第七章 指针(三)

十三、内存分配

十四、常量与指针

十五、引用类型

十六、变量存储方式

十七、参数传递和返回值

十八、函数指针 十九、void*指针

十三、内存分配

13.1 静态内存和动态内存分配

内存分配是指在程序执行的过程中分配或者回收存储空间的分配内存的方法。内存分配方法有**静态内存分配**和**动态内存分配**两种。

- **静态内存分配**：静态分配是在编译时由系统完成的，不占用CPU资源。静态分配内存需要在编译前确定内存块的大小，如声明数组 int a[5];

- **动态内存分配**：动态分配内存是在运行时完成的，动态内存的分配与释放需要占用CPU资源，由程序员决定何时分配以及分配的空间大小；动态内存分配不像数组等静态内存分配方法那样需要预先分配存储空间，而是由系统根据程序的需要即时分配空间。

1.2 五大内存区

在C语言中，内存分成5个区，它们分别是堆、栈、全局/静态存储区、常量存储区和代码区。

⑴ 栈区：编译器自动分配释放，需预先知道分配的内存大小，如局部变量，函数形参等；

⑵ 堆区：动态内存分配，运行时分配，由程序员手动分配和释放，它可以动态决定要分配的内存大小。使用malloc或new分配；

⑶ 常量区：存放常量，不允许修改其值。如常量字符串；

⑷ 全局/静态存储区：存储全局变量和静态变量；

⑸ 代码区：存放函数体的二进制代码。

1.3 动态内存分配malloc

如果程序员需要在堆中分配一块内存空间，需使用malloc函数。

函数原型：

```
void * malloc (size_t size);
```

- 在堆区请求一块内存空间
- size请求的空间大小
- malloc函数返回一个指针指向请求的堆空间
- 如果请求失败，则返回为NULL

Q：如何在堆中分配一块空间来存储一个浮点型数据?

For example:

```
float* pFloat = (float *)malloc (sizeof(float));
```

- 在堆区分配4个字节来存储float类型数据;
- malloc函数返回一个指针指向分配的堆空间;
- malloc函数的返回值原型为void* 所以加上显式类型转换;
- 在堆上分配的空间是没有名字的，只有该空间的地址;
- 该地址保存在另外一个指针变量里，只能通过指针间接访问这块内存，而无法通过变量名访问。

通过指向堆空间的指针来访问该内存空间:

```
*pFloat = 3.14159;
```

空间使用完成后必须由程序员手动释放:

```
free(pFloat);
```

注意：使用malloc分配的堆空间，必须使用free函数来释放，编译器不会自动释放。

```
float* pFloat = (float *)malloc (sizeof(float));

*pFloat = 3.14;

free(pFloat);

pFloat = NULL;
```

分配空间

使用空间

释放空间

指针置空

malloc和free的次数要对应

- 如果malloc > free则内存泄露；
- 如果malloc < free则二次删除，破坏内存；

注意：free(pFloat)是释放指针所指向的堆空间，而不是指针变量本身，那么指针pFloat将会变成野指针，所以应将该指针置空，是程序员的一个好习惯。

十四、常量与指针

- **常量指针**：不能通过指针修改指针所指向的变量的值。但是指针可以指向别的变量。

```
int a = 5;
const int *p = &a;
*p = 20;          //不能通过指针修改指向的变量的值
                  ✗

int b = 10;
p = &b;           //指针可以指向别的变量
                  √
```

- 指向常量的指针（指针常量）：指针常量的值不能被修改，即不能存一个新的地址，不能指向别的变量。但是可以通过指针修改它所指向的变量的值。

```
int a = 5;
int *const p = &a;
*p = 20;    √      //可以通过指针修改指向的变量的值

int b = 10;
p = &b;     ×      //指针不能指向别的变量
```

总结：

(1) const修饰谁，谁不可变；如 int *const p; 修饰指针p则p不可变；

(2) 从前向后读（翻译），const int *p; 常量指针；

(3) 谁在前谁不可变，如指针常量则指针不可变，常量指针则指针指向的值不可变；

十五、引用类型

引用类型是C++标准提出来的一种新型的数据类型,但是不适用于C标准 。

引用类型的作用是为变量起一个别名，引用类型必须在声明时初始化。如黑旋风是李逵的别名。假如有一个变量i，想给它起一个别名，可以这样写:

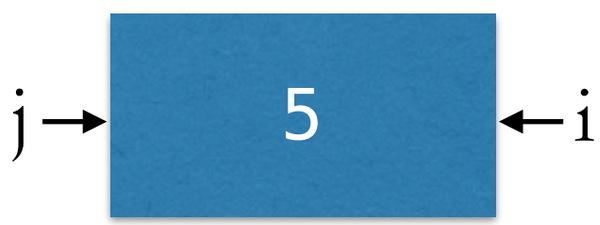
```
int i = 5;
int & j = i;
```

```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    int i = 5;
    int &j = i;//j是i的引用
    printf("%d %d\n",i,j);
    j = 10;
    printf("%d %d\n",i,j);
    printf("%p %p\n",&i,&j);//地址一样
    return 0;
}
```

程序运行结果如下:

```
5 5
10 10
0x7fff5fbff80c
0x7fff5fbff80c
```

上述代码可知，i和j实际是同一个变量，j是i的别名，改变j就是改变i，打印输出i和j的地址一样，也就是说它们在内存中是共享同一块空间。



C++中一个引用类型的变量可以被const修饰,称为常量引用 ， const int &j = i;

常量引用指的是不能修改引用的值，但引用所指向的变量的值自身可以作修改。

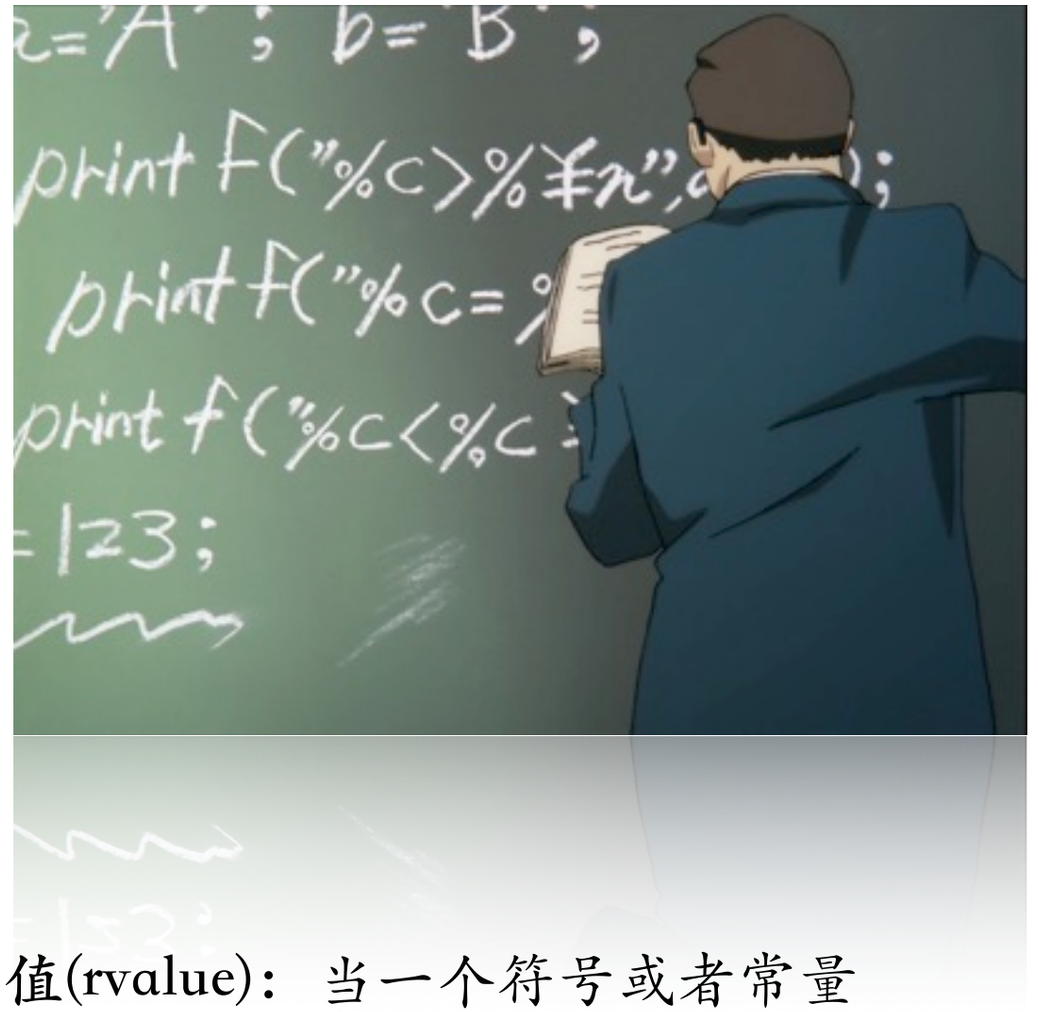
```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    int i = 5;
    const int &j = i;//j是i的引用
    j = 20; ✖  Read-only variable is not assignable

    i = 20; √
    return 0;
}
```

十六、变量存储方式

左值(lvalue) 是C语言/C++语言等类C语言中的一类表达式。“左”(left)的意思是指可以放在赋值符号“=”的左边；



右值(rvalue)：当一个符号或者常量放在操作符右边的时候，计算机就读取他们的“右值”，也就是其代表的真实值，右值相当于数据值；

16.1 变量的作用域

变量的访问范围即变量的作用域。C语言规定程序中一个变量可以有三种不同的作用域:

- **文件域:**

在整个源程序文件中声明的一个标识符(包括变量名和函数名),在整个件中都可以被访问。例如全局标识符,其作用范围是整个文件。

- **函数域:**

在一个函数原型中声明的一个标识符,在整个函数中都可以被访问。例如函数的形式参数标识符,其作用范围是该函数中。

- **语句块域:**

在一个程序结构块中声明的标识符,在该程序结构块中可以被访问，如函数体内的局部变量。

16.2 局部变量和全局变量同名

当在块内部声明了一个与外部标识符(变量名)同名的标识符时，外部标识符可以被临时隐藏。如果想访问被临时隐藏全局变量，要使用全局域操作符::

```
//需要C++源文件支持
#include <stdio.h>

int x = 0;      // a global variable x
int main()
{
    int x = 1;          // a local variable x局部变量, 块域
    x = 2;          // the local x
    {
        int x;      // another x declared in an inner block
        x = 10;         // the inner x
        printf(“%d\n”,::x);
    }
    printf("%d\n",x);
    return 0;
}
```

程序运行结果如下:

```
0
2
```

16.3 链接

链接：把多个目标文件合并成单一的可执行文件。

定义在一个源文件中的全局变量或全局函数可以被另一个源文件引用，需要使用extern关键字。

```
//
//  hello.c  源文件一
//
//  Created by Mr_lee on 16/1/15.
#include <stdio.h>

int p = 100;

void func()
{
    printf("Hello world!");
}
```

```
//
//  main.c 源文件二
//
//  Created by Mr_lee on 16/1/13.
#include <stdio.h>
extern int p;//表示p是在其它文件中定义的
extern void func();
int main()
{
    printf("p = %d\n",p);
    return 0;
}
```

16.4 存储类型

局部变量和函数形参所在的内存空间，叫栈（**stack**）。

- 栈上的变量称为自动变量，自动变量的内存空间的开辟和回收，是系统管理。

- 自动变量定义时，系统为之开辟内存空间；自动变量离开作用域时，系统自动回收所占内存空间。

- 当多次调用一个函数时，函数内部的局部变量和形参，将会在每一次调用时，都自动创建和回收。

- 自动变量的生命周期，只是在函数执行期间，作用域为函数体内。

静态局部变量：局部变量声明时加上static关键字

- 函数返回后，静态局部变量的内存空间不会被回收；
- 当下一次调用该函数时，静态局部变量依然保持上一次调用退出时的值。
- 静态局部变量的作用域不变，依然是在函数内部。
- 静态全局变量在内存里，保存在静态存储区。
- 静态存储区里变量的生命周期，是整个程序运行期间。

```c
#include <stdio.h>
void func()
{
    static int x = 1;
    printf("x = %d\n",x);
    x++;
}

int main()
{
    func();  func();  func();
    return 0;
}
```

程序运行结果如下：

```
x = 1
x = 2
x = 3
```

静态全局变量：全局变量声明时加上static关键字

- 生命周期不变：依然是整个程序运行期间；
- 静态全局变量，也存储在静态存储区；
- 作用域缩小：只限于当前的源文件，不能被其它源文件引用，不能加extern关键字。

变量类型	内存区域	作用域	生命周期
局部变量和形参	栈	块作用域和函数域	进入程序块（或函数）创建；离开程序块（或函数）销毁
静态局部变量	静态存储区	块作用域和函数域	整个程序运行期间函数退出后依然保持变量的值
全局变量	全局区/静态存储区	文件域 加entern可被其他源文件访问	整个程序运行期间
静态全局变量	静态存储区	仅限于文件域 不能再其他源文件加entern访问	整个程序运行期间

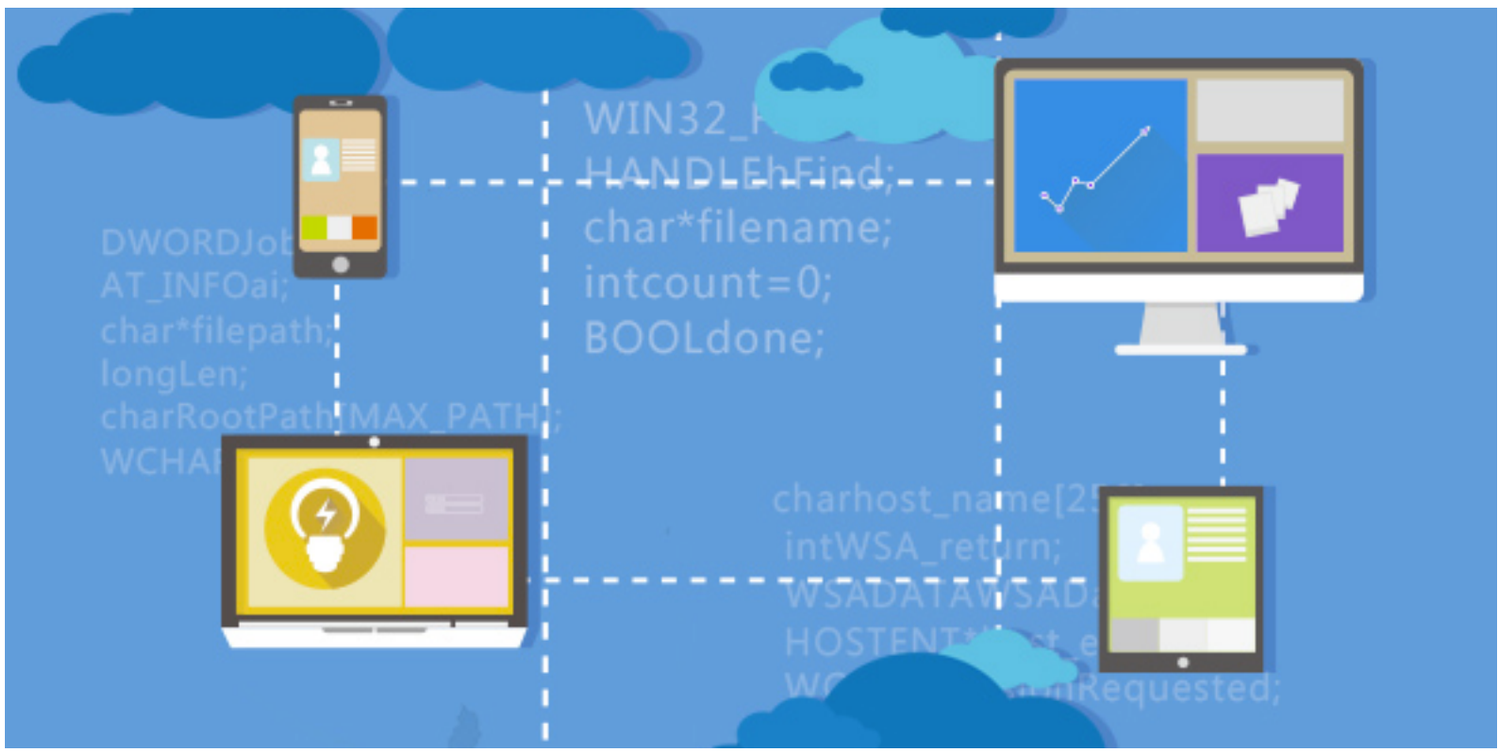
十七、参数传递和返回值

函数传参的三种方式:

(1) 传值：形参是实参的一份拷贝，单向传递，形参的改变不会影响实参;

(2) 传指针：通过形参间接改变实参所指向的变量的值;

(3) 传引用：形参就是实参，改变形参就是改变实参;



- 传值：形参的改变不会影响实参。

```c
#include <stdio.h>

void  swap(int a, int b);

int main(void)
{
    int x,y;
    printf("请输入x和y的值:");
    scanf("%d%d",&x,&y);
    swap (x,y);
    printf("调用函数后的值为:%d,%d\n",x,y);
    return 0;
}
void swap (int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

程序运行结果如下：

```
请输入x和y的值:3 5
调用函数后的值为:3,5
```

- 传指针：通过对形参解引用间接改变实参指向的变量的值。

```
#include <stdio.h>

void  swap(int *a, int *b);

int main(void)
{
    int x,y;
    printf("请输入x和y的值:");
    scanf("%d%d",&x,&y);
    swap (&x,&y);
    printf("调用函数后的值为:%d,%d\n",x,y);
    return 0;
}
void swap (int *a, int *b)//a指向x,b指向y
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

程序运行结果如下：

请输入x和y的值:3 5
调用函数后的值为:5,3

- 传引用（C++）：改变形参就是改变实参。

```
#include <stdio.h>

void  swap(int &a, int &b);

int main(void)
{
    int x,y;
    printf("请输入x和y的值:");
    scanf("%d%d",&x,&y);
    swap (x,y);
    printf("调用函数后的值为:%d,%d\n",x,y);
    return 0;
}
void swap (int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

程序运行结果如下:

请输入x和y的值:3 5

调用函数后的值为:3,5

函数返回值:

一个函数可以返回一个值、一个指针或一个引用。如果传值返回，编译器会创建一个临时变量来接收返回值。

```
#include <stdio.h>

int func()
{
    int x = 5;
    return x;
    //函数返回时,局部变量自动被回收
    //编译器此时会创建一个临时变量来接收x的值
}

int main(void)
{
    int r = func();    //将临时变量的值赋值给r,赋值完成后临时变量被销毁
    printf("%d\n",r);
    return 0;
}
```

注意：不能返回指向局部变量的指针或引用。

十八、函数指针

函数指针是指向函数的指针变量。 因而“函数指针”本身首先应是指针变量，只不过该指针变量指向函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。

C在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样

函数指针的声明方法为:

返回值类型(*指针变量名)([形参列表]);

```
int func(int x); /* 声明一个函数 */
int (*f) (int x); /* 声明一个函数指针 */
```

示例代码：

```c
#include <stdio.h>

int (*fpointer)(int, int);//函数指针要和它指向的函数的参数类型、返回值类型一致。
int add(int, int);
int sub(int, int);
int main()
{
    fpointer = add;
    //函数名本身就是个指针（函数在内存里的地址）。就像数组名本身就是指针。
    printf("%d \n", fpointer(4, 5)); //等效于add(4,5)

    fpointer = sub;//改变指向
    printf("%d \n", fpointer(6, 2)); //等效于sub(6, 2)
    return 0;
}
int add(int a, int b){
    return(a + b);
}
int sub(int a, int b){
    return(a - b);
}
```

十九、void* 指针

void *指针即空类型指针，指针指向的是void类型，空类型指针可以转型为其它类型的指针。不能对void*类型指针解引用。

For example:

```
float* pFloat = (float *)malloc (sizeof(float));
```

```
#include <stdio.h>

void func(void *p){    printf("%p\n",p); }

int main(){
    int a = 5;
    float pi = 3.14;
    func(&a);
    func(&pi);
    return 0;
}
```

The End

渥瑞达·教学部