C 语言和程序设计思想 I

郑艺林

计算机科学与工程系

2016 年 12 月 10 日

课程内容 I

1 函数

- 函数原型、定义
- 头文件
- 函数参数
- 函数返回值
- 函数调用
- 一元运算符 & *
- 递归

2 数组和指针

- 数组
- 多维数组
- 指针
- 函数、数组和指针

课程内容 II

- 指针操作
- 保护数组内容
- 指针和多维数组
- 变长数组
- *复合文字

3 作业

- 作业

课程内容

函数原型

函数的声明语法结构

函数原型

函数的声明语法结构

- 函数原型
  返回值 函数名(传入参数/void);

函数原型

函数的声明语法结构

- 函数原型
  返回值 函数名(传入参数/void);
- e.g. int Add(int num1, int num2);
  int Add(int, int);

函数原型

函数的声明语法结构

- 函数原型
  返回值 函数名(传入参数/void);
- e.g. int Add(int num1, int num2);
  int Add(int, int);

函数原型

函数的声明语法结构

- 函数原型
  返回值 函数名(传入参数/void);
- e.g. int Add(int num1, int num2);
  int Add(int, int);

函数原型是必要的，告知编译器函数的类型

函数定义

函数定义

- 函数定义

```
返回值 函数名(传入参数/void)
{
    statements;
}
```

函数定义

- 函数定义

```
返回值 函数名(传入参数/void)
{
    statements;
}
```

- e.g.

```
int Add(int num1,int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

函数定义

- 函数定义

```
返回值 函数名(传入参数/void)
{
    statements;
}
```

- e.g.

```
int Add(int num1,int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

函数定义确切指定了函数的具体功能

头文件

文件以 .h 结尾的文件是头文件

头文件

文件以 .h 结尾的文件是头文件

- 函数原型和常量经常在头文件中定义

头文件

文件以 .h 结尾的文件是头文件

- 函数原型和常量经常在头文件中定义
- 头文件中不包含函数的定义
  函数的具体定义在另一个库函数文件中

函数参数

形式参数/形式参量

函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内

函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内
- 调用函数时形参会被赋值

函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内
- 调用函数时形参会被赋值

函数参数

形式参数/形式参量

- 形参是局部变量，作用域仅在函数体内
- 调用函数时形参会被赋值

带参数的函数使函数利于重复使用

函数的返回值

函数执行结束可以有返回值也可以没有

函数的返回值

函数执行结束可以有返回值也可以没有

- 基本的数据类型可以作为函数返回值，必须有 return

函数的返回值

函数执行结束可以有返回值也可以没有

- 基本的数据类型可以作为函数返回值，必须有 return
- 无返回值则使用关键词 void

return 的作用

return 具有多项作用

return 的作用

return 具有多项作用

- 返回函数返回值

return 的作用

return 具有多项作用

- 返回函数返回值
- 终止执行函数 - 只能用于 void 类型函数中

函数间的通信

调用带有参数的函数需要使用实际参数

函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值

函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应

函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应
- e.g. `Add(1,2);`

函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应
- e.g. `Add(1,2);`

函数间的通信

调用带有参数的函数需要使用实际参数

- 实际参数对形式参数进行赋值
- 类型必须对应
- e.g. `Add(1,2);`

注意不对应的类型赋值导致的问题

地址运算符 &

地址运算符的作用是取地址

地址运算符 &

地址运算符的作用是取地址

- e.g. `int num; int * p; p = &num;`

地址运算符 &

地址运算符的作用是取地址

- e.g. int num; int * p; p = #
- 输出地址使用格式说明符 %p

间接运算符 *

定义指针变量时使用间接运算符 *，也叫取值运算符

间接运算符 *

定义指针变量时使用间接运算符 *，也叫取值运算符

- e.g. p = # *p ?

间接运算符 *

定义指针变量时使用间接运算符 *，也叫取值运算符

- e.g. p = # *p ?

间接运算符 *

定义指针变量时使用间接运算符 *，也叫取值运算符

- e.g. p = # *p ?

*p 的值是 10

递归

函数调用本身的过程称为递归

递归

函数调用本身的过程称为递归

- 使用示例

```
int Factor(int n)
{
    if( n == 0)
    {
        return 1;
    }
    return n*Factor(n-1);
}
```

递归

函数调用本身的过程称为递归

- 使用示例

```
int Factor(int n)
{
    if( n == 0)
    {
        return 1;
    }
    return n*Factor(n-1);
}
```

思考一下

程序分析

程序分析

- 执行过程

```
Factor(n)
    Factor(n-1)
        Factor(n-2)
            ...
                Factor(base case)
                return value
            ...
        return value to Factor(n-2)
    return value to Factor(n-1)
return value to Factor(n)
```

递归基本原理

递归有如下几点原理

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回
- 位于递归调用前的语句和各级被调函数的执行顺序相同

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回
- 位于递归调用前的语句和各级被调函数的执行顺序相同
- 位于递归调用后的语句和各个被调函数的执行顺序相反

递归基本原理

递归有如下几点原理

- 每一级函数调用都有自己的变量
- 代码不会被复制
- 每次函数调用都会有一次返回
- 位于递归调用前的语句和各级被调函数的执行顺序相同
- 位于递归调用后的语句和各个被调函数的执行顺序相反
- 递归函数中必须包含可以终止递归调用的语句

递归的应用

递归的应用广泛

递归的应用

递归的应用广泛

- 尾递归最简单

递归的应用

递归的应用广泛

- 尾递归最简单
- 递归可以用以反向计算 e.g. 十进制转化二进制输出

递归的优缺点

递归的优缺点

- 递归优点

递归的优缺点

- 递归优点
  - 为某些问题提供了简单的解决方法

递归的优缺点

- 递归优点
  - 为某些问题提供了简单的解决方法
  - 可简化代码

递归的优缺点

- 递归优点
  - 为某些问题提供了简单的解决方法
  - 可简化代码
- 递归缺点

递归的优缺点

- 递归优点
  - 为某些问题提供了简单的解决方法
  - 可简化代码
- 递归缺点
  - 内存资源消耗大

递归的优缺点

- 递归优点
  - 为某些问题提供了简单的解决方法
  - 可简化代码
- 递归缺点
  - 内存资源消耗大

递归的优缺点

- 递归优点
  - 为某些问题提供了简单的解决方法
  - 可简化代码
- 递归缺点
  - 内存资源消耗大

谨慎使用递归！！！

课程内容

数组声明和定义

数组是由同一类元素构成的
数组的声明包括数组元素的数目和类型

数组声明和定义

数组是由同一类元素构成的
数组的声明包括数组元素的数目和类型

- 示例分析

```
float price[365]
int matrix[10][10]
double speed[5][5][5]
...
```

数组的初始化

使用花括号来初始化数组

数组的初始化

使用花括号来初始化数组

- 示例分析

```
int Month[12]={1,2,3,4,5,6,7,8,9,10,11,12};
```

数组的初始化

使用花括号来初始化数组

- 示例分析

```
int Month[12]={1,2,3,4,5,6,7,8,9,10,11,12};
```

初始化的类型要对应

数组的初始化

判断一下是否正确

数组的初始化

判断一下是否正确

- 示例分析

```
int powers[6] = {1, 2, 4, 6, 8}; ?
int powers2[6];
powers = powers2; ?
powers[6] = powers2[6]; ?
powers[6]={1, 2, 4, 6, 8}; ?
```

指定初始化项目

C99 的新特性能够指定初始化的项目

指定初始化项目

C99 的新特性能够指定初始化的项目

- 示例分析

```
int Days[MONTHS]={31, 28, [3] = 31, 31, 30,
    [11]=31, [3]=30};
```

指定初始化项目

C99 的新特性能够指定初始化的项目

- 示例分析

```
int Days[MONTHS]={31, 28, [3] = 31, 31, 30,
    [11]=31, [3]=30};
```

初始化项目之后的值接着初始化后续项目
多次对一个项目初始化，则仅有最后一次有效

数组边界

数组的元素下标从 0 开始

数组边界

数组的元素下标从 0 开始

- 示例分析

```
int Month[12]={1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12};
```

数组边界

数组的元素下标从 0 开始

- 示例分析

```
int Month[12]={1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12};
```

最后一个元素的下标值比元素个数少 1

编译器不会检查索引的合法性 - 段错误

指定数组大小

声明数组时可以使用整数常量
也可以指定数组大小

指定数组大小

声明数组时可以使用整数常量
也可以指定数组大小

- 示例代码

```
int n = 6, m = 9;
int a1[10];
int a2[2*3-1];
int a3[sizeof(int)+1];
int a4[-2];
int a5[0];
int a6[3.1415926];
int a7[(int)3.1415926];
int a8[n];
int a9[m];
```

指定数组大小

声明数组时可以使用整数常量
也可以指定数组大小

- 示例代码

```
int n = 6, m = 9;
int a1[10];
int a2[2*3-1];
int a3[sizeof(int)+1];
int a4[-2];
int a5[0];
int a6[3.1415926];
int a7[(int)3.1415926];
int a8[n];
int a9[m];
```

a8, a9 为变长数组

多维数组的声明

二维数组、三维数组.....多维数组

多维数组的声明

二维数组、三维数组.....多维数组

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

多维数组的初始化

以二维数组为例

多维数组的初始化

以二维数组为例

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

多维数组的初始化

以二维数组为例

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

二维数组的每个元素都是一个数组，多维数组以此类推

多维数组的初始化

以二维数组为例

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

二维数组的每个元素都是一个数组，多维数组以此类推
数组是顺序存储的

多维数组的初始化

以二维数组为例

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

二维数组的每个元素都是一个数组，多维数组以此类推
数组是顺序存储的
初始化的时候可以省略内部的花括号，但是得确保元素个数是正确的

多维数组的初始化

以二维数组为例

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

二维数组的每个元素都是一个数组，多维数组以此类推

数组是顺序存储的

初始化的时候可以省略内部的花括号，但是得确保元素个数是正确的

思考

初始化的个数少了？

多维数组的初始化

以二维数组为例

- 二维数组 - 数组的数组

```
int Matrix[5][5]={ {1, 0, 0, 1, 1},
                   {1, 1, 1, 1, 0},
                   {1, 0, 1, 0, 0},
                   {0, 0, 1, 1, 0},
                   {1, 1, 0, 0, 1} };
```

二维数组的每个元素都是一个数组，多维数组以此类推

数组是顺序存储的

初始化的时候可以省略内部的花括号，但是得确保元素个数是正确的

思考

初始化的个数少了？

多了？

指针 == 数组

指针提供了使用地址的符号方法
数组标记实际上是变相使用指针的形式

指针 == 数组

指针提供了使用地址的符号方法
数组标记实际上是变相使用指针的形式

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
```

指针 == 数组

指针提供了使用地址的符号方法
数组标记实际上是变相使用指针的形式

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
```

numbers == ?

指针 == 数组

指针提供了使用地址的符号方法
数组标记实际上是变相使用指针的形式

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
```

numbers == ?
&numbers[0]

指针 == 数组

指针提供了使用地址的符号方法
数组标记实际上是变相使用指针的形式

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
```

numbers == ?
&numbers[0]
数组元素首地址可赋值给指针

指针 == 数组

对指针的加减可以改变指针指向的数组元素

指针 == 数组

对指针的加减可以改变指针指向的数组元素

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
int *p;
p = numbers;
p ?
*p ?
*(p + 1) ?
*p + 1 ?
```

指针 == 数组

对指针的加减可以改变指针指向的数组元素

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
int *p;
p = numbers;
p ?
*p ?
*(p + 1) ?
*p + 1 ?
```

指针的数值就是它所指向的对象的地址

指针 == 数组

对指针的加减可以改变指针指向的数组元素

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
int *p;
p = numbers;
p ?
*p ?
*(p + 1) ?
*p + 1 ?
```

指针的数值就是它所指向的对象的地址
指针前使用取值运算符 * 可以得到该指针所指向的对象的值

指针 == 数组

对指针的加减可以改变指针指向的数组元素

- 示例分析

```
int numbers[5] = {1, 2, 3, 4, 5};
int *p;
p = numbers;
p ?
*p ?
*(p + 1) ?
*p + 1 ?
```

指针的数值就是它所指向的对象的地址
指针前使用取值运算符 * 可以得到该指针所指向的对象的值
对指针的加 1，
等价于对指针的值加上它所指向的对象的字节数

声明数组参量

编写一个求和函数

声明数组参量

编写一个求和函数

- 示例代码

```
int data[5] = {3, 43, -97, 134, -26};
int *p;
p = data;

int sum(int data[]);
int sum(int *p);
```

声明数组参量

编写一个求和函数

- 示例代码

```
int data[5] = {3, 43, -97, 134, -26};
int *p;
p = data;

int sum(int data[]);
int sum(int *p);
```

两个函数内部如何实现?

函数实现

函数实现

- 示例代码 1

```
int sum(int data[])
{
    int i, sum = 0;
    for(i = 0; i < 5; i++)
        sum += data[i];
    return sum;
}
```

函数实现

- 示例代码 2

```
int sum(int *p)
{
    int i, sum = 0;
    for(i = 0; i < 5; i++)
        sum += p[i]; // sum += *(p+i);
    return sum;
}
```

函数实现

- 示例代码 2

```
int sum(int *p)
{
    int i, sum = 0;
    for(i = 0; i < 5; i++)
        sum += p[i]; // sum += *(p+i);
    return sum;
}
```

对比区别

指针参数

同一个函数实现使用指针参数

指针参数

同一个函数实现使用指针参数

- 示例代码

```
int *begin, *end;
start = data;
end = data+5; // ?
int sum(int *begin, int *end)
{
    int sum = 0;
    while(begin < end)
    {
        sum += *begin;
        begin++; // sum += *begin++;
    }
    return sum;
}
```

思考

* 和 ++ 具有相同优先级

思考

* 和 ++ 具有相同优先级
  - 思考如下表达式

```
*begin++
(*begin)++
*(begin++)
*++begin
```

基本指针操作

C 提供了 6 种基本的指针操作

基本指针操作

C 提供了 6 种基本的指针操作

```
int data[5]; int *p,*r; int **q;
```

基本指针操作

C 提供了 6 种基本的指针操作

```
int data[5]; int *p,*r; int **q;
```

- 赋值

```
p = data;
```

基本指针操作

C 提供了 6 种基本的指针操作

```
int data[5]; int *p,*r; int **q;
```

- 赋值

```
p = data;
```

- 取值

```
int number = *p;
```

基本指针操作

C 提供了 6 种基本的指针操作

```
int data[5]; int *p,*r; int **q;
```

- 赋值

```
p = data;
```

- 取值

```
int number = *p;
```

- 取指针地址

```
q = &p;
```

基本指针操作

C 提供了 6 种基本的指针操作

```
int data[5]; int *p,*r; int **q;
```

- 赋值

```
p = data;
```

- 取值

```
int number = *p;
```

- 取指针地址

```
q = &p;
```

- 加/减

```
p++; // p += 1;
--p; // p -= 1;
```

基本的指针操作

```
r = &data[2];
```

基本的指针操作

```
r = &data[2];
```

- 求差值

```
int difference = p-q; // ?
int difference = p-r; // ?
```

基本的指针操作

```
r = &data[2];
```

- 求差值

```
int difference = p-q; // ?
int difference = p-r; // ?
```

- 比较

```
p > q; //?
p < r; //?
```

基本的指针操作

```
r = &data[2];
```

- 求差值

```
int difference = p-q; // ?
int difference = p-r; // ?
```

- 比较

```
p > q; //?
p < r; //?
```

求指针之间的差值前提是指针均指向同一个数组内的元素
差值单位为对应类型的大小

基本的指针操作

```
r = &data[2];
```

- 求差值

```
int difference = p-q; // ?
int difference = p-r; // ?
```

- 比较

```
p > q; //?
p < r; //?
```

求指针之间的差值前提是指针均指向同一个数组内的元素
差值单位为对应类型的大小
比较指针的值时务必确保类型相同

const 的使用

只有当函数需要使用修改值得时候，才传递指针
使用 const 关键字可以保护数据避免被修改

const 的使用

只有当函数需要使用修改值得时候，才传递指针
使用 const 关键字可以保护数据避免被修改

- 对形式参量使用 const

```
int ar[]={1,2,3,4,5};
int sum(const int ar[], int n);
```

const 的使用

只有当函数需要使用修改值得时候，才传递指针
使用 const 关键字可以保护数据避免被修改

- 对形式参量使用 const

```
int ar[]={1,2,3,4,5};
int sum(const int ar[], int n);
```

ar[] 是否可修改？

const 的使用

只有当函数需要使用修改值得时候，才传递指针
使用 const 关键字可以保护数据避免被修改

- 对形式参量使用 const

```
int ar[]={1,2,3,4,5};
int sum(const int ar[], int n);
```

ar[] 是否可修改?
函数内的是否可以修改 ar[] ?

const 的其他用法

const 还有许多其他用法

const 的其他用法

const 还有许多其他用法

- 创建符号常量

const 的其他用法

const 还有许多其他用法

- 创建符号常量
- 创建指向常量的指针

```
int numbers[] = {5, 1, 0, 6, 2, 9};
const int *p = numbers;
*p = 4; // ?
p[3] = 3; // ?
numbers[0] = 3; // ?
```

函数不会用该指针修改数值，但是该指针的指向可以修改

const 的其他用法

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的
- 只有变量的地址可以赋给普通指针

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的
- 只有变量的地址可以赋给普通指针

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的
- 只有变量的地址可以赋给普通指针
- 创建指针常量，该指针的指向不可修改，
  但是仍旧可以用来修改数据

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的
- 只有变量的地址可以赋给普通指针
- 创建指针常量，该指针的指向不可修改，
  但是仍旧可以用来修改数据

const 的其他用法

- 常量或变量的地址赋给指向常量的指针是合法的
- 只有变量的地址可以赋给普通指针
- 创建指针常量，该指针的指向不可修改，
  但是仍旧可以用来修改数据
- 使用两个 const
  创建指针，既不可修改指针指向，也不了修改数据

指向二维数组的指针

以二维数组为例

```
int Matrix[4][4];
```

指向二维数组的指针

以二维数组为例

```
int Matrix[4][4];
```

- Matrix 是首元素地址，即 Matrix[0][0] 地址

指向二维数组的指针

以二维数组为例

```
int Matrix[4][4];
```

- Matrix 是首元素地址，即 Matrix[0][0] 地址
- 指针增加 1，指向下一个元素，即 Matrix[1][0]

指向二维数组的指针

以二维数组为例

```
int Matrix[4][4];
```

- Matrix 是首元素地址，即 Matrix[0][0] 地址
- 指针增加 1，指向下一个元素，即 Matrix[1][0]
- 对指针取值的结果是指针指向的元素的值

```
*Matrix = ?
*(Matrix[0]) = ?
**Matrix ? *&Matrix
```

指向二维数组的指针

以二维数组为例

```
int Matrix[4][4];
```

- Matrix 是首元素地址，即 Matrix[0][0] 地址
- 指针增加 1，指向下一个元素，即 Matrix[1][0]
- 对指针取值的结果是指针指向的元素的值

```
*Matrix = ?
*(Matrix[0]) = ?
**Matrix ? *&Matrix
```

Matrix 是地址的地址/指针的指针

分析

分析以下表达式

分析

分析以下表达式

- Matrix

分析

分析以下表达式

- Matrix 第一行

分析

分析以下表达式

- Matrix 第一行
- Matrix+2

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行
- *(Matrix+2)

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行
- *(Matrix+2) 第三行首元素

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行
- *(Matrix+2) 第三行首元素
- *(Matrix+2)+1

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行
- *(Matrix+2) 第三行首元素
- *(Matrix+2)+1 第三行第二个元素的地址

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行
- *(Matrix+2) 第三行首元素
- *(Matrix+2)+1 第三行第二个元素的地址
- *(*(Matrix+2)+1)

分析

分析以下表达式

- Matrix 第一行
- Matrix+2 第三行
- *(Matrix+2) 第三行首元素
- *(Matrix+2)+1 第三行第二个元素的地址
- *(*(Matrix+2)+1) 第三行第二个元素的值

例题解析

14 题：使用指针给一个 n*n 数组赋值，第 i 行为 i*100+1，i*100+2，...，i*100+n

例题解析

14 题：使用指针给一个 n*n 数组赋值，第 i 行为 i*100+1，i*100+2，...，i*100+n

- 示例代码

```
void Assign(int n, int (*p)[n])
{
    int i, j;
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            p[i][j] = (i + 1) * 100 + (j + 1)
                ;
}
```

指针兼容性

指针兼容性

- 指针的赋值规则比数值类型的赋值规则更严格

变长数组的定义

- 示例代码

```
int row = 10, col = 8;
int Matrix[row][col];
```

函数中的变长数组

- 示例代码

```
int row, col;
int Matrix[row][col];
int sum(int row, int col, int Matrix[row
    ][col]); //?
int sum(int Matrix[row][col], int row,
    int col); //?
```

函数中的变长数组

- 示例代码

```
int row, col;
int Matrix[row][col];
int sum(int row, int col, int Matrix[row
    ][col]); //?
int sum(int Matrix[row][col], int row,
    int col); //?
```

变长数组允许动态分配内存单元，在程序运行时指定数组大小
常规的数组大小在编译时已经确定。

复合文字

- 数组

```
int array[4] = {2, 4, 6, 8};
```

- 复合文字

```
(int [4]){2, 4, 6, 8}
(int [ ]){2, 4, 6, 8}
```

复合文字

- 数组

```
int array[4] = {2, 4, 6, 8};
```

- 复合文字

```
(int [4]){2, 4, 6, 8}
(int [ ]){2, 4, 6, 8}
```

初始化复合文字也可省略数组大小
符合文字会被表示为一个数组
复合文字也可以作为实际参数被传递和被赋值

课程内容

作业

- 第九章 9.10-2、7、8、9；9.11-2、8、10

作业

- 第九章 9.10-2、7、8、9；9.11-2、8、10
- 第十章 10.12-1、4、5、10、12；10.13-2、8、9