南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

CS205

Prof. Shiqi Yu (于仕琪)

yusq@sustech.edu.cn

http://faculty.sustech.edu.cn/yusq/

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Some Default Operations

Default Constructors

- Default constructor: a constructor which can be called without 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?
```

Implicitly-defined Destructor

- If no destructor is defined, the compiler will generate an empty one.

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

- Memory allocated in constructors is normally released in a destructor.

Default Copy Constructors

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

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

```
MyTime t1(1, 59);
MyTime t2(t1); //copy constructor
MyTime t3 = t1; //copy constructor
```

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

Default Copy Assignment

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

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

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

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

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

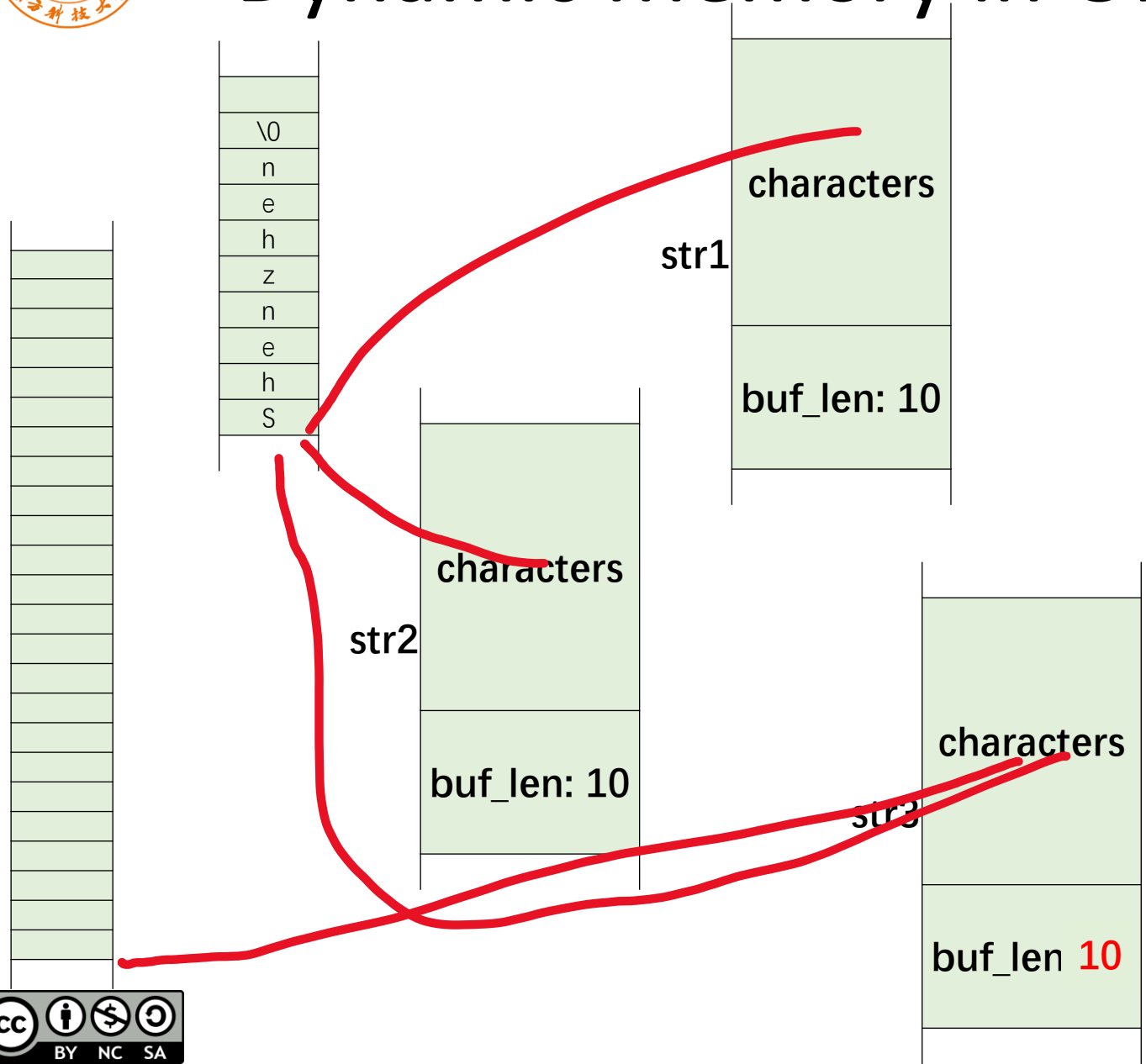
An Example with Dynamic Memory

A Simple String Class

```
class MyString
{
   int buf_len;
   char * characters;
 public:
   MyString(int buf_len = 64, const char * data = NULL)
   {
      this->buf_len = 0;
      this->characters = NULL;
      create(buf_len, data);
   }
   ~MyString()
   {
      delete []this->characters;
   }
   ...
};
```


example1

Dynamic Memory in Objects




```
MyString str1(10, "Shenzhen");

MyString str2 = str1;

MyString str3;
str3 = str1;
```

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Solution 1: Hard Copy

Copy Constructor

- 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!

Copy Assignment

- Provide a user-defined copy assignment

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


example1

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Solution 2: Soft Copy

Problem of Hard Copy

- Frequently allocate and free memory.
- Time consuming when the memory is big.

But...

- If several objects share the same memory, who should release it?

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;
```

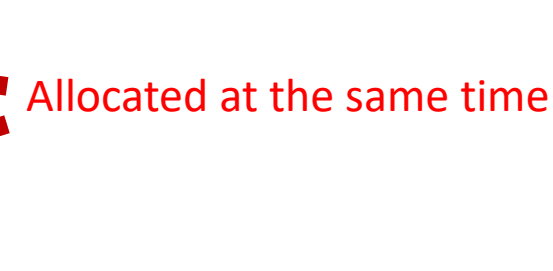
...	...
int	flags
int	dims
int	rows
int	cols
uchar*	data
int*	refcount
...	...

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  };
```



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 }
```

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

Solution in OpenCV

- Copy constructor of cv::Mat

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  }
```

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Smart Pointers

std::shared_ptr

- Smart pointers are used to make sure that an object can be deleted when it is no longer used. 😍
- Several shared pointers can share/point to the same object.
- The object is destroyed when no shared_ptr points to it.

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

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

std::unique_ptr

- Different from std::shared_ptr, a std::unique_ptr will point to an object, and not allow others to point to.
- But an object pointed by a std::unique_ptr can be moved to another pointer.

```
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);
```

How to Understand Smart Pointers

- Let's look at their definitions.

```
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;
```