**Chapter 1 - A Tutorial Introduction**

P9.

快速介绍一下C，以及最基本的要素。

1.1 Getting Started

#include <stdio.h>

告诉编译器要包含标准输入输出库。标准输入输出库会在第7章详细介绍。

1.2 Variables and Arithmetic Expressions

|  |  |  |
| --- | --- | --- |
| Char | A sigle byte |  |
| Short | Short integer |  |
| Long | Long integer |  |

|  |  |  |
| --- | --- | --- |
| %d | 十进制整数 |  |
| %6d | 至少6位宽度 |  |
| %f | 浮点数 |  |
| %.2f | 浮点数，小数点后2位 |  |

1.3 The for statement

1.4 Symbolic Constants

#define

1.5 Character Input and Output

getchar();

putchar();

We can’t use char to be a type big enough to hold EOF in addition to any possible char. We use int.

**File Coping.**

**Character Counting.**

**Line Counting.**

**Word Counting.**

1.6 Arrays

1.7 Functions

A return value of zero implies normal termination.

1.8 Argument – Call by Value

The Story is different for arrays, When the name of an array is used as an argument , the value passed to the function is the location or address of the beginning of the array , there is no copying of array elements.

1.9 Character Arrays

1.10 External Variable and Scope

An external variable must be defined, exactly once, outside any function, this sets aside storage for it. The variable must also be declared in each function that wants to access it; this state the type of the variable.

**Defined – Storage**

**Declared—Access**

In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all extern declarations.

Definition refers to the place where the variable is created or assigned storage;

Declaration refers to places where the nature of the variable is stated but no storage is allocated.

**Chapter 2 - Types, Operators and**

**Expressions**

P35.

2.1 Variable names

2.2 Data types and Sizes

2.3 Constants

Long l or L

Unsigned long ul or UL

Float f or F

Long double l or L

#define VTAB ‘\013’

Or

#define VTAB ‘\xb’

2.4 Declarations

Constant can be applied to the declaration of any variable to specify that its value will not be changed.

Constant double e = 2.71828;

2.5 Arithmetic Operators

2.6 Relational and Logical Operators

2.7 Type Conventions

2.8 Increment and Decrement Operators

2.9 Bitwise Operators

&

|

^

<<

>>

~

2.10 Assignment Operators and Expressions

2.11 Conditional Expressions

2.12 Precedence and Order of Evaluation

**Chapter 3 - Control Flow**

P50.

3.1 Statements and Blocks

3.2 If-Else

3.3 Else-if

3.4 Switch

3.5 Loops While For

3.6 Loops Do While

3.7 Break and Continue

3.8 Goto and Labels

**Chapter 4 - Functions and Program**

**Structure**

P59.

4.1 Basics of Functions

4.2 Functions Returning Non-integers

4.3 External Variables

4.4 Scope Rules

4.5 Header Files

4.6 Static Variables

4.7 Register Variables

4.8 Block Structure

4.9 Initialization

4.10 Recursion

4.11 The C Preprocessor

**File Inclusion**

**Macro Substitution**

**Conditional Inclusion**

**Chapter 5 - Pointers and Arrays**

P78.

**5.1 Pointers and Addresses**

A pointer is a variable that contains the address of a variable.

The unary operator & gives the address of an object,

The unary operator \* is the *indirection* or *dereferencing* operator

**5.2 Pointers and Function Arguments**

The way to obtain the desired effect is for the calling program to pass *pointers* to the values to

swap(&a, &b);

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

void swap(int \*px, int \*py) /\* interchange \*px and \*py \*/

**5.3 Pointers and Arrays**

**5.4 Address Arithmetic**

**5.5 Character Pointers and Functions**

**5.6 Pointer Arrays; Pointers to Pointers**

**5.7 Multi-dimensional Arrays**

**5.8 Initialization of Pointer Arrays**

**5.9 Pointers vs. Multi-dimensional Arrays**

**5.10 Command-line Arguments**

**5.11 Pointers to Functions**

**5.12 Complicated Declarations**

**Char**

Char \*\*argv; // argv : pointer to char

Char (\* (\*x())[] ) ();// function returning pointer to array[] of pointer to function returning char.

Char (\* (\*x[3])() )[5];// a pointer of x[3] pointer to function returning pointer to *array[5] of char;*

在C中，声明的形式为（dcl是declaration的简写）：  
dcl: optional \*'s direct-dcl（含有可选"\*"的direct-dcl）  
direct-dcl name  
                (dcl)  
                direct-dcl()  
                direct-dcl[optional size]  
根据该规则进行逆向解析，就可以得到正确的声明。简化一下：“TypeName Declarator;”其中，Declarator就是声明中的那个名字。当你遇到任何你不能理解的声明时，这个法则就是救命稻草。最简单的例子：  
int aInt;  
这里，int是TypeName，aInt是Declarator。  
再说明一下结合紧密度。在声明/定义变量时，可以使用一些修饰比如“\*”，“[]”，“()”等。“()”（非函数声明中的“()”）具有最高的紧密度，其次才是函数和数组的“()”和“[]”。  
没有“\*”的声明称为直接声明（direct-dcl），而有“\*”称为声明（dcl）。直接声明要比声明结合的紧。分解声明时，先读出结合紧的。在这里，我把direct-dcl称为更紧的结合，它比dcl结合得紧。  
最后，需要你用英语来读出这个声明。对于“[]”，应该读成array of。  
对于复杂的定义，可以将其分解。比如“T (\*p)()”可以分解成“T D1()”，D1读作：function returning T。其中D1是\*p。那么该声明应该读成：p is a poniter to。二者合在一起，就变成了p is a pointer to function returning T，即：p是指向返回T类对象的函数的指针。  
  
再看一个稍微复杂的示例：  
T (\*pfa[])();  
根据dcl和direct-dcl，可以分解成T1 D1（因为结合紧密度），T1， 也就是T ()，那么应该读作：  
D1 is function returning T。  
D1又可以写成T2 D2，其中T2是T1 []，可以分解成T1 D2[]，读作：  
array of D2 function returning T。  
D2是指针，读作：pointers to。那么整个“T (\*pfa[])();”应该读作：  
pfa is an array of pointers to function returning T，即：pfa是个存放指向返回T类对象函数的指针的数组。  
  
换种方式看，在这个例子中，pfa是名字，T(\*[])()是类型。将(\*pfa[])视为一体（direct-dcl），称为D1，那么可以写成T D1()，function returning object of T。在D1中，将\*pfa视为一体（dcl），称为D2，那么\*pfa[]应该是D2[]（direct-dcl），array of D2。合起来就是array of D2 function returning object of T。D2是\*pfa（dcl），替换到前面这句话，结果就是array of pointers to function returning object of T。