# Introduction

## concepts

literal: a symbol which uniquely identifies its value. A literal means itself.

example: 'A', 100, 20.0,

expression: combination of variables, operator and literals

example: i+100;

escape characters: \r return to the beginning of the line. \a alarm. \0 is null

a comment placed anywhere inside the code is a syntactical equivalent of a space.

a file which contains a computer program translated into machine language is called an executable file.

## storage class

storage classes are used to describe the features of a variable/function.

there are 4 types: auto, extern, static, register

|  |  |
| --- | --- |
| int a=1; | define and initialize int variable. should be initialized in the future before calling. can be updated. It works in current scope or children scope if no local variable is defined. |
| static int a=1; | define and initialize a static variable. allocated memory address with the value 0. Only being initialized once. Can't be updated in the future. |
| extern int a;  //extern int a=1 is ok but not required. | extern variable. The scope differs from usual variable. it can be used/updated in other source file. extern class enable to declare a variable without initialization. |
| extern static int a=1; | it can't be used/updated in other source file. |
|  |  |

The auto storage class is deprecated now.

*extern* is used to declare a global variable or function in another file. The **extern**  gives a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

For **functions**, the extern only tells the compiler that linkage is extern. As this is the default (you use the keyword static to indicate that a function is not bound using extern linkage) you don't need to use it explicitly. The function declaration of "extern int fun();" indicates the function fun() may exists in another header file. The extern works as the linker to avoiding duplicating loading.

|  |  |
| --- | --- |
| extern int i  i = 10; | declare extern variable in one source file. Modify it when has link to a definition in another source file. |
| //defined in one file  extern int variable = 2;  void increment(void){      variable++;  }  //declare in another file  int variable;  void increment(void);  int main(void){      variable = 0;      increment();      printf("%d\n", variable);      return 0;  } | output:1  extern variable |
| int main()  {      extern int i;      printf("%d", i);      return 0;  } | compiling error: 'i' has both 'extern' and initializer  can't define extern variable within a function. extern variable should be initialized as zero once it is declared. that can't be initialized again when the function is executed. |
|  |  |
|  |  |

Static variables preserve their value even after they are out of their scope!

1. A static variable inside a function keeps its value between invocations.
2. A static global variable or a function is "seen" only in the file it's declared in

|  |  |
| --- | --- |
| void func(void) {      int i = 0;      printf("entering fun: i = %d\n", i);      i++;      printf("exiting fun: i = %d\n", i);      return;  }  void main(void){      func();      func();  } | entering fun: i = 0  exiting fun: i = 1  entering fun: i = 0  exiting fun: i = 1 |
| void func(void) {  **static int i = 0;**      printf("entering fun: i = %d\n", i);      i++;      printf("exiting fun: i = %d\n", i);      return;  }  void main(void){      func();      func();  } | entering fun: i = 0  exiting fun: i = 1  entering fun: i = 1  exiting fun: i = 2  **the variable I is static type, which is only set up once it is initialized** |
| |  | | --- | | #include <stdio.h>  int main()  {      static int x;      int y;      printf("%d \n %d", x, y);  } | | output: 0, unknown  static variable will be allocated to memory address even if that is not initialized. |
| int initializer(void)  {      return 50;  }    int main()  {      static int i = initializer();      printf(" value of i = %d", i);      getchar();      return 0;  } | compiling error.  static variable is only initialized by literal rather than function returns  The code is ok after remove static |
| int main()  {      int a, b;      static int i = (4)? (a=2, b=3) : (a=3, b=4);      printf(" value of i = %d", i);      return 0;  } | compiling error  remove static keyword |
| /\* module1.c \*/  static int variable;  void fun1(void) {  variable = 1;  }  /\* module2.c \*/  #include <stdio.h>  extern int variable;  void increment(void);  int main(void) {  variable = 0;  increment();  printf("%d",variable);  return 0;  } | A local variable is declared as static. The variable is not going to be accessible in other modules, even with the extern specifier.  "extern static variable" only work in source file. "extern variable" can’t be used in other source file. |
| /\* module1.c \*/  static void fun(void) {  puts("#1");  }  /\* module2.c \*/  void fun(void) {  puts("#2");  }  /\* module3.c \*/  #include <module1.c>  #include <module2.c>  void fun(void);  int main(void) {  fun();  return 0;  } | If the static attribute precedes the declaration of the function, it means that the function has internal linkage to the parent module.  The function's scope is limited to the module in which the function's declaration occurs.  Note that removing the static specifier will cause the program to be no longer correct. |
| int v=10;  int f(int p){  **static int v=1;**      v++;      return v+p;  }  int main(void){      int a = f(1)+f(1);      printf("%d\n",a);      return 0;  } | output: 7  v is static variable. v overrides the global variable v. Only declare once.  if remove static, the output is 6 |

register variables have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available.

|  |  |
| --- | --- |
| int fun(register int arg) | The only storage-class specifier that shall occur in a parameter declaration is register.  Either int fun(static int arg) or int fun(extern int arg) is invalid |
|  |  |
|  |  |
|  |  |
|  |  |

## typedef

the literal typedef is used for defining alias for a data type.

typedef can be used to alias **compound data types** such as struct and union, or pointer to these compound types, or a function pointer, or an array or other data type.

|  |  |  |
| --- | --- | --- |
| typedef int INT, \*INTPTR, ONEDARR[10], TWODARR[10][10];      INT x = 3;      printf("%d\n", x);      INTPTR y =&x;      printf("%d\n", \*y);      ONEDARR z = {1,2,3,4,5,6,7,8,9,0};      printf("%d\n", z[3]);      TWODARR t = {{1,2,3,4,5,6,7,8,9,0}};      printf("%d\n", t[0][3]); | “INT x” would define x of type int. “INTPTR \*y” would define pointer y of type int \*\*. “ONEDARR z” would define z as array of 10 int. “TWODARR t” would define t as array of 10 by 10 int. | |
|  | |  |

|  |  |
| --- | --- |
| usual approach | use typedef |
| struct S{          int age;      };      struct S s={45};      printf("%d\n", s.age); | typedef struct S{          int age;      } user;      user s;      s.age=45;      printf("%d\n", s.age); |
| int main()  {      unsigned long long a= 456;      printf("%d\n", a);      return 0;  } | typedef unsigned long long ull;  int main()  {      ull a= 456;      printf("%d\n", a);      return 0;  } |
| double a = 45.6;      double \*p = &a;      printf("%f\n", \*p); | typedef double\* dp;      double a = 45.6;      dp p = &a;      printf("%f\n", \*p); |
| double a[] = {45.6};      double \*p = &a;      printf("%f\n", \*p); | typedef double \* dp;      double a[] = {45.6};      dp p = &a;      printf("%f\n", \*p); |

#define is a C-directive which is also used to define the aliases for various data types similar to typedef but with the following differences:

typedef is limited to giving symbolic names to types only. #define can be used to define alias for values as well, q., you can define 1 as ONE etc.

typedef interpretation is performed by the compiler. #define statements are processed by the pre-processor.

# Type: int, float, char

## declaration

* Various format: 24 is decimal. 024 is octadecimal. 0x24 is hexadecimal. 0b011 is binary.
* A **long int** is 32 bits(8 bytes), and a **short int** is 16 bits (2 bytes), and normal int can be 16 or 32 bits. 16 (2 bytes) and 32(4 bytes) bits denote minimum size of a memory address, which can store 2^16=65536, 2^32= 4,294,967,296. Unassigned int is only positive integer.
* Each variable has the attributes of type, name, value
* C compiler is case-sensitive
* data of type of float is a fractional number.  
  float a; a=18/5;  
  float a=18/5.0;  
  float a,b=4/3; a=b;
* char type: ASCII code with 0-7 digits  
  'A' 65, 'a' 97, ' ' 32  
    
  c='A' encompassed by apostrophes'' OR  
  c=65, 65 is ASCII code of of 'A'  
  c='\65' octal digits  
  c='\x27' hexadecimal digits  
    
  You can always assign a char value to an int variable;  
  You can always assign an int value to a char variable, but if the value exceeds 255 (the top-most character code in ASCII), you must expect a loss of value;  
  The value of the char type can be subject to the same operators as the data of type int.
* specific character: \n transition to a new line  
  \r return to the beginning of the line  
  \a in alarm  
  \0 Nullus

The data type below for integer is correct:

unsigned integer is zero or positive number. signed integer is default pattern

|  |  |
| --- | --- |
| int a; | 4 bytes or 2 bytes in DOS |
| short a; | 2 bytes |
| long a; | 4 bytes |
| long long a; | 8 bytes |
| unsigned a; | only positive number |
| signed a; | 4 bytes == int a == signed int a;  Note: As per C standard, “int, signed, or signed int” are equivalent. Similarly, “unsigned, or unsigned int” are equivalent. Besides, “long, signed long, long int, or signed long int” are all equivalent. And “long long, signed long long, long long int, or signed long long int“ are equivalent. |
| float a; | 4 bytes |
| double a; | 8 bytes; |
| long double a; | 10 bytes |

The approaches for declaration below are equal:

|  |  |
| --- | --- |
| int a;  a=10;  char b;  b='B';  float c;  c=1.3; | int, char or float is entity of type­. a, b,c is identifier.  unsigned int a; OR unassigned a; unsigned long int a; OR unsigned long a; |
| int a = 3;  char b = 'A';  float c = 1.2; |  |
| int a=4,b=2; |  |
| int a, b=2; | b==2, a is unpredictable value |
|  |  |
|  |  |
| int a=1, char b='B'; | compiling error |
| int a=b='B'; | compiling error |

|  |  |
| --- | --- |
| int a= 024;      printf("%d",a); | output: 20  0 octadecimal lieteral: a=2\*8+4=20 in decimal |
| int a= 0x24;      printf("%d",a); | output: 36  0x is hexadecimal literal. a=2\*16+4=36 |
| int t=0b10;      printf("%d", t); | output: 2  binary literal: 1\*2^1+0=2 |
| int a= 1.12E1;      printf("%d",a); | output: 11  scientific literal 1.12E1=11.2, a is integer. so a=11  3 • 10^8: It reads: “three times ten to the power of eight” -> 3E8. that is "scientific notation" |
| int a= 1000E-2;      printf("%d",a); | output: 10  scientific literal |
| int a= '\0';      printf("%d",a); | output: 0 |
| int a= '0';      printf("%d",a); | **output: 48**  **'0' is char type, so return ASCII code '0'=48**  **In C, character (0-255) and integer are equivalent** |
| int a= '\0';      printf("%d",a); | **output: 0**  **equal to int a=0;** |
| int a= "0";      printf("%d",a); | **output: unpredictable value**  **"0" is char array. no compiling error but that is wrong.** |
| int zip = 92126 | only works in 32 bits. In 16 bits system namely MS-DOS cause error. |
| int a= 00024;      printf("%d",a); | output: 20  no compiling error, but not suggested. |

## type cast

type cast is a conversion from one type to another.

implicit type conversion done by the compiler:

**bool -> char -> short int -> int ->**

**unsigned int -> long -> unsigned ->**

**long long -> float -> double -> long double**

data of type char or short int will be converted to type int (this is called an integer promotion);

if there is any value of type float in the expression, the other data will be converted to float;

if there's any value of type double in the expression, the other data will be converted to double;

if there's any value of type long int in the expression, the other data will be converted to long int.

|  |  |
| --- | --- |
| int x=10;  char y ='a';  x=x+y;  float z=x+1.0 | y implicitly converted to int based on ASCII  x is implicitly converted to float. |
| int i=1;      float a= 1 + i/2;      printf("%d,%f\n", i,a); | output: 1,1.000000  int i/2= int 1/2=0 |
| int i=3;      int a= 1.5 + i/2;      printf("%d\n", a); | output: 2  a= 1+3/2=1+1=2  convert 1.5 to integer 1, convert 3/2 to integer 1 |
| int i=3;      printf("%f\n", 1.5 + i/2); | output: 2.500000  keep int of i |
| int i=3;      printf("%f\n", 1.5 + i/2.0); | output: 3.000000  convert int i to float i |

explicit type conversion

|  |  |
| --- | --- |
| double x=1.2;  int sum = (int)x +1; | convert double to int |
|  |  |
|  |  |
|  |  |

## operator

Unary operators have one operand (e.g., the negate operator; think of a unicycle, with one wheel). The could be !, ~, ++, --, -(unary), + (unary), \*(dereference), &(address of), sizeof()

Binary operators have two operands (e.g., the multiply operator; think of a bicycle, with two wheels).

=/- can be unary prefix operator or binary infix operator. For example: -67, +45, 6-7,5+6

multiply operator \* or division operator / is binary operator rather than unary operator. but \* can work as the dereference operator for pointer. That is unary operator

* asterik \* is multiplication operator.   
  slash / is divisional operation.   
  plus sign + is addition operator.  
  minus sign - is substraction operator  
  unary minus namely -100  
  unary plus namely +100  
  percentage sign % is remainder operator (modulo process)
* a digraph && (ampersand ampersand) AND

the digraph | | (bar bar)

~tilde, ^caret

|  |  |
| --- | --- |
| int a,b;      a=b+10;      printf("%d\n",a); | output: unpredictable  b is not initialized. |
| int a,b=1;  **a=b\*-10;**      printf("%d\n",a); | output: -10  though "a=b\*-10;" is ok, but "a=b\*(-10);" will be good readable.  " |
|  |  |
|  |  |
|  |  |

### priority

* priority:   
  left to right  
  unary ++ -- + - >> \*/ >> =-  
  parentheses from inner to outside
* time for an updated **priority table**. Study it carefully – it’ll come in very handy soon.

|  |  |  |
| --- | --- | --- |
| **1** | !, ~ (type), ++, --, +, -, \*, &, sizeof | **unary** |
| **2** | \*, /, % |  |
| **3** | +, - | **binary** |
| **4** | <<, >> |  |
| **5** | <, <=, >, >= |  |
| **6** | ==, != |  |
| **7** | & |  |
| **8** | | |  |
| **9** | && |  |
| **10** | || |  |
| **11** | =, +=, -=, \*=, /=, %=, &=, ^=, |=, >>=, <<= |  |

### condition operator:

&& and, || or, ! not

|  |  |
| --- | --- |
|  | output |
| int a= 4,b= 3,c= 2,d= 1,m= 2,n= 2;      (m = a < b) && (n = c > d);      printf( "%d%d\n",m,n); | 02  for && and ||, if the first expression meet the condition, the 2nd expression after operator would not be operated. The expression m=a<b is False. the expression n=c>d would be not be executed. so m=0, n=2 |
| int Add(int n){      n && (n += Add(n-1));      return n;  }  int main()  {      int n=10;      int t=Add(n);      printf("%d\n", t);      return 0;  } | output:55 |
|  |  |

### bitwise operator

|  |  |  |
| --- | --- | --- |
| 1>>0 | 1 | 1/(2^0)=1 |
| 1>>1 | 0 | 1/(2^1)=0 |
| 1>>2 | 0 | 1/(2^0)=0 |
| 1<<2 | 4 | 1\*(2^2)=4 |
| 0<<2 | 0 | 0\*(2^2)=0 |
| 0<<0 | 0 |  |
| int i=1,j=0,k=0;      k=(i>>j)+(j>>i)+(i>>i)+(j>>j);      k<<=i;      printf("%d\n",k); | 2 | k=1>>0+0>>1+1>>1+0>>1=1+0+0+0=1  k<<=I, k=k<<i=1<<1=1\*2^1=2 |
| int i=1,j=0,k;      k = (i<<j)+(j<<i)+(i<<i)+(j<<j);      k >>= i;      printf("%d\n",k); | 1 | k=1<<0+0<<1+1<<1+0<<0  =1\*2^0+0\*2^1+1\*2^1+0\*2^0=1+2=3  k=3>>1=3/2=1 |
| 5&12 | 4 | 5= 101  12= 1100  100 = 2^2=5 and |
| 5|12 | 13 | 5= 101  12= 1100  1101 = 2^3+2^2+1=13 or |
| 5^12 | 9 | 5= 101  12= 1100  1001 = 2^3+1=9 only different |
| ~12 | -13 | -(n+1) |
| int i=1,j=0,k;      k=(i&j)+(i|j)+(i^j)+!i+j;      printf("%d\n", k); |  | output: 2  k=1&0+1|0+1^0+!1+0 = 0+1+1+0=2 |
| int a=1, b=2;      int c = a|b;      int d = c&a;      int e = d^0;      cout << e << d << c; |  | output: 113  c=1|2=b01|b10=b11=3  d=b11&b01=1  e=1^0=1 |
| int a=4, b=9;      int c = a|b;      int d = c&a;      int e = d^0;      cout << e << d << c; |  | output: 4413  a=4=b100, b=9=b1001  c=b100|b1001=b1101=2^3+2^2+1=8+4+1=13  d=b1101&b100=b100=4  e=b100^0=b100=4 |

### ++ and --

increment operator ++ and decrement operator -- are unary operator. the statement "++a;" will execute "a plus one".

prefix: b=++a, a plus one and assign a to b.

postfix: b=a++, assign a to b firstly, and then a plus one

|  |  |  |
| --- | --- | --- |
|  | output |  |
| int a=5;      int b=a++;      int c =++a;      printf("%d,%d,%d", a,b,c); | a=7,  b=5  c=7 |  |
| int a=5;      printf("%d", a++);      printf("%d", a); | 56 |  |
| int i=11;      for(i--; i--;i--){          printf("%d", i);      } | 97531 | "for(i--; --i;i--)" will cause dead loop |
| int i=10;      for(i--; --i;i--){          printf("%d", i);      } | 8642 |  |
| int i=1;      int a = i++ \* 5 + ++i \* 3;      printf("%d,%d", i,a); | 3,14 | a=1\*5+3\*3=14 |
| int i, j;      i = 4;      j = 2\*i++;      i += 2\*--j;      printf("%d,%d", i,j); | 19,7 | j=2\*4=8, i=5  i=5+2\*7=19, j=7 |
| int a=2;      printf("%d", a++);      printf("%d\n", a); | 23 |  |
| int a=10;      printf("%d,%d\n", a++, ++a); | 11,12 |  |
| int a=1, b=2, c;      c = ++a+a+++b;      printf("%d,%d\n", a, c); | 3,7 | c=(++a)+(a++)+b |

## use pointer

|  |  |
| --- | --- |
| int \*p;  char \*ptr; | \*p is a pointer to data type of int or char |
|  |  |
| int i=10;  **int \*p=&i;**  **int \*\*p2=&p;**      printf("%d,%d", \*p, \*\*p2); | int \*\*a;  char \*\*a;  define a pointer to another pointer, which is a pointer to data type of int or char.  Note: "int \*(\*p2)=&p;" is acceptable |
| int \*p[3];      for(int i=0; i<3; i++){          p[i]=&i;          printf("%d\n",\*p[i]);      } | int \*p[3] is a pointer array, of which there are pointers to 3 integers |

# Type: array

|  |  |
| --- | --- |
| int a[10];  char b[10]; | declare int, char array |

## integer array

|  |  |  |  |
| --- | --- | --- | --- |
|  | strlen(a) | sizeof(a) |  |
| int a[5]={1,3,5,7,9};  OR  int a[5];  a={1,3,5,7,9}; | 1 | 5\*4=20 |  |
| int a[]={1,3,5,7,9}; | 1 | 20 |  |
| int a[10]={1,3,5,7,9}; | 1 | 10\*4=40 | values of the first 5 address are 1/3/5/7/9. Other values are assigned as 0.  a[5]=0 |
| int a[10]; | 15 | 40 | declare array but doesn't initialize it. |
| int a[10]={}; | 0 | 40 | Values of all 10 address are assigned as 0. |
| int a[3]= {1,3,5,7,9}; | 1 | 3\*4=12 | Illegal: out of bounds. compile could be ok. but the other elements may be lost, and execution may cause segmentation fault  a[3] may return unpredictable value |
| int a[5]= {1,3,5,7,9};  printf("%d, %d\n", a[0], a[10]); |  |  | Illegal: out of bounds. a[10] will return unpredictable value |
|  |  |  |  |

Note: 1. strlen(a) is 1, "a" is equivalent to pointer that address a[0].

|  |  |
| --- | --- |
| int i=2;  printf("int=%d, size=%d\n", sizeof(int),  sizeof(i));  int a[]={0,1,2,3};  printf("size=%d, length=%d\n", sizeof(a),  sizeof(a)/sizeof(a[0]));  a[4]=4;  printf("%d\n", a[4]);  printf("%d\n", a[5]);  printf("size=%d, length=%d\n", sizeof(a),  sizeof(a)/sizeof(a[0])); | output:  int=4, size=4  size=16, length=4  4  6422284  size=16, length=4  Note: It is ok to add value which is out of bounds. The size of an array can't be changed. |
| **int arr[8] = {1,2, [5]=9, 10};**      for(int i=0; i<8; i++){          printf("%d",arr[i]);      } | output: 120009100  initialize array |
| int digits[11]={0}; | initialize integer array. All 11 elements are assigned to zero. |

## 2D array

|  |  |  |
| --- | --- | --- |
| **Pointer Notation** | **Array Notation** | **Description** |
| \*\*ptr | \*array[] | Declares an array of pointers. |
| \*ptr | array[0] | The address of the first pointer in the array; for a string array, the first string. |
| \*(ptr+0) | array[0] | The same as the preceding entry. |
| \*\*ptr | array[0][0] | The first element of the first pointer in the array; the first character of the first string in the array. |
| \*\*(ptr+1) | array[1][0] | The first element of the second pointer in the array; the first character of the second string. |
| \*(\*(ptr+1)) | array[1][0] | The same as the preceding entry. |
| \*(\*(ptr+a)+b) | array[a][b] | Element b of pointer *a.* |
| \*\*(ptr+a)+b | array[a][0]+b | This item isn’t really what you want. What this item represents is the value of element 0 at pointer *a* plus the value of variable *b*. Use the \*(\*(ptr+a)+b) notation instead. |

|  |  |
| --- | --- |
| int a[2][3]={1,2,3,4,5,6};  printf("%d\n", a[1][0]); | 4 |
|  |  |
| int a[2][3]={{1,2,3},{4,5,6}};  printf("%d\n", a[1][0]); | same as the above |
| char a[7][10]={"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};  printf("%c\n", a[1][0]); |  |
| char a[4][20]={"Mary Smith", "James Johnson",  "Patricia Williams", "John Brown"};  for(int i=0;i<4;i++){  printf("%s\n", a[i]);  } | output:  Mary Smith  James Johnson  Patricia Williams  John Brown |
| int \*p [2][4];      for(int i=0; i<2; i++){          for(int j=0; j<4; j++){              p[i][j]=&i;              printf("%d", \*p[i][j]);          }      } | pointer to 2D array |
| int row=2, col=3;      char \*p[row];      for(int i=0;i<row;i++){          p[i] = (char \*) malloc(col\*sizeof(char));          for(int j=0;j<col;j++){              p[i][j] = 'A'+i+j;              printf("%c",p[i][j]);          }      } | output: ABCBCD |
| int row=2, col=3;      char \*\*p;      p = (char \*\*) malloc(row\*sizeof(char \*));      for(int i=0;i<row;i++){          p[i] = (char \*) malloc(col\*sizeof(char));          for(int j=0;j<col;j++){              \*(\*(p+i)+j) = 'A'+i+j;              printf("%c",\*(\*(p+i)+j));          }      } | output: ABCBCD  p is array of pointer  pointer to pointer initialize 3x3 char matrix |
| int row=3, col=3;  **char \*p[row];**  **p[0]=(char \*)malloc(col\*sizeof(char));**  p[0][0]='A';  p[0][1]='B';  p[0][2]='C';  p[1]="abc";  printf("%c, %s\n", p[0][0], p[1]); |  |
|  |  |
| int a[2][3];  a={1,2,3,4,5,6}; | Illegal: wrong syntax |
| int a[][]={1,2,3,4,5,6}; | Illegal: wrong syntax |
|  | compare two matrices |

## char array

### declare and initialize

the below approaches are working.

|  |  |  |  |
| --- | --- | --- | --- |
|  | strlen(a) | sizeof(a) |  |
| char a[5]="ABCDE"; | 5 | 6 | string is denoted as char array |
| char a[]="ABCDE"; | 5 | 6 | same as the above |
| char a[]={'A','B','C','D','E','\0'}; | 5 | 6 | same as the above |
| char a[6]={'A','B','C','D','E','\0'}; | 5 | 6 |  |
| char a[5];  strcpy(a, "ABCDE"); | 5 | 6 | **Here size could not be ignored.** |
| char \*a;  a= (char \*) malloc(10\*sizeof(char));  strcpy(a, "ABCDE"); | 5 | 4 | **\*a is pointer, so its size is 4.** |
| char (\*a)[6];      strcpy(a, "ABCDE");      printf("%d, %d\n",sizeof(a), strlen(a));      for(int i=0;i<10;i++){          printf("%c", \*((\*a)+i));      } | 5 | 4 | **\*a is pointer to char array.** |

Here I summarize some key points

1. string is denoted as char array.

2. C take char as integer 0-255. "1" is denoted 49, and "\1" is denoted as 1

2. string must be encompassed by "" rather than ''.

3. end of a string is identified by '\0'. '\0' is automatic added into the end when initialization is done. In some unusual cases: '\0' is removed manually, or initialized array size is wrong for example, char a[3]="ABC", size should be >=4.

4. sizeof(str) is determined by initialization, and can't be changed. That is initialized number, or the number of characters plus one ('\0') if there is no number initialized.

5. strlen(str) is number of characters starting from current pointer position till '\0'.

|  |  |  |  |
| --- | --- | --- | --- |
|  | strlen(a) | sizeof(a) |  |
| char a[]={'A','B','C','D', '\0'}; | 4 | 5 |  |
| char a[]={'A','B','C','D'}; | unknown | 4 |  |
| char a[]={'A','B','\0','D','\0'}; | 2 | 6 |  |
| char a[]=""; | 0 | 1 | '\0' take one character |
| char a[]="\1\n\0"; | 2 | 4 |  |
| **char a[]="\0\1\2\3\4";** | **0** | **6** | **\0 is on the first place** |
| char a[]= "123""123"; | **6** | **7** | **The string is "123123", where double " will be ignored. But single " will cause compiling error.** |
| char a[6]={'A','B','C','D'}; | 4 | 6 | The 6 values are A,B,C,D,0,0 |
| char a[6]="ABC\0D"; | 3 | 6 | The 6 values are A,B,C,0,D,0 |
| char a[20]={'a','b','c','d'}; | 4 | 20 |  |
| char a[3]="ABC"; | unknown | 3 | illegal: without '\0'. though compiling may pass. |
| char a[3]="ABCDE"; | unknown | 3 | illegal: out of bounds. |
| char a[5];  a="ABCDE"; |  |  | syntax error. compiling failed |

|  |  |
| --- | --- |
|  | Export |
| char a[]="ABCDE";  printf("%s\n", a); | ABCDE |
| char a[]="ABCDE";  printf("%c\n", a[0]); | A |
| char a[]="ABCDE";  printf("%c\n", a[10]); | Illegal: out of bounds. unpredictable value or NULL |
| char a[]="ABCDE";  printf("%c\n", \*(a+3)); | C |
| char a[]="ABCDE";  printf("%c\n", \*(a+1)); | B printf("%c\n", \*(a++)); cause compiling error. |
| char a[]="AB\0CDE";  strcat(a, 'abc');  printf("%s\n", a); | ABabc |
| char a[]="\0\1\2\3\4";      printf("<%s>", a); | nothing is printed |
| char a[]="\1\2\3\4";      printf("<%s>", a); | output: <☺☻♥♦> |

|  |  |
| --- | --- |
| int func(int a){         return a\*2;      }  **int (\*p)(int);**      p = func;      printf("%d\n",p(3)); | declare a pointer to a function which has one integer argument and returns integer. |
| float (\*p) (float x, float y) | a function whose return value is type of float |
| int func(void){          printf("ok\n");          return 1;      }  **int (\*p) ();**      p=func;      p(); | **int (\*p) ();**  pointer p of func(): |
| int func(int a){          return a+10;      }  **int (\*p[3]) (int a);**      for(int i=0; i<3; i++){  **p[i]=func;**  **int b= p[i](i);**          printf("%d\n",b);      } | declare array of pointers, and each pointer is to function func() with one integer argument  **int (\*p[3]) (int a);** |

### '\0'

2. '\0' is denoted as terminator of string. strlen() count number of characters before '\0'. sizeof() is number of character plus by one '\0'.

|  |  |
| --- | --- |
| void f(char \*s){      s[1]='\0';  }  int main(){      char p1[]="ABC", p2[]="XYZ";      f(p1);f(p2);      printf("%d\n",strlen(p1)+strlen(p2));      return 0;  } | output: 2  p1:"A\0C\0", strlen(p1)=1  p2:"X\0Z\0", strlen(p2)=2 |
| struct Q{          char S[3];      };      struct S{          struct Q Q;      };      struct S S={'\0','\0','\0'};      S.Q.S[0]='A';      S.Q.S[2]='B';      printf("%d\n", strlen(S.Q.S)); | output: 1  S="\0\0\0" -> S="A\0B" -> strlen(S)=1 |
| FILE \*f;      char s[]="abcdefgh";      char \*p=s+2;      p[4]='\0';      f=fopen("f","wb");      fputs(s,f);      fclose(f); | output: abcdef= 6 bytes  "**a**bcdefgh"->"ab**c**defgh"->"abcdef\0h"  two pointers: s ->0, p->2 |
|  |  |
| FILE \*f;      char s[]="abcdefgh";      char \*p=s+2;      p[4]='\0';      f=fopen("f","wb");      fputs(s+2,f);      fclose(f); | output: cdef  compared with the above  two pointe: p->2, s->0, but s+2 only export s->2 |
| FILE \*f;  char s[]="Mary had a little lamb";  char \*p=s+2;  p[4]='\0';  f=fopen("f", "wb");  fputs(s,f);  fclose(f); | output: "Mary h"= 6 bytes  p->2 -> p-4='\0' -> "Mary h\0d a little lamb" |
| int a[3]={1,2,3};  **int (\*p)[3];**  **p=a;**      for(int i=0; i<3; i++){  **printf("%d\n", (\*p)[i]);**      } | declare a pointer \*p to an array of 3 integers. |
| char tt[20]="90817263154";      strcpy(tt, tt+3);      printf("%d\n", strlen(tt)-tt[9]+'5'); | output:8  Index: 0 1 2 3 4 5 6 7 8 9 10 11  original: 9 0 8 1 7 2 6 3 1 5 4 \0  strcpy: 1 7 2 6 3 1 5 4 \0 5 4 \0  strlen(tt)-tt[9]+'5' = 8-'5'+'5'=8 |
| char tt[20]="0123456789";      strcat(tt+11,"123");      printf("%d", strlen(tt)-tt[8]+'0'); | output: 2  Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  original: 0 1 2 3 4 5 6 7 8 9 \0  strcat: 0 1 2 3 4 5 6 7 8 9 \0 1 2 3 \0  strlen(tt)-tt[8]+'0'=10-'8'+'0'=10-8=2 |
| char tt[20]="0123456789";      strcat(tt+10,"123");      printf("%d", strlen(tt)-tt[8]+'0'); | output: 5  Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13  original: 0 1 2 3 4 5 6 7 8 9 \0  strcat: 0 1 2 3 4 5 6 7 8 9 1 2 3 \0  strlen(tt)-tt[8]+'0'=13-'8'+'0'=5 |
| struct S{          char \*s;      };      struct S \*S = (struct S \*)malloc(sizeof(struct S));      S->s = "123\0""45678";      printf("%d",strlen(S->s+5)+S->s[3]); | output: 4  Index: 0 1 2 3 4 5 6 7 8 9  original: 1 2 3 \0 4 5 6 7 8 \0  strlen(S->s+5) =4  S->s[3]='\0'=0 |

### char pointer

char pointer works for char array

1. must be allocated with memory address, otherwise that cause runtime error.

2. define a char pointer: "char \*p;"

pointer to the first element: "p=arr; " or "p=&arr[0];"

get value: "\*p;"

get value of next two positions: "\*(p+2);" or "(2)[p]"

return value and then move pointer one position: \*p++;

move pointer and then return value: \*++p;

|  |  |
| --- | --- |
| **char \*p = (char \*) malloc(11\*sizeof(char));**  for(int i=0; i<10; i++){  **\*p ='A'+i; printf("%c",\*p);**  **p++;**  }  \*p='\0'; p-=10;  printf("==%s\n", p); | output: ABCDEFGHIJ==ABCDEFGHIJ |
| char \*p = (char \*) malloc(11\*sizeof(char));  for(int i=0; i<10; i++){  **\*p++ ='A'+i;**  **printf("%c",\*(p-1));**  }  \*p='\0'; p-=10;  printf("==%s\n", p); | output: ABCDEFGHIJ==ABCDEFGHIJ |
| char a[] = "good";      char \*p = "good";      printf("%d%d\n", sizeof(a), sizeof(p)); | output:54 |
| char f(char \*n, int m){  **return \*(n+2\*m);**      }      printf("%c\n",f("aAbBcCdD",1)); | output: b  default pointer \*n address the 1st address of string. |
| char f(char \*n, int m){  **return (m+2)[n];**      }      char n[]="aAbBcCdD";      printf("%c%c\n",f(n,1), \*n); | output: Ba |
| char f(char \*n, int m){          return m+2[n];      }      printf("%c\n",f("aAbBcCdD",1)); | output:c |
| char \*p="\0\2\1\3\4";      printf("%d", p[p[2]]+\*(p+1)+p[0]); | ouput:4  p[p[2]]+\*(p+1)+p[0] = p[1]+2+0=2+2+0=4 |
| char x = 'A';      char\* y = (char\*)malloc(sizeof(char));      y = &x;      for (int i = 0; i < 26; i++) {          printf("%c", x);  **y[0] += 1;**      } | output: ABCDEFGHIJKLMNOPQRSTUVWXYZ  Initial:  00001010 (Binary form of 'A')  In loop  00001010 + 1 = 00001011 = B  00001011 + 1 = 00001100 = C  00001100 + 1 = 00001101 = D  and goes on up till Z. |
| const char \*s;      char str[]="hello";      s=str;      while(\*s){          printf("%c", \*++s);          \*s++;      } | output: el  Note:  "s=str;" is equal to "s=&str[0];"  In this case, "\*s++;" equal to "s++;" |
| void print(char \*c){          printf("%s", \*c);      }      print("test"); | output: test |
| void print(char \*c){          printf("%c", \*c);      }      print("test"); | output: t |

## strings

Most of the functions below are defined in the header file <string.h>

**C standard 7.1.1.1**

*A string is a contiguous sequence of characters terminated by and including the first null character.* The term multibyte string is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A pointer to a string is a pointer to its initial (lowest addressed) character. The length of a string is the number of bytes preceding the null character and the value of a string is the sequence of the values of the contained characters, in order.

int \*ptr[10]: This is an array of 10 int\* pointers, not as you would assume, a pointer to an array of 10 ints

int (\*ptr)[10]: This is a pointer to an array with 10 int

It is I believe the same as int \*ptr; in that both can point to an array, but the given form can ONLY point to an array of 10 ints

### strlen()

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

### sizeof

In C, sizeof is literal rather than function

|  |  |
| --- | --- |
| struct Q{          char S[3];      };      struct S{          char S[3];          struct Q Q;      };      struct R{          int I;      };      printf("%d,", sizeof(struct Q));      printf("%d,", sizeof(struct S));      printf("%d\n", sizeof(struct R)); | output: 3,6,4 |
| int i;      i = 10;      printf( "%d",i);      printf( "%d", sizeof(i++));      printf( "%d",i); | **output: 10410**  **Note: sizeof(i++) == sizeof(int) so i++ would not be executed.** |
|  |  |
|  |  |

sizeof(<>)

|  |  |  |
| --- | --- | --- |
| sizeof(int)  sizeof(unassigned int) | 4 |  |
|  |  |  |
| sizeof(float) | 4 |  |
| sizeof(void) | 1 |  |
| sizeof(NULL) | 4 | NULL=0 |
| sizeof(EOF) | 4 | EOF=-1 |
|  |  |  |
| sizeof(short)  sizeof(short int) | 2 | "short float" is wrong |
| sizeof(long)  sizeof(long int) | 4 | "long float" is wrong |
| int a[]={1,2,3,4,5,6};      printf("%d\n",sizeof(a)); | 24 |  |
| int a[]={{1,2,3},{4,5,6}};      printf("%d\n",sizeof(a)); | 8 | a is 1D array. size=2  but actually a[2]={1,4} |
| int a[2][3]={{1,2,3},{4,5,6}};      printf("%d\n",sizeof(a)); | 24 | a is 2D array |
| char a[]={}; | 0 |  |
| char a[6]={1,2,3,4,5,6};  char a[2][3]={{1,2,3},{4,5,6}}; | 6 |  |
| sizeof(char)=1 | 1 |  |
| sizeof('A') | 4 | 'A' is character, 'A'=65 is integer |
| sizeof("A") | 2 | "A" is a char array. String should be added with '\0'. |
| char a[]="ABC"; | 4 |  |
| char a[]=""; | 1 | '\0' should be counted |
| char a[]="ABCDEF";      printf("%d\n",sizeof(a));      a[2]='\0';      printf("%d\n",sizeof(a)); | 5,5 | array is static type. size can't be change  but length is changeable |
|  |  |  |
|  |  |  |
| int a=34, \*p=&a;      printf("%d\n",sizeof(p)); | 4 |  |
| char a[]="ABCDEF", \*p=a;      printf("%d\n",sizeof(p)); | 4 | pointer address the first character 'A', so size is 4. |
| struct S{          int a;          char b;          char c[10];      };      printf("%d\n", sizeof(struct S)); | 16 | int is 4, and char is 1, and string is 11. |
| struct S{          int a,b;      } s={1,2};      printf("%d\n", sizeof(s)); | 8 |  |
| struct S1{          int a,b;      };      struct S2{          int a,b;          struct S1 next;      };      printf("%d\n", sizeof(struct S2)); | 16 |  |
| #define S "ABC"      printf("%s, %d\n", S, sizeof(S)); | ABC, 4 | S act as pointer |
| #define S(X) #X      printf("%d\n", sizeof(S(3))); | 2 |  |
| #define S(X,Y) X##Y      char ab[]="ABC";      printf("%d\n", sizeof(S(a,b))); | 4 |  |

### atoi() and itoa()

|  |  |
| --- | --- |
| int I = 123;      int ii =I;      char ic[3];      itoa(ii,ic,10);      printf("int=%d, str=%s\n", I, ic); | itoa(): convert integer to string |
| char a[]="456";      int x = atoi(a);      printf("int=%d, str=%s\n", x, a); | convert string to integer  if "a123b", return 0  if "123b", return 123  if "", return 0 |
| // check if illgal unction integer      int is\_digits(char \*ip){          //remove empty string          if(\*ip=='\0') return 0;          //remove string with 0,namely "0012"          if(\*ip=='0'&&\*(ip+1)!='\0') return 0;          while(\*ip!='\0'){              // ACSII 0-9: 48-57              if(\*ip<48 || \*ip>57){                  return 0;              }              ip++;          }          return 1;      }        char ip[] = "127.0.0.1";      char sep[] = ".";      char \*token = strtok(ip, sep);      while(token != NULL){          if (is\_digits(token)){              int ip\_part = atoi(token);              if (ip\_part<=255 && ip\_part>=0){                  printf("%s->%d\n", token, ip\_part);              }          }          token = strtok(NULL, sep);      } | check illegal IP address  Note:  1.is\_digits(): check digits  2. strtok() split string  3. atoi(): convert string to integer |
|  |  |

### strcat()

char \*p = strcat(char \* des, char \* source)

1. two argument should be char type pointer.

**2. the destination pointer should be allocated enough memory.**

3. return value is char type pointer to des pointer

|  |  |
| --- | --- |
| char s[10]="hello";      strcat(s, "123");      printf("%s\n", s); | output: hello123  the same as strcat(s+2, "123"); |
| char s[10]="hello";      strcat(s+6, "123");      printf("%s\n", s); | output: hello  index: 0 1 2 3 4 5 6 7 8  strcat: h e l l o \0 1 2 3 |
| char s[10]="hello";      char \*p =s;      strcat(p, "123");      printf("%s\n", p); | output: hello123  Note: pointer approach |
|  |  |
| strcat("Yello", " World"); *// wrong* | the two arguments should be char type pointer |
| char \*str1 = "India";      char \*str2 = "BIX";      char \*str3;      str3 = strcat(str1, str2);      printf("%s %s\n", str3, str1); | the destination str1 should contain str1+str2. or cause segment fault  correct:      char s[20] = "India";      char \*str1 =s; |

### strncpy()

|  |  |
| --- | --- |
| const char\* from = "john@openedg.org";  char \*t = (char \*) malloc(20);  strncpy(t, from+5, 11);  printf("%s\n",t); | output: opened.org |
|  |  |
|  |  |
|  |  |

### strtok()

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

### strcmp()

strcmp(<str1>, <str2>) return 0 if equal. return 1 if str1 is greater in ASCII or -1 if str2 is greater in ASCII.

The approaches below are equal

|  |  |  |
| --- | --- | --- |
|  | output |  |
| char a[]="abc";      char b[]="abc";      printf("%d\n", strcmp(a,b)); | 0 |  |
| printf("%d\n", strcmp("abc","abc")); | 0 |  |
| char \*a="abc";      char \*b="abc";      printf("%d\n", strcmp(a,b)); | 0 |  |
| char \*a="abc";      char \*b="1abc";      printf("%d\n", strcmp(a,b+1)); | 0 |  |

compare ASCII one by one

|  |  |  |
| --- | --- | --- |
|  | output |  |
| char \*a="abc";      char \*b="1abc"; | 1 | though b is longer than a, ACSII of the first char of string a is 'a', more than that of '1' in string b. |
| char \*a="abc\0";      char \*b="abcd"; | -1 | '\0'=0, 'd'=100 |
| char \*a="";      char \*b="\0"; | 0 | '\0'=0, the first character of string a is '\0' |
| char \*a="abc";      char \*b="abc\0ef"; | 0 | the string b is "abc" |

## enum

Enumeration (keyword enum) is a **user-defined integer collections**, where integers are assigned to readable keywords. It can be used as a flag to check if a certain variable goes through an iteration( as in a for loop, while loop or even an if loop ).

If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0.

The value assigned to enum names must be some integral constant

Two enum names can have same value.

 We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

All enum constants must be unique in their scope.

|  |  |
| --- | --- |
| enum weekday { Monday=1, Tuesday, Wednesday, \          Thursday, Friday, Saturday, Sunday};      enum weekday day;      day = Thursday;      printf("%d\n", day); | output:4  Monday is const with value of 1.  equal to #define Monday 1 |
| enum year {Jan=1, Feb, Mar, Apr, May, \  Jun, Jul, Aug, Sep, Oct, Nov,Dec};      for(int i = Jan; i<=Dec; i++){          printf("%d,", i);      } | output: 1,2,3,4,5,6,7,8,9,10,11,12, |
| enum status {active=1, inactive=0, suspend=0};      printf("%d", active); | output:1 |
|  |  |
|  |  |
|  |  |
| enum state  {working, failed};  enum result {failed, passed};    int main()  { return 0; } | compiling error |

## linked list

a linked list is a chain of items like array, but the length is dynamic.

### basic

|  |  |
| --- | --- |
| struct Node{          char \*value;          struct Node \*next;      }; | define struct node, which is basic component of linked list |
| struct Node \*ptr = (struct Node \*)malloc(sizeof(struct Node));      ptr->value="A";      ptr->next=NULL;      printf("%s\n",ptr->value); | define the linked list with one item. |
| struct Node \*top = (struct Node \*)malloc(sizeof(struct Node));      top->value = "B";      top->next = ptr;      printf("%s, %s\n",top->value, top->next->value); | add one node on the top |
| struct Node \*tail = (struct Node \*)malloc(sizeof(struct Node));      tail->value = "C";      ptr->next= tail;      tail->next = NULL;      printf("%s, %s\n",tail->value); | append one node on the bottom |
| struct Node \*ptr = (struct Node \*)malloc(sizeof(struct Node));      ptr=top;      while(1){          printf("%s\n",ptr->value);          ptr=ptr->next;          if (ptr == NULL)              break;      } | print all nodes |
| struct Node \* a =(struct Node \*)malloc(sizeof(struct Node));      struct Node \* b =(struct Node \*)malloc(sizeof(struct Node));      a->value = "a";      a->next = NULL;  **b=a;**      b->value = "b";      printf("a=%s, b=%s\n",a->value, b->value); | output: a=b,b=b  shallow copy struct pointer: "b=a;"  Changes or deletion will be applied to both. |
| struct Node \* a =(struct Node \*)malloc(sizeof(struct Node));  struct Node \* c =(struct Node \*)malloc(sizeof(struct Node));      a->value = "a";      c->value = a->value;      c->value = "c";      c->next = NULL;      printf("a=%s, c=%s\n",a->value,c->value); | output: a=a,c=c  deep copy |

### create

|  |  |
| --- | --- |
| struct Node \*CreateList(int row, int col, char arr[row][col]){          struct Node \*ptr=(struct Node \*)malloc(sizeof(struct Node));          int i=row-1;          ptr->value=arr[i];          ptr->next=NULL;          for(i=row-2;i>=0;i--){              // printf("%s\n", arr[i]);              struct Node \*new=(struct Node \*)malloc(sizeof(struct Node));              new->value = arr[i];              new->next= ptr;              ptr=new;          }          return ptr;      }      char data[3][10] = {"ABC", "EFG", "HIJ"};      struct Node \*ptr = CreateList(3,10, data);      printf("%s%s%s\n", ptr->value,ptr->next->value,ptr->next->next->value); | convert 2D array to linked list |

### get

|  |  |
| --- | --- |
|  |  |
| void PrintList(struct Node \*ptr){      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      current = ptr;      do{          printf("%s", current->value);          current = current->next;      }while(current != NULL);  } | print values of linked list |
| char \* GetValue(struct Node \*ptr, int index){      struct Node \*current=(struct Node\*)malloc(sizeof(struct Node));      current=ptr;      int i=0;      do{          // printf("%d:%s\n", i, current->value);          if(i==index){              return current->value;          }          current = current->next;          i++;      }while(current != NULL);      return "\0";  } | get value by index in a linked list. |
| int FindValue(struct Node \*ptr, char \*value){      int i=0;      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      current = ptr;      do{          if( strcmp(current->value, value) == 0 ){              return i;              break;          }          current = current->next;          i++;      }while(current != NULL);      return -1;  } | check if a certain value exists in the linked list. return index if that succeeds, or -1 if that fails. |
| struct Node \*GetLastNode(struct Node \* ptr){      struct Node \*tail = (struct Node \*)malloc(sizeof(struct Node));      tail=ptr;      while(tail->next != NULL){          tail=tail->next;      }      printf("%s\n", tail->value);      return tail;  } | get the pointer to the last item of the linked list |
| int GetLength(struct Node \*ptr)  {      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      current = ptr;      int len=1;      while(current->next != NULL){          current = current->next;          len++;      }      return len;  } | count items of a linked list |
|  |  |

### update

|  |  |
| --- | --- |
| int AddValue(struct Node \*ptr, char \*value){      struct Node \*new\_node = (struct Node \*)malloc(sizeof(struct Node));      new\_node->value = value;      new\_node->next = NULL;      struct Node \*tail = GetLastNode(ptr);      tail->next = new\_node;      return ptr;  } | append one items to the end of the linked list |
| struct Node \*InsertNode(struct Node \*ptr, char \*value, int index){      struct Node \* new = (struct Node \*) malloc(sizeof(struct Node));      new->value = value;      // put it on the top      if(index<=0){          new->next = ptr;          ptr=new;      }      // put it in the middle      else{          struct Node \* before = (struct Node \*) malloc(sizeof(struct Node));          struct Node \* current = (struct Node \*) malloc(sizeof(struct Node));          current = ptr;          for(int i=0; i<=index; i++){              if( (i+1)==index ){                  before=current;                  printf("%d:%s\n", i, before->value);              }              else if( i==index ){                  new->next = current;                  before->next = new;                  break;              }              current = current->next;              if(current->next == NULL){                  AddValue(ptr, value);                  break;              }          }      }      return ptr;  } | insert one node by index |
| struct Node \* UpdateValue(struct Node \*ptr, char \*value, int index){      struct Node \* current = (struct Node \*) malloc(sizeof(struct Node));      current = ptr;      if(index < 0){          index = 0;      }      int i=0;      while(1){          if(i==index){              current->value = value;              break;          }          current = current->next;          if(current->next == NULL){              break;          }          i++;      }      return ptr;  } | update one node by index |
| struct Node \*SwitchNode(struct Node \*ptr, int a, int b)  {      if(a>=0 && b>=0 && a!=b)      {          if(a>b)          {              a=a+b;              b=a-b;              a=a-b;          }          struct Node \* a\_switch = (struct Node \*)malloc(sizeof(struct Node));          struct Node \* b\_switch = (struct Node \*)malloc(sizeof(struct Node));          struct Node \* current = (struct Node \*)malloc(sizeof(struct Node));          current = ptr;          int i=0;          while(1)          {              if(i==a){                  a\_switch=current;              }else if (i==b){                  b\_switch = current;                  // switch value                  char \*tmp\_value = a\_switch->value;                  a\_switch->value = b\_switch->value;                  b\_switch->value = tmp\_value;                  break;              }              current = current->next;              if(current->next == NULL)                  break;              i++;          }          free(current);      }      return ptr;  } | switch values between two nodes |
| struct Node \*ChangeValue(struct Node \*ptr, char \*value, int index)  {      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      current = ptr;      for(int i=0; i<=index; i++)      {          if(index==i){              current->value = value;              break;          }          current = current->next;          if(current->next==NULL)              break;      }      return ptr;  } | update value of a node |
|  |  |

### delete

|  |  |
| --- | --- |
| struct Node \* DeleteNodeByValue(struct Node \*ptr, char \* value)  {      struct Node \*before = (struct Node \*)malloc(sizeof(struct Node));      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      before = ptr;      // match the first one      if(strcmp(ptr->value, value) == 0){          before = before->next;          return before;      }      // if match sth in the middle      current = before->next;      while(current != NULL){          // printf("%s\n", current->value);          if(strcmp(current->value, value)==0){              before->next=current->next;              return ptr;          }          before = before->next;          current = current->next;      };      return ptr;  } | delete node by value |
| struct Node \* DeleteNodeByIndex(struct Node \*ptr, int index)  {      struct Node \*before = (struct Node \*)malloc(sizeof(struct Node));      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      before = ptr;      // match the first one      if(index == 0){          before = before->next;          return before;      }      // if match sth in the middle      int i = 1;      current = before->next;      while(index>=1 && current != NULL){          // printf("%s\n", current->value);          if(index==i){              before->next=current->next;              return ptr;          }          before = before->next;          current = current->next;          i++;      };      return ptr;  } | delete a node by index |
| struct Node \* DeleteLastNode(struct Node \*ptr)  {      struct Node \*before = (struct Node \*)malloc(sizeof(struct Node));      struct Node \*current = (struct Node \*)malloc(sizeof(struct Node));      before = ptr;      // only one item      if(ptr->next==NULL){          free(ptr);          return current;      }      current = before->next;      while(1){          if(current->next==NULL){              before->next=NULL;              return ptr;          }          before = before->next;          current = current->next;      };      return ptr;  } | delete the last node |
|  |  |
|  |  |

### transform

|  |  |
| --- | --- |
| struct Node \*ConcatList(struct Node \*ptr, struct Node \*ptr1)  {      struct Node \*current = GetLastNode(ptr);      current->next = ptr1;      return ptr;  } | concatenate two linked list |
| struct Node \*MergeList(struct Node \*ptr, struct Node \*pt1)  {      struct Node \*current=(struct Node \*)malloc(sizeof(struct Node));      current = ptr;      struct Node \*a\_current=(struct Node \*)malloc(sizeof(struct Node));      a\_current = ptr;      struct Node \*a=(struct Node \*)malloc(sizeof(struct Node));      a = ptr;      struct Node \*b\_current=(struct Node \*)malloc(sizeof(struct Node));      b\_current = pt1;      struct Node \*b=(struct Node \*)malloc(sizeof(struct Node));      b = pt1;      int i=0;      while(a->next != NULL && b->next != NULL){          // dock the next position before change          a = a->next;          b = b->next;          // change links          b\_current->next = a\_current->next;          current->next = b\_current;          current=current->next;          current=current->next;          // move to the dock          a\_current = a;          b\_current = b;          i++;      };      // printf("%s,%s\n", a->value, b->value);      // ptr1 is shorter      if(b->next != NULL){          // printf("%s,%s,%s\n", a->value, b\_current->value, current->value);          current->next = b->next;      }      // ptr1 is longer      else{          b\_current->next = a\_current->next;          current->next = b\_current;          current=current->next;          current=current->next;      }      return ptr;  } | merge two linked list by inserting node to each other. |
| struct Node \*ReverseList(struct Node \*ptr)  {      struct Node \*current=(struct Node \*)malloc(sizeof(struct Node));      current = ptr;      struct Node \*new=(struct Node \*)malloc(sizeof(struct Node));      new->value = current->value;      new->next = NULL;      while(current->next != NULL){          current= current->next;          struct Node \*next=(struct Node \*)malloc(sizeof(struct Node));          next->value = current->value;          next->next = new;          new = next;      }      return new;  } | reverse linked list |
| struct Node \*SubList(struct Node \*ptr, int a, int b)  {      struct Node \*current=(struct Node \*)malloc(sizeof(struct Node));      current = ptr;      int i=0;      struct Node \*new=(struct Node \*)malloc(sizeof(struct Node));      do{          if(i>b){              break;          }else if(i==a){              new->value = current->value;              new->next = NULL;          }else if(i>a&&i<=b){              AddValue(new, current->value);          }          current=current->next;          i++;      }while(current->next != NULL);      return new;  } | get sub-list |
|  |  |
|  |  |

# Type: structure

## declare and initialize a structure

Those statements are equivalent.

integer

|  |  |
| --- | --- |
| struct P{          int i;      };      struct P p;      p.i=45;  printf("%i\n", p.i); |  |
| struct P{          int i;      } p;      p.i=45;  printf("%i\n", p.i); |  |
| struct P{          int i;      } p = {45};  printf("%i\n", p.i); |  |
|  |  |
|  |  |

array

|  |  |
| --- | --- |
| struct P{          char name[20];      } p;  **strcpy(p.name, "good");**      printf("%s\n", p.name); | Note: should use strcpy(). The statement p.name="good", is invalid |
| struct P{          char name[20];      } **p = {"good"};**      printf("%s\n", p.name); | equal to the above |
| struct P{          char name[20];      } p;  **strcpy(p.name, "good");**      printf("%s\n", p.name); | equal to the above |
| struct P{          char \*name;      } p;  **p.name = "good";**      printf("%s\n", p.name); | compared with the above  Note: strcpy() is not working  Note: p->name="good" is wrong |
| struct P{          char arr[3];      } p={{1,2,3}};      printf("%d\n", p.arr[2]); | output: 3  here, p={1,2,3}; removing inner {} is ok. |
| struct P{          char arr[3];          int i;      } p={{1,2,3},4};      printf("%d, %d\n", p.arr[2], p.i); | output: 3,4  p={1,2,3,4}; is ok, but not suggested |
| struct P{          char arr[3];          int i;      } p={{1,2},4};      p.arr[2]=p.arr[0]+p.arr[1];      printf("%d, %d\n", p.arr[2], p.i); | output: 3, 4  it is ok to initialize array partially |
| struct P{          char arr[3];          int i;      } p={{},4};      printf("%d, %d\n", p.arr[2], p.i); | output: 0,4 |
|  |  |

|  |  |
| --- | --- |
|  |  |
| struct STUDENT{  char gender;  char \*name;  int age;  float weight;  } s;  int STUDENT=4;  s.gender='M';  s.name="Cary Howard";  s.age=45;  s.weight=123.6;  printf("%d,%f, %c, %s\n", s.age,s.weight, s.gender,s.name); | define a structure known as STUDENT.  declare a variable 's', of which data type is STUDENT. |
| struct STUDENT{  int age;  float weight;  };  struct STUDENT s;  s.age=45;  s.weight=123.6; | Another approach to define structure |
| struct STUDENT{  int age;  float weight;  };  struct STUDENT s[100];  s[0].age=45;  s[0].weight=123.6;  printf("%d,%f\n", s[0].age,s[0].weight); | define a structure array with the size of 100. |
| struct STUDENT{  char \*first\_name;  char last\_name[10];  };  struct STUDENT s;  s.first\_name = "Howard";  strcpy(s.last\_name, "Hope");  printf("%s,%s\n", s.first\_name, s.last\_name); | two patterns for defining string by pointer and char array. |
| struct STUDENT{  char \*name;  int age;  };  struct STUDENT s= { "Howard Hope", 45};  printf("%s,%d\n", s.name, s.age); | Another pattern to pass value into structure variable |
|  |  |
|  |  |
|  |  |
| struct {  int f1;  } str1;  struct {  char f1;  } str2;  str1.f1 = 32;  str2.f1 = str1.f1; | Two structures can contain fields with the same names – the snippet in the editor is correct. |
| struct STR {  int field;  } Structure;  int STR;  Structure.field = 0;  STR = 1; | name of struct type and integer type are identical. Compiling is ok. but try to avoid it. |
| struct Q{          char S[3];      };      struct S{          char S[3];          struct Q Q;      };  **struct S S;**      printf("%d\n", sizeof(struct S));      printf("%d\n", sizeof(S)); | output: 6 6  Note: "struct S S;" will not cause compiling error. but that approach is not suggested. |

## pointer and structure

pattern: "<pointer>-><field>" or "\*(<pointer>.<field.)"

|  |  |
| --- | --- |
|  |  |
| struct STUDENT{  char \*name;  int age  };  **struct STUDENT \*p;**  **p= (struct STUDENT \*) malloc(sizeof(struct STUDENT));**  **struct STUDENT var ={"good", 45};**  **p=&var;**  **printf("%s,%d\n", p->name, p->age);** | pointer to structure |
| struct STUDENT{  char \*name;  int age  };  **struct STUDENT \*p;**  **p= (struct STUDENT \*) malloc(sizeof(struct STUDENT));**  **(\*p).age=10;**  **(\*p).name="good";**  printf("%s,%d\n", p->name, p->age); | Note: Either \*p.age=10 or \*(p.age)=10 is wrong.  Note: printf("%s,%d\n", \*(p.name), \*(p.age) ); is wrong. |
|  |  |

## union

for a union, all members share the same memory. So only access the value of the latest item in one union.

two int types

|  |  |
| --- | --- |
| C code | output |
| union SIZE{          int a;          int b;    } s = {3, 6};     printf("%d, %d\n", s.a, s.b); | output: 3,3  the type is int, so int a and b share the same memory |
| union {          int i1;          int i2;  **} myVar = {.i2 = 100};**      printf("%d %d",myVar.i1, myVar.i2); | output: 100 100  Since fields/members of union share same memory, both i1 and i2 refer to same location. Also, since both i1 and i2 are of same type, initializing one would initialize the other as well implicitly. |
| union {          int i1;          int i2;  **} myVar = {100};**      printf("%d %d",myVar.i1, myVar.i2); | output: 100 100 |
| union {          int i1;          int i2;  **} myVar = {100, 200, 300};**      printf("%d %d",myVar.i1, myVar.i2); | output: 100 100  give extra values will not cause errors. but only the first one is assigned. |
| union {          int i1, i2;      } myVar = {100};      myVar.i2=200;      printf("%d %d",myVar.i1, myVar.i2); | output: 200 200 |
| union un{          int x;          char y;      };      union un u1={10};      union un u2={'a'};      union un u3={20, 'a'};      printf("%d%c%d",u1.x, u2.y, u3.y); | output:10a20 |

int and float type

|  |  |
| --- | --- |
| C code | output |
| union SIZE{  int a;  float b;  } s = {3.3, 6};  printf("%d, %f\n", s.a, s.b); | output: 3, 0.000000  The type is int. only 3.3 is assigned to a. |
| union SIZE{  int a;  float b;  } s;  s.a=5;  s.b=3.66;  printf("%d, %f\n", s.a, s.b); | output: unpredictable value, 3.660000  variable a and b share the same address. Here value of b cover that of a.  the type is int and then changed to float |
|  |  |
|  |  |
| union SIZE{  int a;  float a;  } s;  s.a=5;  s.a=3.66;  printf("%d\n", s.a); | error: duplicate variable |
|  |  |

int type and array

|  |  |
| --- | --- |
| C code | output |
| union a{          int i;          char ch[3];      };      union a u = {512};      printf("%d, <%s>\n", u.i, u.ch);      strcpy(u.ch, "AB");      printf("%d, <%s>\n", u.i, u.ch); | output:  512, <>  unknown, <AB> |
| union test {          int x;          char arr[4];          int y;      };      union test t;      t.x=0;      t.arr[1]='G';      printf("%s", t.arr); | output: nothing printed.  the variable x and array arr[4] share the same address. When we do “t.arr[1] = ‘G'”, arr[] becomes “\0G\0\0”. When we print a string using “%s”, the printf function starts from the first character and keeps printing till it finds a \0. Since the first character itself is \0, nothing is printed. |

## nested struct

basic pattern

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| struct BIRTH{  int year;  int month;  int day;  };  struct STUDENT{  char name[20];  int age;  struct BIRTH birth;  };  struct STUDENT s = {"good", 45, {2000,12,2}};  printf("%s: %d\n", s.name, s.age);  s.birth.year=2010;  printf("%d-%d-%d\n", s.birth.year, s.birth.month, s.birth.day); | output:  good: 45  2010-12-2  nested structure |
| struct S{          int var;          struct S \*str;      };      struct S s[]={{8,NULL},{4, &s[0]}, 2, &s[1]};      printf("%d", s[2].str->str->var); | output: 8 |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| struct house{  int number;  struct house \*previous;  struct house \*next;  };  struct house h[4];  int n[]={1,3,5,7};  for(int i=0; i<4; i++){  h[i].number = n[i];  if(i==0){  h[i].previous = NULL;  }else{  h[i].previous = &h[i-1];  }  if(i==3){  h[i].next = NULL;  }else{  h[i].next = &h[i+1];  }  }  printf("###current=%d, next=%d\n",  h[0].number, h[0].next->number);  printf("###current=%d, previous=%d, next=%d\n",  h[1].number,h[1].previous->number, h[1].next->number);  printf("###current=%d, previous=%d\n",  h[3].number,h[3].previous->number); | output:  current=1, next=3  current=3, previous=1, next=5  current=7, previous=5 |
| struct element{  int value;  struct element \*next;  };    int values[10] = { 2, 4, 5, 6, 7, 8, 9, 1, 3, 0};  struct element e[10];  for(int i=9; i>=0;i--){  e[i].value = values[i];  if(i==9){  e[i].next = NULL;  }else{  e[i].next = &e[i+1];  }    }    void func(struct element \*p, int t){  printf("%d\n", p->value);  if(p->value!=t){  func(p->next, t);  }  }  puts("First 5 values");  func(&e[0],7);  puts("First 9 values");  func(&e[0],9);  freeI; | output:  First 5 values  2  4  5  6  7  First 9 values  2  4  5  6  7  8  9 |
|  |  |
|  |  |
|  |  |

## complicated struct

|  |  |
| --- | --- |
| struct ST{          int id;          char \*name;          char \*date\_birth;          char \*gender;      } person[3];      char names[3][100]={          "1,John Adam,2010-10-01,M",          "2,Smith Howard,2012-08-12,M",          "3,Mary Carie,2000-01-22,F"      };      for(int i=0; i<3; i++){          char \*token = strtok(names[i], ",");          person[i].id= atoi(token);          person[i].name=strtok(NULL, ",");          person[i].date\_birth=strtok(NULL, ",");          person[i].gender=strtok(NULL, ",");          printf("%d, %s, %s, %s\n", person[i].id, person[i].name,          person[i].date\_birth, person[i].gender);      } | Note:  1.char names[3][100]  2.char \*name in struct  3. strtok split string then strtok(NULL, ",") |
| struct S1{      int p1,p2;  };  struct S2{      int p1;      struct S1 s1;      int p2;  };  int main(void){      int s=0;      struct S2 s2 = {1, 2, 3, 4};  **struct S2 \*p;**  **p = (struct S2 \*) malloc(sizeof(struct S2));**  **\*p = s2;**  **s2.p1 = 0;**      printf("%d,%d\n", p->p1, s2.p1);      s=p->p1+s2.p1+p->p2+p->s1.p2;      free(p);      printf("%d\n", s);      return 0;  } | output: 8  "\*p=s2;" copy values of s2 to pointer.  if "p=&s2;", assign address to pointer  output=7 |
| struct {  int a[2], b;} arr[] = {[0].a = {1}, [1].a = {2}, [0].b = 1, [1].b = 2};        printf("%d %d %d and",arr[0].a[0],arr[0].a[1],arr[0].b);      printf("%d %d %d\n",arr[1].a[0],arr[1].a[1],arr[1].b); | output: 1 0 1 and2 0 2  Note: struct {int a[2], b;} arr[] = {{{1},1}, {{2},2}}; |
|  |  |
|  |  |

## sizeof()

The size is the size of pointer==4 times by number of items defined in struct.

|  |  |  |
| --- | --- | --- |
|  | output |  |
| struct P{          char \*name;      } p = {"goodbye"};      printf("%d\n", sizeof(p)); | 4 | pointer |
| struct P{          char \*name;      } p = {"goodbye"};      printf("%d\n", sizeof(p.name)); | 4 | pointer |
| struct P{          float x;          char \*name;      } p = {2.34, "goodbye"};      printf("%d\n", sizeof(p)); | 8 |  |
| struct P{          float x;          int y;          char \*name;          struct P \*next;      };      struct P p2 = {2.34, 20, "goodbye"};      struct P p1 = {2.34, 20, "goodbye"};      p1.next = &p2;      printf("%d\n", sizeof(p1)); | 16 |  |
|  |  |  |
|  |  |  |
|  |  |  |

## copy struct

|  |  |
| --- | --- |
| struct P{          int i;      };      struct P p1={1};      struct P p2 = p1;      printf("%d%d", p1.i, p2.i);      p2.i=2;      printf("%d%d\n", p1.i, p2.i); | output: 1112  hard copy struct |
|  |  |
|  |  |
|  |  |

# pointer

## define

|  |  |
| --- | --- |
| int I;  char I; | integer variable  sizeof(i)=4 |
| int a[10]; | integer array known as 'a' with 10 integers |
| int \*p, i=2;  p=&I; | reference variable known as 'p', which contain address of an integer variable  sizeof(p)=4 |
| int \*p;  int a[]={1,2,3,4,5,6,7,8,9,10};  p=&a[2]; | reference variable known as 'p', which contain address of a[2], of which the value is 3. |
| int \*p[10]; | an array with 10 integer pointers  sizeof(p)=4\*10 |
| int (\*p)[10]  int a[]={1,2,3,4,5,6,7,8,9,10};  p=&a; | reference variable known as 'p', which contain address of an array with 10 integers.  sizeof(p)=4 |
| void \*pVoid;      pVoid = (void\*)0;      printf("%lu",sizeof(pVoid)); | (void \*)0 is basically NULL pointer which is used for many purposes in C. Please note that no matter what is the type of pointer, each pointer holds some address and the size of every pointer is equal to sizeof(int). |
|  |  |
| int a[]; | illegal |

## memory allocation

Wild pointers are different from pointers i.e. they also store the memory addresses but point the unallocated memory or data value which has been deallocated. Such pointers are known as wild pointers.

malloc() and free()

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

## get value and move

int a[5]={1,3,5,7,9};

int \*p, i, j;

p=&a;

precedence: right to left side

|  |  |
| --- | --- |
| **i=\*p++;** | move pointer p from a[0] to a[1]. Assign the value a[0]=1 to i  **equivalent: p=p+1, p+=1, p++, \*(p++)**  **i=1, \*p=3** |
| (\*p)++; | access the value of a[0], and then add 1 to the a[0]  \*p=2, a[10]={2,2,3,4,5,6,7,8,9,10}; |
| **i=\*++p;** | move pointer p from a[0] to a[1], assign value of a[1] to i  **equivalent: \*(++p)**  i=3, \*p=3 |
| **i=++\*p;** | access value of a[0], and add 1 to a[0], assign a[0] to i  **equivalent: ++(\*p)**  i=2, \*p=2, a[10]={2,2,3,4,5,6,7,8,9,10} |
| i=\*(p+3); | Access value of a[3] and assign the value of a[3]=7 to i.  Don’t move pointer  i=7, \*p=1 |
| i=\*p+3; | Access value of a[0], add by 3 and assign to i=a[0]+3=1+3=4  i=4, \*p=1 |
| p=&a[1];  i=\*(p-1)+\*p+\*(P+1) | pointer is to a[1]. Access values of a[0], a[1], and a[2].  Don’t' move pointer. |
| int t[3]={3,2,1}, \*ptr=t+1;      (\*(ptr+1))++;      \*ptr++;      printf("%d%d",t[1],[2]); | output: 22 |

if \* and ++/-- precede pointer variable, the precedence are the same, and right-to-side. But if ++/-- follows the pointer variable. the precedence is the last.

## arithmetic of pointers

|  |  |
| --- | --- |
| int t[4]={8,4,2,1};      int \*p1= t+2, \*p2=p1-1;      p1++;      printf("%d", \*p1-t[p1-p2]); | output =-1  8 4 2 1  t p2 p1  so p1-p2=2, \*p1-t[p1-p2]=1-t[2]=1-2=-1 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## void pointer

void pointer indicate the absence of type. It can hold address of any type

void pointer can't be dereferenced

|  |  |
| --- | --- |
| int a=10;  char b='A';  **void \*p =&a**  p=&b; | Advantage: void pointer can be reusable.  void pointer holds address of integer variable 'a' and then char 'b'. |
| int \*x =malloc(sizeof(int\*\*3); | malloc() return void \* type, which allows to allocate memory of any data type  Note: c++ can't allow it. |
| int a=10;  void \*p=&a;  **printf("%d", \*(int \*)p);** | declare data type which the pointer references. |
| int a[2] = {1, 2};  void \*ptr = &a;  printf("%d", \*(int \*)ptr+1); | arithmetic of void pointer  output: 2 |
| int a[2] = {1, 2};  void \*ptr = &a;  printf("%d", sizeof(ptr)); | The size of the void pointer in C is the same as the size of the pointer of character type.  output: 4 |
| int a[2] = {1, 2};  void \*ptr = &a;  ptr = ptr + sizeof(int);  printf("%d", \*(int \*)ptr); | ??? output 2 |
| int a=10;  void \*p=&a;  printf("%d", \*p); | Compile Error: void \* can't be dereferenced like other pointers namely int pointer. |

## NULL pointer

In UNIX, the first byte of any program is 0.

|  |  |
| --- | --- |
|  |  |
| int \*p=NULL;  printf("%d", p); | don’t assign pointer with address. default value is 0. |
| char \*str = NULL;  print("%s\n", str); | output: (null)  The output is determined in newer compilers. but some old system may return unpredictable values. |
| void \*pVoid;      pVoid = (void\*)0;      printf("%d",sizeof(pVoid)); | (void\*)0 is NULL pointer |
|  |  |

## pointer in array

By default, the pointer denotes the address of the first element of an array

|  |  |
| --- | --- |
| int a[]={1,2};      int \*p=a;      printf("%d\n", \*p); | output: 1 |
| char a[]="ABC";      char \*p=a;      printf("%c\n", \*p); | output: A |
| void func(char \*p){          printf("%c", \*p);      }      char a[]="ABC";      char \*p=a;      func(p);      func(p+1); | output: AB |
| void func(char \*s, int i){          \*(s+i)='\0';      }      char a[]={'a', 'b', 'c', 'd'};  **func(&a[1],1);**  **//wrong: func(a[1],1);**      printf("%d", strlen(a)); | output:2  Note: "func(a,1);" is ok. output is 1. But "func(a[1],1);" will cause compiling failure. |
| int func(char t[]){          return t[1]-t[0];      }      int i=2;      I -= func("ABDGK"+1);      printf("%d", i); | output:0 |
| int \*t1 = 1 + (int \*)malloc(sizeof(int)\*sizeof(int));      \*t1 = 4;      t1[-1]=2;      printf("%d\n",\*t1); | output: 4  t1->1 |

### pointer to integer

|  |  |
| --- | --- |
| int a[][3]={1,2,3,4,5,6};  **int (\*ptr)[3]=a;**      printf("%d%d", (\*ptr)[1], (\*ptr)[2]);      ++ptr;      printf("%d%d\n", (\*ptr)[1], (\*ptr)[2]); | **"int (\*ptr)[3]=a;" means the pointer \*ptr address int array a[0]={1,2,3}. ++ptr means move the pointer to the a[1][0]** |
| int x = 356;      char \*p = (char\*) &x;      printf("%d", p[0]); | output:100  356=0b10110010, position 0 is 110010=2^6+2^5+2^2=100  address0: 0110010  address1: 0000001  address3: 0000000  address3: 0000000 |
| #include <stdio.h>  struct A {  int a;  } A1;  struct B {  int b;  } B1;  int main(void)  {  A1.a = 10;  B1.b = 100;  char\* x = (char\*)&A1;  char\* y = (char\*)&B1;  y = (char\*)0x100;  x[0] = y[1];  printf("%d\n", A1.a);  printf("%d", B1.b);  } | segment error |
|  |  |
|  |  |
|  |  |
|  |  |

# Flow control

## for loop

The syntax of a for loop in C programming language is

for ( init; condition; increment ) {

statement(s);

}

Here is the flow of control in a 'for' loop:

1. **The init step is executed first, and only once.** This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

2. **Next, the condition is evaluated. If it is true, the body of the loop is executed.** If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.

3. After the body of the 'for' loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

|  |  |
| --- | --- |
|  |  |
| int I, j=0;  for(i=0;i<4;i++){  j++;  } | standard pattern  i=4, j=4 |
| int I, j=0;  for(i=0;i<4;i++)  j++; | the same as above. remove curly bracket {} if only one statement in for-loop |
| int i=10, j=0;  for(i=0;i<4;i++){  j++;  } | initialize i within for()  i=4,j=4 |
| int i, j=0;  for( ; ; ){  j++;  } | infinite loop |
| int i=-1, j=3;  for(j>0;j;j--)  i\*=2;  printf("integer: %d\n", i+j); | There are 3 expression: "j>0","j","j--"  execute "j>0" at one time, then execute "j--" and then "j"  export: -8 |
| i=-3, j=0;  **for(i++;i++;i++){**  **j--;**  **}**  printf("%d, %d\n", i,j); | export: i=1, j=-1  i=-3,i=-2->i=-1,j=-1,i=0->i=1 |
| i=-3, j=0;  for(i++;++i;i++){  j--;  }  printf("%d, %d\n", i,j); | dead loop |
| int i=-1, j=1;      for(i++;i++;i++)          j++;      printf("%d\n",i+j); | output:2 |
| int i = 5,j = 4;      for(i--;i--;i--)          j--;      printf("%d\n", i + j); | output:1  i=5,j=4  i=4,i=3,j=3,i=2  i=1,j=2,i=0  i=-1 -> i+j=1 |
| int i, t[5];  for(i=0;i<5;i++)  t[i]=2\*i;  i=0;  for(i=0;i<5;i++)  i +=t[i];  printf("%d",i); | output:13  the loop variable I is both in conditions and inner statement. |
| int i, s=0, t[]={16,8,4,2,1,0};      for(i=5; t[i]>2;i++){          s+=t[i];      }      printf("%d\n", s); | output: 0  before executing body loop, the initialization i=5 and condition t[i]>2 should be executed. |
| int i=1, j;  **for(j=0;j;j--)**          i\*=2;      printf("%d\n",i+j); | output: 1  because j=0, body of loop will not be executed. |
| int i=1;      for(int j;j<10;j++)          printf("%d\n",i); | output: no output  "int j" is true, but "j<10" is false because j is not initialized. |

### use pointer in for loop

|  |  |
| --- | --- |
| char \*p = (char \*)malloc(11\*sizeof(char));  for(int i=0;i<10;i++){  \*p='A';  printf("%c,", \*p);  p++;  }  \*p='\0';  p-=10;  printf("==%s\n",p); | output: A,A,A,A,A,A,A,A,A,A,==AAAAAAAAAA |
| char s[] = "ABC";  char \*p=s;  **for(;\*p;p++){**  printf("%c,", \*p);  }  p-=strlen(s);  printf("==%s\n",p); | output: A,B,C,==ABC  the same as the above |
| char \*mystrcat(char \*destination, char \*source) {  char \*res;  for(res = destination; \*destination++; ) ;  for(--destination; (\*destination++ = \*source++); ) ;  return res;  }  char s1[]="abc";  char s2[]="XYZ";  char \*p = mystrcat(s1, s2);  printf("%s", p); | output: abcXYZ  verify understanding of pointer. |
|  |  |
|  |  |

### unusual approach

|  |  |
| --- | --- |
| int t[]={8,4,3,2,1},i;      for(i=t[4];i<t[0];i++)          t[i-1]= -t[3];      printf("%d",i); | output: 2  The condition expression is changeable though in common patterns that is fixed. |
| for(float val=-10.0; val<100.0; val=-val\*2){          if(val<0 && -val>=40)              break;          cout <<  "\*";      } | outpu: \*\*  Note: break will breaks whole for loop |
| char arr[5]={'a','b','c','d','e'};      for(int i=1;i<5;i++){          cout <<  "\*";          if(arr[i]-arr[i-1]%2)              continue;          cout <<  "\*";      } | output: \*\*\*\*  Note: continue just breaks the current cycle and go into the next cycle. |
| int i=1;  **for(;i>180;i\*=2);**      printf("%d", i); | output: 1  simplify for statement |
| int main(int argc, char \*argv[]){      int a=0, i=2;      for(i>>1;i<40;i<<1)          a++;      printf("%d,%d", i,a);      return 0;  } | output: infinite loop  i>>1 and i<<1 would be executed, but value i would not be updated.  wrong use of expression |
| int main(int argc, char \*argv[]){      int a=0, i=2;  **for(i>>1;i<40;i=i<<1)**          a++;      printf("%d,%d", i,a);      return 0;  } | output: 64,5 |
|  |  |
|  |  |

multiple conditions

|  |  |
| --- | --- |
| for(int i=0, j=0; i<10, j<10; i++){  j += i;  printf("%d:%d, ", i, j);  } | output: 0:0, 1:1, 2:3, 3:6, 4:10,  Here, "i<10,j<10" is equivalent to " i<10&&j<10"  multiple conditions |
| int i,j;      for(i=0,j=1; j<2,i<4; i++,j++);      printf("%d,%d", i, j); | output:4,5  **for checking end of for loop: "i<2,j<4" means i<2 and j<4. If i=2 and j=3, loop should be executed.** |

## switch

1. branches are scanned in the same order as they are defined.

3. break statement should be involved for each case statement or default statement. If not, execution will continue to break statement or the end of switch loop.

|  |  |
| --- | --- |
| int i=3;  switch(i){  case 1: printf("January\n");break;  case 2: printf("February\n");break;  case 3: printf("March\n");  default: printf("wrong\n");  }  printf("%d\n", i); | output:  March wrong |
| i=2,j=0;  switch(i){  case 0: j++;break;  **case 2: j++;**  **case 4: j++;break;**  default: j--;  }  printf("i=%d,j=%d\n", I,j); | Here, some break statements are missing. Execute case 2, and then case 4.  output: i=2,j=2 |
|  |  |
|  |  |
| int a=2;      switch(a<<a){          case 8: a++;          case 4: a++;          case 2: break;          case 1: a--;      }      cout<<a; | output: 4 |
|  |  |
|  |  |

4. Execute default if not case conditions. The default statement is not required but it is suggested to. For example: default:break;

|  |  |
| --- | --- |
| i=6, j=0;  switch(i){  case 0: j++;break;  **default: j--;**  **case 2: j++;**  **case 4: j++;break;**  case 1: j++;  }  printf("i=%d,j=%d\n", I,j); | output: i=6, j=1  Here, default is in the middle.  Execute default, case 2, and case 4 |

2. case statement should be with value (int, char, or enum type). other type such as certain expression or float type will cause compiling error

|  |  |
| --- | --- |
| char a='A', c;      switch (a){          case 'A': c=a;break;          case 'a': c="A";break;          default: c='N';      }      printf("%c", c); | output: A  char type denoted as integer |
| char a="A", c;      switch (a){          case "A": c=a;break;          case "a": c="A";break;          default: c="N";      }      printf("%c", c); | compiling error: "" is wrong  string type is illegal |
| enum a {A, a, no};      char c;      switch (a){          case A: c='A';break;          case a: c='A';break;          case no: c='N';      }      printf("%c", c); | output: A  enum type is acceptable. |
| int c = 'a';      switch(c){          case '2': printf("ok");          case '1': printf("eror");          default: break;      } | The code is correct though pattern is wierd. |
|  |  |

6. In switch scope, only keyword "case" and "default" will be executed. Other statements will be ignored.

|  |  |
| --- | --- |
| int a=1;  switch(a){  puts("output:");  break;  case 1: puts("A");  case 2: puts("B");  } | output: A  B  Here, puts and breaks statements are not executed. switch loop only recognize case and default statements |

## if-elseif-else

basic pattern

|  |  |
| --- | --- |
| int a=2;      if(a==0){          printf("a is zero");      }else if(a==1){          printf("a is 1");      }else{          printf("a is %d\n",a);      } | output: a is 2  standard if statements |
| int a=2;      if(a==0){          printf("a is zero");      }else if(a>=1){          printf("a is >=1");      }else if(a>=2){          printf("a is >=2");      }      else{          printf("a is %d\n",a);      } | output: a is >=1  no compiling errors. but the statements is not suggested because conditions are overlapped. |
| int a;      if(a)          printf("integer is declared"); | output: integer is declared  if statements could be simplified by removing curly brackets if only one statement inside. |
| int i=1;      if(i=0)          i=2;      else          i=3;      printf("%d\n", i); | output: 3  The expression "i=0" is false because i is 0 |
| int i=1;      if(i=10)          i=2;      else          i=3;      printf("%d\n", i); | output: 2  The expression "i=10" is true |
| int i;      if(i)          i=2;      else          i=3;      printf("%d\n", i); | output: 2  The expression "i" is true |
| int i;      if(NULL)          i=2;      else          i=3;      printf("%d\n", i); | output: 3  The expression "NULL" is 0 |
| if(total<0)  printf("good"), total=0; | simplify the statements using comma. comma |

|  |  |
| --- | --- |
| if(int i=1)          printf("%d\n", i); | compiling error  int i should be declared before if statements |
| for(int j=0;j<10;j++)          printf("%d\n",j); | that is correct. compared with the above |
|  |  |
|  |  |

### construct () ? () : ()

usual approach:

<variable> = (<expression>) ? (<expression>) : (<expression>)

|  |  |
| --- | --- |
| int a = 4, b = 5;      int i = a > 4 ? a : b;      printf("%d\n", i); | output: 5  standard approach |
| int func(int a, int b){          return a > 4 ? a : b;      }      int a = 4, b = 5;      int i = func(a,b);      printf("%d\n", i); | output: 5  apply it in the return statement |
| int a=3, b=4, c=5;      int max = a>b ? a>c ? a: c : b>c ? b : c;      printf("%d\n", max); | nested constructor |
| int a=3, b=4, c;  **int i = a>b ? (c=50) : (c=40);**      printf("%d, %d\n", i, c); | i=c=40  Note: 1. assignment statements are always return in paranthesis in the case of conditional operator. 2. variable i and c should be identical. |
| int a=1,b=2, c, d;  **a>b ? (c=10,d=20) : (c=1,d=2);**      printf("%d%d", c,d); | output:12  Note: should encompassed by parentheses, or compiling error |
| int a=1,b=2, c, d;  **a,b = a>b ? (c=10,d=20) : (c=1,d=2);**      printf("%d%d", a,b); | output=12  the same as the above |
| int a=4,b=2;      a>b ? a-- : a++;      printf("%d", a); | output: 3 |
| int a=4,b=2;      printf("%d", a>b ? a-- : a++); | output: 4  compared with the above |
| printf("%d\n", 4 ? : 8); | output: 4 |
| printf("%d\n", 4 ? 3 : 8); | output: 3 |
| cout << 3 ? 2;  cout << 3 ? 2 :; | compiling error. |

## while

For each cycle, the expression defined in the while loop should be executed once.

do{

……

}while(<expression>);

while(<expression>){

……

}

|  |  |
| --- | --- |
| int i = 16,j = 8;      do{          i /= 2;          j = i/2;      }while(j > 0);      printf("%d\n", i + j); | output:1  i=16,j=16/2=8  i=8, j=8/2=4  i=4,j=4/2=2  i=2,j=2/2=1  i=1,j=1/2=0 |
| int t=4;      while(--t)          printf("\*"); | output:\*\*\*  **Note: the expression would be executed in each cycle** |
|  |  |
|  |  |
|  |  |

# Function

## definition, declaration, calling

1. Follow the steps as definition, declaration and calling.

2. Match: if arguments are defined, data type and argument when calling should match those in definition. If return is defined, calling should match with it.

3. pass value and pointer arguments are difference.

### define a function

|  |  |
| --- | --- |
| void func1(void){  printf("no arugments\n");  } | no arguments |
| void func2(int a, char b, char c[]){  printf("a=%d, b=%c, c=%s\n",a,b, c);  } | pass multiple arguments |
| int main(void){  func()  return 0;  }  void func(void){  } | position of function doesn't matter |
|  |  |

### call a function

|  |  |
| --- | --- |
| main(){      //declaration      int i=1, f(int);      //calling      i=f(i);      printf("%d", i);      return 0;  }  // definition  int f(int i){      return ++i;  } |  |
| int f(int i){      return ++i;  }  main(){      int i=1;      i=f(i);      printf("%d", i);      return 0;  } | same as above, but it is ok to remove declaration of functions |
|  |  |
|  |  |
|  |  |
|  |  |

### main()

The patterns below are equal.

|  |
| --- |
| void main(){  …..  } |
| void main(void){  …..  } |
| int main(){  ……  return 1;  } |
| int main(int argc, char \*\*argv){  ……  return 1;  } |
| int main(int argc, char \*argv[]){  ……  return 1;  } |
|  |

|  |  |
| --- | --- |
| #include <stdio.h>  int main(int argc, char \*argv[]) {  int i;  for(i = 0; i < argc; i++)  printf("%s\n", argv[i]);  return 0;  } | F:\C> ./function\_main.exe a b c  F:\C\function\_main.exe  a  b  c |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  int main(int argc, char \*argv[]){    if(argc==2){  float x = atof(argv[1]);  float square = x\*x;  printf("square of %.2f is %.2f", x, square);  }else{  puts("Usage: ./cal\_square.exe <length>");  }  return 0;  } | the function atof(), defined in stdlib.h convert string type of arguments into float type |
| int x=5;  int main(void) {      cout << "x";      if(x-- > 0) main();      return 0;  } | output: xxxxxx |
|  |  |
|  |  |

### variable scope

For a certain function:

In C, use curly braces {} to define the scope of variable. A pair of {} is a block, which could be function, for loop, while loop or if statement, or even blare block.

|  |  |
| --- | --- |
| main(){      int i=1;      {          int i=2;          ++i;          printf("%d", i);      }      printf("%d", i);      return 0;  } | output: 31 |
| main(){      int i=1;      {          ++i;          printf("%d", i);      }      printf("%d", i);      return 0;  } | output:22 |
|  |  |
|  |  |

If variable is defined the parent outer scope of this function, and this function doesn't define it, the calling in this function is acceptable.

variable is different local variable in different scope even if their names are identical

|  |  |
| --- | --- |
|  |  |
| int a=1;  int main(void){      func(a);      printf("main(): a=%d\n",a);      return 0;  }  void func(int data){      printf("func 1(): a=%d\n", a);      a=10;  } | output:  a=1  a=10  a is global variable, could be used by main() and func(). |
| int main(void){      int a=1;      void func(int data){          printf("a=%d\n", a);          a=10;      }      func(a);      printf("a=%d\n",a);      return 0;  } | output:  a=1  a=10  a is local variable defined in main(). It can be used in the function, which is defined in main(), though variable a is not defined in func(). |
| int main(void)  {      int a=1;      func(a);      printf("main(): a=%d\n",a);      return 0;  }  void func(int data){      printf("func 1(): a=%d\n", a);      a=10;  } | compiling error.  variable a is local variable defined in main(). It can't be used in func() if a isn't defined. |
| int a=10;  void func(int data){  int a=2;  printf("func4(): a=%d\n", a);  a=20;  }  func(a); printf("a=%d\n\n",a); | output:  a=2  a=10  int a in func() is local variable. int a=10 is global variable. |
| void func(int a){  a++;  }  a=10;  func(a);  printf("func3, a=%d\n",a); | output: a=10  In func(), a is defined as local variable, and a is also defined as global variable though both of them share the identical name. a++ in func() can't add global a. a |
| void function(int param) {  printf("value %d\n", param);  param++;  }  int main(void) {  int param = 111;  function(param);  printf("variable %d\n", param);  return 0;  } | output:  value 111  variable 111 |
| void func(){  printf("func=%d\n", b);  }  void main(void){  int b=10;  func();  } | compiling error: variable b is defined in main() rather than func() |
| struct S{      int S[3];  };  void f(struct S S){      S.S[0] = S.S[1] + S.S[2] -4;  }  int main(void){      struct S S = {{1,4,2}};      f(S);      printf("%d\n", S.S[1]\*S.S[0]);      return 0;  } | output: 4  The struct S in f() is local. The struct S initialized in main() will not be changed by f(). |
|  |  |

|  |  |
| --- | --- |
| main(){      int x = 1;      {          int x = 10;      }      printf("%d", x);  } | output: 1  the use of curly braces will define the scope of variables. |
| int f(){      double b=1234567890;      return b;  }  int main(){      int i=1;      int b = f(i);    printf("%d", b);      return 0;  } | output: 1234567890 |
| double f(){      int b=1234567890;      return b;  }  int main(){      int i=1;      double b = f(i);    printf("%d", b);      return 0;  } | output: unknown value -1266679808 |
|  |  |
|  |  |

### unusual functions

|  |  |
| --- | --- |
| void f(int i){     return i; }  int main(){      int i=1;      int b = f(i);      return 0;  } | compiling error  void f() means f() should not return values. |
| void f(int i){      printf("%d", i);      return i;  }  int main(){      int i=1;      f(i);      return 0;  } | execution is ok. but involving return statement is not suggested. |
| int f(int i){      int b;      return b=10;  }  int main(){      int i=1;      int b = f(i);    printf("%d", b);      return 0;  } | execution is ok. but "return b=10;" is not suggested.  b=10; return b; |
| int \*f(int i){      int b=10;      return p;  }  int main(){      int i=1;      int \*b = f(i);     printf("%d", \*b);      return 0;  } | compiling error.  should return pointer rather than value.  correct: return &p; |
| **double f(int i){    int b=10;    return b;}**  int main(){      int i=1;  **int b = f(i);**    printf("%d", b);      return 0;  } | output: 10  type casting between int and double. That approach is ok. double is forced resigned to int. |
| int f(int i){    double b=10;    return b;}  int main(){      int i=1;      float b = f(i);    printf("%f", b);      return 0;  } | ok.  type casting. |
|  |  |
|  |  |

## data type

### integer in function

pass int by value

pass int by pointer

return int

|  |  |
| --- | --- |
| int f(void){  }  int i;  i=f();  printf("integer: %d\n", i); | **No return statement: undefined behavior**  return unpredictable value in different system of integer overflow. |
| int f(void){  return 'c';  }  int I;  i=f();  printf("integer: %d\n", i); | output: 99  return integer |
| int f(void){  return "ABC";  }  int I;  i=f();  printf("integer: %d\n", i); | No compiling error but return unpredictable value |
|  |  |
| void f(void){  }  int I;  i=f(); | compiling error |
|  |  |
| #include <stdio.h>  void unction(int \*ptr) {  \*ptr = \*ptr + 100;  }  int main(void) {  int I = 100;  int \*p = &I;  printf("I = %d\n", i);  unction(p);  printf("I = %d\n", i);  return 0;  } | output:  i=100  i=200  pass integer pointer. In some cases, return statements are not needed.  add value stored in \*p; |
| void incr(int \*value) {  **(\*value)++;**  }  int main(void) {  int var = 100;  incr(&var);  printf("var = %d\n", var);  return 0;  } | output: var=101  add value stored in pointer \*p  Note: same as ++\*value or ++(\*value) |
| void incr(int \*value) {  \*value++;  printf("value = %d\n", \*value);  }  int main(void) {  int var = 100;  incr(&var);  printf("var = %d\n", var);  return 0;  } | output:  value = 381708544  var = 100  value is unpredictable value because \*value++ will move \*value. |
|  |  |
|  |  |
|  |  |
|  |  |

### array in function

"int fun(int arr[]);" and "int fun(int arr[2]);" are the same

|  |  |
| --- | --- |
| void func(int \*arr, int len){  int I;  for(i=0; i<len; i++){  arr[i] ++;  printf("%d-%d, ", \*arr, \*(arr+i));  }  }  int arr[]={1,2,3};  func(arr, sizeof(arr)/sizeof(arr[0])); | output: 2-2, 2-3, 2-4,  pass array pointer into function |
| void func(int arr[], int len){  int I;  for(i=0; i<len; i++){  arr[i] ++;  printf("%d, ", arr[i]);  }  }  int arr[]={1,2,3};  func(arr, sizeof(arr)/sizeof(arr[0])); | output: 2, 3, 4,  pass array into function |
| int compare\_matrics(int \*m1, int \*m2, int m, int n){          int I, j, equal=2, greater=0, smaller=0;          for(i=0; i<m;i++) {              for(j=0; j<n;j++) {  **int x= \*((m1+i\*n) + j);**  **int y= \*((m2+i\*n) + j);**                  if(x==y){equal=0;}                  else if (x<y){smaller=-1;}                  else if (x>y){greater=1;}                  // printf("%d,%d\n", x, y);                }          }          if(equal==0&&greater==0&&smaller==0){              printf("Both matrices are equal\n");              return 0;          }          else if(equal==2&&greater==1&&smaller==0){              printf("Matrix A are larger\n");              return 1;          }          else if(equal==2&&greater==0&&smaller==-1){              printf("Matrix B are larger\n");              return -1;          }else{              printf("Unknown");          }          return 2;      }      int row=2, col=3;      int a[2][3]={{1,2,3},{4,5,6}};      int b[2][3]={{1,2,3},{4,5,6}};      int c[2][3]={4,5,6,7,8,9};      int d[2][3]={0,0,0,1,2,3};      compare\_matrics(a, b, row, col);      compare\_matrics(a, c, row, col);      compare\_matrics(a, d, row, col); | compare two matrices |
|  |  |
| #define ROW 2      #define COL 3      void func(int p[ROW][COL]){          for(int i=0;i<ROW;i++){              for(int j=0;j<COL;j++){                  printf("%d",p[i][j]);              }          }      }      int a[ROW][COL]={{1,2,3},{4,5,6}};      func(a); | argument: arr[ROW][COL] |
| #define ROW 2      #define COL 3      void func(int \*p){          for(int i=0;i<ROW;i++){              for(int j=0;j<COL;j++){                  int n = i\*COL+j;                  printf("%d",\*(p+n));              }          }      }      int a[ROW][COL]={{1,2,3},{4,5,6}};      int \*p=a;      func(p); | argument: \*ptr  ptr=arr[0][0] |
| #define ROW 2      #define COL 3      void func(int \*p[ROW]){          for(int i=0;i<ROW;i++){              for(int j=0;j<COL;j++){                  printf("%d",p[i][j]);              }          }      }      int \*a[ROW], n=1;      for(int i=0;i<ROW;i++){          a[i]=(int \*) malloc(COL\*sizeof(int));          for(int j=0;j<COL;j++){              a[i][j]=n;              n++;          }      }      func(a); | argument: \*arr[ROW] |
| #define ROW 2      #define COL 3      void func(int \*\*p){          for(int i=0;i<ROW;i++){              for(int j=0;j<COL;j++){                  printf("%d",p[i][j]);              }          }      }      int \*\*a, n=1;      a = (char \*\*)malloc(ROW\*sizeof(char));      for(int i=0;i<ROW;i++){          a[i]=(int \*) malloc(COL\*sizeof(int));          for(int j=0;j<COL;j++){              a[i][j]=n;              n++;          }      }      func(a); | pass \*\*p arguments into function |

### string in function

In C, string is denoted as char array. No such data type known as string.

|  |  |
| --- | --- |
| void func(char \*str){  for(int i=0; i<strlen(str); i++){  printf("%c", str[i]);  }  for(int j=strlen(str)-2; j>=0; j--){  printf("%c", str[j]);  }  }  char str[]="abcde";  func(str); | output:abcdedcba  pass string pointer into function |
| void func(char \*p){  int i=1;  while(\*p){  printf("%d=%c, ", i, \*p);  i++;  p++;  }  }  char s[20] = "good";  func(s); | output: 1=g, 2=o, 3=o, 4=d, |
| char \* combine\_str(char \*s1, char \*s2){  char \*s = (char \*) malloc(20\*sizeof(char));  int i=0;  while(\*s1&&\*s2){  \*s++=\*s1;  \*s++=\*s2;  i+=2;  printf("%c%c,", \*s1++,\*s2++);  }  \*s='\0';  s-=i;  printf("==%s\n", s);  return s;  }  void func(char \*p){  while(\*p){  printf("%c", \*p++);  }  }  char s1[]="abc";  char s2[]="XYZ";  char \*p = combine\_str(s1,s2);  printf("%s\n", p);  func(p);  //free memory  free(s1);  free(s2);  free(p); | output: 20 |
| int f(char t[]){          return t[0]-t[-1];      }      int i=2;      i-=f("ABDGK"+1);      printf("%d",i); | output:1 |
| char \*f(char \*p){  **return p++;**      };      char \*g(char \*p){          return p+=2;      };      char \*s="ABCDEFGHIJ";      char p = \*f(g(f(s+6)));      printf("%d\n", p -'A'); | output: 8  \*f(s+6)='G'  \*g(f(s+6))='I'  \*f(g(f(s+6)))='I'  Note: "return p++;" is equal to "return p;" |
| char \*f(char \*p){          return ++p;      };      char \*g(char \*p){          return p+=2;      };      char \*s="ABCDEFGHIJ";      char p = \*f(g(f(s+4)));      printf("%c\n", p); | output: 'I'  Note: compare "return ++p;" with "return p++;" The later: first return p then p+1; |

### struct in function

|  |  |
| --- | --- |
| struct STR{  int age;  char gender;  char \*name;  };  void func(struct STR s){  printf("%d,%c,%s\n", s.age,s.gender, s.name);  s.age=34;  s.gender ='M';  s.name = "good";  printf("%d,%c,%s\n", s.age,s.gender, s.name);  }  struct STR s={30, 'F', "yes"};  func(s); | pass struct into function |
| struct STR{  int age;  char gender;  char \*name;  };  void func(struct STR \*p){  printf("%d,%c,%s\n", p->age,p->gender, p->name);  p->age=34;  p->gender ='M';  p->name = "good";  printf("%d,%c,%s\n", p->age,p->gender, p->name);  }  struct STR s={30, 'F', "yes"};  func(&s); | pass struct pointer into function |
| struct element {  int value;  };  struct element func1(struct element x){  struct element y = x;  y.value += 10;  return y;  }  struct element z = {10};  struct element a = func1(z);  printf("%d\n", a.value); | output: 20  pass struct as argument into function. The function returns another struct. |
| struct element {  int value;  };  void func2(struct element \*p){  p->value += 10;  }  struct element z = {10};  struct element \*p = &z;  func2(p);  printf("%d\n", p->value); | pass pointer of struct into function. update the same struct due to the address shared. |
| struct element {          int value;      };      struct element \* func(struct element \*p){          p->value = 20;          return p;      }      struct element z = {10};      struct element \*pp = func(&z);      printf("%d, %d\n", pp->value, z.value); | output: 20,20  pass struct pointer into function, and then return this pointer |
| struct element {          int value;      };      struct element \* func(struct element \*p){          struct element  \*pp = (struct element \*)  malloc(sizeof(struct element));          pp->value = 20;          return pp;      }      struct element z = {10};      struct element \*pp = func(&z);      printf("%d, %d\n", pp->value, z.value); | pass struct to function, define a new struct and then return the pointer. |

### pointers to functions

|  |  |
| --- | --- |
| float rectangle\_area(float x, float y){      return x\*y;  }  float triangle\_area(float e, float h){      return e\*h/2;  }  int main(void)  {  **float (\*ptr) (float, float);**  **ptr = rectangle\_area;**      printf("%.2f\n", ptr(3, 3));  **ptr = triangle\_area;**      printf("%.2f\n", ptr(3, 3));      return 0;  } | output:  9.00  4.00  Note: "ptr=rectangle\_area;" or "ptr=&rectangle\_area;" is correct. but can't include parentheses. "ptr=rectangle\_area();" is wrong. |
|  |  |
|  |  |
|  |  |
|  |  |

## incursive function

|  |  |  |
| --- | --- | --- |
| |  | | --- | | #include “stdio.h”  int main()  {    printf(“GeeksQuiz\n”);    main();    return 0;  } | | run again and again till memory overflow. |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| int factorial(int n){          int res = 1;          while(n){              res \*= n--;          }          return res;      } | int factorial(int n){          if(n>1){              return n\*factorial(--n);          }else{              return 1;          }      } | calculate n! |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Stream

## IO stream

### printf()

1. format:

%d decimal integer,

%x hexadecimal integer,

%o octal integer,

%c char,

%lu long unsigned integer

%f float(4 bytes)

%lf double(8 bytes),

%% itself

2. %10d: integer with 10 width align to the right. %-10d: integer with 10 width aligned to the left

%.2f: float with 2 decimal after pointer

2. printf returns an integer value, which is the total number of printed characters.

|  |  |
| --- | --- |
|  | output |
| printf("The value of seven is: %02d\n", 2); | 02 |
| printf("The value of seven is: %.6f\n", 7.0); | 7.000000 |
| printf("The value of seven is: %f\n", 7.0); |  |
| printf("The value of seven is: %.6f\n", 09.0); | 9.000000 |
| printf("The value of ten is: %f\n", 9E0); |  |
| int i= 43;      printf( "%d", printf( "%d", printf( "%d",i))); | 4321 |
| float a=3.14159;      printf("Pi:%\*.2f\n", 10,a); | Pi: 3.14  \* denote 10 width |
| printf("The value of seven is: %.6f\n", 7); | wrong |
| printf("The value of seven is: %f\n", 7); |  |

### fprintf()

fprintf(<stream>, "<string>"):formatted writing to the stream. if successful, return length of input or return negative value

|  |  |
| --- | --- |
| char s[]="ABC";      char \*p="DEF";      int i=123;      char c='\'';      FILE \*f;      f=fopen("f", "wb");  **fprintf(f, "%d %s %s %c", i,s,p,c);**      int a = ftell(f);      printf("%d\n",a);      fclose(f); | output: 13 |
| FILE \*f;      f=fopen("f", "wb");  **int a=123;**  **fprintf(f, "%d", a);**      int b = ftell(f);      printf("%d\n",b);      fclose(f); | output: 3 |
| FILE \*f;      f=fopen("f", "wb");  **char a=123;**  **fprintf(f, "%c", a);**      int b = ftell(f);      printf("%d\n",b);      fclose(f); | output: 1 |
| FILE \*f;      f=fopen("f", "wb");  **char a[]="123";**  **fprintf(f, "%s", a);**      int b = ftell(f);      printf("%d\n",b);      fclose(f); | output=3 |
| int i;      i=fprintf(stdout,"hello!");      printf("%d", i); | output: hello!6 |
| int i;      i=fprintf(stdin,"hello!");      printf("%d", i); | output: -1  can't write stdin stream |
| int i;      i=fprintf(stderr,"123");      printf("%d", i); | output: 1233 |
| fprintf(stdout, '4');  fprintf(stdout, 123); | The two statements are wrong |

### scanf()

|  |  |
| --- | --- |
| scanf("%d", &MaxSheep); | &means the variable is currently storing |
| char a[3];      scanf("%s",&a);      printf("%s\n", a); | input:abc  output: abc |
| char a[3];      scanf("%c",&a);      printf("%s\n", a); | input:abc  output:a\*\* |
|  |  |
|  |  |

### fscanf()

fscanf(): format reading from stream. **int fscanf(FILE \*stream, const char \*format, ...)** reads formatted input from a stream.

fscanf() read string with space

|  |  |
| --- | --- |
| char s[]="Cary 2013 M";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      char p[100];      int i;      char n;      f=fopen("file", "r");      fscanf(f, "%s %d %c", p, &i, &n);      printf("<%s> <%d> <%c>\n", p, i, n); | output: <Cary> <2013> <M>  fscanf() can string, integer, float, character from text file. |
| char s[]="Cary 2013M";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      char p[100], p2[10];      f=fopen("file", "r");  **fscanf(f, "%s%s", p, p2);**      printf("<%s><%s>\n", p, p2);      fclose(f); | output: <Cary><2013M>  for fscanf(), the default separator is whitespace or \n |
| char s[]="08/5/2010";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      int y,m,d;      f=fopen("file", "r");  **fscanf(f, "%d/%d/%d", &m,&d,&y);**      printf("%d-%d-%d\n", y,m,d);      fclose(f); | output: 2010-8-5  read date string |
| char a;      scanf("%c", &a);      printf("%c", a); | output is a if entering "abc" from keyboard |

## location

ead below table for the difference between open modes r, r+, w, w+, a and a+ in open() function.

|  | **r** | **r+** | **w** | **w+** | **a** | **a+** |
| --- | --- | --- | --- | --- | --- | --- |
| read | \* | \* |  | \* |  | \* |
| write |  | \* | \* | \* | \* | \* |
| create |  |  | \* | \* | \* | \* |
| truncate |  |  | \* | \* |  |  |
| position at start | \* | \* | \* | \* |  |  |
| position at end |  |  |  |  | \* | \* |

#include <stdio.h>

because that's where the declaration of the three streams is placed.

The declaration looks as follows:

FILE \*stdin, \*stdout, \*stderr;

The stdin stream: the stdin stream is normally associated with the keyboard, pre-opened for reading and regarded as the primary data source of running programs;

the scanf() function reads the data from stdin by default.

The stdout stream: the stdout stream is normally associated with the screen, pre-opened for writing, regarded as the primary target for outputting data by the running program;

the printf() function outputs the data to the stdout stream.

The stderr stream : the stderr stream is normally associated with the screen, pre-opened for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;

|  |  |
| --- | --- |
| fprintf(stderr,"error"); | output: error |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### fopen() and fclose()

|  |  |
| --- | --- |
| FILE \*f;      f = fopen("file", "w");      fclose(f); | 1. file handle should be pointer  2. path is different between UNIX and Windows  3. fopen() and fclose() should always be used together. |
|  |  |
|  |  |

### fseek(), ftell() and rewind()

**fseek(<stream>, <offset>, <position>)**: move pointer to the beginning or end of the stream or keep current position

**ftell(<stream>)**: return current position relative to the beginning of the stream. Both fseek() and ftell() are used for calculating size of file.

**void rewind(<stream>)**: sets the file position to the beginning of the file of the given **stream**.

|  |  |
| --- | --- |
| FILE \*f;      int i;      f=fopen("f.txt", "wb");      printf("<><%s>\n",f);      int x= fwrite(f,2,1, f);  **i = ftell(f);**      fclose(f);      printf("%d\n", i);      f=fopen("f.txt", "rb");  **fseek(f, 2, SEEK\_END);**  **i = ftell(f);**      fclose(f);      printf("%d\n", i); | output:  2  4  size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream) writes data from the array pointed to, by ptr to the given stream. |
| FILE \*f;      int i;      f=fopen("f", "wb");      fclose(f);      f=fopen("f", "rb");      // move pointer to the end      fseek(f, 0, SEEK\_END);      //current value of the position indicator      i = ftell(f);      fclose(f);      printf("%d\n", i); | output: 0  int fseek(FILE \*stream, long int offset, int whence)  offset is number of bytes  SEEK\_EST:beginning  SEEK\_END: end  SEEK\_CUR: current position  long int ftell(FILE \*stream) |
| FILE \*f = fopen("f", "w+b");      fputs("123abc", f);  **rewind(f);**      fputs("3",f);      fclose(f);      int i;      f=fopen("f", "rt");      fscanf(f, "%d", &i);      fclose(f);      printf("%s\n",i); | output: 323  rewind() move pointer the beginning of the file stream |
| FILE \*f = fopen("f", "w+b");      fputs("123abc", f);      rewind(f);      fputs("3",f);      fclose(f);      char i[10];      f=fopen("f", "rt");      fscanf(f, "%s", &i);      fclose(f);      printf("%s\n",i); | output:323abc  compared with the outcome of the previous snippet code. |
|  |  |

## read stream

example:

int c = fgetc(f); or char c= fgetc(f);

char p[20]; p=fgets(p, 20, f); #conain 19 character plust '\0'

char p[20];int i; fscanf(f, "%s %d", p,i);

char p[10];fread(p,5,2,f); #read bytes. so no "\0" is included

### fgetc()

fgetc(<stream>): This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

|  |  |
| --- | --- |
| char s[20]="ABC";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      f=fopen("file", "r");  **int i=fgetc(f);**      printf("%c",i); | output:A  The statement of "int i=fgetc(f);" and " char i=fgetc(f);" are equal |
| char s[20]="ABC";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      f=fopen("file", "r");      for(int i=0; i<5;i++){          int a=fgetc(f);          printf("%c,",a);      } | output: A,B,C, ,  return the character or EOF(-1) |
| **char s[20]="AB\0CDE";**      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      f=fopen("file", "r");      for(int i=0; i<4;i++){          int a=fgetc(f);          printf("%c,",a);      } | output: A,B, , ,  only export A and B because '\0' is met |
| FILE \*f = fopen("file", "w");      fputs("ABC\n", f);      fputs("123", f);      fclose(f);      f=fopen("file", "r");      for(int i=0; i<8;i++){          int a=fgetc(f);          printf("%c",a);      }      fclose(f); | output:  ABC  123  Note: fgetc() captures '\n' |
|  |  |
|  |  |
|  |  |
|  |  |

### fgets()

**char \*p = fgets(char \*p, <length in bytes>, <stream>)**: On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned.

If an error occurs, a null pointer is returned.

|  |  |
| --- | --- |
| char s[20];      FILE \*f = fopen("file", "w");      int i = fputs("12ABCDE", f);      fclose(f);      f=fopen("file", "r");  **fgets(s,2,f);**      puts(s); | output: 1  return a string with one character |
| char s[20];      FILE \*f = fopen("file", "w");      int i = fputs("12ABCDE", f);      fclose(f);    char \*p=fgets(s,4,f);      printf("%s", p);      fclose(f); | output: 12A  return a string with 3 character |
| char s[20];      FILE \*f = fopen("file", "w");      fputs("123", f);      fclose(f);      f=fopen("file", "r");      for(int i=0; i<4;i++){          fgets(s,2,f);          printf("%s", s);      } | output: 1233  If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned. |
| char s[20]="ABC";      FILE \*f = fopen("file", "w");      fclose(f);      f=fopen("file", "r");      fgets(s,2, f);      printf("%s",s); | output=ABC  the content of the file stream is empty. So the string pointer keeps the original position. |
| f=fopen("filed", "r");      char \*p=fgets(s,2, f);      printf("%s",p); | output: (null)  the file doesn’t exist. return NULL pointer |
| char s[20]="?";      FILE \*f = fopen("f", "w");      int i = fputs("789", f);      fclose(f);      f=fopen("f", "r");      fgets(s+1, 3, f);      printf("%c\n",s[2]-s[3]);      fclose(f); | output:8  s="?\0"  fgets() store a string "78\0" to s+1  so s is "?78\0" |

### fread()

fread(char \*p, <size>, <number>, <stream>): read bytes from stream. **size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)** reads data from the given **stream** into the array pointed to, by **ptr**. The total number of elements successfully read are returned as a size\_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.

|  |  |
| --- | --- |
| char s[]="abc";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      char p[100];      f=fopen("file", "r");  **fread(p, 3, 1, f);**  **\*(p+3)='\0';**      printf("%s\n", p);      fclose(f); | output: abc  **Note: p is only char pointer. '\0' should be added in order to print it as string type.**  **compare with "fgets(p, 4,1, f);"** |
| char s[]="abcde12345";      FILE \*f = fopen("file", "w");      fputs(s, f);      fclose(f);      char p[10];      f=fopen("file", "r");      fread(p, 4, 1, f);      for(int i=0; i<10; i++){          printf("%c,", \*(p+i));      }      fclose(f);      free(p); | output: a,b,c,d,,P, ,a,,k,  except the first "abcd", the other characters are unpredictable. |
|  |  |
|  |  |

## write to the stream

char c= fputc('a', f);

int a=fputs("abc", f); return 0 or -1

fprintf(f, "%s", a) writes data as a string.

fwrite(p, 10,2,f) writes data with 20 bytes from pointer p to file f.

### fputc()

fputc(char c, <stream>): writing one character to the stream

|  |  |
| --- | --- |
| char s[]="abcde12345";      FILE \*f = fopen("file", "w");  **fputc(s[0], f);**      fclose(f);      free(s);        char p[10];      f=fopen("file", "r");      fgets(p, 10, f);      printf("%s", p);      fclose(f);      free(p); | output:a |
| FILE \*f = fopen("file", "w");      fputc(65, f);      fclose(f);      char p[10];      f=fopen("file", "r");      fgets(p, 10, f);      printf("%s", p);      fclose(f);      free(p); | output: A |
| FILE \*f = fopen("file", "w");      fputc("123", f);      fclose(f); | output: unpredictable character |
| FILE \*f = fopen("file", "w");      int a= fputc('a', f);      printf("input = %c\n", a);      fclose(f); | output: input = a |

### fputs()

fputs("<string>", <stream>): writing a string to the stream.  returns a non-negative value, or else on error it returns EOF.

|  |  |
| --- | --- |
| int a= fputs("abc", stdout);      printf("%d", a); | output:abc0  successful return 0 |
| int a= fputs("abc", stdin);      printf("%d", a); | output:-1  failure return -1 |
|  |  |

### fwrite()

fwrite(char \*p, <size>, <number>, <stream>): writing bytes to the stream. fwrite() is for binary data.

|  |  |
| --- | --- |
| FILE \*f;      f=fopen("file\_write.txt", "wt");      char s[]="abcdef";      printf("%d\n", sizeof(s));  **fwrite(s, 2,2, f);**      fclose(f); | the string size is 7 bytes, but only 4 bytes, that is "abcd", is exported into the file. |
| FILE \*f;      f=fopen("file\_write.txt", "wt");      char s[]="abcdef";  **fwrite(s, 10,2, f);**      fclose(f); | In the file, fwrite() exports 20 bytes, but the string is shorter than 20. Therefore, "abcdef" followed by unpredictable characters in the file. |

## text stream

FILE \*file=fopen(), file is pointer to file or NULL if opening failed.

fprintf() write text data

|  |  |
| --- | --- |
| FILE \*file;      char line[10], lines[2][10];      // read mode, text mode      file = fopen("f:\\C\\file.txt", "rt");      // exist if file not found      if(file == NULL){          exit(0);      }      // read line by line      int i=0;      while(fgets(line, sizeof(line), file) != NULL){          // trim trailing newline character          strcpy(lines[i], strtok(line, "\n"));          printf("%s\n", lines[i]);          i++;      }      fclose(file); |  |
| FILE \*file;      file = fopen("f:\\C\\file\_out.txt", "wt");      char lines[7][10]={"Monday", "Tuesday",  "Wednesday", "Thursday",          "Friday", "Saturday", "Sunday"};      for(int i=0; i<7; i++) {          printf("%s\n", lines[i]);          // write string line by line          fputs(lines[i], file);          fputs("\n", file);      }      fclose(file); |  |
| FILE \*fp1, \*fp2;  fp1 = fopen("file", "r");  fp2 = fopen("file", "r"); | It is ok to open the same file in read mode. but that pattern is not recommended in write mode. |
|  |  |
| FILE \*f;      int i;      f=fopen("f.txt", "wb");      printf("<><%s>\n",f);      int x= fwrite(f,2,1, f);      i = ftell(f);      fclose(f);      printf("%d\n", i);      f=fopen("f.txt", "rb");      fseek(f, 2, SEEK\_END);      i = ftell(f);      fclose(f);      printf("%d\n", i); | output:  2  4  size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream) writes data from the array pointed to, by ptr to the given stream. |

### read csv

|  |  |
| --- | --- |
| FILE \*f = fopen("file", "w");      fputs("1,John Adam,2010-10-01,M\n", f);      fputs("2,Smith Howard,2012-08-12,M\n", f);      fputs("3,Mary Carie,2000-01-22,F", f);      fclose(f);      struct ST{          int id;          char \*name;          char \*date\_birth;          char \*gender;      } person[3];      char p[100];      f=fopen("file", "r");      int i=0;      while(!feof(f)){          fgets(p,100, f);          // printf("%s", p);          char \*token=strtok(p, ",");          person[i].id= atoi(token);          printf("id=%d\n", person[i].id);          person[i].name=strtok(NULL, ",");          person[i].date\_birth=strtok(NULL, ",");          person[i].gender=strtok(NULL, ",");          i++;      }      fclose(f);      free(p); | read lines from csv file and save data into struct. |
|  |  |

## EOF: end of file

[getc()](http://www.cplusplus.com/reference/clibrary/cstdio/getc/)returns EOF when end of file is reached.

[feof()](http://en.wikipedia.org/wiki/Feof) which returns non-zero value only if end of file has reached, otherwise it returns 0. int feof(FILE \*stream); check end of a file

|  |  |
| --- | --- |
| FILE \*f = fopen("f", "wt");      char s[]="abc";      fprintf(f, "%s\n", s);      fprintf(f, "%s", s);      fclose(f);      f=fopen("f", "r");      int ch = getc(f);      while(ch != EOF){          putchar(ch);          ch=getc(f);      }      if(feof(f)){          printf("<End\n");      }      fclose(f); | output:  abc  abc>End |
| printf("%d", EOF); | output: -1 |
|  |  |
|  |  |
|  |  |

## handle errors

|  |  |
| --- | --- |
| FILE \*file;      errno = 0;      file = fopen("f:\\C\\dfile.txt", "rt");      // exist if file not found      if(file == NULL){          printf("errno=%d\n", errno);          printf("%s\n", strerror(errno));          exit(0);      } | output:  errno=2  No such file or directory  errno is none-zero integer type, and represent error  strerror(): contain readable error information. The errno and stderror() are usually used together.  Note: perror() could be equal here. |
| FILE \*file;  file=fopen("c:\\a.txt", "rt");  if(file==NULL){  perror("Error"); }  fclose(file); | output: Error: No such file or directory  **errno=2**  cause: the file doesn't exist  void perror(const char \*str) prints a descriptive error message to stderr. |
| FILE \*file;      file=fopen("f:\\C\\file.txt", "r");      if(file==NULL){          perror("Error");      }else{          fputs("abc", file);          if(ferror(file)){              printf("Error %d: can't write.", errno);          }      }      fclose(file); | output: Error 9: can't write.  **errno=9**  Cause: Can't write to the file which is opened in read-only mode.  ferror() |
| FILE \*file;      file=fopen("f:\\C\\file\_readonly.txt", "wt");      if(file==NULL){          perror("Error");          printf("errno=%d",errno);      }      fclose(file); | output:  Error: Permission denied  **errno=13**  try to open readonly file in written mode. |
|  |  |
|  |  |
|  |  |

## syntax errors

|  |  |
| --- | --- |
| FILE \*file;      file=fopen("f:\\C\file.txt", "r");      if(file==NULL){          perror("Error");      }      fclose(file); | output: Error: Invalid argument  file path in windows should be file=fopen("f:\\C\\file.txt", "r"); |
| FILE \*f;      f=fopen("file\_write.txt", "wt");      float i=65;      fwrite(&i, 1, 1, f);      fclose(f); | unpredictable exports in the file.  fwrite() write binary data in bytes into file. for normal text, fprintf() should be used. |
| FILE \*f;      f=fopen("file\_write.txt", "wt");      int i[]={1,2,3};      printf("%d\n", sizeof(i));      fwrite(i, 4,3, f);      fclose(f); | unpredictable exports in the file. |
| FILE f;  f = fopen("file", "w");  fclose(f); | file handler should be pointer  correct: FILE \*f; |
| FILE \*f = fopen("file", "w");      fputs(f, "ABC");      fclose(f); | wrong arguments for fputs()  correct: fputs("ABC", f); |
|  |  |

# Preprocessor

In preprocessor directive, don’t add semicolon ';', which cause errors.

preprocessor directive starts with '#', a prefix unary operator

## #include

load source code form other places into current code. use #include header file to avoid loading file more than once.

After the pre-processing of a \*.c program, a \*.i file is generated which is passed to the compiler for compilation, and after compilation, a \*.exe file is generated.

|  |  |
| --- | --- |
| #include <stdio.h> | #include means preprocessor directive. The default path for stdio.h will be stored in /usr/include at Unix |
| #include "external.h" | load the head file external.h, which is store in same directory with the source code.  pro |
| int main(void) {  #include "src2.c"  } | #include could be anywhere in the code. |

## #define

### replacement macro

define a replacement macro namely constant or partial C codes.

In most cases, const is preferred to than #define. but #define is useful for conditional compilation

|  |  |
| --- | --- |
| #define SIZE 26  int main(){        int a[SIZE];      char b[SIZE];      for(int i=0; i<SIZE; i++){          a[i]=i+65;          b[i]=i+65;          printf("%d: %d,%c\n",i, a[i],b[i]);      }        return 0;  } | SIZE is replacement macro constant  note: uppercase is suggested for macro variable |
| #define SIZE 20  #define FOR for(int i=0; i<SIZE; i++)  #define BEIGIN {  #define END }  int main(){        int a[SIZE];      FOR BEIGIN          a[i]=i+65;          printf("%c", a[i]);      END        return 0;  } | output: ABCDEFGHIJKLMNOPQRST  Compare with normal C code:      int a[20];      for(int i=0; i<20; i++){          a[i]=i+65;          printf("%c", a[i]);      } |
| #define SIZE 20;  #define FUG SIZE-2;  int main(){        int size=FUG;      printf("%d\n", size);        return 0;  } | output: 20 (should be 18)  #define statement can't end with semicolon. #define doesn't check syntax error, so the compiling will not raise error. but the semicolon will cause unexpected value. |
| **#define DIE \**  **fprintf(stderr, "Fatal Error\n");\**  **exit(8);**  int main(){        int weight=-2;      if(weight<0)          DIE;        return 0;  } | output: Fatal Error  Note: use backslash \ if more than one statements are needed. if removing either of the \ will cause compiling error |
| #define SIZE 4+4  int main(void) {  int i;  i = 2 \* SIZE;  printf("%d\n",i);  return 0;  } | output: 12  the statement i=2\*SIZE should be i=2\*8=16. However actually i=2\*4+4=12 because #define statement only replace SIZE with '4+4' rather than calculating that.  correct: #define SIZE (4+4) |
| #define NULL ((void \*) 0)  #define EOF (-1)  #define SEEK\_SET 0  #define SEEK\_CUR 1  #define SEEK\_END 2 | In stdio.h and stdlib.h define some default macro constants |
| #define A(x,y) x+y      int i =  -1;      int i2 = -2;      printf("%d\n",-A(i,i2)); | -A(i,i2)=--1+-2=-1 |
| #define A(x) -x      int i =  A(2-1);      printf("%d\n",i); | i=A(2-1)=-2-1=-3 |
| #define A -B  #define B -C  #define C 5    int main()  {    printf("%d\n", A);    return 0;  } | output:5 |
| #include <stdio.h>  #define a 10  int main()  {  printf("%d ",a);    #define a 50    printf("%d ",a);  return 0;  } | output: 10,20 |

### macro with parameters

#define identifier(parameter\_list) text

|  |  |
| --- | --- |
| #define F(x) (x\*x)  int main(){     float y=F(3);    printf("y = %.2f",y);      return 0;  } | output: y=9.00  define micro know as F with one parameter |
| #define F(x) (x\*x)  int main(){      int len=10;  **float y=F(len+1);**    printf("y = %.2f",y);      return 0;  } | output: y=21.00  **Note: macro is not a function though it likes a function because just simple replacement**  **The statement** float y=F(len+1); will be compiled to **float y=(len+1\*len+1);**  **correct: "float y=F((len+1));" OR**  **"#define F(x) ((x)\*(x))"** |
| #define MAX(x,y) ((x)>=(y)) ? (x): (y)  int main(){       int a=10, b=-20;      printf("%d\n", MAX(a,b));      printf("%d\n", MAX(EOF,b));      printf("%d\n", MAX(a-20,b\*2));      return 0;  } | output:  10  -1  -10 |
|  |  |
| #define F1(X) X\*X  #define F2(X) (X)\*(X)  #define F3(X) ((X)\*X)  int main(void){      int i=1, j=2, k=3;      int s = F1(i+j)+F2(i-j)+F3(i+k);      printf("%d", s);      return 0;  } | output:13  s=i+j\*i+j+(i-j)\*(i-j)+(i+k)\*i+k |
| int X=100;      int a=X;      a+=X;      #define X 200      a+=X;      #undef X      printf("%d", a); | output:400 |
| # define scanf  "%s Geeks Quiz "  int main(){     printf(scanf, scanf);     return 0;  } | output: %s Geeks Quiz Geeks Quiz  ->printf("%s Geeks Quiz", "%s Geeks Quiz"); |
| **#define CALL(param) \**  **{ if(param) cout << param++; }**  int main(void) {      int i=1;    CALL(i);    cout << i;      return 0;  } | output:12 |
| #define F(arg) if(arg) cout<<"TEST";  int main(void) {      int a= 1;    F(a < 3);      return 0;  } | output: TEST  ->if(a<3) cout<<"TEST" |
|  |  |

### macro-operators: # and ##

|  |  |
| --- | --- |
| int xy=34;      #define F(a,b) a##b      int a=F(x,y);      #undef F      printf("%d\n",a); | output: 34 |
| #define CIT(X) #X  #define CNC(X,Y,Z) X##Y##Z  #define VAL 641221  int main(void){      int i = CNC(64,12,21);      int j = i+ VAL;      char \*s = CIT(i);      printf("%d%s\n",j,s);      return 0;  } | output: 128442i |
| #define SYM  #define BOL 100  #define SMB SYM##BOL  #define SBL #BOL  #undef SYM    int main(void){      #ifdef SYM          int i=100;          #ifdef SYMBOL              int j=i+200;          #else              int j=i+222;          #endif      #else          int i=200;          #ifdef SYMBOL              int j=i+100;          #else              int j=i+111;          #endif      #endif      printf("%d\n",i+j);      return 0;  } | output: 511  Note: "#define SYM" is meaningless though no compiling error rose. the macro SYM is not declared after compiling. |
| #define A(x) #x      int i = -1;  **char \*s = A(i);**      i = -(s[0]=='i');      printf("%d\n",i); | output: -1  char \*s="i"; |
| #define macro(n, a, i, m) m##a##i##n  #define MAIN macro(n, a, i, m)    int MAIN()  {      printf("good");      return 0;  } | output: good |

### scope of #define

|  |  |
| --- | --- |
| int X=0;  #define X 100  int f1(void){      return X;  }  #undef X  int f2(void){      return X;  }  int main(void){      int s;      s=f1()+f2();      printf("%d", s);      return 0;  } | output: 100  f1() in the scope of #define which override the variable X=0; |
|  |  |
|  |  |

## identifier

|  |  |
| --- | --- |
| #include <stdio.h>  int main(void) {  printf("this is line #%d\n", \_\_LINE\_\_);  printf("this is line #%d\n", \_\_LINE\_\_);  printf("this is line #%d\n", \_\_LINE\_\_);  return 0;  } | \_\_LINE\_\_ it is replaced by an integer literal equal to the line number |
| #include <stdio.h>  int main(void) {  puts("Hello from the source file named "\_\_FILE\_\_);  return 0;  } | The \_\_FILE\_\_ identifier is always replaced by a string literal containing the name of the source file in which the identifier was used. |
| #include <stdio.h>  int main(void) {  puts("The program was successfully compiled on " \_\_DATE\_\_);  return 0;  } | The \_\_DATE\_\_ identifier is always replaced by a string literal containing text denoting the day the source file was compiled. |
| #include <stdio.h>  int main(void) {  puts("I was compiled at " \_\_TIME\_\_);  return 0;  } | The \_\_TIME\_\_ identifier is always replaced by a string literal containing text denoting the time (hours, minutes, seconds) the source file was compiled. |
| #define PR(X) \      printf("The value of #X is %d\n", (X));  int main(void) {      int a='A';      PR(a);      return 0;  } | The value of #X is 65  Note: #X represent X itself. |
| #define PR(X) \      printf("The value of "#X" is %d\n", (X));  int main(void) {      int my='A';      PR(my);      return 0;  } | output: The value of my is 65  Note: "#X" represent the name of imported macro itself. |
| #define concat(a, b) (a##b\*2)  int main(void)  {      int x=1, y=2, xy=123;      printf("%d", concat(x, y));      return 0;  } | output: 246  The ## operator is token-pasting operator requiring two arguments, and concatenate two arguments into one macro. |

## conditional compilation: #IF-#ELIF-#ENDIF

|  |  |
| --- | --- |
| #define A  #define C  int main(void){      int i =          #ifdef A              #ifdef B                  -1              #else                  -2              #endif          #else              -3          #endif          ;      printf("%d\n",i);      return 0;  } | output: -2 |
| #if X == 3      #define Y 3  #else      #define Y 5  #endif    int main()  {      printf("%d", Y);      return 0;  } | output: 5 |
| #include <stdio.h>  #define ISEQUAL(X, Y) X == Y  int main()  {      #if ISEQUAL(X, 0)          printf("Geeks");      #else          printf("Quiz");      #endif      return 0;  } | output: Geeks |
| #define ISEQUAL(X, Y) X == Y  int main()  {      #if X          printf("Geeks");      #else          printf("Quiz");      #endif      return 0;  } | output: Quiz  Note: define macro ISEQUAL, X, and Y. X==0, Y==0 |
| #include <stdio.h>  #define X  int main()  {      #if X          printf("Geeks");      #else          printf("Quiz");      #endif      return 0;  } | compiling error:  #ifdef X |
| #define X 3  #if !X      printf("Geeks");  #else      printf("Quiz");    #endif  int main()  {          return 0;  } | compiling error  A program is converted to executable using following steps: 1 Preprocessing. 2) C code to object code conversion. 3) Linking  The first step processes macros. So the code is converted to following after the preprocessing step.  printf("Quiz");  int main()  {  return 0;  }  The above code produces error because printf() is called outside main. |
| int add(int x) { return x + 1; }  int main(void) {  int i = 100;  i = add(i);  #define add(x) (2 \* (x))  i = add(i);  #undef add  i = add(i);  printf("%d",i);  return 0;  } | output: 203  Note: 1. use #define and function  2. use #undef remove definition of macro |

## keyword: const

Variables can be declared as constants by using the “const” keyword before the datatype of the variable. The constant variables can be initialized once only. The default value of constant variables are zero.

1. use const for pointer or reference(C++) arguments in function, if you don't want the change arguments in the function. No need apply const into variable values because it is meaningless.

2. use const for variables if you don't want update it in the future. esp, the variable could be used in other c files. Use const instead of #define: 1. C check syntax error because #define is done by preprocessor. The compiler can't check syntax error. 2. const follow normal C scope rules.#define just simple replace something. #define is none scope control, which can't be used anywhere in the c file.

|  |  |
| --- | --- |
| const int a;     const int b = 12;     printf("%d,%d", a,b); | output: unknown, 12  Note: variable a and be can't be updated in the future because they are const. |
| const int a;     const int b = 12;     printf("%d,%d", a,b);     a=3,b=4;     printf("%d,%d", a,b); | compiling error |
| #define num 25  int main() {     printf("The value of num is: %d", num);     return 0;  } | Variables can be declared as constants by using the #define preprocessor directive as it declares an alias for any value.  The same as const. The macro can be updated after declaration. |
| int f(const int \*p, int a){      return 1;  }  int main() {     int a=3;     int b[]={1};     int \*p=b;     f(p,a);     return 0;  } | The pointer could be updated.For example:  int f(const int \*p, int a){      (\*p)++;      return 1;  }  will cause compiling error. |

# Errors in C

## Syntax erros

### declaration and initiation

Once a variable is declared, that should be initialized in the next statements before you operate it. No compiling error if no initiation, but that variable may return unpredictable values.

* Illegal variable name, such as 6part, ip-part, int, float. Those names below are ok: ip\_part, part2, Int, INT. keyword must not be used in the meaning other than defined in the language standard. namely "float" can't be used.
* Repeat declare: int a=5; int a=6;
* any pointer should be declared, allocated to memory and then initialized. If allocation is ignored, there is not output or unpredictable value though there is no compiling error

|  |  |
| --- | --- |
| int a;      int b=a\*10;      printf("%d", b); | variable is not initialized. output is unpredictable value |
| char \*a[10];      printf("%s", a); | unpredictable value |
| struct P{          int a,b;      } p;      printf("%d", p.a); | unpredictable value |
| int a=2;  **a<<a;**      printf("%d\n", a); | output: 2  "a<<a;" is expression, which returns 8. but That can't change value of a. |
|  |  |
| #include <stdio.h>  int main(void) {  char \*ptr;  \*ptr = 'C';  printf("%c",\*ptr);  return 0;  } | Not output. use uninitialized pointer  should allocate memory to the pointer firstly before initialization  "ptr = (char \*)malloc(sizeof(char));" |
| int main(void) {  char \*ptr;  strcpy(ptr, "you may get into trouble soon");  puts(ptr);  return 0;  } | Not output. use uninitialized pointer |
|  |  |

### semicolon ;

semicolon sign must be used for ending one statement of c code. Don’t include ; for preprocessor. It is optional for for-loop, while-loop, if-statements.

|  |  |
| --- | --- |
| #define S 10;      int a = S \* 10;      printf("%d\n",a); | correct: #define S 10 |
| **#define S 10**      #define A(S) S\*10      int a = A(20) \* 10;      printf("%d\n",a); | output: 2000  it is ok to use "#define S 10;",but that is not recommended because that is followed by another #define. |
|  |  |
|  |  |
|  |  |
|  |  |

### Wrong type

|  |  |
| --- | --- |
| char a="ABC"; | a should be char array rather than char  correct: char a[]="ABC"; |
| char a[]='ABC'; | correct "ABC" rather than 'ABC' |
| float func(float s){          return s\*s;      }      int a=10;      printf("%.2f\n",func(a)); | it is ok in C, but not suggested. |
| int a = '\'; | **compiling error. should add escape operator \**  **correct: int a = '\\';** |
|  |  |

## out of bounds

array in C is static type once it is defined and initialized. C doesn't suggest changing size of array. but it is ok to access values out of bounds. No compiling error but segment fault may be raised.

|  |  |
| --- | --- |
| int a[4]={1,2,3,4};      for(int i=0;i<6;i++){          printf("%d,", a[i]);      } | output: 1,2,3,4,4,6422284, |
| int a[5]={1,2,3,4};      for(int i=0;i<6;i++){          printf("%d,", a[i]);      } | output: 1,2,3,4,0,5,  a[4] is initialized as 0. index=5 is out of bounds. a[5] is unpredictable value or cause segment fault. |
| char a[]={'A','B','C'};      for(int i=0;i<4;i++){          printf("%c,", a[i]);      } | output: A,B,C,♥,  the string a doesn't contain '\0'.  a[3] is out of bounds |
| char a[]="ABC";      char \*p =a;      while(\*p!='\0'){          printf("%c,", \*p++);      } | use '\0' to protect from out of bounds |
| int a[2][3]={1,2,3,4,5,6}; | but either "int a[][]={1,2,3,4,5,6};" or "int a[2][]={1,2,3,4,5,6};" is wrong. |
| char \*str1 = "India";      char \*str2 = "BIX";      char \*str3;      str3 = strcat(str1, str2);      printf("%s %s\n", str3, str1); | It prints 'IndiaBIX IndiaBIX' in TurboC (in 16 bit platform).  It may cause a 'segmentation fault error' in GCC (32 bit platform).  because \*str1 can't include the str1+str2. |
| #include <stdio.h>  #include <string.h>  int main(void) {  char str[10];  strcpy(str,"Welcome to Troubleland!");  printf("%s",str);  return 0;  } | the string is longer than the size of 10. |
| #include <stdio.h>  #include <string.h>  int main(void) {  char str[10];  int i;  strcat(str,"Bump!");  printf("%s",str);  return 0;  } | The strcat function will persistently look for the end of the string, and it’ll probably find one eventually (there’s a good chance that there is a byte containing a zero somewhere in the memory), but more than likely not within the str array itself. |
| struct A {      int a;  } A1;  struct B {      int b;  } B1;  int main(void)  {      A1.a = 10;      B1.b = 100;      char\* x = (char\*) &A1;      char\* y = (char\*) &B1;      y = (char\*) 0x100;      x[0] = y[1];      printf("%d\n", A1.a);      printf("%d", B1.b);  } | segment fault  0x100=16^2=256=1 00000000, but size of y pointer is only 1 byte |

## scope

|  |  |
| --- | --- |
| if(int j)          printf("%d\n",j); | compiling error: j is undeclared |
| for(int j=0;j<10;j++)          printf("%d\n",j); | correct  Note: compare it with the above |
|  |  |
|  |  |
|  |  |

# algorithm

## Tricky problems

check even or odd

|  |  |
| --- | --- |
| int i;      i =10;      if(i%2==0) puts("even");      else puts("odd");      i =17;      if(i%2==0) puts("even");      else puts("odd"); | use modulus operator % |
| int isEven(int i){          return (i/2)\*2 == i;      } | quotient is truncated into floor integer if division is not integer |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

add two numbers without "+" operator

|  |  |
| --- | --- |
| int x, y;  printf("Enter two number: ");  scanf("%d %d",&x,&y);  // method 1  printf("%d\n", x-(-y));  // method 2  printf("%d\n", -(-x-y));  // method 3  printf("%d\n", abs(-x-y));  // method 4  printf("%d", x-(~y)-1); |  |
|  |  |

detect the age:

A guy is x years old. x^3 is 4 digits, and x^4 is 6 digits. Both 4 and 6 digits number address 0-9 digits once. How old he is?

|  |
| --- |
| int power(int base, int exponent){      if(exponent==0){          return 1;      }else{          return base\*power(base, exponent-1);      }  }  int main()  {      for(int i=2; i<200; i++){          int a, b, d, d1,d2;          int digits[11]={0};          a = power(i, 3);          if(a>=1000 && a<=9999){              for(int m=1;m<5;m++){                  d1= a/power(10, 4-m);                  d2= a/power(10, 4-m+1);                  d=d1-d2\*10;                  // printf("%d, %d\n",a, d);                  if(digits[d]==0){                      digits[d]=1;                      digits[10] +=1;                  }else{                      digits[10]-=1;                  }              }          }          b = power(i, 4);          if(b>=100000 && b<=999999){              for(int n=5;n<11;n++){                  d1= b/power(10, 6-(n-4));                  d2= b/power(10, 6-(n-4)+1);                  d=d1-d2\*10;                  // printf("%d, %d\n",b, d);                  if(digits[d]==0){                      digits[d]=1;                      digits[10]+=1;                  }else{                      digits[10]-=1;                  }              }          }          printf("%d: %d-%d, counts=%d\n", i, a,b, digits[10]);          if(digits[10]==10){              printf("age=%d\n",i);              break;          }      }        return 0;  } |
|  |

## recursive

|  |  |
| --- | --- |
| int factorial(int n){      if (n<=1) {        return 1;    }  else{        return n\*factorial(n-1);    }  } | n!=n\*(n-1)\*(n-2)\*…\*1 |
| int fab(int n){      if(n==1){        return 1;    }  else if (n==2){        return 2;    }  else{        return fab(n-1) + fab(n-2);    }  } | f(n)=f(n-1)+f(n-2) |
| int power(int base, int exponent){      if(exponent==0){        return 1;    }  else{          return base\*power(base, exponent-1);      }  } |  |
| struct Binary{      int value;      struct Binary \*next;  };  struct Binary \*DecimalToBinary(struct Binary \*ptr, int i){      struct Binary \*new=(struct Binary \*)malloc(sizeof(struct Binary));      if(i==1){          // printf("%d",i);          new->value=1;          new->next=ptr;          return new;      }else{          int reminder = i%2;          // printf("%d",reminder);          new->value = reminder;          new->next=ptr;          return DecimalToBinary(new, i/2);      }  }  int main(void){      // decimal: 11101010      int i=234;      struct Binary \*res=(struct Binary \*)malloc(sizeof(struct Binary));      struct Binary \*ptr=(struct Binary \*)malloc(sizeof(struct Binary));      ptr->value = i%2;      ptr->next=NULL;      if(i>=2){          res=DecimalToBinary(ptr, i/2);      }        while(res!=NULL){          printf("%d",res->value);          res=res->next;      }        return 0;  } | convert decimal to binary in linked list |
| char \*ReverseDNA(char \*ptr, char \*rev){      if(\*ptr=='\0'){          return ++rev;      }else{          \*rev=\*ptr;         return ReverseDNA(++ptr, --rev);      }  }  int main(void){        char dna[]="AAATTTCCCGGG";      char \*ptr =dna;      int len = strlen(dna);      printf("len=%d\n",len);      char \*rev = (char \*)malloc((len+1)\*sizeof(char));      \*(rev+len+1)='\0';      rev=ReverseDNA(ptr, rev+len);      printf("%s\n", ptr);      printf("%s\n", rev);      free(rev);      free(dna);      return 0;  } | reverse DNA sequence |

recursive print

|  |  |
| --- | --- |
| void f(int n){      if(n<2){        f(++n);  cout << n;    }  }  int main(void) {      int a= 0;     f(a);      return 0;  } | output:21  n=0 ->f(1) ->f(2), print 2 -> f(1) print 1 |
| int f(int n){      if(n==0) return 1;      else {          f(--n); cout << n;      }  }  int main(void) {      int a= 3;      f(a);      return 0;  } | output:012  Note: f(n--) is wrong casuing infinite loop. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## sort

bubble sort (increment):

1. outer loop is n times for n elements. select minimum one at a time.

2. inner loop select minimized one by compared all other elements.

|  |  |
| --- | --- |
| #include <stdio.h>  int main(){      int n=5;      int a[5] = {8, 10, 6, 2, 4};      int i,m;      for(i=0;i<n-1;i++){          for(m=i+1;m<n;m++){              if(a[i]>a[m]){                  a[i]=a[i]-a[m];                  a[m]=a[i]+a[m];                  a[i]=a[m]-a[i];              }          }      }      for(i=0;i<n;i++){          printf("%d,", a[i]);      }        return 0;  } |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## two pointers

|  |  |
| --- | --- |
| int SearchSubStr(char \*p, char \*sub){      char \*next=sub;      int start=-1, end=-1, i=0;      while(\*p!='\0' && \*next!='\0'){          if(\*p==\*next){              if(end==-1)                  start=i, end=0;              printf("%d:%c\n", i, \*p);              next++, end++;          }else{              start=-1, end=-1, next=sub;          }          p++, i++;      }      p -= i;      return start;  }  int main(void){      char str[]="ABCDEF";      char \*ptr = str;      char sub[]="CDEF";      char \*p2=sub;      int i = SearchSubStr(ptr,p2);      printf("%d=%c\n", i, \*(ptr+i));      return 0;  } | search sub-string using two pointers. |
|  |  |
|  |  |
|  |  |