

Lecture Material

Class *string*

- simple
- with reference counting

Exceptions

The *volatile* specifier

Semaphores

The *string* Class

- # Text processing is a common domain in computer applications
- # In C and C++ there is no real string type, `char*` is hard to use and error prone

```
class mystring
{
    char *data;
public:
    mystring ();
    ~mystring ();
    mystring (const char *s);
    unsigned int length () const;
    mystring (const mystring & src);
    mystring & operator= (const mystring & src);
    mystring & operator+= (const mystring & src);
    char operator[] (unsigned int i) const;
    char & operator[] (unsigned int i);
    friend ostream & operator<< (ostream & o, const mystring & s);
};

inline mystring operator+ (const mystring & s1, const mystring & s2);
```

The diagram illustrates the usage of various operators in the `mystring` class. Arrows point from specific operators to two categories:

- Reference:** This category includes the copy constructor (`mystring (const mystring & src)`), assignment operator (`operator= (const mystring & src)`), addition assignment operator (`operator+= (const mystring & src)`), and the friend insertion operator (`operator<<`).
- No reference:** This category includes the copy assignment operator (`&operator= (const mystring & src)`), subscript operator (`operator[]`), and the friend addition operator (`operator+`).

string - Constructor and Member Initializer List

Without member initializer list

```
mystring ()  
{  
    data = NULL;  
}
```

With member initializer list

```
mystring () : data (NULL) {};
```

- # There is no difference in this case
- # The difference appears, when the members have constructors
 - Without initializer list: default constructor called, then the assignment operator
 - With initializer list: constructor with a parameter called
- # Using member initializer list makes program shorter and faster

The *string* Class - operator +

- # The *friend* specifier is not needed, function uses exclusively the public members of the *string* class
- # When returning result from the function, the copy constructor will be used

```
inline mystring operator+ (const mystring & s1, const mystring & s2)
{
    mystring s (s1);
    return s += s2;
}
```

The *string* Class - operator char*

- # In class you can define the conversion operators to different types
- # Frequently useful in case you are using function accepting built-in types (e.g. *char**)

```
mystring::operator char* ()  
{  
    if (!data)  
    {  
        data=new char[1];  
        data[0]='\0';  
    }  
    return data;  
};
```

- # In the *string* class possibility of clash with an indexing operator

```
mystring s;  
s[0]='a'; //s.operator[] (0) or (s.operator char*())[0] ?
```

Exceptions

- # Correct program checks for error codes

```
int fun()
{
//...
    fd=open("file",O_RDWR);
    if(fd== -1)
        return -1;
    if(read(fd,buffer,15) !=15)
        return -1;
    if(write(fd,buffer,4) !=4)
        return -1;
//...
    return 0;
}
```

- # Checking for error conditions every time is boring and it is easy to forget about it
- # In some cases it is difficult to return an error code, e.g. *atoi()* can return any integer and there is no value to signal an error
- # Signalling errors in constructors is a hard problem

Exceptions

C++ provides a new mechanism to signal errors:
exceptions

```
class myexception{};  
void f1()  
{  
    if(...)  
        throw myexception();  
};  
void f2()  
{  
    f1();  
};  
int main()  
{  
try {  
    f2();  
} catch(myexception&)  
{  
    cout << "myexception caught" << endl;  
}  
    return 0;  
};
```

The exception class, allows to
distinguish errors

Signalling an error

Detection of error

Exceptions

- # The errors can be distinguished based on the type of exception thrown

```
class bad_index{};  
class no_memory {};  
void f1()  
{  
    if(...)  
        throw bad_index();  
    if(...)  
        throw no_memory();  
};  
//...  
try {  
    f2();  
} catch(bad_index&)  
{  
//...  
} catch (no_memory&)  
{  
//...  
}
```

- # Operator new throws *bad_alloc* when there is not enough memory

Exceptions

- When the exception is thrown, all destructors of local objects on the way from callee to caller are invoked

```
#include <iostream>
using namespace std;
class myexception{};
class tester
{
    string name;
public:
    tester(const string& n): name(n)
    {
        cout<<name<<"()"<<endl;
    }
    ~tester()
    {
        cout<<"~"<<name<<"()"<<endl;
    }
};
```

```
void f1()
{
    tester f1("f1");
    throw myexception();
    cout << "Exiting f1" << endl;
};

void f2()
{
    tester f2("f2");
    f1();
    cout << "Exiting f2" << endl;
};

int main()
{
    tester main("main");
    f2();
    return 0;
};
```

Exceptions

- # When the appropriate *catch* clause is not present, the execution of program is aborted
- # We can modify the *string* class to use exceptions

```
class mystring
{
    char *data;
public:
    class index_out_of_range{};
//...
char operator[] (unsigned int i) const
{
    if (!data)
        throw index_out_of_range();
    if (i >= strlen (data))
        throw index_out_of_range();
    return data[i];
}
};
```

Problems with Exceptions

- Careless usage of exceptions can lead to resource leaks or data inconsistency

```
void fun()
{
    char* a = new char[1024];
    char* b = new char[1024];
    //...
    delete [] a;
    delete [] b;
}
```

If *new* throws an exception, the memory allocated for *a* will not be freed

```
void fun()
{
    char* a = NULL, *b=NULL;
    try{
        a = new char[1024];
        b = new char[1024];
        //...
    } catch (...){
        delete [] a;
        delete [] b;
        throw;
    }
    delete [] a;
    delete [] b;
}
```

A way to fix the error, correct but not the best

Problems with Exceptions

Careless usage of exceptions can lead to resource leaks or data inconsistency

```
mystring & operator=
    (const mystring & src)
{
    if (this != &src)
    {
        delete[] data;
        if (src.data)
        {
            data = new char
                [strlen(src.data) + 1];
            strcpy (data, src.data);
        }
        else
            data = 0;
    }
    return *this;
}
```

If *new* throws an exception, the contents of the string will be lost, and *data* will have incorrect value, what will cause problems in destructor

```
mystring & operator=
    (const mystring & src)
{
    if (this != &src)
    {
        char* newdata;
        if (src.data)
        {
            newdata = new char
                [strlen (src.data) + 1];
            strcpy (newdata, src.data);
        }
        else
            newdata = 0;
        delete [] data;
        data=newdata;
    }
    return *this;
}
```

Fixed version does not destroy the string in case of lack of memory

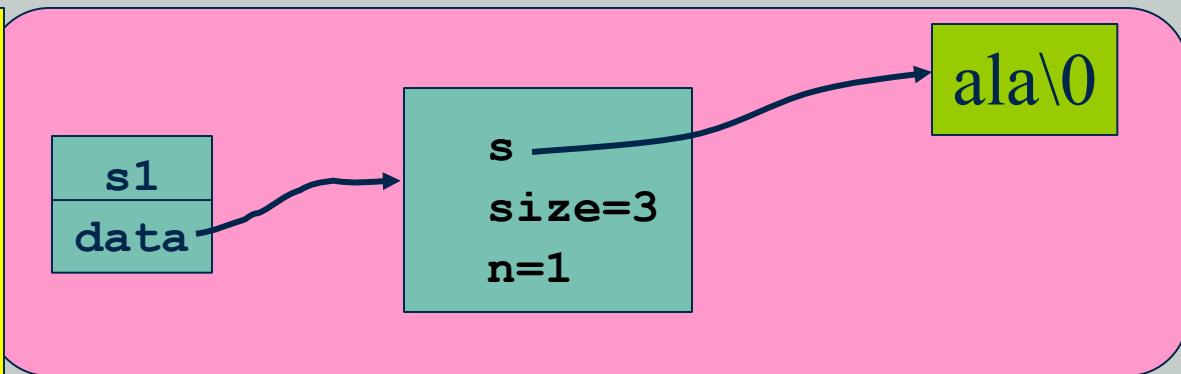
The *string* Class with Reference Counting

- # Copying of a string is a costly operation
- # Strings are frequently passed by value
- # We are looking for an efficient implementation, which will decrease the number of allocations and copying
- # The solution is the reference counting

The *string* Class with Reference Counting

The class is only a handle - it contains the pointer to the real string implementation

```
class string{
    struct rctext;
    rctext* data;
public:
    string(char* s);
    string& operator=
        (const string&);
~string();
...
};
struct string::rctext
{
    char* s;
    unsigned int size;
    unsigned int n;
...
};
string s1("ala");
```



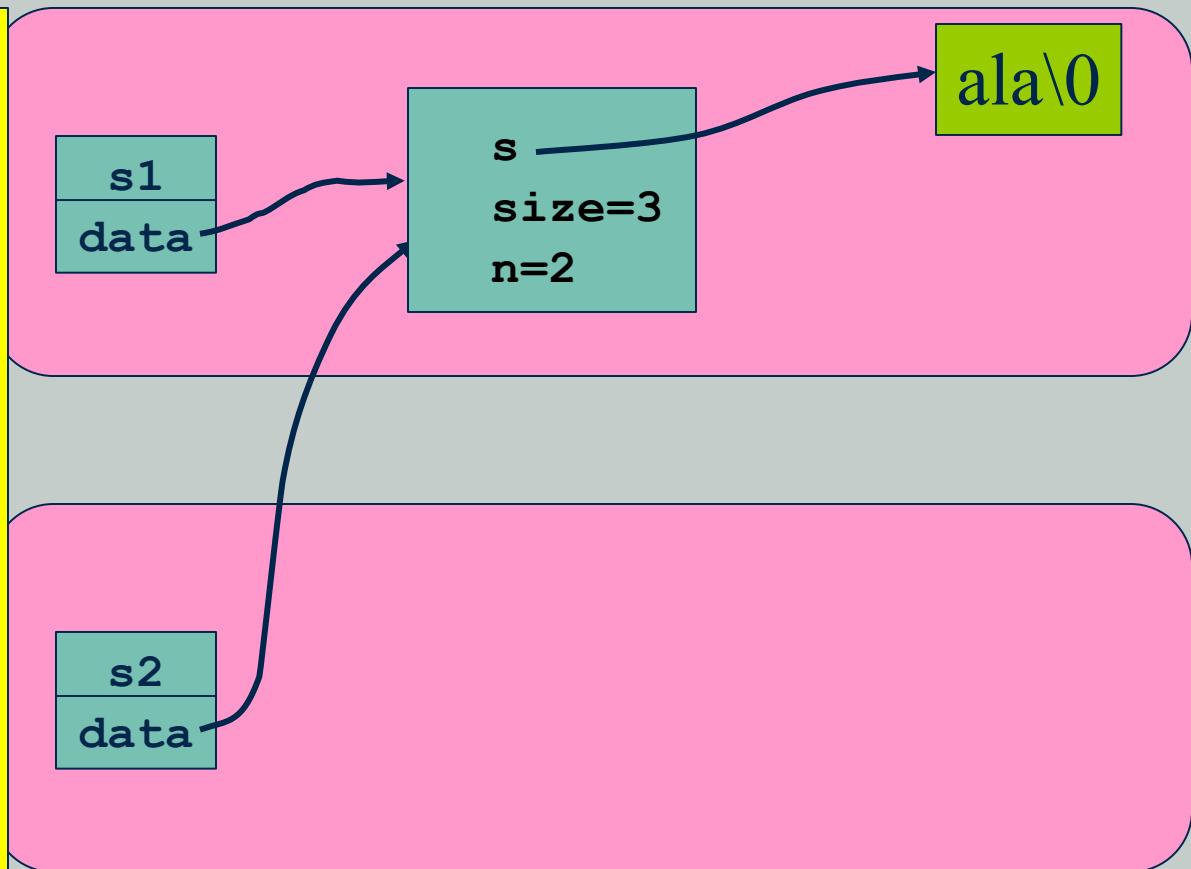
The *string* Class with Reference Counting

- # The copying of string copies the pointer to the internal representation and increments the reference counter

```
class string{
    struct rctext;
    rctext* data;
public:
    string(char* s);
    string& operator=
        (const string&);
~string();
...
};

struct string::rctext
{
    char* s;
    unsigned int size;
    unsigned int n;
...
};

string s1("ala");
string s2=s1;
```

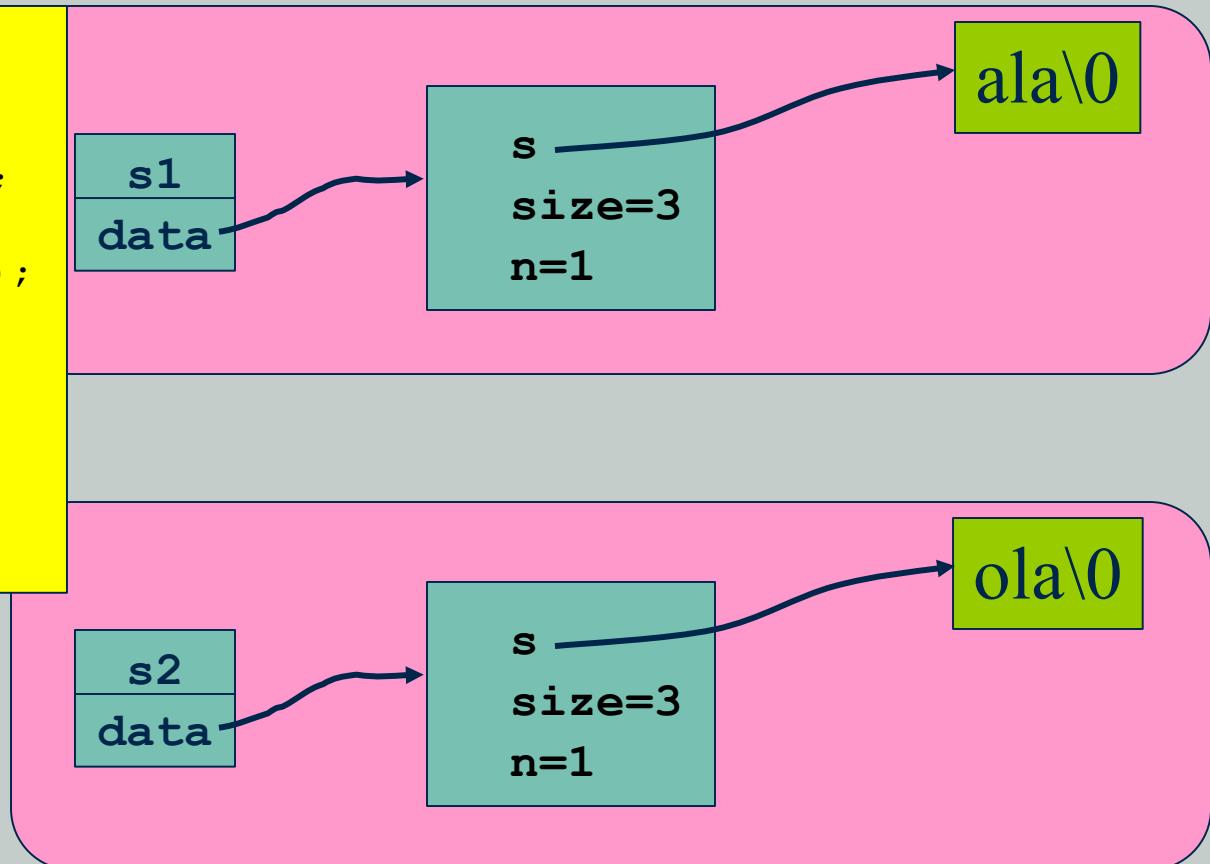


The *string* Class with Reference Counting

Modification of one of the copies causes separation of *s1* and *s2* (*copy-on-write*)

```
class string{
public:
    char read
        (unsigned int i) const;
    void write
        (unsigned int i, char c);
    ...
};

string s1("ala");
string s2=s1;
s2.write(0,'o');
```



The *string* Class with Reference Counting

- # We want the indexing operator to work correctly for reading and writing

```
class string{  
    ...  
};  
  
string s1("ala");  
string s2=s1;  
char a=s2[0];  
s2[0]='o';
```

- # We do not want reading of character to cause making a copy of the string
- # We can achieve this by returning in the indexing operator not the character reference, but the special class, *Cref*

The *string* Class with Reference Counting

- # The *Cref* behaves in the same way as *char*, but can distinguish between reading and writing

```
class string{
public:
    string(char* s);
    string& operator=(const string&);
    ~string();
    void check(unsigned int i) const;
    Cref operator[](unsigned int i);
    char operator[](unsigned int i) const;
    char string::read(unsigned int i) const;
    void string::write(unsigned int i, char c){
        data = data->detach();
        data->s[i] = c;
    }
    ...
};

string::Cref
    string::operator[](unsigned int i)
{
    check(i);
    return Cref(*this,i);
};
```

```
class string::Cref
{
    friend class string;
    string& s;
    unsigned int i;
    Cref (string& ss, unsigned int ii)
        :s(ss), i(ii) {};
public:
    operator char() const
    {
        return s.read(i);
    }
    string::Cref& operator = (char c)
    {
        s.write(i,c);
        return *this;
    };
    string::Cref& operator = (const Cref& ref)
    {
        return operator= ((char)ref);
    };
};
```

rcstring and multithreading

■ Reference-counting implementation can cause problems in multithreaded programs

```
void thread1 ()
{
    for (unsigned int i = 0;
          i < 100000; i++)
    {
        rcstring s1 (s);
    }
}

void thread2 ()
{
    for (unsigned int i = 0;
          i < 100000; i++)
    {
        rcstring s1 (s);
    }
}
```

```
int main ()
{
    thread t1 = thread (thread1);
    thread t2 = thread (thread2);
    t1.join();
    t2.join();
    cout << "RefCount="
        << s.getRefCount () << endl;
}
```

```
inline rcstring::rcstring(const rcstring& x)
{
    x.data->n++;
    data=x.data;
}
inline rcstring::~rcstring() {
    if(--data->n==0)
        delete data;
}
```

Two threads can simultaneously change *n*

rcstring and multithreading

- # Operation "n++" is not atomic
- # It is composed of three phases:
 - Reading the value from memory
 - Incrementation
 - Writing the updated value to memory
- # These operation can be interleaved in a few threads
- # In effect, the variable has the incorrect final value (incrementation by one, instead of two)

```
thread1: d0=n;
thread2: d0=n;
thread1: d0=d0+1;
thread2: d0=d0+1;
thread1: n=d0;
thread2: n=d0;
```

- # Need to use synchronization objects (semaphores) or the special assembler instructions

The *volatile* Specifier

- # The *volatile* specifier denotes, that the variable value can change outside the program and therefore forbids optimization regarding the variable (e.g. storing it in the processor registers)
- # Used, when the variable references are the references to the input/output device registers
- # Unfortunately, it will not help us in case of reference-counting string

The *volatile* Specifier

- Adding the *volatile* specifier causes, that the program will behave according to our expectations (however, this is not guaranteed by the C++ standard)
- Do not use *volatile* for synchronization between threads.

```
int val = 0;

void fun1 ()
{
    val = 0;
    while (1)
        if (val != 0)
            break;
    printf ("out of the loop\n");
}

int main ()
{
    thread w = thread(fun1);
    sleep(1);
    val = 1;
    w.join ();
}
```

```
volatile int val = 0;

void fun1 ()
{
    val = 0;
    while (1)
        if (val != 0)
            break;
    printf ("out of the loop\n");
}

int main ()
{
    thread w = thread(fun1);
    sleep(1);
    val = 1;
    w.join ();
}
```

Semaphores

```
struct rcstring::rctext
{
    char *s;
    unsigned int size;
    unsigned int n;
    std::mutex mutex;

    unsigned int AtomicRead()
    {
        unsigned int retval;
        mutex.lock();
        retval = n;
        mutex.unlock();
        return retval;
    };

    unsigned int AtomicIncrement()
    {
        mutex.lock();
        unsigned int retval=++n;
        mutex.unlock();
        return retval;
    };

    unsigned int AtomicDecrement()
    {
        mutex.lock();
        unsigned int retval=--n;
        mutex.unlock();
        return retval;
    };
//...
};
```

