"A programming language is low level when its programs require attention to the irrelevant."
- Alan J. Perlis.

CSE341
Programming Languages

Lecture 8 – November 26, 2015

ADT

Slides are taken from C. Li & W. He

Abstract Data Types

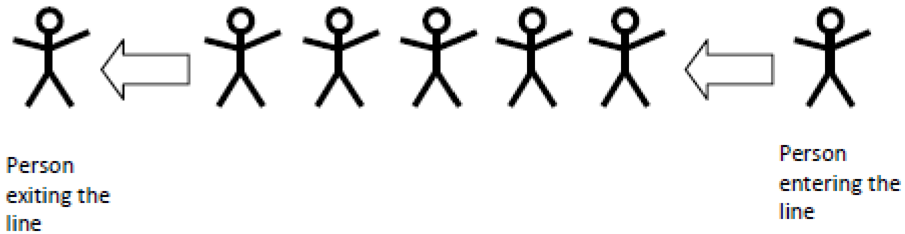
Data Types

- Predefined
- Type constructors: build new data types

- How to provide “queue”?
  - What should be the data values?
  - What should be the operations?
  - How to implement (data representation, operations)?

Queue

- A common abstract data type is a queue.



- A queue is a first in, first out (FIFO) structure or in the other sense, a last in, last out (LILO) structure. A queue is sometimes generalized as a structure where insertion (enqueue) occur at one end and removal (dequeue) occurs at the other end.

Queue Implementation

What are inadequate here?

- **The operations are not associated with the data type**
  - You can use the operation on an invalid value
- **Users see all the details: direct access to date elements, implementations**
  - Implementation dependent
  - Users can even mess up with things

What do we want?

- For basic types:
  - 4 bytes or 2 bytes, users don't need to know.
  - Can only use predefined operations.

- Similarly, for the "Queue" data type:

  ?

Abstract Data Types

- Built-in types have important properties that "abstract" away the implementation: use of int and its operations (+, *, etc.) normally do not require knowledge of bit patterns (2's complement? 4 bytes?)
- User-defined types do not in general have this property: internal structure is visible to all code
- Use of internal structure makes it difficult to change later
- Operations on data (except the most basic) not specified and often hard to find

Kenneth C. Louden, 2005

Abstract Data Types

- A mechanism of a programming language designed to imitate the abstract properties of a built-in type as much as possible
- Must include a specification of the operations that can be applied to the data
- Must hide the implementation details from client code
- These properties are sometimes called encapsulation & information hiding (with different emphases)

Kenneth C. Louden, 2005

Abstract Data Type

- Encapsulation:

  All definitions of allowed operations for a data type in one place.

- Information Hiding:

  Separation of implementation details from definitions. Hide the details .

Algebraic Specification of ADT

- Syntactic specification (signature, interface):
  the name of the type, the prototype of the operations

- Semantic specification (axioms, implementation):
  guide for required properties in implementation
  mathematical properties of the operations

They don't specify:

- data representation
- implementation details

Syntactic Specification

```
type queue(element) imports boolean
operations:
   createq:   queue
   enqueue:   queue × element → queue
   dequeue:   queue → queue
   frontq:    queue → element
   emptyq:    queue → boolean
```

- **`imports:`** the definition queue needs `boolean`
- Parameterized data type `(element)`
- `createq:` not a function, or viewed as a function with no parameter

Algebraic Specification

```
variables: q: queue; x: element
axioms:
   emptyq(createq)        = true
   emptyq(enqueue(q,x))   = false
   frontq(createq)        = error
   frontq(enqueue(q,x))   = if emptyq(q) then x
                            else frontq(q)
   dequeue(createq)       = error
   dequeue(enqueue(q,x))  = if emptyq(q) then q
                            else enqueue(dequeue(q),x)
```

- `error` axiom (exceptions)

Stack

```
type stack(element) imports boolean
operations:
   createstk : stack
   push      : stack × element → stack
   pop       : stack → stack
   top       : stack → element
   emptystk  : stack → boolean

variables: s: stack; x: element
axioms:
   emptystk(createstk)    = true
   emptystk(push(s,x))    = false
   top(createstk)         = error
   top(push(s,x))         = x
   pop(createstk)         = error
   pop(push(s,x))         = s
```

Axioms

- How many axioms are sufficient for proving all necessary properties?

Some Heuristics

```
type stack(element) imports boolean
operations:
   createstk : stack
   push      : stack × element → stack
   pop       : stack → stack
   top       : stack → element
   emptystk  : stack → boolean

variables: s: stack; x: element
axioms:
   emptystk(createstk)    = true
   emptystk(push(s,x))    = false
   top(createstk)         = error
   top(push(s,x))         = x
   pop(createstk)         = error
   pop(push(s,x))         = s
```

Constructor:
createstk
push

Inspector:
pop
top
emptystk

2 * 3 = 6 rules

Binary Search Tree

```
type BST(element) imports boolean, int
operations:
   createbst   : BST
   emptybst    : BST → boolean
   insert      : BST × element → BST
   delete      : BST × element → BST
   getRoot     : BST → element
   getHeight   : BST → int
   max         : BST → element
   search      : BST × element → boolean

variables: t: bst; x: element
axioms:
   emptystk(createbst)      = true
   …
```

Other Examples of ADT

- Stack
- Queue
- Tree
- Set
- Map
- Vector
- List
- Priority Queue
- Graph
- …

Algebraic Specification Notes

- Specifications are usually written in functional form with no side effects or assignment. So no "void" functions
- Specifications are often simplified to make axiom writing easier
  - E.g., in the stack example, pop does not return the top, only the (previously created) stack below the current top.
  - We could have written pop as pop: stack → element x stack, but the axioms are more complex

ADT Language Mechanisms

- Most languages do not have a specific ADT mechanism – instead they have a more general module mechanism
- Specific ADT mechanisms
  - ML abstype – but newer module mechanism is more useful…
- General module mechanism: not just about a single data type and its operations
  - Separate compilation and name control: C, C++, Java
  - Ada, ML
- Class in OO languages (which has many of the properties needed by an ADT mechanism)

Kenneth C. Louden, 2005

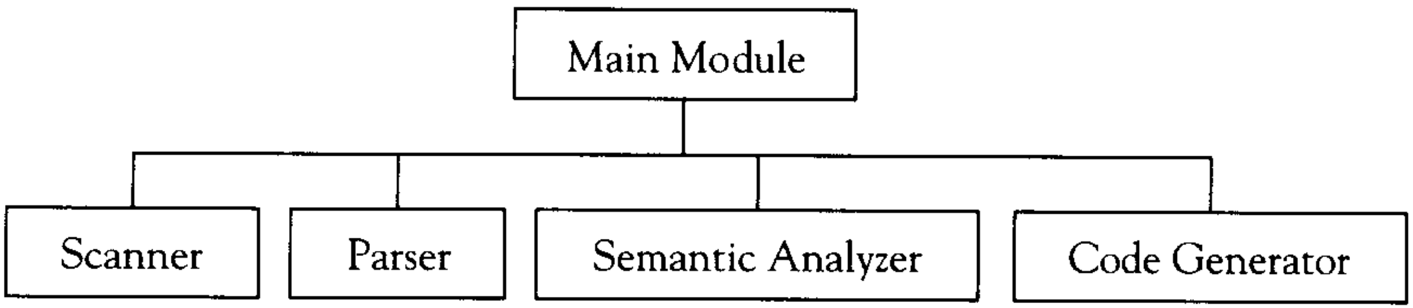
Modules

- **Module**: A program unit with a public interface and a private implementation; all services that are available from a module are described in its public interface and are exported to other modules, and all services that are needed by a module must be imported from other modules.
- A module offers general services, which may include types and operations on those types, but are not restricted to these.
- Modules have nice properties:
  - A module can be (re)used in any way that its public interface allows.
  - A module implementation can change without affecting the behavior of other modules.

- In addition to ADT, module supports structuring of large programs: Separate compilation and name control

Modules

- Modules are the principle mechanism used to decompose large programs
- Example – a compiler:



- Modules usually offer an additional benefit: names within one module do not clash with names in other modules
- Modules usually have a close relationship to separate compilation (though this is often hard to make precise in a specification)

Kenneth C. Louden, 2005

Modules

- Languages that have comprehensive module mechanisms:
  - Ada, where they are called packages (not to be confused with Java packages)
  - ML, where they are called structures
- Languages that have weak mechanisms with some module-like properties:
  - C++, where they are called namespaces
  - Java, where they are called packages
- Languages with no module mechanism:
  - C (but modules can be imitated using separate compilation)
  - Pascal

Kenneth C. Louden, 2005

C++ Namespaces & Java Packages

- C++ and Java do not have modules in the sense of Ada and ML: classes are used instead
- C++ and Java do have mechanisms for controlling name clashes and organizing code into groups: namespaces in C++, packages in Java.
- Clients must use similar dot notation as in Ada and ML to reference names in namespaces/packages.
- Each of these languages has a mechanism for automatically dereferencing names:
- Ada: use
- ML: open
- C++: using [namespace]
- Java: import
- Only Ada has explicit dependency syntax (keyword with). Java class loader automatically searches for code. C++ requires textual inclusions for declarations, linker must search for code. ML "compilation manager" does this too (not in ML specification).

C: Separate Compilation

- `queue.h` : header file

```
#ifdef QUEUE_H
#define QUEUE_H

struct Queuerep;
typedef struct Queuerep * Queue;
Queue createq(void);
Queue enqueue(Queue q, void* elem);
void* frontq(Queue q);
Queue dequeue(Queue q);
int emptyq(Queue q);

#endif
```

Incomplete type:
Separate implementation

Simulate
Parameteric polymorphism

C: Separate Compilation

- `queue.c`: queue implementation

```
#include "queue.h"

struct Queuerep
{ void* data;
  Queue next;
};

Queue createq(void)
{ return 0;
}

void* frontq(Queue q)
{ return q->next->data;
}
...
```

C: Separate Compilation

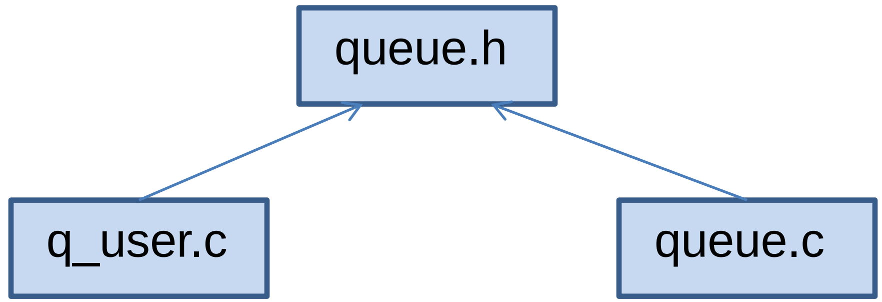
- `q_user.C`: client code

```
#include "queue.h"

int *x = malloc(sizeof(int));
int *y = malloc(sizeof(int));
int *z;
*x =2;
*y =3;

Queue q = createq();
q = enqueue(q,x);
q = enqueue(q,y);
q = dequeue(q);
z = (int*) frontq(q);
```

C: Separate Compilation



- Not real ADT
  - casting, allocation: for parametric polymorphism
  - header file directly incorporated into q_user.c: definition / usage consistent
  - data not protected: user may manipulate the type value in arbitrary ways
  - The language itself doesn't help in tracking changes and managing compilation/linking: thus tools like `make`

C++: Namespaces

- queue.h:

```
#ifdef QUEUE_H
#define QUEUE_H
namespace MyQueue
{
  struct Queuerep;
  typedef struct Queuerep * Queue;
  Queue createq(void);
  ...
}
#endif
```

- queue.c:

```
#include "queue.h‟

struct MyQueue::Queuerep
{ void* data;
  Queue next;
};
...
```

C++: Namespaces

- q_user.cpp:

```
#include "queue.h"

using std::endl;
using namespace MyQueue;
main(){
  int *x = malloc(sizeof(int));
  int *y = malloc(sizeof(int));
  int *z;
  *x =2;
  *y =3;
  Queue q = MyQueue::createq();
  q = enqueue(q,x);
  q = enqueue(q,y);
  q = dequeue(q);
  z = (int*) frontq(q);
  …
}
```

Java: Packages

Queue.java:

```
package queues.myqueue;
…
```

PQueue.java:

```
package queues.myqueue;
…
```

Q_user.java:

```
import queues.myqueue.Queue;
import queues.myqueue.*;
queues.myqueue.Queue;
```

directory:
queues/myqueue

class files:
Queue.class, PQueue.class

queues/myqueue in CLASSPATH

Example

- Package java.util
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/package-summary.html
- Interface Collection
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html
- Class PriorityQueue
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/PriorityQueue.html
- boost: providing free peer-reviewed portable C++ source libraries
  http://www.boost.org/

boost

```
#include <boost/lambda/lambda.hpp>
#include <iostream>
#include <iterator>
#include <algorithm>

int main(){
  using namespace boost::lambda;
  typedef std::istream_iterator<int> in;

  std::for_each(
    in(std::cin), in(), std::cout << (_1 * 3) << " " );
}
```

boost lambda

The Boost Lambda Library (BLL) is a C++ template library, which implements a form of lambda abstractions for C++. The term originates from functional programming and lambda calculus, where a lambda abstraction defines an unnamed function. The primary motivation for the BLL is to provide flexible and convenient means to define unnamed function objects for STL algorithms.

Example:

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

boost queue

```
// In header: <boost/lockfree/queue.hpp>

template<typename T, ... Options>
class queue {
public:
  // types
  typedef T                                      value_type;
  typedef implementation_defined::allocator allocator;
  typedef implementation_defined::size_type size_type;

  // construct/copy/destruct
  queue(void);
  template<typename U>
    explicit queue(typename node_allocator::template rebind< U >::other const &);
  explicit queue(allocator const &);
  explicit queue(size_type);
  template<typename U>
    queue(size_type,
          typename node_allocator::template rebind< U >::other const &);
  ~queue(void);

  // public member functions
  bool is_lock_free(void) const;
  void reserve(size_type);
  void reserve_unsafe(size_type);
  bool empty(void);
  bool push(T const &);
  bool bounded_push(T const &);
  bool unsynchronized_push(T const &);
  bool pop(T &);
  template<typename U> bool pop(U &);
  bool unsynchronized_pop(T &);
  template<typename U> bool unsynchronized_pop(U &);
};
```

Problems with Modules

- Modules are not types
  - Modules sometimes used to imitate OO classes
  - Module interface usually contains types, whose representations may be exposed
- Modules are static
  - Modules are primarily compile-time artifacts
  - Use of a module to imitate a class (without exporting a type) results in only one available object
- Modules do not control values of exported types
  - Assignment can cause undesirable aliasing
  - Equality tests may not be appropriate
  - ML and Ada have some ability to control these (with effort)

Kenneth C. Louden, 2005

Problems with Modules

- These problems can be (mostly) overcome by using OO classes, with modules relegated to code organization status (C++, Java)
- Significant problems still exist, even with OO :
  - Modules do not expose dependencies
    - Only Ada documents compilation dependencies in code (keyword with)
    - Hidden implementation dependencies can be worse: order relation is a common one
    - C++ does a particularly good job of hiding these
    - Ada uses constrained polymorphism
    - Java uses interfaces such as Comparable, Comparator
  - Modules do not express semantics
    - Universally ignored in today's languages
    - Useful for proving code correctness

Kenneth C. Louden, 2005