ПРОГРАММИРОВАНИЕ II

Емельянов Павел Геннадьевич
к.ф.-м.н., с.н.с. ИСИ СО РАН
emelianov@iis.nsk.su

Общий план курса

- Объектно-ориентированное программирование (на примере основных концепций языка C++)
- Объектно-ориентированное проектирование.
- Модели данных и базы данных (на примере XML и *реляционных МД и БД*)
- Организация взаимодействия между компьютером и человеком (на примере программных графических интерфейсов в среде Windows)

Часть I. Основы языка C++

- Концепция класса. Видимость членов. Конструкторы/деструкторы.
- Статические члены и методы.
- Перегрузка имен. Переопределение операторов.
- Друзья класса.
- Области видимости и пространства имен.
- Исключения.
- Шаблоны.
- Наследование.
- STL.

Литература

1. Б. Страуструп. Язык программирования C++. Третье издание: Пер. с англ. – М: М: ДМК Пресс; СПб: Питер, 1999. 991 с.
2. Б. Страуструп. Дизайн и эволюция C++. Пер. с англ. – М: ДМК Пресс; СПб: Питер, 2006. 448 с.
3. Н. Джосьютис. C++ Стандартная библиотека. Для профессионалов. Пер. с англ. – СПб: Питер, 2004. 730с.
4. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++,

   Второе издание: Пер. с англ. – М: Бином, 1999. 560 с.
5. Курсы Интернет университета информационных технологий. http://www.intuit.ru/.

Программа на С

файл myStack.h

```
typedef struct {
  int  sz;
  int  sp;
  int* bf;
} Stack;

void init_stack(Stack*, int);
void deinit_stack(Stack*);

int  empty(Stack*);
void push(Stack*,int);
int  pop(Stack*);
```

файл myStack.c

```
#include "myStack.h"

void init_stack(Stack* st, int size) {
  st->sp=0;
  st->sz=size;
  st->bf=(int*)malloc(size*sizeof(int));
}

void deinit_stack(Stack* st) {
  free(st->bf);
}

int empty(Stack* st) {
  return st->sp==0;
}

void push(Stack* st,int x) {
  if (st->sp==st->sz) {
    printf("error: stack overflow");
    return;
  }
  st->bf[st->sp++]=x;
}

int pop(Stack* st) {
  if (empty(st)) {
    printf("error: stack underflow");
    return 0;
  }
  return st->bf[--st->sp];
}
```

файл prog.c

```
#include "myStack.h"

void main()
{
  Stack st;
  init_stack(&st,100);
  push(&st,5);
  . . .
  while (! empty(&st))
  {
    int x=pop(&st);
    . . .
    push(&st,x*x);
    . . .
  }
  deinit_stack(&st);
}
```

Задачи, которые хотелось бы решить при создании нового языка

- Удобное моделирование математических концепций (в C пользовательские типы не ведут себя как стандартные, конструирование новых типов из существующих, ...).
- Повышение надежности создаваемых программных систем (в C слаб даже статический типовой контроль, неудобные средства обработки исключительных ситуаций, ...).
- Разработка больших программных систем (использование старого кода, переиспользование существующих программных компонент, ...).

Краткая история C++

- Язык C – 1970 год (действующий ISO стандарт – 1990 год).
- Начало 80-х – C с классами.
- Simula 67, Clu, Smalltalk, Ada, Algol 68, Modula-2, ML.
- Первая версия языка C++ – 1983 год (действующий ISO стандарт – 1998 год).
  Managed Extensions for C++ –> Ecma C++/CLI.

Что является C и не является C++

- Отсутствие типа поразумевает int.
- Устаревшее описание параметров функций и процедур.
- В C++ появились новые ключевые слова.
- Макросы C – ключевые слова C++.
- В C возможен goto в обход инициализации.
- В C в инициализаторе массива может быть больше элементов:

```
char a[3]="abc";
```

Новые возможности C++

- Константы, инициализаторы в операторах, новое в приведении типов, параметры по умолчанию, операторы распределения динамической памяти.
- Концепция класса, контроль доступа, виды хранения, конструкторы/деструкторы.
- Производные классы. Виртуальность/абстрактные классы.
- Перегрузка операторов.
- Шаблоны. Стандартная библиотека.
- Обработка исключений.
- Доступ к типовой информации во время исполнения.

Логический тип bool

```
int  x=0;
bool b = x==1;//значения типа bool: true/false
int  y=int(b);//y имеет значение 0
x=int(y == 0);//x имеет значение 1
b=123;        //b имеет значение true
```

Константы

```
#define  PI  3.14159      //стиль C
const float PI = 3.14159;
const int v[]={1,2,3};    //массив констант


void f(const int* n)
{
   *n=123;   //ошибка: менять *n нельзя
}
…
int x;       //переменная
…
f(&x);       //гарантируется, что значение не
      //изменится
…
```

Указатели и константы

```
void Proc(char * p, const char * q)
{
    char s[]="abc";

    const char * pc=s;          //указатель на константу
    pc[1]='z';                  //ошибка
    pc=p;

    char * const cp=s;          //константный указатель
    cp[1]='z';
    cp=p;                       //ошибка

    const char * const cpc=s;  //константный указатель на
                        //константу
    cpc[1]='z';                 //ошибка
    cpc=p;                      //ошибка

   *q='z';                     //ошибка: нельзя изменить *q
}
```

Ссылки: пример из Паскаля

```
procedure Inc(var x:integer)
begin
   x:=x+1;
end;
…
Inc(y);
…
```

```
procedure Inc(x:^integer)
begin
   x^:=x^+1;
end;
…
Inc(@y);
…
```

Ссылки в C++

```
//что можно в C
void Inc(int* x)
{
    *x++;
}
…
Inc(&y);
…
```

```
//уточнение в C++
void Inc(int * const x)
{
    (*x)++;
}
…
Inc(&y);
…
```

```
//новое в C++
void Inc(int& x)
{
    x++;
}
…
Inc(y);
…
```

Ссылки как альтернативные имена для объекта:

```
int  x=123;
int& r=x;   //ссылка обязательно должна быть инициализирована
int  y=r;   //значение y равно 123
r++;        //значение x и r равно 124, значение y не изменлось
```

Концепция класса

```
typedef struct {
   double re;
   double im;
} _Complex;
```

```
class Complex {
   double re;
   double im;
};
```

```
class _Complex {
public:
   double re;
   double im;
};
```

```
class Complex {
private:
   double re;
   double im;
public:
};
```

```
_Complex s; s.re+=s.im; //правильно: оба члена открыты

Complex  c; c.re+=c.im; //ошибка: члены класса закрыты
```

Классы и объекты: реализация стека

```
class Stack {
    int   sz;
    int   sp;
    int*  bf;
public:
 //объединяет варианты Stack() и Stack(int)
    Stack(int=100);
    ~Stack() { delete[] bf; }
    bool empty() const {
        return this->sp==0;
    }
   void push(const int x) {
        bf[sp++]=x;
    }
   int pop();
};

inline Stack::Stack(int size) {
    sp=0;
    bf=new int[sz=size];
}

inline int Stack::pop() {
     return this->bf[--this->sp];
}
```

```
#include "Stack.h"

void main()
{
    Stack st100, st(5000);
    …
    st100.push(123);
    …
    while (!st100.empty()) {
        …
       st.push(st100.pop());
        …
    }
}
```

Конструирование объектов – I

- Автоматическое создание объекта при входе в блок.
- Создание объекта в динамической памяти:

```
Stack* st100=new Stack();
Stack* st=new Stack(5000);
```

- Объект, являющийся элементом массива:

```
Stack  st[25];
Stack* st=new Stack[25];
```

ВАЖНО: Для всех элементов массива будет вызван один и тот же конструктор без параметров!!!

Конструирование объектов – II

- Статические объекты:
  - Глобальные в программе:

```
Stack stk;  //создается при запуске программы до вызова main
int main() {
   return stk.pop(stk.push(123));
   }
```

  - Локальные в программе:

```
void func(int n) {
   static Stack st0;    //создается при первом вызове func
      if (n==1) {
      static Stack st1;//создается при первом вызове func с
n=1
      …
   }
   …
   }
```

  - Локальные в классе:

```
class myClass {
static Stack st0; //создается при запуске программы до вызова main
…
}
```

Конструирование объектов – III

- Создание нестатического объекта, являющегося членом класса:

```
class newClass {
 Stack  stk;
 int    num;
public:
     newClass(int n=20) : stk(n), num(n) { stk.push(num); }
 }
```

- Создание временного объекта при вычислении выражения:

```
class Complex {
 double re;
 double im;
public:
 Complex(double r=0.0, double i=0.0) : re(r), im(i) {}
 Complex operator+(Complex x) { return Complex(re+x.re,im+x.im); }
};
…
Complex  a(1,2),  b(2,1),  c=a+b+Complex(1,1);
…
```

Конструктор по умолчанию

- Конструктор без аргументов называется конструктором по умолчанию.
- Конструктор, для всех аргументов которого заданы умолчательные значения, является конструктором по умолчанию.
- Если конструктор по умолчанию не задан для класса явно, то он сгенерируется компилятором автоматически и будет вызываться неявно всякий раз, когда это требуется. При этом для членов классов будут вызываться конструкторы по умолчанию.
- Класс, содержащий в качестве членов константы и ссылки, обязан содержать определение конструктора по умолчанию:

```
class SomeClass {
 const int  val;
 int&       ref;
public:
  SomeClass(int v=100, int& r) : val(v), ref(r) {}
…
}
```

Копирующий конструктор

- Конструктор, единственным аргументом которого является объект того же самого типа, называется копирующим конструктором.

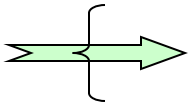
```
//ВАЖНО: параметр – ссылка, иначе бесконечный цикл
Stack::Stack(const Stack& stk) : sz(stk.sz), sp(stk.sp) {
 //действия по копированию массива bf
}
```

- Случаи, когда объект копируется: инициализация, передача аргумента функции, возвращение значения функции и исключения.
- Если копирующий конструктор не задан для класса явно, то он сгенерируется компилятором автоматически и будет вызываться неявно всякий раз, когда это требуется. Смысл – почленное копирование. «Ручная реализация»:

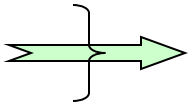
```
class Complex {
 double  re;
 double  im;
public:
 Complex(const Complex& c) : re(c.re), im(c.im) {}
…
}
```

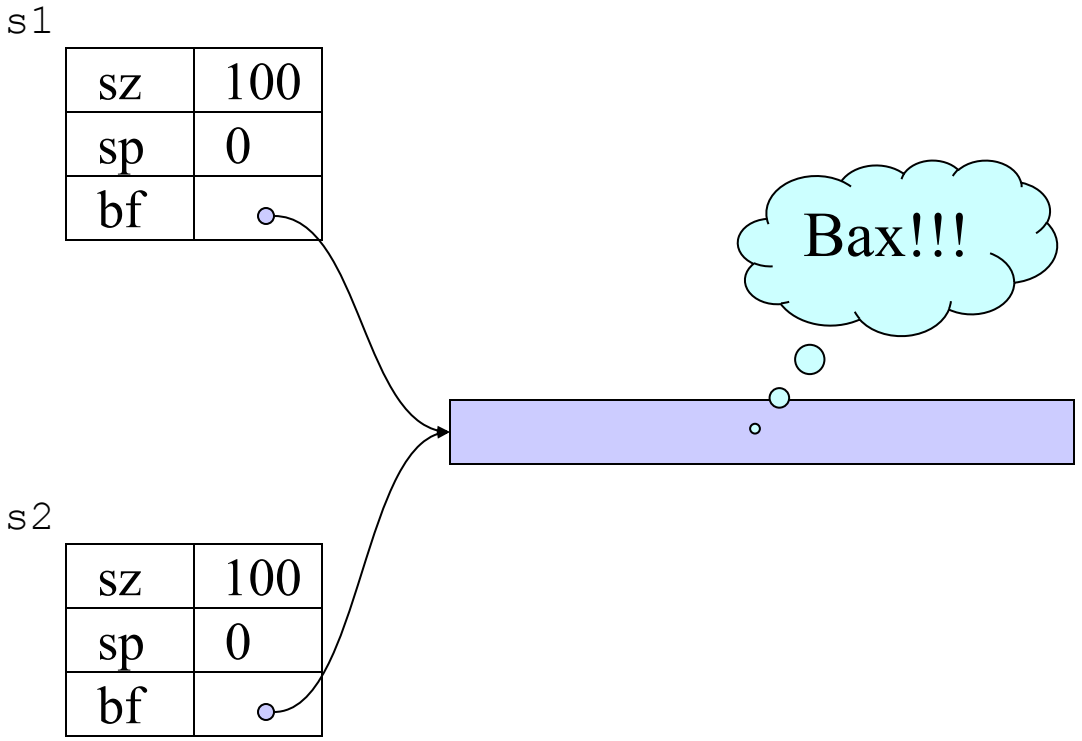
- ВАЖНО: отличия копирующего конструктора и оператора присваивания

“Плохой” копирующий конструктор



```
Stack  s1;
Stack  s2=s1;

...
```





Уничтожение объектов

- Деструктор вызывается:

  – для автоматических объектов – когда управление выходит из блока, где они описаны;

  – для статических объектов - когда программа «завершает свое исполнение»;

  в порядке, обратном вызову конструкторов.
- Обычно деструктор вызывается неявно. Явный вызов деструктора

```
T* po=new T(); po->~T();
```

  встречается редко (например, переопределение оператора `delete`).

Статические члены и методы класса

```
class AAA {
    static const int BOUND=100;//статич. константа интегрального типа
   static int       count;
   int              val;
public:
   AAA(int n=1) : val(n) { ++count; }
   ~AAA() { --count; }
   void proc() {
      if (count>BOUND) val*=2; //имеется доступ ко всем членам
   }
   static int func() {
      val++;          //ошибка: доступ к нестатическому члену
      return count; //правильно: статический метод имеет доступ
                    //           только к статическим членам
   }
}
…
int AAA::count=0;
…
AAA x;  x.proc();
if (AAA::func()>1000)
   cout<<"too many objects!";
```

this->

Объявление `mutable`

логическое и физическое постоянство

```
class SomeClass {
 mutable char *cache;
          …
public:
    SomeClass() : cache(0) { /*инициализация других членов*/ }
    ~SomeClass() {
       delete[] cache;
       /*деинициализация других членов*/
    }
    …
    char* toString() const {
       if (! cache) {  //создать строковое представление объекта
          SomeClass* thisObject=const_cast<SomeClass*>(this);
          thisObject->cache=new char[100];
          …
       }
       return cache;
    }
}

const SomeClass constObject;
… constObject.toString() …          //проблема!!!
```

Перегрузка имен функций – I

```
int abs(int);                          int    abs(int);
long labs(long);                       long   abs(long);
double fabs(double);                   double abs(double);

void print(int);                 void print(long);
void print(float);               void print(double);
void print(char);                void print(char*);
…
    print(1);    //ошибка: print(int) или print(long) или
                 //        print(float) или print(double)?
    print(1.0);  //ошибка: print(float) или print(double)?
```

Явное приведение параметров, определяющее выбор:

```
print(int(1));
print(long(1));  //print(1L); print(static_cast<long>(1));
```

Перегрузка имен функций – II

Процедура поиска (процедура разрешения неоднозначности) соответствия вызова определению функции основывается на проверке набора критериев в следующем порядке:

- Точное соответствие типов, включая следующие случаи:
  - имя массива и указатель

```
void print(char*);
…
char a[]="abc";
print(a);
```

  - имя функции и указатель на функцию

```
void map(void(*action)());
void F() { /*действие*/ };
…
map(F);
```

  - тип и константа такого типа

```
void print(const char);
…
char c='a';
print(c);
```

Перегрузка имен функций – III

- Соответствие, достигаемое за счет продвижения в «охватывающие» типы:
  - `bool, (signed/unsigned) char, short -> int`
  - `float -> double, double -> long double.`
- Соответствие, достигаемое за счет преобразований:
  - `int -> double, double -> int, int -> unsigned int`
  - `произвольный указатель -> void*`
  - `указатели на производные типы -> указатели на их базовые типы.`
- Соответствие, достигаемое за счет преобразований, определяемых пользователем.
- Соответствие за счет (...) в объявлении функции.

Перегрузка имен функций – IV

- Результат разрешения перегрузки не зависит от порядка объявления функций.
- Тип возвращаемого значения не используется при разрешении перегрузки.
- Случай нескольких аргументов:

```
int pow(int,int);
double pow(double,double);
…
double d=pow(2.0,2);    //pow(int(2.0),2) или pow(2.0,double(2))?
```

Правило: вызывается функция, у которой наилучшим образом соответствует один аргумент и лучшим либо таким же остальные аргументы. Иначе – ошибка (неоднозначность).

Переопределение операторов – I

```
class Complex {
    double re;
    double im;
public:
    Complex(double r=0.0, double i=0.0) : re(r), im(i) {}
    Complex operator+(Complex x) {
        return Complex(re+x.re,im+x.im);
    }
    Complex operator*(Complex x);
};
…
Complex Complex::operator*(Complex x) {
    return Complex(…,…);
}

…
Complex  a(1,2),  b(2,1),  c=a+Complex(0,1)*b; //приоритет операций
   обычный
…
```

a+b сокращение для a.operator+(b)

Переопределение операторов – II

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     delete  new[]   delete[]
```

Нельзя переопределить:

- разрешение области видимости – **::**
- выбор члена – **.**
- выбор члена через указатель на член – **.***
- тернарный условный оператор – **?:**

Нельзя ввести новую лексему для оператора или изменить его местность.

Предопределенный (для объектов) смысл операторов **=**, **&** и **,** можно скрыть, сделав их закрытыми.

Переопределение операторов – III

Бинарные операторы могут определяться:

- Либо в виде нестатической функции-члена с одним аргументом:

```
class X {
 …
public:
 X(int);
 void operator+(int);
 …
}
```

- Либо в виде функции-не-члена с двумя аргументами:

```
 void operator+(X,X);
```

за исключением операторов **=, [], ->,** чтобы гарантировать, что первый аргумент изменяемое значение:

```
 operator=(operator[](1,X),123); //!!!
```

Если определены обе операторные функции, то для выбора применяются правила разрешения перегрузки:

```
 X a;
 a+1;        //a.operator+(1);
 1+a;        //operator+(X(1),a);
```

Если выбор неоднозначен, то выдается ошибка.

Переопределение операторов – IV

Унарные операторы могут определяться:

- Либо в виде нестатической функции-члена :

```
class X {
 …
public:
  void operator~();        //без аргументов
  void operator++();       //без аргументов - префиксный
  void operator++(int);    //с одним фиктивным аргументом -
постфиксный
 …
}
```

- Либо в виде функции-не-члена:

```
     void operator~(X);       //с одним аргументом
 void operator++(X);     //с одним аргументом – префиксный
 void operator++(X,int); //с двумя аргументами - постфиксный
```

Если определены обе операторные функции, то для выбора применяются правила разрешения перегрузки. Если выбор неоднозначен, то выдается ошибка.

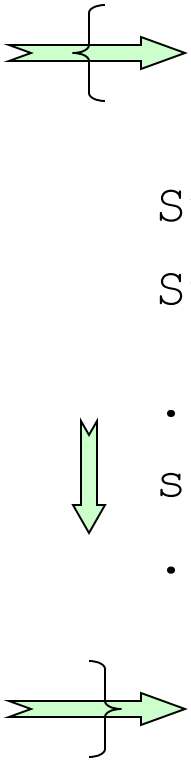
```
enum Day {sun,mon,tue,wed,thu,fri,sat};
Day& operator++(Day& d) {
    return d=(d==sat)?sun:Day(d+1);
}
```

Оператор присваивания

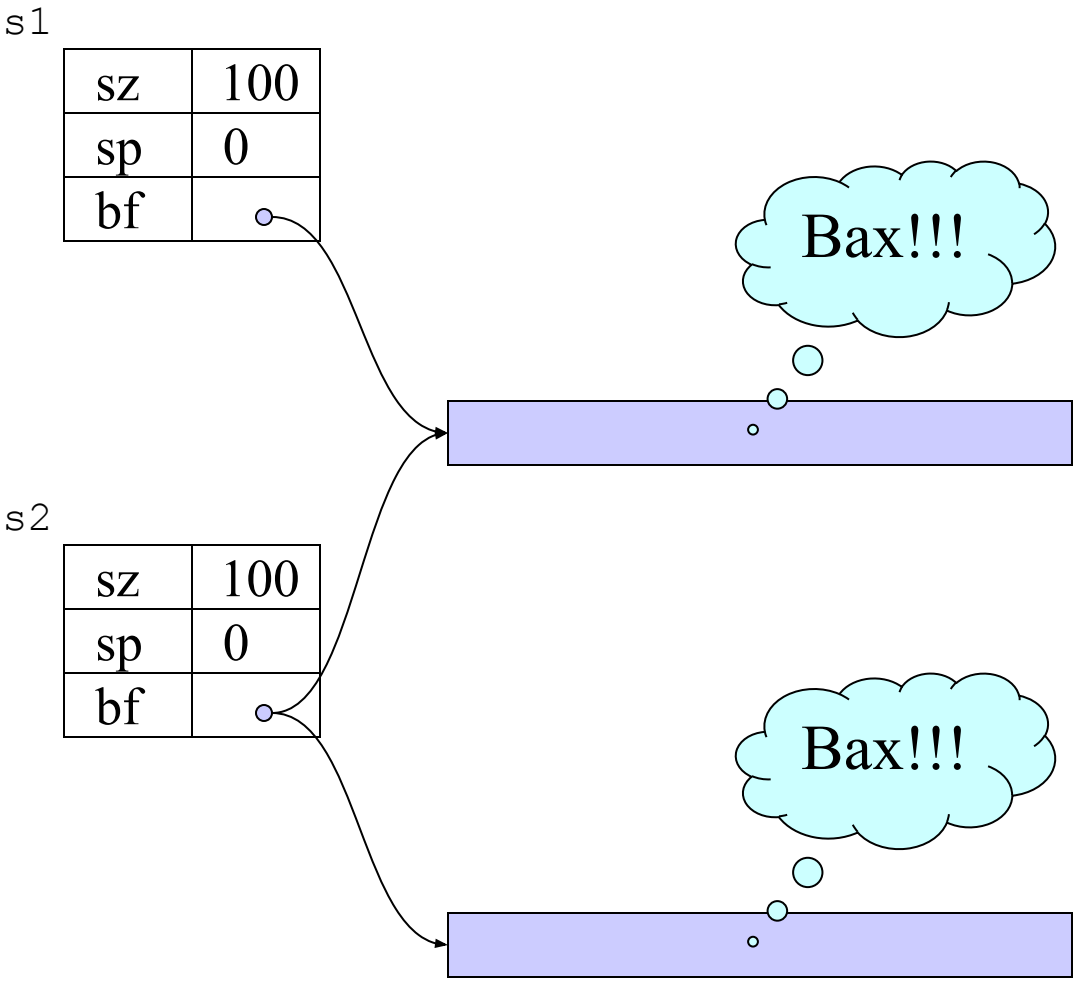
ВАЖНО: В отличии от копирующего конструктора оператор присваивания работает с уже существующим объектом:

```
class Stack{
    int   sp;
    int   sz;
   int*  bf;
public:
    …
    Stack& operator=(const Stack& stk) {
        //this->bf уже указывает на какой-то буфер. после
        //присваивания по умолчанию связь со старым буфером
        //могла бы потеряться. надо сделать что-то умное
       …
    }
}
```

“Плохой” оператор присваивания



```
Stack  s1;
Stack  s2;

...
s2=s1;
...
```



Стандартные потоки ввода/вывода

```
class istream {
…
} cin;
class ostream {
    …
} cout, cerr;

istream& operator>>(istream&, int);
ostream& operator<<(ostream&, int);


#include <iostream>
…
int n;
cin >> n;
cout << n+1 << endl;
…


istream& operator>> ( istream& inputStream, Date& date ) {
    inputStream >> date.Month >> date.Day >> date.Year;
   return inputStream;
}
```

Переопределение операторов – V

```
class Complex {
    double re, im;
public:
    Complex(double r=0.0,double i=0.0) : re(r), im(i) {}
    double real() const { return re; }
    double imag() const { return im; }
    Complex& operator+=(Complex c) {
        re+=c.re; im+=c.im;
        return *this;
    }
    Complex& operator+=(double r) {
        re+=r;
        return *this;
    }
}
//функции помощники для реализации смешанной арифметики
Complex operator+(Complex a, Complex b) {
    Complex c=a;
    return  c+=b;
}
Complex  operator+ (Complex, double);
Complex  operator+ (double, Complex);
Complex  operator+ (Complex);      //унарный плюс
Complex  operator- (Complex);      //унарный минус
bool     operator==(Complex, Complex);
bool     operator!=(Complex, Complex);
istream& operator>>(istream&, complex&);
ostream& operator>>(ostream&, complex&);
```

$Z=M_*X+Y$ и эффективность

- «Типичный» код в типичной программе по линейной алгебре:

  `… while (…) {… Z=M*X+Y; …} …`

  где X, Y, Z – векторы и M – матрица.
- При прямолинейной реализации создаются две временные переменные и осуществляются два копирования.
- Очень неэффективно.
- Широко распространенная операция:
  - The GNU Multiple Precision Arithmetic Library

    `void mpz_addmul(mpz_t rop,mpz_t op1,mpz_t op2) // rop←rop+op1×op2`

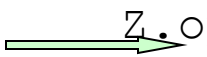
  - Операции вида **rop←rop+op1×op2** стандартизованы в IEEE Standard for Floating-Point Arithmetic (IEEE 754) 2008 года.
  - Анонсировано FMA расширение SSE компаниями Intel и AMD (2011г.).

Z=M*X+Y и отложенные вычисления – I

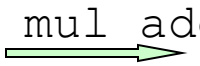
```
void mul_add_and_assign(Vector&, const Matrix&,
                        const Vector&, const Vector& );


struct MVmul {
    const Matrix& m; const Vector& v;
    MVmul(const Matrix& mm, const Vector& vv) : m(mm), v(vv) {}
    operator Vector();  //
};
inline MVmul operator*(const Matrix& mm, const Vector& vv) {
    return MVmul(mm,vv);
}

struct MVmulVadd {
    const Matrix& m; const Vector& v; const Vector& v2;
    MVmulVadd(const Matrix& mv, const Vector& vv)
                  : m(mv.m), v(mv.v), v2(vv){}
    operator Vector();
};
inline MVmulVadd operator+(const MVmul& mv, const Vector& vv) {
    return MVmulVadd(mm,vv);
}
```

Z=M*X+Y и отложенные вычисления – II

```
class Vector {
    //
public:
    //
    Vector(const MVmulVadd& mvv) {
    mul_add_and_assign(this,&mvv.m,&mvv.v,&mvv.v2);
   }
    Vector& operator=(const MVmulVadd& mvv) {
    mul_add_and_assign(this,&mvv.m,&mvv.v,&mvv.v2);
        return *this;
   }
};
…
Matrix M;
Vector X,Y,Z;
…
Z=M*X+Y;
…
```



```
Z.operator=(MVmulVadd(MVmul(M,X),Y));
```



```
mul_add_and_assign(&Z,&M,&X,&Y);
```

Друзья класса – I

Нестатическая функция-член обладает следующими свойствами:

- доступ к закрытой части класса;
- находится в области видимости класса;
- вызывается для объекта класса.

Статическая функция-член обладает только первыми двумя свойствами.

Объявление функции-члена как **`friend`** наделяет ее только первым свойством. Задачи, решаемые в данном случае:

– разделить различные классы объектов;

– предоставить эффективные средства доступа к внутреннему устройству классов, являющихся логически связанными.

Друзья класса – II

```
class Matrix;

class Vector {
     int length;
     …
public:
    …
     friend Vector operator*(const Matrix&, const Vector&);
     …
}

class Matrix {
     int numRows, numColumns;
     …
     friend Vector operator*(const Matrix&, const Vector&);
public:
     …
}

Vector operator*(const Matrix& m, const Vector& v) {
     if (m.numColumns!=v.length)
          //несоответствие размерностей. что-то надо сделать
     …
}
```

Друзья класса – III

- Друзей можно объявлять как в private-, так и в public-части описания класса.
- Отношение «дружбы» не является транзитивным.
- Можно объявить весь класс другом:

```
class A {
friend class B;
    int count;
    …
public:
    …
};
class B {
    …
public:
    B(const A& a) {
        if (a.count>0) /*что-то сделать*/
        …
    }
```

Области видимости и пространства имен

- Глобальная ОВ (на уровне всей программы).
- Локальная ОВ (внутри блока).
- ОВ класса.
- ОВ пространства имен.

```
namespace UsefulThings {
    class Stack;
    class Queue {
      …
      Queue(const Stack& s) { /* использовать s */ }
      …
    };
    …
}
…
class UsefulThings::Stack {
    …
};
…
UsefulThings::Queue* qe=new UsefulThings::Queue();
```

Пространства имен – I

- Синонимы пространства имен:

```
namespace UT=UsefulThings;
…
UT::Stack sp=UT::Stack(128);
```

- «Глобализация» всех описаний из пространства имен:

```
using namespace UsefulThings;
…
Stack sp=Stack(10);
```

- «Глобализация» одного описания из пространства имен:

```
using namespace UsefulThings::Stack;
…
Stack sp=Stack(10); //правильно
Queue qe;           //ошибка: требуется квалификатор
                    //        пространства имен
```

Пространства имен – II

- Разрешение области видимости:

```
namespace ZZZ { int Var; }
int Var;
void main() {
    using namespace ZZZ;
    Var++;       //ошибка: глобальная или из ПИ ZZZ
    ::Var++;     //правильно: глобальная
    ZZZ::Var++;  //правильно: из пространства имен ZZZ
}
```

- Вложенные пространства имен:

```
namespace X { namespace Y { int Var; } }

… X::Y::Var++; …
```

- Пространства имен являются открытыми:

```
namespace ZZZ { int AAA; }
…
namespace ZZZ { int BBB; }
```

Пространства имен – III

- Неименованные пространства имен:

```
namespace { int Var; } //определение
… Var++; … //находится в той же единице трансляции, что и определение
```

- Стандартное пространство имен:

```
//файл stdio.h
namespace std {
    …
    int printf(char* fmt, ...);
    …
}
using namespace std;

//файл myProg.c
#include <stdio.h>
…
printf("Hello!");
…
```

```
#include <cstdio>
…
std::printf("Hello!");
…
```

Исключения – I

При возникновении ошибок, которые невозможно обработать в месте их возникновения, нужно:

- сгенерировать информацию об ошибке;
- перехватить эту информацию в том месте, где это важно;
- обработать эту ошибку или передать дальше в случае невозможности восстановить корректный ход вычислений.

Традиционный подход, основанный на «длинных» переходах или проверке системных переменных `errno/error/...`, оказывается неадекватным для систематического использования.

Исключения – II

```
class Stack {
    …
public:
    class Overflow {
        int limit;
    public:
        Overflow(int n=0) : limit(n) {}
        char* Message() { return /*выдать информацию о допустимом размере*/; }
    }
    …
    bool empty() throw() { return sp==0; }
    void push(int x) throw(Overflow) {
        if (sp==sz)
            throw Overflow(sz);
        bf[sp++]=x;            //если ошибка, то std::unexpected
    }
}
…
try {
    stk.push(123);
}
catch (Stack::Overflow& err) {
    cout<<err.Message()<<endl;
}
…
```

Исключения – III

```
try {
    throw E();
}
catch (H) {
    //управление окажется здесь, если:
    //1. H того же типа, что и E.
    //2. H является однозначной открытой базой E.
    //3. H и E являются указателями, и пункт 1 или 2
    //   выполняется для типов, на которые они ссылаются.
    //4. H является ссылкой, и пункт 1 или 2 выполняется
    //   для типа, на который он ссылается.
}
```

```
try {
    //действия
}
catch (...) {
    //управление окажется здесь в случае любого
    //«C++»-исключения
}
```

Исключения – IV

- Повторная генерация исключения

```
try {
   throw E();
}
catch (...) {
   if (...)
     //внештатная ситуация может быть полностью обработана
   else
     throw; //то же самое исключение генерируется повторно
            //и передается выше
}
```

- Порядок обработки исключений

```
try {
    //возможны исключения
}
catch (ExceptionType1)
    //
catch (ExceptionType2)
    //
…
catch (...)
    //все остальные исключения
```

Исключения – V

Техника обратных вызовов (callback techniques)

```
#include “LexicalAnalyser.h”
namespace LA=LexicalAnalyser;
…
while (true)
    try {
        LA::Lexer(inputStream); //чтение и разбор входного потока
    }
    catch (LA::Number& number) {
        //обработать число
    }
    catch (LA::Keyword& keyword) {
        //обработать ключевое слово
    }
    catch (LA::Name& name) {
        //обработать имя
    }
    catch (LA::Error& err) {
        //обработать известную лексическую ошибку/прервать разбор
    }
    catch (LA::EndOfInput& eof) {
        //достигнут конец входного потока
    }
    catch (...) {
        //неизвестная ошибка
    }
```

Сортировка в стандартной С-библиотеке

файл stdlib.h

```
void qsort(
         void*   Base,
          size_t  NumOfElements,
         size_t  SizeOfElements,
         int     (*Compare)(const void*, const void*)
      );
```

файл myfile.h

```
#include <stdlib.h>
…
int cmp(const int* x,const int* y) {
  return (*x==*y ? 0 : (*x<*y ? -1 : 1));
}
…
int  arr[]={2,243,5,6,76,8,67,1,43,2};
…
qsort(arr, 10, sizeof(int), cmp);
…
```

Шаблоны – I

- Процедурное программирование: один раз пишем – используем многократно; в качестве параметров выступают конкретные значения, типы которых зафиксированы при разработке.
- Обобщенное программирование: один раз пишем – используем многократно; в качестве параметров выступают типы (и некоторые другие объекты языка). После настройки типа (выполняется автоматически компилятором) все как обычно.
- Идеи из языка Ada: generic types and packages – обобщенные/родовые типы и пакеты.

Шаблоны – II

```
template<class T>void qsort(T a[], int p, int r) {
   if (p < r) {
       T x=a[r], t;
      int i=p-1;
      for (int j=p; j<r; ++j) {
         if (a[j]<=x) {
            i++;
            t=a[i]; a[i]=a[j]; a[j]=t;
         }
      }
      t=a[i+1]; a[i+1]=a[r]; a[r]=t;
      int q=i+1;
      qsort<T>(a, p, q-1);
      qsort<T>(a, q+1, r);
   }
}
template<class T>void QuickSort(T a[], int sz) {
   qsort<T>(a, 0, sz-1);
}
void main() {
   int  arr[]={2,243,5,6,76,8,67,1,43,2};
   QuickSort<int>(arr,10);
}
```

Шаблоны – III

```
template<class Type, int size=100>
class Stack {
    int    sz;
    int    sp;
    Type   bf[size];
public:
    Stack() : sp(0), sz(size) {}
    …
    void push(const Type& x) {
        bf[sp++]=x;
    }
    Type pop() {
      return bf[--sp];
    }
};
```

```
#include "Stack.h"

class SomeClass {
  //нужно что-нибудь?
} var;

void main()
{
    Stack<int,50>    intStack;
    Stack<SomeClass> st[10];
    …
    intStack.push(123);
    st.push(var);
    while (!st.empty()) {
        …
        st.push(*new
       SomeClass());
        …
    }
    …
}
```

Шаблоны – IV

```
template<class Type> class Stack {
     int    sz;
     int    sp;
     Type  *bf;
public:
     Stack(int=100);
     ~Stack() {
          delete [] bf;
     }
     bool empty() const {
          return sp==0;
     }
     void push(const Type& x) {
       bf[sp++]=x;
     }
     Type pop();
};

template<class Type>
inline Stack<Type>::Stack(int size) {
     sp=0;
     bf=new Type[sz=size];
}

template<class Type>
inline Type Stack<Type>::pop() {
     return bf[--sp];
}
```

```
#include "Stack.h"

class SomeClass {};

void main()
{
     int n;
     cin >> n;
     Stack<int> intStack(n);
     Stack<SomeClass> st[10];
     …
     intStack.push(123);
     …
     while (!st.empty()) {
        …
     }
}
```

Template-реализация стека

```
namespace UsefulThings {
  template<class Type> class Stack {
    int    sz, sp;
    Type  *bf;
  public:
    class Stack_Overflow {};
    class Stack_Underflow {};
    Stack(int size=100) {
      sp=0;
      bf=new Type[sz=size];
    }
    Stack(const Stack& stk) {
      sp=stk.sp;
      bf=new Type[sz=stk.sz];
      for(int i=0; i<sp; ++i)
        bf[i]=stk.bf[i];
    }
    ~Stack() {
      delete[] bf;
    }
    Stack& operator=(const Stack& stk){
      delete[] bf;
      sp=stk.sp;
      bf=new Type[sz=stk.sz];
      for(int i=0; i<sp; ++i)
        bf[i]=stk.bf[i];
      return *this;
    }
    bool empty() const throw() {
      return sp==0;
    }
    void push(const Type& x) throw(Overflow) {
      if (sp==sz)
        throw Stack_Overflow();
      bf[sp++]=x;
    }
    Type pop() throw(Underflow) {
      if (empty())
        throw Stack_Underflow();
      return bf[--sp];
     }
  }; //конец определения класса
}    //конец определения пространства имен
```

Наследование в ООП – I

```
class SomeClass{
    int val;
public:
    SomeClass(int x) : val(x) {}
   operator int() const { return val; }
};

template<class T1,class T2> void proc(Stack<T1>& s1,Stack<T2>& s2)
{
    while (!s1.empty())
        s2.push(T2(s1.pop()));
}

void main()
{
    Buff_Stack<int> st1;
    List_Stack<SomeClass> st2;
    List_Stack<int> st3;
   //заполнить st1
    proc(st1,st2);
    proc(st2,st3);
}
```

Наследование в ООП – II

```
#include <list>


namespace UsefulThings {


template<class Type> class Stack {
public:
     class Stack_Overflow {};
     class Stack_Underflow {};
     virtual bool empty() const throw()=0;
     virtual void push(const Type& x)
                 throw(Stack_Overflow)=0;
     virtual Type pop()
                 throw(Stack_Underflow)=0;
   //Stack& operator=(const Stack& stk);
};



template<class Type>
class Buff_Stack : public Stack<Type> {
     int   sz, sp;
     Type  *bf;
public:
     Buff_Stack(int size=100);
     …
}
```

```
template<class Type>
class List_Stack : public Stack<Type> {
   std::list<Type>  lst;
public:
   List_Stack(){}     ~List_Stack(){}
   List_Stack(const List_Stack& src) :
      lst(src.lst.begin(),src.lst.end()) {}
   List_Stack& operator=(const List_Stack& src){
      lst.assign(src.begin(),src.end());
      return *this;
   }
   bool empty()const throw(){return lst.empty();}
   void push(const Type& x)throw(Stack_Overflow){
       lst.push_front(x);
   }
   Type pop() throw(Stack_Underflow) {
      if (empty())
         throw Stack::Stack_Underflow();
      Type x=lst.front();
      lst.pop_front();
      return x;
   }
}; //конец определения класса
}  //конец определения пространства имен
```

Наследование в ООП – III

Жизнь до ист.мат. Совмещение типов в С

```
typedef struct {
     int x;
     int y;
} Point;
typedef int Color;
typedef enum { circ, poly } Kind;
typedef struct {
     Kind  kind;
     Point startpoint;
     Color color;
     Color bordercolor;
} Common;
```

```
typedef struct {
     Common com;
     int    radius;
} Circle;
```

```
typedef struct {
     Common com;
     Point*
    vectors;
} Polygon;
```

```
typedef union {
     Circle  circle;
     Polygon polygon;
} Shape;
```

```
void RotatePolygon(Polygon p, int angle)
{ /*...*/ }

void Rotate(Shape* s, int angle)
{
     switch (s->circle.com.kind)
     {
     case circ:
       //вращать не надо!!!
       return;
     case poly:
       RotatePolygon(*s,angle);
       return;
     default:
       //ошибка!!!
     }
}

void RotateAll(Shape* figs,int n,int a)
{
     for(int i=0; i<n; ++i)
       rotate(figs+i,10);
}
```

Наследование в ООП – III

Еще одна попытка

```
typedef struct {
     Point startpoint;
     Color color;
     Color bordercolor;
} Shape;
Point Where(struct Shape s) { return s.startpoint; }
void Draw(struct Shape s) { ??? }
void Rotate(struct Shape s, int angle) { ??? }
```

```
typedef struct {
     Shape common;
     int   radius;
} Circle;

Point where(struct Circle s) {
     return s.common.startpoint;
}
void draw(struct Circle s) {
     //...
}
void rotate(struct Circle s, int angle){}
```

```
typedef struct {
     Shape  common;
     Point* vectors;
} Polygon;

Point where(struct Polygon s) {
     return s.common.startpoint;
}
void draw(struct Polygon s) {
     //...
}
void rotate(struct Polygon s, int angle){
     //...
}
```

Наследование в ООП – IV

Класс, обеспечивающий интерфейс для множества других классов, называется полиморфным типом.

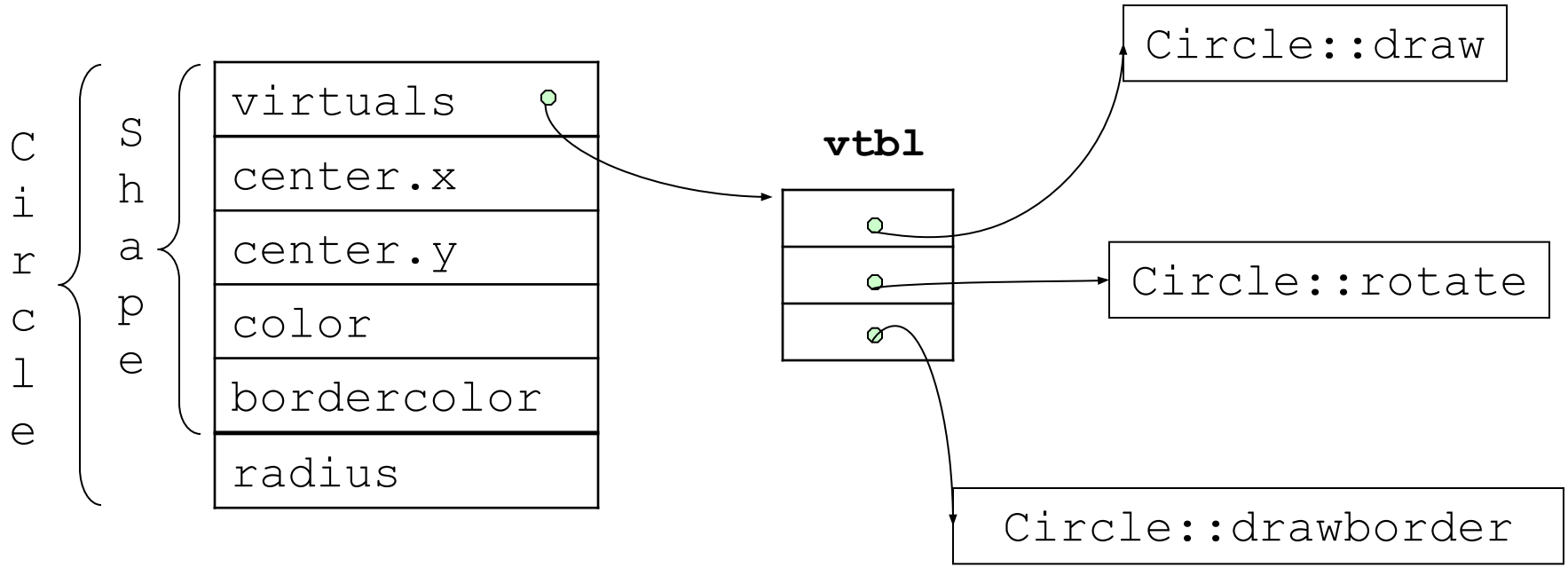
```
class Shape {
   Point center;
    Color color;
protected:
    Color bordercolor;
public:
   Shape(Color c) : color(c) {}
   Point where() {
      return center;
   }
    virtual void draw()=0;
   virtual void rotate(int angle)=0;
   void move(Point position) {
      center=position;
      draw();
   }
   virtual void drawborder()
   {/*...использование bordercolor...*/}
};
```

```
class Circle: public Shape {
    int radius;
public:
    Circle(Color c) :
          Shape(c), radius(0) {}
   void draw() {
      //...
   }
   void rotate(int) {}
    void drawborder() {
     bordercolor=new Color(…);
     Shape::drawborder();
   }
};
```

```
void RotateAll(Shape* figs,int n,int a){
    for(int i=0; i<n; ++i)
      figs[i]->rotate(10);
}
```

Наследование в ООП – V

Реализация



Наследование – VI

```
class Person {
     char name[100];
     Date birthday;
     int  department;
};


class Manager : public Person {
     list<Person*> staff;
     Date          appointment;
public:
     int           level;
};
```

Person p;

name
birthday
department

Manager m;

name
birthday
department
staff
appointment
level

```
void proc(Manager m, Person p) {
   Person*  pp=&m;
    Manager* pm=&p;
    //…=static_cast<Manager*>(&p);

    pm->level=2; //!!!
    pm=static_cast<Manager*>(pp);
    pm->level=2;

    p=m;    //!!!
    m=p;

    m.staff.push_front(&m);
    m.staff.push_front(&p);
}
```

Преобразования типов

- `static_cast`

```
int x=static_cast<int>(1.0);
```

- `reinterpret_cast`

```
int* x=reinterpret_cast<int*>(0xEEEE);
```

- `dynamic_cast`

```
void proc(Base* pb) {                          //pb не нулевой
   Derived* pd=dynamic_cast<Derived*>(pb);
   if (!pd)
      //pb не указывает на объект типа Derived или имеет
      //в качестве базовых классов более одного типа Derived
      …
}
```

- `const_cast`

```
   SomeClass* thisObject=const_cast<SomeClass*>(this);
```

- Конструкторы как преобразователи типов.
- Операторы-преобразователи типов.

Обзор STL

- string (строки);
- типы vector, list, deque, queue, stack, map, multimap, set, multiset, bitset;
- memory (распределение памяти);
- iterator (стандартные механизмы доступа)
- algorithm (стандартные алгоритмы: сортировка, перестановка, мин/макс, …)
- stdexcept (стандартные исключения)
- locale (информация о культурных особенностях)

Стек, предоставляемый STL

```
template<class Ty, class Container = deque<_Ty>> class stack {
    Container c;
public:
    typedef _Container container_type;
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef typename Container::reference reference;
    typedef typename Container::const_reference const_reference;

    stack() : c() {}
    explicit stack(const _Container& _Cont) : c(_Cont) {}
    bool empty() const { return (c.empty()); }
    size_type size() const { return (c.size()); }
    reference top() { return (c.back()); }
    const_reference top() const { return (c.back()); }
    void push(const value_type& _Val) { c.push_back(_Val); }
    void pop() { c.pop_back(); }
};


template<class _Ty, class _Container> inline bool operator==
                    (const stack<_Ty, Container>& Left,
                     const stack<_Ty, Container>& Right) {
    return (_Left.c == _Right.c);
}
```

Пример: for_each-шаблон

```
template<class _InIt, class _Fn1> inline _Fn1
                  for_each(_InIt _First, _InIt _Last, _Fn1_Func) {
    for (; _ChkFirst != _ChkLast; ++_ChkFirst)
        _Func(*_ChkFirst);
    return _Func;
}
...
class Average {
    long amount; long sum;
public:
    Average() : amount(0), sum(0) {}
    void operator()(int val) {
        amount++; sum+=val;
    }
    operator double() {
        return static_cast<double>(sum)/amount;
    }
};
...
vector<int> v1; //заполнить вектор значениями
cout<<for_each(v1.begin(), v1.end(), Average());
```

Современное ООП вне C++. Расширение Микрософт языка C++ – C++/CLI

Managed Extensions for C++ и C++/CLI

- Managed Extensions for C++ компании Микрософт появились в 2002 в связи с переходом на платформу .Net (Visual Studio .Net), в значительной мере унифицированную.
- С выходом Visual C++ 2005, Managed Extensions for C++ включены в расширенный синтаксис языка Visual C++, получивший название C++/CLI.
- CLI (Common Language Infrastructure) – это стандарт операционной обстановки, предназначенной для исполнения программного кода. CLR (Common Language Runtime) – реализация Микрософт стандарта CLI.
- Этот синтаксис не является частью ANSI/ISO стандарта C++. Он стандартизован в Ecma C++/CLI Specification.
- Нами в основном будут рассмотрены аспекты расширения, связанные с управлением памятью.

Функции CLR

- CLR обеспечивает автоматическое размещение объектов и управление ссылками на них, а также освобождение объектов, когда они больше не используются (сборка мусора). Объекты, время жизни которых управляется подобным образом, называются управляемыми данными. Сборка мусора исключает утечки памяти, появление висячих ссылок, удаление используемых объектов и некоторые другие ошибки программирования, возможные при использовании «неуправляемых» данных.
- CLR упрощает разработку компонентов и приложений, разработанных на разных языках, путем предоставления общей системы типов, определяемых средой, и саму эту среду.
- Программы компилируются в общий промежуточный язык (Common Intermediate Language, CIL), что дает возможность исполнения откомпилированного кода на любых процессорах и в любых операционных системах, поддерживающих среду выполнения.
- Управляемые приложения включают метаданные, содержащие сведения о используемых компонентах и ресурсах. Среда выполнения использует эти сведения, чтобы обеспечить наличие всех необходимых ресурсов для компонента или приложения.

Выделение памяти в управляемой куче

- При запуске приложения CLR резервирует для него непрерывную область адресного пространства – управляемую кучу (УК). Все ссылочные типы размещаются в УК. Она хранит указатель, с которого будет выделена память для следующего объекта в куче.
- Изначально этот указатель устанавливается в базовый адрес УК текущего приложения. При создании приложением очередного объекта ссылочного типа для него выделяется память в адресном пространстве, непосредственно следующем за уже размещенными объектами (по базовому адресу УК, если их еще не было).
- Пока имеется доступное адресное пространство, память для новых объектов продолжает выделяться по этой схеме, являющейся очень эффективной.
- Управляемая куча делится на два раздела. Один – для хранения мелких объектов. Второй – для объектов большого размера (>85000 байтов). Идея – большие объекты живут долго и перемещать их затратно.

Освобождение памяти в управляемой куче

- Когда сборщик мусора (СМ) выполняет сборку мусора, он освобождает память, выделенную для объектов, которые больше не используются приложением.
- Они определяются посредством построения графа достижимости для объектов.
- При обнаружении недостижимого объекта СМ выполняет копирование памяти для уплотнения достижимых объектов в памяти, освобождая блоки адресного пространства, выделенные под недостижимые объекты. Память уплотняется, только если при очистке обнаруживается значительное число недостижимых объектов.
- После уплотнения памяти, занимаемой достижимыми объектами, сборщик мусора вносит необходимые поправки в указатели приложения.

Эмпирические законы поведения объектов в куче

- Уплотнять память для части управляемой кучи быстрее, чем для всей кучи.
- Более новые объекты имеют меньшее время жизни, а более старые объекты имеют большее время жизни.
- Новые объекты теснее связаны друг с другом, и приложение обращается к ним приблизительно в одно и то же время.

Поколения объектов

- Объекты, созданные после последней сборки мусора или после запуска приложения до первой сборки мусора, являются объектами поколения 0.
- Если приложение пытается создать новый объект, когда поколение 0 заполнено, сборщик мусора выполняет сборку, пытаясь освободить для этого объекта адресное пространство в поколении 0 (уплотняя объекты поколения 0).
- Для объектов, оставшихся после сборок мусора, их уровень повышается, и они переводятся в поколение 1.
- Если уплотнение объектов поколения 0 не освободило достаточно памяти, то уплотняются объекты поколения 1, а затем снова поколение 0.
- Если это не принесло результатов, то последовательно уплотняются 2-1-0 (полная сборка мусора) и переводятся из поколения в поколение 0->1, 1->2.
- Объекты поколения 2 существуют до тех пор пока не будут удалены при сборке.

Расширения C++/CLI для управления памятью

- ref class / ref struct – управляемые к/с.
- gcnew – выделение памяти в управляемой куче.
- nullptr – константа, представляющая нулевой указатель.
- ^ – управляемый указатель.
- % – управляемая ссылка.
- array – управляемый массив.
- interior_ptr / pin_ptr – специфические указатели на объекты в управляемой куче.

Управляемые классы/структуры

- В то время как «родные» объекты C++ могут создаваться в разделах памяти разных типов (например, стек или куча), объекты ref-типов создаются в УК (managed heap). CLR поддерживает эту специальную кучу и реализует для нее асинхронный сборщик мусора. Обычные указатели C++ не могут указывать на объекты в УК.

```
ref class MyClass {
public:
    int val;
    MyClass(int i) : val(i) {}
};
```

- **^** - управляемый указатель (handle to an object on the managed heap), указывает на объект управляемой кучи целиком, создается с помощью **gcnew** и уничтожается **delete**.

```
MyClass ^ mp=gcnew MyClass(123); cout<<mp->val;
```

- **%** - управляемая ссылка (tracking reference), аналогична обычным ссылкам C++. Однако во время исполнения объект, на который она ссылается, может быть перемещен CLR сборщиком мусора. Управляемая ссылка размещается только на стеке. Она не может быть членом класса, но может указывать на член класса. Ей нельзя присвоить null. Управляемая ссылка может быть переприсвоена многократно.

```
MyClass ^% mr=mp;  int % r=mr->val;  mr->val=r+mr->val;
```

Внутренний указатель **interior_ptr**

Внутренний указатель **interior_ptr** используется для объявления указателя на члены объекта управляемого типа (внутрь и только туда). Внутренний указатель размещается только на стеке. Он не может быть членом класса. За исключением этого **interior_ptr** предоставляет ту же функциональность, что и обычный, включая сравнение и арифметику.

```
ref class IntClass { public: int data; };
...
IntClass^ obj = gcnew IntClass;
obj->data = 1;
interior_ptr<int> p1 = &(obj->data); //при перемещении СМ объекта
*p1 = 2;                             //obj будут корректироваться
interior_ptr<IntClass^> p2 = &obj;   //указатели p1 и p2
(*p2)->data = 3;
...
```

Пришпиленный указатель **pin_ptr**

Пришпиленный указатель **pin_ptr** является вариантом внутреннего указателя, который предотвращает перемещение СМ объекта, внутрь которого он указывает. Он может быть объявлен только как нестатическая локальная переменная на стеке. Главное назначение – охрана управляемых данных при передаче указателя на эти данные в функцию. Если **pin_ptr** присвоено новое значение (в том числе **nullptr**) или управление покинуло его область видимости, то объект, указываемый им ранее, перестает быть пришпиленным.

```
void unmanaged_function(int* p) {
    //делает что-то умное. во время исполнения может быть вызван СМ
}
ref class ArrClass {
   array<int>^ arr;
public:
   ArrClass() { arr = gcnew array<int>(1000); }
   void load() {
    pin_ptr<int> p = &arr[0];
    int* ump = p;
    unmanaged_function(ump);
   }
};
```

Некоторые другие расширения

- value class
- interface class
- property
- delegate
- abstract
- new
- override
- sealed
- finally
- for each
- safe_cast
- typeid
- event