Special Member Functions:

C++会自动提供以下成员函数：

default constructor　　default destructor

copy constructor　　address operator

assignment operator

Default Constructors、Destructor

显性定义默认构造函数：无参、或参数有默认值.

只能拥有唯一一个默认构造函数.

```
Klunk() { klunk_ct = 0 }              // constructor #1
Klunk(int n = 0) { klunk_ct = n; }    // ambiguous constructor #2

Klunk kar(10);     // clearly matches Klunt(int n)
Klunk bus;         // could match either constructor
```

若有多个则会造成调用歧义

若未定义构造函数，编译器会自动加上一个空构造函数；若定义了，则编译器就不会自动加了.

```
Klunk::Klunk() { }  // implicit default constructor
```

- Default constructor: a constructor which can be called with no arguments
- If you define no constructors, the compiler automatically provide one

```
MyTime::MyTime(){}
```

- If you define constructors, the compiler will not generate a default one.

```
class MyTime
{
  public:
    MyTime(int n){ ... }
};
MyTime mt; //no appropriate constructor
```

- To avoid ambiguous

```
class MyTime
{
  public: //two default constructors
    MyTime(){}
    MyTime(int n = 0){ ... }
};
MyTime mt; //which constructor?
```

多个默认构造会造成歧义，引发错误

析构函数类似，不过析构函数只能有一个.

```
MyTime::~MyTime(){}
```

在构造函数中申请的内存通常会在析构函数中释放.

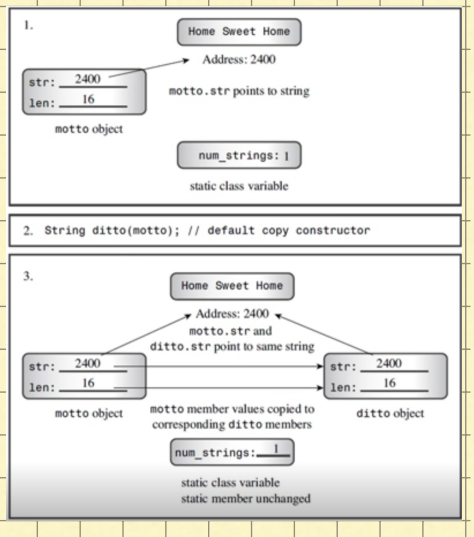
Default Copy Constructors

当新的对象创建并初始化为与现存对象同类型时调用.

在按值传递时调用.

- A copy constructor. Only one parameter, or the rest have default values

```
MyTime::MyTime(MyTime & t){ ... }
```



默认的拷贝函数能将非静态的成员都拷贝过去.(实际上使其指向同一个地方，即复制指针)

这会造成在释放内存时会对内存重复释放，造成报错.

- Default copy constructor:
  - If no user-defined copy constructors, the compiler will generate one.
  - Copy all non-static data members.

```
MyTime t1(1, 59);
MyTime t2 = t1; //copy constructor
```

Default Copy Assignment

- Assignment operators: =, +=, -=, ...
- Copy assignment operator

```
MyTime & MyTime::operator=(MyTime & ){...}
```

同样，默认的赋值拷贝函数会将所有非静态成员都拷贝过去.

- Default copy assignment operator
  - If no user-defined copy assignment constructors, the compiler will generate one.
  - Copy all non-static data members.

Dynamic Memory

① Hard Copy

内容相同，内存不共享

要在释放后将变量值置为0或NULL.

- Provide a user-defined copy constructor.

```
MyString::MyString(const MyString & ms)
{
    this->buf_len = 0;
    this->characters = NULL;
    create(ms.buf_len, ms.characters);
}
```

- create() release the current memory and allocate a new one.
- this->characters will not point to ms.characters .
- It's a hard copy!

- Provide a user-defined copy assignment

```
MyString & operator=(const MyString &ms)
{
    create(ms.buf_len, ms.characters);
    return *this;
}
```

② Soft Copy

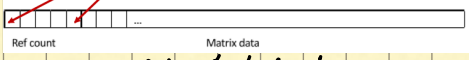
hard copy 会频繁地申请和释放内存，当所需内存过大时会耗费更多时间.

CvMat struct

modules/core/include/opencv2/core/types_c.h

```
468  typedef struct CvMat
469  {
470      int type;
471      int step;
472
473      /* for internal use only */
474      int* refcount;
475      int hdr_refcount;
476
477      union
478      {
479          uchar* ptr;
480          short* s;
481          int* i;
482          float* fl;
483          double* db;
484      } data;
```

...	...
int	flags
int	dims
int	rows
int	cols
uchar*	data
int*	refcount
...	...



Ref count　　Matrix data

cv::Mat class

```
801  class CV_EXPORTS Mat
802  {
803  public:
     .
2103     int flags;
2104     //! the matrix dimensionality, >= 2
2105     int dims;
2106     //! the number of rows and columns or (-1, -1) when the matrix has more than 2 dimensions
2107     int rows, cols;
2108     //! pointer to the data
2109     uchar* data;
2110
2126     //! interaction with UMat
2127     UMatData* u;
2128
2129     MatSize size;
2130     MatStep step;
2131
2132 protected:
2133     template<typename _Tp, typename Functor> void forEach_impl(const Functor& operation);
2134 };
```

用uchar是为了在图像中方便些

Allocated at the same time

refcount指针会在每次Soft copy后+1

modules/core/src/matrix.cpp

```
488 Mat& Mat::operator=(const Mat& m)
489 {
490     if( this != &m )
491     {
492         if( m.u )
493             CV_XADD(&m.u->refcount, 1);
494         release();
495         flags = m.flags;
496         if( dims <= 2 && m.dims <= 2 )
497         {
498             dims = m.dims;
499             rows = m.rows;
500             cols = m.cols;
501             step[0] = m.step[0];
502             step[1] = m.step[1];
503         }
504         else
505             copySize(m);
506         data = m.data;
507         datastart = m.datastart;
508         dataend = m.dataend;
509         datalimit = m.datalimit;
510         allocator = m.allocator;
511         u = m.u;
512     }
513     return *this;
514 }
```

为了防止多线程时refcount计数内存泄漏.

Soft copy

Solution in OpenCV

- The allocated memory can be used by multiple object
- Mat::u->refcount is used to count the times the memory is referenced
- CV_XADD: macro for atomic add

modules/core/src/matrix.cpp

```
405 Mat::Mat(const Mat& m)
406     : flags(m.flags), dims(m.dims), rows(m.rows), cols(m.cols), data(m.data),
407       datastart(m.datastart), dataend(m.dataend), datalimit(m.datalimit), allocator(m.allocator),
408       u(m.u), size(&rows), step(0)
409 {
410     if( u )
411         CV_XADD(&u->refcount, 1);
412     if( m.dims <= 2 )
413     {
414         step[0] = m.step[0]; step[1] = m.step[1];
415     }
416     else
417     {
418         dims = 0;
419         copySize(m);
420     }
421 }
```

modules/core/src/matrix.cpp

Solution in OpenCV

```
551 void Mat::release()
552 {
553     if( u && CV_XADD(&u->refcount, -1) == 1 )
554         deallocate();
555     u = NULL;
556     datastart = dataend = datalimit = data = 0;
557     for(int i = 0; i < dims; i++)
558         size.p[i] = 0;
559 #ifdef _DEBUG
560     flags = MAGIC_VAL;
561     dims = rows = cols = 0;
562     if(step.p != step.buf)
563     {
564         fastFree(step.p);
565         step.p = step.buf;
566         size.p = &rows;
567     }
568 #endif
569 }
```

Smart Pointers (c++11 以后)

智能指针在不使用时能保证被删除.

① std::shared_ptr

多个 shared_ptr 能指向同一个对象

当没有 shared_ptr 指向时，该对象会被销毁.

```
std::shared_ptr<MyTime> mt1(new MyTime(10));
std::shared_ptr<MyTime> mt2 = mt1;

auto mt1 = std::make_shared<MyTime>(1, 70);
```

共用一块内存

都是构造函数

```
std::shared_ptr<MyTime> mt0 = new MyTime(0,70); //error
MyTime * mt0 = std::make_shared<MyTime>(1, 70); //error
// {
```

这样构造都是不合法的.

② std::unique_ptr

unique_ptr 指向的对象禁止其他指针指向，但被 unique_ptr 指向的对象能移至其他指针.

```
std::unique_ptr<MyTime> mt1(new MyTime(10));
std::unique_ptr<MyTime> mt2 = std::make_unique<MyTime>(80); //c++17

std::unique_ptr<MyTime> mt3 = std::move(mt1);
```

都是构造

将 mt1 移交给 mt3，然后 mt1 置为 NULL

两种智能指针的定义：实际上都是类.

```
template< class T > class shared_ptr;

template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```

- mt1 and mt2 are two objects of type shared_ptr<>.
  - You can do a lot in the constructors and the destructor.

```
std::shared_ptr<MyTime> mt1(new MyTime(10));
std::shared_ptr<MyTime> mt2 = mt1;
```