

Chapter 17 vector and Free Store

Bjarne Stroustrup
www.stroustrup.com/Programming

Abstract

- Vector is not just the most useful standard container, it also provides examples of some of the most important/powerful/ interesting implementation techniques. In this and the following lectures, we go through a series of increasingly sophisticated vector implementations, seeing classical problems related to use of memory and providing solutions. Here, we discuss free store (heap storage) management, and pointers.

Stroustrup/Programming

2

Overview

- Vector revisited
 - How are they implemented?
- Pointers and free store
 - Allocation (new)
 - Access
 - Arrays and subscripting: []
 - Dereferencing: *
 - Deallocation (delete)
- Destructors
- Copy constructor and copy assignment
- Arrays
- Array and pointer problems
- Changing size
- Templates
- Range checking and exceptions

Stroustrup/Programming

3

Vector

- Vector is the most useful container
 - Simple
 - Compactly stores elements of a given type
 - Efficient access
 - Expands to hold any number of elements
 - Optionally range-checked access
- How is that done?
 - That is, how is vector implemented?
 - We'll answer that gradually, feature after feature
- Vector is the default container
 - prefer vector for storing elements unless there's a good reason not to

Stroustrup/Programming

4

Building from the ground up

- The hardware provides memory and addresses
 - Low level
 - Untyped
 - Fixed-sized
 - No checking
 - As fast as the hardware architects can make it
- The application builder needs something like a vector
 - Higher-level operations
 - Type checked
 - Size varies (as we get more data)
 - Run-time checking
 - Close-to optimally fast

Stroustrup/Programming

5

Building from the ground up

- At the lowest level, close to the hardware, life's simple and brutal
 - You have to program everything yourself
 - You have no type checking to help you
 - Run-time errors are found when data is corrupted or the program crashes
- We want to get to a higher level as quickly as we can
 - To become productive and reliable
 - To use a language "fit for humans"
- Chapter 17-19 basically shows all the steps needed
 - The alternative to understanding is to believe in "magic"
 - The techniques for building vector are the ones underlying all higher-level work with data structures

Stroustrup/Programming

6

Vector

■ A vector

- Can hold an arbitrary number of elements
 - Up to whatever physical memory and the operating system can handle
- That number can vary over time
 - E.g. by using `push_back()`
- Example


```
vector<double> age(4);
age[0]=.33; age[1]=22.0; age[2]=27.2; age[3]=54.2;
```



Stroustrup/Programming

7

Vector

// a very simplified **vector** of **doubles** (like `vector<double>`):

```
class vector {
    int sz;           // the number of elements ("the size")
    double* elem;     // pointer to the first element
public:
    vector(int s);    // constructor: allocate s elements,
                    // let elem point to them
                    // store s in sz
    int size() const { return sz; } // the current size
};
```

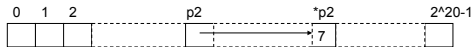
- `*` means "pointer to" so `double*` is a "pointer to `double`"
 - What is a "pointer"?
 - how do we make a pointer "point to" elements?
 - How do we "allocate" elements?

Stroustrup/Programming

8

Pointer values

- Pointer values are memory addresses
 - Think of them as a kind of integer values
 - The first byte of memory is 0, the next 1, and so on



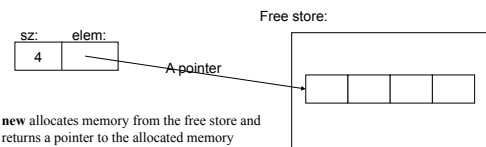
- A pointer points to an object of a given type
 - E.g. a `double*` points to a `double`, not to a `string`
- A pointer's type determines how the memory referred to by the pointer's value is used
 - E.g. what a `double*` points to can be added not, say, concatenated

Stroustrup/Programming

9

Vector (constructor)

```
vector::vector(int s) // vector's constructor
    :sz(s),           // store the size s in sz
    elem(new double[s]) // allocate s doubles on the free store
    // store a pointer to those doubles in elem
{
}
// Note: new does not initialize elements (but the standard vector does)
```

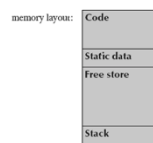


`new` allocates memory from the free store and returns a pointer to the allocated memory

Stroustrup/Programming

10

The computer's memory



- As a program sees it
 - Local variables "lives on the stack"
 - Global variables are "static data"
 - The executable code are in "the code section"

Stroustrup/Programming

11

The free store (sometimes called "the heap")

- You request memory "to be allocated" "on the free store" by the `new` operator
 - The `new` operator returns a pointer to the allocated memory
 - A pointer is the address of the first byte of the memory
 - For example
 - `int* p = new int;` // allocate one uninitialized `int`
// `int*` means "pointer to `int`"
 - `int* q = new int[7];` // allocate seven uninitialized `ints`
// "an array of 7 `ints`"
 - `double* pd = new double[n];` // allocate `n` uninitialized `doubles`
 - A pointer points to an object of its specified type
 - A pointer does *not* know how many elements it points to



12

Access



- Individual elements

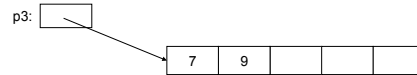
```
int* p1 = new int;           // get (allocate) a new uninitialized int
int* p2 = new int(5);        // get a new int initialized to 5

int x = *p2;                 // get/read the value pointed to by p2
                             // (or "get the contents of what p2 points to")
                             // in this case, the integer 5
int y = *p1;                 // undefined: y gets an undefined value; don't do that
```

Stroustrup/Programming

13

Access



- Arrays (sequences of elements)

```
int* p3 = new int[5];        // get (allocate) 5 ints
                             // array elements are numbered 0, 1, 2, ...

p3[0] = 7;                   // write to ("set") the 1st element of p3
p3[1] = 9;

int x2 = p3[1];              // get the value of the 2nd element of p3

int x3 = *p3;                // we can also use the dereference operator * for an array
                             // *p3 means p3[0] (and vice versa)
```

Stroustrup/Programming

14

Why use free store?

- To allocate objects that have to outlive the function that creates them:
 - For example

```
double* make(int i)
{
    return new double[i];
}
```

- Another example: vector's constructor

Stroustrup/Programming

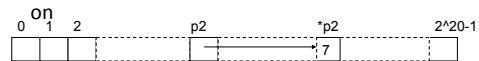
15

Pointer values

■ Pointer values are memory addresses

- Think of them as a kind of integer values

- The first byte of memory is 0, the next 1, and so



// you can see pointer value (but you rarely need/want to):

```
char* p1 = new char('c');    // allocate a char and initialize it to 'c'
int* p2 = new int(7);         // allocate an int and initialize it to 7
cout << "p1==" << p1 << " *p1==" << *p1 << "\n"; // p1==??? *p1=c
cout << "p2==" << p2 << " *p2==" << *p2 << "\n"; // p2==??? *p2=7
```

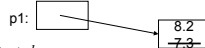
Stroustrup/Programming

16

Access

- A pointer does **not** know the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 7.3;           // ok
p1[0] = 8.2;         // ok
p1[17] = 9.4;        // ouch! Undetected error
p1[-4] = 2.4;        // ouch! Another undetected error
```



```
double* p2 = new double[100];
*p2 = 7.3;           // ok
p2[17] = 9.4;        // ok
p2[-4] = 2.4;        // ouch! Undetected error
```



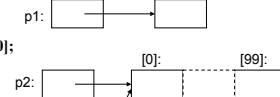
Stroustrup/Programming

17

Access

- A pointer does **not** know the number of elements that it's pointing to

```
double* p1 = new double;
double* p2 = new double[100];
```



```
p1[17] = 9.4; // error (obviously)
```

```
p1 = p2;      // assign the value of p2 to p1
```



```
p1[17] = 9.4; // now ok: p1 now points to the array of 100 doubles
```

Stroustrup/Programming

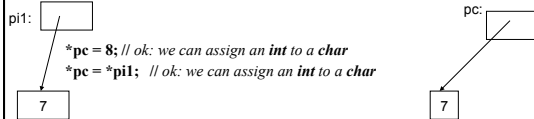
18

Access

- A pointer *does* know the type of the object that it's pointing to


```
int* pi1 = new int(7);
int* pi2 = pi1; // ok: pi2 points to the same object as pi1
double* pd = pi1; // error: can't assign an int* to a double*
char* pc = pi1; // error: can't assign an int* to a char*
```

 - There are no implicit conversions between a pointer to one value type to a pointer to another value type
 - However, there are implicit conversions between value types:



Stroustrup/Programming

19

Pointers, arrays, and vector

Note

- With pointers and arrays we are "touching" hardware directly with only the most minimal help from the language. Here is where serious programming errors can most easily be made, resulting in malfunctioning programs and obscure bugs

- Be careful and operate at this level only when you really need to

- vector is one way of getting almost all of the flexibility and performance of arrays with greater support from the language (read: fewer bugs and less debug time).

Stroustrup/Programming

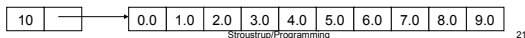
20

Vector (construction and primitive access)

// a very simplified vector of doubles:

```
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    vector(int s):sz(s), elem(new double[s]) { } // constructor
    double get(int n) { return elem[n]; } // access: read
    void set(int n, double v) { elem[n]=v; } // access: write
    int size() const { return sz; } // the current size
};

vector v(10);
for (int i=0; i<v.size(); ++i) { v.set(i,i); cout << v.get(i) << ' '; }
```



Stroustrup/Programming

21

A problem: memory leak

```
double* calc(int result_size, int max)
{
    double* p = new double[max]; // allocate another max doubles
    // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // ... use p to calculate results to be put in result ...
    return result;
}
```

```
double* r = calc(200,100); // oops! We "forgot" to give the memory
// allocated for p back to the free store
```

- Lack of de-allocation (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

Stroustrup/Programming

22

A problem: memory leak

```
double* calc(int result_size, int max)
{
    int* p = new double[max]; // allocate another max doubles
    // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // ... use p to calculate results to be put in result ...
    delete[] p; // de-allocate (free) that array
    // i.e., give the array back to the free store
    return result;
}

double* r = calc(200,100);
// use r
delete[] r; // easy to forget
```

Stroustrup/Programming

23

Memory leaks

- A program that needs to run "forever" can't afford any memory leaks
 - An operating system is an example of a program that "runs forever"
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
 - Trick question: not enough data to answer, but about 130,000 calls
- All memory is returned to the system at the end of the program
 - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
 - i.e., memory leaks aren't "good/bad" but they can be a problem in specific circumstances

Stroustrup/Programming

24

Memory leaks

- Another way to get a memory leak

```
void f()
{
    double* p = new double[27];
    // ...
    p = new double[42];
    // ...
    delete[] p;

    // 1st array (of 27 doubles) leaked
}
```

Stroustrup/Programming

25

Memory leaks

- How do we systematically and simply avoid memory leaks?
 - don't mess directly with **new** and **delete**
 - Use **vector**, etc.
 - Or use a garbage collector
 - A garbage collector is a program that keeps track of all of your allocations and returns unused free-store allocated memory to the free store (not covered in this course; see <http://www.research.att.com/~bs/C++.html>)
 - Unfortunately, even a garbage collector doesn't prevent all leaks
 - See also Chapter 25

Stroustrup/Programming

26

A problem: memory leak

```
void f(int x)
{
    vector v(x); // define a vector
                // (which allocates x doubles on the free store)
    // ... use v ...

    // give the memory allocated by v back to the free store
    // but how? (vector's elem data member is private)
}
```

Stroustrup/Programming

27

Vector (destructor)

```
// a very simplified vector of doubles:
class vector {
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    vector(int s) // constructor: allocates/acquires memory
        :sz(s), elem(new double[s]) {}
    ~vector() // destructor: de-allocates/releases memory
    { delete[] elem; }
    // ...
};
```

- Note: this is an example of a general and important technique:
 - acquire resources in a constructor
 - release them in the destructor
- Examples of resources: memory, files, locks, threads, sockets

Stroustrup/Programming

28

A problem: memory leak

```
void f(int x)
{
    int* p = new int[x]; // allocate x ints
    vector v(x); // define a vector (which allocates another x ints)
    // ... use p and v ...
    delete[] p; // deallocate the array pointed to by p
    // the memory allocated by v is implicitly deleted here by vector's destructor
}
```

- The **delete** now looks verbose and ugly
 - How do we avoid forgetting to **delete[] p**?
 - Experience shows that we often forget
- Prefer **deletes** in destructors

Stroustrup/Programming

29

Free store summary

- Allocate using **new**
 - **new** allocates an object on the free store, sometimes initializes it, and returns a pointer to it
 - `int* pi = new int;` // default initialization (none for `int`)
 - `char* pc = new char{'a'};` // explicit initialization
 - `double* pd = new double[10];` // allocation of (uninitialized) array
 - **new** throws a **bad_alloc** exception if it can't allocate
- Deallocate using **delete** and **delete[]**
 - **delete** and **delete[]** return the memory of an object allocated by **new** to the free store so that the free store can use it for new allocations
 - `delete pi;` // deallocate an individual object
 - `delete pc;` // deallocate an individual object
 - `delete[] pd;` // deallocate an array
 - Delete of a zero-valued pointer ("the null pointer") does nothing
 - `char* p = 0;`
 - `delete p;` // harmless

Stroustrup/Programming

30

void*

- **void*** means "pointer to some memory that the compiler doesn't know the type of"
- We use **void*** when we want to transmit an address between pieces of code that really don't know each other's types – so the programmer has to know
 - Example: the arguments of a callback function
- There are no objects of type **void**
 - `void v;` // error
 - `void f();` // `f()` returns nothing – `f()` does **not** return an object of type **void**
- Any pointer to object can be assigned to a **void***
 - `int* pi = new int;`
 - `double* pd = new double[10];`
 - `void* pv1 = pi;`
 - `void* pv2 = pd;`

Stroustrup/Programming

31

void*

- To use a **void*** we must tell the compiler what it points to

```
void f(void* pv)
{
    void* pv2 = pv; // copying is ok (copying is what void*s are for)
    double* pd = pv; // error: cannot convert void* to double*
    *pv = 7;         // error: you can't dereference a void*
                    // good! The int 7 is not represented like the double 7.0)
    pv[2] = 9;       // error: you can't subscript a void*
    pv++;            // error: you can't increment a void*
    int* pi = static_cast<int*>(pv); // ok: explicit conversion
    // ...
}
```

- A **static_cast** can be used to explicitly convert to a pointer to object type
 - "static_cast" is a deliberately ugly name for an ugly (and dangerous) operation – use it only when absolutely necessary

Stroustrup/Programming

32

void*

- **void*** is the closest C++ has to a plain machine address
 - Some system facilities require a **void***
 - Remember FLTK callbacks?
 - **Address** is a **void***:


```
typedef void* Address;
void Lines_window::cb_next(Address,Address)
```

Stroustrup/Programming

33

Pointers and references

- Think of a reference as an automatically dereferenced pointer
 - Or as "an alternative name for an object"
 - A reference must be initialized
 - The value of a reference cannot be changed after initialization


```
int x = 7;
int y = 8;
int* p = &x; *p = 9;
p = &y; // ok
int& r = x; x = 10;
r = &y; // error (and so is all other attempts to change what r refers to)
```

Stroustrup/Programming

34

Next lecture

- The next lecture discusses copying and arrays

Stroustrup/Programming

35