Today C++

Speaker:
Jyrki

- Template basics
- STL concepts
- Generic programming techniques
- Efficiency of generic code
- Power of templates
- Static code generation

Main source used: Czarnecki and Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley (2000).

Generic Parameters

```
template <typename T>
T sqr(T x) {
  return x * x;
}

int main() {
  int a = 3;
  double b = 3.0;
  int c = sqr<int>(a);
  double d = sqr<double>(b);
}
```

In this case, the type `T` can be deduced from the context, because parameter `x` has type `T`, so in the function calls the type parameter is not necessary.

Default Template Parameters

```
template <typename T, typename A = std::allocator<T> >
class vector;
```

You may omit the second template parameter whenever you instantiate `vector`.

Member Templates

```
template <typename T1, typename T2>
class pair {
public:
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;
  pair()
    : first(T1()), second(T2()) {
  }
  pair(const T1& a, const T2& b)
    : first(a), second(b) {
  }

  template <typename U1, typename U2>
  pair(const pair<U1, U2>& p)
    : first(p.first), second(p.second) {
  }
};
```

Allows a `pair<T1, T2>` to be constructed from any `pair<U1, U2>`, provided that `U1` is convertible to `T1` and `U2` to `T2`.

Partial Specialization

```
template <typename T>
class X {
  // Most general version.
}

template <typename T>
class X<T*> {
  // Version for general pointers.
}

template <>
class X<void*> {
  // Version for one specific pointer type.
}
```

One specialization is more **specialized** than another if every argument that matches its specialization also matches the other, but not vice versa. The most specialized version is preferred over the others in declarations and in overload resolution.

Parametrized Inheritance

It is possible to turn the superclass of a class into a parameter.

```
template <typename superclass>
class someclass: public superclass {
  //...
}
```

Parametrized Components

`R`: random access iterator

`F`: type of the ordering function used in element comparisons

```
template <typename R>
void sort(R, R);

template <typename R, typename F>
void sort(R, R, F);
```

Nontype Template Parameters

We can define nontype template parameters, whose type is an integral type (e.g. `int`, `short`, `char`) or an enumeration type, but not a floating point type or a class type.

```
template <int arity, typename R, typename F>
class heap_policy;
```

In addition, templates can also take templates, pointers, or functions as parameters.

Also, `sizeof` is evaluated at compile time.

Pointer-to-Function Templates

```
inline double f(double x) {
  return 1.0 / (1.0 + x);
}

template<double T_function(double)>
double integrate(double a, double b, int numSamplePoints) {

  double delta = (b - a) / (numSamplePoints - 1);
  double sum = 0.0;

  for (int i = 0; i < numSamplePoints; ++i)
    sum += T_function(a + i * delta);

  return sum * (b - a) / numSamplePoints;
}

// ...
integrate<f>(1.0, 2.0, 100);
```

Compile-Time “Assignments”

`typedef`s are used to introduce new types from other types.

```
typedef unsigned int natural;

class someclass {
public:
  typedef unsigned int capacity;
  // ...
}
```

Member types can be used to propagate information between components at compile time.

Compile-Time "Variables"

The "variables" of static C++ code are `typedef`-names and integral constants. After the initialization the value cannot be changed. If you need a new type or value, you simply create a new type. Just as in functional programming, static C++ code uses **symbolic names** rather than true variables.

Compile-Time “Function Calls”

An inline static function of a `struct` can be seen as a compile-time function call.

```
struct less {
  template <typename T>
  static bool execute(const T& a, const T& b) {
    return a < b;
  }
};
```

Compile-Time "structs"

Traits classes are used to bundle different types together as their members.

```
template <typename P>
struct iterator_traits {
  typedef typename P::difference_type difference_type;
  typedef typename P::value_type value_type;
  typedef typename P::pointer pointer;
  typedef typename P::reference reference;
  typedef typename P::iterator_category iterator_category;
};

// specialization for pointers
template <typename T>
struct iterator_traits<T*> {
  typedef std::ptrdiff_t difference_type;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef typename std::random_access_iterator_tag iterator_
};

// specialization for pointers to const
template <typename T>
struct iterator<const T*> {
  typedef T value_type;
  typedef std::ptrdiff_t difference_type;
  typedef const T* pointer;
  typedef const T& reference;
  typedef typename std::random_access_iterator_tag iterator_
};
```

Real World: Compiler Errors

```
vector<int, std::allocator<int>>;

ambiguity1.cpp:1: ‘>>’ should be ‘> >’
```

```
template <class I, class T>
inline void fill(I first, I last, const T& val) {

   enum{ can_opt = boost::is_pointer<I>::value
                   && boost::is_arithmetic<T>::value
                   && (sizeof(T) == 1) };
   typedef detail::filler<can_opt> filler_t;
   filler_t::/* template */do_fill<I,T>(first, last, val);
}

ambiguity2.cpp: In function ‘void fill(I, I, const T &)’:
ambiguity2.cpp:7: parse error before ‘,’
```

```
template <typename element>
void f(element dummy) {
  /* typename */ element::type_or_method x;
  x = 0;
}

ambiguity3.cpp: In function ‘void f(element)’:
ambiguity3.cpp:3: parse error before ‘;’
```

Real World Code: `min`

```
template <typename E>
const E& min(const E&, const E&);
```

vs.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
```

Develop `min` that satisfies the following requirements:

1. Offers function call semantics (including type checking), not macro semantics.
2. Supports both `const` and non-`const` arguments (including mixing the two in a single call).
3. Supports arguments of different types where that makes sense.

Alexandrescu's Solution

```
template <class L, class R>
typename MinMaxTraits<L, R>::Result
min(L& lhs, R& rhs) {
  if (lhs < rhs)
    return lhs;
  return rhs;
}

template <class L, class R>
typename MinMaxTraits<const L, R>::Result
min(const L& lhs, R& rhs) {
  if (lhs < rhs)
    return lhs;
  return rhs;
}
```

... two more overloads ...

It would all be so nice, but there is a little detail worth mentioning. Sadly, `min` did not work with any compiler he had access to (in 2001). In fairness, each compiler choked on a different piece of code. For more details, see

Andrei Alexandrescu, Generic⟨Programming⟩: Min and Max Redivivus, *C++ Experts Forum*, April 2001. Available at `www.cuj.com/experts`.

STL

The Standard Template Library (STL) is now part of the ISO standard for C++ ratified in 1998.

Its main architect was Alexander A. Stepanov. The implementation written by him, Meng Lee, and David R. Musser was made freely available on the Internet in 1994.

algorithms	functors
iterators	adaptors
sequences	allocators

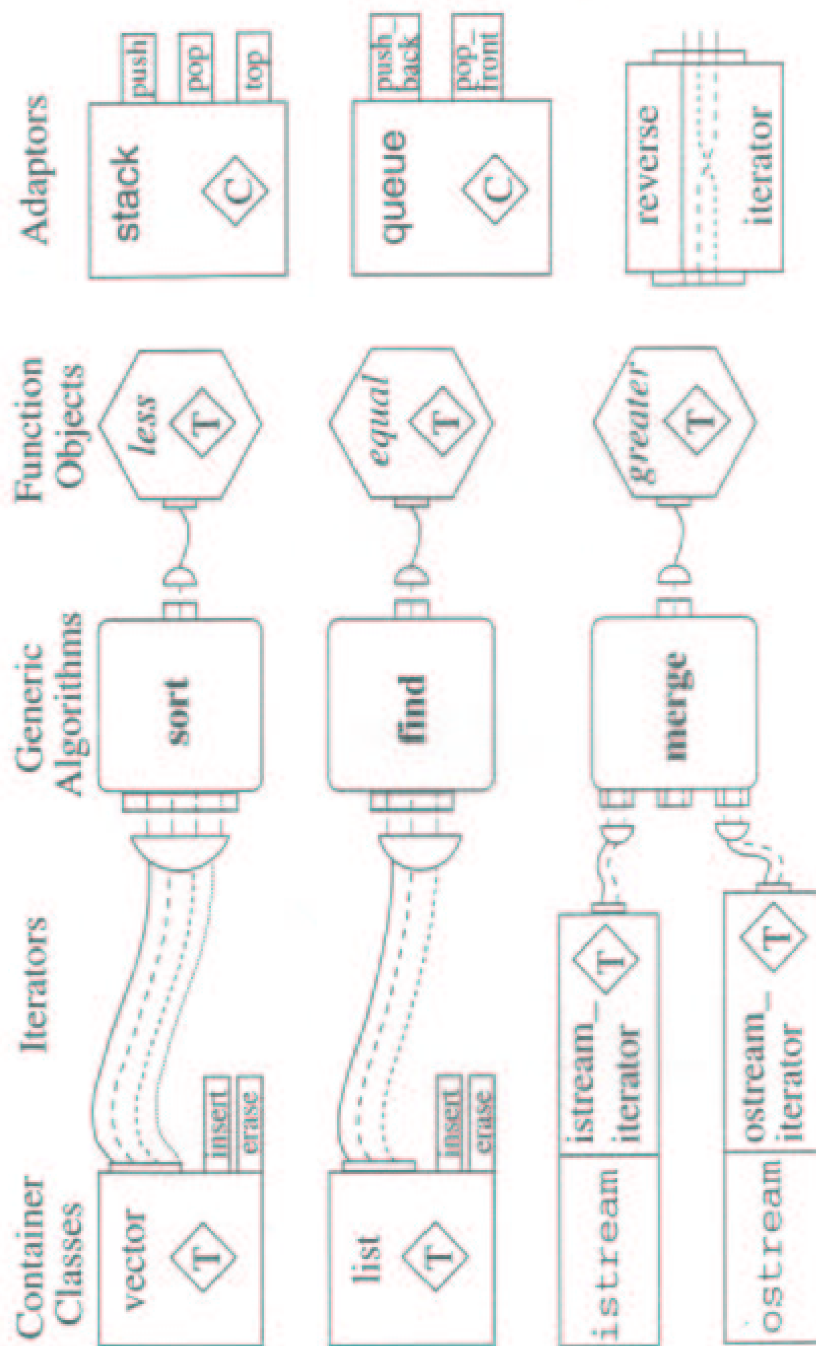


FIGURE 2-1. Five of the Six Major Categories of STL Components (Not Shown Are Allocators).

Source: David R. Musser, Gillmer J. Derge, and Atul Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 2nd Edition, Addison-Wesley (2001), Figure 2.1

Iterators

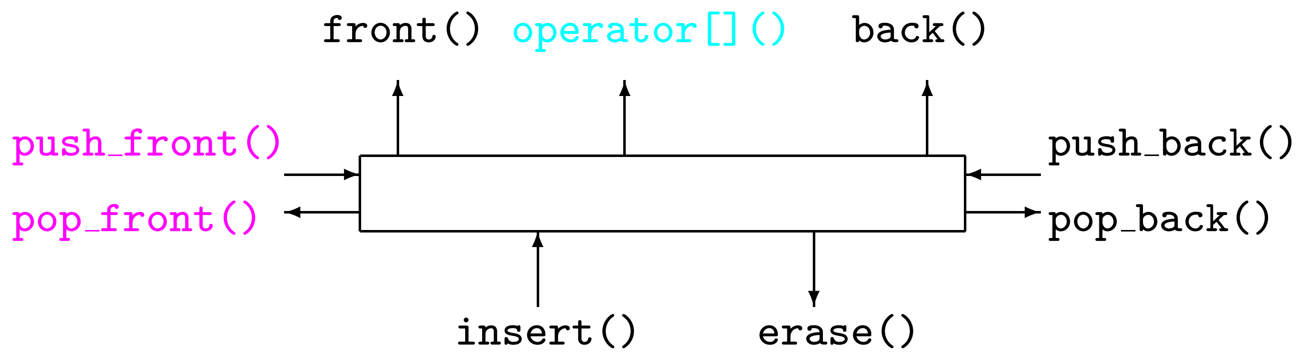
X iterator whose value type is T
p, q objects of type X
t object of type T

Category	Allowed expressions
input	X(p) (copy constructor)
	X p(q) (copy constructor)
	X p = q (copy constructor)
	p = q (assignment)
	p == q (equality)
	p != q (inequality)
	*p (read only once)
	p->m (equivalent to (*p).m)
	++p (preincrement)
	(void) p++ (postincrement)
output	X(p) (copy constructor)
	X p(q) (copy constructor)
	X p = q (copy constructor)
	p = q (assignment)
	*p = t (write only once)
	++p (preincrement)
	p++ (postincrement)

`i` object of `X`'s difference type

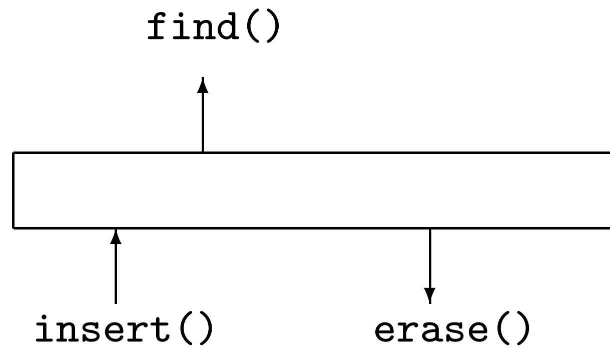
Category	Allowed expressions
forward	all earlier operations
	`X p` (default constructor)
	`X()` (default constructor)
	multiple reads and writes
bidirectional	all earlier operations
	`--r` (predecrement)
	`r--` (postdecrement)
random access	all earlier operations
	`p += i` (iterator addition)
	`p + i` (iterator addition)
	`i + p` (iterator addition)
	`p -= i` (iterator subtraction)
	`p - i` (iterator subtraction)
	`q - p` (difference)
	`p[i]` (equivalent to `*(p + i)`)
	`p < q` (less)
	`p > q` (greater)
	`p <= q` (less or equal)
	`p >= q` (greater or equal)

Sequences



- list
- vector
- deque

Associative Containers



- set
- multiset
- map $\langle$key, value$\rangle$
- multimap

Function Objects

A **function object**, or a **functor**, is a function pointer, or an object of any class that supports the operation `operator()`.

```
template <typename T>
class less {
public:
  bool operator()(const T& a, const T& b) {
    return a < b;
  }
};
```

Functors can be called as normal functions by writing `less(a,b)`.

For example, the `std::sort` function can take a functor, which defines an ordering on the set of elements, as its third parameter.

Adaptors

Iterator adaptors

- E.g., reverse iterators

Container adaptors

- queue
- priority queue
- stack

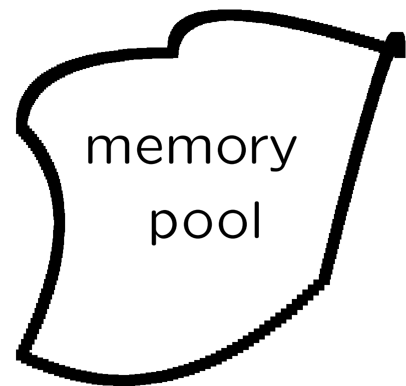
Function adaptors

- E.g., create a unary function from a binary function by fixing one of the parameters

Allocators

Make dynamic containers independent of the memory management.

`X` an allocator whose value type is `T`
`a` object of type `X`
`t` object of type `T`
`n` value of type `X::size_type`
`p` object of type `X::pointer`



`a.allocate(n)`
Allocates `n * sizeof(T)` bytes of memory

`a.deallocate(p,n)`
Deallocates the memory that `p` points to

`a.construct(p,t)`
Equivalent to `new ((void*) p) T(t)`

`a.destroy(p)`
Equivalent to `((T*) p)->~T()`

Generic `merge` Routine

```
#include <list>
#include <deque>
#include <algorithm>
#include <cassert>

template <typename sequence>
sequence make (const char s[]) {
  return sequence(&s[0], &s[std::strlen(s)]);
}

int main () {
  char* vowels = "aeiouy";
  int len = std::strlen(vowels);

  std::list<char> consonants =
    make<list<char> >("bcdfghjklmnpqrstvwxz");

  std::deque<char> alphabet(26, ' ');

  std::merge (
    &vowels[0], &vowels[len],
    consonants.begin(), consonants.end(),
    alphabet.begin()
  );

  assert(alphabet ==
    make< deque<char> >("abcdefghijklmnopqrstuvwxyz"));
  return 0;
}

shell> g++ merge.cpp
shell> a.out
```

Stepanov's contributions

"the task of the library designer is to find all interesting algorithms, find the minimal requirements that allow these algorithms to work, and organize them around these requirements"

[Stepanov 2001]

- Algorithm algebra
- Generic programming
- Programming with concepts
- Semi-formal specification of the components, including complexity requirements
- Generality so that every program works on a variety of types, including C++ built-in types
- Efficiency close to hand-coded, type-specific programs

Goals of the CPH STL Project

The purpose of the project is

- to study and analyse existing specifications for and implementations of the STL to determine the best approaches to optimization,
- to provide an enhanced edition of the STL and make it freely available on the Internet,
- to provide cross-platform benchmark results to give library users better grounds for assessing the quality of different STL components,
- to develop software tools that can be used in the development of component libraries, and
- to carry out experimental algorithmic research.

Abstraction

```
void* memcpy(void* to, const void* from, size_t n) {
  const char* first = (const char*)from;
  const char* one_past_the_end = ((const char*)from) + n;
  char* result = (char*)to;
  while (first != one_past_the_end)
    *result++ = *first++;
  return result;
}
```

Minimal requirements:

- traverse through the sequence using some sort of pointer,
- access elements pointed to,
- write the elements to the destination, and
- compare pointers to know when to stop.

Real World Code: `copy`

`I`: input iterator

`O`: output iterator

```
template <typename I, typename O>
O copy(I first, I one_past_the_end, O result) {
  while (first != one_past_the_end)
    *result++ = *first++;
  return result;
}
```

This is trivial, ikke os'.

copy.cpp File Reference

This file defines the function `copy`. More...

```
#include <iterator>
#include <type>
#include <cstring>
```

Go to the source code of this file.

Namespaces

namespace cphstl

Detailed Description

This file defines the function `copy`.

Original author

Jyrki Katajainen <jyrki@diku.dk>, December 2001

Sources

Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied,* Addison-Wesley (2001), see Section 2.10.5.

John Maddock and Steve Cleary, C++ type traits, *Dr. Dobb's Journal,* ***25****,10 (2000), 38-44.*

Generated at Mon Dec 17 00:52:10 2001 for TheCopenhagenSTL by doxygen 1.2.3 written by Dimitri van Heesch, © 1997-2000

copy.cpp

Go to the documentation of this file.

```
00001
00020
00021 #include <iterator> /* defines cphstl::iterator_traits */
00022 #include <type> /* defines cphstl::int2type and cphstl::query */
00023 #include <cstring> /* defines memcpy */
00024
00025 namespace cphstl {
00026
00027   namespace {
00028
00029     enum copy_algorithm_selector { conservative, fast };
00030
00033
00034     template <typename input_iterator, typename output_iterator>
00035     inline output_iterator
00036     copy (
00037       input_iterator p,
00038       input_iterator one_past_the_end,
00039       output_iterator r,
00040       cphstl::int2type<conservative>
00041     ) {
00042
00043       for (; p != one_past_the_end; ++p, ++r) {
00044         *r = *p;
00045       }
00046       return r;
00047     }
00048
00051
00052     template <typename input_iterator, typename output_iterator>
00053     inline output_iterator
00054     copy (
00055       input_iterator first,
00056       input_iterator one_past_the_end,
00057       output_iterator result,
00058       cphstl::int2type<fast>
00059     ) {
00060
00061       typedef std::iterator_traits<input_iterator>::difference_type size;
00062       size n = one_past_the_end - first;
00063       memcpy(result, first, n * sizeof(*first));
00064       return result + n;
00065     }
00066   }
00067
00144
00145   template <typename input_iterator, typename output_iterator>
00146   output_iterator
00147   copy (
00148     input_iterator first,
00149     input_iterator one_past_the_end,
00150     output_iterator result
00151   ) {
00152     typedef std::iterator_traits<input_iterator>::value_type input_element
00153     typedef std::iterator_traits<output_iterator>::value_type output_eleme
```

```
00154
00155     enum { algorithm_flag =
00156       cphstl::query<input_iterator>::is_pointer &&
00157       cphstl::query<output_iterator>::is_pointer &&
00158       cphstl::query<input_element>::is_fundamental &&
00159       cphstl::query<output_element>::is_fundamental &&
00160       sizeof(input_element) == sizeof(output_element) ? fast : conservativ
00161     };
00162     return copy(first, one_past_the_end, result, cphstl::int2type<algorith
00163   }
00164 }
```

Generated at Mon Dec 17 00:52:10 2001 for TheCopenhagenSTL by doxygen 1.2.3 written by Dimitri van Heesch, © 1997-2000

Real World Code: `greater`

Problem: Your generic program gets $<$ as an ordering function but you need $>$.

```
template <typename F>
class converse_relation
  : public std::binary_function <
      typename F::first_argument_type,
      typename F::second_argument_type,
      bool
    > {
protected:
  F less_;
public:
  explicit converse_relation(const F& less)
    : less_(less) {
  }

  bool operator() (
    const typename F::first_argument_type& x,
    const typename F::second_argument_type& y
  ) const {
    return less_(y, x);
  }
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```

explicit Keyword

```
class rational {
public:
  explicit rational(int n = 0, int d = 1): n(n), d(d) {
  }
  ...
};
```

The statement `r = 100;` is transformed into the following pseudocode:

```
rational _temp;
_temp.rational::rational(100,1); // <--
r.rational::operator=(_temp);
_temp.rational::~rational();
```

```
shell> g++ rational.cpp
rational.cpp: In function ‘int main()’:
rational.cpp:12: conversion from ‘int’ to non-scalar
type ‘rational’ requested
```

But

```
rational r(100);    // ok
r = rational(100); // ok
```

Real World Code: Heaps

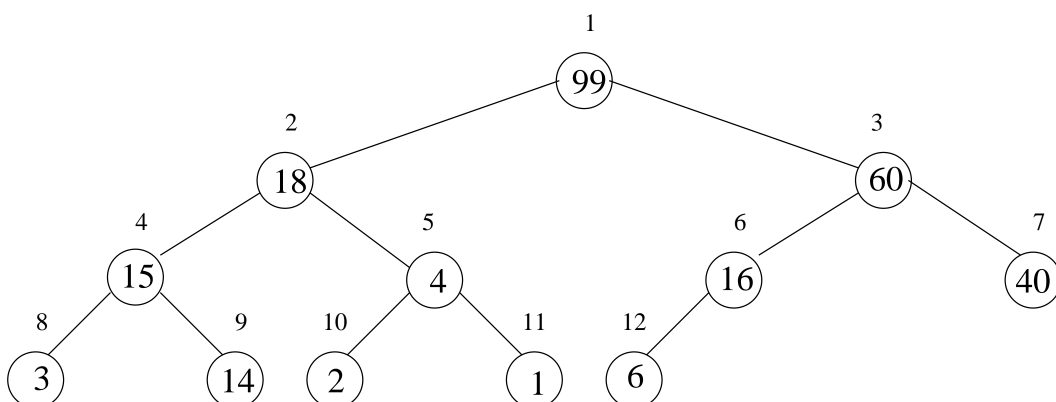
A binary relation $\bigcirc$ on set S is **irreflexive** if $x \bigcirc x$ is false for all $x \in S$, and it is **transitive** if $x \bigcirc y$ and $y \bigcirc z$ implies $x \bigcirc y$ for all $x, y, z \in S$. A binary relation $\olessthan$ is a **strict weak ordering** if it is irreflexive, transitive, and if the relation $\circledequal$, defined by

$$x \circledequal y \iff \text{both } x \olessthan y \text{ and } y \olessthan x \text{ are false,}$$

is transitive.

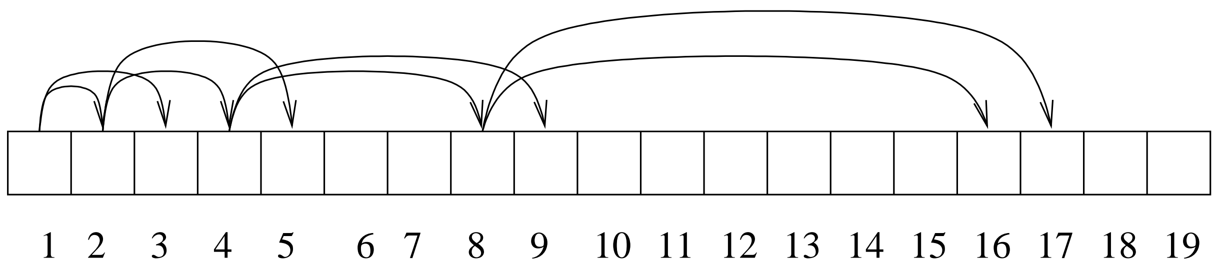
A tree T, where each node stores an element, is a **heap** with respect to $\olessthan$ if it fulfils the following properties:

1. Every subtree of T is a heap.
2. For the key x stored at the root of T and the key y stored at any of children of the root $x \olessthan y$ is false.



Binary Heaps

- A **binary heap** is an almost complete binary tree which is represented with an array $A[1..n]$ and which forms a heap.



- The element at the **root** of the tree is $A[1]$.
- For index i,

Parent(i)
1 **return** $\lfloor i/2 \rfloor$

Left-Child(i)
1 **return** $2 \cdot i$

Right-Child(i)
1 **return** $2 \cdot i + 1$

Standard Heapsort

Standard-Heapsort(A,$\circledless$)
1 $\quad$ $\mathit{heap\text{-}size}[A] \leftarrow \mathit{length}[A]$
2 $\quad$ Make-Heap(A,1,$\circledless$)
3 $\quad$ **for** $i \leftarrow \mathit{length}[A]$ **downto** 2
4 $\qquad$ $A[1] \leftrightarrow A[i]$
5 $\qquad$ $\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$
6 $\qquad$ Top-Down-Heapify(A,i,$\circledless$)

Running time: Let n denote $\mathit{length}[A]$. Make-Heap takes $O(n)$ time and each of the Top-Down-Heapify operations $O(\lg n)$ time, which gives $O(n \lg n)$ time in total.

Generic Heapsort

C (function pointer)

```
void heapsort(void* x, size_t n, size_t es,
              bool (*lt)(const void*, const void*))
```

C++ (function pointer)

```
template<typename R, typename F>
void heapsort(R, R, F))
```

C++ (member function)

```
template<typename R, typename F>
void heapsort(R, R, F))
```

ML

```
signature Heapsort =
sig
   val sort: ('a * 'a -> order) -> 'a Array.array -> unit
end
```

Language Effects

C (function pointer)

```
shell> gcc -O4 Standard.heapsort.c
shell> ./a.out
Sorting 1000000 sorted integers ...
Running time (s): 17.160
```

C++ (function pointer)

```
shell> g++ -O4 Standard.heapsort(pointer).cc
shell> ./a.out
Sorting 1000000 sorted integers ...
Running time (s): 18.800
```

C++ (member function)

```
shell> g++ -O4 Standard.heapsort(member).cc
shell> ./a.out
Sorting 1000000 sorted integers ...
Running time (s): 5.300
```

ML

```
shell> sml
Standard ML of New Jersey, Version 109.32, October 1, 1997
- use "Standard.Heapsort.sml";
...
- test();
Sorting 1000000 sorted integers ...
User: 99.480 System: 0.010 Garbage collection: 0.000
```

Heapsort vs. Quicksort

```
shell> ./a.out
Quicksort comparisons for integer data

n = 1000000
repetitions = 3

Quicksort from the C library:
        Average running time (s):       20.490
Tuned Quicksort by Bentley and McIlroy:
        Average running time (s):        9.710
Quicksort from the STL:
        Average running time (s):        1.767
Iterative Quicksort from Sedgewick's book:
        Average running time (s):        2.480
```

Compile-Time "If"

```
#include <iostream>
using namespace std;

template<bool condition, class Then, class Else>
struct IF
{ typedef Then RET;
};

//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else>
{ typedef Else RET;
};

void main()
{ cout << "sizeof(short) = " << sizeof(short) << endl
       << "sizeof(int) = " << sizeof(int) << endl
       << "sizeof(IF<(1+2>4), short, int>::RET) = "
       << sizeof(IF<(1+2>4), short, int>::RET)
       << endl;

  IF<(1+2>4), short, int>::RET i; //the type of i is int!
}
```

Compile-Time Recursion

```
#include <iostream>
using namespace std;

template<int n>
struct Factorial
{ enum { RET = Factorial<n-1>::RET * n };
};

// the following template specialization terminates
// the recursion
template<>
struct Factorial<0>
{ enum { RET = 1 };
};

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}
```

Compile-Time Lists

```
#include <iostream>
using namespace std;

// list implementation

// tag marking the end of a list
const int endValue = ~(~0u >> 1); //initialize with the smal

struct End
{ enum    { head = endValue };
  typedef End Tail;
};

template<int head_, class Tail_ = End>
struct Cons
{ enum    { head = head_ };
  typedef Tail_ Tail;
};
```

Compile-Time Lists (Cont.)

```
// Length<>

template<class List>
struct Length
{ // make a recursive call to Length and pass Tail of
  // the list as the argument
  enum { RET = Length<typename List::Tail>::RET+1 };
};

// stop the recursion if we've got to End
template<>
struct Length<End>
{ enum { RET = 0 };
};

// IsEmpty<>

template<class List>
struct IsEmpty
{ enum { RET = false };
};

template<>
struct IsEmpty<End>
{ enum { RET = true };
};

// test

void main()
{ typedef Cons<1,Cons<2,Cons<3> > > list1;

  cout << Length<list1>::RET << endl;  // prints 3
}
```

Static Code Generation

```
#include <iostream>
using namespace std;

#include "../meta/meta.h"

struct AlgorithmA
{ static void execute()
  { cout << "AlgorithmA" << endl; }
};

struct AlgorithmB
{ static void execute()
  { cout << "AlgorithmB" << endl; }
};


void main()
{ meta::IF<(1<2), AlgorithmA, AlgorithmB>::RET::execute();
}
```

Turing Completeness

A language is Turing complete if it provides a conditional and a looping contruct. That is, the meta level of C++ can compute the same functions as a Turing machine.

Conclusions

It is theoretically interesting that the template level of C++ has the power of a Turing machine, but template metaprogramming has its problems; in particular, in the following areas:

- debugging,
- error reporting,
- code readability,
- code maintainability,
- separate compilation,
- compilation speed,
- internal capacity robustness of compilers, and
- portability.

Most problems seem to be related to unbounded polymorphism.

Discussion: Wanted

The priority-queue class of the STL is just an adaptor that makes any resizable array to a queue in which the elements stored are ordered automatically according to a given ordering function.

We want to have an extension that

- supports methods decrease() (often called decrease-key()), delete(),
- keeps external references to compartments inside the data structure valid at all times, and
- provides bidirectional iterators.

How should the current design in the C++ standard library be changed?