19. inline-функции.

1. inline – пожелание компилятору заменить вызовы функции последовательной вставкой кода самой функции.

inline-функции по-другому называют встраиваемыми или подставляемыми.

```
inline double average(double a, double b)
{
    return (a + b) / 2;
}
```

В C99 inline означает, что определение функции предоставляется только для подстановки и где-то в программе должно быть другое такое же определение этой же функции.

```
inline int add(int a, int b) {return a + b;}

int main(void)
{
    int i = add(4, 5);
    return i;
}
// main.c:(.text+0x1e): undefined reference to `add'
// collect2.exe: error: ld returned 1 exit status
```

2.Способы исправления проблемы «unresolved reference»

2.1. Использовать ключевое слово static

```
static inline int add(int a, int b) {return a + b;}
int main(void)
{
    int i = add(4, 5);
    return i;
}
```

2.2. Убрать ключевое слово inline из определения функции.

```
int add(int a, int b) {return a + b;}
int main(void)
{
    int i = add(4, 5);
    return i;
}
```

Компилятор «умный» :), сам разберется.

3.3. Добавить еще одно такое же не-inline определение функции где-нибудь в программе.

20. Списки из ядра операционной системы Linux (списки Беркли).

План ответа:

1. идеи реализации (циклический двусвязный список, интрузивный список,универсальный список);
2. описание типа;
3. добавление элемента в начало и конец (list_add, list_add_tail), итерирование по списку(list_for_each, list_for_each_entry), освобождение памяти (list_for_each_safe);
4. особенности реализации макроса list_entry.

1. идеи реализации (циклический двусвязный список, интрузивный список,универсальный список);

Список Беркли – это циклический двусвязный список, в основе которого лежит следующая структура:

```
struct list_head
{
    struct list_head *next, *prev;
};
```

В отличие от обычных списков, где данные содержатся в элементах списка, структура list_head должна быть частью сами данных

Для оптимизации работы над списками их делают кольцевыми, то есть head->prev == last && last->next == head.

Поскольку двусвязный кольцевой список не имеет настоящих начала и конца, операции создания списка, добавления и удаления элементов имеют сложность O(1), а очистка списка, как обычно, O(N).

Циклический двусвязный список:

```
struct list
{
  int field; // поле данных
  struct list *next; // указатель на следующий элемент
  struct list *prev; // указатель на предыдущий элемент
};
```

Интрузивный список: (Пример интрузивного списка пар целых чисел)

```
struct list_link
{
      element *prev, *next;
};

struct element
{
      int x, y;
      list_link link;
};
```

Универсальный список:

2. описание типа;

```
#include “list.h”

struct data
{
int num;
struct list_head list;
};
```

Следует отметить следующее:

Структуру struct list_head можно поместить в любом месте в определении структуры.

struct list_head может иметь любое имя.

В структуре может быть несколько полей типа struct list_head.

3. добавление элемента в начало и конец (list_add, list_add_tail), итерирование по списку **(list_for_each, list_for_each_entry), освобождение памяти (list_for_each_safe);**

```
static inline void __list_add(struct list_head *new, struct list_head *prev, struct list_head *next)
{
	next->prev = new;
	new->next = next;
	new->prev = prev;
	prev->next = new;
}

static inline void list_add(struct list_head *new, struct list_head *head)
{
	__list_add(new, head, head->next);
}

static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
	__list_add(new, head->prev, head);
}

#define list_for_each(pos, head) \
for (pos = (head)->next; pos != (head); pos = pos->next)


#define list_for_each_entry(pos, head, member) \
	for (pos = list_entry((head)->next, typeof(*pos), member); \
	     &pos->member != (head); \
	     pos = list_entry(pos->member.next, typeof(*pos), member))
```

```
#define list_for_each_safe(pos, n, head) \
      for (pos = (head)->next, n = pos->next; pos != (head); \
            pos = n, n = pos->next)
```

4. особенности реализации макроса list_entry.

```
#define list_entry(ptr, type, member) \
container_of(ptr, type, member)

#define container_of(ptr, type, field_name) ( \
   (type *) ((char *) (ptr) - offsetof(type, field_name)))

#define offsetof(TYPE, MEMBER) \
     ((size_t) &((TYPE *)0)->MEMBER)
```

21. Битовые операции. Битовые поля.

План ответа:

1. битовые операции: сдвиг влево, сдвиг вправо, битовое «НЕ», битовое «И», битовое «исключающее ИЛИ», битовое «ИЛИ» и соответствующие им операции составного присваивания;
2. использование битовых операций для обработки отдельных битов и последовательностей битов;
3. различие между битовыми и логическими операциями;
4. битовые поля: описание, использование, ограничения использования.

1. битовые операции: сдвиг влево, сдвиг вправо, битовое «НЕ», битовое «И», битовое «исключающее ИЛИ», битовое «ИЛИ» и соответствующие им операции составного присваивания;

Битовые операции — логические операции, проводимые над каждым битом фигурирующих операндов.
Операнды при этом имеют целый тип и одинаковый размер.

Язык Си поддерживает все битовые операции:

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
~(унар.)	Побит. «НЕ»	~X	Префиксные	15	Справа налево
<<	Сдвиг влево	X << Y	Инфиксные	11	Слева направо
>>	Сдвиг вправо	X >> Y	Инфиксные	11	Слева направо
&	Побитовое «И»	X & Y	Инфиксные	8	Слева направо
^	искл. «ИЛИ»	X ^ Y	Инфиксные	7	Слева направо
|	Побитовое «ИЛИ»	X | Y	Инфиксные	6	Слева направо

Инфиксные 2 слева направо

<<= Присваивание со сдвигом влево X <<= Y
>>= Присваивание со сдвигом вправо X >>= Y
&= Присваивание с побитовым «И» X &= Y
^= Присваивание с побитовым исключающим «ИЛИ» X ^= Y
|= Присваивание с побитовым «ИЛИ» X |= Y

Битовые операции применимы только к целочисленным переменным.
Битовые операции обычно выполняют над беззнаковыми целыми, чтобы не было путаницы со знаком.

2. использование битовых операций для обработки отдельных битов и последовательностей битов;

2.1. & -- поразрядная конъюнкция.
На каждом бите проходит конъюнкция/логическое И.

Проверка битов операнда на установление:
Формируем mask, в которой на проверяемые биты установлена единица, а на остальные биты – ноль. В результате op&mask останутся единицы только в значениях проверяемых с помощью mask битов там, где они установлены в op.

Установка битов операнда в ноль (сброс флагов)
Формируем mask, в которой нужные биты установлены в ноль, а остальные – в один. В результате op&mask появятся нули только в устанавливаемых с помощью mask битов op, остальные биты останутся прежними.

2.2. | -- поразрядная дизъюнкция.
На каждом бите происходит дизъюнкция/логическое ИЛИ

Установка битов операнда в единицу (установка флагов):
Формируем mask, в которой нужные биты установлены в единицу, а остальные – в ноль. В результате op|mask появятся единицы только в устанавливаемых с помощью mask битов op, остальные биты останутся прежними.
Проверка битов операнда на сброс проводится аналогично п2.1.2

2.3. ^ -- поразрядная симметрическая разность.
На каждом бите операндов происходит симметрическая разность/сумма по модулю 2/логическое ИСКЛЮЧАЮЩЕЕ ИЛИ/АНТИЭКВИВАЛЕНТНОСТЬ.

Смена значений битов операнда
Формируем mask, в которой нужные биты установлены в единицу, а остальные – в ноль. В результате op^mask изменятся значения только в устанавливаемых с помощью mask битов op.

2.4. ~ -- поразрядная инверсия/логическое НЕ.
Каждый бит операнда изменяет своё значение. При этом результат – обратный код числа op.

2.5. Логические сдвиги числа на некоторое число бит.
Сдвиг влево.
Синтаксис: op<<n_bit. Все биты сдвигаются на n_bit позиций влево, освободившиеся биты заполнятся нулями, а выдвинутые биты уничтожаются.
Сдвиг вправо.
Синтаксис: op>>n_bit. Все биты сдвигаются на n_bit позиций

вправо, освободившиеся биты заполнятся нулями, а выдвинутые биты уничтожаются.

3. различие между битовыми и логическими операциями;

При проведении логических операций все ненулевые числа (как бы) приводятся к числу (-1), представимому всеми единицами в двоичной записи. Поэтому 00100100&10010010 == 00000000 != (некая единица) == 00100100&&10010010.

4. битовые поля: описание, использование, ограничения использования.

4.1. Стандартный вид объявления:

```
struct имя_структуры
{
тип имя1: длина;
тип имяN: длина;
};
```

Битовые поля должны объявляться как целые, unsigned или signed.

4.2. Представление в памяти:

обычным целым числом, размера не менее суммы размеров полей битового поля (контролируется программистом)

4.3. Необходимые действия над битовыми полями:

4.3.1 Извлечение поля из структуры включает следующую последовательность действий:

4.3.1.1 Конъюнкция с маской битового поля (на битах поля единицы, в остальных местах нули);

4.3.1.2 Побитовый сдвиг вправо.

4.3.2 Сборка одного числа из битовых полей:

4.3.2.1 Обнуление числа

4.3.2.2 Установка битов поля

4.3.2.3 Сдвиг влево

4.3.3 Замена битового поля

4.3.2.1 Обнуление нужных битов с помощью маски

4.3.2.2 Заполнение нужных битов (со сдвигом) поразрядной дизъюнкцией

4.4. Замечание.

Битовые поля применяются тогда, когда нужно компактно записать множество разнообразной перечислимой информации, экономя при этом место в памяти, но не гонясь за сверхвысокой производительностью алгоритма вопреки стремлению написать более читабельный код.

Если требуется более высокая производительность, применяются обычные числа, при этом отдельно прописываются всевозможные маски и значения перечислимых параметров.

22. Неопределенное поведение

План ответа:

1. особенности вычисления выражений с побочным эффектом;
2. понятие «точка следования»;
3. расположение «точек следования».
4. More (этого в плане не было, но упомянуть явно нужно)

- преимущества и недостатки неопределенного поведения;
- способы борьбы с неопределенным поведением.

1. особенности вычисления выражений с побочным эффектом;

Модификация данных.

Обращение к переменным, объявленным как volatile.

Вызов системной функции, которая производит побочные эффекты (например, файловый ввод или вывод).

Вызов функций, выполняющих любое из вышеперечисленных действий.

Примеры неопределенного поведения

Использование неинициализированных переменных.

Переполнение знаковых целых типов.

Выход за границы массива.

Использование «диких» указателей.

2. понятие «точка следования»;

Компилятор вычисляет выражения. Выражения будут вычисляться почти в том же порядке, в котором они указаны в исходном коде: сверху вниз и слева направо.

Точка следования – это точка в программе, в которой программист знает какие выражения (или подвыражения) уже вычислены, а какие выражения (или подвыражения) еще нет.

3. расположение «точек следования».

Определены следующие точки следования:

3.1 Между вычислением левого и правого операндов в операциях &&, || и "," .

```
*p++ != 0 && *q++ != 0
```

3.2 Между вычислением первого и второго или третьего операндов в тернарной операции.

```
a = (*p++) ? (*p++) : 0;
```

3.3 В конце полного выражения.

```
a = b;
if ()
switch ()
while ()
do{} while()
for ( x; y; z)
return x
```

3.4 Перед входом в вызываемую функцию.

Порядок, в котором вычисляются аргументы не определен, но эта точка следования гарантирует, что все ее побочные эффекты проявятся на момент входа в функцию.

3.5 В объявлении с инициализацией на момент завершения вычисления инициализирующего значения.

int a = (1 + i++);

- Почему результата выражения «x[i] = i++ + 1;» не определен?

Порядок вычисления выражений и подвыражений между точками следования не определен.

В выражении «x[i] = i++ + 1;» есть единственная точка следования, которая находится в конце полного выражения.

В выражении «x[i] = i++ + 1;» есть два обращения к переменной i. Множественный доступ и является источником проблемы.

- Почему спецификация языка оставляет открытым вопрос, в каком порядке компиляторы должны вычислять выражения между точками следования?

Причина неопределенности порядка вычислений – простор для оптимизации.

Виды

Undefined behavior. Такое поведение возникает как следствие неправильно написанной программы. Стандарт ничего не требует, может случиться все что угодно.

Unspecified behavior. Стандарт предлагает несколько вариантов на выбор. Компилятор может реализовать любой вариант. При этом на вход компилятора подается корректная программа. Например: все аргументы функции должны быть вычислены до вызова функции, но они могут быть вычислены в любом порядке.

Преимущества неопределенного поведения:

1. освободить разработчиков компиляторов от необходимости обнаруживать ошибки, которые трудно диагностировать.
2. избежать предпочтения одной стратегии реализации другой.
3. отметить области языка для расширения языка (language extension)
4. Ускоряется работа программ (так как не нужно проверять всевозможные «маргинальные» случаи).

Недостатки:

1. Не гарантирует полной совместимости различных реализаций языка.
2. Недопущение ситуаций неопределённого поведения остаётся за программистом.

4. Способы борьбы с неопределенным поведением

Включайте все предупреждения компилятора, внимательно читайте их.
Используйте возможности компилятора (-ftrapv).
Используйте несколько компиляторов.
Используйте статические анализаторы кода (например, clang).
Используйте инструменты такие как valgrind, Doctor Memory и др.
Используйте утверждения.

23. Библиотеки (Не уверен, что полностью ответил на вопросы)

План ответа:

1. Статические и динамические библиотеки: назначение, особенности использования,анализ преимуществ и недостатков.
2. Статические библиотеки: создание, использование при компоновке.
3. Динамические библиотеки: создание (экспорта и импорта функций, динамическая компоновка и динамическая загрузка.
4. Особенности использования динамических библиотек с приложением, реализованным на другом (по отношению к библиотеке) языке программирования

1.1 Статические библиотеки

Связываются с программой в момент компоновки. Код библиотеки помещается в исполняемый файл.

«+»

- Исполняемый файл включает в себя все необходимое.
- Не возникает проблем с использованием не той версии библиотеки.

«-»

- «Размер».
- При обновлении библиотеки программу нужно пересобрать.

1.2 Динамические библиотеки

Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

«+»

- Несколько программ могут «разделять» одну библиотеку.
- Меньший размер приложения (по сравнению с приложением со статической библиотекой).
- Средство реализации плагинов.
- Модернизация библиотеки не требует перекомпиляции программы.
- Могут использовать программы на разных языках.

«-»

- Требуется наличие библиотеки на компьютере.

Версионность библиотек.

Способы использования динамических библиотек
динамическая компоновка;
динамическая загрузка.

2. Использование статической библиотеки. Сборка библиотеки
компиляция
gcc -std=c99 -Wall -Werror -c arr_lib.c
упаковка
ar rc libarr.a arr_lib.o
индексирование
ranlib libarr.a
Сборка приложения
gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe ИЛИ
gcc -std=c99 -Wall -Werror main.c -L. -larr -o test.exe

3.1 Использование динамической библиотеки (динамическая компоновка)

Сборка библиотеки
компиляция
gcc -std=c99 -Wall -Werror -c arr_lib.c
компоновка
gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll

Сборка приложения
gcc -std=c99 -Wall -Werror -c main.c
gcc main.o -L. –larr -o test.exe

3.2 Использование динамической библиотеки (динамическая загрузка)

Сборка библиотеки
компиляция
gcc -std=c99 -Wall -Werror -c arr_lib.c
компоновка
gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll

Сборка приложения
gcc -std=c99 -Wall -Werror main.c –o test.exe

3.3 Экспорт функций:

Чтобы загрузить библиотеку необходимо создать объект класса CDLL:
import ctypes
lib = ctypes.CDLL('example.dll')
Классов для работы с библиотеками в модуле ctypes несколько:
CDLL (cdecl и возвращаемое значение int);
OleDLL (cdecl и возвращаемое значение HRESULT);
WinDLL (cdecl и возвращаемое значение int).

Класс выбирается в зависимости от соглашения о вызовах, которое использует библиотека.

```
# int add(int, int)
add = lib.add
add.argtypes = (ctypes.c_int, ctypes.c_int)
add.restype = ctypes.c_int
```

Чтобы интерпретатор Python смог правильно конвертировать аргументы, вызвать функцию add и вернуть результат ее работы, необходимо указать атрибуты argtypes и restype.

4. Особенности использования динамических библиотек с приложением

Целые числа в Python «неизменяемые» объекты. Попытка их изменить вызовет исключение. Поэтому для аргументов, которые «используют» указатель, необходимо с помощью описанных в модуле ctypes совместимых типов создать объект и передать именно его

В языке Си используются идиомы, которых нет в языке Python. Например, функция divided возвращает одно из значений через свой аргумент. Поэтому решение «в лоб» обречено на неудачу.

24. Абстрактный тип данных

План ответа:

1. Понятие «модуль», преимущества модульной организации программы.
2. Разновидности модулей.
3. Организация модуля в языке Си. Неполный тип в языке Си.
4. Общие вопросы проектирования абстрактного типа данных.

1. Модуль

Программу удобно рассматривать как набор независимых модулей.

Модуль состоит из двух частей: интерфейса и реализации.

Интерфейс описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.

Реализация описывает, как модуль выполняет то, что предлагает интерфейс.

У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько.

Часть кода, которая использует модуль, называют клиентом.

Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

Преимущества использования модулей:

Абстракция (как средство борьбы со сложностью)

Когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику.

Повторное использование

Модуль может быть использован в другой программе.

Сопровождение

Можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу.

Типы модулей:

Набор данных

Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (float.h, limits.h.)

Библиотека

Набор связанных функций.

Абстрактный объект

Набор функций, который обрабатывает скрытые данные.

Абстрактный тип данных

Абстрактный тип данных – это интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

3. Организация модуля в языке Си. Неполный тип в языке Си

В языке Си интерфейс описывается в заголовочном файле (*.h).

В заголовочном файле описываются макросы, типы, переменные и функции, которые клиент может использовать.

Клиент импортирует интерфейс с помощью директивы препроцессора include.

Реализация интерфейса в языке Си представляется одним или несколькими файлами с расширением *.c.

Реализация определяет переменные и функции, необходимые для обеспечения возможностей, описанных в интерфейсе.

Реализация обязательно должна включать файл описания интерфейса, чтобы гарантировать согласованность интерфейса и реализации.

Неполный тип в языке Си:

Стандарт Си описывает неполные типы как «типы которые описывают объект, но не предоставляют информацию нужную для определения его размера».

struct t;

Пока тип неполный его использование ограничено.

Описание неполного типа должно быть закончено где-то в программе.

Допустимо определять указатель на неполный тип

typedef struct t *T;

Можно

определять переменные типа T;
передавать эти переменные как аргументы в функцию.

Нельзя

применять операцию обращения к полю (->);
разыменовывать переменные типа T.

4. Общие вопросы проектирования абстрактного типа данных

Именование

В примерах использовались имена функций, которые подходят для многих АТД (create, destroy, is_empty). Если в программе будет использоваться несколько разных АТД, это может привести к конфликту. Поэтому имеет смысл добавлять название АТД в название функций (stack_create, stack_destroy, stack_is_empty).

Обработка ошибок

-Интерфейс это своего рода контракт.

- Интерфейс обычно описывает проверяемые ошибки времени выполнения и непроверяемые ошибки времени выполнения и исключения.
- Реализация не гарантирует обнаружение непроверяемых ошибок времени выполнения. Хороший интерфейс избегает таких ошибок, но должен описать их.
- Реализация гарантирует обнаружение проверяемых ошибок времени выполнение и информирование клиентского кода.

«Общий» АТД

- Хотелось бы чтобы стек мог «принимать» данные любого типа без модификации файла stack.h.
- Программа не может создать два стека с данными разного типа.

Решение – использовать void* как тип элемента, НО:

- элементами могут быть динамически выделяемые объекты, но не данные базовых типов int, double;
- стек может содержать указатели на что угодно, очень сложно гарантировать правильность.

25. Чтение сложных объявлений.

1. Правила чтения сложных объявлений:

Основные конструкции объявлений:

type [] – массив типа type

type [N] – массив N элементов типа type

type (type1, type2, …) – функция, принимающая параметры типа type1, type2, …, и возвращающая результат типа type

type* -- указатель на type

Приоритеты операций:

Скобки всегда предпочтительнее звёздочек, т.е. выражение char** [] следует читать как «адрес массива указателей на указатель на char»

Круглые скобки всегда предпочтительнее квадратных

Круглые скобки часто ставят для изменения приоритета операций

(пример: int(*func)(void*) - указатель на функцию, возвращающую int

int *func(void*) функция, …, возвращающая указатель на int

Чтение идёт «от самого глубокого идентификатора» «изнутри наружу», то есть из центра скобочной последовательности к краям,

стирая по пути дешифрования объявления пары скобок.

2. Недопустимые конструкции в объявлениях:

Массив функций int a[10](int) (именно функций, массив указателей на функции допустим)

Функция не может вернуть функцию: int g(int)(int)

Функция не может вернуть массив: int f(int)[]

В массиве только левая лексема может быть пустой (или переменной)

Тип void может применяться только вкупе с указателем (звёздочкой) или как результат работы функции.

Не допускаются объявления типа void x; void x[5];

Примеры:

int *(*x[10])(void);

x - массив из десяти элементов типа указатель на функцию, которая не принимает параметров, возвращающую указатель на int

char *(*(**f[][8])())[];

f - массив типа массив из 8 элементов типа указатель на указатель на функцию без параметров, возвращающую указатель на массив типа указатель на char

void (*signal(int, void (*fp)(int)))(int);

fp - указатель на функцию, принимающую параметр типа int и

ничего не возвращающую

signal - функция, принимающая параметры типа int и fp, и возвращающая указатель на функцию, принимающую параметр типа int и ничего не возвращающую.