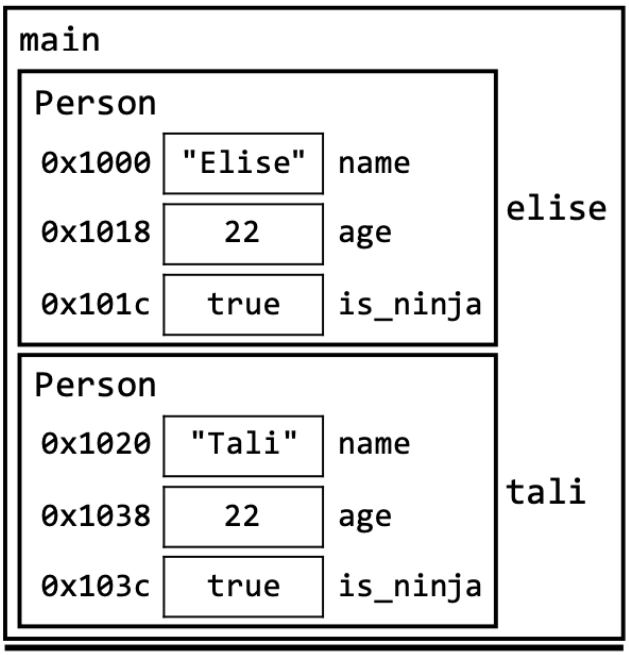
10.1 Struct



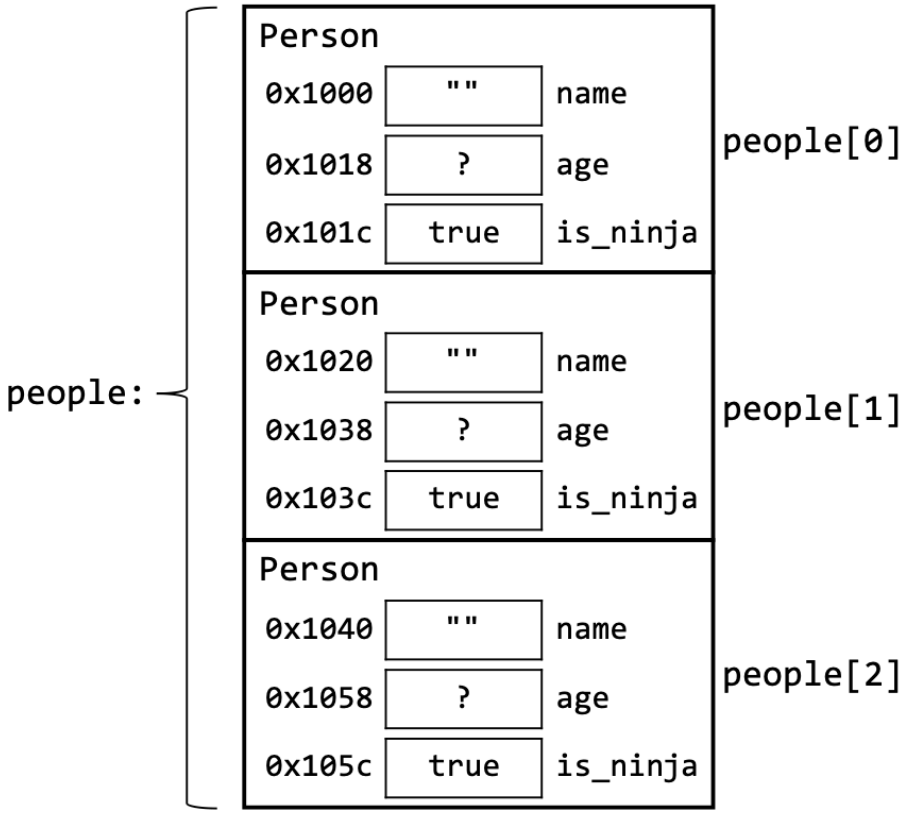


Figure 12.6: An array of class-type objects.

`struct` 就是我们说的 C-Style 的 ADT.

An **abstract data type (ADT)** separates the interface of a data type from its implementation, and it **encompasses both the data itself as well as functionality on the data.**

10.1.1 Definition, Creating Instance and initialize

```
struct Person {
     int age;
     string name;
     bool isNinja;
}; // 注意;

int main() {
     int x;
     Person alex;
     alex = { 75, "granny", false };

     Person jon = { 25, "jon", true };
}
```

We can define a new compound object *type* via a **struct definition** (e.g. `Person`). We define objects as **instances** of that type.

The `struct` definition contains **member variable declarations**, member variables define 这个 compound object 的 subobjects.

可以先创造 object, 但是不 initialize variables.(如图)

10.1.2 Member Access: `.`

Individual members are accessed via the `.` operator (also called the "**member access operator**").

```
struct Person {
     int age;
     string name;
     bool isNinja;
};

int main() {
     Person p = { 17, "Ron", true };
     p.isNinja = true;
}
```

10.1.3 添加 `const` 的 instance

If you've got **a `struct` object that is const-qualified**, that **forbids assignment to the struct as a whole and also forbids assignment to its individual members.**

所有member都不能更改值，整个 instance 也不行.

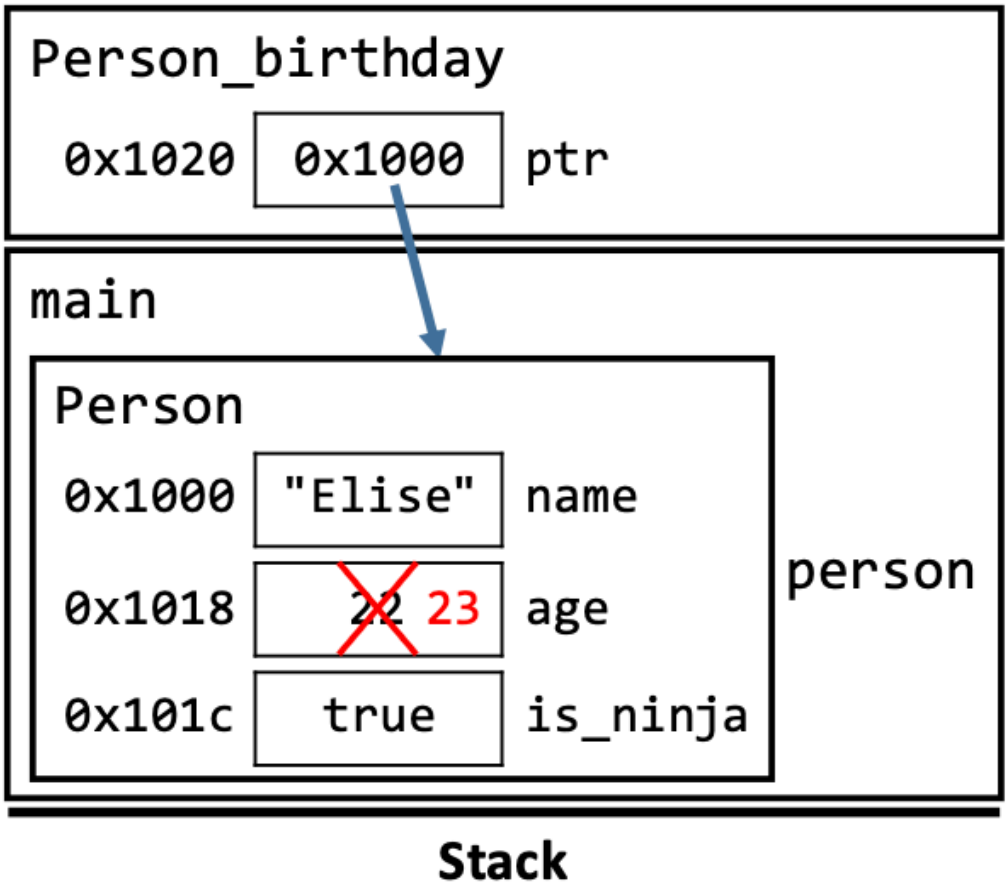
```
struct Person {
     int age;
     string name;
     bool isNinja;
};
int main() {
     const Person p1 = { 17, "Kim", true };
     Person p2 = { 17, "Ron", true };

     p1.isNinja = false; // error
     p1 = p2; // error
}
```

并且即便没有 initialize 也不能改.

```
int main() {
     const Person p1;
     p1.isNinja = false; // error
}
```

10.1.4 指向 struct instance 的 ptr



Access structs via pointers:

- `obj.member`
- `ptr->member`

```
struct Person {
     int age;
     string name;
     bool isNinja;
};

int main() {
     Person p = { 31, "Aliyah", true };
     Person * ptr = &p;

     p.age = 32; // 这三个一样
     (*ptr).age = 32; // 这三个一样
     ptr->age = 32; // 这三个一样
}
```

同样，指向 `const` 的 struct instance 的 ptr 也必须是 ptr-to-const.

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

struct Sandwich {
    string name;
    bool is_veg;
    double price;
};

int main() {
    const Sandwich a = {"a", true, 1.11};
    Sandwich const *ptr = &a;
    cout << ptr->name;
}
```

10.1.5 使用pass-by-pointer 的 init 函数

一个 ADT 的占内存大小是很大的。我们有时候不得不用一个 function 来 init 它，但是我们肯定不想在 function 的 stackframe 里 copy 它，然后再复制回去，因为它太大了，这样我们就把一个很大的 ADT 变成了两个，非常占内存。

之前说了，**pointer 类 object 的大小是恒定的，并且就等于操作系统的位数，所以其实非常小。**

因而我们可以定义一个 pass-by-pointer 的 init 函数来 initialize 一个 ADT struct.

比如一个 Triangle：

```
struct Triangle { double a, b, c;
};
void Triangle_init(Triangle *tri, double a_in,
                   double b_in, double c_in) {
  tri->a = a_in;
  tri->b = b_in;
  tri->c = c_in;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 11);
}
```

10.1.6 `->` 运算符

这里的 `->` 运算符是针对 class 类 object 的指针的运算符。它获取的对象是 `p` 指向的 object 的某个属性. 比如 `tri -> a` 就是 `tri` 这个 `Triangle *` 所指向的 `Triangle` object 的 `a` 这个属性.

当然，我们可以也使用 `(*tri).a`.

但是我们可以发现一个问题就是 `*tri.a` 是不会 compile 的，只有加上括号 `(*tri).a` 才 compile. 这是因为优先级的问题。

所以很不方便，所以我们还是使用 `->` 好（而且直观）。

10.1.7 使用 `assert()` 来检查是否 respect interface

但是我们发现这个 Triangle 的边长构不成一个 Triangle. 应该用 assert 来防止这些情况。 这就是 ADT 要 respect interface 的特点

```
void Triangle_init(Triangle *tri, double a_in,
                   double b_in, double c_in) {
  assert(0 < a_in && 0 < b_in && 0 < c_in);
  assert(a_in + b_in > c_in  && a_in + c_in > b_in &&
         b_in + c_in > a_in);
  tri->a = a_in;
  tri->b = b_in;
  tri->c = c_in;
}
```

写一些函数表示这个 ADT 的操作：

```
struct Triangle {
  double side1;
  double side2;
  double angle;
};
// REQUIRES: tri points to a Triangle object
// MODIFIES: *tri
// EFFECTS: Initializes the triangle with the given side lengths.
void Triangle_init(Triangle *tri, double a_in, double b_in, double c_in) {
  tri->side1 = a_in;
  tri->side2 = b_in;
  tri->angle = std::acos((std::pow(a_in, 2) + std::pow(b_in, 2) -
                          std::pow(c_in, 2)) / (2 * a_in * b_in));
}

// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the first side of the given Triangle.
double Triangle_side1(const Triangle *tri) {
  return tri->side1;
}
// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the second side of the given Triangle.
double Triangle_side2(const Triangle *tri) {
  return tri->side2;
}
```

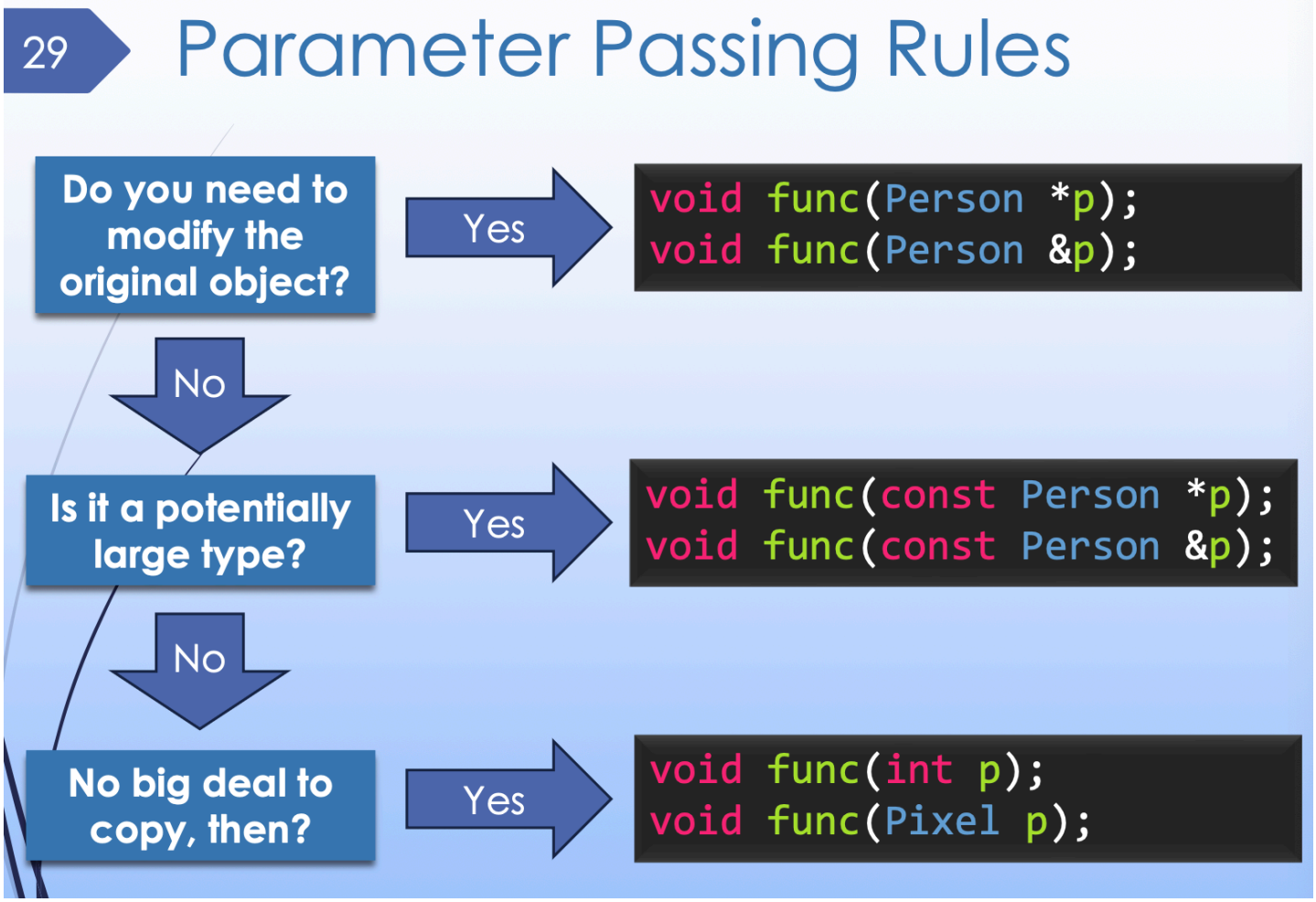
```
// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the third side of the given Triangle.
double Triangle_side3(const Triangle *tri) {
  return std::sqrt(std::pow(tri->side1, 2) + std::pow(tri->side2, 2) -
                   2 * tri->side1 * tri->side2 * std::acos(tri->angle));
}
// REQUIRES: tri points to a valid Triangle
// EFFECTS: Returns the perimeter of the given Triangle.
double Triangle_perimeter(const Triangle *tri) {
  return Triangle_side1(tri) + Triangle_side2(tri) + Triangle_side3(tri);
}
// REQUIRES: tri points to a valid Triangle; s > 0
// MODIFIES: *tri
// EFFECTS: Scales the sides of the Triangle by the factor s.
void Triangle_scale(Triangle *tri, double s) {
  tri->side1 *= s;
  tri->side2 *= s;
}
```

10.1.8 参数为 struct 的函数: parameter passing rules

当然，不止 init 函数，基本所有需要传递 ADT 参数的函数，我们都最好不要 pass by value。因为真的很大。

所以我们可以 使用 pass-by-reference 或者 pass-by-pointer.

如果我们不想用这个 ptr 来 modify the outside object，我们需要加上 `const` 关键词，使用 `pointer-to-const` 或者 `reference-to-const` 来操作。

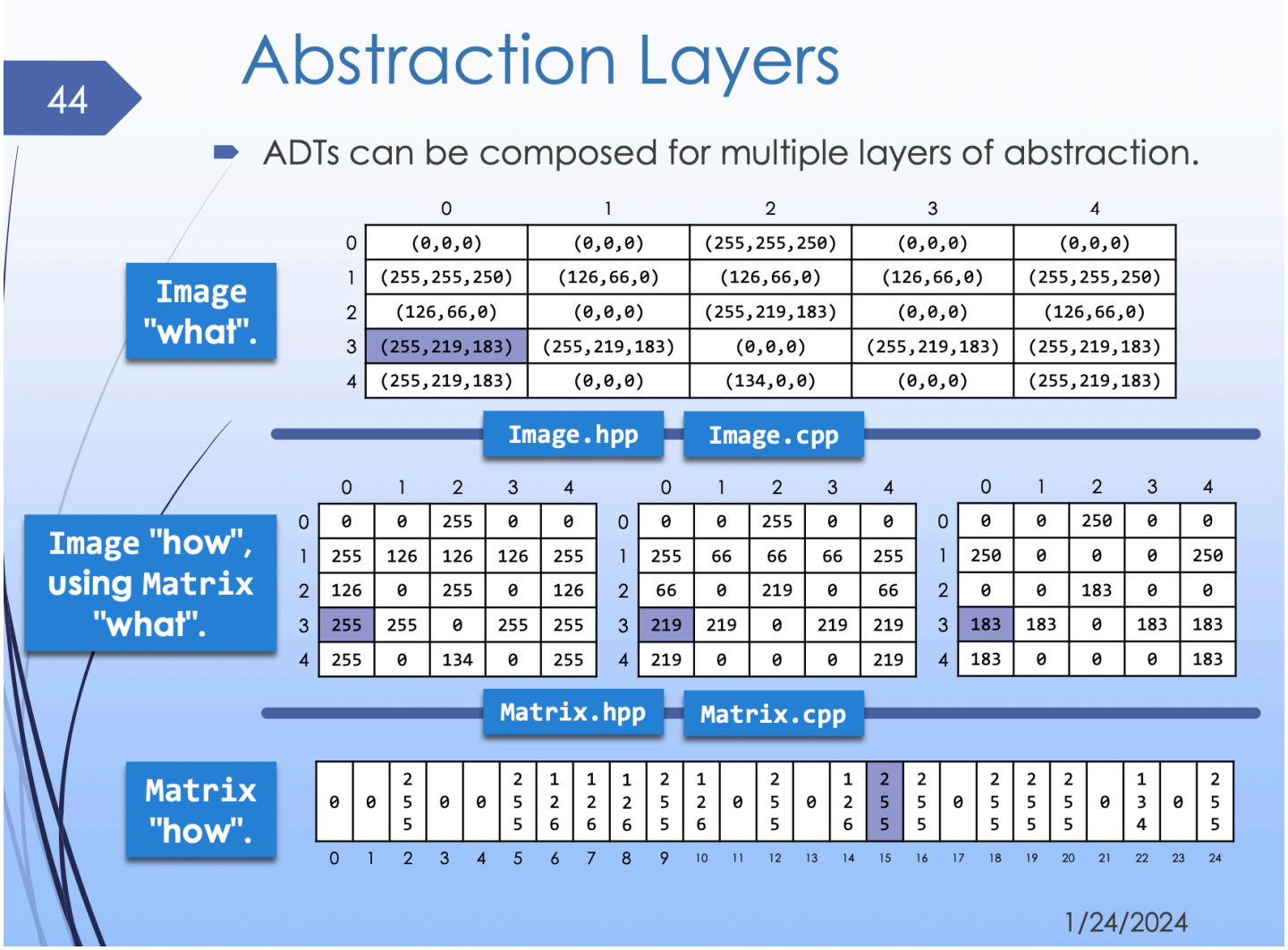


10.1.9 Composing ADTs: Abstraction Layers

One ADT might be a member of another.

```
struct Professor {
  int age;
  Triangle favTriangle;
};
void Professor_init(Professor *prof, int age, int side) {
  prof->age = age;
  Triangle_init(&prof->favTriangle, side, side, side);
}
void Professor_init(Professor *prof, int age, const Triangle &favTriangle) {
  prof->favTriangle = favTriangle;
  prof->age = age;
}
```

我们需要 Abstraction Layer 来制作 ADT.



10.2 Dynamic Allocation

10.2.1 Heap (堆)

我们现在的函数都是在Stack frame里面，这是静态的数据存储方式.

但是给这些 Stack 分配的内存空间的数量是很少的. Matrix 很大，会 crash 掉程序. 在处理这种大规模的数据时，我们应该动用 **Dynamic Allocation(动态内存分配)**.

静态的 memory 储存结构是 **Stack(栈)**，也就是我们已经认识的：

Stack内存：

- 栈内存是自动管理的，主要用于存储 local variables 和管理 function 调用（包括参数传递、返回地址和局部变量等）
- Stack 有固定的大小，当程序启动时由操作系统分配.
- Stack 的生命周期是自动的，当函数调用结束时，该函数的 Stack Frame 就会被销毁.

而 dynamic 的 memory 的储存结构是 **Heap(堆)**.

堆内存：

- Heap memory 是 dynamically allocated 的，需要程序员显式分配和释放.
- 堆的大小通常限制为操作系统允许的进程的最大内存大小，它**远大于 Stack 的大小.**
- 堆的生命周期是**手动管理**的，使用 `new`（在 C++ 中）分配的内存会一直存在直到它被 `delete` 显式释放.

10.3.2 `new` 和 `delete`: 分配和释放 dynamic memory

下面这个函数的 local varibales 静态储存在 stack frame 里面，这样很显然太大了会爆掉.

```
void func() {
  Matrix m;
  Matrix_init(&m, 300, 200);
  // Use the Matrix
  Matrix_fill(&m, 42);
} // mat goes out of scope
```

应该把 Matrix 保持在 dynamic memory 中：

我们需要用 `new` 在 heap 中新建一个 Matrix，注意，`new` **返回的是它的地址！因而我们需要用一个对应的指针变量来储存.**

在最后，我们要用 `delete` 来我们使用完的 Matrix 变量从动态内存中删除掉，这样就把这块内存空出来了.

```
// Fills a 3x5 Matrix with a value and checks that // Matrix_at returns that value for each
element.
TEST(test_fill_basic) {
  Matrix *mat = new Matrix; // 在Dynamic memory 中创造 Matrix object, the result is a ptr to
eh new object.
  const int width = 3;
  const int height = 5;
  const int value = 42; Matrix_init(mat, width, height); Matrix_fill(mat, value);
  for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
          ASSERT_EQUAL(*Matrix_at(mat, row, col), value);
      }
  }
  delete mat; //delete the Matrix object when we no longer need it
}
```