Tipos de datos

Tipo	Espacio ocupado	Rango
char	1 byte	-127 a 127
unsigned char	1 byte	0 a 255
short	2 bytes	-32768 a 32767
unsigned short	2 bytes	0 a 65535
int	4 bytes	2147483648 a 2147483647
unsigned int	4 bytes	0 a 4294967295

Comentar

Se comenta con `/* */` o con `//`

Variables

La sintaxis es:

```
<tipo de dato> nombre_var = <tipo de dato que queremos que tome>;
```

o

```
<tipo de dato> nombre_var; //toma valor predeterminado del tipo de dato
```

typedef

Sirve para definir un nuevo tipo de dato con un nombre a elección a partir de un tipo de dato ya existente. Usualmente se lo utiliza para construir nombres más cortos o más significativos para tipos ya definidos por C o para tipos que haya declarado.

La sintaxis es:

```
typedef <tipo> <nuevo nombre>;
```

Ejemplo:

```
int main(){

   typedef int edad_t;

   edad_t edad_juan = 23;

   printf("%d\n", edad_juan);

   return 0;
}
>> 23
```

Constantes

Hay 2 formas de definirlas:

- `const float PI = 3.1415926`
- `#define PI 3.1415926 //en forma de etiqueta, es más lindo`

Operadores

Operador	Significado
`=`	Asignación
`+`	Suma
`-`	Resta
`sqrt()`	Raíz cuadrada
`*`	Multiplicación
`/`	División entera
`%`	Módulo

- `x (+, - , * , /)= y` : Es lo mismo que hacer `x = x (+, - , * , /) y`
- `x % y` : Es el resto de la división entre `x` e `y`.
- `x / y` : Si los dos valores son int, es una división entera. Si alguno de los dos es un float, hace la división normal. Si no queremos hacer la división entera y tenemos dos int, los casteamos a `float32` y hacemos la división. Ejemplo: `float32(6) / float(10)`.

Control de flujo

Operadores

Operador	Significado
`==`	Igual
`!=`	Distinto
`&&`	AND
`!`	NOT
||	OR
`>`	Mayor
`<`	Menor
`>=`	Mayor o igual
`<=`	Menor o igual

If

La sintaxis es:

```
if(condición)
{
  //código
}
```

if - else

La sintaxis es:

```
if(condición)
{
  //código
}
else if (condición)
{
  //código
}
else
{
  //código
}
```

switch - case

Es una estructura de control que me permite elegir entre varias posibilidades según ciertas condiciones.

Este va a evaluar el valor de la variable en cada case y si la condición se cumple, entra el case y ejecuta ese bloque y TODOS los que le siguen. En la condición tenemos que poner valores que tengan sentido con respecto al valor de dicha variable, sino no compila.
Ningún case tiene el break implícito. Si se quiere modificar ese comportamiento se debe utilizar `break` (este cortará el switch por completo).

La sintaxis es:

```
switch(variable)
{
  case valor:
    //código
  case valor:
    //código
  default:
    //código
}
```

Ejemplo:

```
int main(){

   int dia = 2;

   switch (dia) {

   case (0):

   case (1):

       printf("lunes\n");

   case (2):

       printf("martes\n");

   case (3):

       printf("miercoles\n");

       break;

   default:

       printf("jueves\n");

   }

   return 0;
```

```
}

>>

martes

miercoles
```

Operador Ternario

Sirve para cuando tenemos un solo `if` y un `else`. Lo que hace es si la `condición` es `true` devuelve la `expresión1`, sino la `expresión2`.

Su sintaxis es:

```
<condición> ? <expresión1> : <expresión2>;
```

Ejemplo:

```
int main(){

   int i = 2;
   int j = 5;

   int k = i < j ? 1 : 0;
   int m = i > j ? 1 : 0;

   printf("%d\n", k);
   printf("%d\n", m);

   return 0;

}
>> 1
>> 0
```

Ciclos

Instrucciones de corte

- break
- continue
- return

For

La sintaxis es:

```
for(inicialización de variable; condición; incremento de variable)
{
  //código
}
```

Cualquiera de los segmentos puede dejarse en blanco.

La `inicialización de variable` se ejecuta una sola vez, luego evalúa la `condición`. Si esta se cumple se ejecuta el bloque y luego el `incremento de variable`. Esto se repite hasta que la condición sea Falsa y ahí sale del ciclo.

Puede introducirse más de una `inicialización de variable` o más de un `incremento de variable` separando las sentencias por coma `,`.

Ejemplo:

```
for (int i = 0, j=1; i<10 && j<5 ; i++, j++) {
 ...
}
```

While

La sintaxis es:

```
while(condición)
{
  //código
}
```

Do - While

La sintaxis es:

```
do
{
  //código
}while(condición);
```

Funciones

Declaración de funciones

La sintaxis es:

```
<tipo de dato> nombreFuncion (<tipo de dato> arg1 , ...);
```

Las funciones deben ser declaradas antes de ser utilizadas. Los nombres de los argumentos son opcionales.

Definición de funciones

La sintaxis es:

```
<tipo de dato que se devuelve> nombreFuncion (<tipo de dato> arg1 , ...){
     //código
}
```

Ejemplo:

```
int cuad(int n) {
 return n * n;
}
```

Las funciones no necesitan ser definidas antes de su uso, solo necesitan ser declaradas. Sin embargo, eventualmente la función debe ser definida para que pueda ser enlazada.

Si una función ha sido previamente declarada, debe ser definida con el mismo valor de retorno y tipos de argumentos (o se creará una nueva función sobrecargada), pero los nombres de los parámetros no necesitan ser los mismos.

Si no queremos que devuelva nada utilizamos como tipo de dato void:

```
void func() {

   printf("%d\n", 5);  // no devuelve nada

}
>> 5
```

Punteros

La sintaxis es:

```
int x = 4;
int p = &x; // obtiene la dirección de memoria de x
*p; // resulta en el valor 4
```

- El asterisco desreferencia el puntero.
- El valor de la expresión `*p` es el valor de la variable cuya dirección de memoria es apuntada por el identificador.

Estructuras

Definición de estructuras

La sintaxis es:

```
typedef struct s_nombreStruct { // s_ no es necesario pero se suele usar
    <tipo de dato> atributo_1;
    // ...
    <tipo de dato> atributo_n;
} nombreStruct_t;
```

Declaración de punteros a estructuras

La sintaxis es:

```
struct_nombre_t* nombreInstancia = (struct_nombre_t*)malloc(sizeof(struct_nombre
```

Acceso a atributos de estructura (cuando no es puntero)

La sintaxis es:

```
nombreInstancia.structAtributo;
```

Acceso a atributos a puntero de estructura

La sintaxis es:

```
punteroInstancia -> structAtributo;
```

Arreglos

Declaración de arreglos

La sintaxis es:

```
<tipo de dato> nombre_arr[tamaño_arr / vacio] = {elementos}
```

Ejemplo:

```
int valores[4] = { [1]=20, [3]=40 }; // define [0, 20, 0, 40]

int valores[] = {0, 20, 0 , 40}; // define [0, 20, 0, 40]

int valores[4] = {0, 20, 0 , 40}; // define [0, 20, 0, 40]

int valores[4] = {}; // define [0, 0, 0, 0]
```

Acceso a elementos del arreglo

La sintaxis es:

```
nombre_arr[index];
```

Strings

La sintaxis es:

```
char <nombre_str> [cantidad_elementos/ nada] = {string}
```

La cantidad de caracteres en un string es su longitud+1 ya que, como por dentro es un array de chars, este le indica que es un string con el carácter `\0` a lo último. Por la misma razón, (cada char ocupa un byte) no hay problema con el endianness.

Ejemplo:

```
int main(){

   char cadena[] = "hola"; // = {“h”, “o”, “l”, “a”, “\0”}

   printf("%s\n", cadena[0]);

   return 0;

}

>> h
```

Para comparar cadenas:

```
char s1[] = "hola";

char s2[] = "hola";

if (strcmp(s1, s2) == 0) {...}

>> true
```

Malloc

En C/C++, `malloc` es una función utilizada para asignar memoria dinámica en el heap durante el tiempo de ejecución de un programa. Permite reservar un bloque de memoria de tamaño específico y devuelve un puntero a la dirección base de ese bloque.¿

Existen varias razones por las cuales se utiliza `malloc` en C/C++:

- **Ciclo de vida de los datos:** La memoria asignada con `malloc` persiste hasta que se libera explícitamente mediante la función `free`. Esto permite que los datos persistan más allá del ámbito de una función, lo que es útil cuando se necesita compartir datos entre diferentes partes del programa
- **Tamaño de memoria dinámica:** A diferencia de la memoria estática (declarada en tiempo de compilación), la memoria asignada con `malloc` es dinámica y puede ajustarse durante la ejecución del programa.

Es importante tener en cuenta que con la ventaja de la flexibilidad que proporciona `malloc`, también surgen responsabilidades adicionales. Cuando se utiliza `malloc`, el programador es responsable de liberar la memoria asignada cuando ya no es necesaria, utilizando la función `free`. Si no se libera la memoria adecuadamente, puede provocar pérdidas de memoria (memory leaks) y agotamiento de los recursos del sistema. Además, el uso incorrecto de `malloc` puede llevar a problemas de seguridad, como desbordamientos de búfer, si no se gestiona correctamente el tamaño y la manipulación de los datos almacenados en la memoria asignada.

Casos:

```
int* func(int num){

  int* array[num];

  //...

  return array;
}
```

En el ejemplo hay 2 problemas:

- Al no tener `array` un tamaño fijo, luego `num` puede tomar un valor muy grande que el stack no soporte.
- Cómo `array` se inicializa en la función `func`, este se guarda en el stack relativo de esta función. Luego, después de hacer toda la ejecución, antes de retornar `array`, el stack elimina todo lo que se definió durante la ejecución de esta función, por lo tanto, `array` se va a eliminar. Esto quiere decir que vamos a devolver un puntero que no existe.

En síntesis, lo que sucede es que el puntero apunta a la dirección de memoria donde está ubicado el dato. Este a su vez, como se instanció en la función, se va a almacenar en el stack. No obstante, al finalizar la función, el stack libera todas las variables definidas dentro de ella. En consecuencia, si intentamos devolver ese puntero, se generará un error, ya que se trata de una dirección de memoria que ha sido liberada y, por ende, no se puede acceder a ella.

Por lo tanto hay que declararla en el heap con Malloc:

```
int* func(int num){

  int* array = malloc(sizeof(int) * num);

  //...

  return array;

}
```

Si la queremos liberar en la ejecución de la misma función:

```
void func(int num){

  int* array = malloc(sizeof(int) * num);

  //...

  free(array);

  return;

}
```