计算机程序设计习题课

Qirun Zeng

2024 年 12 月 21 日

目录

数组的定义

定义

数组是一种线性结构，它包含一系列相同类型的元素，这些元素在内存中连续地分布。数组的元素可以通过下标访问。

一般的数组定义如下：

```
DataType arrayName[arraySize];
```

其中 `DataType` 为数组的数据类型，`arrayName` 为数组名，`arraySize` 为数组的大小。

数组的操作

以一维数组为例，数组的操作包括：

- 初始化数组 `DataType arrayName[arraySize] = {value1, value2, ...};`

数组的操作

以一维数组为例，数组的操作包括：

- 初始化数组 `DataType arrayName[arraySize] = {value1, value2, ...};`
- 访问数组元素 `arrayName[index];`

数组的操作

以一维数组为例，数组的操作包括：

- 初始化数组 `DataType arrayName[arraySize] = {value1, value2, ...};`
- 访问数组元素 `arrayName[index];`
- 修改数组元素 `arrayName[index] = newValue;`

数组的操作

以一维数组为例，数组的操作包括：

- 初始化数组 `DataType arrayName[arraySize] = {value1, value2, ...};`
- 访问数组元素 `arrayName[index];`
- 修改数组元素 `arrayName[index] = newValue;`
- 遍历数组 `for (int i = 0; i < arraySize; i++) {...}`

数组的初始化

初始化

数组的初始化可以通过以下几种方式：

- 指定数组大小，不指定初始值 `int a[5];`

数组的初始化

初始化

数组的初始化可以通过以下几种方式：

- 指定数组大小，不指定初始值 `int a[5];`
- 指定数组大小，指定初始值 `int a[5] = {1, 2, 3, 4, 5};`

数组的初始化

初始化

数组的初始化可以通过以下几种方式：

- 指定数组大小，不指定初始值 `int a[5];`
- 指定数组大小，指定初始值 `int a[5] = {1, 2, 3, 4, 5};`
- 不指定数组大小，指定初始值 `int a[] = {1, 2, 3, 4, 5};`

数组的遍历

遍历

数组的遍历可以通过以下几种方式：

- 使用下标 `for (int i = 0; i < arraySize; i++) {...}`

数组的遍历

遍历

数组的遍历可以通过以下几种方式：

- 使用下标 `for (int i = 0; i < arraySize; i++) {...}`
- 使用指针 `for (int* p = arrayName; p < arrayName + arraySize; p++) {...}`

数组的传参

传参

数组的传参可以通过以下几种方式：

- 传递数组名 `void func(int a[]) {...}`

数组的传参

传参

数组的传参可以通过以下几种方式：

- 传递数组名 `void func(int a[]) {...}`
- 传递指针 `void func(int* a) {...}`

字符串的定义

定义

字符串是一种特殊的数组，它包含一系列字符，以 '\0' 结尾。字符串的元素可以通过下标访问。

一般的字符串定义如下：

```
char str[] = "Hello, World!";
```

其中 char 为字符类型，str 为字符串名。

字符串的操作

以字符串为例，字符串的操作包括：

- 初始化字符串 `char str[] = "Hello, World!";`

字符串的操作

以字符串为例，字符串的操作包括：

- 初始化字符串 `char str[] = "Hello, World!";`
- 访问字符串元素 `str[index];`

字符串的操作

以字符串为例，字符串的操作包括：

- 初始化字符串 `char str[] = "Hello, World!";`
- 访问字符串元素 `str[index];`
- 修改字符串元素 `str[index] = 'a';`

字符串的操作

以字符串为例，字符串的操作包括：

- 初始化字符串 `char str[] = "Hello, World!";`
- 访问字符串元素 `str[index];`
- 修改字符串元素 `str[index] = 'a';`
- 遍历字符串 `for (int i = 0; str[i] != '\0'; i++) {...}`

字符串的传参

传参

字符串的传参可以通过以下几种方式：

- 传递数组名 `void func(char str[]) {...}`

字符串的传参

传参

字符串的传参可以通过以下几种方式：

- 传递数组名 `void func(char str[]) {...}`
- 传递指针 `void func(char* str) {...}`

结构体的定义

定义

结构体是一种复杂类型，它包含一系列不同类型的元素，这些元素在内存中连续地分布。结构体的元素可以通过成员访问运算符 . 访问。

一般的结构体定义如下：

```
struct StructName {
    DataType1 member1;
    DataType2 member2;
    ...
};
typedef struct StructName StructName;
```

其中 DataType 为结构体的数据类型，StructName 为结构体名。

结构体的操作

以结构体为例，结构体的操作包括：

- 初始化结构体 `StructName structName = {value1, value2, ...};`

结构体的操作

以结构体为例，结构体的操作包括：

- 初始化结构体 `StructName structName = {value1, value2, ...};`
- 访问结构体成员 `structName.member;`

结构体的操作

以结构体为例，结构体的操作包括：

- 初始化结构体 `StructName structName = {value1, value2, ...};`
- 访问结构体成员 `structName.member;`
- 修改结构体成员 `structName.member = newValue;`

复杂类型的声明

- 以下声明中，哪些是合法的？如果合法，分别是什么意思？
  A. `int *p;`
  B. `int a[10];`
  C. `int *a[10];`
  D. `int (*a)[10];`
  E. `int *(*a)[10];`
  F. `int **a[10];`
  G. `int **a;`
  H. `int *a[];`

解答

A. `int *p;` 合法，p 是一个指向 int 类型的指针。

B. `int a[10];` 合法，a 是一个长度为 10 的 int 数组。

C. `int *a[10];` 合法，a 是一个长度为 10 的 int 指针数组。

D. `int (*a)[10];` 合法，a 是一个指向长度为 10 的 int 数组的指针。

E. `int *(*a)[10];` 合法，a 是一个指向长度为 10 的 int 指针数组的指针。

F. `int **a[10];` 合法，a 是一个长度为 10 的 int 指针指针数组。

G. `int **a;` 合法，a 是一个 int 指针指针。

H. `int *a[];` 不合法，a 是一个未知长度的 int 指针数组。

传参的两种方式

```
int func1(int HP) {
    return HP+1;
}
int func2(int* HP) {
    (*HP)++;
}
```

指针的本质

指针本质是一 32 位整数或 64 位整数，它存储的是一个内存单元的地址。若以 %d 形式输出，则与 `int` 没有区别
在 C 语言中，指针占据的内存空间是固定的，不受指向的数据类型的影响。一般为 4 字节（32 位系统）或 8 字节（64 位系统）。

链表的定义

定义

与数组相对，链表是一种链式结构，每个元素都包含一个指向下一个元素的指针。链表的元素可以在内存中不连续地分布，因此可以动态地分配内存。链表的元素称为节点。

一般的链表节点定义如下：

```
struct Node {
    DataType data;
    struct Node* next;
};
typedef struct Node Node, *List;
```

其中 DataType 为节点的数据类型，next 为指向下一个节点的指针。
Node 为节点类型 struct Node
List 为链表类型 struct Node*。

链表的操作

以带头节点的单链表为例，链表的操作包括：

- 创建链表 `List createList();`

链表的操作

以带头节点的单链表为例，链表的操作包括：

- 创建链表 `List createList();`
- 插入节点 `void insertNode(List L, DataType data);`

链表的操作

以带头节点的单链表为例，链表的操作包括：

- 创建链表 `List createList();`
- 插入节点 `void insertNode(List L, DataType data);`
- 删除节点 `void deleteNode(List L, DataType data);`

链表的操作

以带头节点的单链表为例，链表的操作包括：

- 创建链表 `List createList();`
- 插入节点 `void insertNode(List L, DataType data);`
- 删除节点 `void deleteNode(List L, DataType data);`
- 遍历链表 `void traverseList(List L);`

链表的操作

以带头节点的单链表为例，链表的操作包括：

- 创建链表 `List createList();`
- 插入节点 `void insertNode(List L, DataType data);`
- 删除节点 `void deleteNode(List L, DataType data);`
- 遍历链表 `void traverseList(List L);`
- 释放链表 `void freeList(List L);`

createList()

```
List createList() {
    List L = (List)malloc(sizeof(Node));
    L->next = NULL;
    return L;
}
```

insertNode()

```
void insertNode(List L, DataType data) {
    List p = L;
    while (p->next != NULL) p = p->next;
    List q = (List)malloc(sizeof(Node));
    q->data = data;
    q->next = NULL;
    p->next = q;
}
```

deleteNode()

```
void deleteNode(List L, DataType data) {
    List p = L;
    while (p->next != NULL && p->next->data != data) p =
p->next;
    if (p->next == NULL) return;
    List q = p->next;
    p->next = q->next;
    free(q);
}
```

traverseList()

```
void traverseList(List L) {
    List p = L->next;
    while (p != NULL) {
        printf("%d ", p->data);
        p = p->next;
    }
}
```

freeList()

```
void freeList(List L) {
    List p = L->next;
    while (p != NULL) {
        List q = p;
        p = p->next;
        free(q);
    }
    free(L);
}
```