Intermediate Programming
Day 23

Outline

- Templates and the STL
- `std::vector`
- Iterators
- Review questions

Templates (overview)

Q: What is a *template*?

A: Consider an example:

- What needs to change if I want to make the payload member a float/string/etc. instead of an int?

```
…
using std::cout;
using std::endl;


struct Node
{
	int payload;
	Node* next;
};


void print( const Node< T >* n )
{
	cout << n->payload << " ";
	if( next ) print( n->next );
	else	   cout << endl;
}

int main( void )
{
	Node< int >* n;
	…
	print( n );
}
```

Templates (overview)

Q: What is a *template*?

A: Templates are a way of writing a generic class (Node) or function (print) and allowing it to work with a parameterized family of types*

⇒ Instead of defining a single class / function, we provide the compiler with a recipe for generating the class / function for whichever type we need, as we need it

*More on this later

```
…
using std::cout;
using std::endl;

template< class PType >
struct Node
{
	PType payload;
	Node* next;
};

template< class T >
void print( const Node< T >* n )
{
	cout << n->payload << " ";
	if( next ) print( n->next );
	else	   cout << endl;
}

int main( void )
{
	Node< int >* n;
	…
	print( n );
}
```

Templates (overview)

In this example:

- Node is now a *template* class, parameterized by a type referred to as PType

```
…
using std::cout;
using std::endl;

template< class PType >
struct Node
{
	PType payload;
	Node* next;
};

template< class T >
void print( const Node< T >* n )
{
	cout << n->payload << " ";
	if( next ) print( n->next );
	else	cout << endl;
}

int main( void )
{
	Node< int >* n;
	…
	print( n );
}
```

Templates (overview)

In this example:

- Node is now a *template* class, parameterized by a type referred to as PType
  - We declare payload to be of generic type PType

```
…
using std::cout;
using std::endl;

template< class PType >
struct Node
{
      PType payload;
      Node* next;
};

template< class T >
void print( const Node< T >* n )
{
      cout << n->payload << " ";
      if( next ) print( n->next );
      else       cout << endl;
}

int main( void )
{
      Node< int >* n;
      …
      print( n );
}
```

Templates (overview)

In this example:

- Node is now a *template* class, parameterized by a type referred to as PType
- When declaring a variable of type Node, we specify the payload type, in angle brackets

```
…
using std::cout;
using std::endl;

template< class PType >
struct Node
{
	PType payload;
	Node* next;
};

template< class T >
void print( const Node< T >* n )
{
	cout << n->payload << " ";
	if( next ) print( n->next );
	else	cout << endl;
}

int main( void )
{
	Node< int >* n;
	…
	print( n );
}
```

Templates (overview)

In this example:

- Node is now a *template* class, parameterized by a type referred to as PType
- When declaring a variable of type Node, we specify the payload type, in angle brackets
- When defining a function with a generic Node argument, the function is templated by Node's parameter
  - We are creating a recipe for the function
  - Note:
    We do not need to use the same parameter name
    T ≠ PType

```
…
using std::cout;
using std::endl;

template< class PType >
struct Node
{
      PType payload;
      Node* next;
};

template< class T >
void print( const Node< T >* n )
{
      cout << n->payload << " ";
      if( next ) print( n->next );
      else       cout << endl;
}

int main( void )
{
      Node< int >* n;
      …
      print( n );
}
```

Templates (overview)

In this example:

- Node is now a *template* class, parameterized by a type referred to as PType
- When declaring a variable of type Node, we specify the payload type, in angle brackets
- When defining a function with a generic Node argument, the function is templated by Node's parameter
  - We specify Node's parameter to be the generic type T

```
…
using std::cout;
using std::endl;

template< class PType >
struct Node
{
      PType payload;
      Node* next;
};

template< class T >
void print( const Node< T >* n )
{
      cout << n->payload << " ";
      if( next ) print( n->next );
      else       cout << endl;
}

int main( void )
{
      Node< int >* n;
      …
      print( n );
}
```

The Standard Template Library

- The Standard Template Library (STL) is C++'s compendium of useful data structures and algorithms
  - `pair` – pair of values (possibly of different types)
  - `tuple` – a tuple of values (possibly of different types and arbitrary long)
  - `list` – linked list!
  - `vector` – dynamically-sized array
  - `array` – fixed-length array (not as useful as vector)
  - `map` – associative list, i.e. dictionary
  - `stack` – last-in first-out (LIFO)
  - `deque` – double-ended queue, flexible combo of LIFO/FIFO
  - and many more

Outline

- Templates and the STL
- `std::vector`
- Iterators
- Review questions

std::vector

A dynamically sized array of elements:

- The template parameter specifies the element type

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cin;
using std::cout;
int main( void )
{
	vector< float > grades;
	float grade;
	while( cin >> grade ) grades.push_back( grade );
	grades.insert( grades.begin() , 100 );
	cout << "First grade was " << grades[1] << endl;
	cout << "Last grade was " << grades[ grades.size()-1 ] << endl;
	return 0;
}
```

std::vector

- push_back:
  - inserts an element at the end
- insert
  - Insert an element before the prescribed position*
- [ ] operator:
  - gives access to an element at a prescribed index
- size:
  - returns the number of elements in the vector

*More on this shortly

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cin;
using std::cout;
int main( void )
{
	vector< float > grades;
	float grade;
	while( cin >> grade ) grades.push_back( grade );
	grades.insert( grades.begin() , 100 );
	cout << "First grade was " << grades[1] << endl;
	cout << "Last grade was " << grades[ grades.size()-1 ] << endl;
	return 0;
}
```

std::vector

- back:
  - returns the last element
- pop_back:
  - removes the last element
- resize:
  - resizes the vector to be able to store a specified number of elements
- erase
- clear
- at
- empty

See:
http://www.cplusplus.com/reference/vector/vector/
for more std::vector functionality

std::vector

Using the [ ] operator and the **size** method, we can iterate over the entries of a **vector**

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;
int main( void )
{
	vector< int > values;
	values.push_back( 1 );
	values.push_back( 3 );
	values.push_back( 2 );
	for( int i=0 ; i<values.size() ; ++i )
		cout << values[i] << " ";
	cout << endl;
	return 0;
}
```

```
>> ./a.out
1 3 2
>>
```

std::vector

Using the [ ] operator and the **size** method, we can iterate over the entries of a **vector**

Or we can use **iterators**

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;
int main( void )
{
	vector< int > values;
	values.push_back( 1 );
	values.push_back( 3 );
	values.push_back( 2 );
	for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it )
		cout << *it << " ";
	cout << endl;
	return 0;
}
```

```
>> ./a.out
1 3 2
>>
```

std::vector

Using the [ ] operator and the **size** method, we can iterate over the entries of a **vector**

Or we can use **iterators**:

- The container defines its own iterator sub-class

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;
int main( void )
{
	vector< int > values;
	values.push_back( 1 );
	values.push_back( 3 );
	values.push_back( 2 );
	for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it )
		cout << *it << " ";
	cout << endl;
	return 0;
}
```

```
>> ./a.out
1 3 2
>>
```

std::vector

Using the [ ] operator and the **size** method, we can iterate over the entries of a **vector**

Or we can use **iterators**:

- The container defines its own iterator sub-class
- Iterators are "clever pointers" that know how to move over elements of a container
  - Container could be simple (e.g. array) or more complicated (e.g. linked list) or very complicated (e.g. tree)

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;
int main( void )
{
	vector< int > values;
	values.push_back( 1 );
	values.push_back( 3 );
	values.push_back( 2 );
	for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it )
		cout << *it << " ";
	cout << endl;
	return 0;
}
```

```
>> ./a.out
1 3 2
>>
```

Outline

- Templates and the STL
- `std::vector`
- Iterators
- Review questions

Forward iterators

```
vector< int > values;
…
for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The forward iterator has type T::iterator

Forward iterators

```
vector< int > values;
…
for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The forward iterator has type T::iterator
- The container defines a T::begin method
  - Returns an iterator to the first element in the container

Forward iterators

```
vector< int > values;
…
for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The forward iterator has type T::iterator
- The container defines a T::begin method
- The container defines a T::end method
  - Returns an iterator to the element just past the last element in the container

Forward iterators

```
vector< int > values;
…
for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The forward iterator has type T::iterator
- The container defines a T::begin method
- The container defines a T::end method
- The iterator overloads the pre-increment operator ++
  - Advances the iterator to the next element

Forward iterators

```
vector< int > values;
…
for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The forward iterator has type T::iterator
- The container defines a T::begin method
- The container defines a T::end method
- The iterator overloads the pre-increment operator ++
- The iterator overloads the inequality operator !=
  - Checks if two iterators are different

Forward iterators

```
vector< int > values;
…
for( vector< int >::iterator it=values.begin() ; it!=values.end() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The forward iterator has type T::iterator
- The container defines a T::begin method
- The container defines a T::end method
- The iterator overloads the pre-increment operator ++
- The iterator overloads the inequality operator !=
- The iterator overloads the dereference operator *
  - Returns the contents of what the iterator is “pointing to”

Reverse iterators

```
vector< int > values;
…
for( vector< int >::reverse_iterator it=values.rbegin() ; it!=values.rend() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The reverse iterator has type T::reverse_iterator
- The container defines a T::rbegin method
- The container defines a T::rend method

Reverse iterators

```
vector< int > values;
…
for( vector< int >::reverse_
```

For an STL contain

- The reverse iter
- The container d
- The container d

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;
int main( void )
{
	vector< int > values;
	values.push_back( 1 );
	values.push_back( 3 );
	values.push_back( 2 );
	for( vector< int >::reverse_iterator it=values.rbegin() ; it!=values.rend() ; ++it )
		cout << *it << " ";
	cout << endl;
	return 0;
}
```

```
>> ./a.out
2 3 1
>>
```

Constant iterators

```
vector< int > values;
…
for( vector< int >::const_iterator it=values.cbegin() ; it!=values.cend() ; ++it ) cout << *it << " ";
```

For an STL container of type T (in this example , T=vector<int>):

- The constant iterator has type T::const_iterator
  - The contents of the container cannot be modified
- The container defines a T::cbegin method
- The container defines a T::cend method

Constant iterators

```
vector< int > values;
…
for( vector< int >::const_it
```

For an STL contain

- The constant ite
  - The contents c
- The container d
- The container d

main.cpp

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;
int main( void )
{
	vector< int > values;
	values.push_back( 1 );
	values.push_back( 3 );
	values.push_back( 2 );
	for( vector< int >::const_iterator it=values.cbegin() ; it!=values.cend() ; ++it )
		cout << *it << " ";
	cout << endl;
	return 0;
}
```

```
>> ./a.out
1 3 2
>>
```

Iterators

In general, iterators act like “smart” pointers, allowing us to iterate through the contents of a container and get its values.

For iterators `iter1` and `iter2`, supported operations include:

- Increment: `iter1++` or `++iter1`
- Dereference: `*iter`
- Assignment: `iter1=iter2`
- Comparison: `iter1!=iter2` or `iter1==iter2`

Random access iterators

Like pointers, some iterators also support arithmetic (random access).

For iterators `iter1` and `iter2` and integer `n` supported operations include:

- Arithmetic: `iter1=iter2+n` or `iter1=iter2-n`
- Compound arithmetic: `iter1+=n` or `iter1-=n`
- Comparison: `iter1<=iter2`, `iter1>iter2`, etc.
- Differencing: `n=iter2-iter1`

Note:

Not all iterators support random access:

✓ Iterators for vectors do

✗ Iterators for linked lists do not

Outline

- Templates and the STL
- `std::vector`
- Iterators
- Review questions

Review questions

1. What is a template in C++?

A way of writing an object (generalization of a struct) so that they can work with any type

Review questions

2. What is the standard template library?

A collection of standardly used, templated objects and functions

Review questions

3. How do you iterate a `std::vector` and print out its elements?

```
for( int i=0 ; i<v.size() ; i++ ) std::cout << v[i] << std::endl;
```

or

```
for( vector<type>::iterator it=v.begin() ; it!=v.end ; it++ )
    std::cout << *it << std::endl;
```

Review questions

4. What is an iterator in C++?

Clever pointers that know how to move over the components of a data structure (e.g. support increment and dereferencing)

Review questions

5. How do you add an element to an existing `std::vector`?

Use the `push_back` or `insert` method.

Review questions

6. (Bonus) What is the output of this program?

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

v = {}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=1

v = {0.5}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=2

v = {0.5, 4.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=3

v = {1.5, 0.5, 4.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=4

v = {1.5, 0.5, 4., 8.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=5

v = {2.5, 1.5, 0.5, 4., 8.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=6

v = {2.5, 1.5, 0.5, 4., 8., 12.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=7

v = {3.5, 2.5, 1.5, 0.5, 4., 8., 12.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=8

v = {3.5, 2.5, 1.5, 0.5, 4., 8., 12., 16.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

i=9

v = {4.5, 3.5, 2.5, 1.5, 0.5, 4., 8., 12., 16.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?

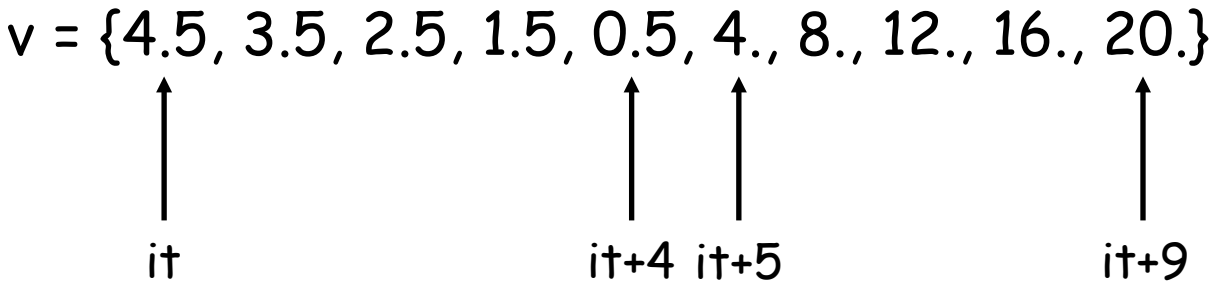
i=10

v = {4.5, 3.5, 2.5, 1.5, 0.5, 4., 8., 12., 16., 20.}

```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Review questions

6. (Bonus) What is the output of this program?




```
#include <iostream>
#include <vector>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
int main( void )
{
	vector< double > v;
	for( int i=1 ; i<=10 ; i++ )
	{
		if( i%2==1 ) v.insert( v.begin() , i/2.0 );
		else v.push_back( i*2.0 );
	}
	vector< double >::iterator it = v.begin();
	cout << "first == " << *it << endl;
	cout << "middle1 == " << *(it + 4) << endl;
	cout << "middle2 == " << *(it + 5) << endl;
	cout << "last == " << *(it + 9) << endl;
}
```

Exercise 23

- Website -> Course Materials -> Exercise 23