什么是C语言?

C语言是一种通用的、`面向过程`的计算机编程语言。它由贝尔实验室的Dennis Ritchie于1972年左右开发，并成为了广泛应用于系统编程和应用程序开发的一种语言。C语言具有简洁的语法、高效的执行速度和强大的底层控制能力，因此在操作系统、嵌入式系统、游戏开发等领域得到广泛应用。

二十世纪八十年代，为了避免各开发厂商用的C语言语法产生差异，由美国国家标准局为C语言制定了一套完整的美国国家标准语法，称为`ANSI C`，作为C语言最初的标准。

C语言是一门`面向过程`的计算机编程语言，与C++，Java等面向对象的编程语言有所不同。 其编译器主要有Clang、GCC、WIN-TC、SUBLIME、MSVC(VS)、Turbo C等。

第一个C语言程序

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");

    return 0;
}
```

解释:

- `#include <stdio.h>`这行代码是一个预处理指令，它告诉编译器包含标准输入输出函数的头文件`stdio.h`。头文件中包含了对输入输出操作的函数原型和常量定义。
- 程序的主函数`main`，它是C语言程序的入口点。`int`是函数的返回类型，`main`是函数的名称，后面的空括号表示该函数不接受任何参数。**main函数必须要有，并且只能有一个。**
- C语言标准库中的`printf`函数，它用于将指定的格式化字符串输出到标准输出（通常是控制台）。在这里，我们将字符串"Hello World"作为参数传递给`printf`函数，它将打印该字符串到控制台。
- `return 0`，用于表示函数的结束，并将整数值0作为函数的返回值返回给调用者。在C语言中，返回值为0通常表示程序执行成功。
- `\n` 是一个特殊的转义字符，称为换行符。它用于在输出中表示换行操作，即将输出光标移动到下一行的开头位置。

printf函数

`printf` 是C语言标准库中的一个函数，用于格式化输出文本到标准输出（通常是控制台）。``

函数原型:

```c
int printf(const char *format, ...);
```

`printf` 函数的主要功能是根据格式化字符串中的占位符，将相应的参数格式化为指定的文本，并输出到标准输出。函数的**返回值**是打印输出的字符数。占位符以 `%` 开头，后面跟有一个或多个字符，用于指定参数的类型和输出的格式。

常见的格式化占位符及其用法如下:

- `%d`：输出带符号的十进制整数。

- `%u`：输出无符号的十进制整数。
- `%f`：输出浮点数。
- `%c`：输出单个字符。
- `%s`：输出字符串。
- `%p`：输出指针地址。
- `%x` 或 `%X`：输出十六进制整数。

除了占位符外，格式化字符串中的普通字符会按照原样输出。

在 `printf` 函数中，可以通过使用格式化选项来设置输出的精度、宽度和填充字符。下面是一些常用的格式化选项：

1.精度（Precision）：用于指定浮点数或字符串的输出精度。

- `%.nf`：设置浮点数的小数点后的精度为 n 位。
- `%.*f`：通过变量指定浮点数的小数点后的精度，例如 `%.*f`，后面再传递一个整数参数 n，指定精度的值。

```
float pi = 3.14159;
printf("%.2f", pi);  // 输出：3.14

int precision = 3;
printf("%.*f", precision, pi);  // 输出：3.142
```

2.宽度（Width）：用于指定输出的字段宽度，可以用空格或其他字符进行填充。

- `%nd`：将整数的输出宽度设置为 n 个字符，不足的部分用空格填充。
- `%*d`：通过变量指定整数的输出宽度，例如 `%*d`，后面再传递一个整数参数 n，指定宽度的值。

```
int num = 42;
printf("%6d", num);  // 输出：    42

int width = 8;
printf("%*d", width, num);  // 输出：       42
```

3.填充字符（Padding Character）：用于指定填充输出字段的字符，默认情况下是空格。

- `%nd`：在整数的输出宽度 n 前添加填充字符，例如 `%6d`，将输出的整数宽度设置为 6 个字符，默认用空格填充。

```
int num = 42;
printf("%06d", num);  // 输出：000042
```

scanf函数

`scanf` 是C语言标准库中的一个函数，用于从标准输入（通常是键盘）读取输入数据。

函数原型：

```
int scanf(const char *format, ...);
```

返回值：

`scanf` 函数返回成功读取的数据项数，如果发生错误或到达输入结束，则返回一个负数。

`scanf` 函数根据格式化字符串中的占位符，从标准输入读取相应的数据，并将其存储到指定的变量中。

常见的格式化占位符及其用法如下：

- `%d`：读取带符号的十进制整数。
- `%u`：读取无符号的十进制整数。
- `%f`：读取浮点数。
- `%c`：读取单个字符。
- `%s`：读取字符串。
- `%p`：读取指针地址。
- `%x` 或 `%X`：读取十六进制整数。

除了占位符外，格式化字符串中的普通字符会与输入数据进行匹配，需要输入相应的字符才能匹配成功。

> `scanf` 函数会根据格式化字符串和相应的参数进行输入，如果输入数据与格式化字符串中的占位符不匹配，可能会导致输入错误或出现运行时错误。因此，在使用 `scanf` 函数时需要确保格式化字符串和参数的正确匹配，避免潜在的问题。

```c
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);    //输入3
    printf("You entered: %d\n", num);  //打印： You entered 3

    return 0;
}
```

基本数据类型

```c
char      //字符数据类型
short     //短整型
int       //整形
long      //长整型
long long//更长的整形
float     //单精度浮点数
double    //双精度浮点数

//类型的使用:
char ch = 'w';
int weight = 120;
int salary = 20000;
```

> **数据类型的大小取决于编译器和目标平台的实现。编译器根据C语言标准规范和目标平台的特性来确定不同数据类型的大小。**

```
printf("%zu\n", sizeof(char));       //1
printf("%zu\n", sizeof(short));      //2
printf("%zu\n", sizeof(int));        //4
printf("%zu\n", sizeof(long));       //4
printf("%zu\n", sizeof(long long));  //8
printf("%zu\n", sizeof(float));      //4
printf("%zu\n", sizeof(double));     //8
printf("%zu\n", sizeof(long double));//8/16
```

变量、常量

生活中的有些值是不变的（比如：圆周率，性别，身份证号码，血型等等）

有些值是可变的（比如：年龄，体重，薪资）。

变量（Variables） 是用于存储和操作可变数据的标识符。在程序中，我们可以声明一个变量并为其分配内存空间，然后可以通过变量名引用和修改存储在内存中的值。变量的值可以在程序执行过程中发生变化。

常量（Constants） 是不可更改的固定值。在程序中，我们可以使用常量来代表不变的值，如数字、字符或字符串。常量的值在定义时就确定，并且不能再进行修改。

- 变量用于存储和操作可变数据，其值可以在程序执行过程中发生变化。
- 常量用于表示不可更改的固定值，其值在定义时确定，并且不能再进行修改。

定义变量的方法

```
int age = 150;
float weight = 45.5f;
char ch = 'w';
```

变量的命名

- 只能由字母（包括大写和小写）、数字和下划线（ _ ）组成。
- 不能以数字开头。
- 长度不能超过63个字符。
- 变量名中区分大小写的。
- 变量名不能使用关键字。

变量的分类

- 局部变量
- 全局变量

```c
#include <stdio.h>

int global = 2019;//全局变量
int main() {
    int local = 2018;//局部变量
    //下面定义的global会不会有问题?
    int global = 2020;                //局部变量
    printf("global = %d\n", global);//global = 2020

    return 0;
}
```

上面的局部变量global变量的定义其实没有什么问题的!

> **当局部变量和全局变量同名的时候，局部变量优先使用。**

变量的使用:

```c
#include <stdio.h>

int main() {
    int num1 = 0;
    int num2 = 0;
    int sum = 0;
    printf("输入两个操作数:>");
    scanf("%d %d", &num1, &num2);  //键盘输入： 3 5
    sum = num1 + num2;
    printf("sum = %d\n", sum);  //sum = 8

    return 0;
}
```

变量的作用域和生命周期

在C语言中，变量的作用域（scope）和生命周期（lifetime）是描述变量在程序中可见性和存在时间的概念。

作用域（Scope） 是指变量在程序中的可见范围。在不同的作用域内，变量的可访问性是不同的。C语言中有以下几种作用域：

- **块作用域（Block Scope）**：变量在一个代码块（由花括号 `{}` 包围）中声明，在该块内部可见。一旦离开块，变量就超出了其作用域。

```c
#include <stdio.h>

int main() {
    int x = 10;
    {
        int y = 20;
        printf("%d\n", x);  // 可以访问外部块中的变量
        printf("%d\n", y);  // 可以访问内部块中的变量
    }
    printf("%d\n", x);  // 可以继续访问外部块中的变量
    // printf("%d\n", y);  // 错误！超出了变量 y 的作用域
```

```
    return 0;
}
```

- **函数作用域（Function Scope）**：变量在函数内部声明，在整个函数中可见。函数参数和在函数内部声明的变量具有函数作用域。

```
#include <stdio.h>

int sum(int a, int b) {
    int result = a + b;
    return result;
}

int main() {
    int x = 10;
    int y = 20;
    int total = sum(x, y);
    printf("%d\n", total);
    // printf("%d\n", result);  // 错误！超出了变量 result 的作用域

    return 0;
}
```

- **文件作用域(全局作用域)（File Scope）**：变量在整个源文件中声明，可以被文件中的任何函数访问。在函数外部声明的变量具有文件作用域。

```
#include <stdio.h>

int globalVar = 10;

void function1() {
    printf("%d\n", globalVar);  // 可以访问文件作用域的变量
}

void function2() {
    printf("%d\n", globalVar);  // 可以访问文件作用域的变量
}

int main() {
    printf("%d\n", globalVar);  // 可以访问文件作用域的变量
    function1();
    function2();

    return 0;
}
```

生命周期（Lifetime） 是指变量存在的时间范围。变量的生命周期取决于其作用域和声明方式。C语言中有以下几种变量的生命周期：

- **自动变量（Automatic Variables）**：在函数内部声明的变量是自动变量。它们的生命周期从变量的定义处开始，到所在代码块结束时结束。每次进入代码块时，会重新创建变量，并在离开代码块时销毁。

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

```c
void function() {
    int x = 0;  // 自动变量，生命周期与函数调用关联
    x++;
    printf("%d\n", x);
}

int main() {
    function();  // 输出：1
    function();  // 输出：1
    function();  // 输出：1

    return 0;
}
```

- **静态变量（Static Variables）**：在函数内部使用 `static` 关键字声明的变量是静态变量。它们的生命周期从程序开始执行时创建，到程序结束时结束。静态变量在整个程序执行期间都存在，并且只被初始化一次。

```c
void function() {
    static int count = 0;  // 静态变量，生命周期与程序运行关联
    count++;
    printf("%d\n", count);
}

int main() {
    function();  // 调用函数，输出：1
    function();  // 调用函数，输出：2
    function();  // 调用函数，输出：3

    return 0;
}
```

- **全局变量（Global Variables）**：在函数外部声明的变量是全局变量。它们的生命周期从程序开始执行时创建，到程序结束时结束。全局变量在整个程序执行期间都存在，并且只被初始化一次。

```c
int globalVar;  // 全局变量

void function() {
    globalVar = 10;
}

int main() {
    function();  // 调用函数
    printf("%d\n", globalVar);  // 输出：10

    return 0;
}
```

常量

C语言中的常量和变量的定义的形式有所差异。

C语言中的常量分为以下以下几种：

- 字面常量
- `const` 修饰的常变量
- `#define` 定义的标识符常量
- 枚举常量

```c
#include <stdio.h>
// 枚举常量
enum Sex{
    MALE,
    FEMALE,
    SECRET
};
// 括号中的MALE,FEMALE,SECRET是枚举常量

int main(){
    // 字面常量演示
    3.14; // 字面常量
    1000; // 字面常量

    // const 修饰的常变量
    const float pai = 3.14f; // 这里的pai是const修饰的常变量
    // pai = 5.14;              // err 被const修饰后，是不能直接修改的!

    // #define的标识符常量 演示
#define MAX 100
    printf("max = %d\n", MAX); // max = 100

    // 枚举常量值
    printf("%d\n", MALE);   // 0
    printf("%d\n", FEMALE); // 1
    printf("%d\n", SECRET); // 2
    // 注：枚举常量的默认是从0开始，依次向下递增1的
    return 0;
}
```

> 上面例子上的 `pai` 被称为 `const` 修饰的常变量， `const` 修饰的常变量在C语言中只是在语法层面限制了变量 `pai` 不能直接被改变，但是 `pai` 本质上还是一个变量的，所以叫常变量。

字符串+转义字符+注释

字符串

```
"hello bit.\n"
```

这种由双引号（Double Quote）引起来的一串字符称为字符串字面值（String Literal），或者简称字符串。

> 注：字符串的结束标志是一个\0的转义字符。在计算字符串长度的时候\0是结束标志，不算作字符串内容。

```
#include <stdio.h>
//下面代码，打印结果是什么？为什么？（突出'\0'的重要性）
int main(){
    char arr1[] = "bit";
    char arr2[] = {'b', 'i', 't'};
    char arr3[] = {'b', 'i', 't',  '\0'};
    printf("%s\n", arr1);  //bit
    printf("%s\n", arr2);  //烫烫
    printf("%s\n", arr3);  //bit

    return 0;
}
```

1. `arr1` 是一个以字符串形式初始化的字符数组。它包含了字符 'b'、'i'、't' 和空字符 '\0'，并且在数组末尾自动添加了空字符。因此，`printf("%s\n", arr1)` 打印的是完整的字符串 "bit"。
2. `arr2` 是一个使用字符列表初始化的字符数组。它包含了字符 'b'、'i'、't'，但没有在末尾添加空字符 '\0'。由于 `%s` 格式要求以空字符结尾的字符串，所以 `printf("%s\n", arr2)` 会继续访问 `arr2` 后面的内存，直到遇到第一个空字符为止。这里后面的内存内容是不确定的，因此打印的结果是不确定的。
3. `arr3` 是一个使用字符列表初始化的字符数组，并在末尾显式添加了空字符 '\0'。因此，`printf("%s\n", arr3)` 打印的是完整的字符串 "bit"。

> C语言中的字符串是以空字符 '\0' 结尾的字符数组，空字符标志着字符串的结束。如果字符串没有以空字符结尾，那么字符串处理函数（如 `printf`、`strcpy`、`strlen` 等）将无法正确处理字符串，可能会导致意外的结果或错误。因此，在操作字符串时，确保字符串以空字符 '\0' 结尾是非常重要的。

转义字符

假如我们要在屏幕上打印一个目录 `c:\code\test.c`

我们该如何写代码？

```
#include <stdio.h>
int main() {
    printf("c:\code\test.c\n");
    return 0;
}
//输出结果：c:code  est.c
```

这里就不得不提一下转义字符了。转义字符顾名思义就是转变意思。

下面看一些转义字符。

- `\n`：换行符（newline）
- `\t`：制表符（tab）
- `\"`：双引号（double quote）
- `\'`：单引号（single quote）
- `\\`：反斜杠（backslash）
- `\b`：退格符（backspace）
- `\r`：回车符（carriage return）
- `\f`：换页符（form feed）
- `\v`：垂直制表符（vertical tab）
- `\a`：响铃符（alert）
- `\0`：空字符（null character）
- `\?`：问号（question mark）
- `\ooo`：八进制字符（其中`ooo`表示一个八进制数，范围为`\000`到`\377`）
- `\xhh`：十六进制字符（其中`hh`表示一个十六进制数，范围为`\x00`到`\xFF`）

需要注意的是，如果在字符串字面值中使用了一个未知的转义字符序列，或者在字符字面值中使用了一个超过范围的八进制或十六进制数，编译器可能会报错或产生未定义的行为。因此，在使用转义字符时，请确保了解其正确的语法和含义。

```c
#include <stdio.h>
int main()
{
    //问题1：在屏幕上打印一个单引号'，怎么做？
    //问题2：在屏幕上打印一个字符串，字符串的内容是一个双引号“，怎么做？
    printf("%c\n", '\'');
    printf("%s\n", "\"");

    return 0;
}
```

笔试题：

```c
//程序输出什么？
#include <stdio.h>
int main()
{
    printf("%d\n", strlen("abcdef"));  //6
    // \62被解析成一个转义字符
    printf("%d\n", strlen("c:\test\628\test.c"));  //14

    return 0;
}
```

注释

1. 代码中有不需要的代码可以直接删除，也可以注释掉
2. 代码中有些代码比较难懂，可以加一下注释文字

比如：

```
#include <stdio.h>

int Add(int x, int y) {
    return x + y;
}
/*C语言风格注释
int Sub(int x, int y)
{
    return x-y;
}
*/
int main() {
    //C++注释风格
    //int a = 10;
    //调用Add函数，完成加法
    printf("%d\n", Add(1, 2));

    return 0;
}
```

注释有两种风格：

- C语言风格的注释 `/*xxxxxx*/`
  - 缺陷：不能嵌套注释
- C++风格的注释 `//xxxxxxxx`
  - 可以注释一行也可以注释多行

常见关键字

```
auto  break   case  char  const   continue  default  do   double else  enum
extern float  for   goto  if   int   long  register    return   short  signed
sizeof   static struct  switch  typedef union  unsigned   void  volatile  while
```

C语言提供了丰富的关键字，这些关键字都是语言本身预先设定好的，用户自己是不能创造关键字的。

关键字 typedef

> typedef 顾名思义是类型定义，这里应该理解为类型重命名。

示例：

```c
//将unsigned int 重命名为uint_32, 所以uint_32也是一个类型名
typedef unsigned int uint_32;
int main() {
    //观察num1和num2,这两个变量的类型是一样的
    unsigned int num1 = 0;
    uint_32 num2 = 0;
    return 0;
}
```

关键字static

在C语言中： static是用来修饰变量和函数的

> 1. 修饰局部变量-称为静态局部变量
> 2. 修饰全局变量-称为静态全局变量
> 3. 修饰函数-称为静态函数

修饰局部变量

示例:

```c
#include <stdio.h>
void test() {
    int i = 0;
    i++;
    printf("%d ", i);
}

int main() {
    int i = 0;
    for (i = 0; i < 10; i++) {
        test();
    }
    return 0;
}
```

输出结果:

```
1 1 1 1 1 1 1 1 1 1
```

加上static:

```c
#include <stdio.h>
void test() {
    //static修饰局部变量
    static int i = 0;
    i++;
    printf("%d ", i);
}

int main() {
    int i = 0;
    for (i = 0; i < 10; i++) {
```

```
        test();
    }
    return 0;
}
```

输出结果：

```
1 2 3 4 5 6 7 8 9 10
```

对比代码1和代码2的效果理解static修饰局部变量的意义。

结论：

> **static修饰局部变量改变了变量的生命周期**
>
> **让静态局部变量出了作用域依然存在，到程序结束，生命周期才结束。**

修饰全局变量

示例：

```
//代码1
//add.c
int g_val = 2018;
//test.c
int main() {
    printf("%d\n", g_val);
    return 0;
}

//代码2
//add.c
static int g_val = 2018;
//test.c
int main() {
    printf("%d\n", g_val);
    return 0;
}
```

代码1正常，代码2在编译的时候会出现连接性错误。

结论：

> **一个全局变量被static修饰，使得这个全局变量只能在本源文件内使用，不能在其他源文件内使用。**

修饰函数

示例：

```
//代码1
//add.c
int Add(int x, int y) {
    return x + y;
}
//test.c
```

```c
int main() {
    printf("%d\n", Add(2, 3));
    return 0;
}
//代码2
//add.c
static int Add(int x, int y) {
    return x + y;
}
//test.c
int main() {
    printf("%d\n", Add(2, 3));
    return 0;
}
```

代码1正常，代码2在编译的时候会出现连接性错误.

结论：

> **一个函数被static修饰，使得这个函数只能在本源文件内使用，不能在其他源文件内使用。**

#define 定义常量和宏

```c
//define定义标识符常量
#define MAX 1000
//define定义宏
#define ADD(x, y) ((x) + (y))
#include <stdio.h>

int main() {
    int sum = ADD(2, 3);
    printf("sum = %d\n", sum);//sum = 5

    sum = 10 * ADD(2, 3);
    printf("sum = %d\n", sum);//sum = 50

    return 0;
}
```