Лекция 5. Динамическое управление памятью. Конструкторы и деструкторы

Казахский национальный университет имени аль-Фараби

Кафедра компьютерных наук

Объектно-ориентированное программирование

Лектор: старший преподаватель, PhD, Кенжебек Ержан Ғалымжанұлы

Цель лекции

Цель лекции

Изучить динамическое выделение и освобождение памяти в C++, причины утечек памяти, а также назначение и виды конструкторов и деструкторов.

Ожидаемые результаты

1. Выделять и освобождать память с помощью new и delete
2. Создавать динамические массивы
3. Предотвращать утечки памяти
4. Писать конструкторы (в том числе копирования) и деструкторы

План лекции

- Динамическое выделение памяти (allocate memory)
- Создания динамического массива
- Динамическое освобождение памяти (deallocate memory)
- Утечка памяти (memory leak)
- Конструкторы
- Деструкторы

Динамическое выделение памяти

- Динамическое выделение памяти достигается в C с помощью функций malloc(), calloc() и realloc(). В C++ это достигается с помощью оператора **new**. Наглядный пример:

```
#include<iostream>
void main()
{
  int * iPtr;
  iPtr=new int;
  *iPtr=10;
  cout<<*iPtr<<endl;
}
```

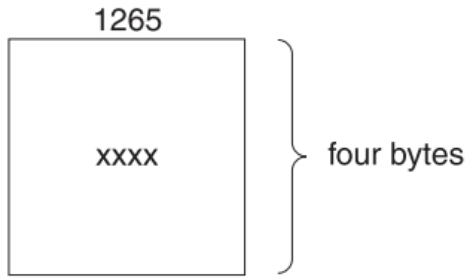
Output : 10

Оператор new выделяет память

синтаксис: new

<pointer> = new <data_type>;

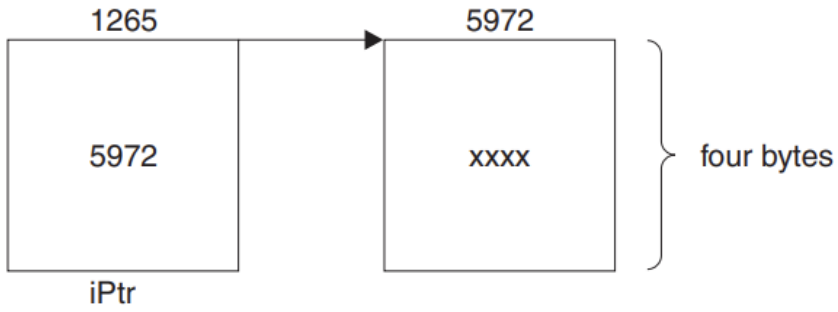
Statement: `int * iptr;`



Four bytes get allocated for `iPtr` containing junk value at the bytes with addresses from **1265** to **1268** (say).

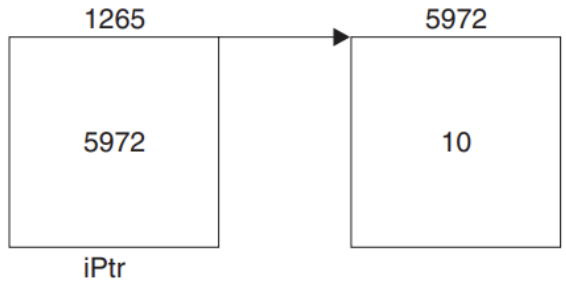
Динамическое выделение памяти

Statement: `iptr = new int;`



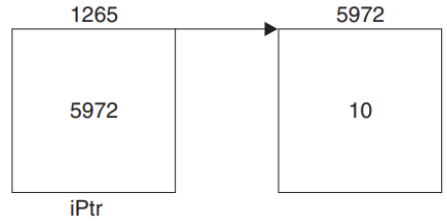
Блок памяти по адресу от 5972-5975 = 4 байт

Statement: `*iPtr = 10;`



*** - оператор разыменования**

Statement: `cout<<*iPtr<<endl;`

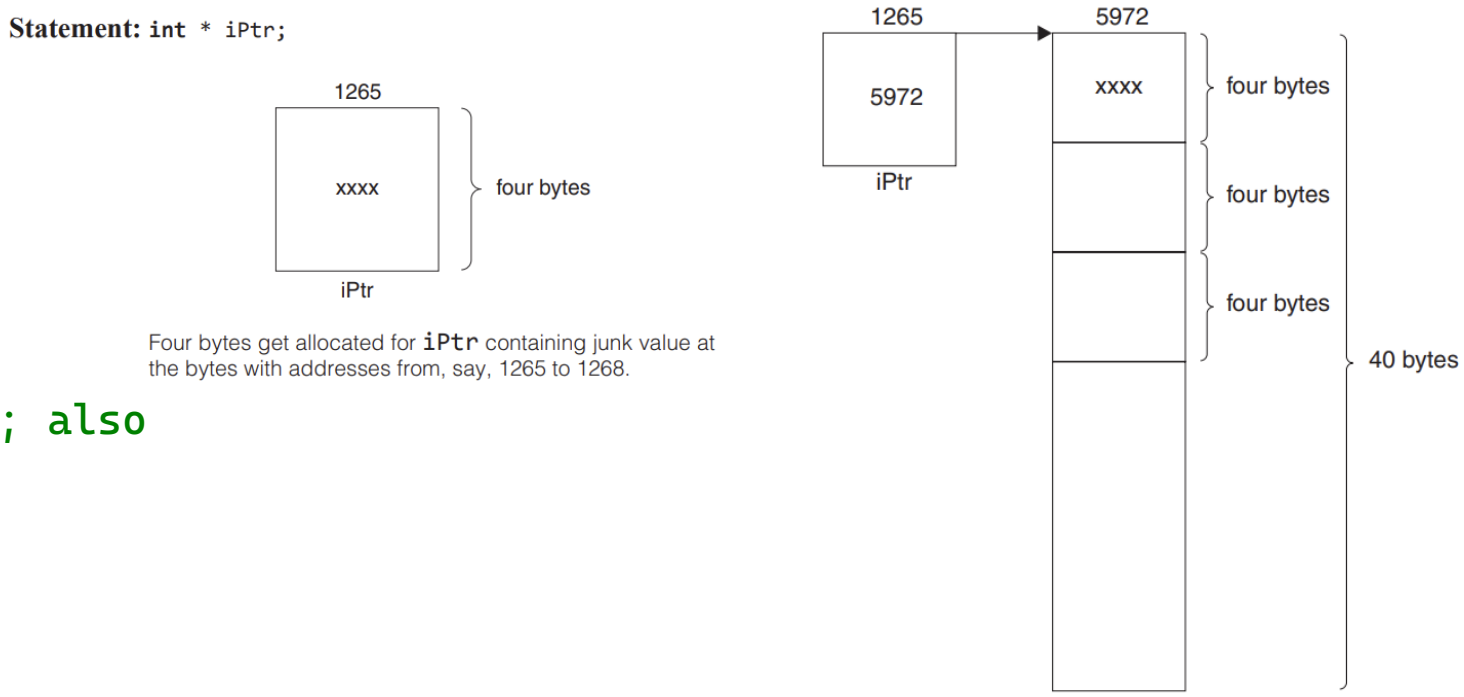


Создание дин. массива с помощью оператора new

```
Синтаксис создания динамического массива
<pointer> = new <data_type>[<number_of_elements>];
```

```
#include<iostream>
#define SIZE 10
void main()
{
int* iPtr;
iPtr = new int[SIZE];
for (int i = 0; i < SIZE; i++)
iPtr[i] = i; //can write cin>>iPtr[i]; also

for (int j = 0; j < SIZE; j++)
cout << iPtr[j] << endl;
}
```



Динамическое освобождение памяти

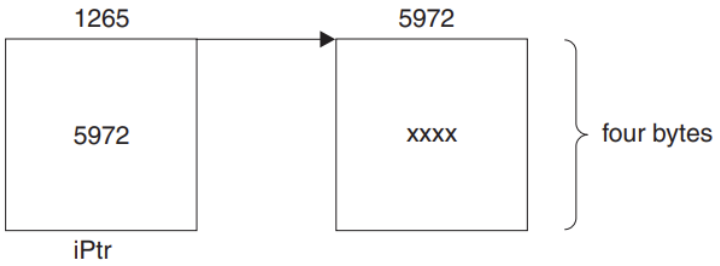
- Мы уже знаем, что динамически выделяемый блок памяти может быть динамически освобожден. Когда он больше не используется, д**инамически выделенный блок памяти обязательно должен быть возвращен в ОС**. В С динамическое освобождение памяти достигается с помощью функции free(). Динамически выделенные блоки памяти могут быть возвращены ОС в С++ с помощью оператора delete.
- Какая необходимость освобождать динамически выделяемый блок памяти? Что будет, если не вернуть ОС динамически выделенный блок памяти?

Memory leak

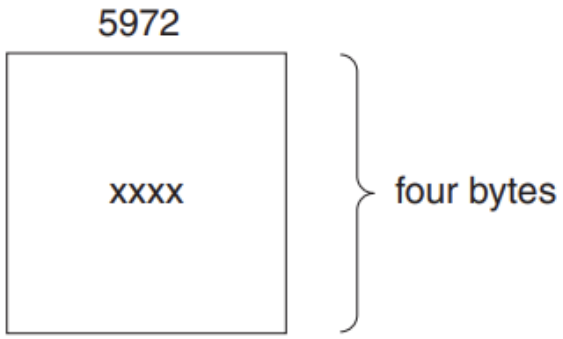
```
#include<iostream>

void abc()
{
int* iPtr;
iPtr = new int;
/*
rest of the abc() function
*/
}

int main()
{
abc(); //call to the abc() function
/*
rest of the main() function
*/
return 0;
}
```



После завершения выполнения abc() память для самого iPtr освобождается. Но память в области кучи остается заблокированной (без ссылок) заблокированный блок памяти.



Утечка памяти

Предотвращения утечки памяти

- Более того, к этому блоку памяти нельзя получить доступ, так как единственный указатель (iPtr), который указывал на него, сам был удален из стека. Этот блок памяти, который больше не нужен, можно и нужно вернуть ОС. Динамически выделяемый блок памяти можно освободить, передав указатель, указывающий на него, в качестве операнда оператору удаления.

```
#include<iostream>

void abc()
{
int* iPtr;
iPtr = new int;
/*
rest of the abc() function
*/
delete iPtr;
}

int main()
{
abc(); //call to the abc() function
/*
rest of the main() function
*/
return 0;
}
```

Когда мы освобождаем, только тогда возвращается память к ОС, в дальнейшем ОС может выделить освобожденную память для последующих запросов блоков памяти.

Предоставление динамически выделяемого блока памяти вызывающей функции

```
void abc(int    ** p)
{
   /*
      some complex algorithm
   */

   *p = new int;

   /*
      rest of the abc() function
   */
}
void main()
{
   int * iPtr;
   abc(&iPtr);
   /*
      rest of the main() function
   */
         delete iPtr;
}
```

- Адрес iPtr хранится в двойном указателе (указатель на указатель должен быть двойным указателем). Блок памяти выделяется и его адрес сохраняется в iPtr путем разыменования указателя, указывающего на него. Наше очевидное желание, чтобы динамически выделяемый блок памяти сохранялся даже после завершения функции abc(). После завершения функции abc() iPtr, являющаяся локальной переменной в вызывающей функции, будет указывать на динамически выделенный блок памяти.

Конструктор

- До этого мы использовали метод сеттер для инициализации полей объекта класса.
- Как правило, удобнее инициализировать поля объекта **автоматически** в момент его создания, а не явно вызывать в программе соответствующий метод. Такой способ инициализации реализуется с помощью особого метода класса, называемого **конструктором**. **Конструктор** — это метод класса, выполняющийся автоматически в момент создания объекта.
- В отличие от обычных методов, конструкторы имеют определенные правила их именования:
  - конструкторы всегда должны иметь то же имя, что и класс (учитываются верхний и нижний регистры);
  - конструкторы не имеют типа возврата (даже void).

Обратите внимание, конструкторы предназначены только для выполнения инициализации. Не следует пытаться вызывать конструктор для повторной инициализации существующего объекта

Конструкторы по умолчанию или zero constructor

- Конструктор, который не имеет параметров (или содержит параметры, которые все имеют значения по умолчанию), называется **конструктором по умолчанию**. Он вызывается, если пользователем не указаны значения для инициализации.

```
#include<iostream>
using namespace std;

class Distance {
private:
 int feet;
 float inches;
public:
//default constructor
 Distance() {
  feet = 0;
  inches = 0.0;
  cout << "Called Default Constructor" << endl;
  }
};

int main() {
Distance d1;
return 0;
}
```

Конструкторы с параметрами

- Хотя конструктор по умолчанию отлично подходит для обеспечения инициализации наших классов значениями по умолчанию, часто может быть нужно, чтобы экземпляры нашего класса имели определенные значения, которые мы предоставим позже. Конструкторы также могут быть объявлены с параметрами.

```
//default constructor
 Distance() {
  feet = 0;
  inches = 0.0;
  cout << "Called Default Constructor" << endl;
  }

//constructor with parameters
Distance(int a, float b) {
feet = a;
inches = b;
cout << "Called Constructor with parameters" << endl;
}
```

```
int main

//Прямая инициализация
Distance d2(3, 4.5);
//uniform-инициализация C++11
Distance d2{3, 4.5};

//можем указать только один параметр для конструктора
// с параметрами, а второе значение будет значением по умолчанию:
Distance d2(5)
```

- Обратите внимание, теперь у нас есть два конструктора: конструктор по умолчанию, который будет вызываться, если мы не предоставим значения, и конструктор с параметрами, который будет вызываться, если мы предоставим значения. Эти два конструктора могут мирно сосуществовать в одном классе благодаря перегрузке функций. Фактически, вы можете определить любое количество конструкторов до тех пор, пока у них будут уникальные параметры (учитывается их количество и тип).

Копирующий конструктор по умолчанию

- Теперь мы рассмотрим третий способ инициализации объекта, использующий значения полей уже существующего объекта. К вашему возможному удивлению, для этого не нужно самим создавать специальный конструктор, поскольку такой конструктор предоставляется компилятором для каждого создаваемого класса и называется копирующим конструктором по умолчанию. В обоих случаях был вызван **копирующий конструктор по умолчанию**.

```
Distance d1; //вызывает дефолт конструктор

Distance d2(3, 4.5); //вызывает конструктор с параметрами (аргументами)

Distance d3(d2); //call default copy constructor

Distance d4 = d2; //call default copy constructor
```

Можно подумать, что данный оператор выполняет операцию присваивания, но это не так. Здесь, как и в предыдущем случае, вызывается конструктор копирования по умолчанию. Оба оператора выполняют одинаковые действия и равноправны в использовании

Результат:
d2 = 3’-4.5’’
d3 = 3’-4.5’’
d4 = 3’-4.5’’

Определение конструктора копирования

- Так же, как мы можем явно определить конструктор по умолчанию, так же мы можем явно определить и конструктор копирования. Конструктор копирования выглядит следующим образом:

```
class Distance {
private:
 int feet;
 float inches;
public:
 void show();
 //default constructor
 Distance() {
  feet = 0;
  inches = 0.0;
  cout << "Called Default Constructor" << endl;
}

//constructor with parameters
Distance(int a, float b) {
 feet = a;
 inches = b;
 cout << "Called Constructor with parameters" << endl;
}

//copy constructor
Distance(const Distance& obj) {
 feet = obj.feet;
 inches = obj.inches;
 cout << "Copy Constructor" << endl;
 }
};
```

Деструктор

- Деструктор – это специальный тип метода класса, который выполняется при удалении объекта класса. В то время как конструкторы предназначены для инициализации класса, деструкторы предназначены для очистки памяти после него.

```
~Distance() {
cout << "Destructor " << endl;
}
```

Выводы

1. Операторы new и delete управляют памятью в куче: каждому new должен соответствовать delete.
2. Динамический массив создаётся выражением new тип[размер]; память освобождается оператором delete[].
3. Утечка памяти возникает, когда указатель на выделенный блок потерян до освобождения блока.
4. Конструктор автоматически вызывается при создании объекта и инициализирует его; бывают конструкторы по умолчанию, с параметрами и копирования.
5. Деструктор вызывается при уничтожении объекта и освобождает занятые им ресурсы.

Контрольные вопросы

1. Чем динамическое выделение памяти отличается от статического?
2. Как работают операторы new и delete? Как создать и удалить динамический массив?
3. Что такое утечка памяти? Приведите пример и способ предотвращения.
4. Что такое конструктор? Чем конструктор по умолчанию отличается от конструктора с параметрами?
5. Когда вызывается конструктор копирования?
6. Для чего нужен деструктор и когда он вызывается?

Список литературы

Основная литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. 4-е изд. — СПб.: Питер, 2004.
2. Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином, 2011.
3. Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Базовый курс. 5-е изд. — М.: Вильямс, 2014.

Дополнительная литература

4. Шилдт Г. Полный справочник по C++. 4-е изд. — М.: Вильямс, 2006.
5. Майерс С. Эффективный и современный C++. — М.: Вильямс, 2016.
6. Страуструп Б. Программирование: принципы и практика с использованием C++. 2-е изд. — М.: Вильямс, 2016.

Электронные ресурсы

7. Учебник по C++ (learncpp): www.learncpp.com
8. Справочник по C++ и стандартной библиотеке: en.cppreference.com

- Спасибо за внимание!