров>, ...)

В любой функции, использующей переменное количество параметров, должен быть как минимум один реально существующий параметр.

```
int func(...); /* ошибка */
```

Идеи реализации:

1. За окончание последовательности переменных принять определенное значение

Пример

```
double avg(double a, ...)
{
	int n = 0;
	double *p_d = &a;
	double sum = 0.0;
	while (*p_d)
	{
		sum += *p_d;
		n++;
		p_d++;
	}
	if (!n) return 0;
	return sum / n;
}
```

2. В качестве аргумента функции передается количество переменных параметров

Пример

```
void print_ch(int n, ...)
{
	int *p_i = &n;
	char *p_c = (char*) (p_i+1);
	for (int i = 0; i < n; i++, p_c++)
	printf("%c %d\n", *p_c, (int) *p_c);
}
```

2 Для прохождения стека с параметрами используют заголовочный файл <stdarg.h>,

в котором объявлены:

тип va_list;

семейство функций

type va_arg(va_list ap, type)

void va_end(va_list ap)

void va_start(va_list ap, parmN) – установка указателя на стековый кадр в место нахождения фактического параметра parmN.

12. Структуры.

План ответа:
1 понятие «структура»;
2 определение структурного типа;
3 структура и ее компоненты (тэг, поле);
4 определение переменной-структуры, способы инициализации переменной-структуры;
5 операции над структурами;
6 особенности выделения памяти под структурные переменные;
7 структуры с полем переменной длины (flexible array member, C99).

1 Структура представляет собой одну или несколько переменных (возможно, разного типа), которые объединены под одним именем, при этом каждая переменных занимает своё место в памяти. Это позволяет связать в единое целое различные по типу и значению данные, связанные между собой.

2 Формат определения:

```
struct [tag]
{
      type1 field1;
      type2 [field2];
      ...;
      typeN fieldN;
} [variable] = { value1, value2, ..., valueN }
```

3 Тег структуры – имя, позволяющее идентифицировать структуру в самых разных частях программы.
Его передают в качестве спецификатора типа (вместе с ключевым словом struct) формального параметра.
Тег может быть опущен.
Тело структуры представляет собой самостоятельную область видимости: имена в этой области не конфликтуют с именами из других областей.
Поля - перечисленные в структуре переменные
Поля структуры располагаются в памяти в порядке описания.
С целью оптимизации доступа компилятор может располагать поля в памяти не одно за другим, а по адресам кратным, например, размеру поля.
Адрес первого поля совпадает с адресом переменной структурного типа.
Поля структуры могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него.

4 Инициализация
Для инициализации переменной структурного типа необходимо указать список значений, заключенный в фигурные скобки.
Значения в списке должны появляться в том же порядке, что и имена полей структуры.
Если значений меньше, чем полей структуры, оставшиеся поля

инициализируются нулями.

Инициализация в C99

```
struct date exam =
      {.date = 4, .month = 1, .year = 2016};
```

+ Такую инициализацию легче читать и понимать.
+ Значения могут идти в произвольном порядке.
+ Отсутствующие поля получают нулевые значения.
+ Возможна комбинация со старым способом.

```
struct date exam =
      {.date = 4, 1, .year = 2016};
```

5 Операции над структурами:

1) Обращение к полю по переменной: структура.поле
2) Обращение к полю по указателю на переменную: структура->поле
3) Присвоение одной структуре значений другой структуры того же типа. //корректно передаются и копируются статические массивы
4) Структуры могут передаваться в функцию в качестве параметра и возвращаться как значения.
5) Структуры НЕЛЬЗЯ сравнивать (==, !=)

6 При объявлении переменной типа заданной структуры выделяется память для всех членов структуры.

При этом память под отдельный член структуры может дополняться до 4 или 8 байт в зависимости от компилятора

Для того, чтобы память под структуру использовалась эффективно, можно воспользоваться директивой #pragma pack

```
#pragma pack(push, 1)
struct
{
      char a;
      int b;
} c2;
#pragma pack(pop)
sizeof(c2) == 5
```

7 C99: особенности использования структур - flexible array member

```
struct {int n, double d[]};
```

Подобное поле должно быть последним.

Нельзя создать массив структур с таким полем.

Структура с таким полем не может использоваться как член в «середине» другой структуры.

Операция sizeof не учитывает размер этого поля (возможно, за исключением выравнивания).

Если в этом массиве нет элементов, то обращение к его элементам – неопределенное поведение.

Особенность выделения памяти:

```
struct s *elem = malloc(sizeof(struct s) + n * sizeof(double));
```

13. Объединения.

План ответа:

1 понятие «объединение»;

2 определение переменной-объединения, способы инициализации переменной-объединения;

3 особенности выделения памяти под объединения.

4 использование объединений.Структуры.

1 Объединение – это место в памяти, которое используется для хранения переменных, разных типов.

Объединение дает возможность интерпретировать один и тот же набор битов не менее, чем двумя разными способами.

Все поля объединения разделяют одну и ту же область памяти.

2

```
union тег
{
      тип имя-члена;
      тип имя-члена;
      ...;
} переменные-этого-объединения;
```

Инициализация

```
union u_t
{
      int i;
      double d;
};
...
union u_t u_1 = {1};
// только c99
union u_t u_2 = { .d = 5.25 };
```

Присвоение значения одному члену объединения обычно изменит значение других членов.

3 Когда переменная объявляется с ключевым словом union, компилятор автоматически выделяет столько памяти, чтобы в ней поместился самый большой член нового объединения.

4 Использование

Экономия места

Создание структур данных из разных типов

Разный взгляд на одни и те же данные (машинно-зависимо). Пример:

```
union word
{
      unsigned short word;
      struct word_parts
      {
            unsigned char lo;
            unsigned char hi;
      } parts;
} a;
a.word = 0xABCD;
printf("word 0x%4x, hi part 0x%2x, lo part 0x%2x",
            a.word, a.parts.hi, a.parts.lo);
```

14. Динамический расширяемый массив.

План ответа:

1. функция realloc и особенности ее использования;
2. описание типа;
3. добавление и удаление элементов.

1. функция realloc и особенности ее использования;

```
void* realloc(void *ptr, size_t size);
```

ptr == NULL && size != 0=> Выделение памяти (как malloc)

ptr != NULL && size == 0 => Освобождение памяти аналогично free().

ptr != NULL && size != 0 => Перевыделение памяти.

В худшем случае:

выделить новую область
скопировать данные из старой области в новую
освободить старую область

```
int *p = malloc(10 * sizeof(int));

p = realloc(p, 20 * sizeof(int));// НЕВЕРНО А если realloc вернула NULL?
```

Правильно

```
int *tmp = realloc(p, 20 * sizeof(int));
if (tmp)
    p = tmp;
else
    // обработка ошибки
```

2. описание типа;

Для уменьшение потерь при распределении памяти изменение размера должно происходить относительно крупными блоками.

Для простоты реализации указатель на выделенную память должен храниться вместе со всей информацией, необходимой для управления динамическим массивом.

```
struct dyn_array
{
    int len;
    int allocated;
    int step;
    int *data;
};

#define INIT_SIZE 1
```

```
void init_dyn_array(struct dyn_array *d)
{
    d->len = 0;
    d->allocated = 0;
    d->step = 2;
    d->data = NULL;
}
```

3. добавление и удаление элементов.

```
int append(struct dyn_array *d, int item)
{
    if (!d->data)
    {
        d->data = malloc(INIT_SIZE * sizeof(int));
        if (!d->data)
            return -1;
        d->allocated = INIT_SIZE;
    }
    else
        if (d->len >= d->allocated)
        {
            int *tmp = realloc(d->data,
                        d->allocated * d->step * sizeof(int));
            if (!tmp)
                return -1;
            d->data = tmp;
            d->allocated *= d->step;
        }
    d->data[d->len] = item;
    d->len++;
    return 0;
}
```

Удвоение размера массива при каждом вызове realloc сохраняет средние «ожидаемые» затраты на копирование элемента.

Поскольку адрес массива может измениться, программа должна обращаться к элементами массива по индексам.

Благодаря маленькому начальному размеру массива, программа сразу же «проверяет» код, реализующий выделение памяти.

```
int delete(struct dyn_array *d, int index)
{
    if (index < 0 || index >= d->len)
        return -1;

    memmove(d->data + index, d->data + index + 1,
```

```
                        (d->len - index - 1) * sizeof(int));
        d->len--;

        return 0;
    }
```

«+»
- Простота использования.
- Константное время доступа к любому элементу.
- Не тратят лишние ресурсы.
- Хорошо сочетаются с двоичным поиском.

«-»
- Хранение меняющегося набора значений.

15. Линейный односвязный список.

План ответа:
1. описание типа;
2. основные операции.

1. описание типа;

head->data1->data2->data3->NULL

Отличия списка от массива
- Размер массива фиксирован, а списка нет.
- Списки можно переформировывать, изменяя несколько указателей.
- При удалении или вставки нового элемента в список адрес остальных не меняется.
- Позволяют при вставке нового элемента и удалении некоторого элемента не изменять адреса остальных элементов;
- Занимают больше места, так как каждый элемент списка должен содержать указатель на следующий элемент.

```
    struct list
    {
        int value;
        struct list *next;
    };

    struct list *create_node(int value)
    {
        struct list *node = malloc(sizeof(struct list));
        if (node)
        {
            node->value = value;
            node->next = NULL
        }
        return node;
```

```
}
```

2. основные операции.

NB: функции, изменяющие список, должны возвращать указатель на новый первый элемент.

2.1 Добавление элемента в список

```
struct list *add_front(struct list *head, struct list *node)
{
    node->next = head;
    return node;
}

struct list *add_end(struct list *head, struct list *node)
{
    struct list *cur = head;
    if (!head)
        return node;

    for (; cur->next; cur = cur->next);
    cur->next = node;

    return head;
}
```

2.2 Поиск

```
struct list *search(struct list *head, int value)
{
    for ( ; head; head = head->next)
        if (strcmp(head->value, value) == 0)
            return head;

    return NULL;
}
```

2.3 Обход всех (В данном случае печать)

```
void print(struct list *head)
{
    for (; head; head = head->next)
    {
        printf("%d ", head->value);
    }
    printf("\n");
}
```

2.4 Удаление

```
void free_all(struct list *head)
{
    struct list *next;
```

```
            for (; head; head = next)
            {
                next = head->next;
                free(head);
            }
        }
```

16. Двоичные деревья поиска.

План ответа:

1. описание типа;
2. основные операции;
3. рекурсивный и нерекурсивный поиск;
4. язык DOT.

1. описание типа;

Дерево - это связный ациклический граф.

Двоичным деревом поиска называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя.

2. основные операции;

```
struct tree_node
{
    const char *name;

    // меньшие
    struct tree_node *left;
    // большие
    struct tree_node *right;
};

struct tree_node* create_node(const char *name)
{
    struct tree_node *node = malloc(sizeof(struct tree_node));
    if (node)
    {
        node->name = name;
        node->left = NULL;
        node->right = NULL;
    }

    return node;
}

struct tree_node* insert(struct tree_node *tree, struct tree_node *node)
```

```
{
    int cmp;

    if (tree == NULL)
        return node;

    cmp = strcmp(node->name, tree->name);
    if (cmp == 0)
        assert(1);
    else if (cmp < 0)
        tree->left = insert(tree->left, node);
    else
        tree->right = insert(tree->right, node);

    return tree;
}
```

```
            f
      h           k
  a   b   g   l
```

Прямой (pre-order)
f b a d k g l
Фланговый или поперечный (in-order)
a b d f g k l
Обратный (post-order)
a d b g l k f

```
void apply(struct tree_node *tree, void (*f)(struct tree_node*, void*),
void *arg)
{
    if (tree == NULL)
        return;

    // pre-order
    // f(tree, arg);
    apply(tree->left, f, arg);
    // in-order
    f(tree, arg);
    apply(tree->right, f, arg);
    // post-order
    // f(tree, arg);
}
```

3. рекурсивный и нерекурсивный поиск;

```
//рекурсивный
struct tree_node* lookup_1(struct tree_node *tree, const char *name)
{
    int cmp;

    if (tree == NULL)
        return NULL;

    cmp = strcmp(name, tree->name);
    if (cmp == 0)
        return tree;
    else if (cmp < 0)
        return lookup_1(tree->left, name);
    else
        return lookup_1(tree->right, name);
}

struct tree_node* lookup_2(struct tree_node *tree, const char *name)
{
    int cmp;

    while (tree != NULL)
    {
        cmp = strcmp(name, tree->name);
        if (cmp == 0)
            return tree;
        else if (cmp < 0)
            tree = tree->left;
        else
            tree = tree->right;
    }

    return NULL;
}
```

4. язык DOT.

DOT – язык описания графов.

Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате.

В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например Graphviz.

```
// Описание дерева на DOT
      digraph test_tree {
      f -> b;
      f -> k;
      b -> a;
```

```
b -> d;
k -> g;
k -> l;
}
```

17. Директивы препроцессора, макросы.

План ответа:

1. классификация директив препроцессора;
2. правила, справедливые для всех директив препроцессора;
3. макросы (простые, с параметрами, с переменным предопределенные);
4. сравнение макросов с параметрами и функций;
5. скобки в макросах;
6. создание длинных макросов;
7. преопределенные макросы.

1. классификация директив препроцессора;

Макроопределения

#define, #undef

Директива включения файлов

#include

Директивы условной компиляции

#if, #ifdef, #endif и др.

Остальные директивы (#pragma, #error, #line и др.) используются реже.

2. правила, справедливые для всех директив препроцессора;

Директивы всегда начинаются с символа "#".

Любое количество пробельных символов может разделять лексемы в директиве.

```
#  define   N   1000
```

Директива заканчивается на символе '\n'.

Директива заканчивается на символе '\n'.

```
#define DISK_CAPACITY   (SIDES *        \
                TRACKS_PER_SIDE *    \
                SECTORS_PER_TRACK *  \
                BYTES_PER_SECTOR)
```

Директивы могут появляться в любом месте программы.

3. макросы (простые, с параметрами, с переменным предопределенные);

3.1 простые

```
#define идентификатор список-замены

#define PI 3.14
#define EOS '\0'
#define MEM_ERR "Memory allocation error."
```

Используются:

В качестве имен для числовых, символьных и строковых констант.

Незначительного изменения синтаксиса языка.
```
#define BEGIN {
#define END }
#define INF_LOOP for( ; ; )
```

Переименования типов.
```
#define BOOL int
```

Управления условной компиляцией.

3.2 с параметрами
```
#define идентификатор(x1, x2, ..., xn) список-замены
```

Не должно быть пробела между именем макроса и (.
Список параметров может быть пустым.

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

Где-то в программе
```
i = MAX(j + k, m - n);        // i = ((j + k) > (m - n) ? (j + k)
: (m - n));
```

3.3 с переменным предопределенные
```
#ifndef NDEBUG
#define DBG_PRINT(s, ...) printf(s, __VA_ARGS__)
#else
#define DBG_PRINT(s, ...) ((void) 0)
#endif
```

4. сравнение макросов с параметрами и функций;

Преимущества
- программа может работать немного быстрее;
- макросы "универсальны".

Недостатки
- скомпилированный код становится больше; n = MAX(i, MAX(j, k)));
- типы аргументов не проверяются;
- нельзя объявить указатель на макрос;
- макрос может вычислять аргументы несколько раз. n = MAX(i++, j);

Общие свойства макросов
- Список-замены макроса может содержать другие макросы.
- Препроцессор заменяет только целые лексемы, не их части.
- Определение макроса остается «известным» до конца файла, в котором этот макрос объявляется.

Макрос не может быть объявлен дважды, если эти объявление не тождественны.

Макрос может быть «разопределен» с помощью директивы #undef.

5. скобки в макросах;

Если список-замены содержит операции, он должен быть заключен в скобки.

Если у макроса есть параметры, они должны быть заключены в скобки в списке-замены.

```
#define TWO_PI  2 * 3.14

f = 360.0 / TWO_PI;
// f = 360.0 / 2 * 3.14;

#define SCALE(x)  (x * 10)

j = SCALE(i + 1);
// j = (i + 1 * 10);
```

6. создание длинных макросов;

```
// 1
#define ECHO(s)   {gets(s); puts(s);}

if (echo_flag)
    ECHO(str);
else
    gets(str);

// 2
#define ECHO(s)  (gets(s), puts(s))
ECHO(str);

// 3
#define ECHO(s) \
do              \
{               \
    gets(s);    \
    puts(s);    \
}               \
while(0)        \
```

7. преопределенные макросы.

Эти идентификаторы нельзя переопределять или отменять директивой undef.

__LINE__ - номер текущей строки (десятичная константа)

__FILE__ - имя компилируемого файла
__DATE__ - дата компиляции
__TIME__ - время компиляции и др.
__func__ - имя функции как строки (GCC only, C99 и не макрос)

18. Директивы препроцессора, условная компиляция, операции # и ##.

План ответа:
1. классификация директив препроцессора;
2. правила, справедливые для всех директив препроцессора;
3. директивы условной компиляции, использование условной компиляции;
4. директива error;
5. операция “#”,
6. операция “##”;
7. директива pragma (once, pack).

1. классификация директив препроцессора;
Макроопределения
#define, #undef
Директива включения файлов
#include
Директивы условной компиляции
#if, #ifdef, #endif и др.
Остальные директивы (#pragma, #error, #line и др.) используются реже.

2. правила, справедливые для всех директив препроцессора;
Директивы всегда начинаются с символа "#".
Любое количество пробельных символов может разделять лексемы в директиве.
```
#  define   N   1000
```
Директива заканчивается на символе '\n'.
Директива заканчивается на символе '\n'.
```
#define DISK_CAPACITY   (SIDES *        \
                TRACKS_PER_SIDE *    \
                SECTORS_PER_TRACK *  \
                BYTES_PER_SECTOR)
```

Директивы могут появляться в любом месте программы.

3. директивы условной компиляции, использование условной компиляции;
Директивы #if, #else, #elif и #endif

```
#if defined(OS_WIN)
...
#elif defined(OS_LIN)
...
#elif defined(OS_MAC)
```

```
...
#endif
```

Возможно, самыми распространенными директивами условной компиляции являются #if, #else, #elif и #endif.
Они дают возможность в зависимости от значения константного выражения включать или исключать те или иные части кода.

Использование условной компиляции:
- программа, которая должна работать под несколькими операционными системами;
- программа, которая должна собираться различными компиляторами;
- начальное значение макросов;
- временное выключение кода.

4. директива error;

```
#error сообщение

#if defined(OS_MAC)
...
#else
#error Unsupported OS!
#endif
```

5. операция “#”,

«Операция» # конвертирует аргумент макроса в строковый литерал.

```
#define PRINT_INT(n) printf(#n " = %d\n", n)

#define TEST(condition, ...) ((condition) ?        \
         printf("Passed test %s\n", #condition) :  \
                                printf(__VA_ARGS__))

Где-то в программе

PRINT_INT(i / j);
// printf("i/j" " = %d", i/j);

TEST(voltage <= max_voltage,
      "Voltage %d exceed %d", voltage, max_voltage);
```

6. операция “##”;

«Операция» ## объединяет две лексемы в одну.

```
#define MK_ID(n)  i##n

Где-то в программе

int MK_ID(1), MK_ID(2);
```

```
// int i1, i2;
```

Более содержательный пример

```
#define GENERAL_MAX(type)         \
type type##_max(type x, type y)   \
{                                 \
    return x > y ? x : y;         \
}
```

7. директива pragma (once, pack).

Директива #pragma позволяет добиться от компилятора специфичного поведения.

Каждая реализация C поддерживает некоторые функции, уникальные для хост-компьютера или операционной системы.

Некоторые программы, например, должны осуществлять точный контроль над областями памяти, где размещаются данные, или контролировать способ получения параметров определенными функциями.

Директивы #pragma предлагают каждому компилятору возможность предоставлять функции для конкретного компьютера и операционной системы, сохраняя общую совместимость.