Lenguaje C, Headers, Pointes and Pointes

Álvaro Rojas V.

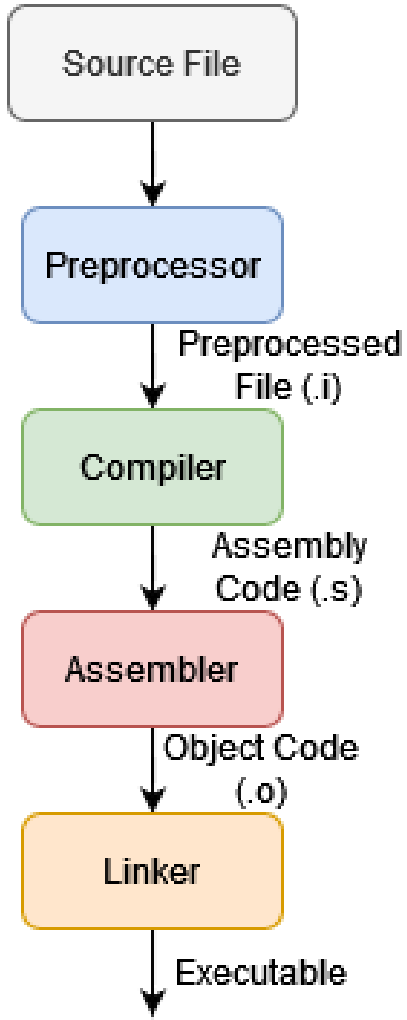
UTFSM

Agosto 2024-2

Table of Contents

C es un lenguaje Compilado

C es un lenguaje Compilado, por lo que se requiere un programa que lea uno o múltiples códigos fuentes para generar el ejecutable. Este proceso es complejo en detalle, pero se puede simplificar en 4 etapas:



Preprocesado

En esta etapa, el compilador se encarga de: remover los **comentarios**, expandir los **macros**, copiar los códigos incluidos (`#include`) y tratar la **compilación condicional**.

```c
#ifndef H_TDA
#define H_TDA
struct TDA {
    int value;
};
extern int globalVariable;
void foo(int x, int y);
#endif
```

(a) TDA.h

```c
#include "TDA.h"
#define PI 3.14159265359

int main() {
    int r = 3;
    int area = PI * r * r;
    return 0;
}
```

(b) main.c

Preprocesador

El resultado del pre-procesado es un archivo `.i`, este sigue siendo código legible:

```
struct TDA {
    int value;
};
extern int globalVariable;
void foo(int x, int y);

int main() {
    int r = 3;
    int area = 3.14159265359  * r * r;
    return 0;
}
```

Figure: main.i

Compiler y Assembler

Estas dos etapas se encarga de transformar el código anterior a código Assembly y posteriormente a código máquina. Es en este momento en que ocurre el proceso de análisis léxico, sintáctico y semántico. Como también se genera optimizaciones para decrecer el tiempo de ejecución de los programas.
Finalmente, se genera un archivo objeto con extensión `.o`.

Linker

Esta etapa permite enlazar otros objetos para formar el ejecutable final. Es aquí donde se enlaza los llamados a funciones definidas en diferentes módulos ya previamente compilados. Esto permite crear un sistema de librerías, permitiendo que el código fuente sea modular.
Esto conlleva múltiples ventajas, como por ejemplo la **reusabilidad de código en múltiples programas. Mejora la** portabilidad de un proyecto para tener diferentes objetos generados por diferentes compiladores escritos en diferentes lenguajes. Y también permitiendo la facilidad de **mejorar los módulos por separados**, sin la necesidad de recompilar todo el proyecto (compilación separada).

Compilación Separada

Por lo que la compilación separada consiste en modularizar el proyecto en diferentes archivos `.c`. Estos no se deben importar usando `#include`, si no más bien se utilizan los **headers** para traer las declaraciones de los componentes de un módulo y con ello el **Linker** poder identificarlos y enlazarlos correctamente.

Comandos gcc

`gcc -Wall -c` *`sourceCode`*`.c -o` *`objectName`*`.o`, Creación de archivo objeto compilado.

`gcc -Wall -o` *`exeName`*`.exe` *`objectNames`*`.o`, Llamado al linker y creación del ejecutable.

Headers

Los headers files (`.h`) contiene un conjunto de declaraciones de prototipos de funciones, estructuras, data types, macros, etc. Permite modularizar código para ser usado en diferentes códigos fuentes a la vez.
Se pueden definir nuevos header para un proyecto en C. Para impedir la inclusión del mismo header, se puede usar directivas de pre procesado como protección al header.
Para incluir los header se usan la directiva `#include "header.h"`.

```c
#ifndef HEADER_H
#define HEADER_H
void foo(int a, int b);
void goo(int a, int b);
#endif
```

Pointers & Pointers

Los punteros son un tipo de dato en que almacena una **dirección de memoria**. Esa dirección puede contener algún tipo de dato, desde un número, una estructura, arreglos e incluso a otros punteros que apuntan a otra dirección.

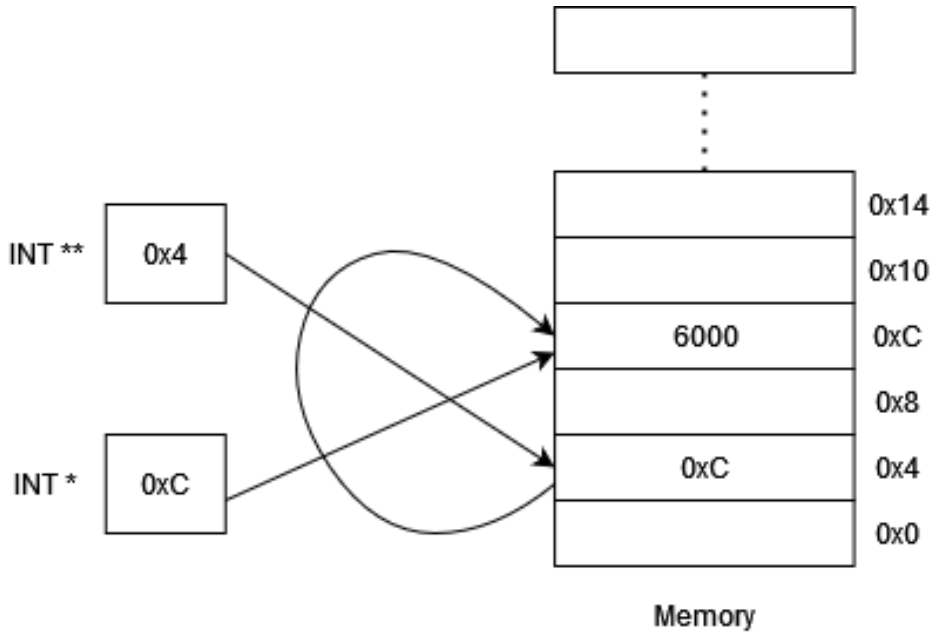


Figure: Diagrama Punteros

Pointers in C

En el lenguaje C tiene fama por el uso de los punteros. Cada vez que se use la función `malloc()`, este retorna una dirección de memoria a alguna parte del **heap**. Como programadores, tenemos la responsabilidad de guardar la dirección, avisar al compilador qué tipo de dato apunta y desalocar esa memoria una vez usada.

Pointers to Pointers

La habilidad de pedir memoria al **heap** nos permite crear estructuras de datos de tamaño dinámico. Como se ha visto en clases, un arreglo es un sector de memoria donde hay un tipo de dato de manera continúa. Por lo que se puede usar un puntero para que apunte al primer elemento de ese arreglo.

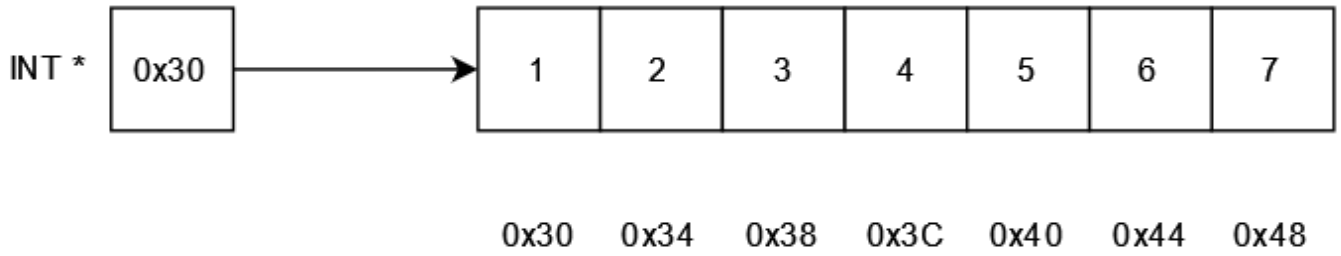


Figure: Diagrama Punteros a Array

Se puede construir un puntero que apunte a un array de punteros, y que cada puntero apunten otro array de punteros, y que cada puntero apunte a una dirección de memoria de algún tipo de dato cualquiera. Es decir, un `void ***`.

Void *

Los **void *** es un puntero que apunta a una dirección de memoria sin saber qué tipo de dato almacena esa dirección. Como programadores podemos decirle a un **void *** que actué como un puntero a un tipo de dato ya conocido. Es decir, aplicar un **Type cast** al puntero.

```
typedef struct {
    int a;
} data;
void ** ptr = (void **) malloc(5*sizeof(void *)); // Alocar
    array 5 void *
for(int i=0; i < 5; i++) {
    ptr[i] = (void *) malloc(sizeof(data)); // Alocar struct
    ((data *) ptr[i])->a = i;  // Acceder al struct data
}
```

Figure: Type cast a un Puntero Void

Pointers To Functions

Los punteros a funciones son punteros que apuntan a la **dirección de memoria de una función**. Las funciones se encuentran almacenado en la parte más baja de la memoria. Un puntero que almacena una función puede invocar a esa función entregando los parámetros y retornado el resultado de este.

```
float (*x)(int, int) // X es un puntero a una funcion que
    recive dos enteros y retorna un float
x = &foo; // Asigna una funcion a X
float res = x(10, 4) // Se llama a la funcion almacenada
```

Figure: Ejemplo Puntero a Función

Bibliografía

- https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/
- https://medium.com/@prem112/header-files-in-c-e306e685c148
- https://www.geeksforgeeks.org/function-pointer-in-c/
- https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html