

CS100

Introduction to Programming

Lectures 18. STL containers

Today's learning objectives

- Basic introduction to STL data structures:
 - Vector
 - List
 - Map

Outline

- Refresher
- STL vector
- STL list
- STL pairs and map

The C++ Standard Template Libraries

- In 1990, Alex Stepanov and Meng Lee of HP Laboratories extended C++ with a library of class and function templates which has come to be known as the STL
- In 1994, STL was adopted as part of ANSI/ISO Standard C++

The C++ Standard Template Libraries

- STL had three basic components:
 - Containers
 - Generic class templates to store data
 - Algorithms
 - Generic function templates to operate on containers
 - Iterators
 - Generalized 'smart' pointers that facilitate use of containers
 - They provide an interface that is needed for STL algorithms to operate on STL containers
- **String abstraction was added during standardization**

Why use STL?

- STL
 - offers an assortment of containers
 - releases containers' time/storage complexity
 - containers grow/shrink in size automatically
 - provides built-in algorithms to process containers
 - provides iterators that make the containers and algorithms flexible and efficient.
 - is extensible which means that users can add new containers and new algorithms such that
 - algorithms can process STL containers as well as user defined containers
 - User defined algorithms can process STL containers as well as user defined containers

Standard Template Library

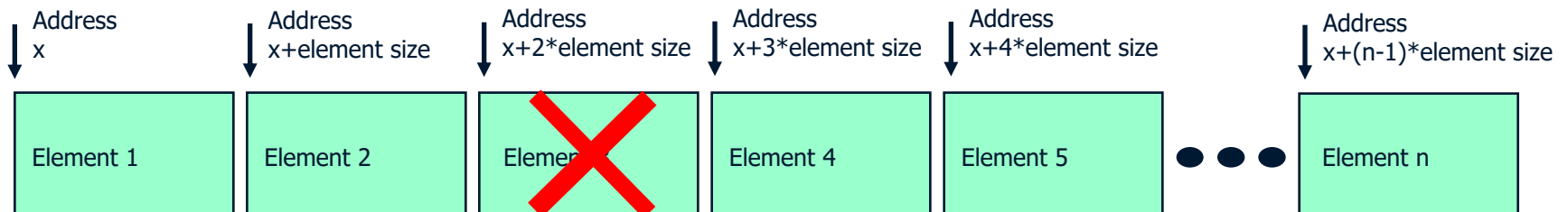
- Uses template mechanism for generic ...
 - ... containers (classes)
 - Data structures that hold anything
 - Ex.: `list`, `vector`, `map`, `set`
 - ... algorithms (functions)
 - handle common tasks (searching, sorting, comparing, etc.)
 - Ex.: `find`, `merge`, `reverse`, `sort`, `count`, `random shuffle`, `remove`, `nth-element`, `rotate`, ...

Outline

- Refresher
- **STL vector**
- STL list
- STL pairs and map

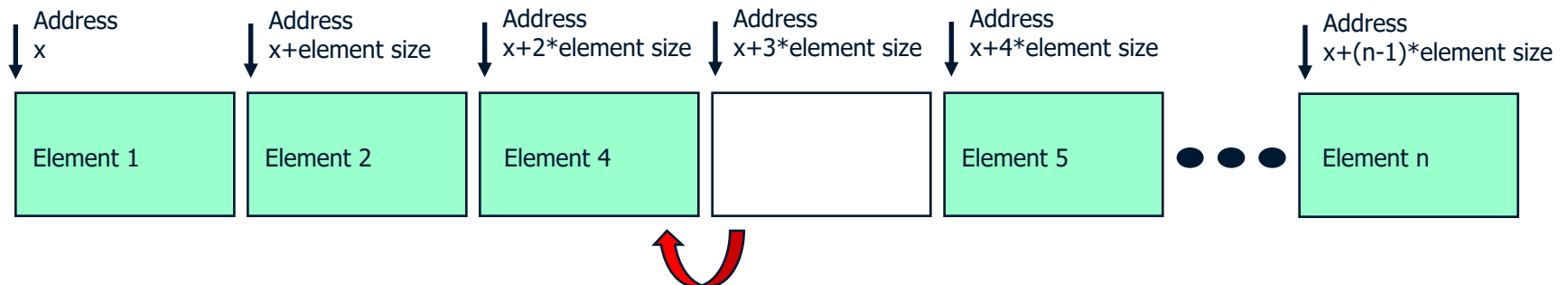
Vector

- Provides an alternative to the built in array
- A vector is self grown (dynamic in size)
- Use it instead of the built in array!
- Contiguous placement in memory
 - Constant-time look-up given known element size!
 - Expensive when adding/removing elements!



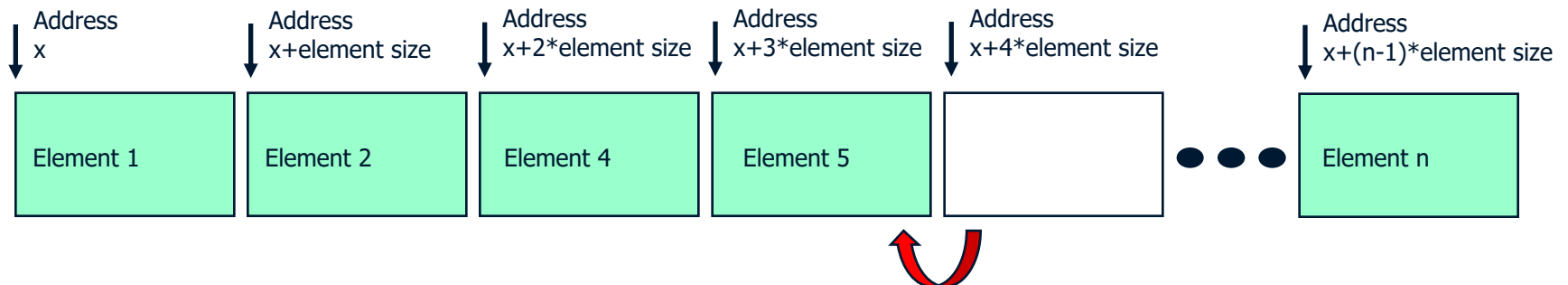
Vector

- Provides an alternative to the built in array
- A vector is self grown (dynamic in size)
- Use it instead of the built in array!
- Contiguous placement in memory
 - Constant-time look-up given known element size!
 - Expensive when adding/removing elements!



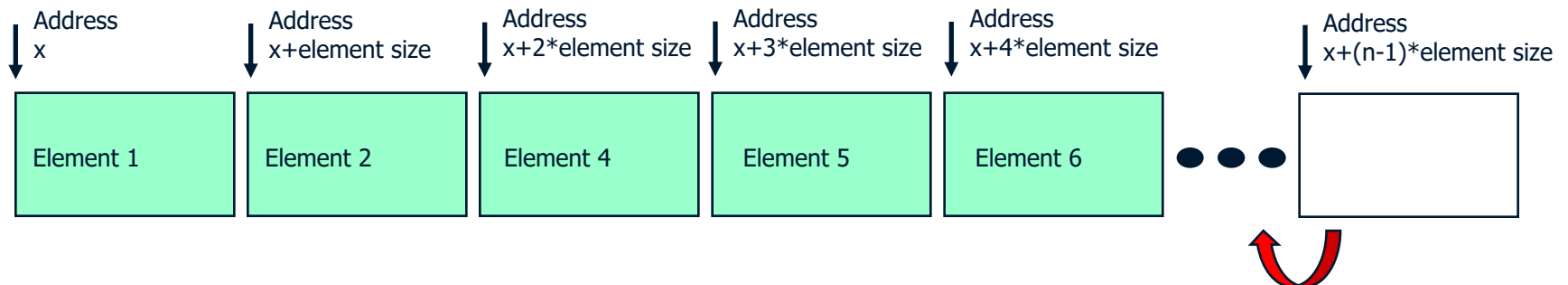
Vector

- Provides an alternative to the built in array
- A vector is self grown (dynamic in size)
- Use it instead of the built in array!
- Contiguous placement in memory
 - Constant-time look-up given known element size!
 - Expensive when adding/removing elements!



Vector

- Provides an alternative to the built in array
- A vector is self grown (dynamic in size)
- Use it instead of the built in array!
- Contiguous placement in memory
 - Constant-time look-up given known element size!
 - Expensive when adding/removing elements!



Defining a new vector

- Syntax:
 - `vector<of what>`
- For example :
 - `vector<int>` - vector of integers
 - `vector<string>` - vector of strings
 - `vector<int * >` - vector of pointers to integers
 - `vector<Shape>` - vector of Shape objects, where Shape is a user defined class

Using Vector

- `#include <vector>`
- Two ways to use the vector type:
 - Array style
 - STL style

Using a Vector – Array Style

- We mimic the use of the C-style array

```
void simple_example()
{
    const int N = 10;
    vector<int> ivec(N);
    for (int i=0; i < 10; ++i)
        cin >> ivec[i];

    int ia[N];
    for (int j = 0; j < N; ++j)
        ia[j] = ivec[j];
}
```

Using a vector – STL style

- We define an empty vector
`vector<string> svec;`
- we insert elements into the vector using the method `push_back`

```
string word;  
while ( cin >> word ) //# words "unlimited"  
{  
    svec.push_back(word) ;  
}
```


Insertion

```
void push_back(const T& x) ;
```

- Inserts an element with value x at the end of the controlled sequence

- Example:

```
svec.push_back(str) ;
```

Size

```
unsigned int size();
```

- Returns the length of the controlled sequence (how many items it contains)

- Example

```
unsigned int size = svec.size();
```

Example

- A program that read integers from the user, sorts them, and prints the result
- Requirements:
 - Easy way to read in input
 - A “place” to store the input
 - A way to sort the stored input
 - Easy way to print the input

Using STL

```
int main()  
{  
    int input;  
    vector<int> ivec;  
  
    /* rest of code */  
}
```

STL - Input

```
while ( cin >> input )  
    ivec.push_back(input) ;
```

STL - Sorting

```
sort(ivec.begin(), ivec.end());
```

- Sort prototype:

```
void sort(Iterator first, Iterator last);
```

- What are iterators?

Iterators

- Provide a **general way for accessing** each element in sequential (vector, list) or associative (map, set) containers

Pointer Semantics

- Let `iter` be an iterator then :
 - `++iter` (or `iter++`)
Advances the iterator to the next element
 - `*iter` returns element addressed by the iterator

Begin and End

- Each container provide a **begin()** and **end()** member functions
 - **begin()** returns an iterator that addresses the first element of the container
 - **end()** returns an iterator that addresses 1 past the last element

Iterating Over Containers

- Iterating over the elements of any container type

```
for ( iter = container.begin() ;  
      iter != container.end() ;  
      ++iter )  
{  
    // do something with the element  
}
```

STL - Output

```
for ( int i = 0; i < ivec.size(); ++i )  
    cout << ivec[i] << " ";  
cout << endl;
```

- Or (more recommended):

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

STL - Include files

```
#include <iostream>    // I/O
#include <vector>        // container
#include <algorithm>    // sorting

using namespace std;
```

Putting it all together

```
int main() {  
    int input;  
    vector<int> ivec;  
  
    // input  
    while (cin >> input )  
        ivec.push_back(input);  
  
    // sorting  
    sort(ivec.begin(), ivec.end());  
  
    // output  
    vector<int>::iterator it;  
    for ( it = ivec.begin();  
          it != ivec.end(); ++it ) {  
        cout << *it << " ";  
    }  
    cout << endl;  
  
    return 0;  
}
```

Operations on vector

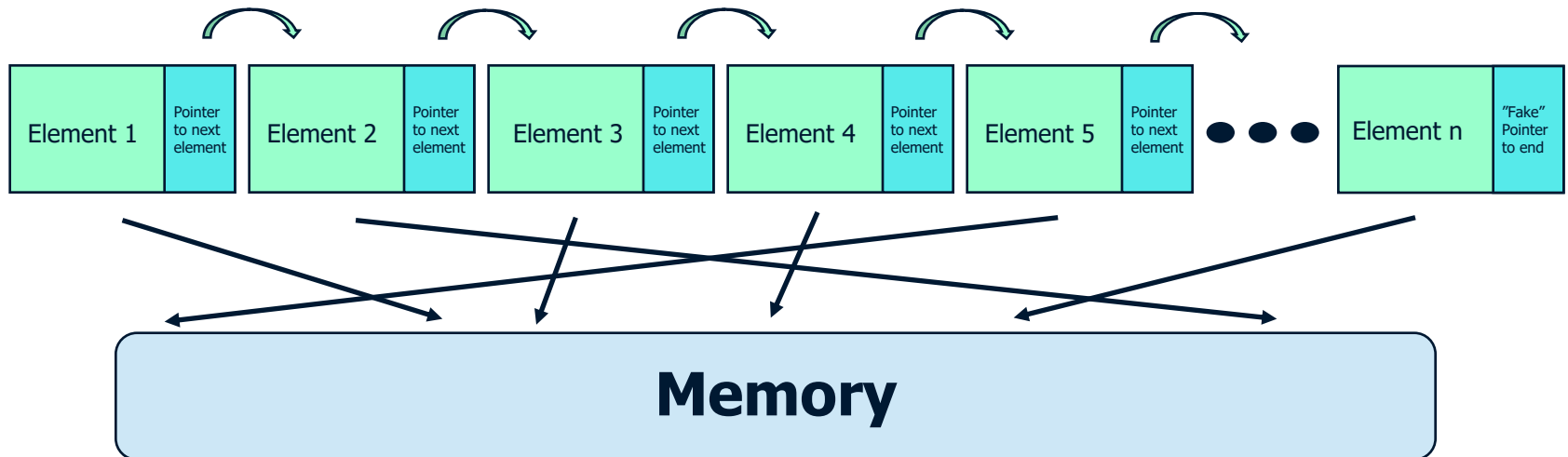
- `iterator begin()` ;
- `iterator end()` ;
- `bool empty()` ;
- `void push_back(const T& x)` ;
- `iterator erase(iterator it)` ;
- `iterator erase(iterator first, iterator last)` ;
- `void clear()` ;
- ...

Outline

- Refresher
- STL vector
- **STL list**
- STL pairs and map

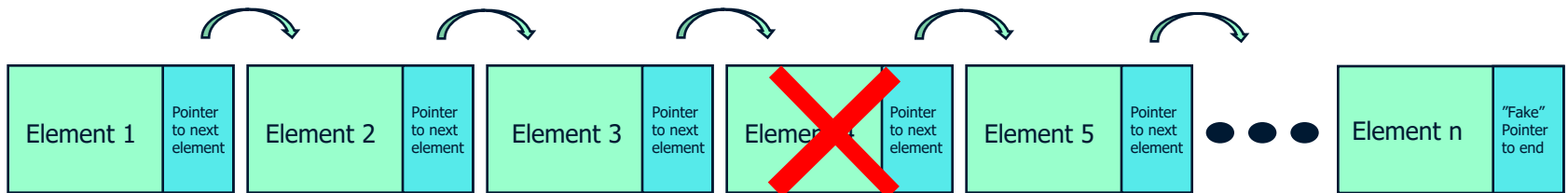
List

- Linked list
- Arbitrary location of elements in memory
 - Expensive to access n^{th} element (have to iterate)



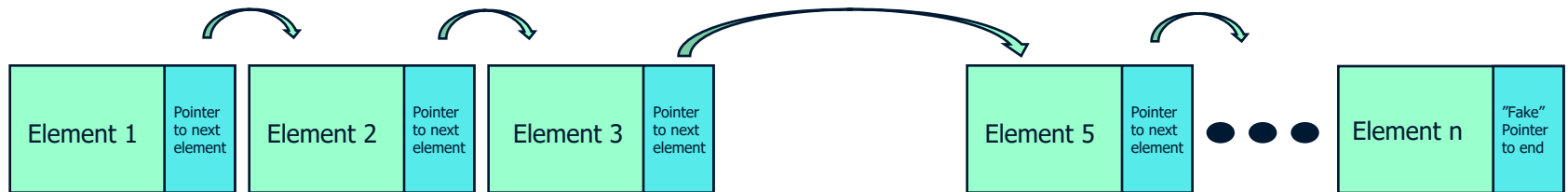
List

- Linked list
- Arbitrary location of elements in memory
 - Expensive to access n^{th} element (have to iterate)
 - Easy to add/remove elements!



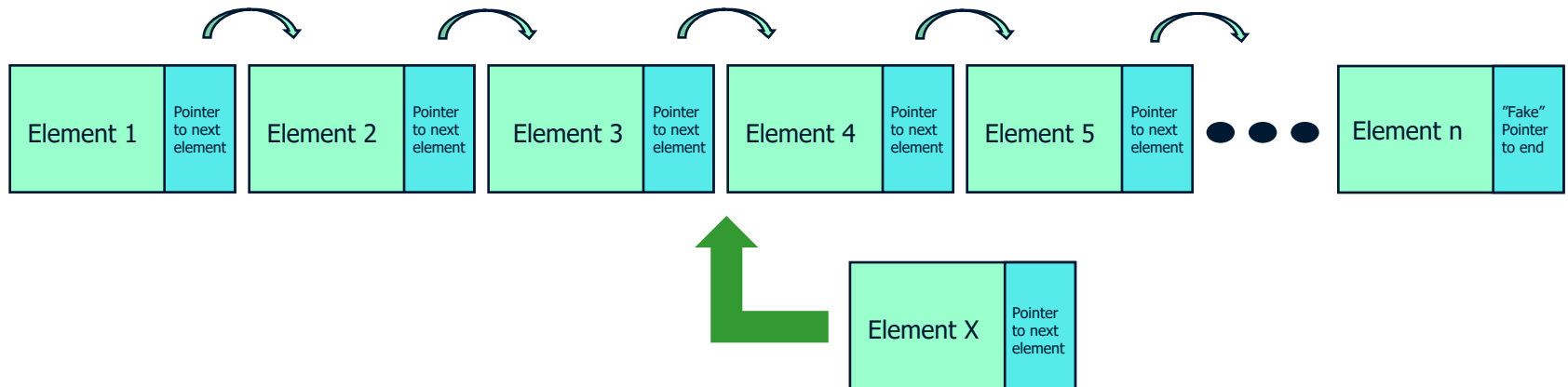
List

- Linked list
- Arbitrary location of elements in memory
 - Expensive to access n^{th} element (have to iterate)
 - Easy to add/remove elements!



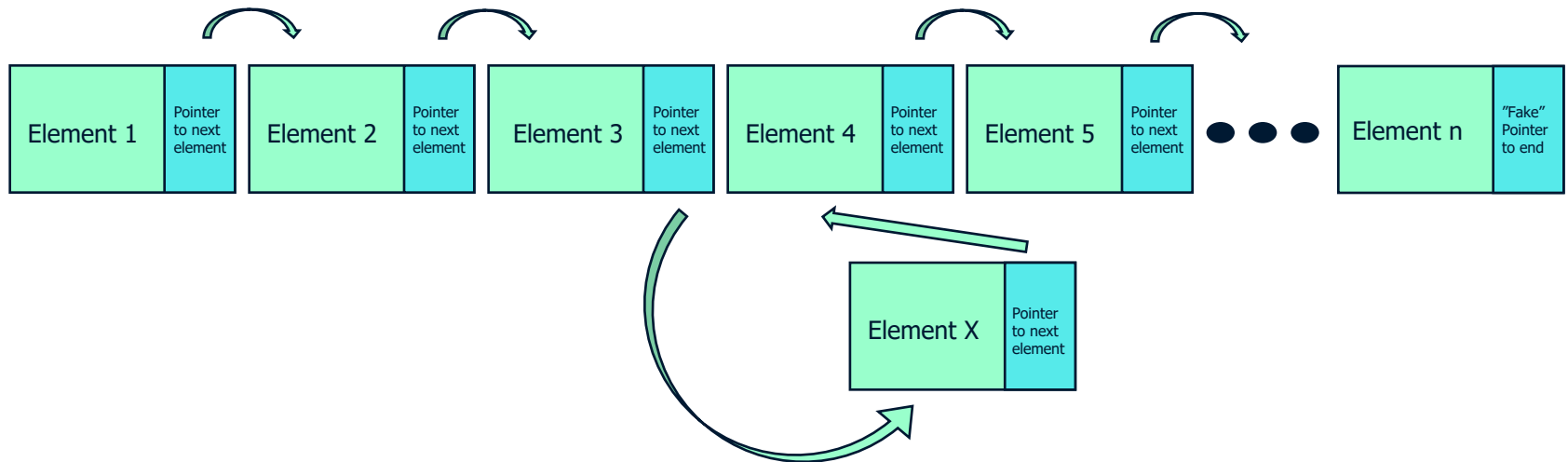
List

- Linked list
- Arbitrary location of elements in memory
 - Expensive to access n^{th} element (have to iterate)
 - Easy to add/remove elements!



List

- Linked list
- Arbitrary location of elements in memory
 - Expensive to access n^{th} element (have to iterate)
 - Easy to add/remove elements!



Insertion

```
void push_back(const T& x) ;
```

- Inserts an element with value x at the end of the list

- Example:

```
slist.push_back(str) ;
```

Size

```
unsigned int size();
```

- Returns the number of elements in the list
 - Example

```
unsigned int size = slist.size();
```

Back to our example

- A program that read integers from the user, sorts them, and prints the result
- Requirements:
 - Easy way to read in input
 - A “place” to store the input
 - A way to sort the stored input
 - Easy way to print the input

Using STL

```
int main()  
{  
    int input;  
    list<int> ilist;  
  
    /* rest of code */  
}
```


STL – Input and sorting

- No random access iterators!

```
while ( cin >> input )  
    ilist.push_back(input) ;  
  
ilist.sort() ;
```

STL - Output

- Not possible to use like array anymore!

```
list<int>::iterator it;  
for ( it = ilist.begin(); it != ilist.end(); ++it )  
    cout << *it << " ";  
cout << endl;
```

STL - Include files

```
#include <iostream>    // I/O
#include <list>          // container
#include <algorithm>    // sorting

using namespace std;
```

Putting it all together

```
int main() {
    int input;
    list<int> ilist;

    // input
    while (cin >> input )
        ilist.push_back(input);

    // sorting
    ilist.sort();

    // output
    list<int>::iterator it;
    for ( it = ilist.begin();
          it != ilist.end(); ++it ) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Operations on list

- No difference!
- `iterator begin()` ;
- `iterator end()` ;
- `bool empty()` ;
- `void push_back(const T& x)` ;
- `iterator erase(iterator it)` ;
- `iterator erase(iterator first, iterator last)` ;
- `void clear()` ;
- ...

vector vs list

Where are the bottlenecks?

- Sorting: needs easy access to n^{th} element → use **vector**!
- Storing: number of elements unknown → use **list**!

```
int main() {  
    int input;  
    vector<int> ivec;  
  
    // input  
    while (cin >> input )  
        ivec.push_back(input);  
  
    // sorting  
    sort(ivec.begin(), ivec.end());  
  
    // output  
    vector<int>::iterator it;  
    for ( it = ivec.begin();  
          it != ivec.end(); ++it ) {  
        cout << *it << " ";  
    }  
    cout << endl;  
  
    return 0;  
}
```



```
int main() {  
    int input;  
    list<int> ilist;  
  
    // input  
    while (cin >> input )  
        ilist.push_back(input);  
  
    // sorting  
    ilist.sort();  
  
    // output  
    list<int>::iterator it;  
    for ( it = ilist.begin();  
          it != ilist.end(); ++it ) {  
        cout << *it << " ";  
    }  
    cout << endl;  
  
    return 0;  
}
```



vectors and unknown size

- What is the problem?
 - Needs to be continuous memory block!
 - If pushing back 1 element, and space is insufficient
 - → Entire vector is copied to different place
- How to prevent?
 - Reserve sufficient size upfront!

```
// input
ivec.reserve(100);
while (cin >> input )
    ivec.push_back(input);
```

Outline

- Refresher
- STL vector
- STL list
- STL pairs and map

Example

Declaration:

```
class Employee {  
public:  
    // Constructors:  
    Employee ();  
    Employee ( string & name );  
  
    // Member functions ....:  
    void set_salary( int salary);  
    int salary();  
    void set_name( string& name );  
    string& name();  
  
    // ...  
private:  
    int m_salary;  
    string m_name;  
};
```

Implementation:

```
Employee::Employee(){};  
Employee::Employee( string & name ) {  
    m_name = name;  
}  
  
void  
Employee::set_salary( int salary ) {  
    m_salary = salary;  
}  
int  
Employee::salary() {  
    return m_salary;  
}  
void  
Employee::set_name( string & name ) {  
    m_name = name;  
}  
string&  
Employee::name() {  
    return m_name;  
}
```

Locating an Employee

- Save all employees in a vector.
- When we need to find a specific employee:
 - go over all employees until you find one for which the name matches the requested name
- **Bad solution - not efficient!**

STL - Map

- **Solution:** Map – Associative Array
- We provide a key/value pair. The key serves as an index into the map, the value serves as the data to be stored
- Insertion/find operation:
 - $O(\log n)$

Using Map

- Have a map, where the key will be the employee name and the value the employee object.

name —————> employee

string —————> Employee

```
map<string, Employee *> employees;
```

Populating a Map

```
void main()  
{  
    map<string, Employee *> employees;  
    string name("Pascal");  
  
    Employee * employee;  
    employee = new Employee(name);  
  
    //insertion  
    employees[name] = employee;  
}
```

Locating an Employee

```
map<string, Employee *> employees;
```

- Looking for an employee named **Pascal** :

```
//use  
Employee *pascal = employees["Pascal"];  
//or  
map<string, Employee *>::iterator iter =  
    employees.find("Pascal");
```

- The returned value is an iterator to map. If **"Pascal"** exists on map, it points to this value, otherwise, it returns the **end()** iterator of map

Iterating Across a Map

- Printing all map contents.

```
map<string, Employee *>::iterator it;  
for ( it = employees.begin();  
      it != employees.end(); ++it )  
{  
    cout << ???  
}
```

Map Iterators

```
map<key, value>::iterator iter;
```

- What type of element iter does addresses?
 - The key ?
 - The value ?
- It addresses a key/value pair

STL - Pair

- Stores a pair of objects, first of type T1, and second of type T2

```
template<class T1, class T2>
struct pair
{
    T1 first;
    T2 second;
};
```

Our Pair

- In our example it addresses a

```
pair<string, Employee *>
```

Element

- Accessing the **name** (key)

```
it->first
```

- Accessing the **Employee*** (value)

```
it->second
```

Printing the Salary

```
for ( it = employees.begin() ;  
      it != employees.end() ;  
      ++it )  
{  
    cout << it->first << " "  
          << (it->second)->salary() ;  
}
```

Map Sorting Scheme

- Map holds its content sorted by key
- We would like to sort the map using another sorting scheme (by salary)

Problem: Map Sorting

- **Problem:**

- Since map already holds the elements sorted, we can't sort them

- **Solution:**

- Copy the elements to a container where we can control the sorting scheme

STL - Copy

```
copy(Iterator first, Iterator last,  
     Iterator where );
```

- Copy from 'first' to 'last' into 'where'
- Example:

```
int ia[] = { 0, 1, 1, 2, 3, 5, 5, 8 };  
vector<int> ivec1(ia, ia + 8 ), ivec2;  
  
// ...  
copy(ivec1.begin(), ivec1.end(),  
     ivec2.begin() );
```

Problem: No allocated space

- **ivec2** has been allocated no space
- The **copy** algorithm uses assignment to copy each element value
- **copy** will fail, because there is no space available

Solution: use `back_inserter()`

- Causes the container's `push_back` operation to be invoked.
- The argument to `back_inserter` is the container itself.

```
//ok.copy now inserts using ivec2.push_back()  
copy(ivec1.begin(), ivec1.end(),  
      back_inserter(ivec2) );
```


Inserter iterators

- Puts an algorithm into an “insert mode” rather than “over write mode”.



- `*iter =` causes an insertion at that position, (instead of overwriting).

Back to map sorting problem

- Step 1: copy content into other container

```
map<string, Employee *> employees;
```

```
vector< pair<string, Employee *> > evec;
```

```
copy( employees.begin(), employees.end(),  
      back_inserter( evec ) );
```

Sort

- Formal definition :

```
void sort(Iterator first, Iterator last) ;
```

- Example:

```
vector<int> ivec;
```

```
// Fill ivec with integers ...
```

```
sort(ivec.begin(), ivec.end())
```

Inside Sort

- Sort uses operator $<$ to sort two elements.
- What happens when sorting is meaningful, but no operator $<$ is defined ?

The meaning of operator <

- What does it mean to write :

```
pair<string, Employee *> p1, p2;  
if ( p1 < p2 ) {  
    ...  
}
```

Inside Sort

- No operator $<$ is defined between two pairs!
- How can we sort a vector of pairs ?

Sorting Function

- Define a function that knows how to sort these elements, and make the sort algorithm use it!

Ordering function

```
bool  
lessThen(pair<string, Employee *> &l,  
         pair<string, Employee *> &r )  
{  
    return (l.second)->salary() <  
           (r.second)->salary()  
}
```


Using the ordering function

```
vector< pair<string, Employee *> > evec;
```

```
// Use lessThen to sort the vector.
```

```
sort(evec.begin(), evec.end(), lessThen);
```

pointer to function



Putting it all Together

```
bool lessThen( pair<...> &p1, pair<...> &p2 ) { ... }

int main() {
    map<string, Employee *> employees;
    /* Populate the map. */

    vector< pair<string, Employee *> > employeeVec;
    copy( employees.begin(), employees.end(),
          back_inserter( employeeVec ) );

    sort( employeeVec.begin(), employeeVec.end(),
          lessThen );

    vector< pair<string, Employee *> >::iterator it;
    for ( it = ...; it != employeeVec.end(); ++it ) {
        cout << (it->second)->name() << " "
              << (it->second)->salary() << endl;
    }
    return 0;
}
```