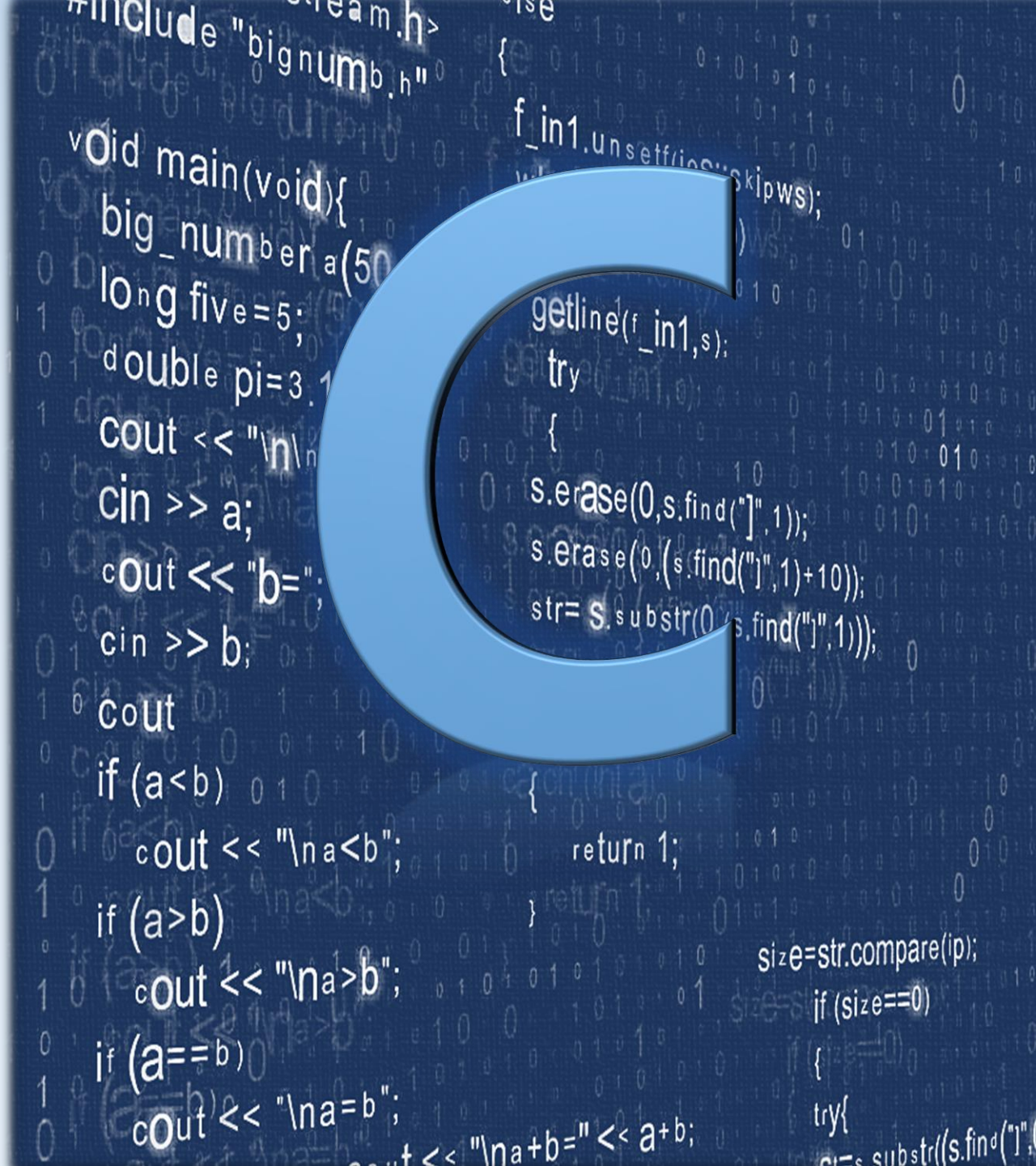


# 《C语言程序设计》

C语言课程组



# 上一讲知识复习

- ◆掌握指针变量的声明方法。
- ◆理解指针变量的两个关键点：
- ◆存放地址
- ◆“捆绑”一块内存空间(它是有类型的)
- ◆掌握多重指针的声明及初始化。
- ◆掌握通过指针访问所指内存空间中数据对象的方法。
- ◆理解const指针。
- ◆了解空指针及通用指针的作用。
- ◆了解指针变量的运算。

# 本讲教学目标

- ◆理解“函数”与“面向过程的编程”的关系。
- ◆掌握函数原型声明与原型定义的方法。
- ◆深入理解参数传递，尤其是“传递地址”的情况。
- ◆掌握嵌套调用、递归函数的分析方法。

# 本章授课内容



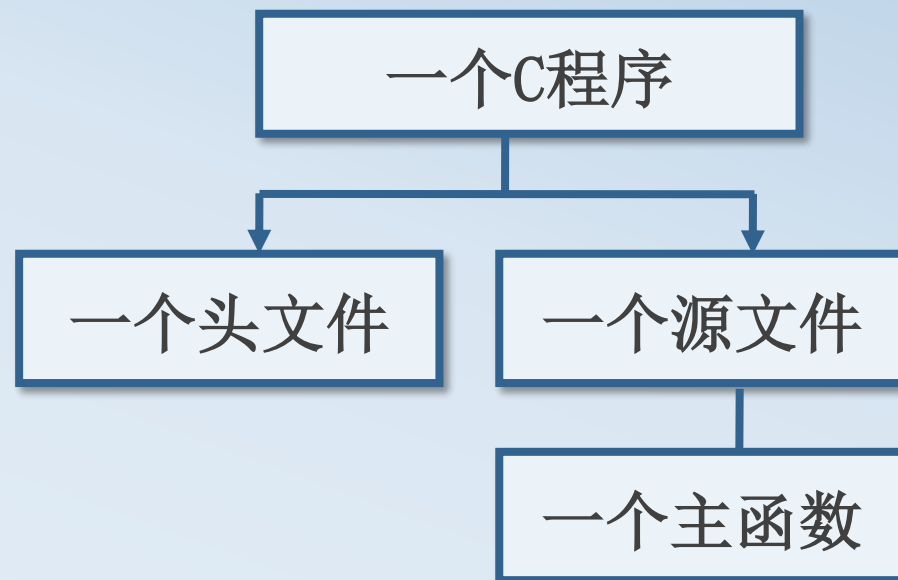
# C程序基本结构

❖ 一个简单的C程序具有如下结构

```
#include <stdio.h>

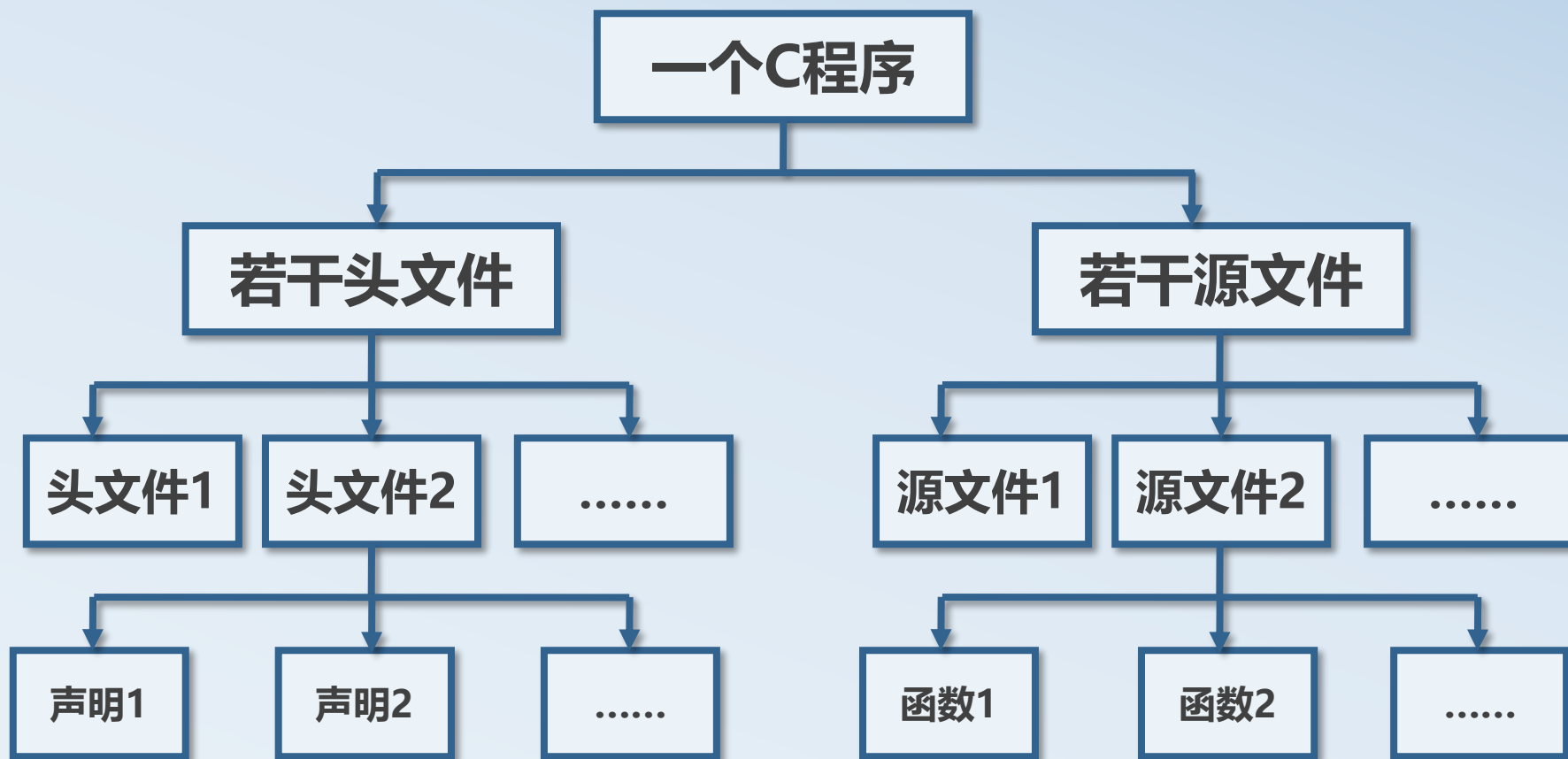
int main(void)
{
    int a=2,b=3;
    printf("%d",a+b);

    return 0;
}
```



# C程序基本结构

❖ 一个典型的C程序具有如下结构





# C程序基本结构

- ❖ 函数是能够完成特定功能的代码集合。
- ❖ 函数的作用
  - ◆ 减少程序的代码量
  - ◆ 代码复用
  - ◆ 使程序具有良好的结构
- ❖ 一个合格的计算机专业学生必须熟练掌握
  - ◆ 函数的定义
  - ◆ 函数的调用
  - ◆ 函数的嵌套和递归

# 本章授课内容



C程序基本结构



函数定义与声明



函数调用



函数嵌套调用和递归调用



生存周期与声明作用域



# 函数定义与声明

## ❖ 函数定义

返回值类型

函数名

形式参数

```
double areaOfCircle (double r)
{
    const double PI = 3.1415926;
    double area;

    area = PI * r * r;

    return area;
}
```

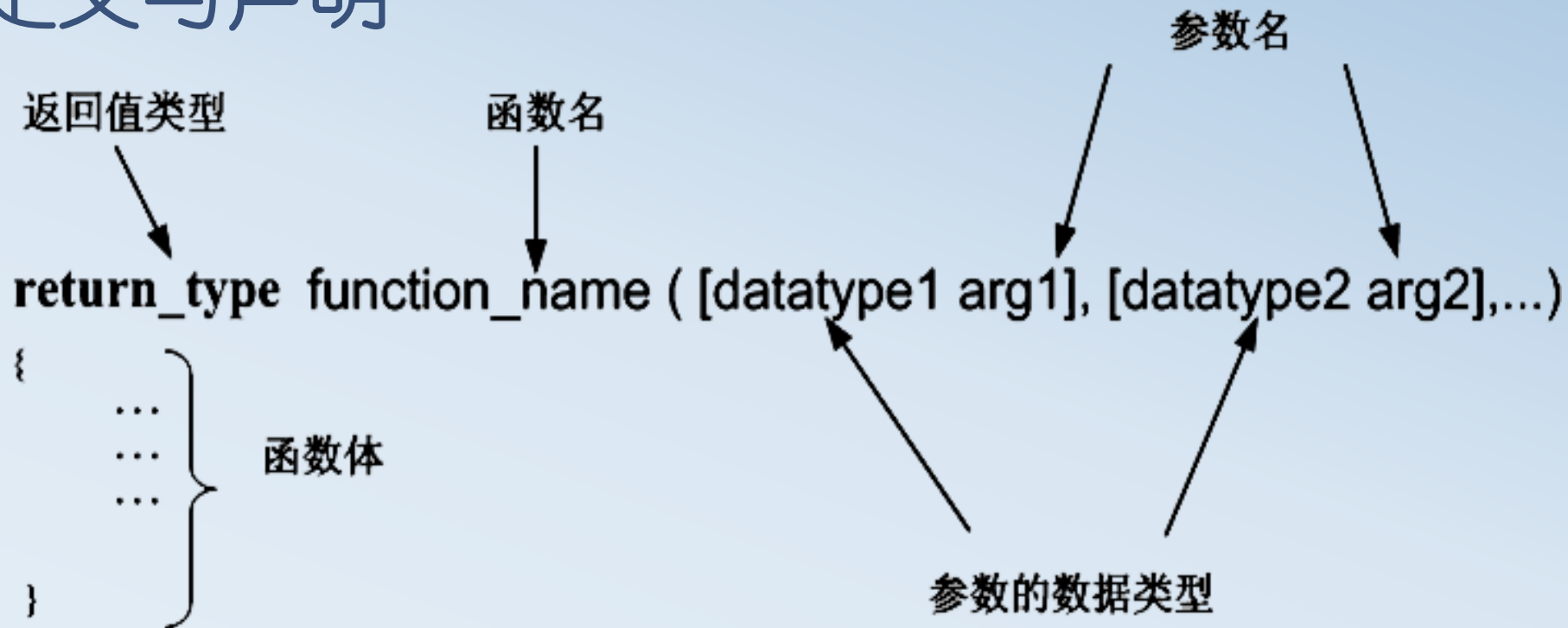
复合语句构成函数体

# 函数定义与声明

❖ 请指出下面函数定义中的各种成分

```
int add(int x, int y)
{
    return x+y;
}
```

# 函数定义与声明



**注意：**

- 1.在函数定义中若有参数，写明参数类型及参数名，称作有参函数.若无参数称作无参函数，需写void.
- 2.若函数有返回值，写明类型，否则写void.

# 函数定义与声明

```
int max(int x, int y)
{
    int m;
    m = x > y ? x : y;

    return m;
}
```

该函数名为max，它有两个int类型的参数，返回值为int类型。在函数体内有三条语句实现了求两个数中较大的数，并将它返回。

# 函数定义与声明

```
void printHello(void)
{
    printf("Hello\n");

    return;
}
```

该函数名为printHello，**无参数**，使用void说明**无返回值**，参数为void，说明无参数。函数体内的语句用于根据产品的价格求折扣后的价格。

# 函数定义与声明

## ❖ 函数定义

1. 函数定义不可嵌套.
2. 在函数定义中若有参数，写明参数类型及参数名，称作有参函数.若无参数称作无参函数，需写void.
3. 若函数有返回值，写明类型，否则写void.
4. 函数的命名规范，从第二个单词起首字母大写.

# 函数定义与声明

- ❖ 形参：定义函数时函数名后面括号中的变量列表
- ❖ 实参：调用函数时函数名后面括号中的表达式列表

```
#include <stdio.h>
int add(int x, int y)
{
    return x + y;
}
int main(void)
{
    int a = 3, b = 4;
    printf("%d\n", add(a, b));
    return 0;
}
```

形参

个数、类型、顺序一致

实参



# 函数定义与声明

## ❖ 形参和实参

- ◆ 形参是局部变量，调用时分配内存，调用结束后系统自动回收形参所占内存。
- ◆ 实参可以是任意合法的常量、变量、表达式。
- ◆ 实参与形参个数一致，类型一致（可能会发生类型转换），顺序一致。

# 函数定义与声明

❖ 阅读下面的代码，指出其中错误

```
int * foo(void)
{
    int a;

    return &a;
}
```

函数返回值**不能是局部变量的地址！**

# 函数定义与声明

## ❖ 返回值

1. 如果函数没有返回值一定要注意写void类型.不需return.
2. 如果函数有返回值的话一定要注意返回值的类型与接收函数返回值变量的类型.
3. 接收函数返回的变量的类型需同返回值类型相同.
4. 一个函数可能有零个或者多个参数，最多只能由一个返回值.

# 本章授课内容



C程序基本结构



函数定义与声明



函数调用



函数嵌套调用和递归调用



生存周期与声明作用域

# 函数调用

❖ 定义一个函数后，就可以在程序中调用这个函数。

◆ 自定义函数必须**先定义后调用**

◆ 库函数调用必须**包含相应的头文件（包含定义）**

❖ 函数调用的形式：**函数名(实参列表)**

**主调函数**

```
#include <stdio.h>
int main(void)
{
    int a = 3, b = 4;
    printf("%d\n", add(a, b));
    return 0;
}
```

**被调函数**

```
int add(int x, int y)
{
    return x + y;
}
```

- ✓ 主调函数通过实参将数据传给被调函数
- ✓ 被调函数通过返回值将数据返回主调函数

# 函数调用

## ❖ 函数调用的三种方式

◆ 函数语句 `add(3, 4);`

◆ 函数表达式 `c = 10 * add(3, 4);`

◆ 函数参数 `printf("%d\n", add(3, 4));`

# 函数调用

## ❖ 函数原型定义

```
返回值类型 函数名(参数类型 参数1, ..., 参数类型 参数n)
{
    /* 语句; */
}
```

## ❖ 函数原型声明

返回值类型 函数名(参数类型 参数1,...,参数类型 参数n);

- ◆ 说明函数的类型和参数的情况，以**保证程序编译时能判断对该函数的调用是否正确**。
- ◆ 函数原型声明标示了函数的**返回值类型、函数名、参数个数、类型和顺序**，是函数的“名片”。
- ◆ 函数原型声明和函数原型定义在**返回类型、函数名、参数个数、类型和顺序**必须完全一致。
- ◆ 函数原型声明不必包含参数的名字，而只要包含参数的类型。



# 函数调用

```
#include <stdio.h>

int add(int x, int y)
{
    return x + y;
}

int main(void)
{
    int a = 3, b = 4;
    printf("%d\n", add( a, b ));

    return 0;
}
```

```
#include <stdio.h>

int add(int x, int y);

int main(void)
{
    int a = 3, b = 4;
    printf("%d\n", add( a, b ));
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

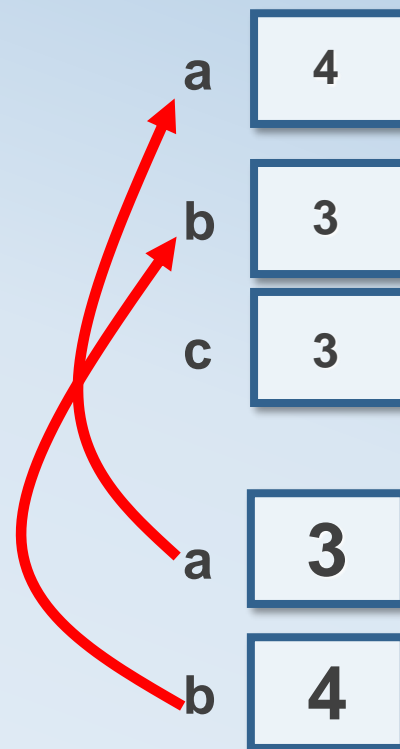
- ❖ 当函数原型定义出现在函数调用前，则原型定义既是原型声明，否则，必须在函数调用前先声明该函数。

# 函数调用

```
void exchangeData(int a, int b)
{
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
int main(void)
{
    int a = 3, b = 4;
    printf ("交换前: ");
    printf(" a = %d b = %d\n", a, b);
    exchangeData(a, b);
    printf ("交换后: ");
    printf(" a = %d b = %d\n", a, b);
    return 0;
}
```

## 内存变化



交换前 : a=3 b=4

交换后 : a=3 b=4

# 函数调用

```
void exchangeData(int *a, int *b)
{
    int c;
    c = *a;
    *a = *b;
    *b = c;
}
```

```
int main(void)
{
    int a = 3, b = 4;
    printf("交换前: ");
    printf("a = %d b = %d\n", a, b);
    exchangeData(&a, &b);
    printf("交换后: ");
    printf("a = %d b = %d\n", a, b);
    return 0;
}
```

内存地址

内存变化

0x0012fe7c

a

0x0012ff60

0x0012fe80

b

0x0012ff54

0x0012fe6c

c

3

0x0012ff60

a

4

0x0012ff54

b

3

交换前 : a=3 b=4

交换后 : a=4 b=3

# 函数调用

- ❖ **值传递**是将实际参数中存放的“值”传递给形式参数，实际参数与形式参数完成“传递接力”后，两者再无干系。在 exchangeData 函数内部无法改变实际参数中的值或通过实际参数的值改变函数中其他变量存储单元的值。
- ❖ **地址传递**是将实际参数中存放的“地址”传递给了形式参数，完成传递接力后，**两者也再无干系**。尽管在 exchangeData 函数内部无法改变实际参数中的“地址”但可以对**该“地址”所指向的单元**进行赋值或取值的操作。

# 函数调用

```
#include <stdio.h>

void printArr(int p[10], const int len)
{
    int i;
    for(i=0; i<len; i++)
    {
        printf("%d ", *(p+i));
    }
    printf("\n");
    return;
}

int main(void)
{
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    printArr(arr, 10);
    return 0;
}
```

# 函数调用

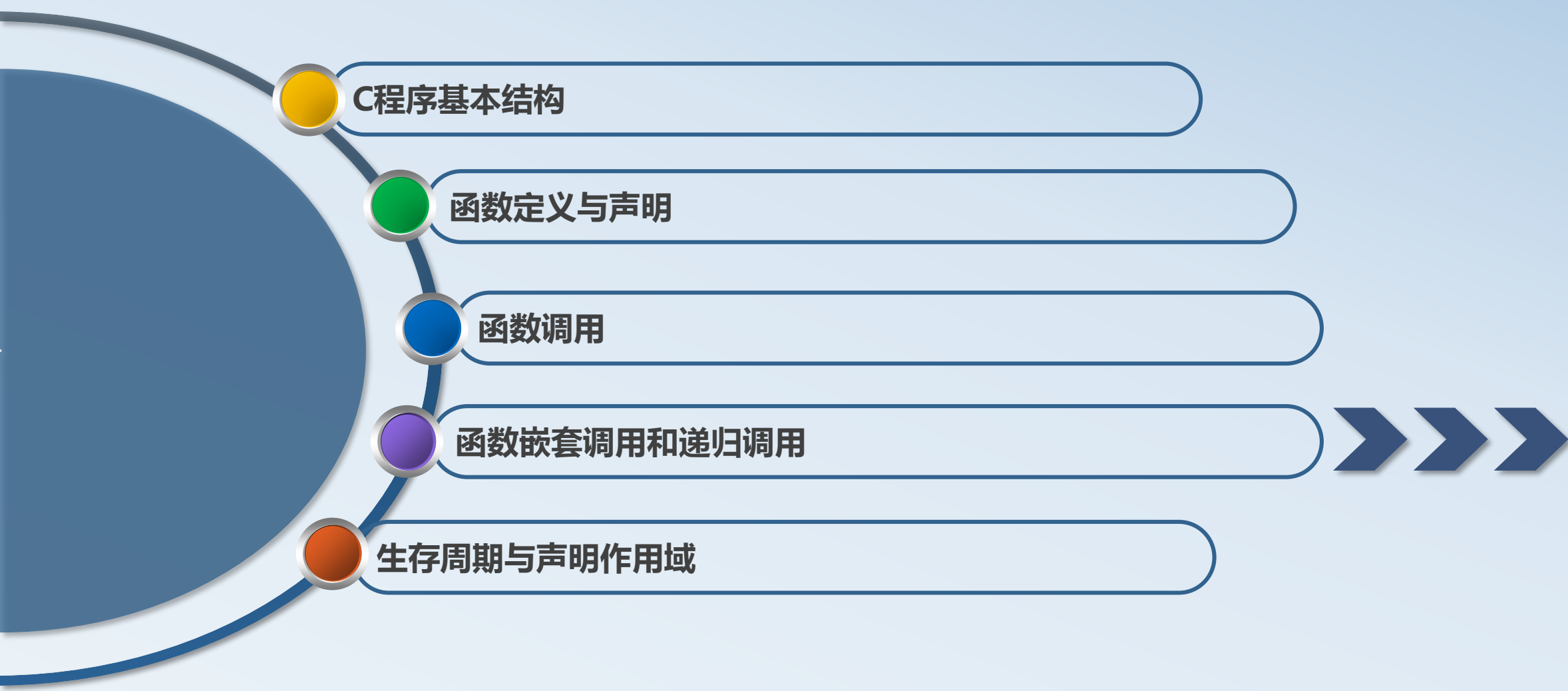
- ❖ 一维数组做形参时，数组名就首元素地址；传递的是数组首元素的地址。

`void printArr(const int * p, const int len)` 等同于

`void printArr(const int p[], const int len)`

- ◆ 编译器**不检查数组长度**，数组长度需要以整型参数传递
- ◆ 形参指针和实参指针指向一段**共同的内存**
- ◆ 只拷贝数组首地址，肯定比拷贝整个数组**效率高**

# 本章授课内容



C程序基本结构

函数定义与声明

函数调用

函数嵌套调用和递归调用

生存周期与声明作用域



# 函数嵌套调用和递归调用

❖ 函数直接或间接调用自己为递归调用

❖ 例 编程求n的阶乘。

$$n! = \begin{cases} 1, & \text{当 } n = 0 \text{ 时} \\ n * (n - 1)!, & \text{当 } n \geq 1 \text{ 时} \end{cases}$$

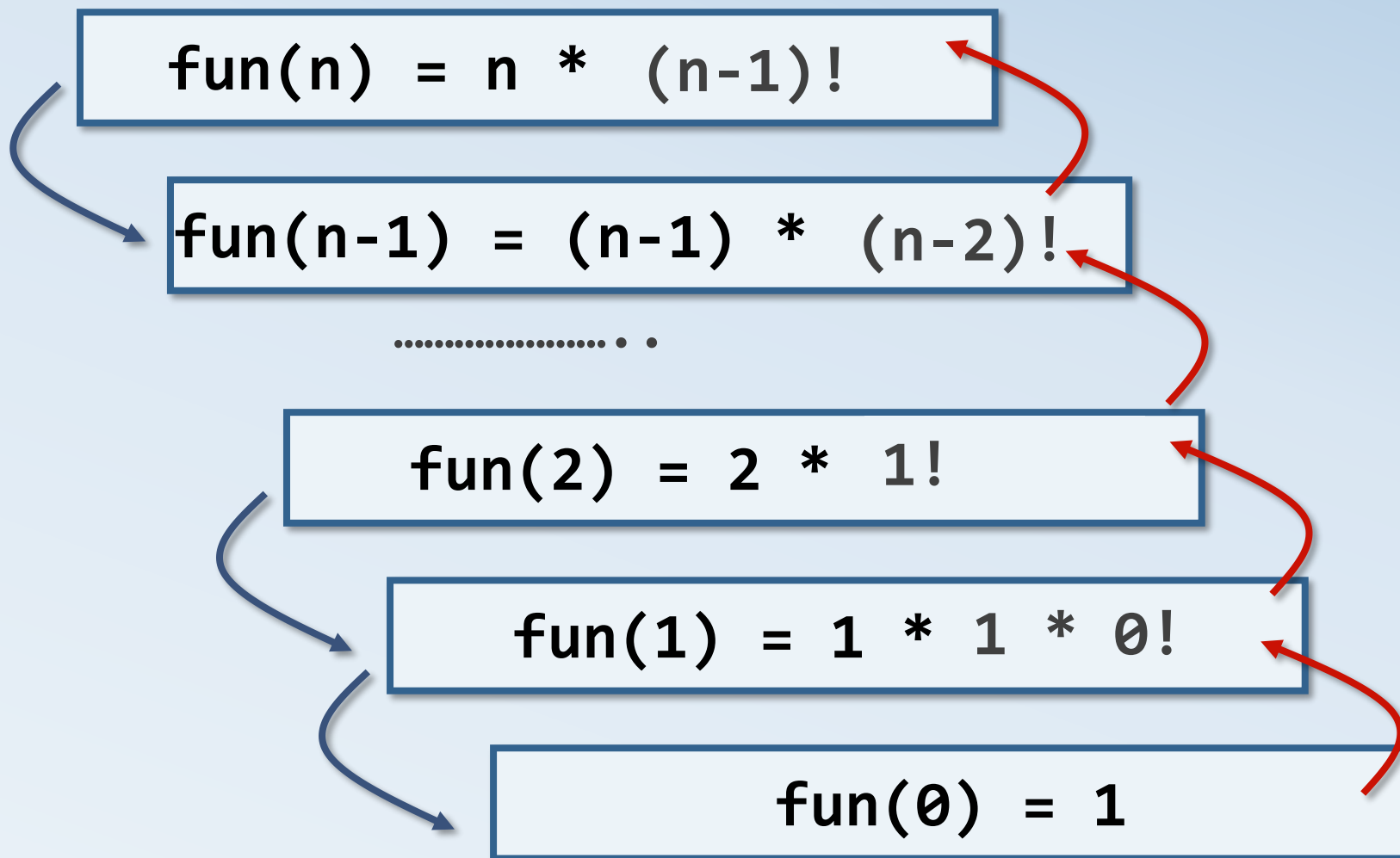
# 函数嵌套调用和递归调用

```
/* 此函数用于计算 a 的阶乘 */  
int fun(int a)  
{  
    if (a == 1)  
        return 1;  
    else  
    {  
        return a * fun(a-1);  
    }  
}
```

❖ 在一个函数体内调用自身称为函数的递归调用

# 函数嵌套调用和递归调用

❖ 递归调用该函数计算 $n!$ 的过程大致如下：



# 函数嵌套调用和递归调用

❖ 任何一个递归调用程序必须包括两部分

◆ 递归调用**继续**的过程

◆ 递归调用**结束**的过程

```
void foo(参数n)
{
    if(递归结束条件) // key1 递归出口在递归调用前
    { //TODO; }
    else
    {
        // TODO;
        foo(参数i); // key2
        // TODO;
    }
}
```

# 函数嵌套调用和递归调用

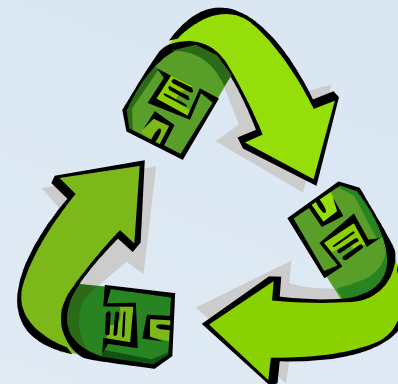
## ❖ 递归原理

- ◆ 问题的求解可通过降低问题规模实现，而小规模的问题求解方式与原问题的一样，小规模问题的解决导致问题的最终解决


## ❖ 递归调用应该能够在有限次数内终止递归

- ◆ 递归调用如果不加以限制，将**无数次的循环调用**

- ◆ 必须在函数内部加控制语句,只有当**满足一定条件**时，**递归终止**



# 本章授课内容



C程序基本结构

函数定义与声明

函数调用

函数嵌套调用和递归调用

生存周期与声明作用域



# 生存周期与声明作用域

## ❖生存周期

变量保持所分配存储空间的时间，称为变量的存储期间或生存期。

◆**静态生存周期** 若某对象在程序**开始执行之前**即分配到存储空间，而且保持到**程序终止**，则称该对象具有静态生存期。

（所有函数、在顶层声明的变量、静态变量。）

◆**本地生存期** 如果对象在**进入块或函数**时分配到存储空间，且在**退出块或函数**时删除，则称该对象具有本地生存期。C 语言把具有本地生存期的变量称为自动变量。



# 生存周期与声明作用域

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

```
int x = 3;
```

→ 静态生存周期

```
void foo(void)
```

→ 静态生存周期

```
{
```

```
    int a = 3;
```

→ 本地生存期

```
}
```

```
int main(void)
```

→ 静态生存周期

```
{
```

```
    int i = 0;
```

→ 本地生存期

```
    printf("x = %d\n", x);
```

```
    for(i = 0; i < 3; i++)
```

```
    {
```

```
        int j = 1;
```

→ 本地生存期

```
    }
```

```
    return 0;
```

```
}
```

# 生存周期与声明作用域

❖ **声明作用域**是声明变量**有效**的区域(空间)

- ◆ 顶层标示符

- ◆ 函数定义中的形式参数

- ◆ 块内标示符

- ◆ 标号标示符

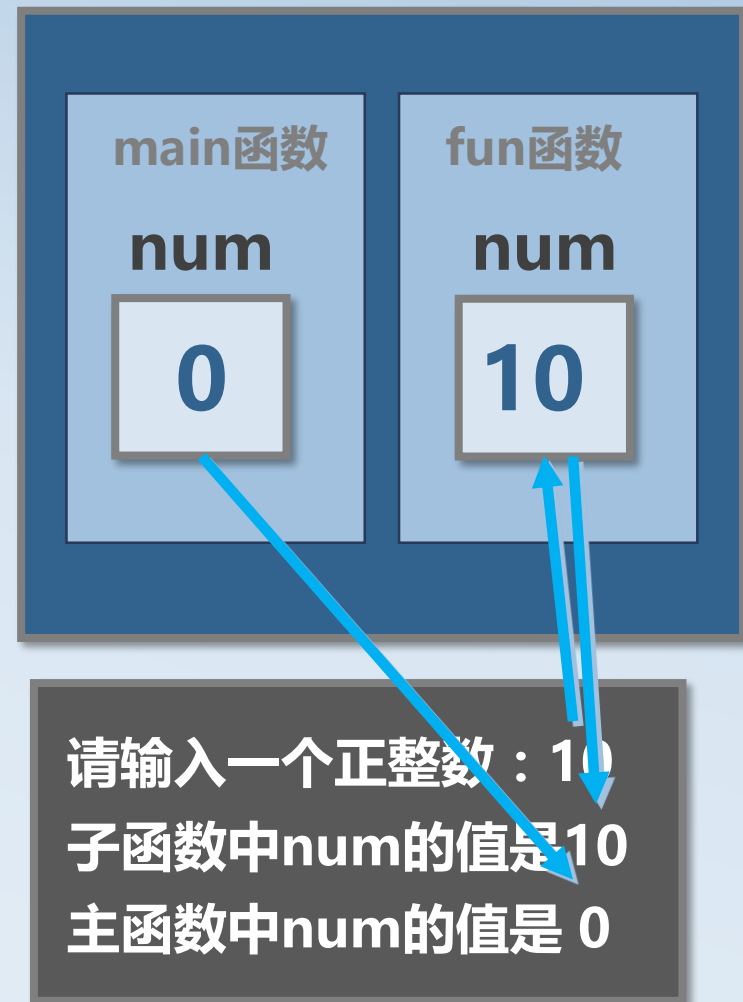
❖ **全局变量和局部变量**

- ◆ 称在顶层声明的变量为全局变量，与全局变量相对的是局部变量（如：函数体内的变量、形式参数、块内的变量）。

# 生存周期与声明作用域

```
#include <stdio.h>
void fun(void)
{
    int num;
    printf("请输入一个正整数: ");
    scanf("%d", &num);
    printf("子函数中num的值是%d\n", num);
}
void main(void)
{
    int num = 0;
    fun();
    printf("主函数中num的值是%d\n", num);
}
```

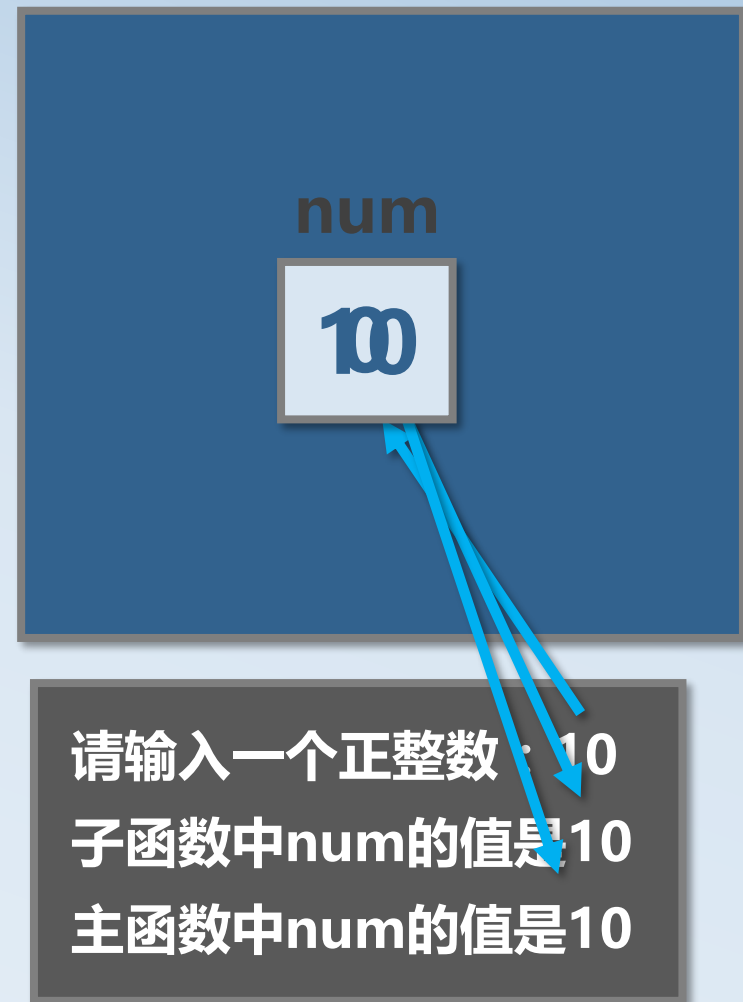
## 内存



# 生存周期与声明作用域

```
#include <stdio.h>
int num = 0;
void fun(void)
{
    printf("请输入一个正整数: ");
    scanf("%d", &num);
    printf("子函数中num的值是%d\n", num);
}
void main(void)
{
    fun();
    printf("主函数中num的值是%d\n", num);
}
```

## 内存



# 生存周期与声明作用域

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

```
int x = 3;
```

```
int main(void)
```

```
{
```

```
    int x = 4;
```

```
    int i = 0;
```

```
    printf("x = %d\n", x);
```

```
    for(i = 0; i < 1; i++)
```

```
    {
```

```
        int x = 5;
```

```
        printf("x = %d\n", x);
```

```
    }
```

```
    return 0;
```

```
}
```

作用域二

作用域三

作用域一


# 生存周期与声明作用域

- ❖ C语言程序中，在**同一作用域内不允许有同名的变量、函数**等，但在不同的作用域内，可能会出现同名的变量。
- ❖ 若使用的变量是在不同作用域内的同名变量，则**作用域小的变量优先**。

## 总结：

- 1.局部变量的作用域为定义它的块儿.**
- 2.全局变量的作用域为从定义它开始到程序结束.**

# 本讲授课程内容



存储类型说明符



类型限定符

动态内存分配标准库函数

指向函数的指针

# 存储类型说明符

❖ 存储类说明符用于确定声明对象的生存期（typedef 除外），存储类说明符出现在声明说明符中，一个声明说明符中最多允许一个存储类说明符。存储类说明符包括：

- ◆ auto
- ◆ register
- ◆ extern
- ◆ static
- ◆ typedef



# 存储类型说明符

存储类型	说明	
<b>auto</b>	自动变量	局部变量在缺省存储类型的情况下归为自动变量。
<b>register</b>	寄存器变量	存放在CPU的寄存器中。对于循环次数较多的循环控制变量及循环体内反复使用的变量均可定义为寄存器变量。
<b>static</b>	静态变量	在程序执行时存在，并且只要整个程序在运行，就可以继续访问该变量。作用在本文件内。
<b>extern</b>	外部变量	作用域是整个程序，包含该程序的各个文件。生存期非常长，它在该程序运行结束后，才释放内存。

# 存储类型说明符

// 源文件 **foo.c**

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

```
int x = 3;
```

```
void foo(int y)
```

```
{
```

```
    printf("in void foo(int);\n");
```

```
}
```

//源文件: **main.c**

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

```
extern int x; //是声明不是定义
```

```
extern void foo(int y);
```

```
int main(void)
```

```
{
```

```
    printf("%d\n", x);
```

```
    foo(1);
```

```
    return 0;
```

```
}
```

◆使用extern扩展对象的作用域：

- ✓扩展在本文件内的作用域

- ✓扩展对象到其它文件中

# 存储类型说明符

```
#include <stdio.h>

int foo1(void)
{
    static int x = 0;
    x++;
    return x;
}

int foo2(void)
{
    int n = 0;
    n++;
    return n;
}
```

```
int main(void)
{
    int y = 0;
    y = foo1();
    y = foo1();
    y = foo1();
    printf("in foo1 %d\n", y);

    y = foo2();
    y = foo2();
    y = foo2();
    printf("in foo2  %d\n", y);

    return 0;
}
```

# 存储类型说明符

- ❖ 静态局部变量生存期为整个源程序
- ❖ 静态局部变量作用域与自动变量相同，即只能在定义该变量的函数内使用该变量
- ❖ 对基本类型的静态局部变量若在说明时未赋以初值，则系统自动赋予0值
- ❖ 函数被调用时，其中的静态局部变量的值将保留前次被调用的结果

# 存储类型说明符

- ❖ 静态全局变量与普通全局变量在存储方式上完全相同

- ❖ 区别在于：

  - ◆ 非静态全局变量的作用域是**整个源程序中的所有文件**

  - ◆ 静态全局变量的作用域只是**定义它的文件中**

- ❖ static

  - ◆ 在函数定义中，说明此函数不可在其他文件中使用。

  - ◆ 在变量声明中，说明此变量不可在其他文件中使用。

  - ◆ static变量在编译时初始化，只初始化一次，如果没有初始化，默认值为0。

# 存储类型说明符

- ❖ typedef用于定义新的数据类型名
  - ◆ 给原数据类型取个更顾名思义的名字
  - ◆ 出于程序跨平台编译的需要
  - ◆ 为复杂的声明定义一个新的简单的别名
- ❖ 基本语法

**typedef    已有数据类型名    新数据类型名;**

- ❖ 例


```
typedef int * intPointer; // 使intPointer 成为了int*的别名  
intPointer x; // 在编译阶段, 该语句将被处理成int* x;
```

# 存储类型说明符

❖ typedef用于简化复杂类型名

- ◆ `typedef unsigned long ULONG;`
- ◆ `typedef const int KINT;`
- ◆ `typedef int const * const KKINT。`
- ◆ `typedef struct student STU`

# 本讲授课程内容



存储类型说明符

类型限定符

动态内存分配标准库函数

指向函数的指针






# 类型限定符

- ❖ 类型限定符仅能对左值访问某对象进行“限定”
  - ◆ const限定符“限定”对象具有只读属性，即对象是不可修改的。
  - ◆ volatile关键字告诉编译器在每次使用它所修饰的对象时，都要重新读取，即使程序本身并没有修改它的值(多用于多进程)。
  - ◆ restrict限定符仅用于修饰指针，它“限定”所修饰的指针是唯一的访问一个数据对象的方式。(不是所有编译器都支持)

# 本讲授课程内容



存储类型说明符



类型限定符



动态内存分配标准库函数



指向函数的指针



# 动态内存分配标准库函数

❖ 前面我们讲的都是自定义函数，除了自定义的函数外，我们也可以调用系统已经定义好的函数，根据函数是否是自己定义，我们可以将函数分为：

- ◆ 自定义函数

- ◆ 库函数 例如 `printf()` , `scanf()` 等等

❖ 库函数

- ◆ 库函数中有一类函数可以让我们自己分配内存

- `malloc`

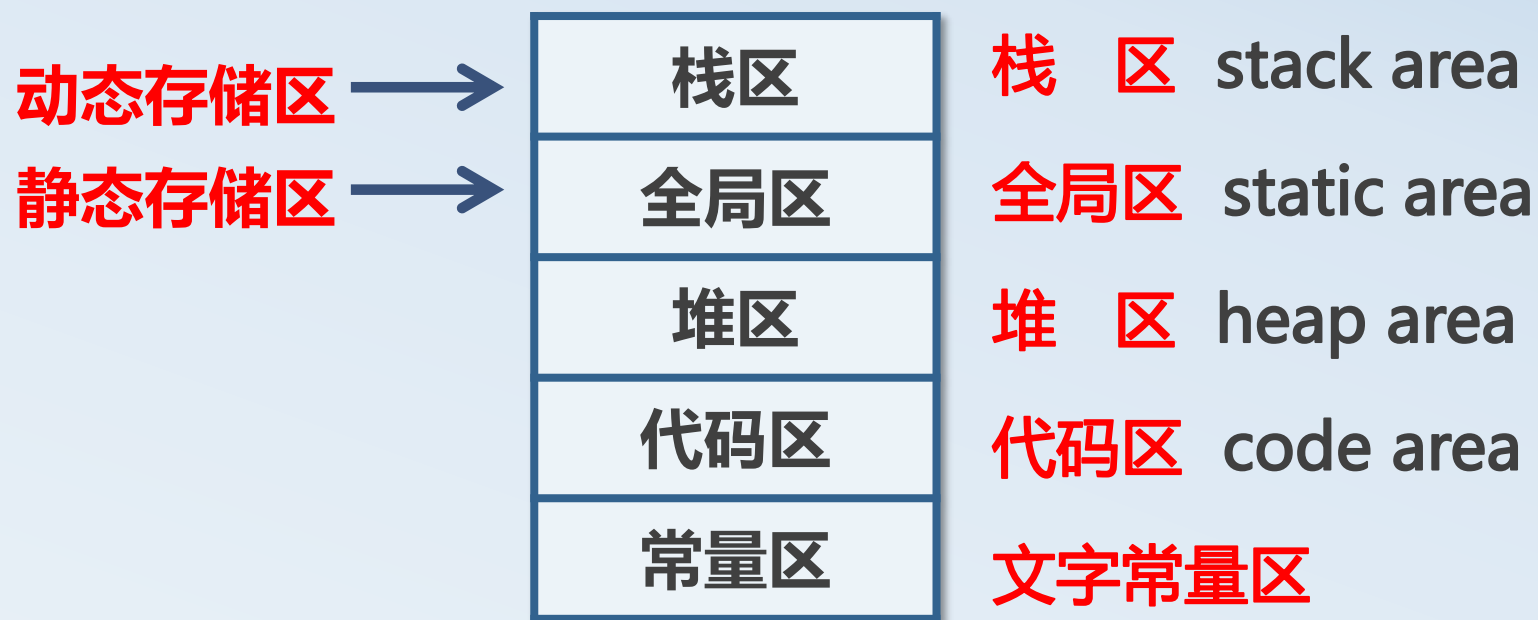
- `calloc`

- `free`

# 动态内存分配标准库函数

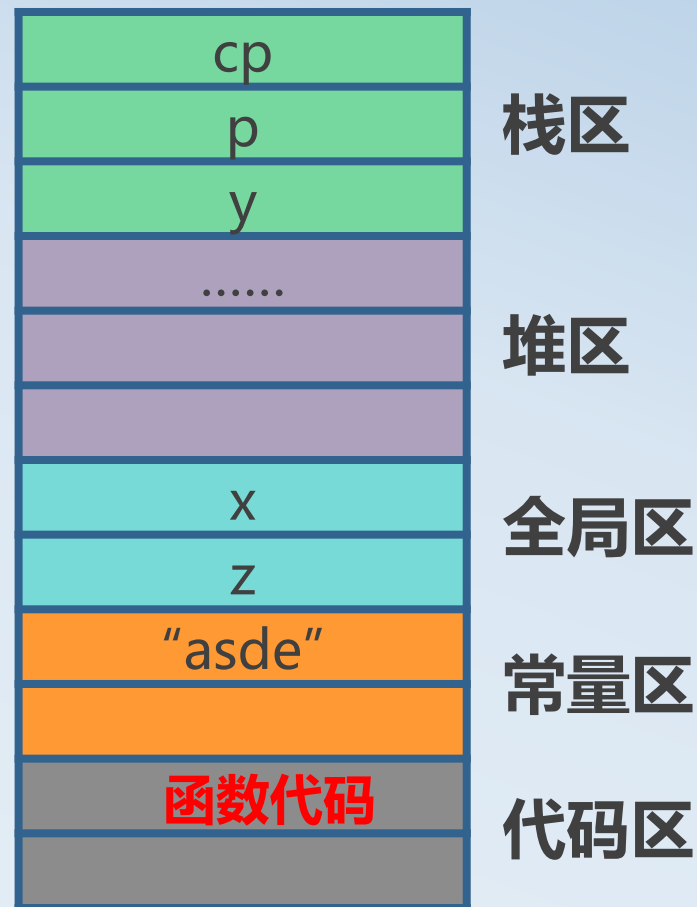
## ❖ 程序在内存的分布区域

- ◆ 一个程序将操作系统分配的内存空间分为五个区域，如下图所示。这五个区域的用途和性能是不同的。



# 动态内存分配标准库函数

```
#include <stdio.h>
int x = 3;
void foo()
{
    /* TODO */
}
int main(void)
{
    char *cp = "asde";
    int *p = &x;
    int y = 4;
    static int z = 5;
    return 0;
}
```



# 动态内存分配标准库函数

## ❖ 动态分配存储

- ◆ 根据需要开辟或释放存储单元

## ❖ 动态存储分配函数

- ◆ malloc函数

- ◆ calloc函数

- ◆ free函数

## ❖ 说明

- ◆ 应包含malloc.h或stdlib.h

# 动态内存分配标准库函数

## ❖ 函数原型

◆ **void\*** malloc(**size\_t size**);

## ❖ 参数

◆ **size**: 分配存储空间的字节数

## ❖ 返回值

◆ 若成功，返回指向分配区域起始地址的**void类型指针**

◆ 若**失败**，返回**NULL**

```
int *p = (int *)malloc(1*sizeof(int)); //必须进行强转
if(NULL == p) //必须检测内存是否分配成功
{
    printf("内存分配失败!\n");
}
```

# 动态内存分配标准库函数

## ❖ 函数原型

◆ **void\*** **calloc**( **size\_t** **num**, **size\_t** **size**);

## ❖ 参数

◆ **num** :分配内存的块数

◆ **size**:分配内存的每块的字节数

## ❖ 返回值

◆ 若成功，返回指向分配区域起始地址的**void**指针

◆ 若失败，返回**NULL**

◆ **calloc**和**malloc**函数区别：

◆ **calloc**分配完毕后该内存所有内容都被**初始化为0**



# 动态内存分配标准库函数

## ❖ 函数原型

◆ **void free(void \* Memory);**

## ❖ 参数

◆ **Memory**: 要释放块内存的指针

## ❖ 说明

◆ 释放由malloc()和calloc()申请的内存块


◆ 释放后的内存区能够分配给其他变量使用

# 动态内存分配标准库函数

## ❖ 动态内存分配

- ◆ 使用malloc、calloc分配的内存一定要自己用free释放。
- ◆ 用malloc、calloc进行内存分配，**一定要检测成功与否**。
- ◆ malloc、calloc函数的返回值是**void类型**，一定要根据需要进行强制类型转换。

# 本讲授课程内容




存储类型说明符



类型限定符



动态内存分配标准库函数



指向函数的指针



# 指向函数的指针

❖ 函数指针指向代码区中的某个函数，通过该指针可以访问或者说调用该函数。

❖ 一般形式

◆ **返回值类型** (\***指针变量名**) ( **形参列表** )

❖ 例：

◆ `int (*fp)(char, char) = 0;`

◆ `int *(*fp)(int, double) = 0;`

# 指向函数的指针

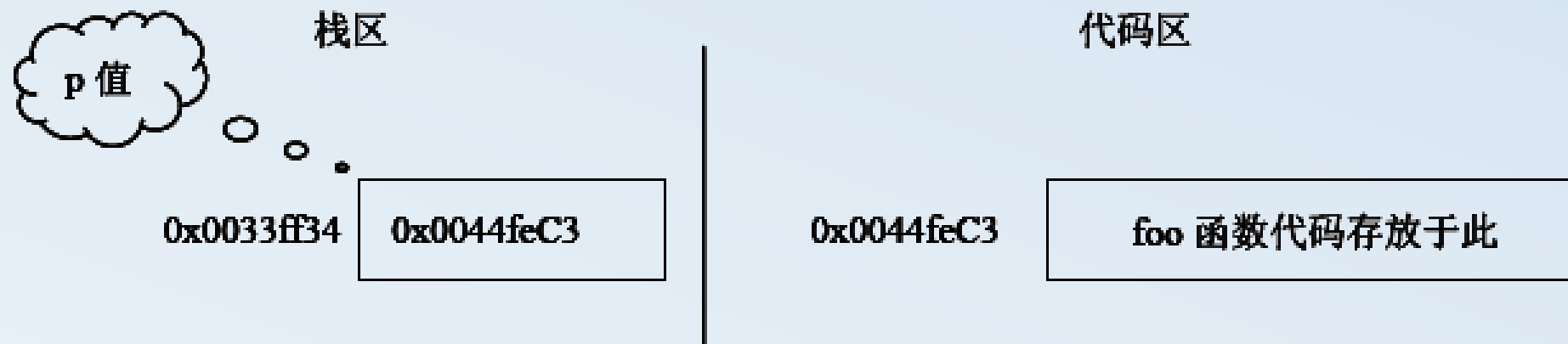
## ❖ 赋值形式

◆ 函数指针 = 函数名;

## ❖ 调用形式

◆ 函数指针 ( 实参列表 );

◆ (\*函数指针名)(实参列表);



指向函数的指针存储示意图

Thank You !