Организация программ. Утилита make. Статические и динамические библиотеки.

Организиция программного проекта

Для демонстрации общих подходов к разработке программ и организации кода воспользуемся имеющимся в нашем распоряжении кодом для работы с однонаправлеными списками. Основные функции оставим без изменения:

```
#include <stdio.h>
#include <stdlib.h>

struct list
{
    int data;
    struct list *next;
};

void print_list(struct list *head)
{
    struct list *ptr = head;
    int counter = 0;
    while(ptr != NULL)
    {
        printf("list[%d] = %d", counter++, ptr->data);
        ptr = ptr->next;
    }
    printf("\n");
}

struct list *create_new_list_elem(int data)
{
    struct list *res_ptr = NULL;
    res_ptr = (struct list*)malloc(sizeof(struct list));
    res_ptr->data = data;
    res_ptr->next = NULL;

    return res_ptr;
}

void add_elem_to_list(struct list *head, struct list *new_elem)
{
    struct list *ptr = head;
    while(ptr->next != NULL)
        ptr = ptr->next;

    ptr->next = new_elem;
}

void clear_list(struct list *head)
{
    struct list *ptr = head;
    while(ptr != NULL)
    {
        struct list *elem_to_remove = ptr;
        ptr = ptr->next;
        free(elem_to_remove);
    }
}
```

Внесём изменения в функцию main, сделав её более осмысленной. Предположим, мы хотим запрашивать у пользователя размер однонаправленного списка, а так же запрашивать значения элементов и выводить их на экран, не забывая при этом освобождать память.

Код функции main для этого будет следующий:

```
int main()
{
    int size;
    printf("Enter size: ");
    scanf("%d", &size);

    struct list *head_list = NULL; //указатель для хранения «головы» списка.

    for(int i = 0; i < size; i++)
    {
        int data = 0;
        printf("Enter number: ");
        scanf("%d", &data);

        struct list *new_elem = create_new_list_elem(data);
        if(head_list == NULL)       //Если указатель на начало пуст.
            head_list = new_elem;   //Делаем «головой» созданный элемент
        else
            add_elem_to_list(head_list, new_elem);//Добавляем к списку.
    }

    print_list(head_list);    //Печатаем список

    clear_list(head_list);    //Очищаем память

    return 0;
}
```

Заметим, что при написании, мы не вносили никаких изменений в уже существовавшие функции по работе со списками. Тем не менее, все время они находились в исходном тексте нашей программы. В одном конкретном случае это не создает серьёзных неудобств, но разрабатывая другую программу, работающую со списками, придётся копировать этот текст снова, что будет уже неудобно. Хорошо бы выделить всё, что касается работы со списками, в отдельный файл и по необходимости вносить изменения только в него.

Сделать это необходимо правильно, поскольку если мы просто скопируем все функции в отдельный файл my_list.c и попытаемся собрать программу, то получим ошибки.

my_list.c

```
#include <stdio.h>
#include <stdlib.h> //malloc();

struct list
{
    int data;
    struct list *next;
};

void print_list(struct list *head)
{
```

```
    struct list *ptr = head;
    int counter = 0;
    while(ptr != NULL)
    {
        printf("list[%d] = %d", counter++, ptr->data);
        ptr = ptr->next;
    }
    printf("\n");
}

struct list *create_new_list_elem(int data)
{
    struct list *res_ptr = NULL;
    res_ptr = (struct list*)malloc(sizeof(struct list));
    res_ptr->data = data;
    res_ptr->next = NULL;

    return res_ptr;
}

void add_elem_to_list(struct list *head, struct list *new_elem)
{
    struct list *ptr = head;
    while(ptr->next != NULL)
        ptr = ptr->next;

    ptr->next = new_elem;
}

void clear_list(struct list *head)
{
    struct list *ptr = head;
    while(ptr != NULL)
    {
        struct list *elem_to_remove = ptr;
        ptr = ptr->next;
        free(elem_to_remove);
    }
}
```

main.c

```
#include <stdio.h> //В int main() используется printf() и scanf().

int main()
{
    int size;
    printf("Enter size: ");
    scanf("%d", &size);

    struct list *head_list = NULL;

    for(int i = 0; i < size; i++)
    {
        int data = 0;
        printf("Enter number: ");
        scanf("%d", &data);

        struct list *new_elem = create_new_list_elem(data);
        if(head_list == NULL)
            head_list = new_elem;
        else
            add_elem_to_list(head_list, new_elem);
    }
```

```
    print_list(head_list);

    clear_list(head_list);

    return 0;
}
```

Если мы попытаемся откомпилировать main.c, то получим ошибки двух типов.

```
$ gcc main.c -o list
```

Первая (хоть и warning, но на самом деле ошибка):

```
»  warning: implicit declaration of function
‘create_new_list_elem’
```

Вторая:

```
 » undefined reference to `create_new_list_elem'
```

С ошибками первого типа мы сталкивались, когда использовали `void* malloc(…)` без добавления `#include<stdlib.h>`. Следовательно, что-то похожее необходимо и в этом случае. Этим необходимым является заголовочный файл, содержащий прототипы функций работы со списками.

Прототип функции — объявление имени функции и типов её параметров без определения её кода. Например, у нас имеется файл:

```
#include <stdio.h>

int main()
{
    foo();
    return 0;
}

void foo()
{
    printf("Hello world\n");
}
```

Если мы попробуем откомпилировать данную программу, то получим похожее сообщение об ошибке:

```
>> warning: implicit declaration of function ‘foo’
```

Добавим перед `int main()` строчку:

```
void foo();
```

Теперь компиляции проходит без ошибок. Иными словами, после объявления прототипа компилятор «знает» о существовании этой функции, и порядок их определений (программного описания) уже не важен.

Создадим заголовочный файл:

my_list.h

```
//Прототипы функций.
void print_list(struct list *head);
struct list *create_new_list_elem(int data);
void add_elem_to_list(struct list *head, struct list *new_elem);
void clear_list(struct list *head);

//Добавление определений структур способствует читабельности кода
//Допустимо также объявить структуру как struct list;
//Описание полей при этом сделать my_list.c
struct list
{
    int data;
    struct list *next;
};
```

Подключим этот файл в `main.c` (так же его необходимо включить в `my_list.c`)

```
#include "my_list.h"
```

Кавычки указывают здесь, что в первую очередь заголовочный файл необходимо искать в той же директории, что и исходный файл, или в директории, указанной флагом компиляции «-I»

`(gcc foo.c -I/path/to/find/header/files)`

Теперь при компиляции `main.c` останутся только ошибки второго типа (*undefined reference*).

Мы уже давно сталкивались с этими ошибками при рассмотрении работы линковщика. Они означают, что линковщик не может найти готовый машинный код для используемых в `main.c` функций.

Чтобы этот машинный код получить, необходимо сперва скомпилировать `my_list.c` с флагом «-c» и получить объектный файл.

`gcc -c my_list.c -o my_list.o`

И добавить этот файл для компиляции `main.c`

`gcc main.c my_list.o -o list`

Теперь компиляция проходит без ошибок и предупреждений.

Главное неудобство здесь состоит в том, что для сборки программы необходимо выполнять два шага (сколько таких шагов в больших программах?). Попробуйте внести изменения в функции работы со списками и перекомпилировать только `main.c`. В результате вы не увидите работы внесенных изменений.

Хорошо бы иметь инструмент, который бы следил за внесёнными изменениями и пересобирал только части проекта зависимые от этих изменений. Такой инструмент есть и называется **`make`**.

Утилита make

Утилита make «следит» за изменениями в программных файлах и осуществляет пересборку обновленных и всех зависимых от них файлов программы.

На вход утилиты make поступает текстовый make-файл. Чаще всего он называется *Makefile*.

Make-файл определяет зависимости между файлами программы и действия, необходимые для сборки. Эти зависимости называются правилами. Общая структура следующая:

цель : зависимости

    скрипт

Целью может являться объект, получаемый в результате работы *скрипта*. В *зависимостях* могут быть описаны файлы, необходимые для достижения цели, или даже другие *цели*. Таким образом make-файл позволяет строить достаточно сложные зависимости.

В нашем случае make-файл будет выглять достаточно скромно:

```
list: my_list.o
    gcc main.c my_list.o -o list


my_list.o: my_list.c my_list.h
    gcc -c my_list.c -o my_list.o
```

Прочитать можно следующим образом:

- Чтобы собрать программу `list`, нам необходим обновленный объектный файл `my_list.o` (Тогда мы сможем добавить его на вход компилятора)
- Чтобы собрать объектный файл my_list.o, нам необходимо иметь самые последние `my_list.c` и `my_list.h`, тогда мы сможем скомпилировать объектный файл.

Если мы внесем изменения в `my_list.c`, то получиться, что файл в *зависимостях* новее файла *цели* (`my_list.o`), поэтому make запустит *скрипт* для компиляции. По этой же причине будет перекомпилирована программа `list`.

Второе удобство заключается в том, что, определив зависимости в Make-файле, мы можем запускать компиляцию одной командой:

```
$ make list
```

Или

```
$ make -f /path/to/make/file list
```

Если наш make-файл называется отлично от «Makefile»

Что означает осуществить сборку цели `list`. Если бы цель «`list`» называлась «`all`» то было бы достаточно просто:

```
$ make
```

Функционал утилиты `make` огромен, мы рассмотрели только вершину айсберга. Поскольку make изначально был создан для управления программными проектами, он уже знает, как осуществлять многие операции. В частности для нашего проекта было бы достаточно описать:

```
my_list.o:
```

```
list: my_list.o
```

И сборка прошла бы успешно. Это происходит благодаря наличию автоматических правил.

Начать подробное знакомство с make можно тут. Полная информация есть на официальном сайте.

Использование библиотек

Программный код, который мы часто используем повторно, но редко вносим изменения, удобно скопилировать один раз в некий большой файл и обращаться по необходимости. Это особенно удобно, когда время, требуемое на компиляцию библиотеки — существенно. Здесь существует два подхода для организации такого рода «хранилищ».

При первом каждый раз, когда нам при компиляции понадобятся машинные коды программ, мы просто скопируем их из библиотеки в свою программу (исполняемый файл) и в процессе работы вспоминать про библиотеку уже не будем. В результате наш исполянемый файл будет большого размера, но без внешних зависимостей. Библиотеки, созданные для такого рода взаимодействий, называются *статическими*. В Linux-системах имеют расширение «.a».

Второй подход заключается в создании при компиляции ссылок на бибилотеку, откуда можно будет брать готовый машинный код уже процессе выполнения по необходимости. Исполняемый файл программы, зависящий от бибилиотеки, значительно меньше по объему, но если в процессе работы бибилиотека не будет найдена, произойдет ошибка, и программа прекратит работу. Бибилиотеки, обеспечивающие такое взаимодействие, называются *динамическими*. В Linux-системах имеют расширение «.so».

Рассмотрим, как создать статическую и динамическую библиотеки для работы со списками.

Создание статической бибилиотеки

Чтобы создать статическую библиотеку, нам необхобходимо создать объектный файл.

```
$ gcc -c my_list.c -o my_list.o
```

Далее все созданные файлы объектные файлы (один в нашем случае) объединяются в один ar архив.

```
$ ar rcs libmy_list.a my_list.o
```

После этого необходимо обеспечить сквозную индексацию для архива. Если объектных файлов несколько, индексы функций, машинный код которых они содержат, надо объединить в один для быстрого доступа.

```
$ ranlib libmy_list.a
```

(Чаще всего утилита `ar` делает это сразу).

После чего main.c можно компилировать как:

```
$gcc main.c -o list -L. -lmy_list
```

`-L.` (В общем случае `-L/path/to/libraries/files`) — указание линковщику директории с файлами библиотек . (« . » - текущая директория)

`-lmy_list` (в общем случае `-llibrary_name`) — использовать библиотеку `libmy_list.a` (приставку `lib` линковщик «додумает» автоматически, в имени библиотеки должна быть обязательно).

Запускать программу можно по-прежнему

```
$./list
```

Создание динамической библиотеки

Для создания динамической библиотки по-прежнему надо создать объектный файл с небольшим отличием — код должен быть позиционно-независимым (https://en.wikipedia.org/wiki/Position-independent_code). Для этого используем флаг `-fPIC`:

```
$ gcc -fPIC -c my_list.c -o my_list.o
```

После чего объектные файлы объединяем в разделяемую (динамическую) библиотеку:

```
$ gcc -shared -o libmy_list.so my_list.o
```

Компилировать main.c после этого можно так же, как и в случае статической библиотеки:

```
$ gcc main.c -o list -L. -lmy_list
```

Однако запустить так же просто, как и в предыдущий раз, мы не сможем:

```
$ gcc ./list
```

```
libmy_test.so: cannot open shared object file: No such file or directory
```

Это происходит, потому что динамический линковщик (тот, кто отвечает за связывание между ссылками в исполняемом файле и реальным кодом библиотеки) не знает, что разделяемая библиотека находится в той же самой директории. Поэтому можно либо добавить свою библиотеку к общим системным `/usr/lib` или `/usr/lib64`. Или задать переменную окружению `LD_LIBRARY_PATH`, определяющую директорию с разделяемыми библиотеками.

```
$ env LD_LIBRARY_PATH=/path/to/libmy_test.so/directory ./list
```

Полезные ссылки:

Про создание библиотек

Организация программных проектов (здесь немного подробнее про make и про заголовочные файлы).