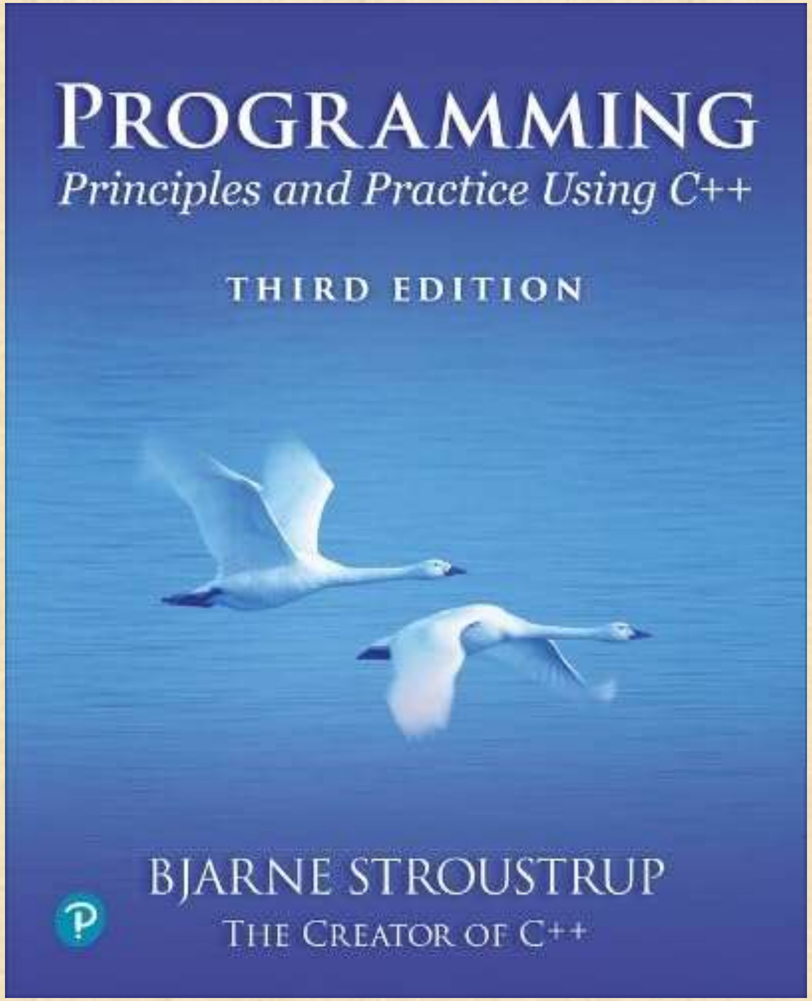
Chapter 18 – Templates and Exceptions



Success is never final.
– Winston Churchill

Overview

- Templates
- Generalizing **Vector**
  - Allocators
  - Range checking and exceptions
- Resources and exceptions
  - RAII for **Vector**
- Resource-management pointers
  - Return by moving, **unique_ptr**, and **shared_ptr**

We now have a decent **Vector** of **double**s

- But we don’t just want a **Vector** of **double**s
- We want vectors with element types we specify
  - **Vector<double>**
  - **Vector<int>**
  - **Vector<Month>**
  - **Vector<Record*>** *// vector of pointers*
  - **Vector<Vector<Record>>** *// vector of vector of Records*
  - **Vector<char>**
- We must make the element type a parameter to **Vector**
  - both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler
  - we can define our own parameterized types, called **template**s

Templates

- The basis for generic programming in C++
  - Sometimes called “parametric polymorphism”
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (*e.g.,* hard real time and numerics)
    - Used where flexibility is essential (*e.g.,* the C++ standard library)

- Template definitions

```
template<class T, int N> class Buffer { /* … */ };
template<class T, int N> void fill(Buffer<T,N>& b) { /* … */ }
```

- Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf;        // for buf, T is char and N is 1024

// for a function template, the compiler deduces the template arguments:
fill(buf);        // for fill(), T is char and N is 1024; that's what buf has
```

Parameterize with element type

```
// an almost real vector of Ts:
template<class T> class Vector {
  // …
};

Vector<double> vd;                  // T is double
Vector<int> vi;                     // T is int
Vector<Vector<int>> vvi;            // T is Vector<int>  in which T is int
Vector<char> vc;                    // T is char
Vector<double*> vpd;                // T is double*
Vector<Vector<double>*> vvpd;       // T is Vector<double>* in which T is double
```

Basically, **Vector<double>** is

```
// an almost real Vector of doubles:
class Vector {
  int sz;                 // the size
  double* elem;           // a pointer to the elements
  int space;              // size+free_space
public:
  Vector() : sz(0), elem(0), space(0) { }                             // default constructor
  explicit Vector(int s) :sz(s), elem(new double[s]), space(s) { }    // constructor
  Vector(const Vector&);                                              // copy constructor
  Vector& operator=(const Vector&);                                   // copy assignment
  ~vector() { delete[ ] elem; }                                       // destructor
  // …
  double& operator[ ] (int n) { return elem[n]; }                     // access: return reference
  int size() const { return sz; }                                     // the current size
  // …
};
```

Basically, **Vector<char>** is

```
// an almost real Vector of chars:
class Vector {
  int sz;                 // the size
  char* elem;             // a pointer to the elements
  int space;              // size+free_space
public:
  Vector() : sz{0}, elem{0}, space{0} { }                           // default constructor
  explicit Vector(int s) :sz{s}, elem{new char[s]}, space{s} { }    // constructor
  Vector(const Vector&);                                            // copy constructor
  Vector& operator=(const Vector&);                                 // copy assignment
  ~Vector() { delete[ ] elem; }                                     // destructor
  // …
  char& operator[ ] (int n) { return elem[n]; }                     // access: return reference
  int size() const { return sz; }                                   // the current size
  // …
};
```

Basically, **vector<T>** is

```
// an almost real Vector of Ts:
template<class T> class Vector {          // read "for all types T" (just like in math)
    int sz;                               // the size
    T* elem;                              // a pointer to the elements
    int space;                            // size+free_space
public:
    Vector() : sz{0}, elem{0}, space{0};                          // default constructor
    explicit Vector(int s) :sz{s}, elem{new T[s]}, space{s} { } // constructor
    Vector(const Vector&);                                      // copy constructor
    Vector& operator=(const Vector&);                             // copy assignment
    ~vector() { delete[ ] elem; }                                 // destructor
    // …
    T& operator[ ] (int n) { return elem[n]; }                    // access: return reference
    int size() const { return sz; }                               // the current size
    // …
};
```

Template instantiation

- Consider

```
template<typename T> class Vector {/* … */ };

void fct(Vector<string>& v)
{
    int n = v.size();
    v.push_back("Ada");
    // ...
}
```

- Given that, the compiler generates definitions
  - `Vector<string>::size() { /* … */ }`
  - `Vector<string>::push_back(const string&) { /* … */ }`
- That’s commonly called “Template instantiation”
- Note: the compiler can use information from different places to generate great code
  - Template definition
  - Template arguments
  - Calling context

What is “Generic Programming”?

- Suggestions
  - The most general, most efficient, most flexible representation of concepts” – Alex Stepanov
  - Using **templates**
  - Writing code that works for a set of arguments that meets some requirements
  - Parametric polymorphism
  - Compile-time resolution of overloaded calls
    - As opposed to the run-time resolution of virtual function calls in object-oriented programming
  - Represent separate concepts separately in code and combine concepts freely wherever meaningful
  - Focus on the design and use of generic functions: “algorithm-oriented programming”

- To simplify
  - ***Generic programming***: supported by templates, relying on compile-time resolution
  - ***Object-oriented programming***: supported by class hierarchies with virtual functions, allowing run-time resolution

Concepts – specifying template requirements

- Least specific, worst error messages:

```
template<typename T>                    // for all types T
class Vector { /* ... */ };
```

Specify requirements on T by a predicate:

```
template<typename T>                    // for all types T
        requires Element<T>()           // such that Element<T>() is true
class Vector { /* ... */ };
```

- Convenient shorthand:

```
template<Element T>                     // for all types T, such that T is an Element
class Vector { /* ... */ };
```

We have always had concepts

- A **concept** is a predicate stating what **template** requires of its **template** arguments
- Every successful generic library has some form of concepts
  - In the designer's head
  - In the documentation
  - In comments
- Examples
  - C/C++ built-in type concepts: arithmetic and floating
  - Mathematical concepts like monad, group, ring, and field
  - Graph concepts like edges and vertices, graph, DAG, ...
  - STL concepts like iterators, sequences, and containers
- C++ offers direct language support
  - A concept is a compile-time predicate
  - Using concepts is easier than not using them

Some useful concepts

- **range<C>()**: **C** can hold **Element**s and be accessed as a sequence
- **random_access_iterator<Ran>()**: **Ran** can be used to read and write a sequence repeatedly and supports subscripting using **[ ]**
- **random_access_range<Ran>()**: **Ran** is a **range** with **random_access_iterator**s
- **equality_comparable<T>()**: We can compare two **T**s for equality using **==** to get a Boolean result
- **integral<T>()**: A **T** is an integral type (like **int**)
- **floating_point<T>()**: A **T** is a floating-point type (like **double**)
- **copyable<T>()**: A **T** can be copied
- **invocable<F,T...>()**: We can call **F** with a set of arguments of the n specified types **T1**, **T2**, ...
- **semiregular<T>()**: A **T** can be copied, moved, and swapped (that is, “a pretty normal type”)
- **regular<T>()**: **T** is **semiregular** and **equality_comparable**

Generalizing Vector yet again

- Soon it will behave much like **std::vector**
- Containers needs to be able to control where their elements are stored
  - **Vector<string, my_string_allocator > v;**
- Elements may be of types with non-trivial constructors and destructors
  - **Vector<Vector<string>> v;**
- We need to be able to change the number of elements of a Vector
  - **v.push_back(x);**
  - **v.resize(1’000’000);**

Allocators – controlling memory management

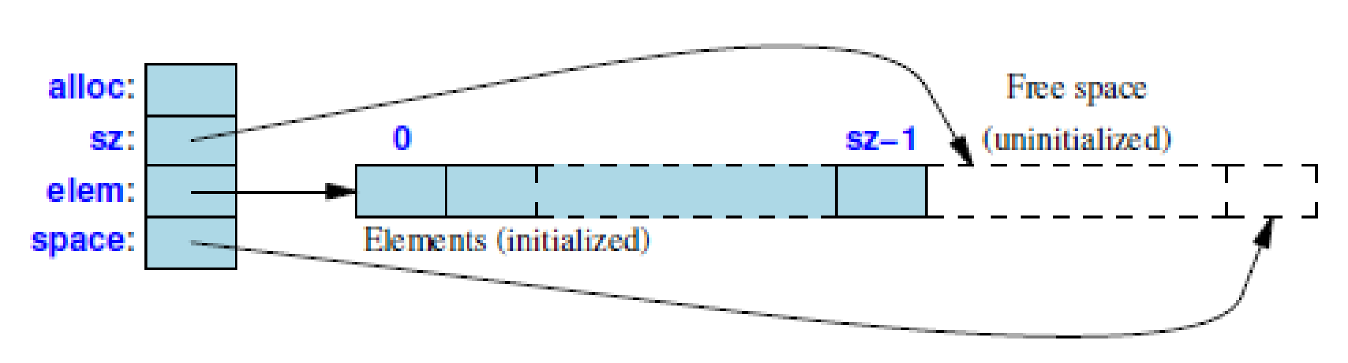
- We don’t always want to get our memory from **new**
  - Many, important applications rely on their containers using specialized me memory pools

```
template<typename T>
class allocator {
public:
        // ...
        T* allocate(int n);                          // allocate space for n objects of type T
        void deallocate(T* p, int n);        // deallocate n objects of type T starting at p
};

<Element T, typename A = allocator<T>>
class Vector {
        A alloc;            // use alloc to handle memory for elements
        // ...
};
```

Element construction and destruction

- Every element must be constructed before use
  - Like every object
- Every element of a type with a destructor must be destroyed after its last use
  - Like every object of a type with a destructor



- The space for elements are acquired from the Vector's allocator
  - And must be given back to that allocator when the Vector is done with it

reserve()

- The key operation for relocating elements into a new and larger space

```
template<Element T, Allocator A>
void Vector<T,A>::reserve(int newalloc)
{
        if (newalloc<=space)                        // never decrease allocation
                return;
        T* p = alloc.allocate(newalloc);            // allocate new space
        uninitialized_move(elem,&elem[sz],p);       // move elements into uninitialized space
        destroy(elem,space);                        // destroy old elements
        alloc.deallocate(elem,capacity());          // deallocate old space
        elem = p;
        space = newalloc;
}
```

push_back()

- Given **reserve()**, **push_back()** is very simple

```
template<Element T, Allocator A>
void Vector<T,A>::push_back(const T& val)
{
	reserve((space==0) ? 8 : 2*space);		// get more space
	construct_at(&elem[sz],val);			// construct val at end (using a standard-library function)
	++sz;						// increase the size
}
```

resize()

- Again, all the hard work is done in **reserve()**

```
template<Element T, Allocator A>
void Vector<T,A>::resize(int newsize, T val = T())
{
	reserve(newsize);
	if (sz<newsize)
		uninitialized_fill(&elem[sz],&elem[newsize],val);	// initialize added elements to val
	if (newsize<sz)
		destroy(&elem[newsize],&elem[sz]);		// destroy surplus original elements
	sz = newsize;
}
```

Range checking

```
// an almost real Vector of Ts:
template<class T> class Vector {
  T* elem;
  // ...
  T& operator[ ](int n) { return elem[n]; }          // access not range checked
  // ...
};

Vector<int> v(10);
for (int i; cin>>i; )
  v[i] = 7;               // horror! Maul arbitrary memory location
```

- We need to do something about that!

Range checking

```
void fill_vec(vector<int>& v, int n)          // initialize v with factorials
{
   for (int i=0; i<n; ++i)
         v.push_back(factorial(i));
}

int main()
{
   vector<int> v;
   try {
         fill_vec(v,10);
         for (int i=0; i<=v.size(); ++i)
               cout << "v[" << i << "]==" << v[i] << '\n';
   }
   catch (out_of_range) {             // we'll get here (why?)
         cout << "out of range error";
         return 1;
   }
}
```

Vector range checking – first attempt

```
template<Element T, Allocator A = allocator<T>>
class Vector {
        // ...
        T& at(int n) {                                          // checked access
                if (0<n || size()<=n) throw out_of_tange{};
                return elem[n];
        }

        T& operator[](int n) { return elem[n]; }                // unchecked access
        // ...
};

Vector v(100);
V[100] = 7;                 // error not caught
V.at(100) = 7;              // error caught
```

Vector range checking – first attempt

- Rather messy, error-prone
- The standard-library vector guarantees checking only for **at()**
  - Use a standard-library implementation that check’s **[ ]** ; every major implementation has one
  - Use PPP’s vector; it checks
- Why doesn’t the standard guarantee checking?
  - Checking cost in speed and code size
    - Not much; don’t worry about that cost
      - No student project needs to worry
      - Few real-world projects need to worry
  - Some projects need optimal performance
    - Think huge (e.g., server farm) and tiny (e.g., smart watch)
  - The standard must serve everybody
    - You can build checked on top of optimal
    - You can’t build optimal on top of checked
  - Some projects are not allowed to use exceptions
    - Old projects with pre-exception parts
    - High reliability, hard-real-time code (think airplanes)
  - That design was approved in the 1990s
    - The world was different than

PPP::Checked_vector

- This is what PPP gives you (called **vector** in user code)

```
namespace PPP {
        template<Element T>                                    // constrain element types
        struct Checked_vector : public std::vector<T> {
                using size_type = typename std::vector<T>::size_type;
                using value_type = T;
                using vector<T>::vector;                       // use vector<T>'s constructors

                T& operator[](size_type i)
                {
                        return std::vector<T>::at(i);// here, we check
                }
                // ...
        }; // Checked_vector
        // ... checked span and string ...
} // namespace PPP
```

Resources and exceptions

- A resource is something that we must acquire and later release
  - If resources aren't released the system will eventually grind to a halt
  - If resources are held for too long the system slows down
- Examples of resources
  - Memory
  - Locks
  - File handles
  - Thread handles
  - Sockets
  - Windows
  - Shaders in graphics systems
- We are not good at returning/releasing resources
  - Release must be made implicit

No concern about resources

- Asking for trouble

```
void suspicious(int s, int x)
{
        int* p = new int[s];            // acquire memory
        // ...
        if (x==0)
                return;                         // we may return
        if (0<x)
                p[x] = v.at(x);                 // subscripting may throw
        // ...
        delete[] p;                             // release memory
}
```

- Imagine if this was in a large function with complex control structures
  - First: avoid large functions with complex control structures

Ad hoc resource management

- Naïve ad hoc patch

```
void suspicious(int s, int x)                    // messy code
{
        int* p = new int[s];            // acquire memory
        vector<int> v;
        // ...
        try {
                if (x)
                        p[x] = v.at(x);          // subscripting may throw
                // ...
        }
        catch (...) {                            // catch every exception
                delete[] p;                      // release memory
                throw;                           // re-throw the exception so that a called can handle the error
        }
        // ...
        delete[] p;                              // release memory
}
```

Ad hoc resource management doesn't scale

- How many **try-catch** clauses do you need to get this right?

```
void suspicious(int s)
{
        int* p = new T[s];
        // ...
        int* q = new T[s];
        // ...
        delete[] p;
        // ...
        delete[] q;
}
```

Resource management

- Simple, general solution
  - RAII: “Resource Acquisition is initialization”
  - Also known as scoped resource management

```
void f(vector<int>& v, int s)
{
        vector<int> p(s);
        vector<int> q(s);
        // . . .
}  // vector's destructor releases memory upon scope exit
```

RAII (Resource Acquisition Is Initialization) for **Vector**

- Vector
  - acquires memory for elements in its constructor
  - Manage it (changing size, controlling access, etc.)
  - Gives back (releases) the memory in the destructor
- This is a special case of the general resource management strategy called RAII
  - Also called “scoped resource management”
  - Use it wherever you can
  - It is simpler and cheaper than anything else
  - It interacts beautifully with error handling using exceptions
  - Examples of resources:
    - Memory, file handles, sockets, I/O connections (iostreams handle those using RAII), locks, widgets, threads.

Naïve constructor

- Leaks memory and other resources
  - but does *not* create bad vectors

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
  :alloc{a}, sz{n}, space{n}           // copy allocator, set size, no added spare space
{
  v = alloc.allocate(n);               // get memory for elements
  for (T* p = v; p!=v+sz; ++p)
          a.construct_at(p,val);       // copy val into elements
}
```

- What happens if the allocation fails?
- What happens if an element constructor fails?

std::unititialized_fill()

- offers the strong guarantee:
  - either the initialization succeeds or no change

```
template<class For, class T>                    // a standard-library algorithm
void uninitialized_fill(For beg, For end, const T& val)
{
  For p;
  try {           // construct elements:
        for (p=beg; p!=end; ++p)
              a.construct_at(&*p,val);           // construct val in *p
  }
  catch (…) {     // undo construction:
        for (For q = beg; q!=p; ++q)
              a.destroy_at(&*q)                  // destroy
        throw;                                   // rethrow
  }
}
```

Naïve constructor (2)

- Better, but it still leaks memory

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
   :alloc{a}, sz{n}, space{n}          // copy allocator, set size, no added spare space
{
   v = a.allocate(n);                  // get memory for elements
   uninitialized_fill(v,v+sz,val);     // copy val into elements
}
```

RAII for **Vector**

- Represent ideas at types
  - So, we define a type to represent “the memory used by a **Vector**”
  - It deal only with memory, not with typed objects

```
template<typename T, typename A = allocator<T>>
struct Vector_rep {
        A alloc;                        // allocator
        int sz;                         // number of elements
        T* elem;                        // start of allocation
        int space;                      // amount of allocated space

        Vector_rep(const A& a, int n)
                : alloc{ a }, sz{ n }, elem{ alloc.allocate(n) }, space{ n } { }
        ~Vector_rep() { alloc.deallocate(elem, space); }
};
```

RAII for **Vector**

- Now **Vector** deals only with turning memory into objects (and back again)
  - And controlling access to those objects

```
template<typename T, typename A = allocator<T>>
class Vector {
        Vector_rep<T,A> r;
public:
        Vector() : r{A{},0} { }

        explicit Vector(int s, const T val = T{})
                :r{A{},s}
        {
                uninitialized_fill(elem,elem+sz,val); // elements are initialized
        }
        // ...
}
```

reserve()

```
template<Element T, Allocator A>
void Vector<T,A>::reserve(int newalloc)
{
	if (newalloc<=r.space)                      // never decrease allocation
		return;

	T* p = alloc.allocate(newalloc);            // allocate new space
	uninitialized_move(elem,&elem[sz],p);       // move elements into uninitialized space

	destroy(elem,space);                        // destroy old elements
	alloc.deallocate(elem,capacity());          // deallocate old space

	elem = p;
	space = newalloc;
}
```

resize()

```
template<Element T, Allocator A>
void Vector<T,A>::resize(int newsize, T val = T{})
{
	reserve(newsize);
	if (sz<newsize)
		uninitialized_fill(&elem[sz],&elem[newsize],val);	// initialize added elements to val
	if (newsize<sz)
		destroy(&elem[newsize],&elem[sz]);		// destroy surplus original elements
	sz = newsize;
}
```

Resource management

- But what about functions creating objects?
  - Traditional, error-prone solution: return a pointer

```
vector<int>* make_vec()          // make a filled vector
{
        vector<int>* p = new vector<int>;  // we allocate on free store
        // . . . fill the vector with data; this may throw an exception . . .
        return p;
}

auto q = make_vec();

// now users must remember to delete q
// they will occasionally forget: leak!
```

Return by moving

- Thanks to move semantics and copy elision returning a container is cheap
  - Just return by value

```
vector<int> make_vec()          // make a filled vector
{
        vector<int> res;                    // local variable with elements on free store
        // . . . fill the vector with data; this may throw an exception . . .
        return res;                 // moves the vector; doesn't copy elements
}

auto v = make_vec();

// No manual resource management
```

unique_ptr

- But what about functions that create polymorphic objects?
  - use **std::unique_ptr**; it's a pointer but it handles deletion

```
unique_ptr<Shape> read_shape(istream& is)
{
        // ... read a variety of shapes ...
        switch (p)  {
        case read_a_circle:        return make_unique<Circle>(center,radius);
        case read_a_triangle:     return make_unique<Triangle>(p1,p2,p3);
        // ...
        }
}

auto q = read_shape(ifile);
```

*// users don't have to **delete**; no **delete** in user code*
*// a **unique_ptr** owns its object and deletes it automatically*

shared_ptr

- But what if we need to share the polymorphic object?
  - use **std::shared_ptr;** *there can be many* **shared_ptr***s to an object;*
  - *the last* **shared_ptr** *destroys destroys the object*

```
shared_ptr<Shape> read_shape(istream& is)
{
	// ... read a variety of shapes ...
	switch (p)  {
	case read_a_circle:		return make_shared<Circle>(center,radius);
	case read_a_triangle:	return make_shared<Triangle>(p1,p2,p3);
	// ...
	}
}

auto q = read_shape(ifile);
other_fct(q);

// users don't have to delete; no delete in user code
```

Next lecture

- An introduction to the STL, the containers and algorithms part of the C++ standard library. Here we'll meet sequences, iterators, and containers.
- Further lectures will introduce more containers (such as **list** and **map**) and algorithms on containers and other sequences.