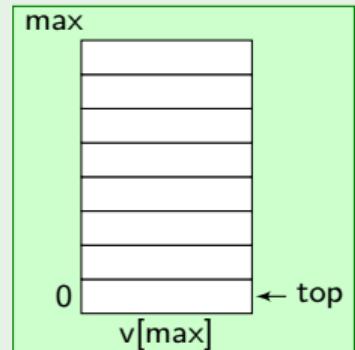


Концепция классов

☞ Что дает класс по сравнению с С-структурой?

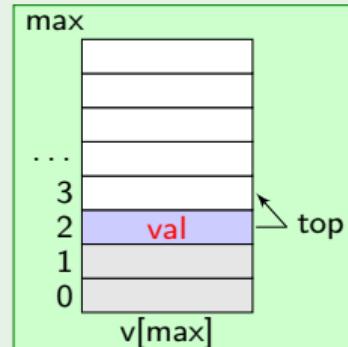
Пример: реализации стека на С на базе динамического массива v[]

```
struct stack {  
    double* v;  
    int top;  
};  
typedef struct stack Stack;  
  
void init(Stack *S) {  
    const int max = 20;  
    S->v = (double*)  
        malloc(max*sizeof(S->v[0]));  
    S->top = 0;  
}
```



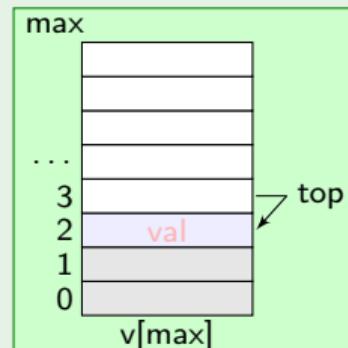
2 добавление элемента

```
void push(Stack *S, double val) {  
    S->v[ S->top ] = val;  
    (S->top)++;  
}
```



3 извлечение элемента

```
double pop(Stack *S) {  
    (S->top)--;  
    return (S->v[S->top]);  
}
```



Пример использования в С

```
Stack S;
init(&S);           // initialize
push(&S,1.1);       // push a few elements
push(&S,2.2);       // on the stack...
push(&S,pop(&S)+pop(&S)); // replace top two elements by their sum
printf("sum= %g\n",pop(&S)); // sum= 3.3
```

Тем не менее легко получить ошибку исполнения (run time errors)

```
Stack A,B;
init(&A);
push(&B, 3.);      // core dump: didn't initialize B

A.top = -4;        // don't do this!
A.v[3] = 2.;       // don't do this!

init(&A);          // just wiped out A, memory leak
init(&B);

B = A;             // weird, but legal, memory leak
```

Недостатки реализации стека с C-struct

- 👉 Инициализация стека должна быть выполнена «вручную», но повторная инициализация очищает старые данные и приводит к утечке памяти
- 👉 Все внутренние переменные доступны для модификации, данные могут быть легко испорчены, что приводит к трудно находимым ошибкам
- 👉 Операция присвоения `A=B` для структур содержащих указатели приводит к появлению «двойных ссылок» и утечке памяти
- 👉 Связь функций работающих со структурами и самих структур неочевидна: имена `init()`, `pop()`, `push()` могут встретиться где угодно
- 👉 Стек только для `double`, для `int` надо всё переписывать

Класс: основные понятия

- ① Инкапсуляция (*encapsulation*): и данные и функции помещаются внутрь класса: **class ≈ C-structure + functions**
- ② Разграничение уровня доступа (*data hiding*)
public: данные и функции из этой части доступны всюду
private: можно использовать только в функциях этого же класса

```
class Stack {  
public:                      // открытая часть (доступно всем)  
    Stack(unsigned int max); // конструктор (инициализация)  
    ~Stack();                // деструктор  
    void push(double val);   // функция-1  
    double pop();            // функция-2  
private:                     // закрытая часть (для внутреннего использования)  
    double* v;               // контейнер для стека  
    int top;                 // «верхушка» стека  
};
```

- ③ Конструктор – специальная функция инициализации: **автоматически вызывается** в момент создания объекта; таких функций может быть несколько, для разных сценариев инициализации

Имя конструктора совпадает с именем класса

```
Stack(unsigned int max=20U) {  
    top = 0;  
    max = (max > 20U) ? max : 20U;  
    v = new double[max];  
}
```

- ④ Деструктор – функция завершения жизни объекта:
автоматически вызывается чтобы освободить ресурсы
 в случае Stack надо освободить память

Имя деструктора = \sim + имя класса

```
 $\sim$ Stack() { delete[] v; } // освобождение памяти
```

⑤ Функции класса (методы): их можно переопределять

- ☞ можно вызывать только для объектов данного класса
- ☞ имеют доступ ко всем данным и функциям собственного класса

Пример: две функции push()

```
void    push(double val) { v[top++]=val; }
void    push(int val)     { v[top++]=double(val); }
```

⑥ Можно переопределять (перегружать) существующие операций:

= + - * ++ -- += << ...

Пример: переопределение оператора вывода на печать

```
std::ostream& operator<<(std::ostream& os, const Stack& s) {
    ...
    return os;
}
```

⑦ Механизм наследования классов, иерархия классов ...

Стек в C++, простая, «наивная» реализация

Описание класса обычно помещают в отдельный файл Stack.h

```
#ifndef _Stack_h
#define _Stack_h
class Stack {
public:
    Stack(unsigned int max=20U) : top(0) {
        max = (max > 20U) ? max : 20U;
        v = new double[max];
    }
    ~Stack() { delete[] v; }
    void push(double val) { v[top++]=val; }
    double pop()           { return v[--top]; }
private:
    double* v;
    int     top;
};
#endif
```

Проверка на ошибки: Stack в C++

```
Stack A,B;           // ctor -> initialize A and B
// init(&A);          // forget it, it done in ctor
B.push(3.);          // OK!

A.top = -4;          // Compilation error: 'int Stack::top' is private
A.v[3] = 2.;          // Compilation error: 'double* Stack::v' is private

// init(&A);          // impossible, let's try call ctor
Stack A;              // Compilation error: redeclaration of 'Stack A'
// init(&B);          // forget it, it done in ctor

B = A;                // still weird... we will fix it: see operator=()
```

Декларация класса

```
class Name_of_class {  
    int var1;                      // default section = private  
public:  
    double var2;                   // public section  
    int func2();  
private:                         // private section  
    int* var3; void* func3();  
public:  
    Name_of_class(int a);         // ctor: may be missing  
    ~Name_of_class();            // dtor: may be missing  
}; // <- semicolon required!
```

- ➊ Действие меток `public:` и `private:` простирается от места их объявления до следующей метки или до конца класса
- ➋ По умолчанию в классе действует правило `private`

Служебное слово struct

struct в C++: класс в котором по умолчанию действует public
пример

```
struct print {  
    int count;                                // variable  
    print() : count(0) {}                      // constructor  
    void operator() (int x) {                  // member function  
        cout << x << ' '; ++count; }  
};
```

Агрегатная инициализация структур в C++

пример

```
struct Simple { int a; double b; };  
Simple A {1, 0.2}; // Aggregate initialization  
cout << "A.a= " << A.a << " A.b= " << A.b << '\n'; // A.a= 1 A.b= 0.2
```

☛ Не должно быть: конструкторов, приватных членов, наследования ...

Деструктор (dtor)

«Функция» автоматически выполняемая при уничтожении объекта

- основное назначение – освобождение ресурсов «захваченных» объектом: освобождение памяти, закрытие файлов и так далее
- имя деструктора строится по схеме: тильда + имя класса: `~Stack()`
 - ☞ нет ни параметров ни возвращаемого значения: имеется единственный вариант написания
- если нет явно объявленного деструктора, компилятор сам создаст деструктор по умолчанию
 - ✓ можно попросить компилятор создать деструктор по умолчанию:

```
~Stack(){}; // default dtor in C++98
```

```
~Stack() = default; // default dtor in C++11
```

☞ явно вызывать деструктор не надо!

Конструктор (ctor)

Конструктор – «функция» для инициализации объектов класса

- имя конструктора совпадает с именем класса: `Stack()`
☞ нет никакого возвращаемого значения

Может быть несколько конструкторов различающихся сигнатурой

```
// конструкторы по умолчанию  
Stack S;  
Stack* pS = new Stack;  
Stack S2 = Stack();  
Stack S3 {};
```

```
// конструкторы "общего вида"  
Stack T1(30), T2(50);  
Stack* pT = new Stack(50);  
Stack T3 {1,2,3};
```

```
// копирующие конструкторы  
Stack C(S);  
Stack* pC = new Stack(S);  
Stack C2 = Stack(S);  
Stack C3 {S};
```

// инициализация списком C++11

Конструктор по умолчанию

☞ это конструктор, который можно вызвать без аргументов

- если **нет ни одного явно объявленного конструктора**, компилятор сам создаст конструктор по умолчанию

✓ можно попросить компилятор создать конструктор по умолчанию:

```
Stack(){};           // default ctor in C++98  
Stack() = default; // default ctor in C++11
```

```
struct A {  
    int a;  
    A(int b=0): a(b) {} // user defined default ctor  
};  
struct B {  
    A a;  
}; // B::B() is implicitly defined ctor, calls A::A()
```

Инициализация внутри класса в C++11

In-class member initializers

- для любой переменной внутри класса можно указать значение, которое будет использовано при инициализации
- в конструкторах эти переменные могут использоваться и изменяться

```
class Stack {  
    explicit Stack(int cap = 0) {  
        if ( cap > capacity ) {capacity = cap;}  
        v = new double[capacity];  
    }  
    ...  
private:  
    double*      v = nullptr;  
    int          top = 0;  
    int          capacity = 20;  
};
```

Копирующий конструктор

- Один аргумент, ссылка на объект класса того же типа:
`Stack(const Stack& S); // copy ctor for Stack`
- Если не объявлен явно, то создается компилятором: **копирует каждый элемент класса**
- ☞ Использование копирующего конструктора «от компилятора» может приводить к ошибкам времени исполнения

Пример: проблемы с копированием Stack

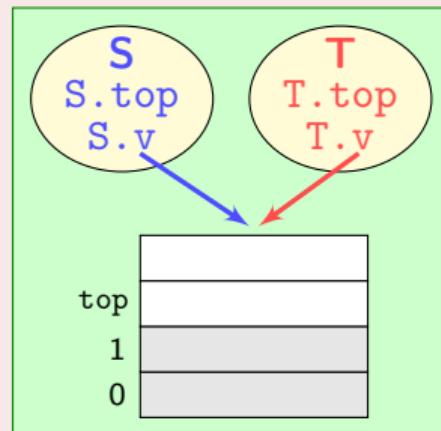
```
Stack T(S); // copy constructor, или Stack T = S;  
T.push(3.3);  
S.push(2.2);  
cout << T.pop() << endl; // 2.2  
при завершении программы ☞ Aborted: double free or corruption
```

Что происходит?

- «псевдокод» создаваемый компилятором:

```
Stack(const Stack& S) {  
    v = S.v; // здесь ошибка!  
    top = S.top;  
    ...  
}
```

- ✗ Вызов **Stack T(S)** приводит к тому, что S и T используют одну и ту же области памяти для хранения содержимого стека



Явное определение копирующего конструктора

Разрешающее определение: правильный конструктор в public

```
Stack::Stack(const Stack& a) : top(a.top) {  
    v = new double[max(top,20)]; // Выделяем новую память!  
    for(int i = 0; i < top; i++) {v[i] = a.v[i];} // копируем содержимое  
}
```

Запрещающее определение:

- C++98: помещаем конструктор в **private** секцию:

```
private:  
    Stack(const Stack& a);
```

- C++11: в любом месте, например, в списке конструкторов:

```
Stack(const Stack& a) = delete;
```

Оператор присваивания =

оператор типа: CLASS& operator=(const CLASS& a)

- копирует в имеющийся объект содержимое другого объекта класса
- по умолчанию создается компилятором

могут возникнуть проблемы ...

```
Stack S,T; // создаем стеки S и T  
T = S; // копируем S T  
T.push(3.3);  
S.push(2.2);  
cout<<T.pop()<<endl; // 2.2 ... в конце ↗ double free or corruption
```

☞ Дополнительная проблема: `T = S` переписывает `T.v` и, следовательно, теряется доступ к ранее выделенной памяти: `memory leak`

Переопределение оператора присваивания:

```
Stack& Stack::operator=(const Stack& a) {  
    if ( this == &a ) return *this; // проверка что это не S = S;  
    top=a.top;  
    delete [] v;                // free old memory  
    v = new double [max(20,top)]; // allocate new memory  
    for(int i = 0; i < top; i++) v[i] = a.v[i]; // copy content  
    return *this; // must be for chain: S1=S2=S3  
}
```

Запрещающая декларация:

- C++98: декларация оператора в **private**

private:

```
Stack& operator=(const Stack& a);
```

- C++11: в любом месте класса:

```
Stack& operator=(const Stack&) = delete;
```

Декларации =default и =delete в C++11

Используются для конструкторов, деструктора и operator=()

☞ указание компилятору:

- = default – создать «функцию» по умолчанию
- = delete – запретить использование «функции» данного вида

```
struct A {  
    A() = default;           // ctor по умолчанию  
    ~A() = default;          // dtor по умолчанию  
};  
  
struct B{  
    B() = delete;            // запрет ctor по умолчанию  
    B(const B&) = delete;     // запрет copy ctor  
    B& operator=(const B&) = delete; // запрет присваивания  
    ~B() = delete;           // запрет dtor  
};
```

Ссылка как член класса

☞ Ссылку можно инициализировать только в конструкторе

- ❶ Конструктор по умолчанию невозможен
- ❷ Оператора копирования нет: должен быть запрещен

```
struct TestRefMem {  
    int& v;  
    TestRefMem(int& i): v(i) {} // 'TestRefMem::v' ссылка на 'i'  
    TestRefMem& operator=(const TestRefMem&) = delete; // явный запрет  
};  
  
int i {1}, j {2};  
TestRefMem a(i), b(j);  
a=b;           // compilation error  
TestRefMem c(a); // copy ctor is OK!  
i = 10;  
cout << " c.v= " << c.v << endl; // c.v= 10
```

Дополнительные слайды