Управление на динамичната памет (*Rev: 1.3*)

Любомир Чорбаджиев[1]
lchorbadjiev@elsys-bg.org

[1]Технологическо училище "Електронни системи"
Технически университет, София

5 март 2007 г.

Съдържание

Пример: стек

Основните операции, които се извършват със стека са:

- `push()` — поставя елемент на върха на стека;
- `pop()` — изтрива последния 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 #include <exception>
3 using namespace std;
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  {}
```

Пример: стек

```
12
13    void push(int v) {
14      if(top_>=(size_-1)) {
15        throw exception();
16      }
17      data_[++top_]=v;
18    }
19    int pop(void) {
20      if(top_<0) {
21        throw exception();
22      }
23      return data_[top_--];
24    }
25 };
```

Пример: стек

```
27 int main(void) {
28   Stack st;
29
30   try {
31     st.push(1);
32     st.push(2);
33     st.push(3);
34   } catch(const exception &e) {
35     cout << "exception() catched in push..."
36          << endl;
37   }
```

Пример: стек

```
38   try {
39     cout << st.pop() << endl;
40     cout << st.pop() << endl;
41     cout << st.pop() << endl;
42   } catch(const exception &e) {
43     cout << "exception()␣catched␣in␣pop..."
44          << endl;
45   }
46   return 0;
47 }
```

Пример: стек

Резултатът от изпълнението на програмата е следният:

```
lubo@kid ~/school/cpp/notes $ g++ code/lecture05-stack01.cpp
lubo@kid ~/school/cpp/notes $ ./a.out
exception() catched in push...
2
1
exception() catched in pop...
```

Пример: стек

- Основният недостатък на представената реализация е, че размерът на стека (броят не елементите, които можем да поставим в стека), се определя по време на компилация на програмата.
- Ако е необходимо размерът на стека да не е фиксиран по време на компилация, а да нараства в зависимост от броя на поставените в него елементи, то трябва да се използват 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 и трябва да се избягва. 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 };
```

Динамична памет: C–стил

```
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 конструкторът `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 два аргумента.

```
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`. Размерът 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 };
```

Динамична памет

```
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

```
1  class Foo {
2    int size_;
3    int* bar_;
4  public:
5    Foo(int size)
6      : size_(size),
7        bar_(new int[size])
8    {}
9    //...
10 };
```

Конструктори и деструктори

```
11 int bar() {
12   Foo foo(100);
13   //...
14   return 42;
15 }
```

При създаването на обекта `foo` в ред 12 динамично се заделя памет за масив от 100 цели. Тази памет не се освобождава никъде.

Конструктори и деструктори

```
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 };
```

Конструктори и деструктори

```
11 int bar() {
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 конструктора 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 разгледания 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 size_;
3    int* bar_;
4  public:
5    Foo(int size)
6      : size_(size), bar_(new int[size])
7    {}
8    ~Foo(void) {
9      delete [] bar_;
10   }
11   //...
12 };
```

Пример: Деструктор

```
13 int bar() {
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 };
```

Пример: Деструктор

```
14 int bar() {
15   Foo foo("temp.txt");
16   //...
17   return 0;
18 }
```

Пример: динамичен стек

```
1 #include <iostream>
2 #include <exception>
3 using namespace std;
4 class Stack {
5   const static int chunk_=2;
6   int size_;
7   int *data_;
8   int top_;
```

Пример: динамичен стек

```
 9 public:
10   Stack(void)
11     : size_(chunk_),
12       data_(new int[chunk_]),
13       top_(-1)
14   {}
15   ~Stack(void) {
16     delete [] data_;
17   }
```

Пример: динамичен стек

```
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      throw exception();
27    }
28    return data_[top_--];
29  }
```

Пример: динамичен стек

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

Пример: динамичен стек

```
43 int main(void) {
44   Stack st;
45   st.push(1);
46   st.push(2);
47   st.push(3);
48   try {
49     cout << st.pop() << endl;
50     cout << st.pop() << endl;
51     cout << st.pop() << endl;
52   } catch(const exception& e) {
53     cout<<"exception()␣catched␣in␣pop..."<<endl;
54   }
55   return 0;
56 }
```

Пример: масив с проверка на границите

```
1 #include <iostream>
2 #include <exception>
3 using namespace std;
4
5 class Array {
6   unsigned int size_;
7   int* data_;
8 public:
9   Array(unsigned int size=10)
10    : size_(size), data_(new int[size])
11  {}
12  ~Array(void) {
13    delete[] data_;
14  }
```

Пример: масив с проверка на границите

```
15    int& operator[](unsigned int index) {
16      if(index>=size_) {
17        cerr << "index␣out␣of␣bounds..." << endl;
18        throw exception();
19      }
20      return data_[index];
21    }
22    unsigned size() const {
23      return size_;
24    }
25  };
```

Пример: масив с проверка на границите

```
26 int main(void) {
27   Array v(3);
28   for(int i=0;i<3;++i) {
29     v[i]=i;
30   }
31   for(int i=0;i<3;i++) {
32     cout << "v[i]=" << v[i] << endl;
33   }
34   try {
35     v[3]=5;
36   } catch(const exception& e) {
37     cout << "exception␣catched..." << endl;
38   }
39   return 0;
40 }
```

Пример: масив с проверка на границите

```
lubo@kid ~/school/cpp/notes $ g++ code/lecture08-array02.cpp
lubo@kid ~/school/cpp/notes $ ./a.out
v[i]=0
v[i]=1
v[i]=2
index out of bounds...
exception catched...
```

Копиращ конструктор

- По подразбиране всички обекти могат да бъдат копирани. Всеки клас притежава копиращ конструктор, който е отговорен за копирането на обектите от съответният клас.
- Копиращият конструктор за класа `X` има сигнатура `X::X(const X&)`.
- Ако за даден клас не е дефиниран копиращ конструктор, то компилаторът генерира копиращ конструктор по подразбиране. Семантиката на този конструктор е да копира всички член-променливи на класа.

Копиращ конструктор

- За класа `Point` поведението на генерирания от компилатора копиращ конструктор е еквивалентно на следното:

```
1 class Point {
2   double x_, y_;
3 public:
4   Point(const Point& p)
5     : x_(p.x_), y_(p.y_)
6   {}
7   //...
8 };
```

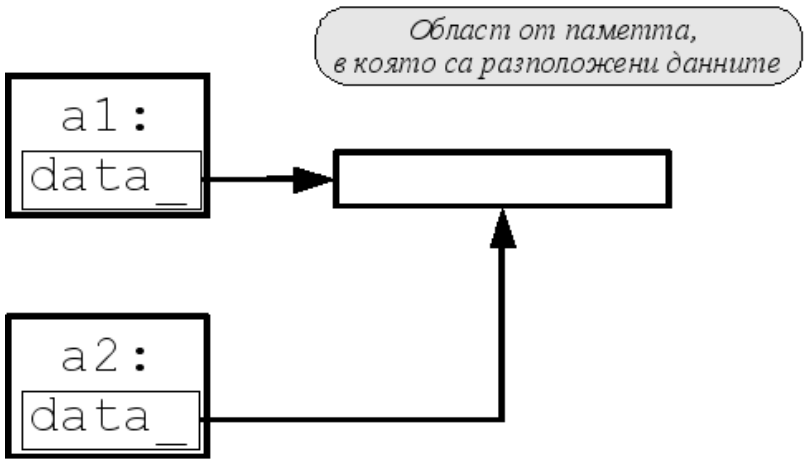
- Ако подразбиращото се поведение на този конструктор е неподходящо за даден клас, то потребителят трябва да дефинира сам копиращ конструктор.

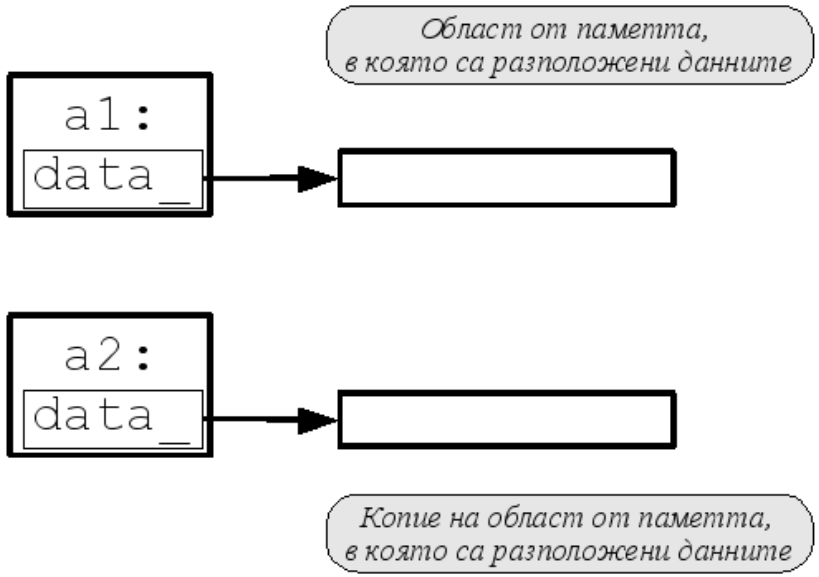
Копиращ конструктор

- В повечето случаи подразбиращото се поведение на копиращия конструктор е напълно удовлетворително.
- Нека отново да разгледаме дефинирания от нас масив, с проверката на границите.

```
5  class Array {
6    unsigned int size_;
7    int* data_;
8  public:
9    Array(unsigned int size=10)
10     : size_(size), data_(new int[size])
11   {}
12   ~Array(void) {
13     delete[] data_;
14   }
```

Копиращ конструктор





Копиращ конструктор

- Подразбиращият се копиращ конструктор копира член-променливите на класа. Това означава, че ще се копират член променливите `data_` и `size_`. Областта от паметта, към която сочи `data_`, няма да бъде копирана.
- За да се обезпечи коректно поведение на масива при копиране е необходимо да се предефинира копиращият конструктор.

Копиращ конструктор: пример

```
6  class Array {
7    unsigned int size_;
8    int* data_;
9  public:
10   Array(unsigned int size=10)
11     : size_(size), data_(new int[size_])
12   {}
13   ~Array(void) {
14     delete[] data_;
15   }
```

Копиращ конструктор: пример

```
16  Array(const Array& other)
17    : size_(other.size_), data_(new int[size_])
18  {
19    for(unsigned int i=0; i< size_; i++)
20      data_[i]=other.data_[i];
21  }
22  int& operator[](unsigned int index) {
23    if(index>=size_) {
24      cerr << "index␣out␣of␣bounds..." << endl;
25      throw exception();
26    }
27    return data_[index];
28  }
```

Копиращ конструктор: пример

```
29    unsigned size() const {
30      return size_;
31    }
32  };
33  int main(void) {
34    Array a1(3);
35    for(int i=0;i<3;++i) {
36      a1[i]=i;
37    }
38    Array a2=a1;
39    for(int i=0;i<3;i++) {
40      cout << "a2[i]=" << a2[i] << endl;
41    }
42    return 0;
43  }
```

Копиращ конструктор: пример

```
lubo@kid:~/school/notes> ./a.out
a2[i]=0
a2[i]=1
a2[i]=2
```

Копиращ конструктор

- Обърнете внимание, че като аргумент на копиращия конструктор се използва препратка — `X::X(const X& x)`.
- Ако в дефиницията на копиращия конструктор не се използва препратка — `X::X(X x)`, — то това ще доведе до безкрайна рекурсия. Проблемът е, че при предаване на аргумента по стойност, се извършва копиране, което води до извикване на копиращ конструктор.
- Ако е необходимо да се забрани копирането на обектите на даден клас, то копиращият конструктор на класа трябва да се декларира като **private**.

Оператор за присвояване

- По подразбиране за всички обекти може да се използва оператор за присвояване. Всеки клас притежава оператор за присвояване, който е отговорен за присвояване на обекти от съответния клас.
- Операторът за присвояване на класа `X` има сигнатура `X& X::operator=(const X&)`.
- Ако за даден клас не е дефиниран оператор за присвояване, то компилаторът генерира оператор за присвояване по подразбиране. Семантиката на този оператор е да копира всички член-променливи на класа.

Оператор за присвояване

- За класа `Point` поведението на подразбиращия се (генериран от компилатора) оператор за присвояване е еквивалентно на следното:

```
1 class Point {
2   //...
3   Point& operator=(const Point& other){
4     x_=other.x_;
5     y_=other.y_;
6     return *this;
7   }
8 };
```

- Ако подразбиращото се поведение на този оператор е неподходящо за даден клас, то потребителят трябва да дефинира сам оператор за присвояване.

Оператор за присвояване: пример

- В повечето случаи подразбиращото се поведение на оператора за присвояване е напълно удовлетворително.
- За да се обезпечи коректно поведение на масива при присвояване е необходимо да се предефинира оператора за присвояване.

```
25  Array& operator=(const Array& other) {
26    if(this!=&other) {
27      delete [] data_;
28      size_=other.size_;
29      data_=new int[size_];
30      for(unsigned i=0;i<size_;i++)
31        data_[i]=other.data_[i];
32    }
33    return *this;
34  }
```