Object Oriented Programming C++ STL (Standard Template Library) Containers

PHYS2G03

© James Wadsley,
McMaster University

Modular Program Design

- Goal: Independent code =

  re-usable, maintainable code
- If code is to be independent, it is critical to understand interfaces: how data is exchanged
- Software Engineering is 80% agreeing on interfaces, e.g. the arguments to functions and what functions do

Key difference

- Traditional (or procedural) programming emphasizes functions to perform specific tasks

  Data and functions are independent

- OOP emphasizes designing and creating objects (classes) as the basis of a program:

  Data and Procedures are linked – combined in a single “object” or abstract data type

Abstract Data Types

- Data should stored and accessed in easy to understand and use ways
- *Object oriented programming: That doesn’t have to mean that you know how it is stored, just how to access it*
- *Can this help create modular, re-usable code…?*

New Data Types

- Lecture: PHYS2G03_NewTypes
- *Recap: C/C++ structures and classes*
- C/C++: struct

  Compound data types

  e.g.

```
struct vector3d { float x,y,z; };
vector3d a;
a.x = 1;  a.y = 2;  a.z = 0.5;  // ok
```

New Data Types

- Lecture: PHYS2G03_NewTypes
- *Recap: C/C++ structures and classes*
- C++: class

  By default data is private (hidden)

```
class vector3d { float x,y,z; };
vector3d a;

a.x = 1;  a.y = 2;  a.z = 0.5; // error
```

New Data Types: objects

- Classes can contain data, functions (methods) and operators to work on their data
- `e.g. vector3d a,b,c;`

```
 float y,r;

 y = a.x;  // raw data
 r = a.magnitude();  // function: size of vector
 c = a+b;  // operator: + add two vectors together
```

vector3d type (not a class)

From PHYS2G03_NewTypes

/home/2G03/newtypes/

Separate data: vector.h

struct vector { float x,y,z; };

From functions: vector.cpp

The new vector data type simplifies the functions, 1 argument per vector instead of 3 floats. But it is still procedural.

vector in → function → Data (vector) out

vector.cpp

```
#include <cmath>
#include "vector.h"

// Dot product of two vectors
float dot( vector a, vector b) {
  return (a.x*b.x + a.y*b.y +
           a.z*b.z);
}
// Magnitude of a vector
float magnitude( vector a ) {
  return sqrt(dot( a, a ));
}
// Cross product of two vectors
vector cross( vector a, vector b) {
  vector c;
  c.x = a.y*b.z - a.z*b.y;
  c.y = a.z*b.x - a.x*b.z;
  c.z = a.x*b.y - a.y*b.x;
  return c;
}
```

vector3d class

vector3d.h

```
#include <cmath>
class vector3d {
public:
  float x,y,z;

  // Constructors
  vector3d() {
    x=0; y=0; z=0;
  }
  //initializing object with values.
  vector3d(float x1, float y1, float
z1) {
    x=x1; y=y1; z=z1;
  }
  // Magnitude of the vector
  float magnitude() {
    return sqrt(x*x+y*y+z*z);
  }
```

```
  // add two vectors
  vector3d operator+(const vector3d &vec)
{
    vector3d ret;
    ret.x = x+vec.x;
    ret.y = y+vec.y;
    ret.z = z+vec.z;
    return ret;
  }
  // Dot product of two vectors
  float dot( vector3d vec) {
    return (x*vec.x + y*vec.y + z*vec.z);
  }
  // Cross product of two vectors
  vector3d cross( vector3d vec ) {
    vector3d ret;
    ret.x = y*vec.z - z*vec.y;
    ret.y = z*vec.x - x*vec.z;
    ret.z = x*vec.y - y*vec.x;
    return ret;
  }
};
```

vector3d class

testvector3d.cpp

Using the vector3d class

Now the functions are operating on the vector as an object

v1.magnitude() ; // vectors magnitude method (function)

v3=v1+v2; // add vectors together

Code in /home/2G03/oo_stl

make testvector3d

```
#include "vector3d.h"
int main() {
  vector3d v1,v2,v3;
  std::cout << "The dot product is " <<
v1.dot(v2) << "\n";

  std::cout << "The angle is " <<
180/M_PI*acos(v1.dot(v2)/v1.magnitude()/v
2.magnitude())<< " degrees\n";

  v3 = v1.cross(v2);
  std::cout << "The cross product is " <<
v3.x << " " << v3.y << " " << v3.z <<
"\n";

  v3 = v1+v2;
  std::cout << "The sum is " << v3.x << "
" << v3.y << " " << v3.z << "\n";
};
```

vector3d class

testvector3d.cpp

Public vs. Private

Now the outside user may not need to see the raw data to do the task.

In software engineering, the internal data is often hidden (private by default) but in scientific computing, we usually want direct access to the vector components:

v1.x v1.y v1.z

To allow that we had to declare the data to be **public:**

In the vector object they are **x y z**

```
include "vector3d.h‟

int main {
 vector3d v1,v2,v3;

  std::cout << "Enter 3 components of 
vector 1\n";
  std::cin >> v1.x >> v1.y >> v1.z;
…
```

vector3d.h

```
class vector3d {
public:
  float x,y,z;
…
```

vector3d class

vector3d.h

```
class vector3d {
  float x,y,z;
  // Constructors
  vector3d() {
    x=0; y=0; z=0;
  }
  //initializing object with values.
  vector3d(float x1, float y1, float z1) {
    x=x1; y=y1; z=z1;
  }
…
```

Data Hiding

In software engineering, the internal data is often hidden but in scientific computing, we usually want direct access to the vector components:

Instead, the components are generated when the vector object is **constructed**

The idea here is that the data could be stored in some fancy way that the programmer does not want us to mess with.

Other code

```
// make a vector with components 1,2,0.5
  vector3d v(1,2,0.5);
```

Modular Program Design

- Goal: Independent code =

  re-usable, maintainable code
- If code is to be independent, it is critical to understand interfaces: how data is exchanged
- Objected oriented programing (OOP) is supposed to help – cleaner code with obvious ways to use the data (and bad data access explicitly disallowed!)

Key difference

- Traditional (or procedural) programming emphasizes functions to perform specific tasks

  Data and functions are independent

- Object Oriented Programming (OOP) emphasizes designing and creating objects (classes) as the basis of a program:

  Data and Procedures are linked

Objects

- Objects contain data and procedures that act upon data
- The idea of an internal state works naturally with objects
- Objects can take data in and put data out
- Conceptually a task is requested of an object via a message
- The message may include data or even other objects

Visualizing OOP

Procedural Approach:

data → Procedure → data

Array of numbers → Sort Function → Sorted Array

Object-Oriented Approach:

message → Object → data

“Sort yourself” → Set of Numbers Object

→ Internally Numbers now sorted

Procedural

list_proc.cpp

Programmer chooses a structure to store the data, such as an array of floats.   Data is stored very simply (efficiently) in memory.

Programmer uses a function: Data goes into function and comes out changed.

The data can be accessed directly for printing to the screen using a[i] for each element.

```
int main {
   float a[3]={3,1,2};

     // HW function
     quicksort(a,3);
     // a now sorted

    for (int i=0;i<3;i++)
        std::cout << a[i] << " ";
    std::cout << "\n";
}
```

Code in /home/2G03/oo_stl
cp –r /home/2G03/oo_stl ~/; cd oo_stl

Object Oriented

made up C++

Programmer chooses an internal representation for the raw data.

Programmer chooses how outside users access data.

The programmer has a sort function specially for the object so it can sort itself.

This is partly because outside users don't know how its stored internally.

```
mylist a({3,1,2});

// object has its own sort
a.sortyourself();
// a now sorted
```

Objects (Old C++ STL list)

list_oo_old.cpp

Programmer chooses an object to store data. Here I use a old-style **C++ standard list**. Internal storage is unknown.

Programmer invokes a sort method function on the object a.

A **method is a function** that belongs just to that object.

Access the list is a bit more complicated. We don't know how they are stored but we can count from beginning to end.

```
int main {
  std::list<float> a;
  a.push_back (3);
  a.push_back (1);
  a.push_back (2);



  a.sort();
  // a now sorted


  for (std::list<float>::iterator
    i = a.begin(); i != a.end() ; i++)
        std::cout << *i << " ";


   std::cout << "\n";
}
```

Objects (C++ 11 STL list)

list_oo.cpp

Programmer chooses an object to store data. Here I use a **C++ standard list**. Internal storage is unknown.

C++ 11 lets you build the list from a list of numbers directly.

Programmer invokes a method on the object a.

A method is a function that belongs just to that object.

Access to the list was streamlined with C++ 11 (2011).

auto : was added as a shorthand to iterate through everything in the list.

```
int main {

     std::list<float> a({3,1,2});

     a.sort();
     // a now sorted

      for (auto &i : a)
        std::cout << i << " ";

      std::cout << "\n";
}
```

For older compilers may need to force C++11 with c++ -std=c++11

Objects: Python List

list.py

Python is an object oriented language. It includes a list as a basic type. Any variable initialized with [] is a list object.

Similar to C++ lists, they have internal data and functions you can call on them – like sort.

```
a=[3,1,2]

a.sort()
# Now sorted

print(a)
```

Objects: Python List vs C++11 List

list_oo.cpp

```
std::list<float> a({3,1,2});

a.sort();
// a now sorted
```

list.py

```
a=[3,1,2]

a.sort()
# Now sorted
```

In practice C++ and Python use fairly similar code under the hood for standard objects like lists.
In the C++ case we explicitly say – store it as <float> which will guarantee fairly efficient compiled code
In Python, the run-time decides how it will do it which may not be very efficient. This is why the Python numpy package exists – to force simple types to make Python more efficient.

C++ Standard Template Library

C++ is a fairly rapidly evolving language and the authors are aware of Python and other competition.

The template library makes standard data structure (containers) using standard types

Template refers to the fact that a type must be given:

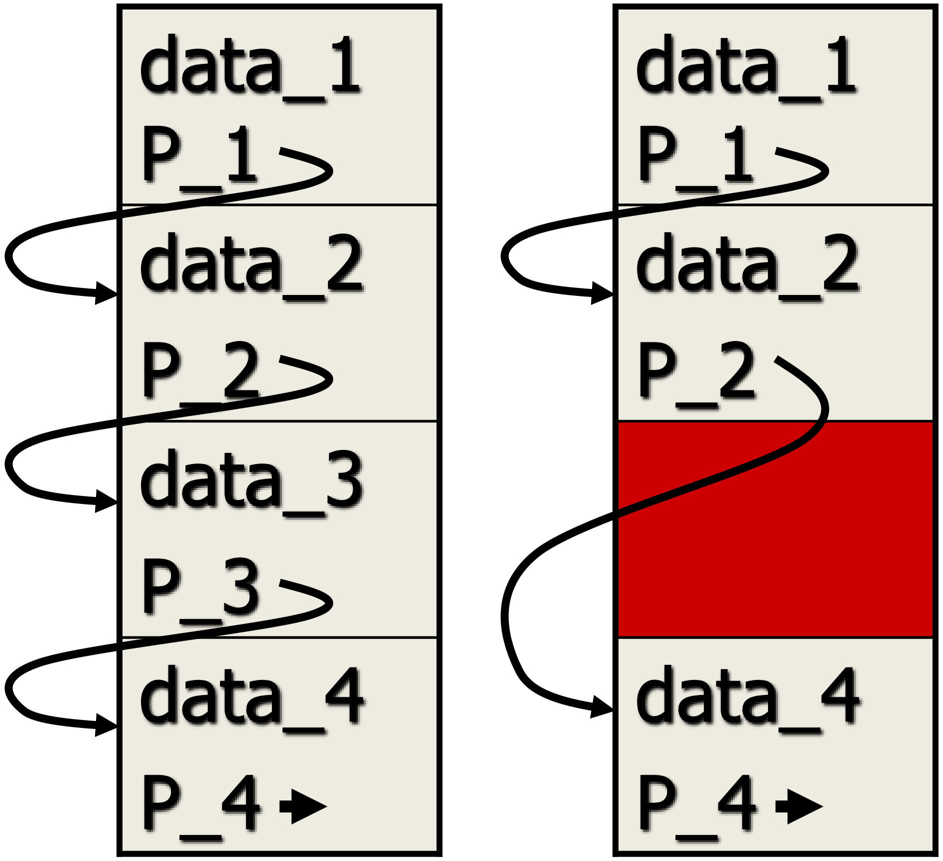
e.g. `std::list<float>  a;`

C++ Standard Containers

std::array	Fixed size array
	[] size begin end
	underneath: data in array, but knowns the size
std::list	Variable length list, optimized for insert
	insert push_back size begin end
	underneath: data in linked-list
std::vector	Dynamic length, optimized for access
	[] size begin end *insert*
	underneath: re-sizeable array
	(insert, re-size is slow)

Data Structure Example: Linked List

- Ordered list of data with pointers to the next entry
- Insert and Remove new data easily



Imagine doing this with an array

Associative containers

C++ std::set

Set of sorted unique keys

Can find, add and remove in logarithmic time O(log N) where N is the number of keys

C++ std::map

set of key, data pairs, e.g.

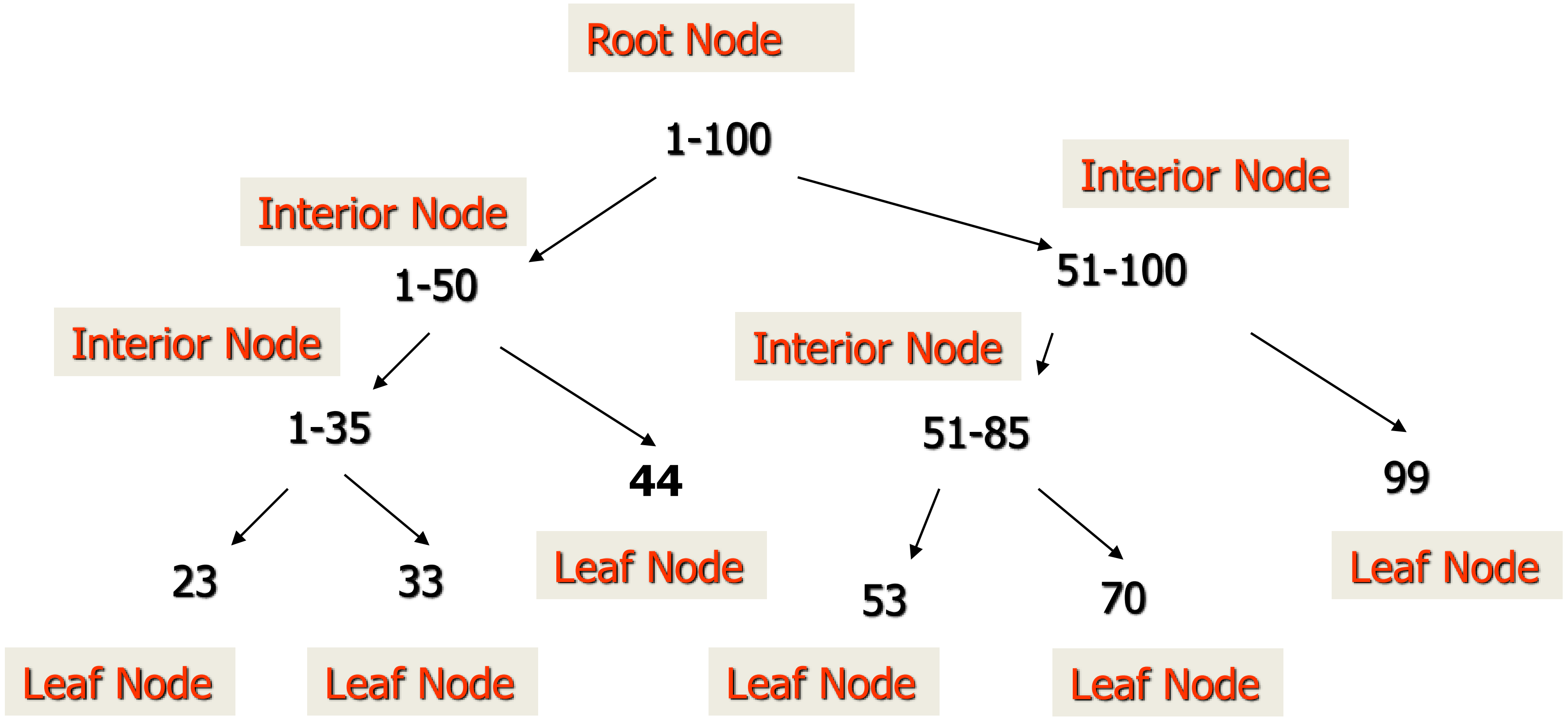
```
std::map<char,int> mp;
mp['a']=10;
mp['b']=30;
```

Very similar to python dictionary, `a = { ‘a’:10, ‘b’:30 }`

Underneath: Typically implemented as red-black (balanced) trees

Closely related to sorting, O(log N) timing

Balanced Binary Tree: Data on Leaf Nodes



C++ Standard Containers Adapters

std::priority_queue Prioritized data, variable size

Built out of another container, e.g. std::vector

[] size begin end *insert* push pop

pop gets biggest element

push adds new element

underneath: data in std::vector

Similar to a tree data structure

Useful for managing events, e.g. in a simulation of colliding particles to do collision in order

Binary Tree: Priority Queue

$$\begin{tikzcd}
99 \arrow[dr] & & & \\
& 70 \arrow[dl] \arrow[dr] & & \\
53 & & 44 \arrow[dl] \arrow[dr] & \\
& 23 & & 33
\end{tikzcd}$$

References

- Containers
  - en.wikipedia.org/wiki/Collection_(abstract_data_type)
- Overview of standard C++ containers:
  - embeddedartistry.com/blog/2017/08/02/an-overview-of-c-stl-containers/
- C++ reference
  - https://en.cppreference.com/w/cpp/container
- Trees
  - en.wikipedia.org/wiki/Tree_(data_structure)
  - en.wikipedia.org/wiki/Binary_search_tree