第21章 内存访问

数组和指针是C程序中低级的构建块，它们提供了对硬件内存的快速访问。本章将探讨组织和访问内存的不同方式。

对内存的直接操作存在很大的风险，要求有很好的经验和全面的测试，以避免出现严重的错误。误用指针和动态内存，可以使程序崩溃，导致堆冲突和内存泄漏。堆错误尤其难以调试，因为它通常会导致段错误(segmentation fault)，挂起程序所在的代码点，可能会远离出现损坏的堆所在的那一点。

Qt 和标准库容器类都允许安全地使用动态内存而不会影响到性能[①]。数组实现了大多数容器类，这种实现对客户代码是隐藏的。这些安全因素来自于对每一个容器API的精心设计，使得不允许存在可能导致内存问题的动作。

Qt提供了许多容器，从高级模板类(如6.8节中讨论过的一个)到低级容器(如`QBitArray`和`QByteArray`)。

通常而言，当编写复用这些容器的程序时，很容易就能完全避免使用数组。如果无法使用Qt，或者需要编写一个与C代码的接口，则可能需要使用数组和指针，并要直接操作所分配的内存。

现代的软件经常包含彩色图形以及声音，对于它们的执行要求能够快速地处理。这通常意味着需大量使用动态内存。在一般的计算设备上，已经有大容量的内存和辅助存储设备可用——就在几年前，这还是不可想像的。尽管如此，图形、动画和声音所要求的大量内存，都必须小心而有效率地处理。这就是本章关注内存资源的正确管理以及分析错误管理如何会导致严重后果的原因。

21.1 指针误用

1.15节中讲解过指针，并演示过它的一些基本用法。下面给出的两个短的代码示例表明，如果没有正确地处理指针，就可能发生奇怪的和危险的事情。示例21.1中给出了声明指针的多种方法。

示例21.1 src/pointers/pathology/pathologydecls1.cpp

```
[ . . . . ]

int main() {
    int a, b, c;                    1
    int* d, e, f;                   2
    int *g, *h;                     3
    int* i, * j;                    4
```

① 以后，当使用术语“容器”时，如果没有进一步的限制，指的就是Qt或者标准库容器。

```
    return 0;
}
```

1 正如所期望的，这一行会创建三个 int 变量。

2 这一行创建了一个指向 int 变量的指针和两个 int 变量。

3 这一行创建了指向 int 变量的两个指针。

4 这一行也创建了指向 int 变量的两个指针。

初学者会想当然地认为 main()中的第二行会创建三个指针——毕竟在第一行中，同样的语法创建了三个 int 变量。但是，当在一行中声明多个变量时，星号类型修饰符只会作用于紧跟在它后面的那个变量，而不会作用于前面的那个类型符号。正是由于这个原因，推荐对每一个指针都用一个单独的声明(单独一行)。示例 21.2 中包含了三组语句。

示例 21.2 src/pointers/pathology/pathologydecls2.cpp

```
[ . . . . ]
int main() {
    int myint = 5;
    int* ptr1 = &myint;
    cout << "*ptr1 = " << *ptr1 << endl;
    int anotherint = 6;
//  *ptr1 = &anotherint;                                    1

    int* ptr2;                                              2
    cout << "*ptr2 = " << *ptr2 << endl;
    *ptr2 = anotherint;                                     3

    int yetanotherint = 7;
    int* ptr3;
    ptr3 = &yetanotherint;                                  4
    cout << "*ptr3 = " << *ptr3 << endl;
    *ptr1 = *ptr2;                                          5
    cout << "*ptr1 = " << *ptr1 << endl;

    return 0;
}
[ . . . . ]
```

1 错误，从 int*到 int 的无效转换。

2 未初始化的指针。

3 不可预测的结果。

4 常规的赋值。

5 危险的赋值。

这段代码中存在一些严重的指针问题，其中最坏的情况没有被编译器检测到。首先出现的是一个简单的类型失配问题。编译结果如下：

```
src/pointers/pathology> g++ pathologydecls2.cpp
pathologydecls.cpp: In function `int main()':
pathologydecls.cpp:17: error: invalid conversion from `int*' to `int'
src/pointers/pathology>
```

将这个无效转换操作注释掉之后，再次尝试编译：

```
*ptr1 = 5
*ptr2 = -1218777888
*ptr3 = 7
*ptr1 = 6
Segmentation fault
```

对未初始化的指针 ptr2 进行解引用操作，会导致不可预测的(即未定义的)结果。

为读而解引用未初始化的指针，就已经是一种很不好的操作，更不用说写这种指针了。这是内存冲突(memory corruption)的一种形式，它能够导致程序在以后执行时出现问题。段错误是由解引用 ptr2 时发生的内存冲突而引起的。

段错误还不是最坏的情况。至少，它会警告程序员存在一个严重的问题。更糟糕的情形是，如果不中断程序的执行而是让它运行，则会产生错误的、看似合理的结果。没有几个人愿意花时间和精力来检验计算机的运算结果！如果发生内存冲突，则任何事情都有可能发生。

21.2 带有堆内存的更多指针误用

对拥有堆中一个有效对象的地址的指针进行删除操作的结果，是将这个堆内存的状态从“使用中”变为“可用”。对指针进行删除操作后，指针本身的状态是未定义的。它也许依旧会保存所删除内存的地址，也许不会。所以如果再次对同一个指针进行删除操作会导致运行时问题，可能是堆冲突。

通常而言，编译器无法检测到对同一个对象的多次删除操作，尤其是当内存块(或其一部分)被重新分配时。为了避免这种重复删除所带来的不良后果，一种好的做法是在删除指针后立即对它赋值 0 或者 NULL。

对空指针执行删除操作，不会有任何动作，也不会有错误发生。

对不是由 new 操作返回的非空指针执行删除操作，会导致未定义的结果。通常而言，编译器无法判断出指针是否是由 new 操作返回的，所以可能导致未定义的运行时行为。需牢记的一条根本原则是：正确地使用删除操作是程序员的责任。

运行时错误的最多来源之一是内存泄漏。如果程序分配了内存，但随后丢失了它的踪迹，导致既无法访问也不能删除它，这就是内存泄漏。没有被正确地删除的对象，在进程终止之前将一直占据内存。

有些程序(例如，服务器程序、操作系统程序)会长时间保持有效状态。假设这样的程序中包含了一个经常要执行的例程，每次运行它时都会导致内存泄漏。由于充满了这些不可访问的、未删除的内存块，堆会逐渐变得支离破碎。在某一刻，如果有例程需要大量的连续动态内存，就有可能拒绝这种请求。如果程序没有为处理这种事件做好准备，它就会中断。

运算符 new 和 delete 使 C++程序员具备更强的能力，同时也增加了责任性。

以下是演示内存泄漏的一些代码样本。在定义完两个指针后，内存应如图 21.1 所示。

```
int* ip = new int;          // allocate space for an int
int* jp = new int(13);      // allocate and initialize
cout << ip  << '\t'  <<  jp  << endl;
```

执行完下面的代码行之后，内存应如图 21.2 所示。

```
jp = new int(3);    // reassign the pointer - MEMORY LEAK!!
```

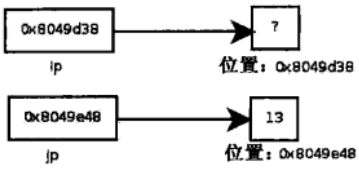

图 21.1 内存中的初始值

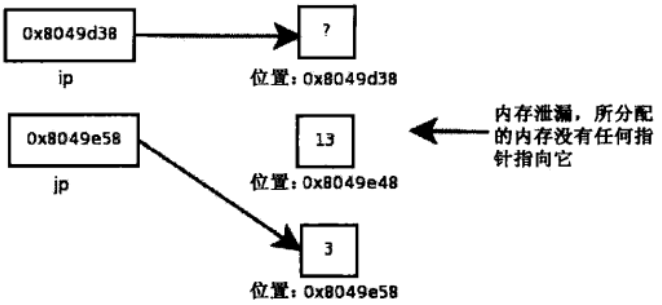

图 21.2 发生内存泄漏后的结果

示例 21.3 中，对指针 jp 删除了两次。

示例 21.3 src/pointers/pathology/pathologydemo1.cpp

```
#include <iostream>
using namespace std;

int main() {
    int* jp = new int(13);                                    1
    cout << jp << '\t' << *jp << endl;
    delete jp;
    delete jp;                                                2
    jp = new int(3);                                          3
    cout << jp << '\t' << *jp << endl;
    jp = new int(10);                                         4
    cout << jp << '\t' << *jp << endl;
    int* kp = new int(17);
    cout << kp << '\t' << *kp << endl;
    return 0;
}
```

1 分配并初始化。

2 错误：指针已经被删除。

3 重新分配指针，导致内存泄漏。

4 重新分配指针，导致内存泄漏。

输出如下所示。

```
OOP> g++ pathologydemo1.cpp
OOP> ./a.out
0x8049e08       13
0x8049e08       3
0x8049e08       10
Segmentation fault
OOP>
```

第二个删除操作是一种严重的错误，但是编译器不会报错。这个错误会损坏堆，使得无法进行进一步的内存分配，导致此后的程序行为是未定义的。例如，如果在重新为指针 jp 分配内存时出现内存泄漏，就不会再获得新的内存空间了。当试图使用另一个指针变量时，就会发生段错误。这是一种未定义的行为，且在不同的平台或者不同的编译器下有不同的表现。

21.3 内存访问小结

以下是关于内存访问的最重要的几点：

- 运算符 new 和 delete 使 C++程序员具备更强的能力，同时也增加了责任性。
- 误用指针和动态内存，可以使程序崩溃，导致堆冲突和内存泄漏。
- Qt 和标准库容器类都允许安全地使用动态内存而不会影响到性能。
- 当在一行中声明多个变量时，星号类型修饰符只会作用于紧跟在它后面的那个变量，而不会作用于前面的那个类型符号。
- 解引用一个未初始化的指针，是一种不会被编译器捕获的严重错误。
- 对指针进行删除操作后，指针本身的状态是未定义的。
- 一种好的做法是在删除指针后立即对它赋值 0 或者 NULL。
- 对不是由 new 操作返回的非空指针执行删除操作，会导致未定义的结果。
- 编译器无法判断出 delete 的错误使用，所以正确地使用删除操作是程序员的责任。
- 如果程序分配了内存，但随后丢失了它的踪迹，导致既无法访问也不能删除它，这就是内存泄漏。

21.4 数组简介

数组是一组连续的内存单元，这些内存单元具有相同的大小。每一个单元被称为数组元素或者数组项。

当声明数组时，必须告知数组的大小。数组大小可以显式地给出，也可以通过初始化来确定：

```
int a[10]; // explicitly creates uninitialized cells a[0], a[1],..., a[9]
int b[] = {1,3,5,7}; // implicitly creates and initializes b[0],..., b[3]
```

数组名称是指向数组第一个单元的 const 类型的指针的别名。指针声明

```
int* ptr;
```

只会创建这个指针变量。指针变量没有自动的默认初始化值。如果解引用未初始化的指针，则是一个错误。

指针索引是从基地址开始的相对偏移量：

```
a[k]等价于*(a + k)
```

示例 21.4 中给出了数组索引的一个有趣特性。

示例 21.4 src/pointers/pathology/pathologydemo2.cpp

```
#include <iostream>
using namespace std;
int main(){
  int a[] = {10, 11, 12, 13, 14, 15};

  int* b = a + 1;
  cout << "a[3] = " << a[3] << '\n'
       << "b[3] = " << b[3] << endl;
```

```
    //It gets even worse.
    int c = 123;
    int* d = &c;
    cout << "d[0] = " << d[0] << '\n'
         << "d[1] = " << d[1] << '\n'
         << "d[2] = " << d[2] << endl;

}
```

编译并运行这个程序的输出如下。

```
pointers/pathology> g++ -ansi -pedantic -Wall pathologydemo2.cpp
pointers/pathology>
a[3] = 13
b[3] = 14
d[0] = 123
d[1] = -1075775392
d[2] = -1219610235
```

注意，b 和 d 都没有被声明成数组，但是编译器允许使用使用下标运算符[]。c 是一个常规的 int 变量，而 d 是一个常规的 int 指针。d[0]，d[1]和 d[2]没有定义，但是编译器不会对它们的出现给出任何警告——尽管使用了三个命令行选项。

这是一种特殊的语法，用于定义由给定数量的某种类型的元素组成的动态数组：

```
uint n;
ArrayType* pt;
pt = new ArrayType[n];
```

这种 new 操作会分配 *n* 个连续的内存块，每一个内存块的大小为 sizeof(ArrayType)，并会返回一个指向第一个内存块的指针。新分配的数组的每一个元素都会被默认初始化。为了正确地解分配(deallocate)这个数组，需要使用语法：

```
delete[] pt;
```

进行 delete 操作删除动态数组时，如果没有包含空的方括号，则会导致未定义的结果。

关于系统无法完成动态内存请求时所发生的异常以及其他动作，在 dist 目录下单独的一篇文章中进行了讨论[①]。

21.5 指针的算术运算

对指针进行+, -, ++或--运算的结果，与它所指向的对象类型有关。当算术运算符用于类型为 *T**的指针 p 时，假定 p 指向的数组元素为 *T* 类型的对象。

- p + 1 指向数组中的下一个元素。
- p - 1 指向数组中的前一个元素。
- 通常而言，p + k 的地址值会比 p 的地址值大 k * sizeof(T)字节。

指针的减法只有当两个指针都指向同一个数组中的元素时才有意义。这种情况下，其结果是一个与两个元素间数组元素的个数等价的 int 值。

① 参见 articles/exceptions.html。

在数组环境之外进行的指针算术运算结果是未定义的，程序员有责任确保正确地使用了指针算术运算。示例21.5中演示了这种情况。

示例21.5 src/arrays/pointerArith.cpp

```
[ . . . . ]
int main()  {
    using namespace std;
    int y[] = {3, 6, 9};
    int x = 12;
    int* px;
    px = y;                                                    1
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    cout << "What's next: " << *++px << endl;
    return 0;
}
```

1 y或者任何数组名称，是指向数组第一个元素的指针的别名。

编译并运行这个程序的输出如下[①]。

```
src/arrays> g++ -ansi -pedantic -Wall pointerArith.cpp
pointerArith.cpp: In function 'int main()':
pointerArith.cpp:6: warning: unused variable 'x'
src/arrays> ./a.out
What's next: 6
What's next: 9
What's next: -1080256152
What's next: 12
```

注意，编译器和运行时系统都不会给出错误消息。但是，给出了没有使用变量x的一个警告。C++很乐意读取任意内存地址，并以所选择的类型报告它们，这样就在为C++开发人员提供强大功能的同时，也为犯错提供了大量机会。

21.6 数组，函数与返回值

正如C语言中那样，为函数所声明的返回类型不能是数组(例如，不能是int[]，char[]或Point[]形式)。从函数返回数组(的地址)为指针类型，这是允许的。但是，对类的公共接口不推荐这样做。

我们已经看到，数组是一个没有受保护的内存块。封装了这块内存的类，不应具有返回指向它的指针的公共成员。如果这样做，客户代码就有可能错误地使用这块内存。正确设计的类，会全面封装它的实现中使用的任何数组与这块内存的交互操作。

数组从来不会将值传递给函数。也就是说，数组元素不会被复制。如果调用函数时其实参表中包含数组，例如，

```
int a[] = {10, 11, 12, 13, 14, 15};
void f(int a[]) {
```

① 通常而言，访问位于数组边界之外的内存会导致未定义的结果，这里由于是故意这样做的，所以结果是未定义的。

```
    [ ... ]
}
    [ ... ]
f(a);
```

则实际传递的值仅仅是数组中第一个元素的指针。示例 21.6 演示了这一点，输出了传递给函数的数组和返回的数组。

示例 21.6 src/arrays/returningpointers.cpp

```
#include <assert.h>

int paramSize;

void bar(int* integers) {
    integers[2]=3;                                          1
}

int* foo(int arrayparameter[]) {
    using namespace std;
    paramSize = sizeof(arrayparameter);
    bar(arrayparameter);                                    2
    return arrayparameter;                                  3
}

int main(int argc, char** argv) {
    int intarray2[40] = {9,9,9,9,9,9,9,2,1};
    char chararray[20] = "Hello World";                     4
    int intarray1[20];                                      5
    int* retval;                                            6

//  intarray1 = foo(intarray2);                             7

    retval = foo(intarray2);
    assert (retval[2] == 3);
    assert (retval[2] = intarray2[2]);
    assert (retval == intarray2);
    int refSize = getSize(intarray2);
    assert(refSize == paramSize);
    return 0;
}
```

1 改变输入数组中的第三个元素。
2 用指针将数组传递给函数。
3 从函数将数组当作指针返回。
4 用于初始化 char 数组的特殊语法。
5 未初始化的内存。
6 未初始化的指针。
7 错误：intarray1 类似 char* const，不能这样赋值。

21.7 不同类型的数组

基本类型的数组，例如 int，char 和 byte 类型的数组，被用来实现缓存。出于与 C 语言中 struct 数组后向兼容的考虑，对象数组在 C++语言中是支持的，但是只能将它用在同一种结构的相同集合中，而不能用于相似的多态对象的集合中。

如果需要随机访问所保存的项目，(Qt 中的)QList 或者(STL 中的)vector 都可以用来替换数组。在底层，它们都是用动态数组实现的。只要有可能，就应优先使用这样的容器而不是数组，因为容器能够正确且安全地分配和释放内存。

21.8 有效的指针操作

以下给出的是能够正确地在指针上执行的操作。

创建——指针的初始值可以有三种来源：

- 由所声明的指针变量或者 const 指针(例如，数组名称)获得的栈地址。
- 由取址运算符&获得的地址。
- 由动态内存分配运算符(如 new)获得的堆地址。

赋值

- 指针能够被赋予同一种类型或者派生类型的指针所保存的地址。
- 可以将 void*类型的变量赋予任何类型的指针，不需要显式的转型。
- 将另一种(或者非派生)类型的指针所保存的地址赋予非 void*类型的指针时，只能使用显式的转型。
- 数组名称就是一个 const 指针，不能被赋值。
- 可以将 NULL 指针(值为 0)赋予任何指针(Stroustrup 推荐使用 0 而不是 NULL)。

算术运算

- 可以对指针进行增 1 操作或者减 1 操作，即 p++或者 p--。
- 可以将指针与整数相加或者相减，即 p + k 或者 p - k。
- 只有当得到的指针值位于同一个数组的范围之内时，这种加减操作才是被定义的。唯一的例外情况是：只要指针不试图解引用位于数组末尾后一个位置处的内存单元地址，这个指针就被允许指向这个位置。
- 两个指针可以进行减法操作。指向同一数组两个成员的两个指针相减后，其结果是表示两个元素之间的数组元素数量的一个 int 值。

比较

- 指向同一个数组中不同元素的两个指针，可以用==, !=, <, >等运算符进行比较。
- 任何指针都能够与 0 进行比较。

间接赋值

- 如果 p 为 *T**类型的指针，则*p 为 *T* 类型的变量，且可以将其用于赋值运算的左边。

索引

- 指针 p 可以用于数组索引运算符 p[i]，其中 i 为 int 类型。编译器会将这样的表达式解释为*(p + i)。
- 索引只有用于数组时才有意义并可定义，但是编译器不会阻止它用于非数组指针中，其结果是未定义的。

示例 21.7 中清楚地演示了最后一种情况。

示例 21.7 src/arrays/pointerIndex.cpp

```
#include <iostream>
using namespace std;

int main()  {
    int x = 23;
    int y = 45;
    int* px = &x;
    cout << "px[0] = " << px[0] << endl;
    cout << "px[1] = " << px[1] << endl;
    cout << "px[2] = " << px[2] << endl;
    cout << "px[-1] = " << px[-1] << endl;
    return 0;
}
```

输出如下所示。

```
// g++ on Mac OSX:
px[0] = 23
px[1] = 1606413624
px[2] = 32767
px[-1] = 45

// g++ on Linux (Ubuntu):
px[0] = 23
px[1] = -1219095387
px[2] = -1216405456
px[-1] = 45

// g++ on Windows XP (mingw)
px[0] = 23
px[1] = 45
px[2] = 2293588
px[-1] = 2009291924

// Windows XP with MS Visual Studio compiler:
px[0] = 23
px[1] = 45
px[2] = 1245112
px[-1] = 1245024
```

有一个小型的具体例子，它给出了“未定义行为”的含义。对初学者而言，对于程序栈中连续定义的变量是如何排列的，可以原谅他们对此做出的一些假设。对非数组指针使用数组下标，或者使用的下标超出了数组的范围，只不过是暴露这些假设的情形之一。现在，尝

试使用下标-1，然后努力使不同平台上的不同结果保持一致。这个例子太短了，以至于无法给出“未定义行为”的全貌。指针的未定义行为，通常会导致内存冲突，而内存冲突是编程的噩梦——程序员更期望看到的结果是运行时中断。

21.9 数组与内存

以下是本章中已经提到的最重要的几点：

- 数组是一组连续的内存单元，这些内存单元具有相同的大小。
- 数组名称是指向数组第一个单元的 const 类型的指针的别名。
- 指针变量没有自动的默认初始化值。
- 指针索引是从基地址开始的相对偏移量。
- 只有当用来访问数组的成员，且访问是位于数组范围之内时，数组下标才是有效的。
- C++标准不保证编译器会捕获到将指针用于非数组的下标运算符的企图。
- 将数组传递给函数和从函数返回，是通过指针进行的。
- 可以将算术运算符+, -, ++和--运用到数组指针，只要其结果是有效的。
- 在数组环境之外进行的指针算术运算结果是未定义的。
- C++标准不保证编译器能够捕获误用指针运算的企图。
- 指针只能通过如下方式获得值：
  - 在创建时初始化。
  - 创建后对其赋值。
  - 作为指针运算的结果。
- *ArrayType* 的 size 元素的动态数组是用如下语法分配内存的：

```
uint size;
ArrayType* pt;
pt = new ArrayType[size] ;
```

- 当为数组分配内存时，动态数组的每一个元素都会被默认初始化。
- 为了正确地解分配这个动态数组，需要使用语法：

```
delete[] pt;
```

如果无法执行分配内存的请求，ANSI/ISO 标准要求 new 运算符抛出 bad_alloc 异常而不是返回 NULL。关于异常的更多细节，请查看 dist 目录下的一篇文章[①]。如果无法执行分配内存的请求，则运算符 new(nothrow) 可以返回 0。应将动态数组小心地封装到类中，类应当包含合适的析构函数、复制构造函数以及复制赋值构造函数。

21.10 练习：内存访问

预测示例 21.8 的输出，然后链编并运行它。解释输出的结果。如果与你的预测有所不同，给出理由。

① 参见 articles/exceptions.html。

示例 21.8 src/arrays/arrayVSptr.cpp

```
#include <iostream>
using namespace std;

int main()  {
   int a[] = {12, 34, 56, 78};
   cout << a << "\t" << &a[1] - a << endl;
   int x = 99;
     a = &x;
   int* pa;
   cout << pa << endl;
   pa = &x;
   cout << pa << "\t" << pa - &a[3] << endl;
   cout << a[4] << "\t" << a[5] << endl;
   cout << *(a + 2) << "\t" << sizeof(int) <<endl;
   void* pv = a;
   cout << pv << endl;
   int* pi = static_cast<int*>(pv);
   cout << *(pi + 2) << endl;
   return 0;
```

21.11 复习题

1. 下面的语句中定义了什么?

   ```
   int* p, q;
   ```

2. 什么是内存泄漏? 它是如何发生的?
3. 比较将+, -, ++, --运算符用于指针和将它们用于 int 或者 double 类型的值上的不同。
4. 如果将 delete 运算符用于已经被删除的指针上，会发生什么?
5. 如果将数组作为参数传递给函数，程序栈中被复制的是什么?
6. 什么是动态内存? C++中如何获得它?