Специальные функции-члены класса

- Функция-член класса с тем же именем, что и у класса, называется **конструктором**.
- Она используется для построения объектов этого класса.
- Конструктор не должен возвращать никакого значения, даже `void`.

```
class Complex
 { private:
     double r, m;
   public:
     Complex(double r, double m) :
         r(r), m(m) { }
   ...
  };
```

- *Конструктор умолчания* класса *X* – это конструктор класса *X*, вызываемый без параметров.

```
X::X()
```

```
X::X(int = 0)
```

```
class Complex
 { private:
     double r, m;
   public:
     Complex() : r(0), m(0) { }
   ...
  };

Complex x;
```

```
class Complex
 { private:
     double r, m;
   public:
     Complex(double nr = 0, double nm = 0) :
         r(nr), m(nm) { }
   ...
  };

Complex y1(-6, 3);
Complex y2;
```

- *Конструктор копирования* для класса *X* – это конструктор, который может быть вызван для копирования объекта класса *X*, т.е. такой конструктор, который может быть вызван с одним параметром – ссылкой на объект класса *X*.

```
X::X(const X&)
```

```
X::X(X&, int = 0)
```

Конструктор копирования используется:

- при инициализации переменных (объектом того же класса);
- при передаче аргументов;
- при возврате значения из функции;
- при обработке исключений.

```
Complex x = 2;

Complex y = Complex(2, 0);
```

- Функция-член класса *X* с именем ~X называется **деструктором**.
- Она используется для разрушения значения класса *X* непосредственно перед разрушением содержащего его объекта.
- Деструктор не имеет параметров и возвращаемого типа, нельзя задавать даже `void`.

```
class X
{ private:
    int n;
  public:
    X();
    ~X();
 };

X xx;
xx.~X();
```

- Конструктор с одним параметром задаёт преобразование типа своего параметра к типу своего класса.

```
class X
{ private:
    int x;
  public:
    X(int n);
  ...
 };

X::X(int n) { x = n; }

X a = 1;    // X a = X(1)
```

```
class Str
 { private:
     char *str;
   public:
     Str(int n)
       { str = new char [n]; *str = 0; }
     Str(const char *p)
       { str = new char [strlen(p) + 1];
         strcpy(str, p); }
     ~Str() { if (str) delete str; }
  };

Str s = 'a';      // int('a')
```

```
class Str
 { private:
     char *str;
   public:
     explicit Str(int n)
       { str = new char [n]; *str = 0; }
     Str(const char *p)
       { str = new char [strlen(p) + 1];
         strcpy(str, p); }
     ~Str() { if (str) delete [] str; }
  };

Str s1 = 'a';
Str s2(10);
```

- Функция-член класса *X*, имя которой имеет вид **`operator`** `<`*`имя типа`*`>`, определяет преобразование из *X* в тип, заданный *именем типа*.
- Такие функции называются *преобразующими функциями* или *функциями приведения*.

```
class X
{ private:
    int x;
  public:
    X(int n)        { x = n; }
    operator int() { return x; }
  ...
 };

int a;
X b(0);
a = (int)b;
a = b;
```

```
class X { … X(int); X(char*); … };
class Y { … Y(int); … };
class Z { … Z(X); …   };

X f(X);
Y f(Y);
Z g(Z);

void main()
 { f(1);
   f(X(1));
   f(Y(1));

   g("Mask");
   g(X("Mask"));
   g(Z("Mask"));
  }
```

```
class XX { XX(int); };

void h(double);
void h(XX);

void main()
 { h(1); }
```

```
class Stack
 { public:
    Stack();       // Конструктор
    ~Stack();      // Деструктор
    int  Push(int n);   // Добавление элемента в стек
    int  Pop();       // Выбор элемента из стека
    int  IsEmpty() const;  // Проверка, пуст ли стек
    int  IsError() const;  // Проверка, была ли ошибка
    const char* LastError() const;
  };
```

```
#include <cstdio>

class Stack
 { private:
    enum { SIZE = 100 };
    enum { NO_ERROR, STACK_EMPTY, STACK_FULL };
    int stack[SIZE];
    int *cur;
    int error;
   public:
    Stack();
    ~Stack();
    int  Push(int n);
    int  Pop();
    int  IsEmpty() const;
    int  IsError() const;
    const char* LastError() const;
  };
```

```
Stack::Stack()
{ cur = stack; error = NO_ERROR; }

Stack::~Stack()
{ }

int Stack::Push(int n)
{ if (cur - stack < SIZE)
   { *cur++ = n; error = NO_ERROR; return 1; }
  else
   { error = STACK_FULL; return 0; }
 }

int Stack::Pop()
{ if (cur != stack)
   { error = NO_ERROR; return *--cur; }
  else
   { error = STACK_EMPTY; return 0; }
 }
```

```
inline int Stack::IsEmpty() const
{ return cur == stack; }

inline int Stack::IsError() const
{ return error != NO_ERROR; }

const char* Stack::LastError() const
{ if (error == NO_ERROR)
   return "There is no error";
  else if (error == STACK_EMPTY)
   return "Stack is empty";
  else
   return "Stack is full";
 }
```

```
void main()
 { Stack s;

   s.Push(1);
   s.Push(2);
   s.Push(3);
   while (!s.IsEmpty())
    printf("%d\n", s.Pop());
   printf("%d\n", s.Pop());
   printf("%s\n", s.LastError());
   s.Push(4);
   s.Push(5);
   s.Push(6);
   s.Push(7);
   if (s.IsError())
    printf("%s\n", s.LastError());
  }
```

```
#include <cstdio>
#include <malloc.h>

class Stack
 { private:
    enum { SIZE = 100 };
    enum { NO_ERROR, STACK_EMPTY, NOT_ENOUGH_MEMORY };
    int size;
    int *stack;
    int *cur;
    int error;
   public:
    Stack();
    Stack(const Stack& s);    // Конструктор копирования
    ~Stack();
    int  Push(int n);
    int  Pop();
    int  IsEmpty() const;
    int  IsError() const;
    const char* LastError() const;
  };
```

```
Stack::Stack()
{ size  = SIZE;
  if (stack = (int *)malloc(size * sizeof(int)))
   { cur = stack;
     error = NO_ERROR;
    }
  else
   { error = NOT_ENOUGH_MEMORY;
     size = 0;
    }
 }
```

```
Stack::Stack(const Stack& s)
{ size  = s.size;
  stack = NULL;
  if (size)
   if ((stack = (int *)malloc(size * sizeof(int))) == NULL)
    { error = NOT_ENOUGH_MEMORY;
      size = 0;
     }
   else
    for (int i = 0; i < size; i++)
     *(stack + i) = *(s.stack + i);
  cur = stack;
 }

Stack::~Stack()
{ if (stack) free(stack); }
```

```
int Stack::Push(int n)
{ if (!stack) return 0;
  if (cur - stack < size)
   { *cur++ = n; error = NO_ERROR; return 1; }
  else
   if (stack = (int *)realloc(stack,
                              (size + SIZE) * sizeof(int)))
    { cur = stack + size;
      size += SIZE;
      *cur++ = n;
      error = NO_ERROR;
      return 1;
     }
   else
    { error = NOT_ENOUGH_MEMORY; size = 0; return 0; }
 }
```

```
int Stack::Pop()
{ if (cur != stack)
   { error = NO_ERROR; return *--cur; }
  else
   { error = STACK_EMPTY; return 0; }
 }

inline int Stack::IsEmpty() const
{ return cur == stack; }

inline int Stack::IsError() const
{ return error != NO_ERROR; }

const char* Stack::LastError() const
{ if (error == NO_ERROR)
   return "There is no error");
  else if (error == STACK_EMPTY)
   return "Stack is empty");
  else
   return "There is not enough memory");
 }
```

```
void main()
 { Stack s;

   s.Push(1);
   s.Push(2);
   s.Push(3);
   while (!s.IsEmpty())
    printf("%d\n", s.Pop());
   printf("%d\n", s.Pop());
   printf("%s\n", s.LastError());
   s.Push(4);
   s.Push(5);
   s.Push(6);
   s.Push(7);
   if (s.IsError())
    printf("%s\n", s.LastError());
  }
```