Тема: Инкапсуляция в классах **C++**

- Достоинства инкапсуляции при программировании классов
- Правила умолчания в **C++** и проблемы инкапсуляции
- Конструкторы и деструкторы
- Инициализация и присваивание
- Свойства – данные типа класса

Инкапсуляция при программировании классов

- Инкапсуляция данных при программировании классов – доступ к свойствам через методы
- Достоинства инкапсуляции
  - позволяет вводить инвариант представления для типа данных (упрощение разработки)
  - уменьшает зависимость кода пользователей класса от реализации методов и облегчает корректировку кода методов (сопровождение)
- Требуются языковые средства, обеспечивающие полный контроль доступа

Правила умолчания в C++

- Присваивание объектов одного и того же типа (побитовое копирование памяти)
- Создание объекта (отведение под структуру данных неинициализированной памяти)
- Разрушение объекта (освобождение памяти, распределенной под структуру данных)
- Инициализация объекта объектом того же типа (побитовое копирование памяти)

Вывод. *Эти предопределенные в **C**++ операции препятствуют принципу инкапсуляции свойств*

Проблема инварианта представления

- Инвариант представления для строки типа ***String*** – наличие терминального байта – не выполнен после создания объекта

Пример

```
#include "mystring.h"
#include <stdio.h>
void fill_array( char* );
void main( int, char** ) {
  String  s, t ;
  char very_big_text[1000] ;
  fill_array(very_big_text);
  for( int i = 0 ; i < s.length() ; i++ )    s[i] = very_big_text[i] ;
  t = s ;
}
```

Проблемы инкапсуляции

- Для классов с динамическим управлением памятью предопределенные в **C++** операции будут вызывать проблемы

Пример

```
class String {
  public:
    void setString(char* ) ;
    void freeString( ) ;
  private:
      char* ps;
};
void String::freeString(void)
 { delete ps ; }
```

```
void String::setString(char* s) {
      ps=new char[strlen(s)+1];
      if( ps == NULL ) exit(1);
      else   strcpy(ps, s);
 }
void main ( int, char* [ ] ) {
  String a;
  a.setString("Hello");
  String b, c=a;
  b = a;     a.freeStrinf();
  b.freeString();  c.freeString();
}
```

Проблема снимается

- перегрузкой операции присваивания для аргумента типа класса

  typename& **operator=(const** *typename*& **)**

- ***конструкторами*** класса (создание и инициализация объектов, передача аргументов и возврат значений)
- ***деструктором*** класса (разрушение объекта)

```
const int max_string_length = 128;
class String {
    public:
     String( ) ;     // конструктор по умолчанию
     ~String( ) ;   // деструктор
     String& operator= ( const String& );
     String& operator= ( const char* );
     int  length( ) const ;
     int  read( );              // чтение из stdin
     void  print( ) const; // вывод в  stdout
     char& operator[ ] ( int );
     const char& operator[ ] ( int ) const;
     String substring (int start, int len) const;
     friend String operator+ (const String&, const String& );
     friend String operator+ (const String&, const char* );
     friend String operator+ (const char*, const String& );
    private:
     char text [ max_string_length+1 ];
};
```

Определение присваивания

```
#include "mystring.h"
#include <string.h>
  // использование:  s = "Hello World"
String& String::operator=(const char* s ) {
    strncpy( text, s, max_string_length );
    text [max_string_length] = '\0';
    return *this;
 }
  // использование:  s = t
String& String::operator=(const String& s ) {
    strcpy( text, s.text );  // или *this = s.text;
    return *this;
 }
```

Конструктор по умолчанию и пустой деструктор

```
#include "mystring.h"

// Конструктор по умолчанию обеспечивает инвариант
// представления при простом описании переменной
String::String( ) {
  text[0] = '\0' ;
}
// Теперь для переменной, описанной как  String s;
// s.length() вернет 0

// Деструктору ничего не надо делать (его м. опустить)
String::~String( ) { }
```

Конструкторы

- Конструктор – особая функция-член класса
  - имя совпадает с именем класса
  - не возвращает значения
  - неявно вызывается при создании объекта
  - вызов происходит сразу после выделения память под структуру $\Rightarrow$ для ряда объектов (глобальные, внешние статические) конструктор вызывается до того, как функция **main** получит управление
- Основное назначение конструкторов – это обеспечить корректность объекта при его создании и инициализации (в т.ч. при передаче аргументов и возврате значений)

Деструктор

- Деструктор – особая функция-член класса
  - имя образуется из тильды “~” и имени класса
  - не возвращает значения
  - не имеет аргументов
  - неявно вызывается при разрушении объекта
  - вызов происходит непосредственно перед освобождением памяти под структуру $\Rightarrow$ для ряда объектов (глобальные, статические) деструктор вызывается после того, как управление получит функция **main**
- Основное назначение деструктора – это обеспечить корректное удаление объекта при завершении его времени жизни

Вызов конструкторов и деструкторов

```
#include "mystring.h"
#include <stdio.h>
String s_extrn ;
static String s_stat ;
void example( ) {
    printf( "Entering the example() function.\n" );
    String s_auto ;
    static String s_loc_stat ;
    String s_array[30] ;
    printf( "Leaving the example() function. \n" );
}
void main( int, char** ) {
    printf( "Start of the program. \n" );
    example( ) ;
    example( ) ;
    printf( "End of the program. \n" );
}
```

Инициализация

- Отличие инициализации и присваивания

Пример int *i* = 7; // инициализация

```
int k;
k = 75000;     // присваивание
i = 0;         // присваивание
```

- Два синтаксических способа инициализации

Пример

```
# include “complex.h”
complex c = 7.4;   // инициализация 7.4+0i
complex d (8.5);   // инициализация 8.5+0i
complex e (9.1,10.0);  // инициализация 9.1+10.0i
complex f ;
```

Конструкторы и инициализация

- Инициализацию обеспечивает конструктор

Пример Класс **complex** должен иметь такие конструкторы:

```
complex ( double );              // для инициализации 1 и 2
complex (double, double );  // для инициализации 3
complex ( ) ;                       // для описания без инициализации
```

или ***один*** конструктор вида

```
complex (double=0, double=0 );  // для всех описаний
```

- Поместив соответствующий конструктор в закрытую часть класса можно запретить конкретный синтаксис описания

Пример для класса ***String***

```
#include "mystring.h"
#include <stdio.h>
void main ( int, char*[ ] ) {
    String  output ;
    // Здесь требуется String::String( ) или отсутствие конструктора
    String  firstname = "Bilbo" ;
    String  middlename("J.") ;
    // В этих случаях требуется String::String( const char* )
    String  lastname = firstname ;
    String  name( firstname +" "+middlename+" "+lastname);
    // Здесь требуется String::String( const String& )
    // или отсутствие конструктора вообще
    output = "Name is: " + name ;      // operator=(const String&)
    output.print() ;
}
```

Особенности конструкторов и деструкторов

- Класс может не иметь конструктора. В этом случае создание объекта состоит в выделении памяти под структуру
- Класс может не иметь деструктора. В этом случае разрушение объекта состоит в освобождении памяти, выделенной под структуру
- Класс может иметь несколько конструкторов. Если есть хотя бы один, то создание объекта требует конструктора
- Класс может иметь только один деструктор.

Особенности конструкторов и деструкторов

- Конструктор без аргументов называется ***конструктором по умолчанию***
- Конструктор с аргументом типа класса называется ***копирующим конструктором***
- Конструктор с одним аргументом обеспечивает преобразование типа
- Конструктор может вызываться явно
- Конструктор, имеющий квалификатор **explicit** может быть вызван *только явно*

Расширение класса *String*

```
const int max_string_length = 128;
class String {
    public:
      String() ;
      String( const char* ) ;
      String( const String& ) ;
      ~String() ;
      String& operator= ( const char* );
      int  length( ) const ;
      int  read( );                  // чтение из stdin
      void  print( ) const;// вывод в  stdout
      char& operator[ ] ( int );
      const char& operator[ ] ( int ) const;
      String substring (int start, int len) const;
      friend String operator+ (const String&, const String& );
      friend String operator+ (const String&, const char* );
      friend String operator+ (const char*, const String& );
    private:
      char text [ max_string_length+1 ];
};
```

Дополнительные конструкторы

```
#include "mystring.h"
#include <string.h>
// Конструктор с параметром
String::String( const char* s) {
  strncpy( text, s, max_string_length ) ;
  text [ max_string_length ] = '\0'
}
// Копирующий конструктор
String::String( const String&  s) {
  strcpy( text, s.text ) ;
  text [ max_string_length ] = '\0'
}
```

Свойства – данные типа класса

- Класс может иметь члены-данные типа класса.

```
#include "mystring.h"
class Employee {
    public:
      void  set_name( const String& );
      String get_name( ) const ;
      void  set_salary( float );
      float get_salary( ) const ;
      // ..............
    private:
      String name;
      float salary ;
};
```

- Какие у него права относительно таких данных?

Создание объекта типа *Employee*

- Создание объекта типа *Employee* включает в себя создание свойства *name* типа *String*
- Класс *String* имеет конструктор, который должен вызываться при создании объектов типа *String*
- Следовательно, объект типа *Employee* создается в такой последовательности:
  - выделение памяти под структуру в порядке описания полей
  - после выделения памяти свойству (например, *name*) вызывается конструктор, если он есть (*String::String()* для *name*)
  - вызывается конструктор класса-владельца, если он есть

Присваивание объекта типа *Employee*

- Если у класса нет операции присваивания, то присваивание объектов типа *Employee* происходит по правилам умолчания, т.е. копируется содержимое памяти
- **НО!** Если свойства – данные типа класса, имеющего операцию присваивания, то для них будет использоваться эта операция присваивания (*String::***operator=**(**const** *String&*) для *name*)

Инициализация объекта типа *Employee*

- Если у класса нет копирующего конструктора, то инициализация объектов типа *Employee* происходит по правилам умолчания, т.е. копируется содержимое памяти
- **НО!** Если свойства – данные типа класса, имеющего копирующий конструктор, то он будет использован для инициализации этих свойств (*String::String*(**const** *String*&) для *name*)

Разрушение объекта типа *Employee*

- Разрушение объекта типа *Employee* включает в себя разрушение свойства *name* типа *String*
- Класс *String* имеет деструктор, который должен вызываться при разрушении объектов типа *String*
- Следовательно, объект типа *Employee* разрушается такой последовательности:
  - вызывается деструктор класса-владельца (*Employee*), если он есть
  - освобождается память, отведенная под структуру в порядке, обратном описанию полей
  - **до** освобождением памяти для свойства (например, *name*) вызывается деструктор, если он есть (*String::~String()* для *name*)

Пример использования

```
#include "employee.h"
int main ( int, char*[ ] ) {
    String  jones = "Jones" ;
    Employee e1, e2 ;   //  String::String( ) применяется к
        //  e1.name и e2.name
    e1.set_name( jones ) ;
    e1.set_salary( 3000 ) ;
    e2 = e1 ;      //  String::operator=( const String & ) используется
            //  для присваивания e2.name значения e2.name
    Employee e3=e2 ;   //  String::String(const String&)
        //  применяется для инициализации                    //
    e3.name значением e2.name
    return 0; //  String::~String( ) применяется к  e1.name,
    //  e2.name и  e3.name
}
```

Подмена встроенных операций

- Класс-владелец может иметь свои
  - операцию присваивания
  - конструктор по умолчанию
  - копирующий конструктор
  - конструктор с параметрами
  - деструктор

Подмена встроенных операций

```
#include "mystring.h"
class Employee {
    public:
     Employee( );
     Employee( const Employee& );
     Employee( const String& , float );
     ~Employee( );
     Employee& operator=(const Employee&);
     void  set_name( const String& );
     String get_name( ) const ;
     void  set_salary( float );
     float get_salary( ) const ;
     // ..............
    private:
     String name;
     float salary ;
};
```

Определение операций

```
#include "Employee.h"

Employee::Employee( ) { }

Employee::Employee( const Employee& e )
      : name(e.name), salary(e.salary)  { }

Employee::Employee( const String& n, float f)
      : name(n), salary(f)  { }

Employee::~Employee( ) { }

Employee::Employee& operator=(const Employee& e) {
    name = e.name;
    salary = e.salary;
    return *this ;
}
```

Определите семантику вызовов

```
#include "employee.h"

int main ( int, char*[ ] ) {
    String  jones = "Jones" ;
    Employee e1( jones , 3000 ) ;     // ?
    Employee e2 ;                     // ?
    e2 = e1 ;                         // ?
    Employee e3=e2 ;                  // ?
    return 0;                         // ?
}
```

Ответ

```
#include "employee.h"
int main ( int, char*[ ] ) {
    String  jones = "Jones" ;

    Employee e1( jones , 3000 ) ;   // String::String(const String& )
                // и Employee::Employee(const String&, float)

    Employee e2 ;       //  String::String( ) и Employee::Employee( )

    e2 = e1 ;     // Employee::operator=( const Employee & )

   Employee e3=e2 ;   // String::String(const String&) и
       // Employee::Employee(const Employee&)

    return 0; // Employee::~Employee( ) и затем String::~String( )
}
```

РЕЗЮМЕ

- Язык **C++** имеет все средства обеспечения при программировании класса полного контроля над свойствами объекта
- Для этого используются
  - методы класса
  - функции-друзья класса
- А также средства корректировки разрешенных в языке **C** по умолчанию операций
  - присваивания: *String*& *String*::**operator=(const** *String*&**)**
  - создания: *String*:: *String*( )
  - инициализации: *String*:: *String*(**const** *String*&)
  - уничтожения: *String*:: ~*String*( )
- Эти средства корректно работают с членами-данными типа класса

Упражнение

- Добавьте в конструкторы, операцию присваивания и деструктор класса *String* вызов функции printf, сообщающий, какой метод вызывается.
- Поэкспериментируйте с тестовыми программами, обращая внимание на вызов методов - когда, какие и в какой последовательности.