Conception Objets Avancée

Exceptions and templates

Giuseppe Lipari

CRIStAL - Université de Lille

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

Try/catch

- An exception object is *thrown* by the programmer in case of an error condition
- An exception object can be caught inside a try/catch block

```
try {
     //
    // this code can generate exceptions
    //
} catch (ExcType1& e1) {
    // all exceptions of ExcType1 are handled here
}
```

- If the exception is not caught at the level where the function call has been performed, it is automatically forwarded to the upper level (calling function)
    - Until it finds a proper try/catch block that catches it
    - or until there is no upper layer (in which case, the program is aborted)

More catches

- It is possible to put more catch blocks in sequence
  - they will be processed in order, the first one that catches the exception is the last one to execute

```
try {
   //
   // this code can generate exceptions
   //
} catch (ExcType1&e1) {
   // all exceptions of ExcType1
} catch (ExcType2 &e2) {
  // all exceptions of ExcType2
} catch (...) {
  // every exception
}
```

Re-throwing

- It is possible to re-throw the same exception to the upper layers

```
catch(...) {
  cout << "an exception was thrown" << endl;
  // Deallocate your resource here, and then rethrow
  throw;
}
```

Terminate

- In case of abort, the C++ run-time will call `terminate()`, which calls `abort()`
  - It is possible to change this behaviour

```
void terminator() {
  cout << "I'll be back!" << endl; exit(0);
}
void (*old_terminate)() = set_terminate(terminator);
```

- see terminate.cpp

Inheritance

```
double mylog(int a)
{
  if (a < = 0) throw LogErr();
  else return log(double(a));
}

void f(int i)
{
  mylog(i);
}

...

try {
  f(-5);
} catch(MathErr &e) {
  cout << e.what() << endl;
}
```

- This code will print *Log of a negative number - log module*
- you can also pass any parameter to `LogErr`, such as the number that cause the error, or the name of the function which caused the error, etc.

Exception specification

- Before C++17, it was possible to specify which exceptions a function might throw, by listing them after the function prototype
  - By doing so, exceptions were part of the interface!

```
void f(int a) throw(Exc1, Exc2, Exc3);
void g();
void h() throw();
```

- `f()` can only throw exception `Exc1`, `Exc2` or `Exc3`
- `g()` can throw any exception
- `h()` does not throw any exception

Exception specification

- Exception specification has been deprecated since C++11 and removed altogether in C++17
    - it was found that it caused more problems than solutions
- Now, you can only specify keyword `noexcept` to declare a function that does not throw any exception

```
void h() throw();   // still ok, but not recommended
void g() noexcept;  // recommended !!
```

Stack unrolling

```
void f() {
   A a;

   if (cond) throw Exc();
}

void g() {
   A *p = new A;

   if (cond) throw Exc();

   delete p;
}
```

At this point, a is destructed

Stack unrolling

```
void f() {
   A a;

   if (cond) throw Exc();
}

void g() {
   A *p = new A;

   if (cond) throw Exc();

   delete p;
}
```

At this point, a is destructed

memory pointed by `p` is **not** automatically deallocated

Resource management

- When writing code with exceptions, it's particularly important that you always ask yourself:

  If an exception occurs, will my resources be properly cleaned up?

- Most of the time you're fairly safe,
- but in constructors there's a particular problem:
  - if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object.
  - If an exception occurs in the constructor, the destructor doesn't get a chance to deallocate the resource.
  - Thus, you must be especially diligent while writing your constructor.
- see `examples/rawp.cpp`

How to avoid the problem

- To prevent such resource leaks, you must guard against these “raw” resource allocations in one of two ways:
  1. You can catch exceptions inside the constructor and then release the resources
  2. You can place the allocations inside an object’s constructor, and you can place the deallocations inside an object’s destructor.
- The last technique is called Resource Acquisition Is Initialization (RAII for short) because it equates resource control with object lifetime.
- see `examples/exception_wrap.cpp`

Automatic wrapper

- Dynamic memory is the most frequent resource used in a typical C++ program,
- the new C++11 standard provides an RAII wrapper for pointers to heap memory that automatically frees the memory.
- The unique_ptr class template, defined in the `<memory>` header, has a constructor that takes a pointer to its generic type
- The `unique_ptr` class template also overloads the pointer operators `*` and `->` to forward these operations to the original pointer
- So you can use the `unique_ptr` object as if it were a raw pointer

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

Containers

- Consider the problem of providing a generic container of objects
  - We designed and developed a Stack class container
  - It is an object that *contains* other objects, and provides operations for inserting, extracting, finding object, and visiting them in a certain order
  - Our stack class contains integers
  - However, the code is generic enough and depends only in minimal part from the fact that it contains integers
- Problem:
  - How to extend it to contains other types of objects?
  - In Java you would use *generics*, in C++ you use templates

Cut & Paste, pointer to void *

1. In the early days of programming the solution would have been Copy & Paste the code
   - then, modify it to use the desired class instead of `int`
   - Can you enumerate the problems with this approach?
2. Another possibility is store pointers to void
   - This is the C-style approach
   - Also, Java before generics were introduced
   - There is no type checking, and the need to typecast to the correct type when accessing the object

Templates

- Templates are used for generic programming
  - The general idea is: what we want to reuse is not only the abstract concept, but the algorithm
- with templates we reuse algorithms by making them general
- Example: consider the code for swapping two integers

```
void swap(int &a, int &b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...
int x=5, y=8;
swap(x, y);
```

- can we make it generic ?

Solution

```
template<class T>
void swap(T &a, T &b)
{
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...

int x=5, y=8;
swap<int>(x, y);
```

- Apart from the first line, we have just substituted the type `int` with a generic type `T`

How does it work?

- The template mechanism resembles the macro mechanism in C
- with macros:

```
#define swap(type, a, b) { type tmp; tmp=a; a=b; b=tmp; }
...
int x = 5; int y = 8;

swap(int, x, y);
```

- the substitution is done by the pre-compiler
  - it is a sort of Cut & Paste
  - can lead to problems if not the correct type, if you pass expressions, etc.
- with templates, the substitution is done by the compiler
  - it is like code generation on the fly
  - template are easier to use and much safer

Code duplicates

- The compiler instantiates a version of `swap()` with integer as an internal type
- if you call `swap()` with a different type, the compiler will generate a new function
- The code is generated only if the template is instantiated
    - If we do not use `swap()`, the code is never generated, even if we include it!
    - So, if there is an error in the function, you will never obtain a compilation error if you do not instantiate it.

Swap for other types

- What happens if we call swap for a different type:

```
class A { ... };
A x;
A y;
...

swap<A>(x, y);
```

  - A new version of swap is automatically generated
  - Class A must support default constructor and the assignment operator, otherwise the generation fails to compile

- see examples/swap.cpp

Type deduction

- Parameters can be automatically deducted by the compiler

```
int a = 5, b = 8;

swap(a, b);   // equivalent to swap<int>(a, b);
```

- The rules are not always so straightforward,
  - We will dedicate a section of this course on type deduction later on

Parameters

- A template can have any number of parameters
- A parameter can be:
  - a class, or any predefined type
  - a function
  - a constant value (a number, a pointer, etc.)

```
template<T, int sz>
class Buffer {
   T v[sz];
   int size_;
public:
   Buffer() : size_(0) {}
};
...
Buffer<char, 127> cbuf;
Buffer<Record, 8> rbuf;
int x = 16;
Buffer<char, x> ebuf; // error!
```

Default values

- A parameter can have a default value

```
template<class T, class Allocator = allocator<T> >
class vector;
```

- In this case, the vector class has two parameters
  - `T` is the type of object contained in the vector
  - `Allocator` is the algorithm (function) used to allocate the memory
    - function `allocator` is itself a template function that takes as parameter the type `T`

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

STL

- The Standard Template Library is normally distributed with the compiler
- It contains generic code (templates), including:
  - containers (vector, list, deque, map, set)
  - algorithms (sort, foreach, etc.)
  - I/O streams (cout, cin, fstreams, etc.)
  - strings
  - . . . and much more

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

out of range: raises exception

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

out of range: raises exception

copies the entire vector

Vector of objects

- Vector requires the following basic properties of the template class
  - Copy constructor; (otherwise you cannot insert elements)
  - Assignment operator; (otherwise you cannot return an object)
- It is possible to pre-allocate space for the vector:
  - This is used to avoid excessive allocation overhead when we have an idea of the size we need

```
vector<MyClass> v(10); // reserves 10 elements
```

Iterators

- Iterators are a generic way to access elements in a container, according to a predefined order
- The iterator is usually a class provided by the container itself
- It can be seen as a *pointer* to the elements of the container
  - `begin()` returns an iterator to the first element
  - `end()` returns the iterator pointing *beyond* the last element of the array
  - it is possible to use `++` and `--` to increment/decrement the iterator (i.e. move to the next/previous element)
  - it is possible to access the *pointed element* by using the dereferencing `operator*()`

```
vector<int> v;
vector<int>::iterator i;

for (i = v.begin(); i != v.end(); i++) cout << *i;
```

Iterators

```
int a[4] = {2, 4, 6, 8};
vector<int> v = {2, 4, 6, 8};

// visit the container with indexes
for (int i=0; i<4; i++) cout << a[i];
cout << endl;
for (int i=0; i<4; i++) cout << v[i];
cout << endl;

// visit the container with pointers/iterators
for (int *p=a; p!=&a[4]; p++) cout << *p;
cout << endl;
for (vector<int>::iterator q=v.begin();
     q != v.end(); q++) cout << *q;
cout << endl;

// visit with iterators
vector<int>::iterator q = v.end();
do {
    q--; cout << *q;
} while (q != v.begin());
cout << endl;
```

Why iterators ?

- Iterators are available for all containers in the standard library, with the same uniform interface
- They represent a simple and uniform way to visit a container
- Many template functions and member functions accept iterators parameters
- see `examples/iterator-example2.cpp`
- Exercise: generalise function print, so that it can print the content of vectors of any type
- solution: `examples/iterator-example3.cpp`

Use non-member begin() and end()

- `begin()` is a member of any container in the std lib
- however, it is also a global function in namespace standard

```
vector<int>::iterator i = v.begin();
// is equivalent to
i = begin(v);
```

- you should use the global `std::begin()` and `std::end()` because they work also on non standard containers, like C-arrays

```
char *array[10] = "My Array";

auto p = begin(array);
while (p != end(array) ) cout << *p;
cout << endl;
```

- (we will see auto in the next slide)

Auto

- the keyword auto tells the compiler to automatically deduce the type:

```
char *array[10] = "My Array";

auto p = begin(array);
```

  in this case, the type of p is char *, because an iterator to a standard array is a pointer to the elements of the array

```
vector<int> v = {1, 3, 5, 7, 11, 13, 17, 19};

auto p = begin(v);
```

  in this case, the type of p is vector<int>::iterator.
- We will analyse the rules for deducing the type later on
- For the moment, let's observe that the use of auto and begin() reduces the number of characters to type, and makes the code more generic

Vector internal implementation

- The vector is internally implemented as a variable size array
    - Therefore, internally it allocates and deallocates memory depending on the current number of elements
    - However, all elements are stored sequentially in memory
    - Therefore, when you perform an insertion in the middle using function member `insert()`, it simply moves all elements one step ahead to make space for the additional element to be inserted in the right place
    - Similarly, a `push_back()` may imply a copy of all elements!
    - Therefore, insertion in a vector is a costly operation which potentially takes $O(n)$.

Lists

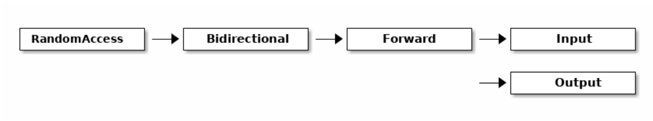
- The STL also provides the simple linked list

```
list<int> lst;
for (int i=0; i<10; i++)
    lst.push_back(i);

// going through all elements
auto i = begin(lst);
int sum = 0;
while (i!=end(lst)) {
    sum += *i;
    i++;
}
```

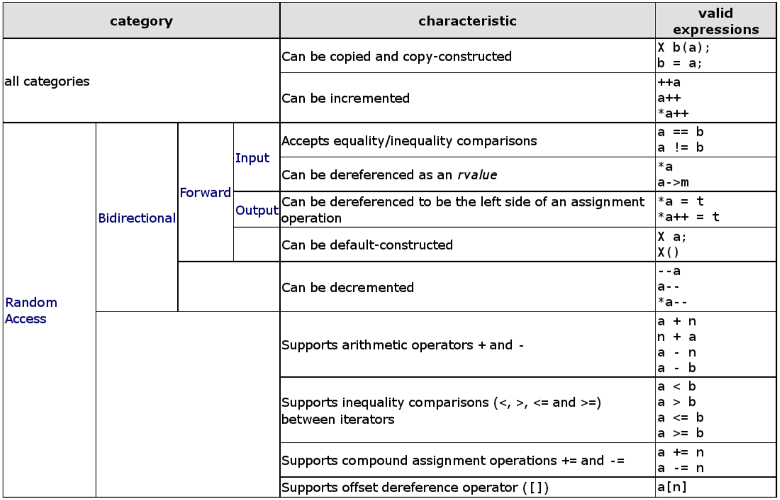
Iterator Types

- There are five types of iterators, depending on the functionality they provide:



- The difference consists in the type of operations that are supported:
  - all types support operator `++` and `*`
  - *input* supports copy construction and assignment, operator `->`, equality `==` and inequality `!=`
  - *output* supports assignment as *lvalue* (to the left of an assignment operator)
  - *forward* is as *input* and *output*, but also supports default constructor
  - *bidirectional* is as *forward*, and it also supports operator `--`
  - *random* is as *bidirectional*, and it also supports operators `+`, `-`, `+=`, `-=`, comparison (`<`, `<=`, `>=`, `>`), offset `[]`

Comparative Table



category				characteristic	valid expressions

all categories				Can be copied and copy-constructed	`X b(a);`
					`b = a;`

all categories				Can be incremented	`++a`
					`a++`
					`*a++`

Random Access	Bidirectional	Forward	Input	Accepts equality/inequality comparisons	`a == b`
					`a != b`

Random Access	Bidirectional	Forward	Input	Can be dereferenced as an *rvalue*	`*a`
					`a->m`

Random Access	Bidirectional	Forward	Output	Can be dereferenced to be the left side of an assignment operation	`*a = t`
					`*a++ = t`

Random Access	Bidirectional	Forward		Can be default-constructed	`X a;`
					`X()`

Random Access	Bidirectional			Can be decremented	`--a`
					`a--`
					`*a--`

Random Access				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

Random Access				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

Random Access				Supports compound assignment operations += and -=	`a += n`
					`a -= n`

Random Access				Supports offset dereference operator ([])	`a[n]`

STL iterators

- All STL containers support at least bidirectional iterators
  - list, deque
- some support random iterators:
  - vector, map
- Of course, iterators are compatible: I can copy from a list to a vector, and viceversa

```
int main()
{
    vector<int> v1 = {1,3,5,7,11,13,17,19};
    list<int> l1(v1.rbegin(), v1.rend());   // copy of v1
    vector<int> v2;

    // copies l1 into v2
    copy(l1.begin(), l1.end(), back_inserter(v1.begin()));
}
```

Associative arrays

- An associative array generalizes the concept of array
  - Two subtypes: sets and maps
- `set<key>` and `multiset<key>` contain ordered sets of objects
  - in `set<key>` the key must be unique
  - in `multiset<key>`, the same key can be inserted several times
- `map<key,value>` and `multimap<key,value>` contain pairs `<key,value>`, where `key` is the *index* in the array
  - in `map<key, value>`, each different key must be associated one unique value
  - in `multimap<key, value>`, several values can be associated to the same key

Map example

```
int main()
{
    map<string, int> age;

    age["Peppe"] = 40;
    age["Roberto"] = 25;
    age["Giovanna"] = 30;

    pair<string, int> elem = {"Pippo", 32};

    cout << elem.first << " = " << elem.second << endl;

    age.insert(elem);

    map<string, int>::iterator i;
    for (i = age.begin(); i != age.end(); i++)
        cout << i->first << " = " << i->second << endl;
    ...
}
```

- see examples/map_example.cpp

The typename keyword

- Typename is used to denote that the following element is a type:

```
template<class T> class X {
    typename T::id i;    // Without typename, this is an error!
public:
    void f() { i.g(); }
};

class Y {
public:
    class id {
    public:
        void g() {}
    };
};

int main() {
    X<Y> xy;
    xy.f();
}
```

Typical use of typename - I

- Why it is an error to not use typename ?
  - long answer here
  - short answer: `id` is a dependent name of the type T
  - when the compiler processes the code, it does not know yet what `id` is
    1. is it a variable? in which case, the code is not syntactically
    2. is it a type? in which case, the code is correct
    3. is it another template? in which case the compiler has no idea
  - To help the compiler, we specify the second option with `typename`
- Where do I need to use `typename`?
  - when writing template code which uses a dependent name;
  - as a replacement for `class` in the template parameter specification

Typical use of typename - II

- The typical example of use is for iterators:

```
template <typename T> printSeq(const T &container) {
    typename T::const_iterator i;
    for (i = begin(container); i != end(container); ++i) {
       cout << *i << ", ";
    }
    cout << endl;
}
```

- We can write this code more easily by using auto:

```
template <typename T> printSeq(const T &container) {
    for (auto i = begin(container); i != end(container); ++i) {
       cout << *i << ", ";
    }
    cout << endl;
}
```

Member template

- You can make a member template

```
template<typename T> class complex {
public:
  template<class X> complex(const complex<X>&);
  ...
};

complex<float> z(1, 2);
complex<double> w(z);
```

  - In the declaration of `w`, the complex template parameter `T` is double and `X` is automatically deduced as float. Member templates make this kind of flexible conversion easy.

Another example

```
int data[5] = { 1, 2, 3, 4, 5 };
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

- As long as the elements in `v1` are assignment-compatible with the elements in `v2` (as `double` and `int` here), all is well.
- the vector class template has the following member template constructor:

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
    const Allocator& = Allocator());
```

- `InputIterator` is deduced as `vector<int>::iterator`

Restrictions

- Member template functions cannot be declared virtual.
  - Current compiler technology expects to be able to determine the size of a class's virtual function table when the class is parsed.
  - Allowing virtual member template functions would require knowing all calls to such member functions everywhere in the program ahead of time.
  - This is not feasible, especially for multi-file projects.

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

Functions

- The standard template library defines many function templates in the header `algorithm`
  - sort, find, accumulate, fill, binary_search, copy, etc.

- An example

```
#include <algorithm>
...
int i, j;
...
int z = min<int>(i, j);
```

- Usually, type can be deduced by the compiler, but it is sometimes impossible to make the right choice

```
double x; int j;

int z = min(x, j); // x is a double, error, not the same types
int z = min<double>(x, j); // this one works fine
```

Return type

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
  return (a < b) ? a : b;
}
```

- The problem is: which return value is the most correct? `T` or `U` ?
- If the return type of a function template is an independent template parameter, you must always specify its type explicitly when you call it, since there is no argument from which to deduce it.

Example of std functions

```
using namespace std;

vector<int> vec = {4, 3, 1, 9, 5, 5, 10, 15, 3, 7, 9};

sort(begin(vec), end(vec));

// sums all elements
int s = accumulate(begin(vec), end(vec));

// multiplies all elements
int m = accumulate(begin(vec), end(vec), multiplies<int>());

// sums all elements, except the first and the last
int s2 = accumulate(next(begin(vec)), prev(end(vec)));

// moves all duplicates to the end
auto it_end = unique(begin(vec), end(vec));

// delete duplicates
vec.erase(it_end, end(vec));
```

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

How to generalize Stack to any object

- Now, let's go back to our Stack class, and generalize it to contain any type of object
  - first, in the interface we simply substitute `int` with `T`

```
template <typename T>
class Stack {
     ...
public:
     Stack();
     Stack(const Stack &other);

     T top() const;
     void pop();
     void push(const T& elem);
     ...
};
```

Implementation

- Allocating the array is not so simple

```
template <typename T>
class Stack {
    T *array;
    ...
};

template<typename T>
Stack<T>::Stack() {
    array = new T[maxsize];
}
```

The new will create an array of maxsize objects of type T. So, the default constructor of T is called maxsize times.

- Two problems:
  1. If the type `T` has no default constructor, we cannot put it inside a Stack
  2. Actually, those objects are not really in the stack (the stack is initially empty!), they will be overwritten by the `push`

Allocation and construction

- The new operator can be see as two separate phases
  1. Allocation of memory
  2. Construction of the object on that memory
- In C++, we can separate the two phases into two different functions
  1. void *operator new(size_t size) does the allocation, and it is the equivalent of malloc in C
  2. the placement new operator T* operator new(ptr) type does the construction
- Example:

```
MyClass *p = new MyClass();
//can be translated as

// allocate memory
void *ptr = operator new(sizeof(MyClass));
// Construct the object
MyClass *p = new (ptr) MyClass();
```

  - see examples/operatornew.cpp

Allocation without construction

So, now we can write the constructor for our stack class:

```
template<typename T>
Stack<T>::Stack()
{
    // allocate maxsize * sizeof(T) bytes of memory, returns a void*
    // which is then casted into a T*
    // *does not call* the constructor for T
    array_ = static_cast<T*>(operator new(maxsize_*sizeof(T)));
}
template<typename T>
Stack<T>::~Stack()
{
    clear(); // removes all elements

    // deallocates memory without calling the destructor
    operator delete(array_);
}
```

- see the rest of the code on examples/stack_template.hpp

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

The rules

- The general case is

```
// definition
template<typename T>
void f(ParamType param);

// call
f(expr);
```

- The compiler examines expr to *compute* the types `T` and `ParamType`.
- For example:

```
template <typename T>
void f(const T& param);

int x:
f(x);
```

  - in this case,
    - `T` is `int`,
    - `ParamType` is `const int &`

Rules

- Three cases to consider
  1. `ParamType` is a pointer or a reference
  2. `ParamType` is an universal reference
  3. `ParamType` is neither a pointer, nor a reference
- We will see the meaning of universal reference later on
- We will now examine point 1 and 3

ParamType is a pointer or a reference

- Rules
  - If expr is a pointer (or a reference), the pointer (reference) part is removed from the type
  - Then, a "pattern matching" between the type of expr and the form of ParamType is performed to determine T
- Example:

```
template<typename T>
void f(T& param);

int x = 27;
const int cx = x;
const int &rx = x;
```

  - f(x) : T is int, ParamType is int &
  - f(cx) : T is const int, ParamType is const int &
  - f(rx) : T is const int, ParamType is const int &

Example 2

- Example:

```
template<typename T>
void f(const T& param);

int x = 27;
const int cx = x;
const int &rx = x;
```

  - `f(x)` : T is `int`, ParamType is `const int &`
  - `f(cx)` : T is `int`, ParamType is `const int &`
  - `f(rx)` : T is `int`, ParamType is `const int &`

ParamType is neither a pointer, nor a reference

In this case, the parameter is passed by value

- If expr is a reference, the reference part is removed from the type
- If expr is a const, the const part is removed from the type
- Then, a "pattern matching" between the type of expr and the form of ParamType is performed to determine T
- Example:

```
template<typename T>
void f(T param);

int x = 27;
const int cx = x;
const int &rx = x;
```

  - f(x) : both T and ParamType are int
  - f(cx) : both T and ParamType are int
  - f(rx) : both T and ParamType are int

Example 2

- Example:

```
template<typename T>
void f(T param);

const char * const ptr = "Hello";

f(ptr);
```

  - In this case, `T` and `ParamType` are both `const char *`
- Example:

```
const char name[] = "Lipari";

f(name);
```

  - once again, both `T` and `ParamType` are `const char *`

Some trick

How to obtain the size of a C array at compile time?

```
int primesLessThan20[] = {1, 2, 3, 5, 7, 11, 13, 17, 19};

template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept;

cout << arraySize(primesLessThan20) << endl;
```

std::array

- C++ tries to overcome some of the deficiencies of C
  - however, it also tries to maintain compatibility with C
- The idiomatic way to declare an array in *modern C++* is

```
// an array of 5 elements
std::array<int, 5> myarray;

// an array with the same size as the previous one
std::array<int, arraySize(primesLessThan20)> primes;
```

- You can use iterators and all sort of std functions on it.

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

The auto keyword

- auto is a new keyword for the C++11 standard that tells the compiler to automatically deduce the type of the following variable from the result of an expression

```
int func(int h){
    // code
   return 4*h;
}

auto x = func(y);
```

- You can use it whenever the type can be deduced by the compiler
  - You cannot use it to declare the parameter of a function!

auto and iterators

- One nice property of `auto` is that it reduces the need to use type specification when dealing with iterators

```
vector<int> vec;
vector<int>::iterator i = vec.begin(); // old writing

auto i = vec.begin(); // new writing
```

auto and templates

- auto can be useful also when dealing with template parameters:
  - suppose you have the following template function

```
template <typename BuiltType, typename Builder>
void makeAndProcessObject (const Builder& builder)
{
    BuiltType val = builder.makeObject();
    // do stuff with val
}
```

- This technique is called *builder* (it is actually a pattern)
- Notice that to declare variable val, we need to specify the type
- With the old standard, we have to pass two template parameters:
  - the Builder class
  - the type of the objects being built

auto and templates

- With C++11 we can avoid the built type:

```
template <typename Builder>
void makeAndProcessObject (const Builder& builder)
{
    auto val = builder.makeObject();
    // do stuff with val
}
```

Auto and return type

- In the new standard there is an alternative syntax for specifying the return type

```
int func(int x, double y); // classical syntax

auto func(int x, double y) -> int; // alternative syntax
```

- This is useful in combination with another construct, called *decltype*
  - `decltype` *extracts* the type from an expression

decltype example

- Suppose we want to write a template function that returns the 5-th element of a container
- The container type will be the template parameter
  - What type is contained within?

```
template <class C>
auto get_5th(const C &cont) -> decltype(*cont.begin())
{
    auto i = cont.begin();
    // move to the 5th element
    return *i;
}
```

- We do not know the return type before knowing the type `C`, so with auto+decltype we postpone the type deduction until after type `C` has been instantiated.
- Notice that `decltype` is resolved by the compiler at compile time, no run-time overhead

auto and references

- auto uses the same deduction rules as the template deduction
- What if the type of an expression is a reference?

```
int& foo();

auto bar = foo(); // is bar int or int& ??
```

- The answer is that bar is int ! (see rules above)
  - the compiler will always choose the copy semantic when it is most natural
- You can specify that you want a reference by writing the symbol near the auto

```
int& foo();

auto& bar = foo(); // bar is a reference
```

- You can also specify const to make the variable a constant.

Outline

Exceptions

Generic code and templates

Standard Template Library

STL algorithms

Example: the stack template

Template deduction

Auto

Range-based loops

Basic syntax

- Sometimes a for loop over a container can be boring to write. C++11 provides a convenient syntax:

```
vector<int> vec;
...
for (int i : vec)
    cout << i << ", ";
```

- Notice that `i` is not an iterator here, but the actual variable that holds the value of the elements inside the vector
- Of course, if the content has a strange type, you can also use auto:

```
map<string, int> m;
...
for (auto i : m)
    cout << i.first << ", " << i.second << endl;
```

References

- If you want to modify the content of the container, use references

```
map<string, int> m;
...
for (auto &i : m) i.second++;
```

- You can use range-based loops over any container that provides:
  - `begin()` and `end()` functions that return an iterator
  - The iterator must support pre-increment (`operator++()`), dereferencing (`operator*()`) and inequality (`operator!=()`)
- Therefore, you can write your own container that provides such functions, and use the new range-based loop with your container