SWINBURNE UNIVERSITY OF TECHNOLOGY

ADT, Stack, STL and Doubly Linked List

Abstract Data Type

- Abstract
  - Existing in thought or as an idea but not having a physical or concrete existence.
  - Consider something theoretically
    - i.e., A summary of the contents of a book, article, or speech.
- The technique used to define new data types independently of their actual representation is called data abstraction.
- Data abstraction provides only essential information to the outside world and hiding their background details,
  - i.e., to represent the needed information in program without presenting the details.

Abstract Data Type –Data abstraction

- Data abstraction is a programming *and design technique that relies on the separation of interface* and implementation.
- Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players,
- BUT you do not know its internal details, that is, you do not know
  - how it receives signals over the air or through a cable,
  - how it translates them, and finally displays them on the screen.
- Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Abstract Data Type –Data abstraction

- C++ classes provides great level of **data abstraction. They provide sufficient public methods** to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

Data abstraction

- Any C++ program where you implement a class with public and private members is an example of data abstraction.
- The public members `addNum` and `getTotal` are the interfaces to the outside world and **a user** needs to know them to use the class.
- The private member `total` is something that **the user** doesn't need to know about, but is needed for the class to operate properly.

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

class Adder{
   public:
      // constructor
      Adder(int i = 0)
      {
        total = i;
      }
      // interface to outside world
      void addNum(int number)
      {
          total += number;
      }
      // interface to outside world
      int getTotal()
      {
          return total;
      };
   private:
      // hidden data from outside world
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Abstract Data Type

- Abstract data types : A set of data values and associated operations that are precisely specified independent of any particular implementation.
- A client (program) can use values of an abstract data type by means of the interface without knowing their actual representation (which can change over time)

Implement lists as abstract data types

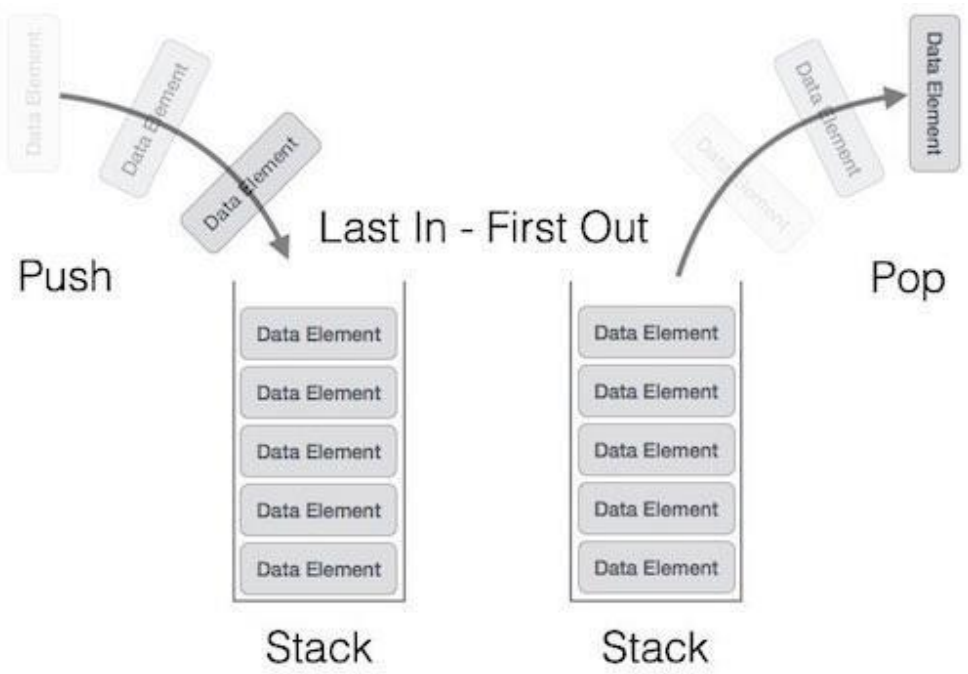
```
#include "DoublyLinkedNode.h"
#include "DoublyLinkedNodeIterator.h"

template<class T>
class List
{
private:
    // auxiliary definition to simplify node usage
    typedef DoublyLinkedNode<T> Node;

    Node* fTop;     // the first element in the list
    Node* fLast;    // the last element in the list
    int fCount;     // number of elements in the list

public:
    // auxiliary definition to simplify iterator usage
    typedef DoublyLinkedNodeIterator<T> Iterator;

    List();                                     // default constructor - creates empty list
    List( const List& aOtherList );             // copy constructor
    ~List();                                    // destructor - frees all nodes

    List& operator=( const List& aOtherList );  // assignment operator

    bool isEmpty() const;                       // Is list empty?
    int size() const;                           // list size

    void prepend( const T& aElement );          // add a node initialized with aElement at front
    void append( const T& aElement );           // add a node initialized with aElement at back
    void remove( const T& aElement );           // remove node that matches aElement from list

    const T& operator[]( int aIndex ) const;    // list indexer

    Iterator getIterator() const;               // return an iterator for the nodes of the list
};
```

Data Structure: Stack

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack
- A real-world stack allows operations at one end only.
  - For example, we can place or remove a card or plate from the top of the stack only.
- Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

Stack

- It is a LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first.
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



Stack operations

- We have seen the function `push_back()`.
  - It appends an element to the end of the controlled sequence.
- There is a counterpart function, `pop_back()`,
  - It removes the last element in the controlled sequence.

```
std::vector<int> v;
v.push_back(999);
v.pop_back();
```

Applications of Stacks

- Reversal of input
- Checking for matching parentheses
- Backtracking (e.g., Prolog or graph analysis)
- State of program execution (e.g., storage for parameters and local variables of functions)
- Tree traversal
- ...

Container

- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Stack Interface

- When defining a container type we wish to minimize the number of value copies required for the objects stored in the container.
- In order to achieve this, we use const references (e.g. const T&).

```
4   template<class T>
5   class Stack
6   {
7   public:
8       Stack( int aSize );
9       ~Stack();
10
11      bool isEmpty() const;
12      int size() const;
13      void push( const T& aItem );
14      void pop();
15      const T& top() const;
16  };
```

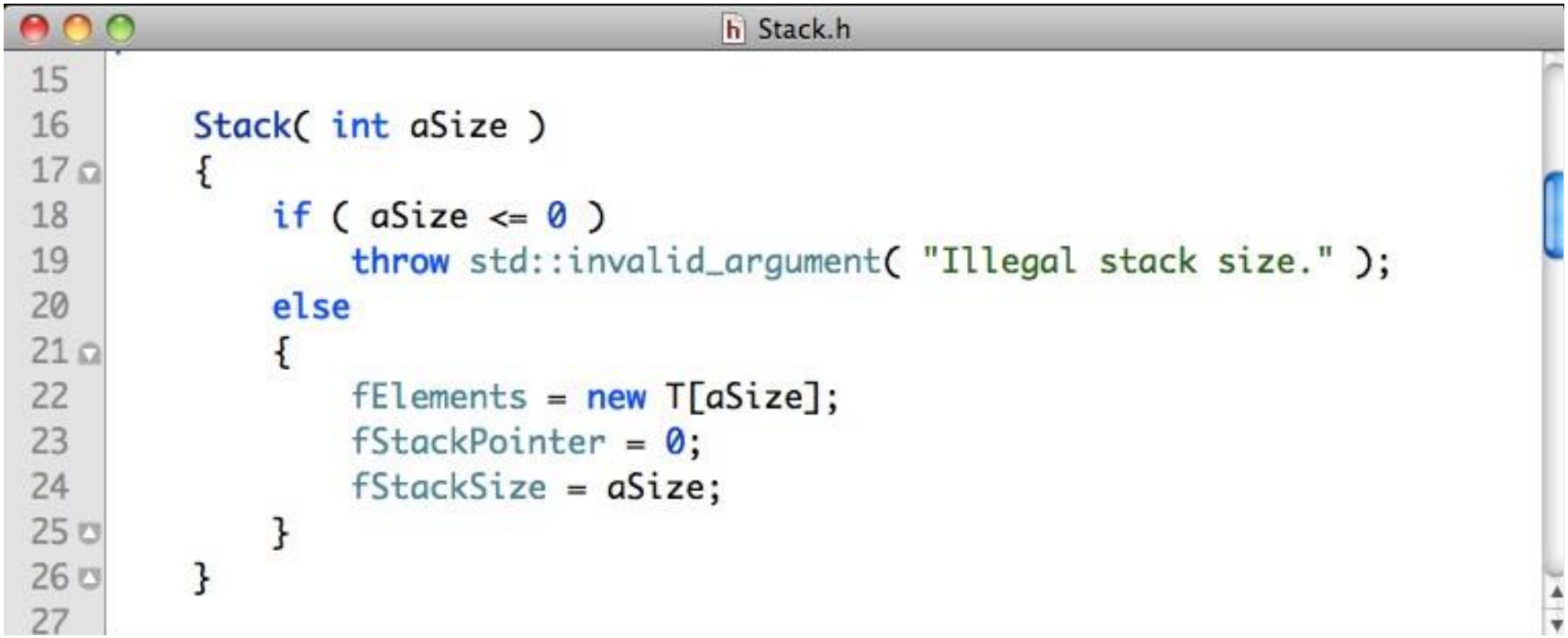
  - const references will allow us to refer to the actual object or value rather than a mere copy of it.

The Stack's Private abstraction

- Inside Stack we need to be able to store objects of type T.
- Hence we need to allocate dynamically memory (i.e, an array of type T) and
- store the address of the first element in a matching pointer variable.

```
4   template<class T>
5   class Stack
6   {
7   private:
8       T* fElements;
9       int fStackPointer;
10      int fStackSize;
11
12  public:
13
14      ...
15
16  };
```

Stack Constructor



Stack Destructor

```
~Stack()
{
    delete [] fElements;
}
```

- There are two forms of delete:
  - delete ptr - release the memory associated with pointer ptr.
  - delete [] arr - release the memory associated with all elements of array arr and the array arr itself.
- Whenever one allocates memory for an array, for example char* arr = new char[10], one must use the array form of delete to guarantee that all array cells are released.

More on delete

- `int *p_scalar = new int(5);`
  - allocates an integer, set to 5. (same syntax as constructors)
- `int *p_array = new int[5];`
  - allocates an array of 5 adjacent integers. (undefined values)
- `delete p_scalar`
- `delete[] p_array`

Stack utilities

- `isEmpty()`: Boolean predicate to indicate whether there are elements on the stack.
- `size()`: returns the actual stack size.

```
bool isEmpty() const
{
    return fStackPointer == 0;
}

int size() const
{
    return fStackPointer;
}
```

Push and pop

- The push method stores a item at the next free slot in the stack, if there is room.

```
void push( const T& aItem )
{
    if ( fStackPointer < fStackSize )
        fElements[fStackPointer++] = aItem;
    else
        throw std::overflow_error( "Stack full." );
}
```

- The pop method shifts the stack pointer to the previous slot in the stack, if there is such a slot. Note, the element in the current slot itself is not yet destroyed.

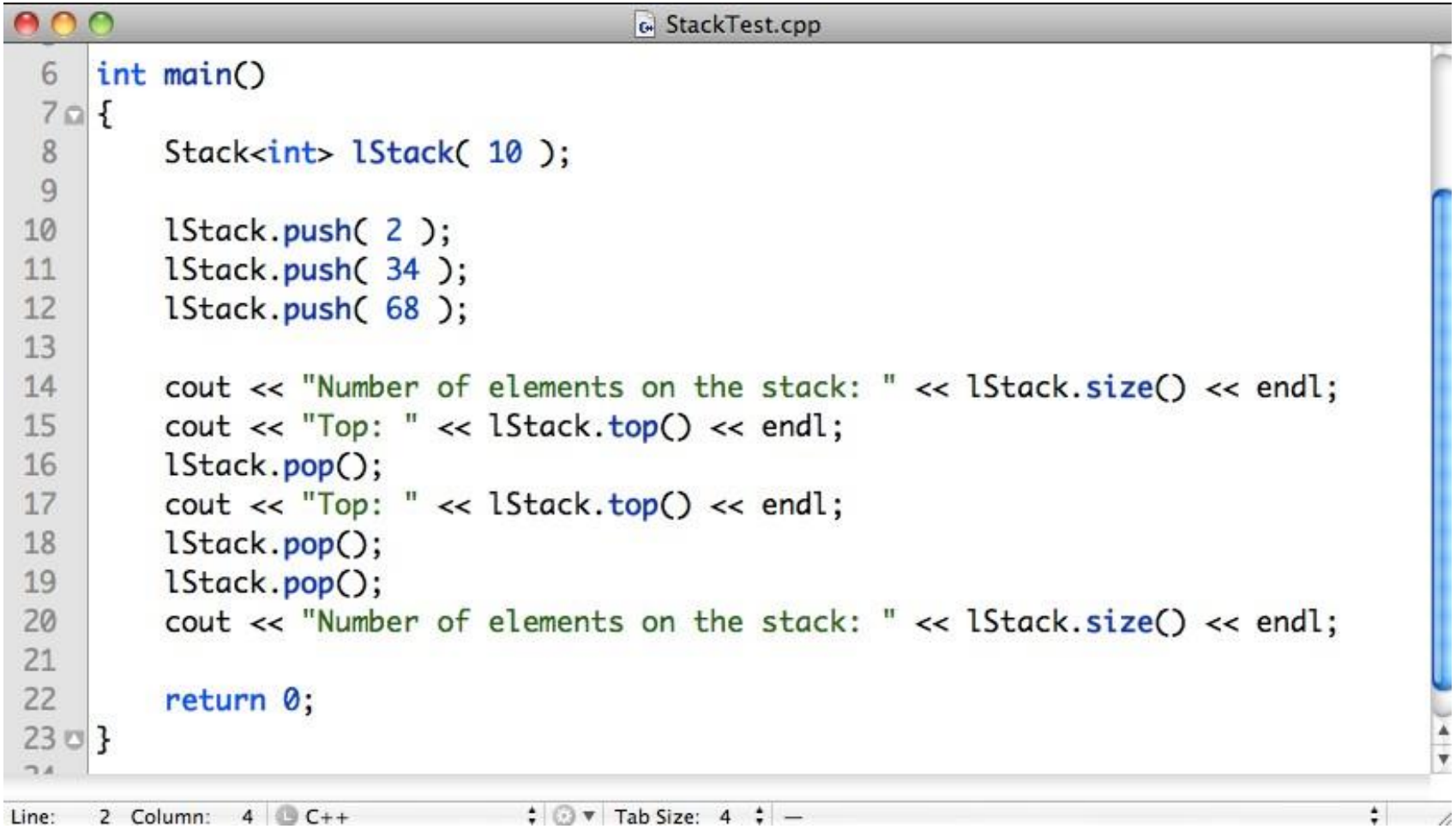
```
void pop()
{
    if ( !isEmpty() )
        fStackPointer--;
    else
        throw std::underflow_error( "Stack empty." );
}
```

Top

```
const T& top() const
{
    if ( !isEmpty() )
        return fElements[fStackPointer-1];
    else
        throw std::underflow_error( "Stack empty." );
}
```

- The top method returns a const reference to the item in the current slot in the stack, if there is such a slot.

Stack Sample



Dynamic Stack

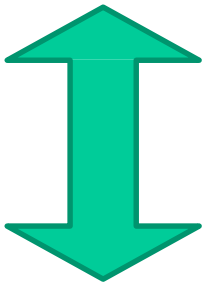
- We can define a dynamic stack that uses a list as underlying data type to host an arbitrary number of elements:

```
template<class T>
class DynamicStack
{
private:
    List<T> fElements;

public:

    bool isEmpty() const;
    int size() const;
    void push( const T& aItem );
    void pop();
    const T& top() const;
};
```

Reverse Polish Notation

- Reverse Polish Notation (RPN) is a prefix notation wherein operands come before operators.

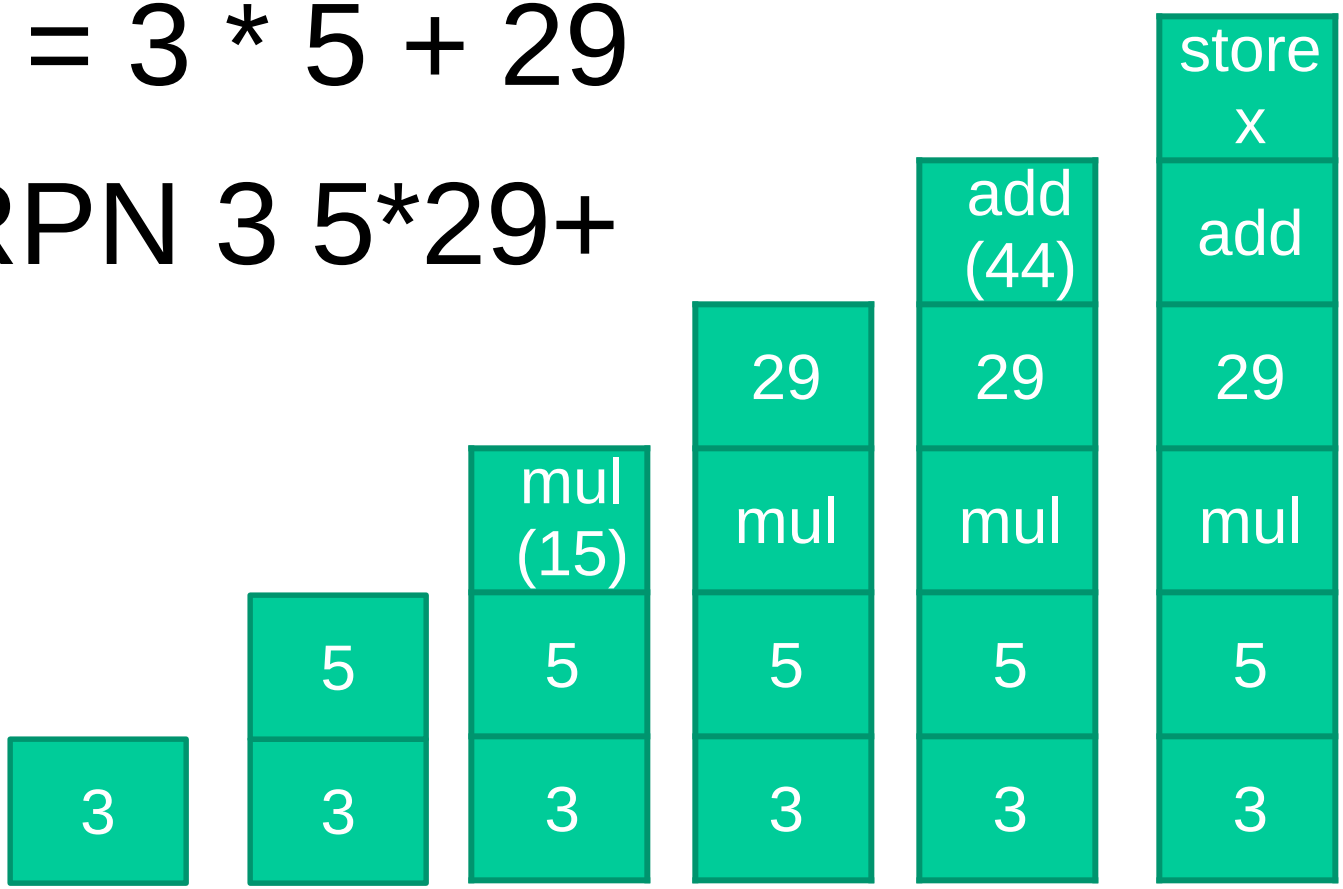
RPN: a b * c +



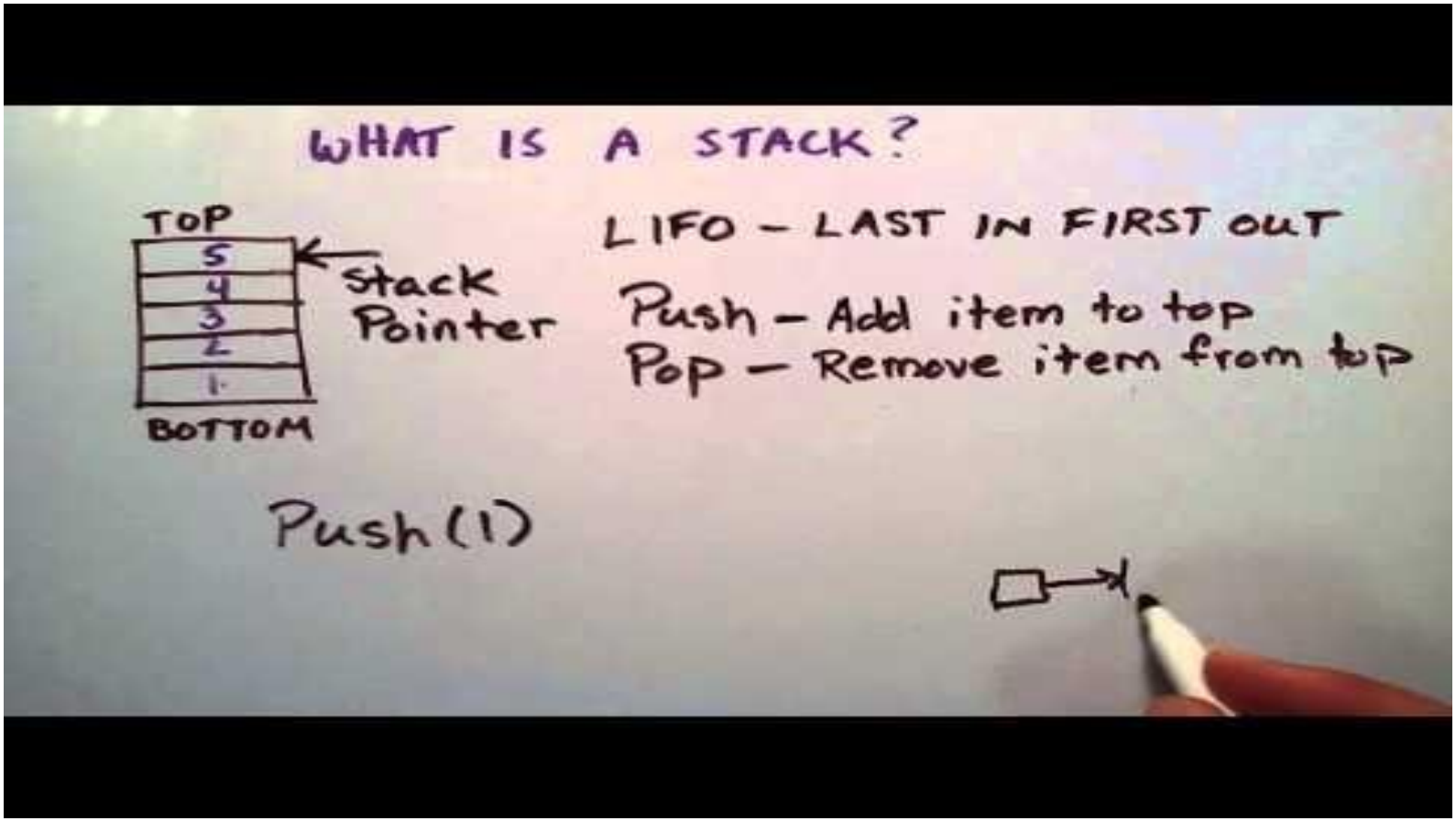
Infix: a * b + c

RPN Calculation using a Stack

- x = 3 * 5 + 29
- RPN 3 5*29+

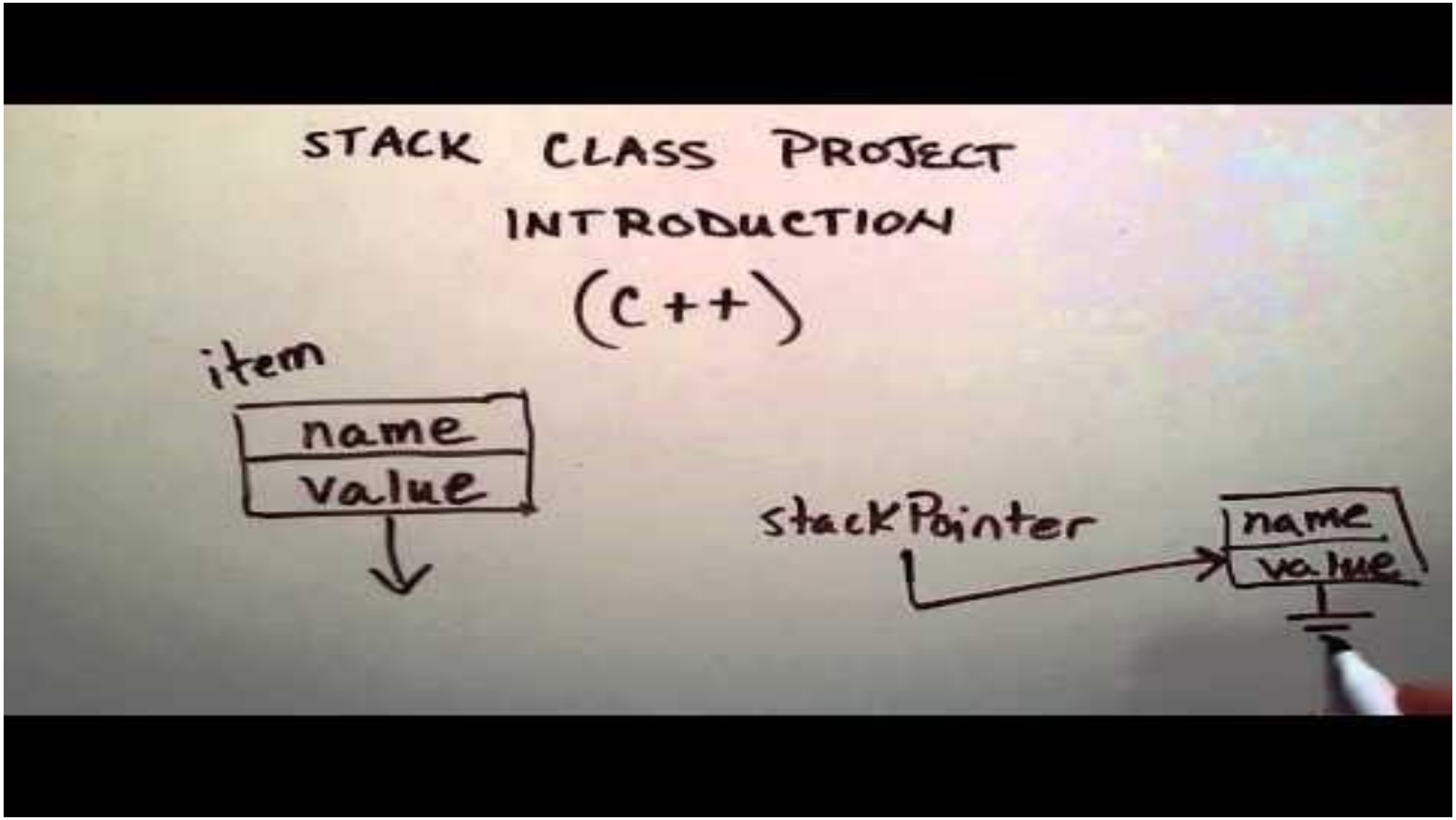


What is a Stack Data Structure



URL: https://www.youtube.com/watch?v=FNZ5o9S9prU

Stack Class (Intro)



URL:
https://www.youtube.com/watch?v=rDG6pUveq4o&list=PLTxllHdfUq4eD0L7K2UHU16A_sudFPR_x&index=1

SWINBURNE UNIVERSITY OF TECHNOLOGY

STL of C++ part 1

Containers

- The STL makes seven basic containers available, and three derived.
- There are two categories of containers in the STL
  - sequence: vector, list and deque
  - associative: set, multiset, map and multimap
  - specialised derived container: stack, queue and priority queue

Vector

- Vectors are sequence containers representing arrays that can change in size. (dynamic arrays)
  - contiguous storage locations
  - efficiently direct element access as arrays
  - (unlike array) size can change dynamically
  - internally vectors do not reallocate each time an element is added/removed to the container.
  - allocate some extra storage to accommodate for possible growth
- Compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Vector: member function

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Vector: member function

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

Allocator:

get_allocator	Get allocator (public member function)

fx Non-member function overloads

relational operators	Relational operators for vector (function template)
swap	Exchange contents of vectors (function template)

Template specializations

vector<bool>	Vector of bool (class template specialization)

Reference
http://www.cplusplus.com/reference/vector/vector/

Vector: Constructors

```
typedef std::vector<std::string> str_vec_t;
str_vec_t v1;                          // create an empty vector
str_vec_t v2(10);                      // 10 copies of empty strings
str_vec_t v3(10, "hello");             // 10 copies of the string
                                       // "hello"
str_vec_t v4(v3);                      // copy ctor

    std::list<std::string> sl;         // create a list of strings
                                       // and populate it
    sl.push_back("cat");
    sl.push_back("dog");
    sl.push_back("mouse");

str_vec_t v5(sl.begin(), sl.end()); // a copy of the range in
                                       // another container
                                       // (here, a list)

v1 = v5;                               // will copy all elements
                                       // from v5 to v1
```

Vector: *assign()*

```
v1.assign(sl.begin(), sl.end());     // copies the list into
                                     // the vector
v1.assign(3, "hello");               // initializes the vector
                                     // with 3 strings "hello"
```

- The assignment completely changes the elements of the *vector*.
- The old elements (if any) are discarded and the size of the *vector* is set to the number of elements assigned.
- *assign()* may trigger an internal reallocation.

Array vs. vector

```
size_t size = 10;
int sarray[10];
int *darray = new int[size];
// do something with them:
for(int i=0; i<10; ++i){
    sarray[i] = i;
    darray[i] = i;
}
// don't forget to delete darray when you're done
delete [] darray;
```

static array (on stack)

dynamic array (on heap)

- It takes a non-const size parameter as the dynamic
- Automatically deletes the used memory like the static one

```
#include <vector>
//...
size_t size = 10;
std::vector<int> array(size);    // make room for 10 integers,
                                 // and initialize them to 0
// do something with them:
for(int i=0; i<size; ++i){
    array[i] = i;
}
// no need to delete anything
```

use operator `[]` also it is possible to use `at()`

Array vs. vector

- ordinary array:
  - fixed size
  - quick random access by index
  - slow to insert and erase in the middle
  - size can’t be changed at runtime
- vector
  - relocating, expandable array
  - quick random access by index
  - slow to insert and erase in the middle
  - quick to insert or erase at end

Vector

- What if you don't know how many elements you will have?
  - implement a logic that allows to grow your array from time to time (complicated)
  - allocate an array that is "big enough." (expensive and poor practice)
- not the case for *vector*

```
#include <vector>
#include <iostream>
//...
std::vector<char> array;
char c = 0;
while(c != 'x'){
  std::cin>>c;
  array.push_back(c);
}
```

- *push_back()* appends one element at a time to the array.
- You can *push_back()* elements until the allocated memory is exhausted
- Then, *vector* will trigger a reallocation and will grow the allocated memory block
- move(copy) to a larger block.
- slow down your application dramatically.

- *vector* has a so-called 'controlled sequence' and a certain amount of allocated memory for that sequence.
- The *controlled sequence* is just another name for the array in the guts of the *vector*. To hold this array, *vector* will allocate some memory, mostly more than it needs.

Vector

- In the previous example, we declared the *vector* using its default constructor. This creates an empty *vector*.
- Depending on the implementation of the Standard Library being used, the empty *vector* might or might not allocate some memory "just in case."
- If we want to avoid a too-often reallocation of the *vector*'s storage, we can use its *reserve()* member function:

```
#include <vector>
#include <iostream>
//...
std::vector<char> array;
array.reserve(10);    // make room for 10 elements
char c = 0;
while(c != 'x'){
  std::cin>>c;
  array.push_back(c);
}
```

The function *reserve()* will ensure that we have room for at least 10 elements in this case

- If the *vector* already has room for the required number of elements,
- *reserve()* does nothing.
- In other words, *reserve()* **will grow the allocated storage of the *vector*, if necessary, but will never shrink it.**

Vector –size and capacity

```
#include <vector>
#include <iostream>
//...
std::vector<int> array;
int i = 999;            // some integer value
array.reserve(10);      // make room for 10 elements
array.push_back(i);
std::cout<<array.capacity()<<std::endl;
std::cout<<array.size()<<std::endl;
```

For this example, only 0 is a valid index for *array*.

- We have made room for at least 10 elements with *reserve()*, but the memory is not initialized
- `array.at()` returns *out_of_range* if it access other elements
- the role of *reserve()* is to minimize the number of potential reallocations and that it **will not** influence the number of elements in the controlled sequence
    - A call to *reserve()* with a parameter smaller than the current *capacity()* is benign—it simply does nothing.

Vector - resize

- The correct way of enlarging the number of contained elements is to call *vector*'s member function *resize()*.
  - If the new size is larger than the old size of the *vector*, it will preserve all elements already present; the rest will be initialized according to the second parameter.
  - If the new size is smaller than the old size, it will preserve only the first new_size elements. The rest is discarded and shouldn't be used any more—consider these elements invalid.
  - If the new size is larger than *capacity()*, it will reallocate storage so all new_size elements fit. *resize()* will never shrink *capacity()*.

Vector - resize example

```
std::vector<int> array;     // create an empty vector
array.reserve(3);           // make room for 3 elements
                            // at this point, capacity() is 3
                            // and size() is 0
array.push_back(999);       // append an element
array.resize(5);            // resize the vector
                            // at this point, the vector contains
                            // 999, 0, 0, 0, 0
array.push_back(333);       // append another element into the vector
                            // at this point, the vector contains
                            // 999, 0, 0, 0, 0, 333
array.reserve(1);           // will do nothing, as capacity() > 1
array.resize(3);            // at this point, the vector contains
                            // 999, 0, 0
                            // capacity() remains 6
                            // size() is 3
array.resize(6, 1);         // resize again, fill up with ones
                            // at this point the vector contains
                            // 999, 0, 0, 1, 1, 1
```

Vector

```
double p[] = {1, 2, 3, 4, 5};
std::vector<double> a(p, p+5);
```

- Another constructor provided by *vector*.
- It takes two parameters: a pointer to the first element of a C- style array and a pointer to one past the last element of that array

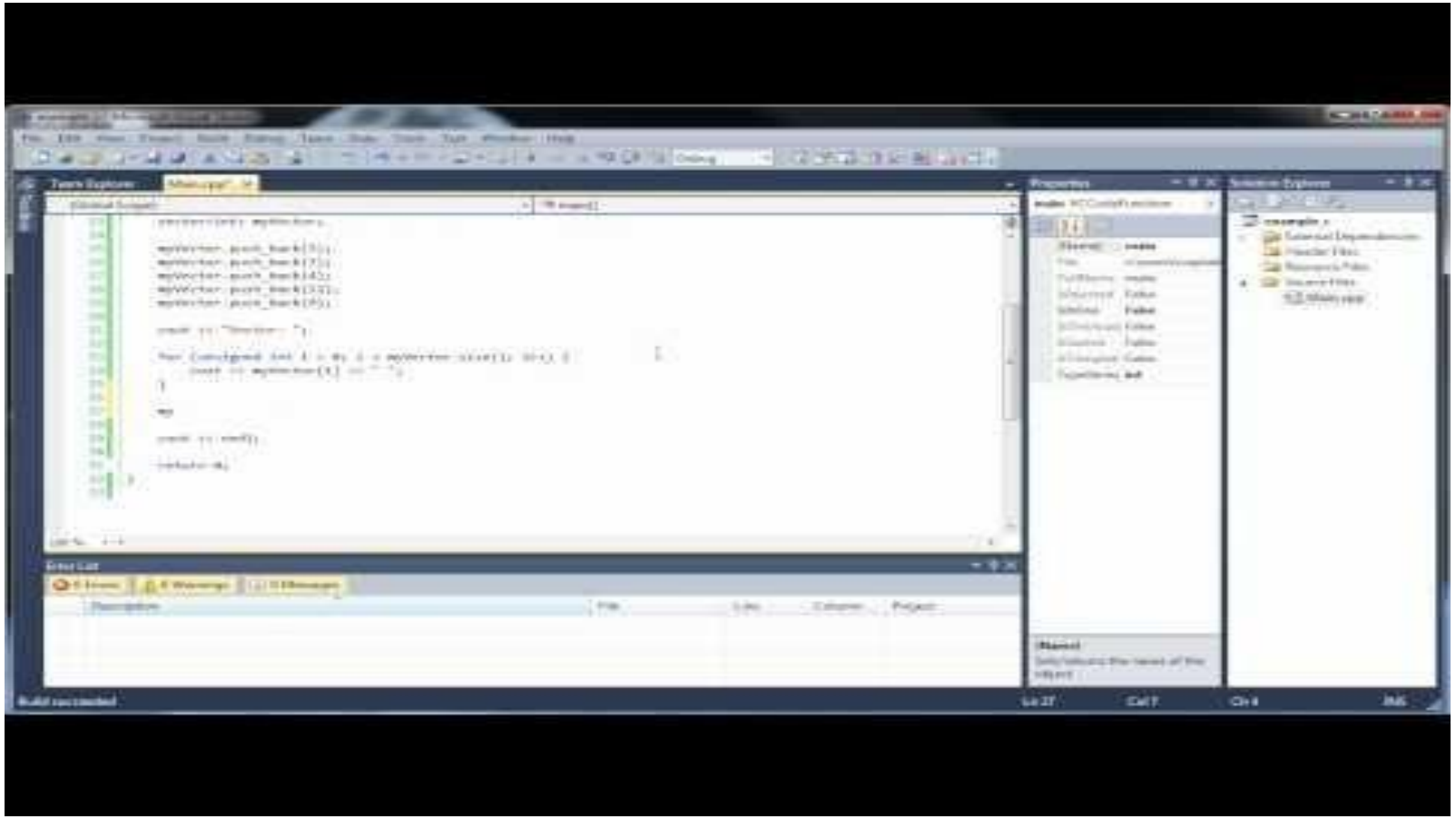
Vector

- When taking the address of elements contained in a *vector*, there is something you have to watch out for: an internal reallocation of the *vector* will invalidate the pointers you hold to its elements.

```
d:\code\vtut\Debug\vtut.exe
The size is 5
The address of pi is 00A8558C
The address of v[3] is 00A8558C
The address of v[3] after push is 00A85674
0
```

```
int main()
{
    std::vector<int> v(5);

    std::cout<<"The size is "<<v.size()<<std::endl;

    int *pi = &v[3];

    std::cout<<"The address of pi is "<<pi<<std::endl;
    std::cout<<"The address of v[3] is "<<&v[3]<<std::endl;

    v.push_back(999); // <-- may trigger a reallocation

    std::cout<<"The address of v[3] after push is "<<&v[3]<<std::endl;

    print_v(&v);

    std::cout<<"The size after push_back is "<<v.size()<<std::endl;

    //*pi = 333;        // <-- probably an error, pi isn't valid any more
```

Vectors and Vector Functions



URL: https://www.youtube.com/watch?v=Cq1h1KPoGBU

Iterator

- Iterators are the way the Standard Library models a common interface for all containers—*vector, list, deque*, *set* and so on.
- The reason is that operations that are "natural" for one container (like subscripting for *vector*) do not make sense for other containers.
- The Standard Library needs a common way of applying algorithms like iterating, finding, sorting to all containers—thus the concept of iterators.
- iterators are generalisation of the concept of points to contained elements.

Iterator

- Iterating: a process of moving sequentially from element to element.
- increment iterator with ++ operator
  - so it points to next element in the container
- decrement iterator with – operator
  - so it points to previous element in the container
- dereference iterator with *
  - so you can get the content/value of the element it points to.
- In STL, an iterator is represented by an object of an iterator class

Iterator for Vector

- The important thing is that if you have an iterator, you can dereference it to obtain the element it "points" to
  - what is dereferencing? : get you a reference to the actual object the pointer/iterator is pointing to
  - **Dereferencing** a pointer/iterator means getting the value that is stored in the memory location pointed by the pointer/iterator.
- For *vector* the most natural implementation of an iterator is indeed a plain pointer— *but don't count on this*

Iterator for Vector

```
1  #include <vector>
2  #include <iostream>
3
4  int main()
5  {
6       std::vector<double> a;
7       std::vector<double>::const_iterator i;
8       a.push_back(1);
9       a.push_back(2);
10      a.push_back(3);
11      a.push_back(4);
12      a.push_back(5);
13
14      for(i=a.begin(); i!=a.end(); ++i){
15          std::cout<<(*i)<<std::endl;
16      }
17
18      system("pause");
19      return 0;
20 }
```

This declares a `const` iterator *i* for a *vector<double>*. We are using a const iterator because we do not intend to modify the contents of the *vector*.

The member function begin() returns an iterator that "points" to the first element in the sequence.

The member function *end()* returns an iterator that "points" to one-past-the-last- element in the sequence.

Iterator for Vector

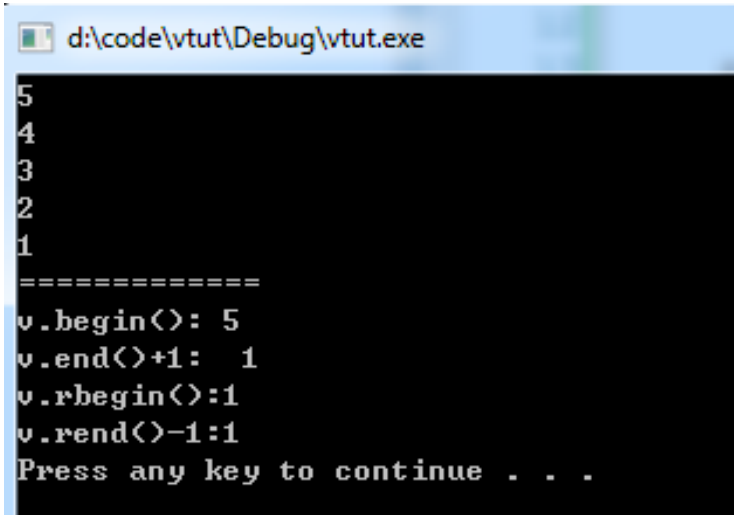
- dereferencing the iterator returned by *end()* is illegal and has undefined results. e.g.
  - `std::cout<<*(a.end())<<std::endl;`
- The last element is `*(a.end()-1)`

Why bother with iterators at all?

- The answer is that we have to use iterators if we want to apply some standard algorithm, like sorting, to the *vector*.
- The Standard Library does not implement the algorithms as member functions of the various containers, but as free template functions that can operate on many containers
  - Implementing an iterator as separate type allows additional functionality.
    - Consistency with other containers. Pointers may not make sense to other types of containers.

Vector: iterator- element content

```
1  #include <vector>
2  #include <iostream>
3
4  using namespace std;
5
6  int main()
7  {
8      int p[]={5,4,3,2,1};
9      vector<int> v(p, p+5);
10
11     vector<int>::reverse_iterator r;
12
13     cout << v[0] << '\n';
14     cout << v[1] << '\n';
15     cout << v[2] << '\n';
16     cout << v[3] << '\n';
17     cout << v[4] << '\n';
18     cout << "=============="<< '\n';
19     cout <<"v.begin(): "<< *(v.begin()) << '\n';
20     cout <<"v.end()+1:  "<< *(v.end()-1) << '\n';
21     cout <<"v.rbegin():"<< *(v.rbegin()) << '\n';
22     cout <<"v.rend()-1:"<< *(v.end()-1) << '\n';
```



begin() and *end()* point to the first, respectively, to one-past-the-last element.

rbegin() and *rend()* point to the first, respectively, to the one-past- the-last element of the reverse sequence.

Element access

```
std::vector<int> v;
v.push_back(999);
// fill up the vector
//...
// following statements are equivalent:
int i = v.front();
int i = v[0];
int i = v.at(0);
int i = *(v.begin());
// following statements are equivalent:
int j = v.back();
int j = v[v.size()-1];
int j = v.at(v.size()-1);
int j = *(v.end()-1);
```

Vector: iterator- element content

```cpp
#include <vector>
#include <iostream>
int main()
{
    std::vector<int> q;
    q.push_back(10); q.push_back(11); q.push_back(12);

    std::vector<int> v;
    for(int i=0; i<5; ++i){
        v.push_back(i);
    }
    // v contains 0 1 2 3 4

    std::vector<int>::iterator it = v.begin() + 1;
    // insert 33 before the second element:
    it = v.insert(it, 33);
    // v contains 0 33 1 2 3 4
    // it points to the inserted element
```

Vector: iterator- element content

```
it = v.begin() + 3;
// it points to the fourth element of v
// insert three time -1 before the fourth element:
v.insert(it, 3, -1);
// v contains 0 10 11 -1 -1 -1 12 33 1 2 3 4
// iterator 'it' is invalid
// erase the fifth element of v
it = v.begin() + 4;
v.erase(it);
// v contains 0 10 11 -1 -1 12 33 1 2 3 4
// iterator 'it' is invalid
// erase the second to the fifth element:
it = v.begin() + 1;
v.erase(it, it + 4);
// v contains 0 12 33 1 2 3 4
// iterator 'it' is invalid
// clear all of v's elements
v.clear();
```

- *insert()* and *erase()* may invalidate any iterators you might hold
- Erasing elements never triggers a reallocation, nor does it influence the *capacity()*. However, all iterators that point between the first element erased and the end of the sequence become invalid.
- Calling *clear()* removes all elements from the controlled sequence. The memory allocated is not freed, however. All iterators become invalid

Comparison operations

- You can compare the contents of two vectors on an element- by-element basis using the operators ==, *!*= and <.
- Two *vectors* are equal if both have the same *size()* and the elements are correspondingly equal.
  - Note that the *capacity()* of two equal *vectors* need not to be the same. The operator < orders the *vector*'s lexicographically.

https://www.quora.com/Whats-the-difference-between-lexicographical-and-alphabetical-order

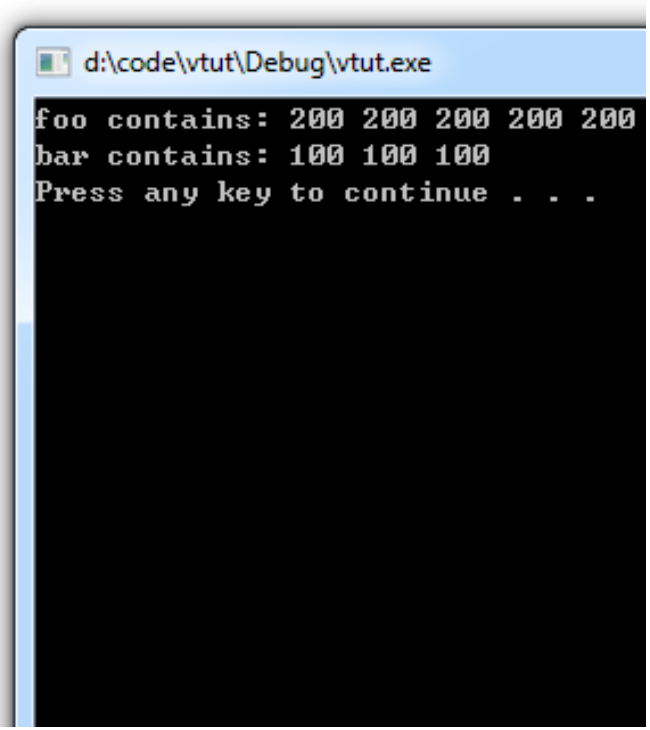
Vector: Swapping contents

- Sometimes, it is practical to be able to `swap()` the contents of two vectors.
- A common application is forcing a *vector* to release the memory it holds. We have seen that erasing the elements or clearing the *vector* doesn't influence its `capacity()` (in other words, the memory allocated).

```
std::vector<int> v;
//...
v.clear();
v.swap(std::vector<int>(v));
```

Vector: Swapping contents

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> foo (3,100);   // three ints with a value of 100
  std::vector<int> bar (5,200);   // five ints with a value of 200

  foo.swap(bar);

  std::cout << "foo contains:";
  for (unsigned i=0; i<foo.size(); i++)
    std::cout << ' ' << foo[i];
  std::cout << '\n';

  std::cout << "bar contains:";
  for (unsigned i=0; i<bar.size(); i++)
    std::cout << ' ' << bar[i];
  std::cout << '\n';

  system("pause");

  return 0;
}
```



List

- each element contains a pointer not only to the next element but also to the preceding one.
- The container stores the address of both the front and also the back elements
  - the front: the first element
  - the back: the last element
- This ensures the fast access of both ends of the list.

List: push_front, front, pop_front

```
int main(){

    list<int>* ilist;
    size_t n;

    ilist = new list<int>;
    ilist->push_back(30);
    ilist->push_back(40);
    //print_l(ilist);
    ilist->push_front(20);
    ilist->push_front(10);
    print_l(ilist);
```

```
the list elements are :
10
20
30
40
====================
the size of the list is 4
pop front:10
pop front:20
pop front:30
pop front:40
the list is empty
====================
Press any key to continue . . .
```

```
    cout<<"the size of the list is "<<ilist->size()<<endl;
    n=ilist->size();
    for(size_t i =0; i<n; i++){
    //for(size_t i =0; i<ilist->size(); i++){
        cout<<"pop front:"<<ilist->front()<<endl;
        ilist->pop_front();
    }


    print_l(ilist);


    system("pause");
    return 0;
}
```

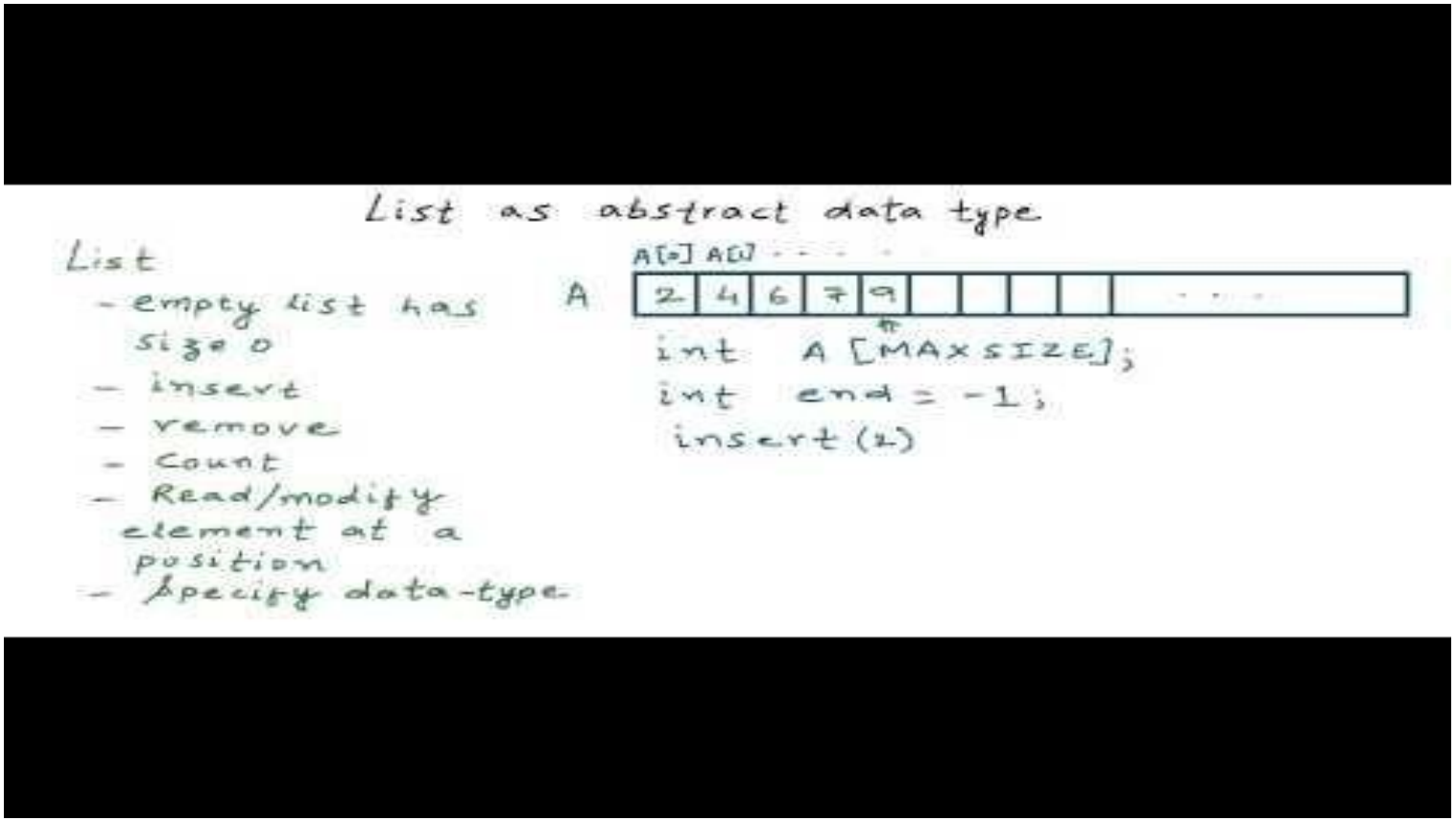
```
the list elements are :
10
20
30
40
====================
the size of the list is 4
pop front:10
pop front:20
the list elements are :
30
40
====================
Press any key to continue . . .
```

List: reverse(), merge() and unique()

```
int main(){

    list<int> ilist1, ilist2;

    int arr1[]={40,30,20,10};
    int arr2[]={15,20,30,45,55};


    for(size_t i=0; i<4; i++){
        ilist1.push_back(arr1[i]);
    }

    cout<<"List 1"<<endl;
    print_l(&ilist1);
    for(size_t j=0; j<5; j++){
        ilist2.push_back(arr2[j]);
    }
    cout<<"List 2"<<endl;
    print_l(&ilist2);

    ilist1.reverse();
    cout<<"reverse list 1 "<<endl;
    print_l(&ilist1);


    ilist1.merge(ilist2);

    cout<<"merge reverse list 1 with list 2"<<endl;

    print_l(&ilist1);

    ilist1.unique();
    cout<<"unique elements of list1"<<endl;
    print_l(&ilist1);
```

```
d:\code\ltut\Debug\ltut.exe
List 1
the list elements are :
40
30
20
10
====================
List 2
the list elements are :
15
20
30
45
55
====================
reverse list 1
the list elements are :
10
20
30
40
====================
merge reverse list 1 with list 2
the list elements are :
10
15
20
20
30
30
40
45
55
====================
unique elements of list1
the list elements are :
10
15
20
30
40
45
55
====================
```

List as an Abstract Data Type (ADT)

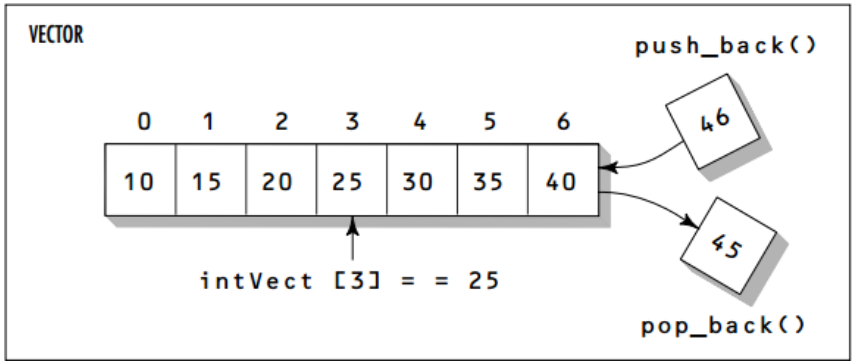


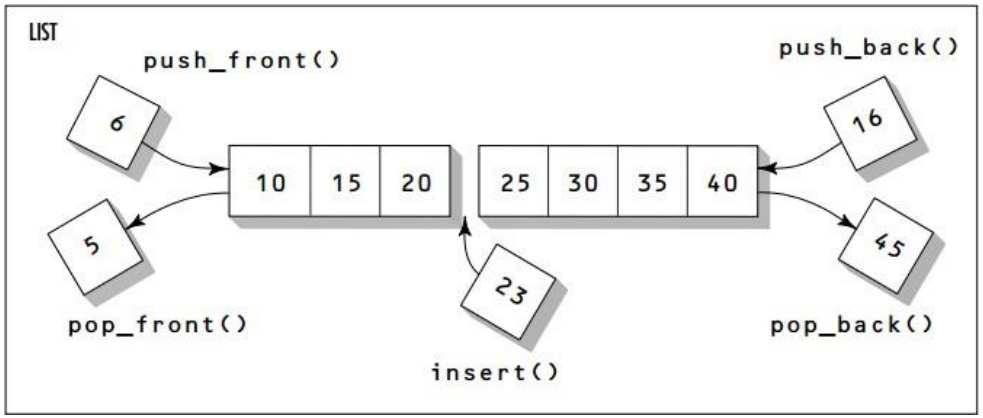
URL: https://www.youtube.com/watch?v=HdFG8L1sajw

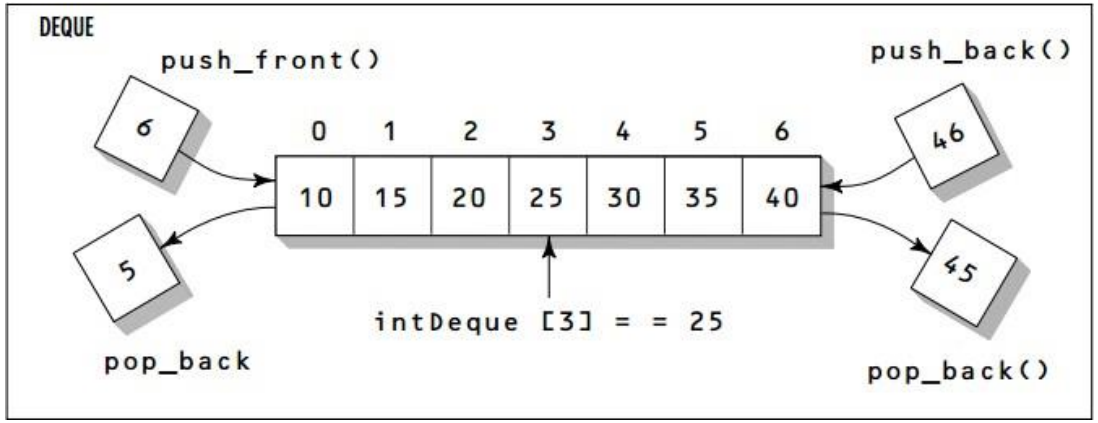
Deque

- Deque is short for Double-Ended Queue
- A deque is like
  - a vector : it supports random access using [] operator
  - a linked list: can be access from front and end
- a double ended vector and supports push_front(),pop_front () and front().
- Deque stores data in several different non-contiguous memory location.(segmented)

Vector, List and Deque







Deque

URL: https://www.youtube.com/watch?v=sQj1rmJ1q2A

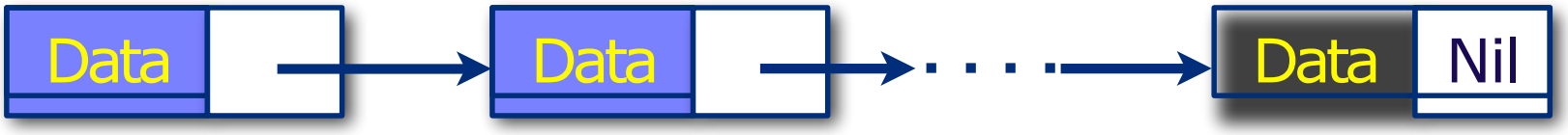
SWINBURNE UNIVERSITY OF TECHNOLOGY

Doubly Linked List

SWINBURNE UNIVERSITY OF TECHNOLOGY

What is a singly-linked list?

- A singly-linked list is a sequence of data items, each connected to the next by a pointer called next.

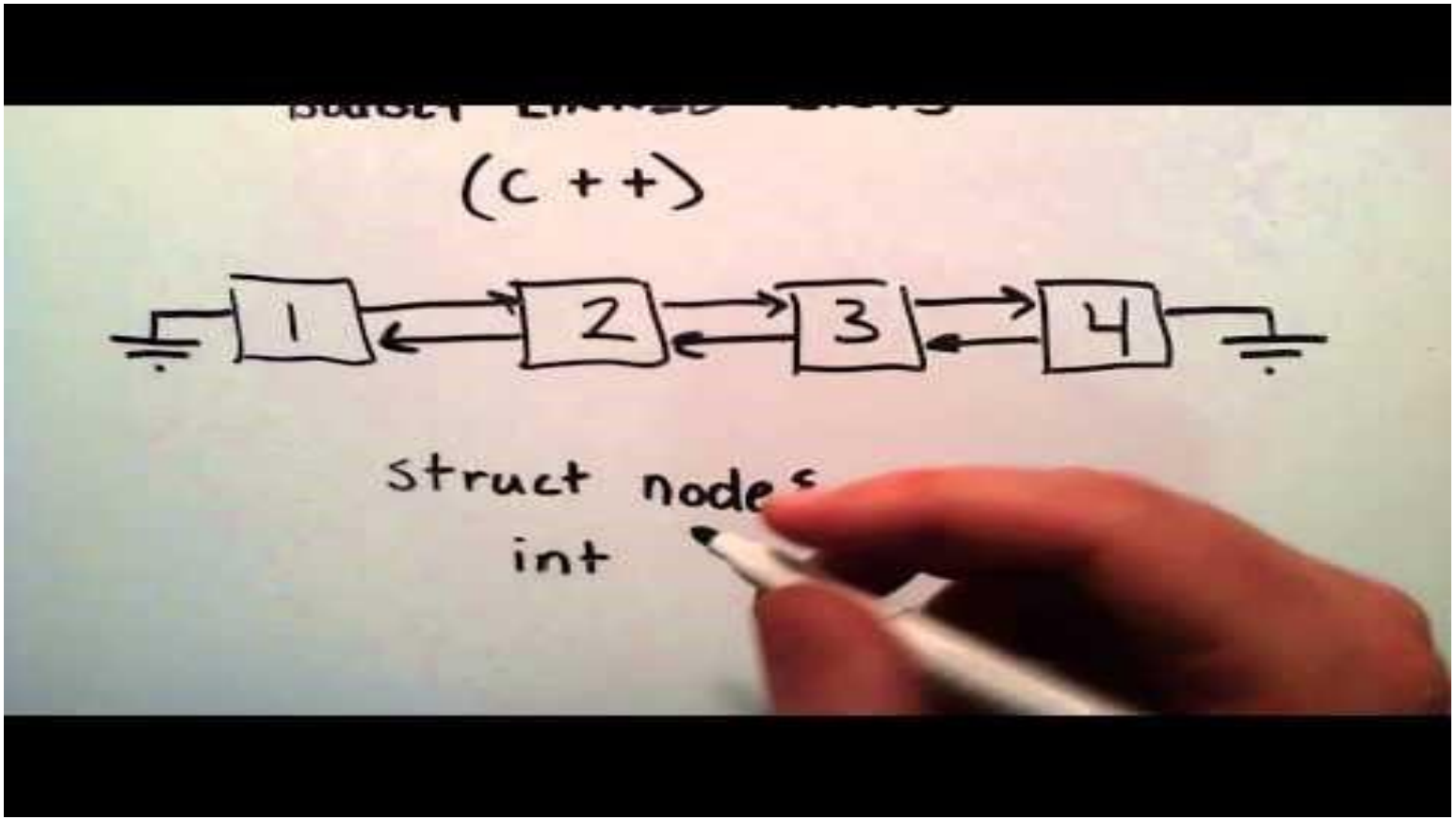


- A data item may be a primitive value, a composite value, or even another pointer.
- A singly-linked list is a recursive data structure whose nodes refers to nodes of the same type.

What is doubly-linked list?

- A doubly-linked list is a sequence of data items, each connected by two links called next and previous.
- A data item may be a primitive value, a composite value, or even another pointer.
- Traversal in a double-linked list is bidirectional.
- Deleting of a node at either end of a doubly-linked list is straight forward.

What is a Doubly Linked List



URL: https://www.youtube.com/watch?v=k0pjD12bzP0

Sentinel Node: NIL

- A sentinel node is a programming idiom used to replace null-pointers with a special token denoting "no value" or nil.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinels denote proper, yet empty, values.

A Doubly-Linked List Node

```
template<class DataType>
class DoublyLinkedNode
{
public:
    typedef DoublyLinkedNode<DataType> Node;

private:
    DataType fValue;
    Node* fNext;
    Node* fPrevious;

    DoublyLinkedNode()
    {
        fValue = DataType();
        fNext = &NIL;
        fPrevious = &NIL;
    }

public:
    static Node NIL;

    DoublyLinkedNode(const DataType& aValue);

    void prepend(Node& aNode);
    void append(Node& aNode);
    void remove();

    const DataType getValue() const;
    const Node* getNext() const;
    const Node* getPrevious() const;
};

template<class DataType>
DoublyLinkedNode<DataType> DoublyLinkedNode<DataType>::NIL;
```

Template Implementation: Variant A

```
#pragma once

template<class DataType>
class DoublyLinkedNode
{
private:

    …

  public:

    …

      void prepend( Node& aNode )                    // aNode becomes left node of this
      {
              aNode.fNext = this;                    // make this the forward pointer of aNode

          if ( fPrevious != &NIL )                   // make this's backward pointer aNode's
          {                                          // backward pointer and make previous'
              aNode.fPrevious = fPrevious;           // forward pointer aNode
              fPrevious->fNext = &aNode;
          }

          fPrevious = &aNode;                        // this' backward pointer becomes aNode
       }

    …
};
```

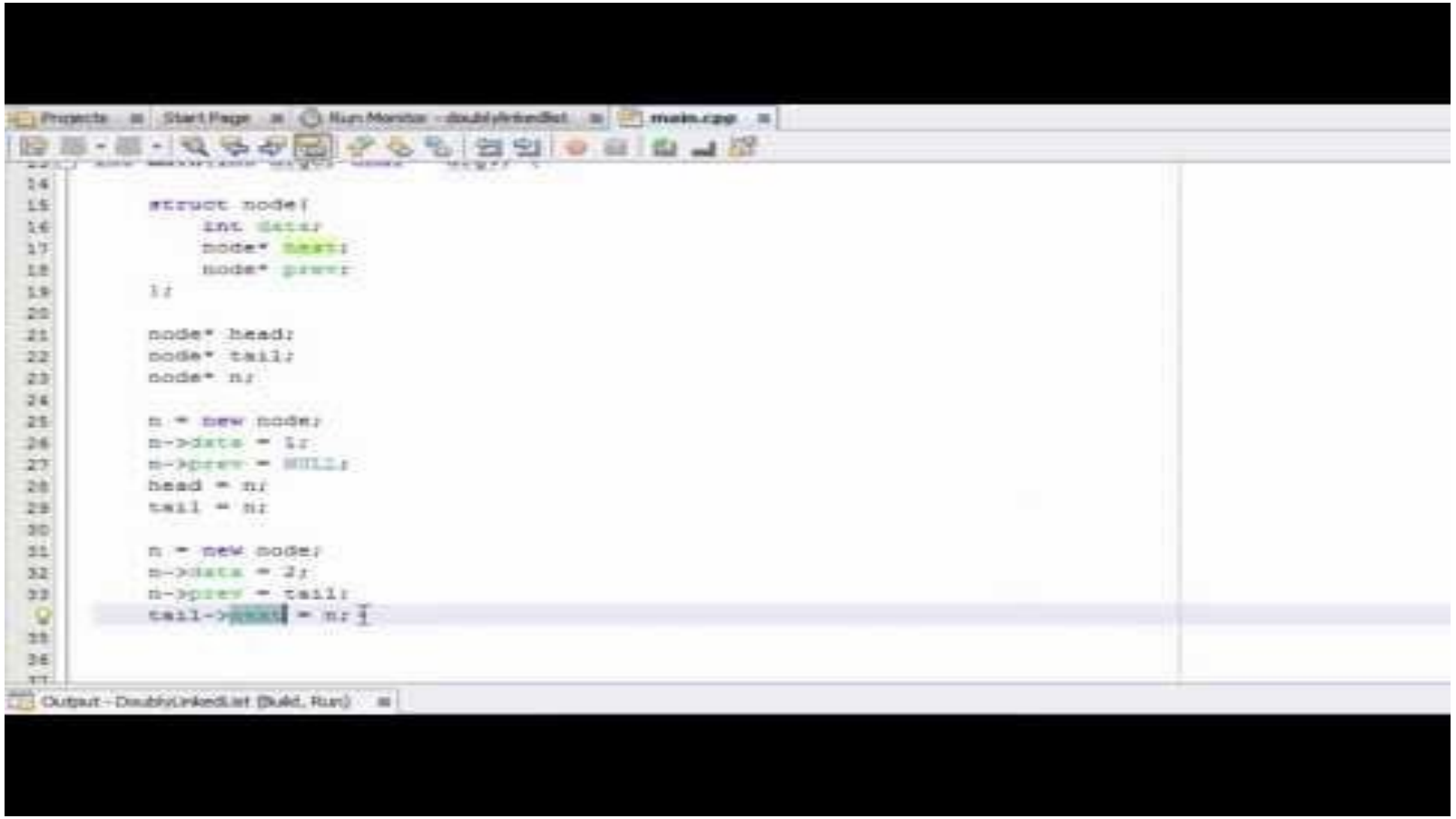
Implementation within template class specification

Template Implementation: Variant B

```
#pragma once

template<class DataType>
class DoublyLinkedNode
{
    …
};

template<class DataType>
void DoublyLinkedNode<DataType>::prepend( Node& aNode )
{
    aNode.fNext = this;                          // make this the forward pointer of aNode

    if ( fPrevious != &NIL )                     // make this's backward pointer aNode's
    {                                            // backward pointer and make previous'
        aNode.fPrevious = fPrevious;             // forward pointer aNode
        fPrevious->fNext = &aNode;
    }

    fPrevious = &aNode;                          // this' backward pointer becomes aNode
}
```

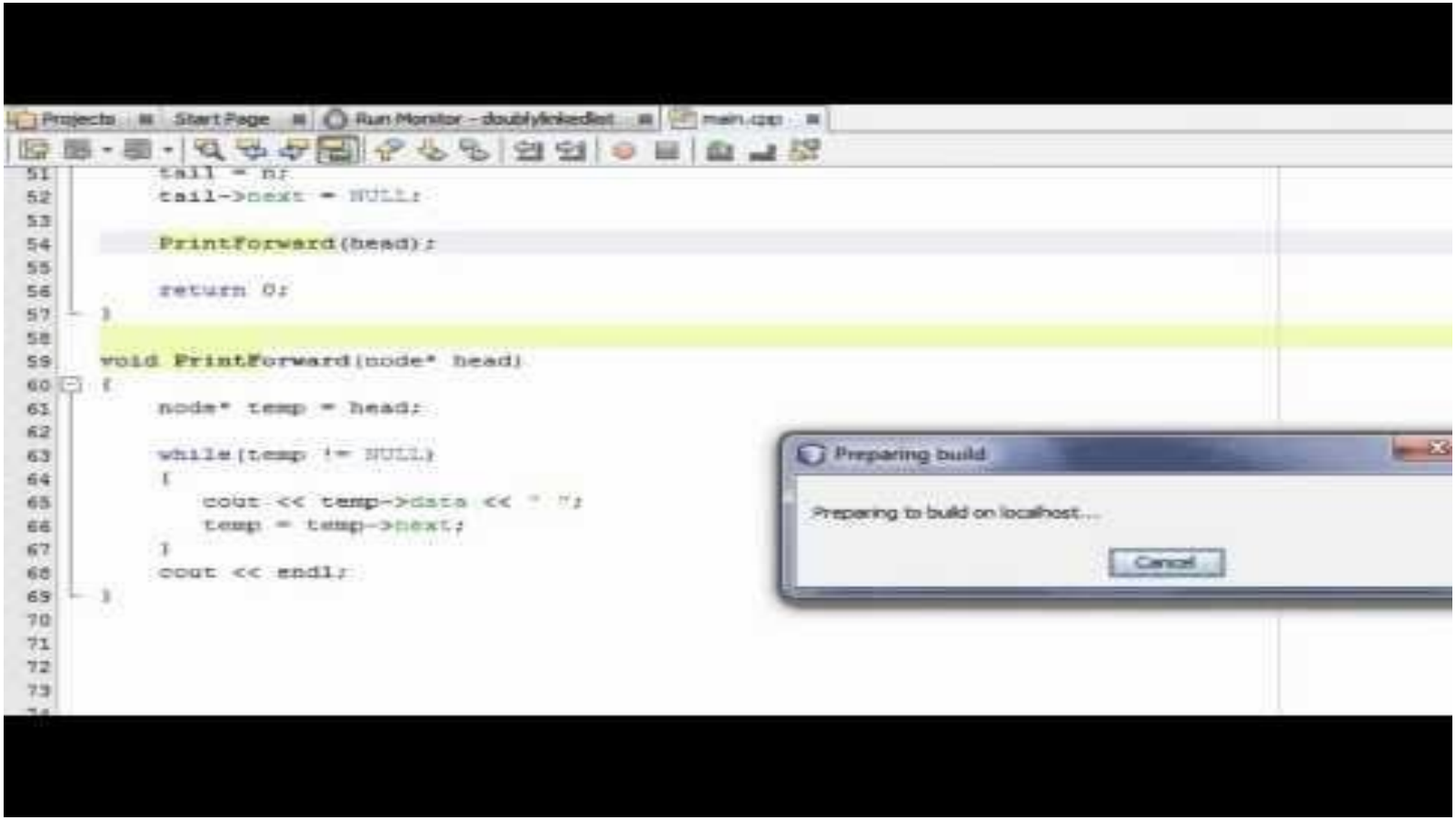
Implementation outside template class specification, but within same header file

How to Create a Doubly Linked List (Part 1)



URL: https://www.youtube.com/watch?v=YJRRpXYldVQ

How to Create a Doubly Linked List (Part 2)



URL: https://www.youtube.com/watch?v=5s0x8bc9DvQ

SWINBURNE UNIVERSITY OF TECHNOLOGY

End of ADT, Stack, STL and Doubly Linked List