תכנות מתקדם
מצגת 3

בניית מחלקה

נושאים

- מחלקה
- בניית המחלקה וקטור
- הזזה
- חריגות

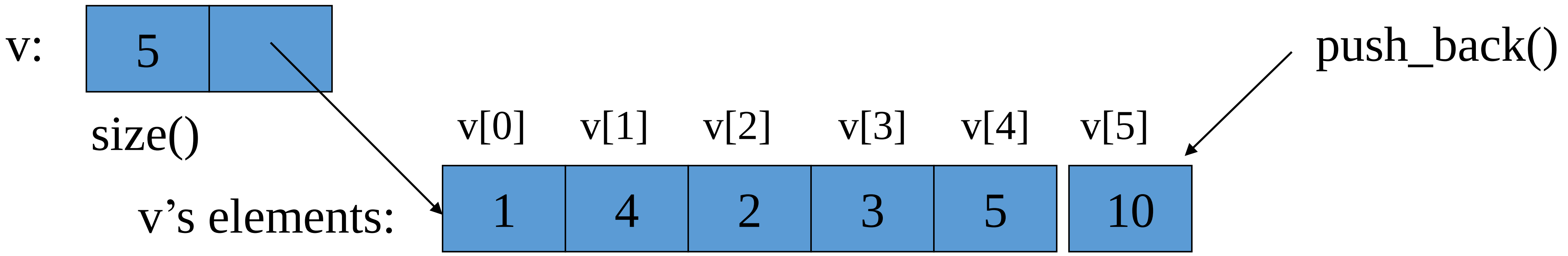
מחלקה

- מחלקה היא הרחבה של struct שבשפת C
  - במחלקה ניתן להגדיר בנוסף למשתנים של struct גם פונקציות חברות במחלקה (Member Functions)
- **מחלקה מאפשרת:**
- **הפשטת נתונים (Data Abstraction)** התעלמות מפרטי המימוש של העצם והתרכזות במאפיינים שלו
- **כימוס (Encapsulation)** הסתרת פרטי המימוש מהמשתמש
  - ניתן לקבוע הרשאות גישה לחברי המחלקה:
  - חברי מחלקה המוגדרים private נגישים רק לפונקציות חברות במחלקה
  - חברי מחלקה המוגדרים public נגישים גם לשאר פונקציות התכנית

בניית המחלקה וקטור

- וקטור הוא אחד המיכלים בספריה הסטנדרטית והשימושי ביותר
- וקטור בדומה למערך המובנה בשפה מכיל סדרה של נתונים מאותו סוג, אך יש לו תכונות נוספות, לדוגמה: אפשר להגדילו, להעתיקו, לדעת את גודלו
- פעולות שכיחות בוקטור:

```
vector<int> v = {1,2,3,4,5} // initialize with a list
v[i] = 7; // access element i
v.push_back(10); // add an element at end
```



מעבר על וקטור באמצעות אינדקס

```
int main() { // compute average temperatures
     vector<double> temps;
     double temp;
// cin >> temp returns a reference to cin
// if end of input it is converted to false
     while (cin >> temp) // idiom
          temps.push_back(temp);
     double sum = 0;
     for (int i = 0; i < temps.size(); ++i)
          sum += temps[i];
     cout << "Average: " << sum/temps.size() << '\n';
}
```

מעבר על וקטור עם הוספות של C++11

```
// use list initialization
vector<int> v = {10,20,30,40,50,60,70,80,90,100};
for (vector<int>::size_type i = 0; i != 5; ++i)
    cout << v[i] << " ";
// use range for to process all the elements
for (int i : v) cout << i << " ";
// let auto deduce the type of i
for (auto i : v) sum +=  i;
// use decltype instead of vector<int>::size_type
for (decltype(v.size()) i = 5; i != 10; ++i)
    cout << v[i] << " ";
```

מעבר על וקטור באמצעות איטרטור

```
vector<int> v = {10, 20, 30, 40, 50};

vector<int>::iterator iter = v.begin();
decltype(v.end()) end_iter = v.end();
while (iter != end_iter) {
    cout << *iter << endl;
    ++iter;
}

for (auto it = v.cbegin(); it != v.cend(); ++it)
    cout << *it << endl;
```

מימוש בסיסי של וקטור

```
class Vector {
    int sz;          // the size
    double* elem; // a pointer to the elements
public:
    using size_type = unsigned long;
    Vector(): sz{0}, elem{nullptr} {} //default constructor
    Vector(int s) // constructor (s is the element count)
      :sz{s}, elem{new double[s]} // initialize
      { for (int i = 0; i<sz; ++i) elem[i] = 0.0; }
    ~Vector()        // destructor
      { delete[] elem; }
    int size() { return sz; }};
Vector v1;      // use default constructor, not Vector v1();
Vector v2(10); // create a vector with 10 elements
```

nullptr

- We try to ensure that a pointer always points to an object, so that dereferencing it is valid
- When we don't have an object to point to, we give the pointer the value nullptr
- In older code, **0** or **NULL** is typically used, but . . .

```
void func(int n); void func(char *s); func( NULL );
                 // which function is called? (int)
```

- using **nullptr** eliminates confusion between integers and pointers

```
func( nullptr ); // func(char *s) is called

double* pd = nullptr;
int x = nullptr; // error : nullptr is a pointer
```

בנאי ברירת מחדל (= default)

- If our class does not explicitly define any constructors, the compiler will implicitly define the default constructor for us
- It default-initializes the members
- Objects of builtin or compound type (such as arrays and pointers) that are defined inside a block have **undefined** value
- we can ask the compiler to generate the default constructor for us by writing = default

```
class Vector {
    Vector() = default;
```

- We are defining this constructor only because we want to provide other constructors

בנאי שמבצע המרה

- A constructor that takes a single argument defines a conversion from its argument type to its class, for example:

```
class complex {
   complex(double,double);
   complex(double); // defines double-to-complex
                    // conversion
   // . . .
};
complex z = complex{1.2,3.4};
z = 5.6; // OK, converts 5.6 to complex(5.6,0)
         // and assigns to z
```

explicit

- However, implicit conversions may cause unexpected effects:

```
Vector(int); // defined constructor with int parameter
Vector v = {2, 5, 8};
v = 10; // converts 10 to Vector(10) and assigns to v
void do_something(vector v);
do_something(7); // call with a vector of 7 elements
```

- A constructor defined **explicit** provides only the usual construction semantics and **not the implicit conversions**

```
class Vector {
   explicit Vector(int);
Vector v(10); // OK, explicit
v = 40; // error, no int-to-vector conversion
```

אתחול וקטור

- Initialize to default and then assign:

```
Vector v1(2); // error prone:
v1[0] = 1.2; v1[1] = 2.4; v1[2] = 7.8;
```

- Use push_back:

```
Vector v2;      // tedious
v2.push_back(1.2); v2.push_back(2.4); v2.push_back(7.8);
```

- push_back is useful for input:

```
Vector read(istream& is) {
Vector v;for(double d; is >> d;) v.push_back(d);return v}
```

- Best use { } delimited list of elements:

```
Vector v3 = {1.2, 7.89, 12.34}; // C++11
```

בנאי לאתחול מרשימה

- A { } delimited list of elements of type T is presented to the programmer as an object of type initializer_list<T>

```
class Vector {
    int sz;          // the size
    double* elem; // a pointer to the elements
public:
    Vector(initializer_list<double> lst) // constructor
        :sz{lst.size()}, elem{new double[sz]}
        { copy( lst.begin(),lst.end(),elem); }
        // copy using standard library algorithm
};
Vector v1(3);     // three elements
vector v2{3};     // one element
vector v3 = {3}; // one element
```

בנאי העתקה

- A constructor is the copy constructor if its first parameter is a reference to the class:

```
Vector(const Vector& rhs) ; // copy constructor
```

- copy constructor can be used with direct initialization and copy initialization:

```
string dots(10, '.');
string s(dots); // direct initialization
string null_book = "99999"; // copy initialization
string nines = string(100, '9'); // copy initialization
```

- Copy initialization happens also when passing an object to a function or returning an object from a function
- if we use an initializer that requires conversion by an explicit constructor:

```
vector<int> v1(10); // ok: direct initialization
vector<int> v2 = 10; // error: constructor is explicit
void f(vector<int>); // f's parameter is copy initialized
f(10); // error: can't use an explicit constructor
f(vector<int>(10)); // ok: construct a temporary vector
```

בנאי העתקה (ברירת מחדל)

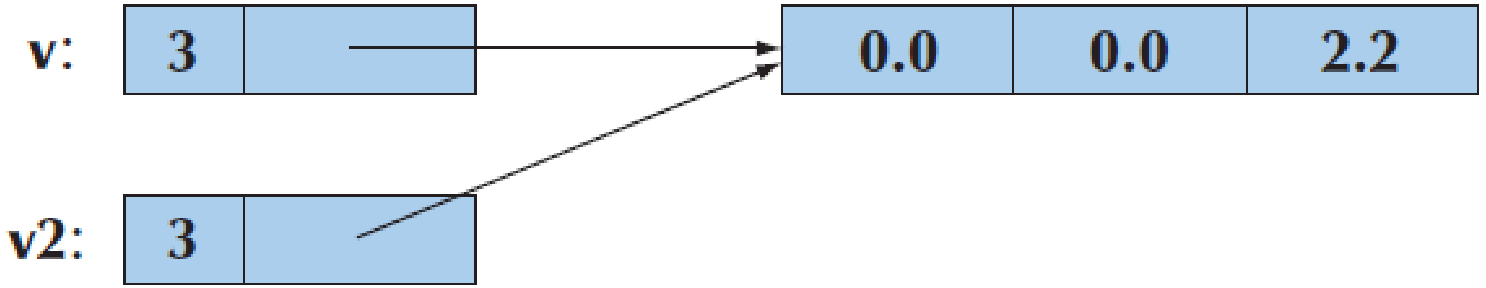
- The default meaning of copy is member-wise copy
- For the vector **pointer member** it means that after:

```
Vector v2 = v; // use copy constructor
```

- We have:

```
v.elem == v2.elem
```

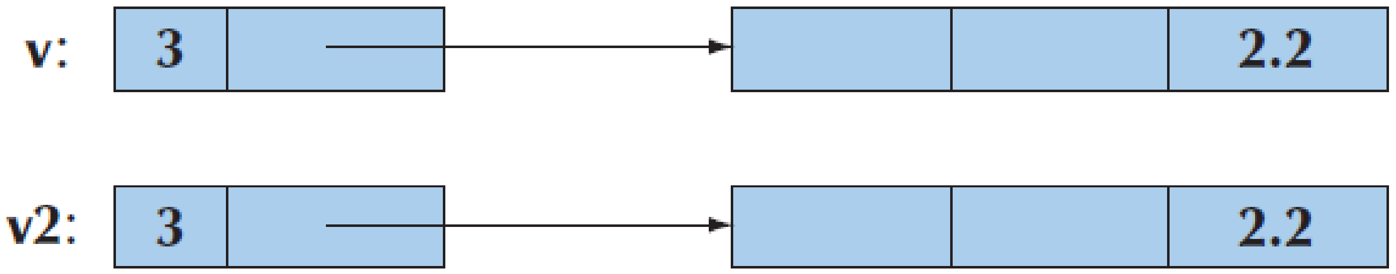
- **v2** doesn’t have a **copy** of **v** elements as expected , but **shares v** elements
- When the destructors for **v** and **v2** are implicitly called, memory will be **freed twice**



בנאי העתקה שמעתיק כראוי

- The copy constructor **allocates** memory for the elements before copying

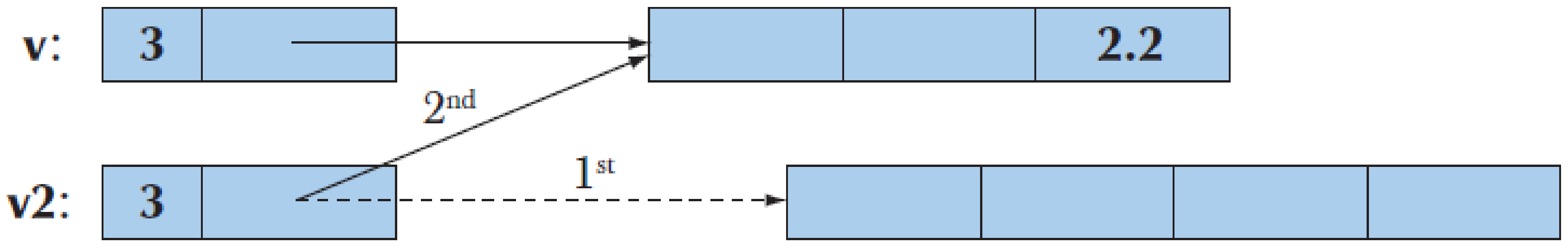
```
class Vector {
    int sz;
    double* elem;
public:
    Vector(const Vector& rhs) ; // copy constructor
        :sz{rhs.sz}, elem{new double[rhs.sz]};
        { copy(rhs.elem, rhs.elem+sz, elem); }
```



השמת העתקה (ברירת מחדל)

- As with copy initialization, the default meaning of copy assignment is member-wise copy
- Assignment will cause a **double deletion** and **memory leak**

```
Vector v(3);
v.set(2,2.2);
Vector v2(4);
v2 = v;
```

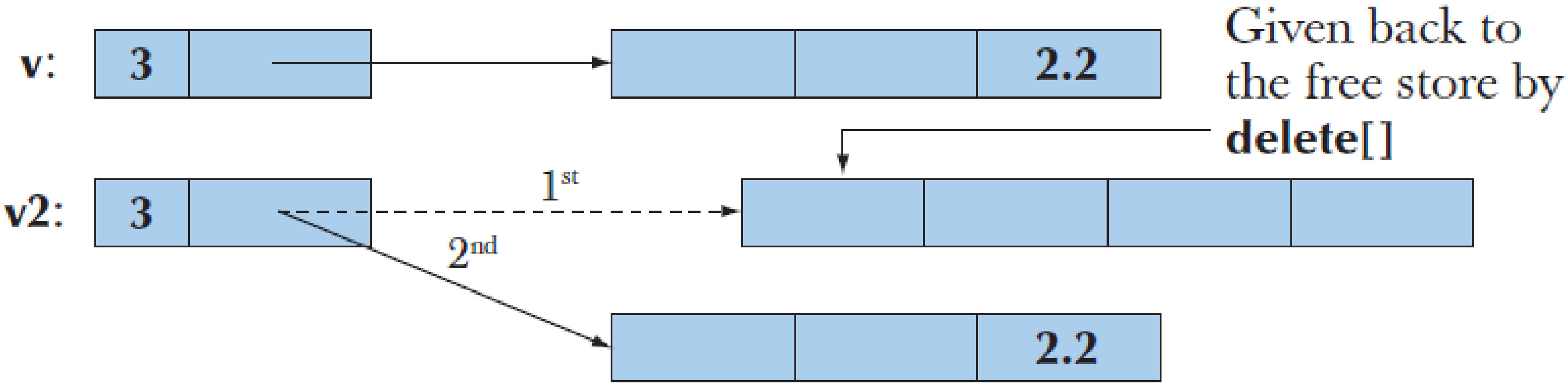


השמת העתקה שמעתיקה כראוי

```
class Vector {
    Vector& operator=(const Vector&) ; // copy assignment
// . . .
Vector& Vector::operator=(const Vector& rhs)
{
    double* p = new double[rhs.sz]; // allocate new space
    copy(rhs.elem, rhs.elem+rhs.sz, p); // copy elements
    delete[] elem; // deallocate old space
    elem = p; // now we can reset elem
    sz = rhs.sz;
    return *this; // return a self-reference
    // To be consistent with built-in types
}
```

העתקה כראוי

- We make a copy of the elements from the source vector
- **Then** we free the old elements from the target vector
- Finally, we let elem point to the new elements
- The case of **self assignment** (`v = v;`) is handled correctly



- **Shallow copy** copies only a pointer so that the two pointers now refer to the same object
- **Deep copy** copies what a pointer points to so that the two pointers now refer to distinct objects

מניעת העתקה (= delete)

- we can prevent copies by defining the copy constructor and copy assignment operator as deleted functions:

```
struct NoCopy {
NoCopy() = default; // use the default constructor
NoCopy(const NoCopy&) = delete; // no copy
NoCopy &operator=(const NoCopy&) = delete; // no
assignment
~NoCopy() = default; // use the default destructor
// other members
};
```

lvalue and rvalue

- An **lvalue** can appear on the left side of an assignment operator
  - It is is an object that **can be modified**
- An **rvalue** appears on the right side of an assignment expression
  - It is an expression that identifies something **temporary** that **can not be modified**
- In the assignment statements:

```
y = x + 2;  // y is an lvalue,  x + 2 is an rvalue
z = 7;      // z is an lvalue,  7 is an rvalue
s = f(x);   // f(x) is an rvalue
x + 2 = y;  // Error
7 = z;      // Error
f(x) = s;   // Error
```

rvalue references

- It is illegal to assign a temporary rvalue to a reference variable:

```
int& r = x + 3;   // Error
int i = 42;
int &r = i; // ok: r refers to i
```

- The following function call is illegal:

```
int f(int& n) { return 10 * n; }
x = f(x + 2);
```

- C++ **does** have an **rvalue reference**:

```
int&& r = x + 3; // OK: note the two ampersands
int&& rr = i; // error: cannot reference an lvalue
```

- The following function call is OK:

```
int g(int&& n) { return 10 * n; }
x = g(x + 2);
```

העמסת פונקציות עם & ו- &&

```
void ref(int& n) {
  cout << "reference parameter: " << n << endl;
}
void ref(int&& n) {
  cout << "rvalue reference parameter: " << n << endl;
}

int main() {
    int x = 10;
    ref(x);                         // lvalue
    ref(x + 10);                    // rvalue
    ref(30);                        // rvalue
    ref(std::move(x));              // lvalue cast to rvalue
}
```

בנאי הזזה

```
Vector::Vector(Vector&& a)
    :sz{a.sz}, elem{a.elem} // move a.elem to elem
{
    a.sz = 0; // make a the empty vector
    a.elem = nullptr;
}

vector fill(istream& is) {
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}
vector vec = fill(cin);
```

- Copying res out of fill() and into vec could be expensive, the **move constructor** is implicitly used to implement the return

השמת הזזה

```
Vector& Vector::operator=(Vector&& a)
{
    delete[] elem; // deallocate old space
    elem = a.elem; // move a.elem to elem
    sz = a.sz;
    a.elem = nullptr; // make a the empty vector
    a.sz = 0;
    return *this; // return a self-reference
}
```

- If the caller passes an **rvalue**, the compiler generates code that invokes the **move constructor** or **move assignment** operator
- We want to avoid making a copy of the temporary

פעולות נדרשות במחלקה שתופסת משאבים

- A class needs a **destructor** if it **acquires resources**:
  - The obvious example is **memory** that you get from the free store (using **new**) and have to give back to the free store (using **delete** or delete[])
  - Other resources are **files** (if you open one, you also have to close it), **locks**, **thread handles**, and **sockets** (for communication)
- If a class has a **destructor**, it is likely to need all the following functions:

```
X(Sometype);                  // ordinary constructor
X();                          // default constructor
X(const X&);                  // copy constructor
X(X&&);                       // move constructor
X& operator=(const X&);       // copy assignment
X& operator=(X&&);            // move assignment
~X();                         // destructor
```

העמסת [ ]

```
double operator[] (int i) {
    return elem[i];
}
```

- However, letting the subscript operator return a value enables **reading** but not **writing** of elements:

```
Vector v(10);
double x = v[2]; // fine
v[3] = x;        // error, v[3] is not an lvalue
```

- We **have to return a reference** from the subscript operator:

```
double& operator[ ](int n)
{
    return elem[n];
}
```

העמסת [ ] לפי const

- The subscript operator defined so far has a problem, it cannot be invoked for a **const** vector.
- Only **const** member functions can be invoked for **const** objects
- For example:

```
void f(const vector& cv)
{
    double d = cv[1]; // Error, but should be fine
    cv[1] = 2.0;      // Error, as it should be
}
```

- The solution is to provide a version that is a **const** member function:

```
double& operator[] (int n);           // for non-const
const double& operator[] (int n) const; // for const
```

כללי העמסת +

- we define the arithmetic and relational operators as **nonmember** functions
  - in order to allow conversions for either the left- or right-hand operand
- These operators need not change the state of either operand
  - so the parameters are ordinarily references to **const**
- Classes that define an arithmetic operator generally define the corresponding **compound assignment operator** as well
- It is usually more efficient to define the arithmetic operator **to use compound assignment**:

```
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; // copy from lhs into sum
    sum += rhs; // add rhs into sum
    return sum;
}
```

העמסת + לוקטור

```
Vector operator+(const Vector& a, const Vector& b)
{
   if (a.size()!= b.size())
       throw Vector_size_mismatch{};
   Vector res(a.size());
   for (int i=0;i!=a.size();++i) res[i]=a[i]+b[i];
   return res;
}

Vector r;
r = x + y;
```

כללי העמסת אופרטור הפלט <<

- The first parameter of an output operator is a **reference** to a **nonconst** ostream object
  - **nonconst** because writing to the stream changes its state.
  - **reference** because we cannot copy an ostream object
- The second parameter should be a **reference** to **const** to avoid copying and to avoid change
- To be consistent with other output operators, operator<< **returns** its ostream parameter
- output operators **should not print a newline** in order to let users print descriptive text along with the object on the same line
- IO Operators must be **nonmember** functions, the left-hand operand cannot be an object of our class
- IO operators usually need to read or write the nonpublic data members, so they usually must be declared as **friends**

העמסת אופרטור הפלט << לוקטור

```
ostream& operator<<(ostream& os, const Vector& vec)
{
    os << '{';
    int n = vec.size();
    if (n > 0) { // Is the vector non-empty?
        os << vec[0]; // Send first element
        for (int i = 1; i < n; i++)
            os << ',' << vec[i];
    }
    os << '}';
    return os;
}

cout << vec1 << endl;
```

כללי העמסת אופרטור הקלט >>

- The first parameter is a **reference** to the stream from which it is to read
- The second parameter is a **reference** to the **nonconst** object into which to read, because the operator reads data into this object
- The operator usually **returns a reference** to its given stream

```
class Sales_data {
     std::string bookNo;
     unsigned units_sold = 0;
     double price = 0;
     double revenue = 0.0;

istream &operator>>(istream &is, Sales_data &item)
{
     is >> item.bookNo >> item.units_sold >> item.price;
```

כללי העמסת אופרטור הקלט <<

- Input operators must deal with the possibility that the input might fail
- we check once after reading all the data and before using those data:

```
if (is) // check that the inputs succeeded
      item.revenue = item.units_sold * item.price;
else
      item = Sales_data();
```

- If there was an **error**, we **reset** the entire object to the empty Sales_data

איטרטורים

```
class Vector {
    int sz;         // the size
    double* elem; // a pointer to the elements
public:
    typedef double* iterator;
    typedef const double* const_iterator;

    iterator begin() { return elem; }
    const_iterator cbegin() const { return elem; }
    iterator end() { return elem+sz; }
    const_end cend() const { return elem+sz; }
// . . .
};
```

תבנית

- We don’t want just vectors of doubles, we want to specify the element type

```
template<typename T>
class Vector {
   T* elem; // elem points to an array of type T
   int sz;
public:
   explicit Vector(int s);
   T& operator[](int i);
   const T& operator[](int i) const;
};
template<typename T>
Vector<T>::Vector(int s) { . . . elem = new T[s]; . . .}
```

exceptions חריגות

- One effect of the modularity of a program, is that the point where a run-time error can be detected is separated from the point where it can be handled
- Consider a Vector, what ought to be done when we try to access an element that is out of range for the vector
  - The writer of Vector doesn’t know what the user would like to do in this case
  - The user of Vector cannot consistently detect the problem
- The solution is for the Vector implementer to detect the attempted out-of-range access and then tell the user about it

throw

- Vector::operator[ ] can detect an attempted out-of-range access and throw an out_of_range exception

```
double& Vector::operator[](int i)
{
    if (i < 0 || i >= size())
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

- The throw transfers control to a handler for exceptions of type **out_of_range** in some function that called Vector::operator[ ]
- The out_of_range type is defined in the standard library

try and catch

- The implementation will unwind the function call stack as needed to get back to the context of the caller that has expressed interest in handling that kind of exception
- The standard library does not throw out_of_range for subscript operator, but throws for at()

```
try { // exceptions are handled below
   // v[v.size()] = 7; // returns an undefined value
   v.at(v.size) = 7     // reports a bad index
}
catch (out_of_range) { // oops: out_of_range error
   // ... handle range error ...
}
```

push_back()

```
void Vector::push_back(const double& val)
{
        double* p = new double[sz+1];
        copy(elem, elem+sz, p);
        p[sz] = val;
        delete[] elem;
        elem = p;
        ++sz;
}
```

- Problem, for each push_back() we have to **copy** the whole vector

```
v.push_back(7); // need more space
v.push_back(8); // need more space
```

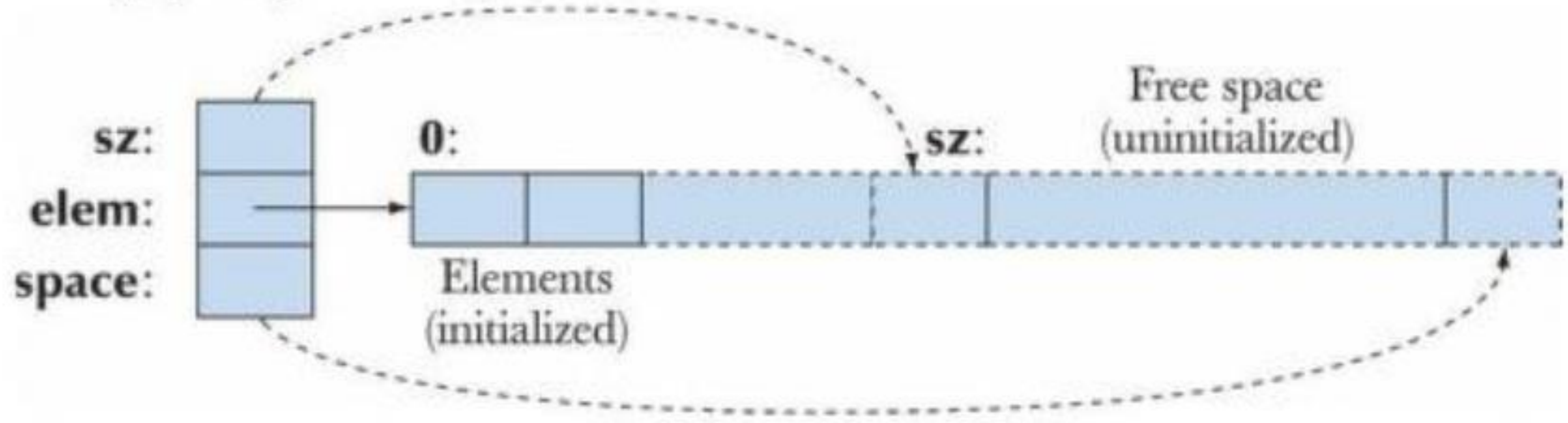
push_back()

- To avoid copying, we have to allocate extra space and keep track of both the **number of elements** and **amount of space** allocated

```
class vector {
  int sz;       // one beyond the last vector element
  int space;    // one beyond the last allocated element
  double* elem; // a pointer to the elements
```

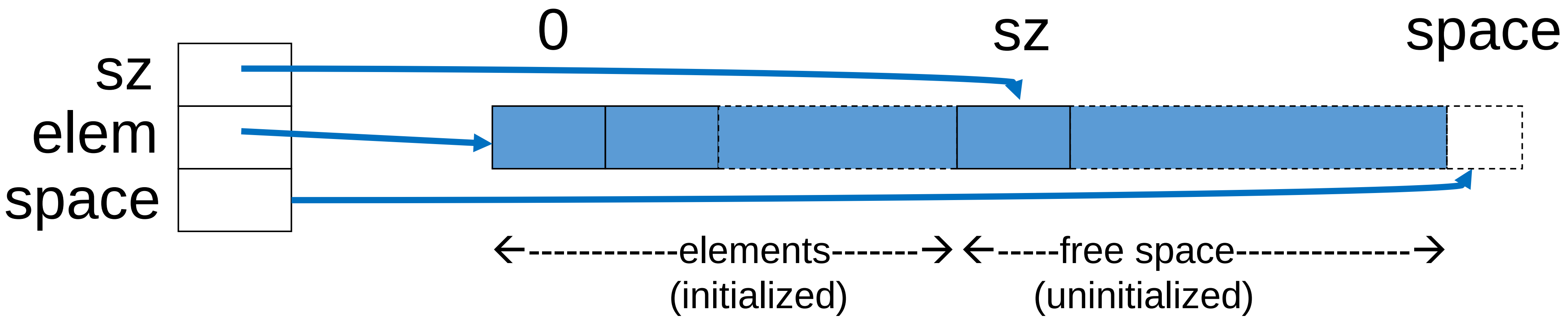
- The default constructor creates an empty vector:

```
vector():sz{0}, space{0}, elem{nullptr} {}
```

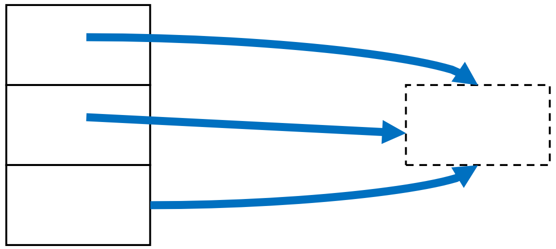


ייצוג וקטור

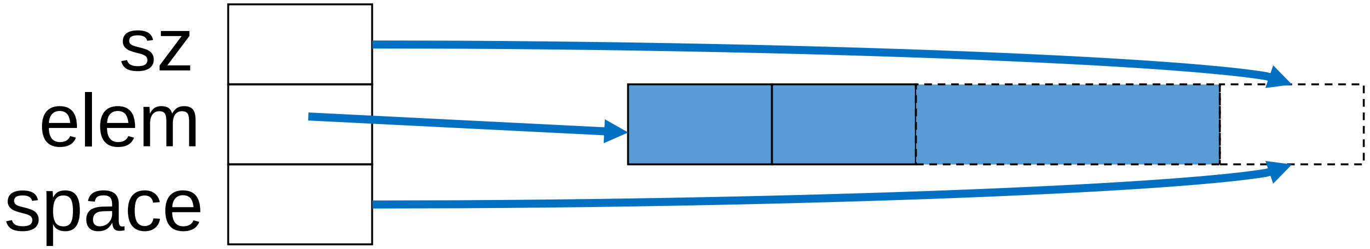
A vector(n) (free space)



An empty vector(no free store use)



A vector(n) (no free space)



push_back()

```
void Vector::reserve(int newalloc) {
    double* p new double[newalloc];
    for (int i=0; i<sz; ++i) p[i] = elem[i];
    delete[] elem;
    elem = p;
    space = newalloc;
}
void Vector::push_back(double val) {
    if (space == 0) reserve(8);
    else if (sz == space) reserve(2*space);
    elem[sz] = val;
    ++sz;
}
```