1、宏定义的使用

2.1 常量宏

```
#define  PI  3.14
#define  CH  'A'
#define  STRING "hello world"

宏定义在预处理阶段进行替换。

如果在宏定义中使用算数运算时，尽量多加()，使用()将宏定义的值括起来;

// 圆的直径的宏定义
#define D(r)  r + r

// 求圆的周长的宏定义
#define  CIRCLE(r)    D(r) * 3.14

// 宏定义的使用
CIRCLE(3)  ----> 3 + 3 * 3.14           ---> 结果错误
```

```
-------------------------------------
// 圆的直径的宏定义
#define D(r)  ((r) + (r))

// 求圆的周长的宏定义
#define  CIRCLE(r)    D(r) * 3.14

// 宏定义的使用
CIRCLE(3)  ----> ((3) + (3)) * 3.14
```

2.2 宏函数的返回值

```
1. 如果定义的宏函数，只有一条语句，宏定义的返回值直接使用即可。
    #define max_value(a, b)  (((a) > (b)) ? (a) : (b))

    int max = max_value(10, 20);


2. 如果定义的宏函数中有多条语句，最后一条语句的结果被返回，
    并且多条语句需要使用({语句1;语句2;语句2;...;})括起来。
    宏函数的返回值不是通过return返回，而是最后一条语句的结果被返回。
```

```c
#include <stdio.h>
// 使用宏定义的方式实现求两个数的最大值
#define  MAX_VALUE(a,b)  (a > b) ? (a) : (b)

// 带返回值的宏函数
#define  MIN_VALUE(a,b)  ({(a > b) ? (b) : (a);})

// 带返回值的宏函数，最后一个表达式的结果被返回，
// 不是用过return返回
#define  MIN_VALUE1(a,b)  ({int ret; \
                            if (a > b)  \
                                ret = b;  \
                            else    \
                                ret = a;   \
                            ret;})


int main(int argc, const char *argv[])
{
    /*your code*/
    int max=MAX_VALUE(100,300);
    printf("max value = %d\n",max);
    int min=MIN_VALUE(100,300);
    printf("min value = %d\n", min);

    min=MIN_VALUE1(600,500);
    printf("min value = %d\n", min);
    return 0;
}
```

2.3 #ifdef...#else...#endif使用

```
C语言的注释手段：
    //
    /**/
    #if 0/1
    #else
    #endif

#define 宏定义名

#ifdef  宏定义名
    // 如果“宏定义名”被定义了，则此段代码有效
#else
    // 如果“宏定义名”没有被定义了，则此段代码有效
#endif
```

```
#include <stdio.h>
#define DEBUG
int main(int argc, const char *argv[])
{
    /* 操作系统提供的几个调试宏：
     * __FILE__ ：  表示文件名
     * __func__ ：  表示函数名
     * __LINE__ ：  表示行号
    */
    /*your code*/
#ifdef DEBUG
    // 如果“宏定义名”被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#else
    // 如果“宏定义名”没有被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#endif

    return 0;
}
```

2.4 #ifndef...#else...#endif使用

```
#define 宏定义名

#ifndef 宏定义名
    // 如果“宏定义名”没有被定义了，则此段代码有效
#else
    // 如果“宏定义名”被定义了，则此段代码有效
#endif
```

```
#include <stdio.h>
#define DEBUG
int main(int argc, const char *argv[])
```

```
{
    /* 操作系统提供的几个调试宏：
     * __FILE__ ：  表示文件名
     * __func__ ：  表示函数名
     * __LINE__ ：  表示行号
    */
    /*your code*/
#ifndef DEBUG
    // 如果“宏定义名”没有被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#else
    // 如果“宏定义名”被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#endif

    return 0;
}
```

2.5 #if defined...#else...#endif使用

```
#if defined(宏定义名)
    // 如果“宏定义名”被定义了，则此段代码有效
#else
    // 如果“宏定义名”没有被定义了，则此段代码有效
#endif
-----------------------------------
#if !defined(宏定义名)
    // 如果“宏定义名”没有被定义了，则此段代码有效
#else
    // 如果“宏定义名”被定义了，则此段代码有效
#endif
-----------------------------------
#if defined(宏定义名1) && defined(宏定义名2)
    // 如果“宏定义名1”和“宏定义名2”都被定义了，则此段代码有效
#else
    // 如果“宏定义名1”和“宏定义名2”只要有一个没有被定义，则此段代码有效
#endif
-----------------------------------
#if defined(宏定义名1) || defined(宏定义名2)
    // 如果“宏定义名1”和“宏定义名2”只要有一个被定义了，则此段代码有效
#else
    // 如果“宏定义名1”和“宏定义名2”都没有被定义，则此段代码有效
#endif

#if defined可以进行逻辑运算，而#ifdef和#ifndef不可以进行逻辑运算。
```

```
#include <stdio.h>
#define DEBUG
int main(int argc, const char *argv[])
{
    /* 操作系统提供的几个调试宏：
     * __FILE__ ：  表示文件名
```

```
     * __func__ :  表示函数名
     * __LINE__ :  表示行号
    */
    /*your code*/
#if defined(DEBUG)
    // 如果“宏定义名”被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#else
    // 如果“宏定义名”没有被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#endif

#if !defined(DEBUG)
    // 如果“宏定义名”被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#else
    // 如果“宏定义名”没有被定义了，则此段代码有效
    printf("%s-%s-%d\n", __FILE__, __func__, __LINE__);
#endif

    return 0;
}
```

2.6 宏定义和do...while的结合

进行宏定义时，如果宏值中有多条C语言，使用do{}while(0)包含一下。

案例：do...while循环和宏定义配合使用的案例。

```
#include <stdio.h>

// # : 将参数转换为字符串
// ## :  字符串的拼接

#define  PRINT(str,err)  printf("%s\n",#str);return err

// 宏定义默认要求写到1行，如果分多行进行书写，要求加续行符“\”
#define  PRI_ERR(str,err) do {                    \
                            printf("%s\n", #str);  \
                            return  err;           \
                          } while(0)

#define PRI_DEBUG(str,err)  {printf("%s\n", #str); \
                                return err;        \
                            }

int main(int argc, const char *argv[])
{
    int retValue;
    retValue = putchar('A');
    if (retValue == -1)
    {
#if 0
        printf("put char failed\n");
        return -1;
```

```
#endif
        // 如果if分支只有1条语句可以省略花括号，
        // 但是这里调用的宏定义，展开之后有多条语句，
        // 因此if的{}不可以省略。
        PRINT(put char failed, -1);
    }

    retValue = putchar('B');
    if (retValue == -1)
        PRI_ERR(put char failed, -1);


    retValue = putchar('C');
    if (retValue == -1)
        // PRI_DEBUG(send char failed, -1);  // error
        PRI_DEBUG(send char failed, -1)   // OK
    else
        printf("send char success\n");

    return 0;
}

```

2.7 宏定义和#结合使用

```
在宏定义中使用#时，可以将传递的参数转换为字符串。
```

```
#include <stdio.h>
#define NAME1    "zhangsan"
#define NAME2(n)  "n"
#define NAME3(n)  n

// 在宏定义的值中的变量名前加#，将变量的值转换为字符串
#define NAME4(n)  #n
int main(int argc, const char *argv[])
{
    /*your code*/
    printf("my name is %s\n", "zhangsan");
    printf("my name is %s\n", NAME1);
        // ---> 预处理阶段进行替换为以下结果：
        // printf("my name is %s\n", "zhangsan");

    printf("my name is %s\n", NAME2(zhangsan));
        // ---> 预处理阶段进行替换为以下结果：
        // printf("my name is %s\n", "n");

    printf("my name is %s\n", NAME3("zhangsan"));
        // ---> 预处理阶段进行替换为以下结果：
        // printf("my name is %s\n", "zhangsan");

    printf("my name is %s\n", NAME4(zhangsan));
        // ---> 预处理阶段进行替换为以下结果：
        // printf("my name is %s\n", "zhangsan");
    return 0;
```

```
}
```

2.7 宏定义和##结合使用

```
在宏定义中使用两个##,可以实现字符串的拼接。
```

```c
#include <stdio.h>
// 实现pri和ntf字符串的拼接
#define DEBUG()  pri##ntf("test code\n")
// 将参数a,b对应的字符串进行拼接
#define SHOW(a,b)  a##b("test code\n")

int main(int argc, const char *argv[])
{
    DEBUG();
    // 替换结果  printf("test code\n");

    SHOW(print, f);
    // 替换结果  printf("test code\n");
    return 0;
}
```

2、goto跳转语句的使用

2.1 语法格式

```
1. 语法格式
    goto  Label(标签);    ---> 跳转到Label标签下边的语句开始执行

    Label:
        C语句

2. 注意事项
    1> goto语句只能在函数内进行跳转；
    2> 在应用层开发中一般不使用goto跳转语句；
    3> 在linux驱动开发中，每个驱动中基本上都会看到goto的使用。
```

2.2 参考案例

```
案例1，使用goto实现1-n之间的数据的求和？
    及使用goto跳转语句实现一个循环的代码。
```

```c
#include <stdio.h>

int main(int argc, const char *argv[])
{
    /*your code*/
    int sum = 0;
    int n;
    int i = 1;
```

```
    printf("请输入要给整数给n变量 > ");
    scanf("%d", &n);
loop:
    sum = sum + i;
    i++;
    if (i <= n)
    {
        goto loop;
    }

    printf("sum = %d\n", sum);
    return 0;
}
```

```
案例2：定义一个unsigned int类型的整型变量num，从终端对此变量进行初始化，
    使用goto语句的方式实现将num变量转换为二进制进行输出。

    利用位运算的方式：num & (0x1 << 31)

#include <stdio.h>

int main(int argc, const char *argv[])
{
    /*your code*/
    unsigned int num;
    int i = 31;
    printf("请输入一个整数 > ");
    scanf("%u", &num);
    printf("0b");
loop:
    // if (num & (1 << i))
    //     printf("1");
    // else
    //     printf("0");
    printf("%d", num & (1 << i) ? 1 : 0);
    i--;
    if (i >= 0)
        goto loop;

    putchar('\n');
    return 0;
}
```

3、在堆区分配内存空间

3.1 malloc和free函数的使用

```
man 3 malloc/free查看函数的帮助手册

#include <stdlib.h>

void *malloc(size_t size);
```

```
功能：手动在堆区分配内存空间
参数：
    @ size : 分配堆区内存空间的大小
        以字节为单位进行分配
返回值：
    成功返回分配的堆区空间的首地址，
    失败返回NULL

void free(void *ptr);
功能：手动释放堆区的空间
参数：
    @ ptr : 释放的堆区空间的首地址
返回值：
    无
```

3.2 参考案例

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, const char *argv[])
{
    /*your code*/
    // 回顾：定义指针变量如何进行初始化
#if  0
    int *p = NULL;
    int a = 100;    // 变量a在栈区分配空间
    p = &a;   // 指针变量指向栈区的空间

    char arr[10] = "hello"; // 字符数组，在栈区分配空间
    char *str_p = arr;   // 字符指针指向栈区空间

    char *str_p2 = "world"; // 字符指针指向字符串对应的常量区
#endif

    /*
    * 定义指针类型的变量，指向堆区的空间，
    * 在堆区分配的空间，使用malloc手动在堆区分配空间，
    * 使用堆区空间完成之后，需要使用free手动释放堆区空间，
    * 即堆区的空间要求手动分配，手动释放。
    * 如果堆区的空间没有手动的释放，当程序结束之后，
    * 系统也会帮助我们回收堆区的空间。
    */

    // 1. 定义指针类型的变量，使用malloc在堆区分配空间
    int *m_p = NULL;
    m_p = (int *)malloc(sizeof(int));   // m_p指向堆区空间
    if (m_p == NULL)
    {
        printf("malloc failed!\n");
        printf("%s:%s:%d\n", __FILE__, __func__, __LINE__);
        return -1;
```

```
    }

    // 2. 对m_p指向的堆区空间进行初始化
    *m_p = 10086;   // 对堆区的空间赋值

    printf("*m_p = %d\n", *m_p);

    // 3. 使用free释放堆区的空间
    free(m_p);

    // 4. 释放完成之后一定要将指针变量赋值为NULL,
    //      防止野指针的出现
    m_p = NULL;

    return 0;
}
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, const char *argv[])
{
    /*your code*/
    // 1. 定义指针类型的变量，指向堆区的空间
    char *name = (char *)malloc(sizeof(char) * 20);
    if (name == NULL)
    {
        printf("malloc failed!\n");
        printf("%s:%s:%d\n", __FILE__, __func__, __LINE__);
        return -1;
    }
    printf("堆区 name = %p\n", name);
    // 2. 对name指向的堆区空间初始化
    // name = "zhangsan";  // name指向字符串对应的常量区
    // printf("常量区 name = %p\n", name);
    strcpy(name,"zhangsan");
    printf("%s\n", name);

    // 3. 释放堆区的空间
    printf("释放之前 name = %p\n", name);
    // free释放堆区的空间，释放完成堆区的空间之后，
    // 并不会将name中的值清0(NULL)
    free(name);
    printf("释放之后 name = %p\n", name);

    // 释放name指向的堆区空间之后，如果name没有指向NULL，
    // 依然可以通过name访问对应的堆区空间，测试访问的就是
    // 非法的内存的空间。
    // 此时name就会变成一个野指针，有可能会出现段错误。
    // 当name指向的空间被释放之后，如果再次被分配，
    // 程序的运行结果就不可预知。
#if 0
    // 没有将name指向NULL,依然可以通过name访问对应的
    // 内存空间，并且运行程序不会报错
    strcpy(name, "lisi");
```

```
    printf("name = %s\n", name);
#else
    // 如果将name指向NULL,再次对name进行操作，
    // 编译不会报错，运行程序会出现段错误，
    name = NULL;
    strcpy(name, "lisi");
    printf("name = %s\n", name);

#endif
    return 0;
}
```

```
goto的使用场合：
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, const char *argv[])
{
    /*your code*/
    int *i_p = NULL;
    short *s_p = NULL;
    char *c_p = NULL;

    // 对以上三个指针变量分别分配堆区的空间

    if ((i_p = (int *)malloc(sizeof(int) * 10)) == NULL)
    {
        printf("malloc int failed\n");
        return -1;
    }

    if ((s_p = (short *)malloc(sizeof(short)*10)) == NULL)
    {
        printf("malloc short failed\n");
        free(i_p);
        i_p = NULL;
        return -2;
    }

    if ((c_p = (char *)malloc(sizeof(char)*10))==NULL)
    {
        printf("malloc char failed");
        free(s_p);
        s_p = NULL;
        free(i_p);
        i_p = NULL;
        return -3;
    }

    free(c_p);
    c_p = NULL;
    free(s_p);
    s_p = NULL;
    free(i_p);
    i_p = NULL;
```

```
    return 0;
}
-------------------------------------
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, const char *argv[])
{
    /*your code*/
    int *i_p = NULL;
    short *s_p = NULL;
    char *c_p = NULL;
    int retVal;
    // 对以上三个指针变量分别分配堆区的空间

    if ((i_p = (int *)malloc(sizeof(int) * 10)) == NULL)
    {
        printf("malloc int failed\n");
        retVal = 1;
        goto ERR1;
    }

    if ((s_p = (short *)malloc(sizeof(short)*10)) == NULL)
    {
        printf("malloc short failed\n");
        retVal = 2;
        goto ERR2;
    }

    if ((c_p = (char *)malloc(sizeof(char)*10))==NULL)
    {
        printf("malloc char failed");
        retVal = 3;
        goto ERR3;
    }

    free(c_p);
    c_p = NULL;
    free(s_p);
    s_p = NULL;
    free(i_p);
    i_p = NULL;


    return 0;
ERR3:
    free(s_p);
    s_p = NULL;
ERR2:
    free(i_p);
    i_p = NULL;
ERR1:
    return -retVal;
}
```

```
练习题：
    定义一个int *类型的指针变量，使用malloc在堆区分配sizeof(int) * 10大小的空间，
    使用从终端输入的方式对malloc分配的堆区空间初始化，
    使用冒泡排序的方式对堆区空间的数据进行排序。
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define LEN 10

int *malloc_space(int len)
{
    int *p = (int *)malloc(sizeof(int) * LEN);
    if (p == NULL)
    {
        printf("malloc failed!\n");
        return NULL;
    }
    return p;
}

void init_malloc_space(int *arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        scanf("%d", &arr[i]);
    }
}

void bullue_sort(int *arr, int len)
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (*(arr + j) > *(arr + j + 1))
            {
                int tmp;
                tmp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = tmp;
            }
        }
    }

}
void print(int *arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        printf("%d ", arr[i]);
    }
    putchar('\n');
}

void destory_malloc_space(int **pp)
```

```c
{
    free(*pp);
    *pp = NULL;
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 1. 定义指针变量，并分配堆区空间
    int *arr_p = malloc_space(LEN);
    // 2. 初始化
    init_malloc_space(arr_p, LEN);

    // 3. 冒泡排序
    bullue_sort(arr_p, LEN);
    // 4. 打印排序之后的结果
    print(arr_p, LEN);

    // 5. 释放堆区空间
    destory_malloc_space(&arr_p);

    return 0;
}
```

4、多文件的编程

4.1 多文件编程相关的概念

```
    在实际开发过程中，每个综合的项目都可分成很多个不同的功能模块，
不可能将所有的代码都写到一个.c文件中，而是采用多文件编程的方式，
将具有不同功能的代码写到对应的.c文件中，最后只需要在main函数对应的.c
文件中调用其他.c文件中定义的函数即可。

比如：温湿度报警的项目(led,蜂鸣器,温湿度传感器)
    main.c
    led.c       ---> led.h
    beep.c      ---> beep.h
    temp-hum.c  ---> temp_hum.h

    .h 文件被称为头文件，主要书写的函数的声明，宏定义，变量的声明
    .c 文件被称为源文件，主要书写的函数的定义，变量的定义

头文件中必须先写防止头文件重复包含的机制。
    #ifndef  __头文件名大写_H__
    #define  __头文件名大写_H__
        // 宏定义
        // 变量的声明
        // 函数的声明
    #endif

    解释：当第一次包含头文件时，#ifndef判断"__头文件名大写_H__"
        没有被定义，则以下的代码有效，然后使用#define定义
        "__头文件名大写_H__".
```

```
        当第二次再次包含此头文件时，#ifndef判断"__头文件名大写_H__"
        被定义了，则以下代码将无效。
```

4.2 编写一个多文件编程的代码

main.c文件

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// 使用""包含自己编写的头文件，可以在文件前添加对应的头文件的路径
#include "./cal.h"
int main(int argc, const char *argv[])
{
    /*your code*/
    printf("100 + 200 = %d\n",cal_func(100,200,'+', add_func));
    return 0;
}
```

cal.c文件

```c
int add_func(int a, int b)
{
    return (a+b);
}
int sub_func(int a, int b)
{
    return (a-b);
}
int mul_func(int a, int b)
{
    return (a*b);
}
int div_func(int a, int b)
{
    return (a/b);
}

int cal_func(int a, int b, char oper, int (*func_p)(int a, int b))
{
    int ret = 0;
    switch(oper)
    {
    case '+':
        ret = func_p(a, b);
        break;
    case '-':
        ret = func_p(a, b);
        break;
    case '*':
        ret = func_p(a, b);
        break;
    case '/':
```

```
        ret = func_p(a, b);
        break;
    }
    return ret;
}
```

cal.h文件

```
#ifndef __CAL_H__
#define __CAL_H__
int add_func(int a, int b);
int sub_func(int a, int b);
int mul_func(int a, int b);
int div_func(int a, int b);

int cal_func(int a, int b, char oper, int (*func_p)(int a, int b));


#endif  // __CAL_H__
```

```
编译多文件的代码：
    gcc main.c  cal.c
```

```
练习题：将上节课的malloc的代码使用多文件编程实现。
```

main.c文件

```
#include <stdio.h>
#include "malloc.h"
int main(int argc, const char *argv[])
{
    /*your code*/
     // 1. 定义指针变量，并分配堆区空间
    int *arr_p = malloc_space(LEN);
    // 2. 初始化
    init_malloc_space(arr_p, LEN);

    // 3. 冒泡排序
    bullue_sort(arr_p, LEN);
    // 4. 打印排序之后的结果
    print(arr_p, LEN);

    // 5. 释放堆区空间
    destory_malloc_space(&arr_p);
    return 0;
}
```

malloc.c文件

```
#include "malloc.h"

int *malloc_space(int len)

```

```c
{
    int *p = (int *)malloc(sizeof(int) * LEN);
    if (p == NULL)
    {
        printf("malloc failed!\n");
        return NULL;
    }
    return p;
}

void init_malloc_space(int *arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        scanf("%d", &arr[i]);
    }
}

void bullue_sort(int *arr, int len)
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (*(arr + j) > *(arr + j + 1))
            {
                int tmp;
                tmp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = tmp;
            }
        }
    }

}
void print(int *arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        printf("%d ", arr[i]);
    }
    putchar('\n');
}

void destory_malloc_space(int **pp)
{
    free(*pp);
    *pp = NULL;
}
```

malloc.h文件

```c
#ifndef __MALLOC_H__
#define __MALLOC_H__

```

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

#define LEN 10

int *malloc_space(int len);
void init_malloc_space(int *arr, int len);
void bullue_sort(int *arr, int len);
void print(int *arr, int len);
void destory_malloc_space(int **pp);

#endif  // __MALLOC_H__
```

5、6种存储类型

5.1 定义变量的格式

```
存储类型   数据类型    变量名 = 初始值;

```

5.2 6种存储类型

```
auto const extern  static  register  volatile
```

5.3 auto存储类型 --> 自动类型

```
auto ：  自动存储类型

非自动类型的变量：全局变量，使用static修饰的全局变量，
    使用static修饰的局部变量都属于非自动类型的变量，不可以使用auto修饰。

自动类型的变量：局部变量属于自动类型的变量，可以使用auto进行修饰，
    即使不加auto修饰，默认也是自动类型的变量，因此在开发中基本不使用auto.
```

```c
#include <stdio.h>
// 全局变量，全局变量不属于自动存储类型的变量
// 不可以加auto进行修饰
// auto  int global;   // error
// auto static int  global_s;  // error


int main(int argc, const char *argv[])
{
    /*your code*/
    // 局部变量
    auto  int local = 0;  // ok,省略auto一样属于自动存储类型的变量
```

```
    // auto static int local_s;  // error,

    return 0;
}
```

5.4 const存储类型 --> 常量

```
const : 只读的变量
const可以修饰普通类型的变量，也可以修饰指针类型的变量。

const可以用来修饰函数的形参和返回值
    int strlen(const char *s)
    {
        int len = 0;
        while (*s != '\0') {
            len++;
        }
        return len;
    }
    char *strcpy(char *s1, const char *s2);
    char *strcat(char *s1, const char *s2);
    int strcmp(const char *s1, const char *s2);
```

```c
#include <stdio.h>
// 1. const 修饰全局变量
const int global = 1000;
// const修饰的全局变量存储在只读数据段(常量区)，
// 不可以被修改，即使通过指针也不可以被修改。
// 定义常量时，必须在定义的时候进行初始化。

int main(int argc, const char *argv[])
{
    /*your code*/
    // 2. const修饰局部变量
    const  int local = 2000;
    // const修饰的局部变量在栈区分配空间，
    // 不可以通过常量的变量名本身进行修改，
    // 但是可以通过指针修改使用const修饰的局部变量。
    // local = 3000;
    int *p = &local;
    printf("修改之前，local = %d\n", local);
    *p = 3000;       // 通过指针修改使用const修饰的局部变量。
    printf("修改之后，local = %d\n", local);


    // 3. const修饰指针变量
    // const修饰*p1 , 不可以修改*p1指向的空间的内容
    int a = 10086;
    int b = 10010;
    int const *p1 = &a;
    // *p1 = 10010;      // 不可以修改*p1指向的空间的内容
    p1 = &b;         // 可以修改p1的指向
```

```c

    // const修饰*p2 , 不可以修改*p指向的空间的内容
    const int *p2 = &a;
    // *p2 = 10010;      // 不可以修改*p2指向的空间的内容
    p2 = &b;         // 可以修改p2的指向

    // const修饰p3 , 不可以修改p3的指向
    int * const p3 = &a;
    *p3 = 10010;     // 可以修改*p3指向的空间的内容
    // p3 = &b;         // 不可以修改p3的指向

    // 第一个const修饰*p4, 不可以修改*p4指向的空间的内容
    // 第二个const修饰P4, 不可以修改p4的指向
    const int *const p4 = &a;
    // *p4 = 10010;     // 不可以修改*p3指向的空间的内容
    // p4 = &b;         // 不可以修改p3的指向
    return 0;
}
```

5.5 extern存储类型 --> 外部的

```
extern : 外部的，在其他文件中定义的全局变量或者函数如果想在当前文件中使用，
         需要使用extern进行声明，表示在其他文件中定义的。
1. 修饰全局变量：表示此全局变量是在其他.c文件中定义的。

2. 修饰函数：表示此函数是在其他.c文件中定义的。
```

main.c文件

```c
#include <stdio.h>

// 在main函数中调用extern.c文件中定义的全局变量或者函数时，
//  需要使用extern进行声明
extern int global;  // 注，使用extern声明全局变量时，不要进行赋值的操作
// extern int global = 10010;  // 错误的

// 对于函数来说，加不加extern效果是一样的。
extern void print(void);


int main(int argc, const char *argv[])
{
    /*your code*/
    printf("main.c::global = %d\n", global);
    printf("main.c::&gloabl = %p\n", &global);

    print();
    return 0;
}
```

extern.c文件

```c
#include <stdio.h>
// 定义全局变量
int global  = 10086;

// 定义函数
void print(void)
{
    printf("extern.c::global = %d\n", global);
    printf("extern.c::&global = %p\n", &global);
}
```

5.6 static存储类型 --> 静态

```
static : 静态的存储类型
1. 修饰全局变量：不可以被外部的其他文件使用

2. 修饰函数：不可以被外部的其他的文件使用

3. 修饰局部变量：延长变量的生命周期到整个程序结束，
    使用static修饰的局部变量如果初始化，则在.data段分配空间；
    如果使用static修饰的局部变量没有初始化，则在.bss段分配空间，
    并且初始化为0.
    使用static修饰的局部变量，只在函数第一次调用时被初始化一次，
    后边再次调用此函数，则不在执行初始化的代码
```

main.c文件

```c
#include <stdio.h>
// 在main.c文件中不可以调用static.c文件中，
// 使用static修饰的全局变量和函数，以下编译报错
// extern int global;
// extern void print(void);

// 使用static的好处，可以在不同的文件中定义
// 全局变量名和函数名相同的变量和函数

// 定义的global变量和print函数和static.c文件的变量和函数
// 不是同一个变量和函数，只是名字相同而已。
int global = 10086;

void print(void)
{
    printf("main.c::global = %d\n", global);
    printf("main.c::&global = %p\n", &global);
}

// static修饰局部变量，延长局部变量的生命周期
int auto_add(void)
{
    // 变量i，在.data段分配空间
    static int i = 100; // 只在第一次调用此函数时，对i进行赋值操作
    i++;
```

```
    return i;
}
int auto_sub(void)
{
    // 变量i, 在.bss段分配空间，默认初始化为0
    static int i;
    i--;
    return i;
}
int main(int argc, const char *argv[])
{
    /*your code*/
    // 使用的时main.c文件中的global变量和print函数，
    // 而不是使用的static.c文件中的global变量和print函数
    printf("main.c::global = %d\n", global);
    printf("main.c::&global = %p\n", &global);
    print();

    int a = 0;
    a=auto_add();   // 101
    a=auto_add();   // 102
    a=auto_add();   // 103
    a=auto_add();   // 104
    printf("a = %d\n", a);  // 104

    int b = 0;
    b=auto_sub();   // -1
    b=auto_sub();   // -2
    b=auto_sub();   // -3
    b=auto_sub();   // -4
    printf("b = %d\n", b);  // -4
    return 0;
}
```

static.c文件

```
#include <stdio.h>

// 定义全局变量，和函数使用static修饰，
// 使用static修饰的函数不可以被其他文件调用
static int global = 10086;

static void print(void)
{
    printf("static.c::global = %d\n", global);
    printf("static.c::&global = %p\n", &global);
}


```

5.7 register存储类型 --> 寄存器

```
```

```
1. register : 寄存器类型的存储变量

2. 定义寄存器类型的变量使用时，和普通变量的用法一样。

3. 对于寄存器类型的变量，不可以进行取地址的运算。
    原因：寄存器的访问是通过编号进行访问的，没有地址。

4. 在实际开发中，尽量不要过多的定义寄存器类型的变量。
    原因：寄存器的个数有限。

5. 定义寄存器类型的变量的读写速度比普通的变量的读写速度快。
    原因：寄存器的读写速度比内存的读写速度快。

    疑问1：为什么寄存器类型的变量不能取地址运算？
    疑问2：为什么寄存器的个数有限？
    疑问3：寄存器读写速度为什么快？
    《ARM体系结构及接口技术》时，详细的分析寄存器。

```

```c
#include <stdio.h>

int main(int argc, const char *argv[])
{
    /*your code*/
    // 1. 定义寄存器类型的变量
    register int r = 1000;
    // 2. 使用寄存器类型的变量
    printf("r = %d\n", r);
    r = 2000;
    printf("r = %d\n", r);

    // 4. 不可以进行取地址的运算
    // printf("&r = %p\n", &r);  // error
    return 0;
}
```

5.8 volatile存储类型 --> 易变的

```
volatile : 易变的
底层课程时讲解volatile关键字的使用。
```

6、作业

```
理解今天的代码，自己动手写一遍。
C语言的试卷，第7套，前20题
```

7、明天的授课内容

```
1. typedef  重新定义新的数据类型
2. 结构体(结构体指针，结构体数组，结构体指针数组，结构体数组指针)

    学生成绩管理系统。
```