具有Eigen成员的结构

执行摘要

如果您定义的结构具有固定大小的可矢量化特征类型的成员，则必须确保在其上调用 operator new 分配正确对齐的缓冲区。如果您仅使用足够新的编译器（例如，GCC>=7、clang>=5、MSVC>=19.12）在**[c++17]**模式下进行编译，那么编译器会处理一切，您可以停止阅读。

否则，您必须重载它，`operator new`以便它生成正确对齐的指针（例如，Vector4d 和 AVX 的 32 字节对齐）。幸运的是，Eigen 为您提供了一个`EIGEN_MAKE_ALIGNED_OPERATOR_NEW`可以为您执行此操作的宏。

需要改什么样的代码？

需要更改的代码是这样的：

```
class Foo
{
  ...
  Eigen::Vector2d v;
  ...
};

...

Foo *foo = new Foo;
```

换句话说：您有一个类，该类的成员是固定大小的可矢量化特征对象，然后您动态地创建该类的对象。

这样的代码应该如何修改？

很简单，您只需要`EIGEN_MAKE_ALIGNED_OPERATOR_NEW`在类的公共部分放置一个宏，如下所示：

```
class Foo
{
  ...
  Eigen::Vector4d v;
  ...
public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW
};

...

Foo *foo = new Foo;
```

这个宏使得`new Foo`总是返回一个对齐的指针。

在**[c++17] 中**，这个宏是空的。

如果这种方法太麻烦，另请参阅其他解决方案。

为什么需要这个？

好的，假设您的代码如下所示：

```
class Foo
{
  ...
  Eigen::Vector4d v;
  ...
};

...

Foo *foo = new Foo;
```

Eigen::Vector4d 包含 4 个双精度值，即 256 位。这正是 AVX 寄存器的大小，这使得可以使用 AVX 对该向量进行各种操作。但是 AVX 指令（至少是 Eigen 使用的那些，它们是快速的）需要 256 位对齐。否则，您会遇到分段错误。

出于这个原因，Eigen 自己通过做两件事来要求 Eigen::Vector4d 进行 256 位对齐：

- Eigen 要求 Eigen::Vector4d 的数组（4 个双精度数）进行 256 位对齐。在**[c++11] 中，**这是通过 alignas关键字或 c++98/03 的编译器扩展完成的。
- Eigen 重载了`operator new`Eigen::Vector4d，因此它将始终返回 256 位对齐的指针。（在**[c++17] 中**删除）

因此，通常情况下，您不必担心任何事情，Eigen 会为您处理运算符 new 的对齐...

除了一种情况，当你有一个`class Foo`像上面一样，并且你像上面那样动态分配一个 new 时`Foo`，那么，由于`Foo`没有对齐`operator new`，返回的指针 foo 不一定是 256 位对齐的。

成员的对齐属性`v`然后是相对于类的开始`Foo`。如果`foo`指针没有对齐，那么`foo->v`也不会对齐！

解决方案是让`class Foo`一个对齐的`operator new`，正如我们在上一节中展示的那样。

这种解释也适用于需要 16 字节对齐的 SSE/NEON/MSA/Altivec/VSX 目标，以及需要 64 字节对齐的 64 字节倍数的固定大小对象（例如 Eigen::Matrix4d）的 AVX512。

然后我应该把 Eigen 类型的所有成员放在类的开头吗？

那不是必需的。由于 Eigen 负责声明适当的对齐，因此所有需要它的成员都会自动相对于类对齐。所以这样的代码工作正常：

```
class Foo
{
  double x;
  Eigen::Vector4d v;
public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW
};
```

也就是说，像往常一样，建议对成员进行排序，以便对齐不会浪费内存。在上面的示例中，对于 AVX，编译器必须在 `x` 和之间保留 24 个空字节 `v`。

动态大小的矩阵和向量呢?

动态大小的矩阵和向量，例如 Eigen::VectorXd，会动态分配它们自己的系数数组，因此它们会自动处理要求绝对对齐的问题。所以他们不会导致这个问题。此处讨论的问题仅适用于固定大小的可矢量化矩阵和向量。

那么这是 Eigen 中的错误吗?

不，这不是我们的错误。它更像是 c++ 语言规范的一个固有问题，它已在 c++17 中通过称为过度对齐数据的动态内存分配的特性得到解决。

如果我想有条件地执行此操作（取决于模板参数）怎么办?

对于这种情况，我们提供了宏 `EIGEN_MAKE_ALIGNED_OPERATOR_NEW_IF(NeedsToAlign)`. 它将生成对齐的运算符，如 `EIGEN_MAKE_ALIGNED_OPERATOR_NEW` if `NeedsToAlign` 为真。如果 `NeedsToAlign` 为 false，它将生成具有默认对齐方式的运算符。在**[c++17] 中**，这个宏是空的。

例子:

```
template<int n> class Foo
{
  typedef Eigen::Matrix<float, n, 1> Vector;
  enum { NeedsToAlign = (sizeof(Vector)%16)==0 };
  ...
  Vector v;
  ...
public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW_IF(NeedsToAlign)
};

...

Foo<4> *foo4 = new Foo<4>; // foo4 is guaranteed to be 128bit-aligned
Foo<3> *foo3 = new Foo<3>; // foo3 has only the system default alignment
guarantee
```

其他解决方案

如果将 `EIGEN_MAKE_ALIGNED_OPERATOR_NEW` 宏无处不在太麻烦，至少还有两种其他解决方案。

禁用对齐

第一个是禁用固定大小成员的对齐要求:

```
class Foo
{
  ...
  Eigen::Matrix<double,4,1,Eigen::DontAlign> v;
  ...
};
```

这`v`与对齐的 Eigen::Vector4d 完全兼容。这只会使加载/存储`v`更加昂贵（通常略有，但这取决于硬件）。

私有结构

第二个是将固定大小的对象存储到一个私有结构中，该结构将在主对象的构建时动态分配：

```
struct Foo_d
{
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW
  Vector4d v;
  ...
};


struct Foo {
  Foo() { init_d(); }
  ~Foo() { delete d; }
  void bar()
  {
    // use d->v instead of v
    ...
  }
private:
  void init_d() { d = new Foo_d; }
  Foo_d* d;
};
```

这里的明显优势是该类`Foo`在对齐问题方面保持不变。缺点是无论如何都需要额外的堆分配。