Лекция 4. Введение в C++

Наследование, множественное наследование.
Конструкторы, деструкторы.
Виртуальные функции.

Наследование.

- Объект *производного* класса обладает всеми методами и атрибутами *базового*. Помимо них, в него можно добавить новые.

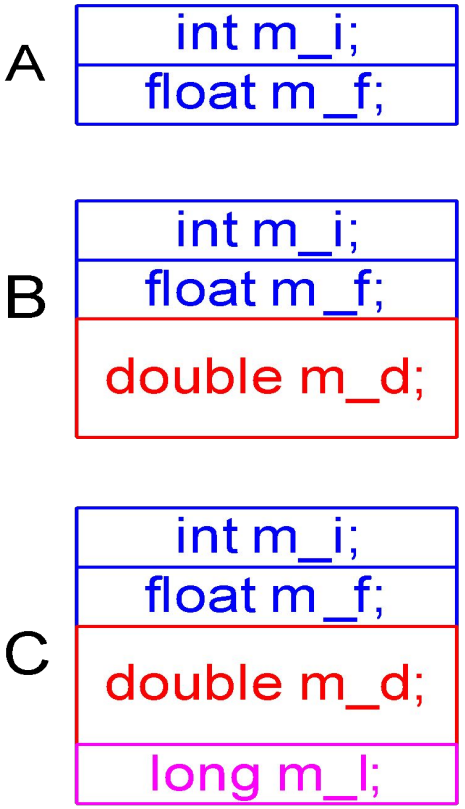
- *Производный* класс может быть *базовым* для какого-то другого класса, т.е. иерархия классов может быть сколь угодно глубокой.

Наследование. Пример.

```
class A{
    int   m_i;
    float m_f;
};

class B: A{
    double m_d;
};


class C: B{
    long m_l;
};
```

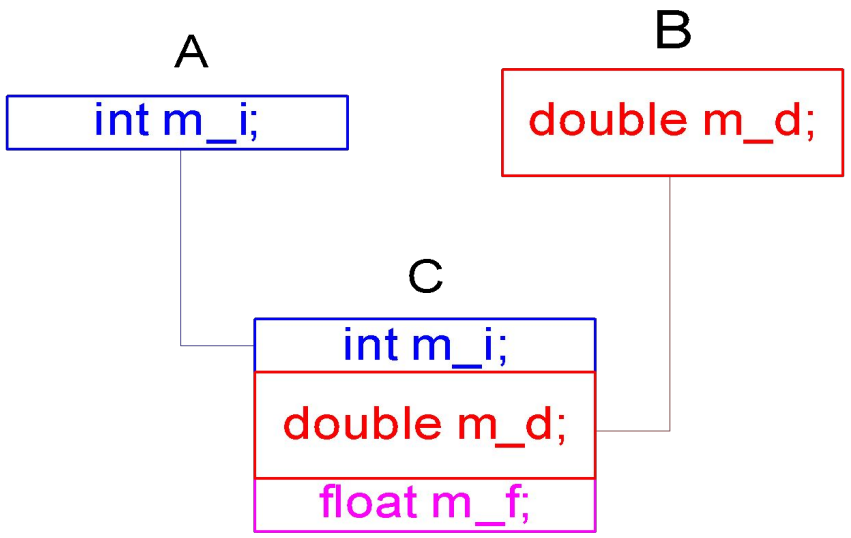


Множественное наследование.

```
class A{
    int m_i;
};

class B {
    double m_d;
};

class C: A, B{
    float m_f;
};
```



Модификаторы доступа.

- *public*
  Поле/метод с таким модификатором доступно отовсюду (из самого класса, из его потомков, из глобальных функций).
- *protected*
  Доступно из самого класса и производных от него, но недоступно извне.
- *private*
  Доступно только из самого класса.

Модификаторы доступа.

- При наследовании также указывается модификатор доступа. В соответствии с ним в производном классе изменяются уровни доступа.
- Для класса можно указать дружественные классы и функции (*friend*). Они будут иметь доступ ко всем полям класса.

Модификаторы доступа. Пример.

```
class A{
  private:
    int m_priv;
  protected:
    int m_prot;
  public:
    int m_pub;
};
class B: public A;
class C: protected A;
class D: private A;
```

- m_priv доступно только из класса A;
- m_prot доступно из классов A и производных от него (в B,C это поле остается protected, в D становится private);
- m_pub доступно из классов A, в производных от него, а также извне. В классе B это поле остается public, в классе C становится protected, в классе D – private.

Конструкторы.

- *Конструктор* – специальный метод класса, который выполняется каждый раз, когда создается новый объект этого класса.
- Имя конструктора совпадает с именем класса.
- Возвращаемого значения конструктор не имеет.

Конструкторы.

Стандартный конструктор (без аргументов).

```
class String
{
  char *str;
  int length;
public:
  String();
};

String::String()
{
  str = NULL;
  length = 0;
}
```

- Теперь при создании переменных будет вызываться наш метод:

- // Конструктор будет вызван
  // в каждой из этих строк:

```
String str;
String *sptr = new String;
```

Конструкторы.

Конструкторы с дополнительными параметрами.

```
class String
{
  char *str;
  int length;
  public:
  String(const char* p);
};

String::String(const char* p)
{
    length = strlen(p);
    str = new char[length + 1];
    if (str == 0) {
       // обработка ошибок
    }
    // копирование строки
    strcpy(str, p);
}
```

- Теперь при создании переменных можно инициализировать их с помощью нового конструктора:

- ```
  String s2("Строчка");

  char* cp;
  String* ssptr =
        new String(cp);
  ```
  ```

Конструкторы.

Конструктор копирования.

```
class String
{
  char *str;
  int length;
  public:
  String(const String& s);
};

String::String(const String& s)
{
  length = s.length;
  str = new char[length + 1];
  strcpy(str, s.str);
}
```

- // Создаем объект **a**
  // с начальным значением
  `String a("Astring");`
  // Используем конструктор
  // копирования для создания
  // объекта **b**
  `String b(a);`
  // Изменяем объект **b**
  // Объект **a** остается
  // неизменным
  `b.Append(a);`

Конструкторы.

Конструктор копирования по умолчанию.

```
class String
{
  char *str;
  int length;
  public:
  String(const String& s);
};

String::String(const String& s)
{
  length = s.length;
  str = s.str;
}

void String::Append(const String& s)
{
  length += s.length;
  char* tmp = new char[length + 1];
  if (tmp == 0) {
    // обработка ошибки
  }
  strcpy(tmp, str);
  strcat(tmp, s.str);
  delete [] str;
  str = tmp;
}
```

- // Создаем объект **a**
  // с начальным значением
  `String a("Astring");`
  // a.str указывает на строку “Astring”

  // Используем конструктор
  // копирования для создания
  // объекта **b**
  `String b(a);`
  // b.str указывает на ту же самую строку “Astring”

  // Изменяем объект **b**
  // Объект **a** остается
  // неизменным
  `b.Append(a);`
  // Эта функция уничтожила старую строчку
  // и создала новую в другом месте (на
  // которую теперь указывает b.str).
  // Но a.str по-прежнему указывает на уже
  // несуществующую старую строчку.

Деструкторы.

- *Деструктор* – специальный метод класса, который выполняется при уничтожении объекта.
- Обычно в деструкторе освобождаются ресурсы, использованные данным объектом.
- Имя деструктора – это имя класса, перед которым добавлен знак ‘~’.

Деструкторы.
Пример.

```
// Деструктор для класса String
String::~String()
{
  if (str)
    delete [] str;
}

// Некоторая функция
int funct(void)
{
  // Вызывается конструктор для Str
  String Str (“String”);
  // Здесь вызывается конструктор для pStr
  String *pStr = new String(“String 2”);
  . . .
  // Здесь вызовется деструктор для pStr
  delete pStr;
  // Здесь вызовется  деструктор для Str
  return 0;
}
```

Виртуальные функции.

- *Виртуальные функции* используются для того, чтобы можно было работать с объектами разных типов так, как будто они одного типа.

- Выбор функции, которую необходимо вызвать, производится во время исполнения (не во время компиляции).

Виртуальные функции. Пример.

```
class Shape {
    int cx, cy;
    virtual void print();
};

class Circle: Shape {
    int r;
    virtual void print();
};

class Rect: Shape {
    int w,h;
    virtual void print();
};
```

- Пусть у нас есть массив Shape* shapes[];
- Несмотря на то, что тип элементов массива – Shape*, т. е. указатель на объект типа Shape, реально по указателю может лежать любой объект.
- Если мы напишем такой цикл, то вызываться будет функция print того типа, который на самом деле находится по адресу shapes[i]:

```
for(int i = 0; i < N; i++) {
    shapes[i]->print();
}
```

Q&A

Thanks!