Partes de un programa

- Los programas se subdividen en “partes” genéricamente llamadas rutinas, la que suelen clasificarse como:
  - Procedimientos: no devuelven valores
  - Funciones: devuelven un resultado
- El mecanismo para pasar datos entre el código que llama y el invocado puede clasificarse, básicamente, así:
  - Por valor
  - Por referencia
- En lenguaje C “todo” es una función, incluso “el programa en si mismo” es una función con un nombre especial: main
- En el único mecanismo para pasar parámetros a una función es por valor. En realidad se puede usar el mecanismo “por referencia” pero el programador debe hacerlo explícitamente.

Declaración de funciones

- Las funciones se declaran y se implementan
  - La declaración da la “firma” de la función, o sea, su nombre, que parámetros lleva y que valor devuelve
  - La implementación es el código completo de la función.
- Para declarar una función se sigue el siguiente patrón:

```
tipo-que-devuelve nombre-función (tipo-parámetro-1 nombre-parámetro-1, … ,  tipo-parámetro-n nombre-parámetro-n);
```

  Ejemplo:

```
double division(double dividendo, double divisor);
```

- Nota: en la declaración NO es necesario incluir el nombre del parámetro, pero se puede y de hecho lo vamos a incluir porque aumenta la legibilidad del código

Ejemplo declaración e implementación

Archivo main.c :

```c
#include <stdio.h>

double division(double dividendo, double divisor){
   return dividendo/divisor;
}

int main()
{
   double a = 21.32;
   double b = 4.1;
   double z;

   z = division(a, b);
   printf("%4.2f / %4.2f = %4.2f\n", a, b, z);
   return 0;
}
```

Salida:

```
21.32 / 4.10 = 5.20
```

Si cambiamos el orden

Archivo main.c :

```c
#include <stdio.h>

int main()
{
   double a = 21.32;
   double b = 4.1;
   double z;

   z = division(a, b);
   printf("%4.2f / %4.2f = %4.2f\n", a, b, z);
   return 0;
}

double division(double dividendo, double divisor){
   return dividendo/divisor;
}
```

Salida:

```
warning: implicit declaration of function ‘division’
error: conflicting types for ‘division’
note: previous implicit declaration of ‘division’ was here
```

Explicación

- El problema en el ejemplo anterior es que invocamos una función (división) que no fue declarada previamente.
- Todo identificador es reconocido desde el punto de su declaración (o definición) hasta el final del bloque donde fue declarado
- Como en lenguaje C no se permiten declarar funciones anidadas (declarar una función adentro de otra) todas las funciones están declaradas a nivel de archivo
- Las variables se pueden declarar a nivel de archivo, es decir afuera de toda función, se las conoce como variables estáticas (“globales”). O dentro de una función, conocidas como variables automáticas (locales)
- Incluso, una variable puede declararse dentro de un bloque interno a una función.

Solución

Archivo main.c :

```c
#include <stdio.h>

double division(double dividendo, double divisor); //declaración

int main()
{
   double a = 21.32;
   double b = 4.1;
   double z;

   z = division(a, b);
   printf("%4.2f / %4.2f = %4.2f\n", a, b, z);
   return 0;
}

double division(double dividendo, double divisor){ //defínción
   return dividendo/divisor;
}
```

Salida:

```
21.32 / 4.10 = 5.20
```

Múltiples fuentes

- Dado que es común usar múltiples fuentes, como hacemos entonces para usar funciones que están implementadas en otro fuente ?
- Igual que como hacemos para usar, por ejemplo, printf, usando la directiva del preprocesador #include
  - include referencia un archivo encabezado, que se estila guardar con la extensión .h
  - El resultado es como si la linea que contiene la directiva #include fuera reemplazada por el contenido del archivo que menciona
- Lo habitual es que cada fuente **archivo.c** tenga su correspondiente encabezado **archivo.h** que contiene las declaraciones de las funciones implementadas en su fuente, de modo que se lo pueda incluir en otros fuentes
  - Archivos fuente (.c): definición de funciones y variables
  - Archivos de encabezado (.h): Declaraciones de funciones y variables

Ejemplo múltiples fuentes

Archivo main.c :

```c
#include <stdio.h>
#include “mate.h”

int main()
{
    double a = 21.32;
    double b = 4.1;
    double z;

    z = division(a, b);
    printf("%4.2f / %4.2f =”
         “%4.2f\n", a, b, z);
    return 0;
}
```

Archivo mate.h :

```c
#ifndef MATE_H_INCLUDED
#define MATE_H_INCLUDED

double division(double dividendo,
                double divisor);

#endif // MATE_H_INCLUDED
```

Archivo mate.c :

```c
double division(double dividendo,
                double divisor)
{
    return dividendo/divisor;
}
```

Ejemplo Recursión

- En lenguaje C las funciones pueden invocarse recursivamente.
- Cada instancia de la función tiene su “propio juego” de variables locales

```
long factorial(long numero)
{
        if (numero < 2) // factorial de 0 y de 1 vale 1
                return 1;
        else // llamada recursiva
               return numero * factorial(numero - 1);
}
```

Ámbito y alcance

- El ámbito de un identificador es el contexto en cuál está definido. Esto define su alcance, es decir, en que partes del programa el identificador puede ser usado
- Las variables declaradas dentro de un bloque son por defecto automáticas. Son creadas cuando el programa ingresa en el bloque que las define y se destruyen al salir del mismo. Al crear la variable su valor es indeterminado, el programador debe asignar un valor explícitamente
- Las variables declaradas a nivel de archivo son estáticas y existen durante toda la ejecución del programa. Si el programador no las inicializa explícitamente son puestas “a cero” antes de comenzar el mismo

Ámbito y alcance

- Las variables declaradas dentro de un bloque con clase de almacenamiento **static** pasan a ser estáticas, como las declaradas a nivel de archivo, y por tanto vale para ellas el párrafo anterior.
  - Existen durante toda la ejecución del programa
  - Inicializadas en cero por defecto
- Para poder usar una variable declarada a nivel de archivo en un fuente distinto a donde se la definió, se usa la clase de almacenamiento **extern** para indicar que no se define una nueva variable, sino que se quiere usar una ya definida en otro fuente.
- Si una variable a nivel de archivo se la declara con clase de almacenamiento **static**, entonces no es posible usarla desde otro fuente, ni aún con **extern**

Ejemplos

Archivo main.c :

```c
#include <stdio.h>
#include “mate.h”

int main()
{
    double a = 21.32;
    double b = 4.1;
    double z;

    printf("glob = %f\n", glob_x_defecto);
    z = division(a, b);
    printf("%4.2f / %4.2f = %4.2f\n",
            a, b, z);

    b = 0.0;
    z = division(a, b);
    printf("%4.2f / %4.2f = %4.2f\n",
            a, b, z);
    return 0;
}
```

Archivo mate.h :

```c
#ifndef MATE_H_INCLUDED
#define MATE_H_INCLUDED

double division(double dividendo,
                double divisor);
extern double glob_x_defecto;

#endif // MATE_H_INCLUDED
```

Archivo mate.c :

```c
double glob_x_defecto = 1;

double division(double dividendo,
                double divisor)
{
   if (divisor == 0.0)
   divisor = glob_x_defecto;
   return dividendo/divisor;
}
```

Salida:

```
glob = 1.000000
21.32 / 4.10 = 5.20
21.32 / 0.00 = 21.32
```

```c
#include <stdio.h>

int funcion(int x);

int global = 5;

int main()
{
        int local_x = 2;
        int local = 1;

        printf("En main: local = %d\tlocal_x = %d\n", local, local_x);
        local_x = funcion(local_x);
        printf("En main: local = %d\tlocal_x = %d\n", local, local_x);

        while(--local_x) {
                int local = 2;
                printf("En while: local = %d\tlocal_x = %d\n", local, local_x);
        }
        printf("En main: local = %d\tlocal_x = %d\n", local, local_x);
        return 0;
}

int funcion(int x)
{
        int local = 2;
        printf("En funcion: local = %d\tx = %d\n", local, x);
        return local * x;
}
```

Salida

```
En main: local = 1 local_x = 2
En funcion: local = 2       x = 2
En main: local = 1 local_x = 4
En while: local = 2         local_x = 3
En while: local = 2         local_x = 2
En while: local = 2         local_x = 1
En main: local = 1 local_x = 0
```

Casting

- Se llama casting a la conversión del valor de una variable o expresión a un tipo de dato distinto del que tiene.
- El compilador hace algunos casting implícitos, por ejemplo si a una variable de tipo double le asigno 5 el compilador convierte primero ese 5 en 5.0 (es de decir de int a double)
- Si en una expresión se mezclan tipos de datos el compilador los convierte (promoción entera) al tipo de datos con mayor capacidad de representación
  - En términos generales: char → int → long → float → double
- Se puede hacer un casting explícito, indicado por el programador anteponiendo entre paréntesis el tipo de datos al que queremos convertir
  - Ejemplo: (double) 'c' convierte a 99.0

Alias de tipos de datos

- Los alias de tipos de datos permiten aislar cambios que dependan del “objetivo de compilación”, es decir de la combinación arquitectura del computador más el sistema operativo
- Por ejemplo cierta cantidad entera en un objetivo puede ser suficiente un int y en otro no alcanzar y necesitar un long (p.ej: dirección de memoria)
- El modo de definir un tipo de datos es con la palabra reservada **typedef**. El modo de usarlos es:
  - typedef tipo-de-dato nuevo-alias;
- En nuestro ejemplo podríamos definir en un header para el primer objetivo:
  - typedef int numero;
- Y en el segundo objetivo
  - typedef long numero;
- El resultado es que luego podemos declarar variables de tipo numero y saber que nuestro fuente será correcto en ambos objetivos
  - numero a, b;

Definir constantes

- Para definir constantes lo tradicional es usar la directiva #define del preprocesador
  ```
  #define PI 3.14159265358
  ```
  Cada vez que en el fuente figure PI el preprocesador reemplazará por 3.14159265358
- Si en el define ponemos un identificador, pero no un valor, simplemente queda como “definido” el identificador, sirve para compilación condicional.
- Si hay que definir varias constantes relacionadas entre sí, lo que se usa es una enumeración

Enumeraciones

- permite “definir constantes” y a la vez “definir un tipo”. Las constantes son enteras y comienzan con cero y se incrementan de a uno.

```
enum dia {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO,
DOMINGO};
```

- permite luego declarar una variable “de tipo dia”

```
enum dia hoy, ayer;
hoy = LUNES; // asigna 0 a hoy
ayer = JUEVES; // asigna 3 a ayer
```

- Si quiero asignar distintos valores (en lugar de 0,1,2 …) puedo hacerlo asignando, como en el siguiente ejemplo

```
enum operaciones {LEER = 1, ESCRIBIR, VERIFICAR, BORRAR = 8,
ALERTAR};
```

- En este caso LEER valdrá 1 en lugar de cero como valdría si no hubiese usado el “= 1”, en tanto que ESCRIBIR valdrá 2 porque al no aclarar nada en contrario, vale uno más que el anterior. Así siguiendo VERIFICAR valdrá 3, BORRAR 8 y ALERTAR 9.

Compilación condicional

- Se hace que dependiendo del valor de un identificador del preprocesador, el compilador reciba un texto u otro. Ejemplo:

```
#ifdef DEBUG_FULL
   printf("debug full %s\n", infofull);
#elif DEBUG_LEVEL > 2
   printf("debug nivel alto %s\n", infoalta);
#else
   printf("debug nivel bajo %s\n", infobaja);
#endif
```

Macros

- El preprocesador también sirve para hacer macros, “tipo” funciones simples (pero en realidad reemplaza textualmente)

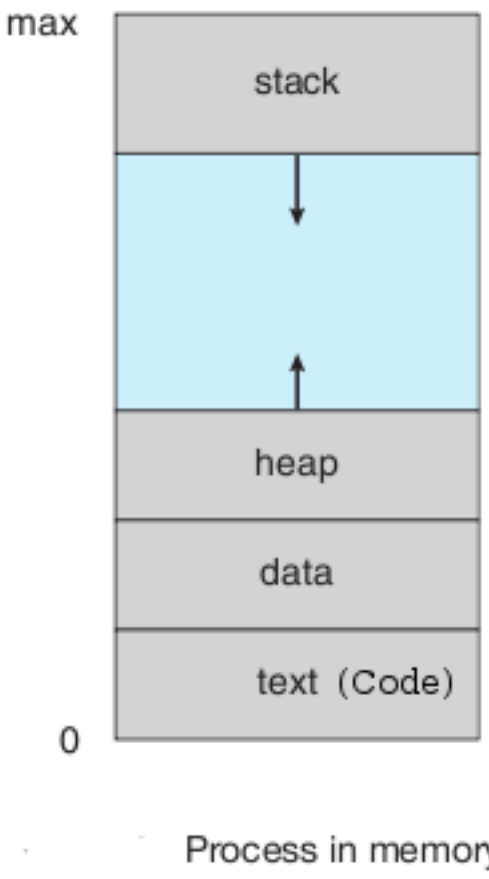
```
#define FOREVER for (;;)
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

- Notar los paréntesis en MAX, son por si es una expresión, por ejemplo

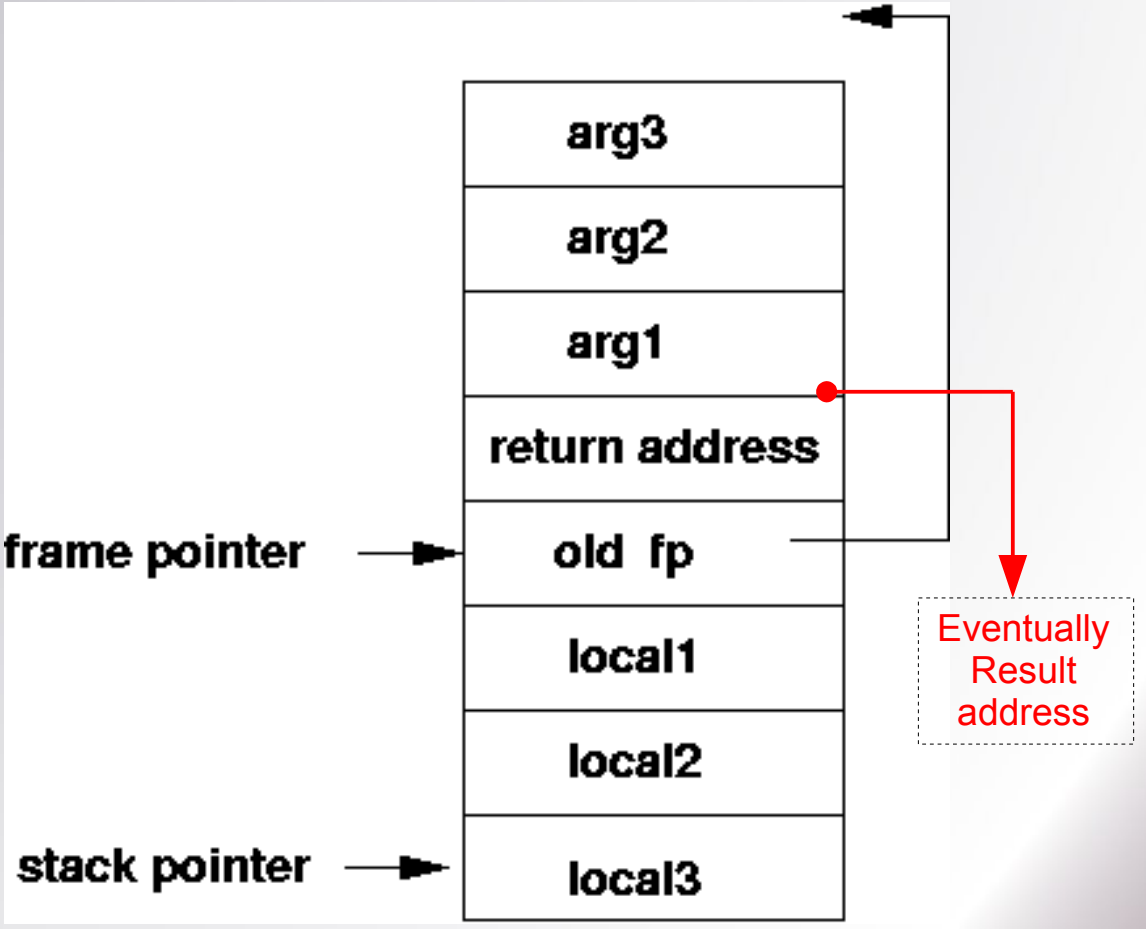
```
MAX(a*2, b+5)
```

- Alternativa: el uso de inline (tiene sus detractores)

Manejo de la memoria



Process in memory.



Licencia