

Управление на динамичната памет

Любомир Чорбаджиев¹
lchorbadjiev@elsys-bg.org

¹Технологическо училище “Електронни системи”
Технически университет, София

29 март 2009 г.



Съдържание

- 1 Пример: статичен стек
- 2 Динамична памет
- 3 Конструктори и деструктори
- 4 Пример: динамичен стек
- 5 Пример: масив с проверка на границите
- 6 Копиращ конструктор
- 7 Оператор за присвояване

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

```
1
2 void push(int v) {
3     if(top_ >=(size_-1)) {
4         throw exception();
5     }
6     data_[++top_]=v;
7 }
8 int pop(void) {
9     if(top_ <0) {
10        throw exception();
11    }
12    return data_[top_--];
13 }
14 };
```

Пример: стек

```
1 int main(void) {  
2     Stack st;  
3  
4     try {  
5         st.push(1);  
6         st.push(2);  
7         st.push(3);  
8     } catch(const exception &e) {  
9         cout << "exception() caught in push..."  
10            << endl;  
11     }
```

Пример: стек

```
1  try {
2      cout << st.pop() << endl;
3      cout << st.pop() << endl;
4      cout << st.pop() << endl;
5  } catch(const exception &e) {
6      cout << "exception() caught in pop..."
7          << endl;
8  }
9  return 0;
10 }
```

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() caught in push...
2
1
exception() caught in pop...
```


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.
- 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

```
1 int main() {  
2     Foo* ptr=(Foo*)malloc(sizeof(Foo));  
3     //...  
4     free(ptr);  
5     return 0;  
6 }
```

- В ред 2 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.

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

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.

```
1 Foo* ptr1=new Foo ;
2 Foo* ptr2=new Foo (21 ,21) ;
3 Foo* arr1=new Foo [10] ;
```

- Когато **`new`** се използва по начина показан в ред 2, конструкторът, който се извиква, е конструкторът по подразбиране (конструктор без аргументи). Ако конструктор по подразбиране не е дефиниран, то ред 2 ще предизвика грешка при компилация.

Динамична памет

- Другата форма, за използване на оператора **new**, показана в ред 3, позволява да се извика конкретен конструктор и да му се предадат необходимите аргументи.

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

- Третият начин за извикване на оператора **new** е показан на ред 4:

```
1 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**.

```
1 delete ptr1;  
2 delete ptr2;  
3 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

```
1 int main() {
2     Foo* ptr1=new Foo;
3     Foo* ptr2=new Foo(21,21);
4     Foo* arr1=new Foo[10];
5     cout<<"ptr1->get_bar():"<<ptr1->get_bar()<<endl;
6     cout<<"ptr2->get_bar():"<<ptr2->get_bar()<<endl;
7     cout<<"arr1->get_bar():"<<arr1->get_bar()<<endl;
8     delete ptr1;
9     delete ptr2;
10    delete [] arr1;
11    return 0;
12 }
```

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

```
1 int bar() {  
2     Foo foo(100);  
3     //...  
4     return 42;  
5 }
```

При създаването на обекта `foo` в ред 2 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

```
1 int bar() {  
2     Foo foo("temp.txt");  
3     //...  
4     return 0;  
5 }
```

При създаването на обекта `foo` в ред **2** 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

```
1 int bar() {  
2     Foo foo(100);  
3     //...  
4     return 0;  
5 }
```

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

```
1 int bar() {  
2     Foo foo("temp.txt");  
3     //...  
4     return 0;  
5 }
```

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

```
1 public :  
2 Stack(void)  
3     : size_(chunk_),  
4       data_(new int [chunk_]),  
5       top_(-1)  
6 {}  
7 ~Stack(void) {  
8     delete [] data_;  
9 }
```

Пример: динамичен стек

```
1 void push(int v) {
2     if(top_ >=(size_-1)) {
3         resize();
4     }
5     data_[++top_]=v;
6 }
7 int pop(void) {
8     if(top_ <0){
9         throw exception();
10    }
11    return data_[top_--];
12 }
```


Пример: динамичен стек

```
1 private :
2     void resize(void) {
3         cout << "Stack::resize()_called..." << endl;
4         int *temp=data_;
5         data_=new int[size_+chunk_];
6         for(int i=0;i<size_;i++)
7             data_[i]=temp[i];
8         delete [] temp;
9         size_+=chunk_;
10        cout << "Stack::resize()_new_size_is_" <<
11            << size_ << ">..." << endl;
12    }
13};
```

Пример: динамичен стек

```
1 int main(void) {  
2     Stack st;  
3     st.push(1);  
4     st.push(2);  
5     st.push(3);  
6     try {  
7         cout << st.pop() << endl;  
8         cout << st.pop() << endl;  
9         cout << st.pop() << endl;  
10    } catch(const exception& e) {  
11        cout<<"exception() caught in pop..."<<endl;  
12    }  
13    return 0;  
14 }
```

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

```
1  int& operator[](unsigned int index) {  
2      if(index>=size_) {  
3          cerr << "index out of bounds..." << endl;  
4          throw exception();  
5      }  
6      return data_[index];  
7  }  
8  unsigned size() const {  
9      return size_;  
10 }  
11};
```

Пример: масив с проверка на границите

```
1 int main(void) {  
2     Array v(3);  
3     for(int i=0;i<3;++i) {  
4         v[i]=i;  
5     }  
6     for(int i=0;i<3;i++) {  
7         cout << "v[i]=" << v[i] << endl;  
8     }  
9     try {  
10        v[3]=5;  
11    } catch(const exception& e) {  
12        cout << "exception caught..." << endl;  
13    }  
14    return 0;  
15 }
```

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 caught...
```

Копирац 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 $X::X(\text{const } X\&)$.
- 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 };
```

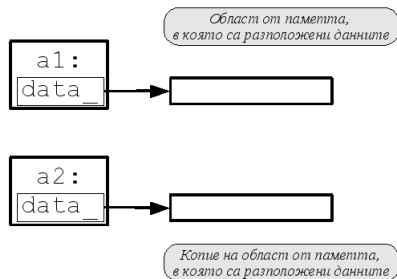
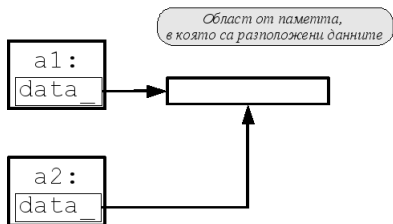
- Ако подразбиращото се поведение на този конструктор е неподходящо за даден клас, то потребителят трябва да дефинира сам копиращ конструктор.

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.

```
1 class Array {  
2     unsigned int size_  
3     int* data_  
4 public:  
5     Array(unsigned int size=10)  
6         : size_(size), data_(new int[size])  
7     {}  
8     ~Array(void) {  
9         delete[] data_  
10    }
```

Копирац 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.

Копирац конструктор: пример

```
1 class Array {
2     unsigned int size_;
3     int* data_;
4 public:
5     Array(unsigned int size=10)
6         : size_(size), data_(new int[size_])
7     {}
8     ~Array(void) {
9         delete[] data_;
10    }
```

Копирац конструктор: пример

```
1  Array(const Array& other)
2      : size_(other.size_), data_(new int[size_])
3  {
4      for(unsigned int i=0; i< size_; i++)
5          data_[i]=other.data_[i];
6  }
7  int& operator[](unsigned int index) {
8      if(index>=size_) {
9          cerr << "index out of bounds..." << endl;
10         throw exception();
11     }
12     return data_[index];
13 }
```

Копирац конструктор: пример

```
1  unsigned size() const {
2      return size_;
3  }
4 };
5 int main(void) {
6     Array a1(3);
7     for(int i=0;i<3;++i) {
8         a1[i]=i;
9     }
10    Array a2=a1;
11    for(int i=0;i<3;i++) {
12        cout << "a2[i]=" << a2[i] << endl;
13    }
14    return 0;
15 }
```

Копирац конструктор: пример

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

Копирац 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 $X\& X::\mathbf{operator}=(\mathbf{const} X\&)$.
- 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.

```
1  Array& operator=(const Array& other) {  
2      if(this != &other) {  
3          delete [] data_;  
4          size_ = other.size_;  
5          data_ = new int[size_];  
6          for(unsigned i=0; i<size_; i++)  
7              data_[i] = other.data_[i];  
8      }  
9      return *this;  
10 }
```