4 复合类型

内容提要

- 字符串
- string类
- 指针和自由存储空间
- vector

2 C字符串

- 字符串是存储在内存的连续字节中的一系列字符
- C-风格字符串(C-style string)
  - 将字符串存储在char 数组中
  - 以空字符(null character) 结尾
    - 空字符被写作'\0'
    - 其ASCII码为0
    - 用来标记字符串的结尾

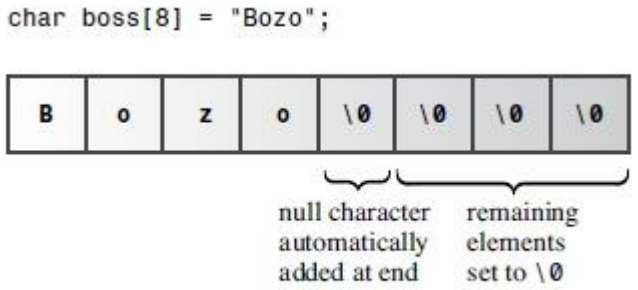

Figure 4.2 Initializing an array to a string.

2.2 在数组中使用字符串

P4.2 string.cpp

- #include <cstring> //C++版本
- 字符串长度
  - strlen
- 字符串数组长度
  - sizeof
- 单个字符索引
  - []

```
1.  #include <iostream>
2.  #include <cstring>  // for the strlen() function
3.  int main(){
4.      using namespace std;
5.      const int Size = 15;
6.      char name1[Size];               // empty array
7.      char name2[Size] = "C++owboy";
8.      cout << "Howdy! I'm " << name2;
9.      cout << "! What's your name?\n";
10.     cin >> name1;
11.     cout << "Well, " << name1 << ", your name has ";
12.     cout << strlen(name1) << " letters \n";
13.     cout << "in an array of " << sizeof(name1);
14.     cout << "Your initial is " << name1[0] << ".\n";
15.     name2[3] = '\0'; // set to null character
16.     cout << name2 << endl;

17.     return 0;
18. }
```

2.3 字符串输入

P4.3 instr1.cpp

- #include <iostream> //表2.1
- using namespace std;
- cout
  - <<
- cin
  - >>
- endl
- cin使用空白来确定字符串的结束位置

2.4 每次读取一行字符串输入

P4.4 instr2.cpp

- 面向行的输入
  - cin.getline(buffer, bufferSize);
  - 每次读取一行，通过换行符确定行尾，但不保存换行符。而是直接替换成'\0'
- cin.get()….
- 混合输入数字，字符串
  - 一种数据一行。。。

```
1.  int main()
2.  {
3.      using namespace std;
4.      const int ArSize = 20;
5.      char name[ArSize];
6.      char dessert[ArSize];

7.      cout << "Enter your name:\n";
8.      cin.getline(name, ArSize);
9.      cout << "Enter your favorite dessert:\n";
10.     cin.getline(dessert, ArSize);
11.     cout << "I have some delicious " << dessert;
12.     cout << " for you, " << name << ".\n";

13.     return 0;
14. }
```

混合输入数字和字符串

P4.6 numstr.cpp

- cin>>year; //输入year
- cin.get();//取出并丢弃换行符
- cin.getline(address, 80);

3 string 类

- 将字符串作为一种数据类型

- 初始化
- 复制，拼接和附加
- 其它操作
- 字符比较

示例

- 基础（初始化等）
  - P4.7 strtype1.cpp
- 赋值、拼接和附加
  - P4.8 strtype2.cpp
- 其它操作（<cstring>）
  - P4.9 strtype3.cpp
  - 使用字符数组时，存在数组过小、无法存储指定信息的危险
- IO
  - P4.10 strtype4.cpp
  - cin.getline

```
1.  #include <iostream>
2.  #include <string> // make string class available
3.  int main(){
4.      using namespace std;
5.      string s1 = "penguin";
6.      string s2, s3;
7.      s2 = s1;
8.      s2 = "buzzard";
9.      s3 = s1 + s2;
10.     s1 += s2;
11.     s2 += " for a day";
12.     return 0;
13. }
```

数据的3种基本属性

- 信息存储在何处
- 存储的值为多少
- 存储的信息是什么类型

- 基本策略：定义一个变量
  - 声明语句指出了值的类型和符号名
  - 让程序为值分配内存
  - 并在内部跟踪该内存单元。

指针和自由存储空间

- 新的策略
  - 指针为基础，指针是一个变量，其存储的是值的地址，而不是值本身
- 常规变量的地址（P4.14 address.cpp）
  - 对变量应用地址运算符（&）
  - 值是指定的量，而地址为派生量
- 处理存储数据的策略（P4.15 pointer.cpp）
  - 将地址视为指定的量，将值视为派生量
  - *运算符
    - 间接值(indirect value) 或解除引用(dereferencing)运算符。应用于指针，可以得到该地址处存储的值
    - 注意：在C++ 中， int*是一种复合类型，是指向int 的指针。
    - 警告：一定要在对指针应用解除引用运算符（*）之前，将指针初始化为一个确定的、适当的地址

```
1.  #include <iostream>
2.  int main(){
3.      using namespace std;
4.      int updates = 6;  // declare a variable
5.      int * p_updates;  // declare pointer to an int
6.      p_updates = &updates;
7.      // express values two ways
8.      // express address two ways
9.      cout << "Addresses: &updates = " << &updates;
10.     cout << ", p_updates = " << p_updates << endl;
11.     // use pointer to change value
12.     *p_updates = *p_updates + 1;
13.     cout << "Now updates = " << updates << endl;
14.     // cin.get();
15.     return 0;
16. }
```

使用new来分配内存

- 指针真正的用武之地在于，在运行阶段分配未命名的内存以存储值
  - 在C 语言中，可以用库函数malloc()来分配内存
- 用法(P4.17 use_new.cpp)
  - typeName * pointer_name = new typeName; //申请内存
  - delete pointer_name; //释放内存

- 释放内存：
  - 一定要配对地使用new 和delete；否则将发生内存泄溯(memory leak)
    - 即被分配的内存再也无法被管理（继续使用）。循环中出现，容易导致内存耗光
  - 对空指针使用delete 是安全的。
  - delete实际上针对的是new出来的地址（不一定是那个pointer_name指针）

使用new来创建动态数组

- 为什么new？
  - new 从称为堆(heap) 或自由存储区(free store)的内存区域中分配内存
    - 比较大，和系统内存大小有关
  - 常规变量声明分配的内存块，都存储在被称为栈(stack) 的内存区域中
    - 比较小，和设置有关，所以不能类似定义数组：int arr[100000];
- 使用new 创建动态数组（P4.18 arraynew.cpp）
  - typeName * pointer_name = new typeName[len]; //申请内存，长度为len
  - delete[] pointer_name; //释放内存
- 使用动态数组
  - 把指针当作数组名使用即可

```
1.  #include <iostream>
2.  int main()
3.  {
4.      using namespace std;
5.      double * p3 = new double [3];
6.      p3[0] = 0.2;
7.      p3[1] = 0.5;
8.      p3[2] = 0.8;
9.      cout << "p3[1] is " << p3[1] << ".\n";
10.     p3 = p3 + 1; // increment the pointer
11.     cout << "Now p3[0] is " << p3[0];
12.     cout << "p3[1] is " << p3[1] << ".\n";
13.     p3 = p3 - 1;  // point back to beginning
14.     delete [] p3; // free the memory
15.     return 0;
16. }
```

指针与C++基本原理

- OOP 强调的是在运行阶段（而不是编译阶段）进行决策。
- 运行阶段指的是程序正在运行时，编译阶段指的是编译器将程序组合起来时。

- 运行阶段决策提供了灵活性，可以根据当时的情况进行调整
  - 如，在运行阶段确定数组的长度

8 指针、数组和指针算术

P4.19 addpntrs.cpp

- 指针和数组基本等价
  - 原因在于指针算术(pointer arithmetic) 和C++内部处理数组的方式
    - 指针变量加1 后，增加的量等于它指向的类型的字节数。
  - C++将数组名解释为数组第1 个元素的地址
    - arrayname[i] becomes *(arrayname + i)
- 指针和数组区别
  - 可以修改指针的值，而数组名是常量:
  - 对数组应用sizeof 运算符得到的是数组的长度，而对指针应用sizeof 得到的是指针的长度，即使指针指向的是一个数组

数组的地址（***）

- 数组名被解释为其第1个元素的地址，而对数组名应用地址运算符时，得到的是整个数组的地址

```
short tell [10];   // tell an array of 20 bytes
cout<<tell<<endl;  // displays &tell[O]
cout<<&tell<<endl; // displays address of whole array
```

- 从数字上说，这两个地址相同；
- 从概念上说， &tell[0]（即tell）是一个2字节内存块的地址，而&tell 是一个20字节内存块的地址
- 因此，表达式tell+1 将地址值加2，而表达式&tell+1 将地址加20
  - 换句话说， tell是一个short 指针(short*)，而&tell 指向包含20 个元素的short 数组(short (*) [20])。

指针小结

- 声明指针
- 给指针赋值
- 对指针解除引用
- 区分指针和指针所指向的值
- 数组名
- 指针算术
- 数组的动态联编和静态联编
- 数组表示法和指针表示法

自动存储、静态存储和动态存储

- 根据用于分配内存的方法， C++有3种管理数据内存的方式：
  - 自动存储
  - 静态存储
  - 动态存储（有时也叫作自由存储空间或堆）

- 自动存储（存储在栈中）
  - 在函数内部定义的常规变量使用自动存储空间，被称为自动变量(automatic variable),意味着它们在所属的函数被调用时自动产生，在该函数结束时消亡。
  - 自动变量是一个局部变量，其作用域为包含它的代码块。代码块是被包含在花括号中的一段代码
  - 自动变量通常存储在栈中。这意味着执行代码块时，其中的变量将依次加入到栈中，而在离开代码块时，将按相反的顺序释放这些变量，这被称为后进先出(LIFO)

自动存储、静态存储和动态存储

- 静态存储
  - 静态存储是整个程序执行期间都存在的存储方式。
  - 使变量成为静态的方式有两种
    - 一种是在函数外面定义它；
    - 另一种是在声明变量时使用关键字static:
    - `static double fee= 56.50;`
- 自动存储和静态存储的关键在于
  - 这些方法严格地限制了变量的寿命。变量可能存在于程序的整个生命周期（静态变量），也可能只是在特定函数被执行时存在（自动变量）。

自动存储、静态存储和动态存储

- 动态存储
  - new 和delete 运算符提供了一种比自动变量和静态变量更灵活的方法。
  - 它们管理了一个内存池，这在C++中被称为自由存储空间(free store) 或堆(heap)。该内存池同用于静态变量和自动变量的内存是分开的。
  - P4.22 delete.cpp 表明， new 和delete 能够在一个函数中分配内存，而在另一个函数中释放它。因此，数据的生命周期不完全受程序或函数的生存时间控制。

```
1.  char * getname(void); // function prototype
2.  int main(){
3.      char * name;
        name = getname();
4.      cout << name << " at " << (int *) name;
5.      delete [] name;   // memory freed

6.      name = getname(); // reuse freed memory
7.      cout << name << " at " << (int *) name;
8.      delete [] name;   // memory freed again
9.      return 0;
10. }
11. char * getname(){   // return pointer to new string
12.     char temp[80];  // temporary storage
13.     cin >> temp;
14.     char * pn = new char[strlen(temp) + 1];
15.     strcpy(pn, temp);
16.     return pn; // temp lost when function ends
17. }
```

10 数组的替代品

4.24 choices.cpp

- vector
  - vector<typeName> vt(n_elem);
  - 参数n_elem 可以是整型常量，也可以是整型变量。

```
#include <vector>
using namespace std;
vector<int> vi; // create a zero-size array
int n;
cin >> n;
vector<double> vd(n); // create an array
```

- 之后使用和数组一样

```
1.  #include <vector>    // STL C++98
2.  #include <array>     // C++0x
3.  int main(){
4.      using namespace std;
5.      // C, original C++
6.      double a1[4] = {1.2, 2.4, 3.6, 4.8};
7.      // C++98 STL
8.      vector<double> a2(4);// create vector
9.     // no simple way to initialize in C98
10.     a2[0] = 1.0/3.0;
11.    // C++0x -- create and initialize array object
12.     array<double, 4> a3 = {3.14, 2.72, 1.62, .41};
13.     array<double, 4> a4;
14.     a4 = a3; // valid for array objects of same size
15.     // use array notation
16.     cout << "a1[2]: " << a1[2] << &a1[2] << endl;
17.     a1[-2] = 20.2;
18.     cout << "a1[-2]: " << a1[-2] <<" at " << &a1[-2];
19. }
```