Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger

LS 11 Algorithm Engineering

Lecture 11 • Winter 2011/12 • Jan 10

technische universität dortmund

department of computer science

Automatic Variables

- All variables so far have been defined inside a block
  → automatic variables
- Memory is automatically
  - allocated at the beginning of the block
  - freed at the end of the block
- This is very comfortable for us: We do not have to deal with any issues on handling memory
- Today, we discuss:
  1. **Static and global variables**
     → live throughout the whole program
  2. **Static data members**
     → shared by all objects of a class
  3. **Dynamic memory allocation**
     → We decide on the lifetime of an object!

Static Variables

- You can declare variables in functions as `static`
- A static variable lives throughout the whole program
  → It retains its value between calls to that function

```
int newnumber() {
    static int counter;
    return ++counter;
}

int main() {
    cout << newnumber() << endl;
    cout << newnumber() << endl;
    return 0;
}
```

Output:

```
1
2
```

- Static variables of built-in types are initialized to 0; for custom types, the default constructor is called

Global Variables

- You can define variables outside of any scope → global variables
- For global variables, the same rules as for static variables apply (e.g. initialization)

```
int counter;  // global variable

int newnumber() {
    return ++counter;
}

int main() {
    cout << newnumber() << endl;
    cout << newnumber() << endl;
    return 0;
}
```

- Good programming practice:
  - try to limit scope of variables
  - → use as few global variables as possible!

Static Data Members

- We can also define data members to be `static`
- Static data members are shared by all objects of that class
- Same rules for initialization as for static variables
- Similarly, we can define member functions to be static
  - such functions can only access static data members of the class

Static Data Members

```
class C {
     static int counter;
public:
     C() { ++counter; }
     static int getCounter() { return counter; }
};

int C::counter;

int main() {
     cout << C::getCounter() << ' ';
     C a;
     cout << C::getCounter() << ' ';
     C b;
     cout << C::getCounter() << endl;
     return 0;
}
```

static data member

static member function

static data members must be defined outside the class!
could also be initialized: `int C::counter = 10;`

calling a static member function; equivalent:
`a.getCounter(),b.getCounter()`

Output:

```
0 1 2
```

Allocating Memory: The new Operator

- So far, we are not able to create new objects dynamically (i.e. decide at runtime to create objects)
- The **new** operator allows us to allocate a new piece of memory and to invoke a constructor:

```
int *intPtr;  // stores address of an int object
intPtr = new int(17); // reserves memory for an int
                      // initializes int with 17
```

- The general form of a **new** expression is:

```
new type-name
new type-name ( constructor-args )
```

- **new** returns a pointer to the newly created object; you should always store this pointer, so you can free the memory later

Freeing Memory: The delete Operator

- When you no longer need an object created with new, you must free the occupied memory using **delete**:

```
int *intPtr = new int(17);
// do something with *intPtr
delete intPtr;  // free memory
```

- If you don't free memory allocated by **new** explicitly, this memory will never be freed automatically!
  → memory leak!
- After calling **delete**, all pointers to this object become invalid
- ***Accessing objects through invalid pointers (dangling pointers) is a serious programming error!***

Safety Caveats

1. Initialize all pointers to 0 or such that they point to an object
2. Set pointers to 0 immediately after you free the memory they point to:

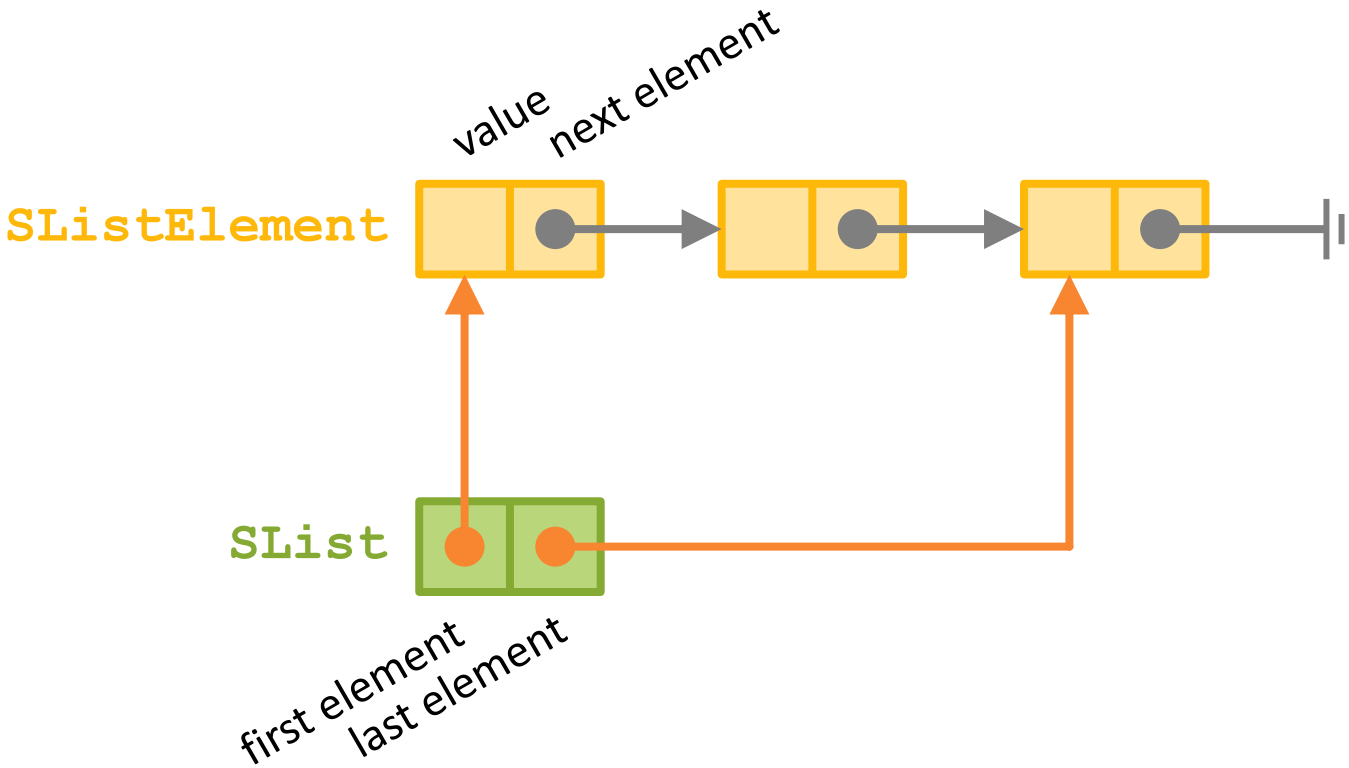
```
delete intPtr;
intPtr = 0;
```

3. When you are in doubt if a pointer is valid, test if it is not 0 before dereferencing it

- **Remark:** It is also allowed to call `delete` for a 0-pointer (in this case nothing is done)

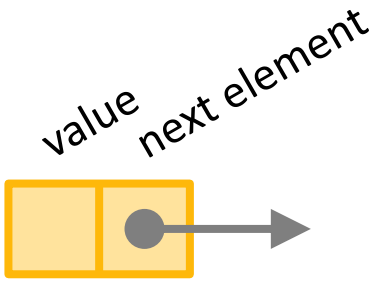
Example: Singly-linked Lists

- We want to implement our own singly-linked list, which allows us to dynamically add and remove elements



Modeling the Elements: SListElement

```
// data type for values stored in the list
typedef int ValueType;

// structure for list elements
struct SListElement {
     SListElement *pNext;
     ValueType value;

     SListElement(ValueType x) : pNext(0), value(x) { }
     SListElement(ValueType x, SListElement *p)
        : pNext(p), value(x) { }
};
```



- The type definition **`ValueType`** allows us to easily change the value type if required

Iterators

- Similar as standard container classes, we want to provide some nifty iterators for our list
- We realize iterators by a new class **`SListIterator`**:
  - encapsulates a pointer to **`SListElement`**
  - provides typical functionality, like comparison, assignment, dereferencing, and advancing an iterator

Iterators: The Interface

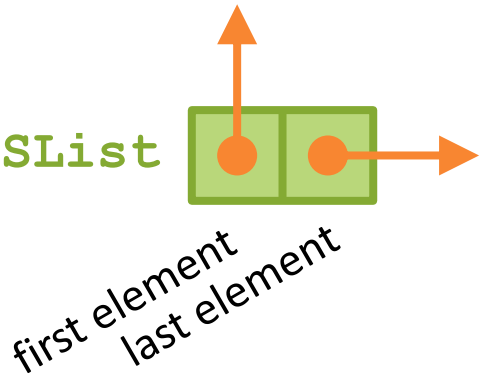
```
class SListIterator {
    SListElement *pElement;

public:
    SListIterator() : pElement(0) { }
    SListIterator(SListElement *p) : pElement(p) { }
    SListIterator(const SListIterator &it) : pElement(it.pElement) { }

    bool operator==(const SListIterator &it) const;
    bool operator!=(const SListIterator &it) const;

    SListIterator &operator=(const SListIterator &it);

    ValueType &operator*() const;

    SListIterator &operator++();    // operator ++ (prefix)
    SListIterator operator++(int); // operator ++ (postfix)
};
```

Iterators: The Implementation

```
bool SListIterator::operator==(const SListIterator &it) const {
     return pElement == it.pElement; }

SListIterator &SListIterator::operator=(const SListIterator &it) {
     pElement = it.pElement;
     return *this; }

ValueType &SListIterator::operator*() const {  // dereference
     return pElement->value; }

SListIterator &SListIterator::operator++() {   // prefix notation
     pElement = pElement->pNext;
     return *this; }

SListIterator SListIterator::operator++(int) { // postfix notation
     SListIterator it = *this;
     pElement = pElement->pNext;
     return it; }
```

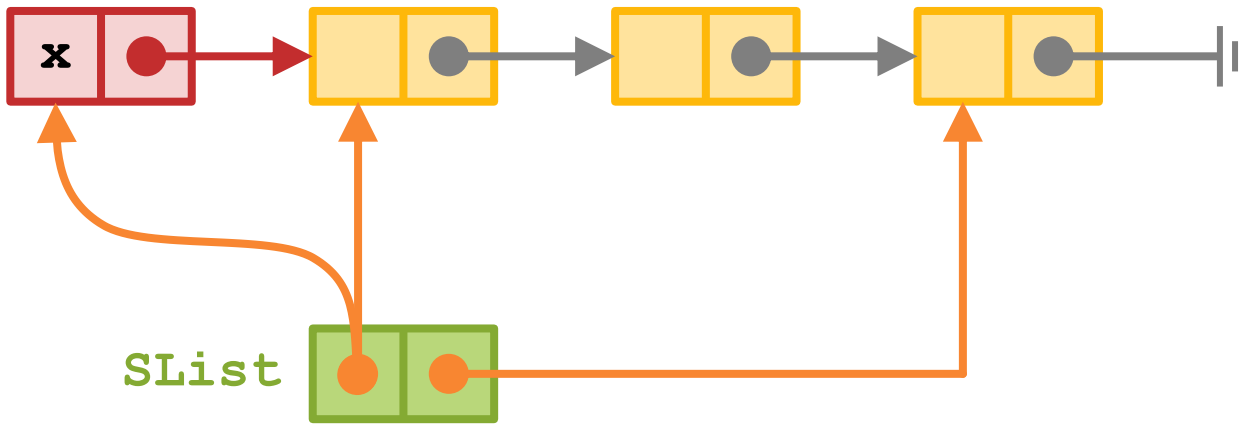
Singly-linked Lists: The Interface

```
class SList {
    SListElement *pHead, *pTail;

public:
    typedef SListIterator iterator;

    SList() : pHead(0), pTail(0) { }
    ~SList() { clear(); }

    // iterators
    iterator begin() const { return iterator(pHead); }
    iterator end() const { return iterator(); }

    iterator push(ValueType x);   // add element at the front
    iterator append(ValueType x); // add element at the back

    ValueType pop(); // remove first element and return its value

    void clear();    // remove all elements
};
```



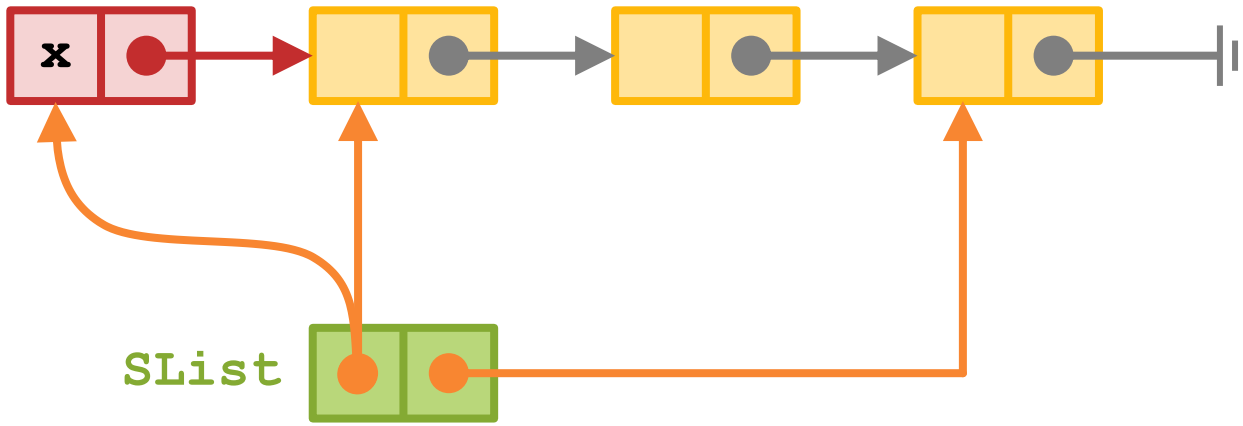
Push Method

```
SList::iterator SList::push(ValueType x)
{
     pHead = new SListElement(x, pHead);
     if(pTail == 0)
        pTail = pHead;

     return iterator(pHead);
}
```



Pop Method

```
ValueType SList::pop()
{
     SListElement *p = pHead;
     ValueType x = p->value;

     pHead = p->pNext;
     if(pHead == 0)
        pTail = 0;

     delete p; return x;
}
```



Schedule for last two lectures

- Lecture 12:
  How to debug programs with Visual Studio
- Lecture 13:
  Function objects and automatic pointers