

Динамична памет. Конструктори и деструктори

Любомир Чорбаджиев
Технологическо училище “Електронни системи”
Технически университет, София
lchorbadjiev@elsys-bg.org

Revision : 1.3

16 ноември 2004 г.

Пример: стек

Основните операции, които се извършват със стека са:

- `push()` — поставя елемент на върха на стека;
- `pop()` — изтрива последният елемент, поставен на върха на стека и го връща като резултат от операцията.

Има различни начини да се реализира стек. Нека разгледаме някои от тях.

Пример: стек

```
1 #include <iostream>
2 using namespace std;
3
4 class Stack {
5     const static int size_=2;
6     int data_[size_];
7     int top_;
8 public:
9     Stack(void)
10        : top_(-1)
11    {}
12
13    void push(int v) {
14        if(top_ >= (size_ - 1)) {
15            cout << "ERROR: stack is full..." << endl;
16            return;
17        }
18        data_[++top_] = v;
19    }
```

```
20  int pop(void) {
21      if(top_ < 0) {
22          cout << "ERROR: stack is empty..." << endl;
23          return 0;
24      }
25      return data_[top_--];
26  }
27 };
28
29 int main(void) {
30     Stack st;
31     st.push(1);
32     st.push(2);
33     st.push(3);
34
35     cout << st.pop() << endl;
36     cout << st.pop() << endl;
37     cout << st.pop() << endl;
38     return 0;
39 }
```

Пример: стек

Резултатът от изпълнението на програмата е следния:

```
lubo@dobby:~/school/cpp/notes> g++ code/lecture05-stack01.cpp
lubo@dobby:~/school/cpp/notes> a.out
ERROR: stack is full...
2
1
ERROR: stack is empty...
0
```

Пример: стек

- Основният недостатък на представената реализация е, че размерът на стека (броят на елементите, които можем да поставим в стека), се определя по време на компилация на програмата.
- Това се дължи на факта, че при представената реализация на стек се използва масив, чийто размер се определя по време на компилация на програмата и не може да бъде променян по време на изпълнение на програмата.
- За да се реши този проблем трябва да се използва механизмите за динамично управление на паметта.

Динамична памет: C–стил

- В езика C за динамично управление на паметта се използват функциите `malloc()` и `free()`.
- Работата на `malloc()` е да задели парче от динамичната памет, а с помощта на `free()` заделеното парче памет се освобождава.
- В езика C++ програмистите могат да използват тези функции, но тяхното използване е неудобно. Проблемът е че при използването на функцията `malloc()` не се извикват конструктори.

Динамична памет: C-стил

```
1 #include <cstdlib>
2 using namespace std;
3 class Foo {
4     int bar_;
5 public:
6     Foo(void) : bar_(0) {}
7 };
8 int main() {
9     Foo* ptr=(Foo*)malloc(sizeof(Foo));
10    //...
11    free(ptr);
12    return 0;
13 }
```

- В ред 9 се заделя памет за обект от типа Foo. Този обект, обаче, не са инициализира правилно, тъй като за него не се извиква конструктора Foo().
- Нужен е механизъм, който да обединява заделянето на динамична памет с извикването на конструктор.

Динамична памет

- В езика C++ за работа с динамичната памет се използват операторите **new** и **delete**.
- Нека е дефиниран класът `Foo`, който има два конструктора — конструктор по подразбиране и конструктор, който приема един аргумент.

```
3 class Foo {  
4     int bar_  
5 public:  
6     Foo(void) : bar_(0) {}  
7     Foo(int v, int w): bar_(v+w) {}  
8     int get_bar() const {return bar_;}  
9 };
```

Динамична памет

- Ако искаме да създадем обект от типа `Foo` в динамичната памет, трябва да използваме оператора **`new`**. Операторът **`new`** заделя необходимата за обекта памет и извиква конструктор, така че създаденият обект е правилно инициализиран.

```
11   Foo* ptr1=new Foo;  
12   Foo* ptr2=new Foo(21,21);  
13   Foo* arr1=new Foo[10];
```

- Когато **`new`** се използва по начина показан в ред 11, конструкторът, който се извиква, е конструкторът по подразбиране (конструктор без аргументи). Ако конструктор по подразбиране не е дефиниран, то ред 11 ще предизвика грешка при компилация.

Динамична памет

- Другата форма, за използване на оператора **new**, показана в ред 12, позволява да се извика конкретен конструктор и да му се предадат необходимите аргументи.

```
12 Foo* ptr2=new Foo(21,21);
```

- Третият начин за извикване на оператора **new** е показан на ред 13:

```
13 Foo* arr1=new Foo[10];
```

При тази употреба се създава масив от обекти от типа Foo. Размера на масива се предава в квадратни скоби. Конструкторът, който се извиква за всеки един от създадените обекти е конструкторът по подразбиране.

Динамична памет

- За унищожаване на динамично създадени обекти се използва операторът **delete**.

```
17  delete ptr1;  
18  delete ptr2;  
19  delete [] arr1;
```

- Когато трябва да се унищожи единичен обект, се използва операторът **delete**. Когато трябва да се унищожи масив от обекти се използва операторът **delete []**.

Динамична памет

```
1 #include <iostream>
2 using namespace std;
3 class Foo {
4     int bar_;
5 public:
6     Foo(void) : bar_(0) {}
7     Foo(int v, int w): bar_(v+w) {}
8     int get_bar() const {return bar_;}
9 };
10 int main() {
11     Foo* ptr1=new Foo;
12     Foo* ptr2=new Foo(21,21);
13     Foo* arr1=new Foo[10];
14     cout<<"ptr1->get_bar():"<<ptr1->get_bar()<<endl;
15     cout<<"ptr2->get_bar():"<<ptr2->get_bar()<<endl;
16     cout<<"arr1->get_bar():"<<arr1->get_bar()<<endl;
17     delete ptr1;
18     delete ptr2;
19     delete [] arr1;
20     return 0;
21 }
```

Динамична памет

Изходът на представената програма е следният:

```
lubo@dobby:~/school/cpp/notes> g++ code/lecture05-new01.cpp
```

```
lubo@dobby:~/school/cpp/notes> a.out
```

```
ptr1->get_bar():0
```

```
ptr2->get_bar():42
```

```
arr1->get_bar():0
```

Конструктори и деструктори

Нека разгледаме следният пример:

```
1 class Foo {
2     int* bar_;
3     int size_;
4 public:
5     Foo(int size)
6         : size_(size), bar_(new int[size])
7     {}
8     // ...
9 };
10 int main() {
11     Foo foo(100);
12     // ...
13     return 0;
14 }
```

При създаването на обекта `foo` в ред 11 динамично се заделя памет за масив от 100 цели. Тази памет не се освобождава никъде.

Конструктори и деструктори

Нека разгледаме следният пример:

```
1 #include <cstdio>
2 using namespace std;
3 class Foo {
4     FILE* bar_;
5 public:
6     Foo(const char* filename) :bar_(0) {
7         bar_=fopen(filename,"rw");
8     }
9     //...
10 };
11 int main() {
12     Foo foo("temp.txt");
13     //...
14     return 0;
15 }
```

При създаването на обекта `foo` в ред 12 се отваря файл, който не се затваря никъде.

Конструктори и деструктори

- Основната задача на конструкторът е да инициализира обекта за да може член-функциите на обекта да работят правилно.
- Коректната инициализация на даден обект понякога включва заделянето на динамична памет (като в разгледаният пример), отварянето на файлове или използването на някакъв друг ресурс, който изисква да бъде освободен след приключване на употребата му.

Деструктори

- Именно поради това такива класове се нуждаят от член-функция, която гарантирано се извиква при унищожаването на обектите. Тази функция се нарича **деструктор**.
- Основната задача на деструкторите е да освободят ресурсите, използвани от обекта.
- Деструкторите се извикват автоматично при унищожаването на обекта — при излизането му от областта на действие или при изтриване на обекта от динамичната памет.
- Най-честата употреба на деструктора е да освободи заделената в конструктора динамична памет.

Пример: Деструктор

```
1 class Foo {
2     int* bar_;
3     int size_;
4 public:
5     Foo(int size)
6         : size_(size), bar_(new int[size])
7     {}
8     ~Foo(void) {
9         delete [] bar_;
10    }
11    // ...
12 };
13 int main() {
14     Foo foo(100);
15     // ...
16     return 0;
17 }
```

Пример: Деструктор

```
1 #include <cstdio>
2 using namespace std;
3 class Foo {
4     FILE* bar_;
5 public:
6     Foo(const char* filename) :bar_(0) {
7         bar_=fopen(filename,"rw");
8     }
9     ~Foo(void) {
10        fclose(bar_);
11    }
12    //...
13 };
14 int main() {
15     Foo foo("temp.txt");
16     //...
17     return 0;
18 }
```

Пример: стек

```
1 #include <iostream>
2 using namespace std;
3
4 class Stack {
5     const static int chunk_=2;
6     int size_;
7     int *data_;
8     int top_;
9 public:
10    Stack(void)
11        : size_(chunk_),
12          data_(new int [chunk_]),
13          top_(-1)
14    {}
15    ~Stack(void) {
16        delete [] data_;
17    }
```

```
18  void push(int v) {
19      if(top_ >= (size_ - 1)) {
20          resize();
21      }
22      data_[++top_] = v;
23  }
24  int pop(void) {
25      if(top_ < 0) {
26          cout << "ERROR: □stack□is□empty..." << endl;
27          return 0;
28      }
29      return data_[top_--];
30  }
```

```
31 private:
32     void resize(void) {
33         cout << "Stack::resize()_called..." << endl;
34         int *temp=data_;
35         data_=new int[size_+chunk_];
36         for(int i=0;i<size_;i++)
37             data_[i]=temp[i];
38         delete [] temp;
39         size_+=chunk_;
40         cout << "Stack::resize()_new_size_is_"
41             << size_ << ">..." << endl;
42     }
43 };
```

```
44 int main(void) {  
45     Stack st;  
46     st.push(1);  
47     st.push(2);  
48     st.push(3);  
49  
50     cout << st.pop() << endl;  
51     cout << st.pop() << endl;  
52     cout << st.pop() << endl;  
53     return 0;  
54 }
```


Пример: стек

```
1 #include <iostream>
2 using namespace std;
3
4 class Stack {
5     struct Elem {
6         Elem* next_;
7         Elem* prev_;
8         int data_;
9
10        Elem(int v)
11            : next_(0),
12              prev_(0),
13              data_(v)
14        {}
15    };
16
17    Elem* first_;
18    Elem* last_;
```

```
19 void appendElem(Elem* newElem) {
20     if (last_ == 0) {
21         first_ = newElem;
22         last_ = newElem;
23     }
24     else {
25         last_ ->next_ = newElem;
26         newElem ->prev_ = last_;
27         last_ = newElem;
28     }
29 }
```

```
30 Elem* removeLastElem(void) {
31     if(last_==0)
32         return 0;
33     if(last_==first_) {
34         Elem* res=last_;
35         last_=0;
36         first_=0;
37         return res;
38     }
39
40     Elem* res=last_;
41     last_=res->prev_;
42     last_->next_=0;
43
44     res->prev_=0;
45     res->next_=0;
46     return res;
47 }
```

```
48 public :
49     Stack(void)
50         : first_(0) ,
51           last_(0)
52     {}
53
54     ~Stack(void) {
55         while(last_!=0){
56             Elem* el=removeLastElem();
57             delete el;
58         }
59     }
60     void push(int val) {
61         Elem* newElem=new Elem(val);
62         appendElem(newElem);
63     }
64
65     int pop(void) {
66         Elem* elem=removeLastElem();
67         if(elem==0){
68             cout << "ERROR: □stack□empty..." << endl;
69             return 0;
70         }
71         int res=elem->data_;
```

```
72     delete elem;
73     return res;
74 }
75 };
76
77 int main(void) {
78     Stack st;
79     st.push(1);
80     st.push(2);
81
82     cout << st.pop() << endl;
83     cout << st.pop() << endl;
84     cout << st.pop() << endl;
85     return 0;
86 }
```