Week 11 : Graphs, Approximation and Sequential Containers

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 11

Outline

Lambda expressions

- Defining a function just to pass it as an argument to our `Function` constructor can get tedious
- C++ has a notation for lambda expressions which define something that acts like a function
- The lambda expression for the sloping_cos function looks like one of the following two lines of code:

```
1 [](  double x ){ return cos(x)+slope(x); }
2 [](double x) -> double { return cos(x)+slope(); }
```

- The second one explicitly states the return type, which is often not required
- The [] is called the *lambda* introducer, which is followed by the arguments to the lambda
- We could therefore define our `Function` as:

```
1 Function s4{ [](  double x ){ return
      cos(x)+slope(x); },
2   r_min, r_max, orig, 400, 30, 30 };
```

`Axis` interface

- The interface for `Axis` is:

```
struct Axis : Shape {
  enum Orientation { x, y, z };
  Axis(Orientation d, Point xy, int length,
    int number_of_notches=0, string label = "");
  void draw_lines() const;
  void move(int dx, int dy);
  void set_color(Color c);
  Text label;
  Lines notches;
};
```

- Note that it has no private data
- Just a collection of semi-independent objects

`Axis` constructor

```
1  Axis::Axis(Orientation d, Point xy, int length, int n,
       string lab) : label(Point(0,0),lab) {
2    if (length<0) error("bad axis length");
3    switch (d){
4      case Axis::x: {
5        Shape::add(xy); // axis line
6        Shape::add(Point(xy.x+length,xy.y)); // axis line
7        if (1<n) {
8          int dist = length/n;
9          int x = xy.x+dist;
10         for (int i = 0; i<n; ++i) {
11           notches.add(Point(x,xy.y),
12                       Point(x,xy.y-5));
13           x += dist;
14         }
15       }
16       // label under the line
17       label.move(length/3,xy.y+20);
18       break;
19     }
20     // ...
```

`Axis` constructor cont...

```
    // ... continued
    case Axis::y: {
      Shape::add(xy); // a y-axis goes up
      Shape::add(Point(xy.x,xy.y-length));
      if (1<n) {
        int dist = length/n;
        int y = xy.y-dist;
        for (int i = 0; i<n; ++i) {
          notches.add(Point(xy.x,y),Point(xy.x+5,y));
          y -= dist;
        }
      }
      // label at top
      label.move(xy.x-10,xy.y-length-10);
      break;
    }
    case Axis::z:
      error("z axis not implemented");
  }
}
```

Notes on `Axis`

- Uses the `Shape` to store the line part of the Axis, by using `Shape::add()`
- `notches` are kept as a separate `Lines` object
- enumeration `Orientation` is used for the axis direction
- `Axis` has three parts to draw, so `draw_lines()` looks like:

```
1 void Axis::draw_lines() const {
2   // draw the line
3   Shape::draw_lines();
4   // the notches may have a different color from
        the line
5   notches.draw();
6   // the label may have a different color from the
        line
7   label.draw();
8 }
```

- Note the use of `draw()` for notches and label which first sets color, then calls `draw_lines()`, finally resetting the color

Notes on `Axis`

- Could set axis line, notches, and label to have different colors, or use method provided to set all three to same color:

```
void Axis::set_color(Color c) {
  Shape::set_color(c);
  notches.set_color(c);
  label.set_color(c);
}
```

- Similarly move all three parts of axis:

```
void Axis::move(int dx, int dy) {
  Shape::move(dx,dy);
  notches.move(dx,dy);
  label.move(dx,dy);
}
```

Introduction to approximations

- We will use graphics to illustrate approximating a function by Taylor series
- For example Taylor series of exponential function is:

$$\mathsf{e}^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + ... \qquad (1)$$

- where ! is usual factorial (ie. $4! = 4 \times 3 \times 2 \times 1$)
- The more terms we add in this sequence the more precise our value of e^x
- We can compare graphs of functions:

```
exp0(x) = 0; // no terms
exp1(x) = 1; // one term
exp2(x) = 1+x; // two terms
exp3(x) = 1+x+pow(x,2)/factorial(2);
exp4(x) = exp3(x) + pow(x,3)/factorial(3);
...
```

Functions to solve our approximation problem

- Need a factorial function (not in std library):

```
double fac( int n ){
  double r = 1.0;
  while (n>1){
    r*=n;
    --n;
  }
  return r;
}
```

- We also need nth term in series:

```
double termn( const double x, const int n ){
  return pow( x, n ) / fac( n );
}
```

Functions to solve our approximation problem

- Now exponential up to term n is:

```
double expo( const double x, const int n ){
  double sum = 0;
  for ( int i=0; i<n; ++i ) sum += termn( x, i );
  return sum;
}
```

- We can define the `Function` that uses standard library exponential to compare to:

```
Function realexp{ exp, r_min, r_max, orig, 200,
    x_scale, y_scale };
realexp.set_color( Color::blue );
```

- But how to use our function `expo` that takes two arguments?

Using lambda to adapt our function

- We can use a lambda expression to wrap our function, so our code to successively display better and better approximation is:

```
for ( int n = 0 ;  n < 50 ; ++n ){
  ostringstream ss;
  ss << "Exp approximation to n=" << n;
  win.set_label( ss.str() );
  Function e{ [n](double x){ return expo(x,n); },
    r_min, r_max, orig, 200, x_scale, y_scale };
  win.attach(e);
  win.wait_for_button();
  win.detach(e);
}
```

- lambda inducer `[n]` says lambda expression may access local variable n
- For this to work however, need to use std::function from `<functional>` header and make an overloaded version of `Function`... not so simple

Result of `approxExpo.cpp` for $n = 0$

- Note that there was a glitch in the drawing
- Was an integer over-run somewhere due to large values of exponential

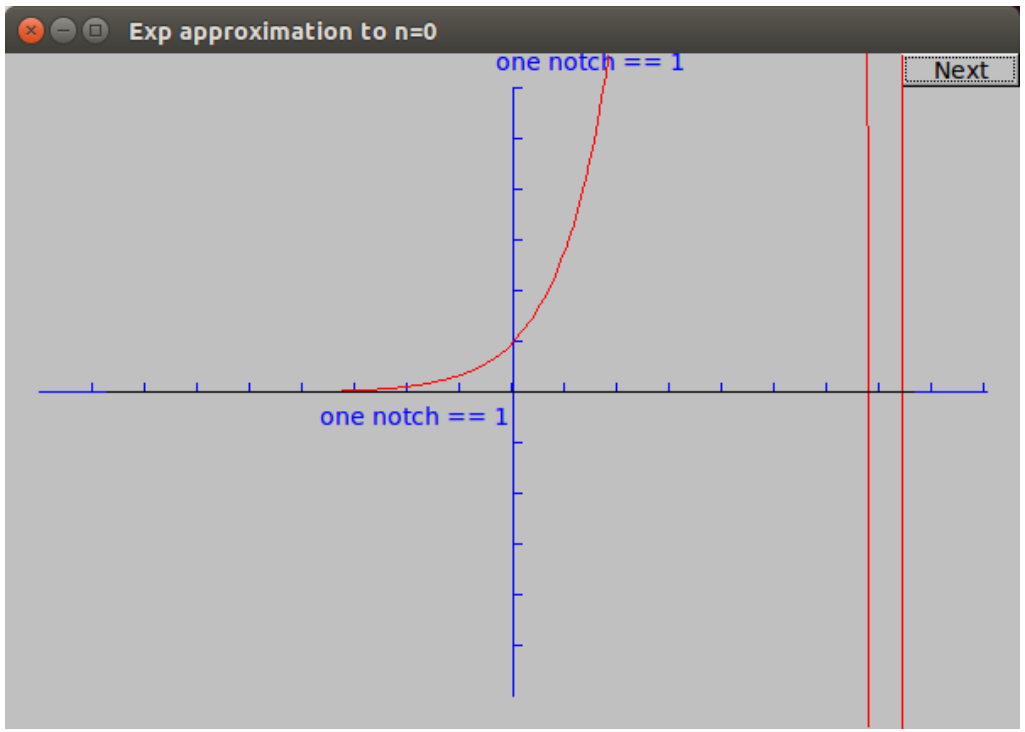

Figure: For $n = 0$ have straight line along $y == 0$

Result of `approxExpo.cpp` for $n = 1$

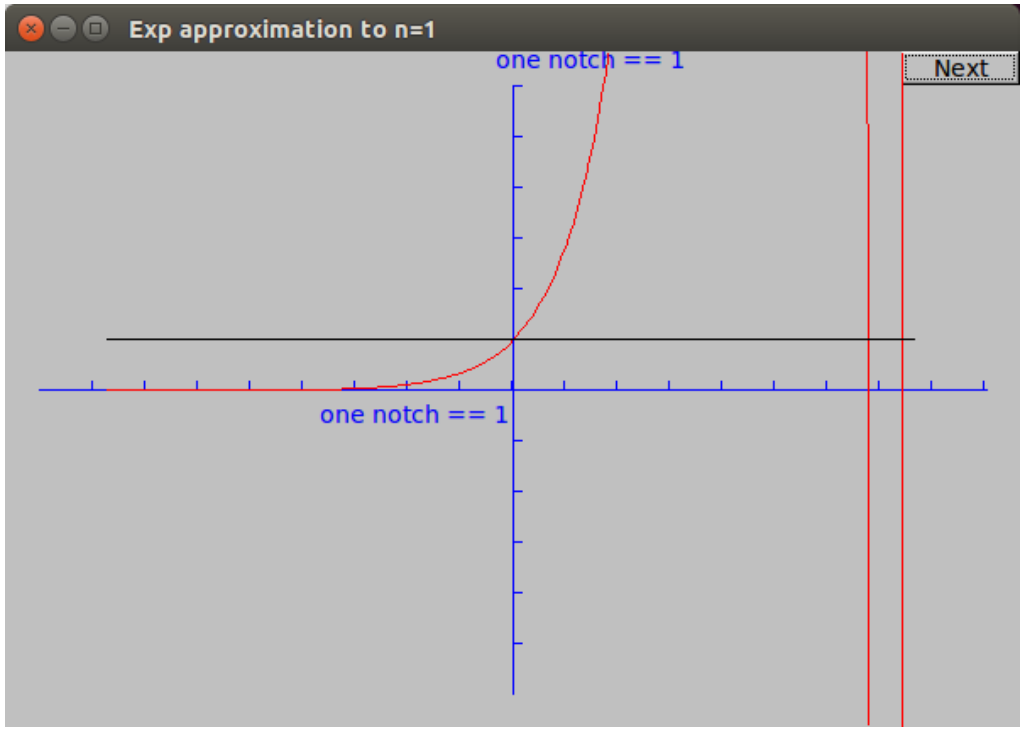

Figure: For $n = 1$ have straight line $y = 1$

Result of `approxExpo.cpp` for $n = 2$

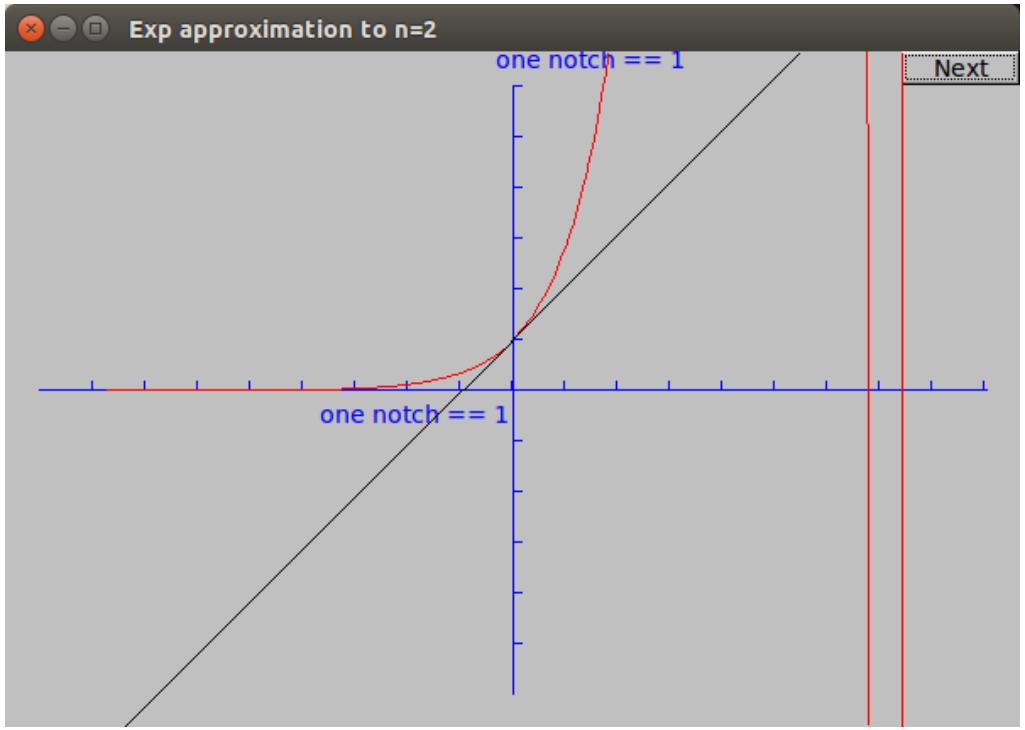

Figure: For $n = 2$ have slope line along $y = 1 + x$

Result of `approxExpo.cpp` for $n = 3$

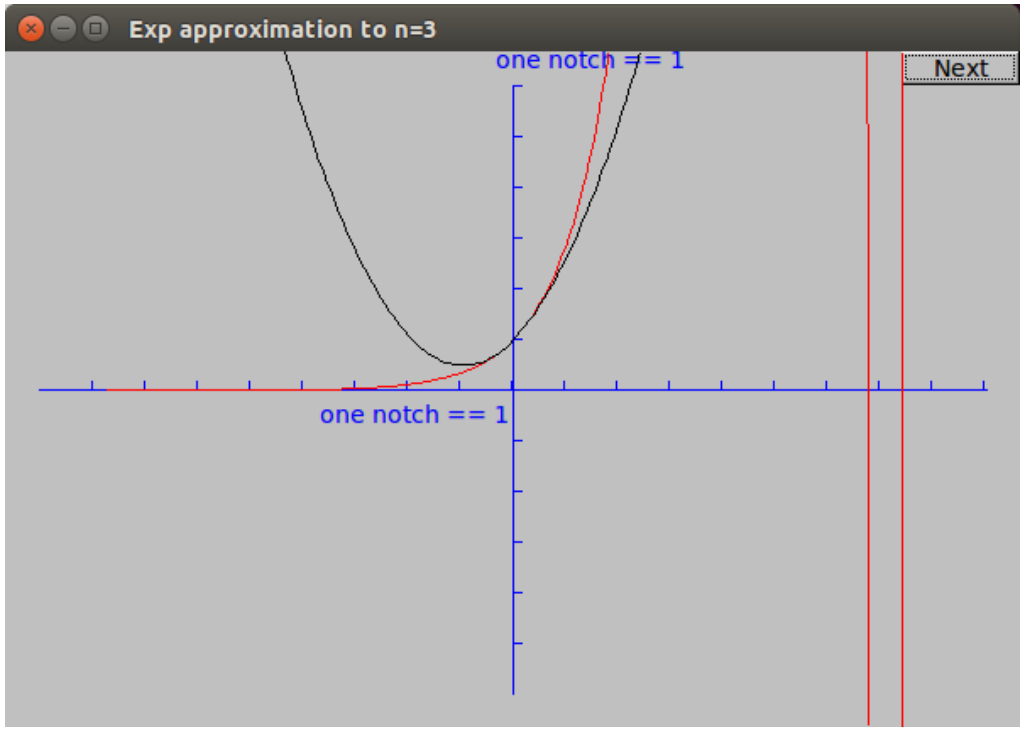

Figure: For $n = 3$ have parabola along $y = 1 + x + x^2$

Result of `approxExpo.cpp` for $n = 10$

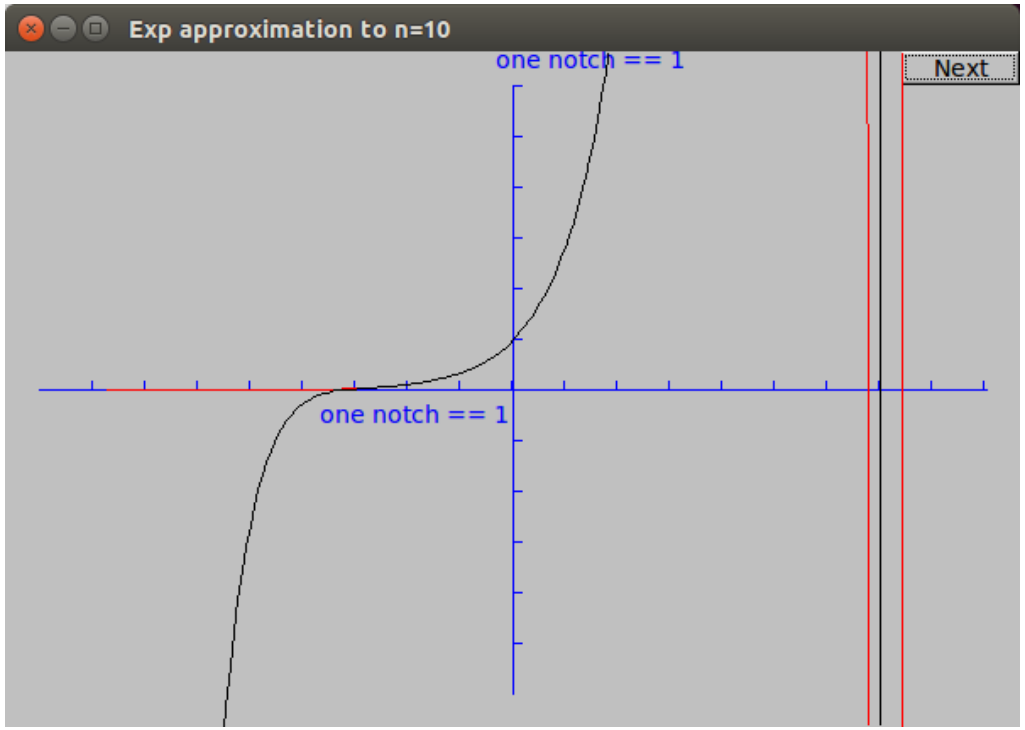

Figure: For $n = 10$ approximation is pretty good above $x = -3$

Result of `approxExpo.cpp` for $n = 18$

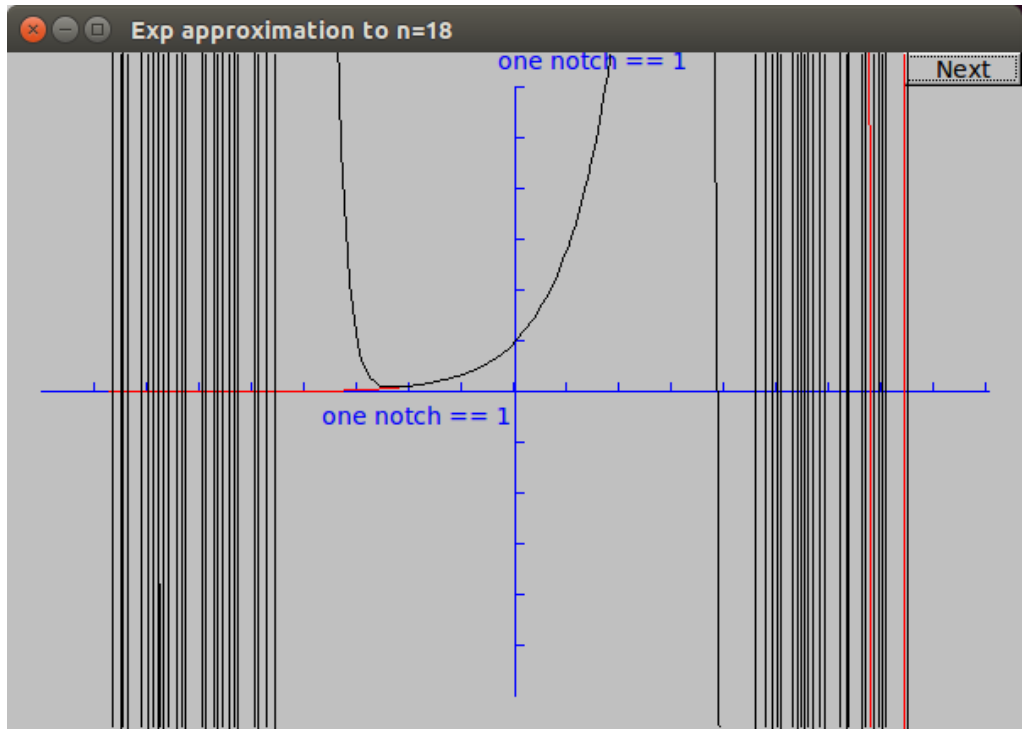

Figure: For $n = 18$ seems worse than previous iterations

- Have a look at `approxExpo.cpp` and try running it.

Introduction to graphing data

- Graphing data is a highly valued skill
- Doing it well is as much an art as science
- Our simple plots will not reach a publication quality
- We will work towards a plot of the age of Japanese people and projection into future

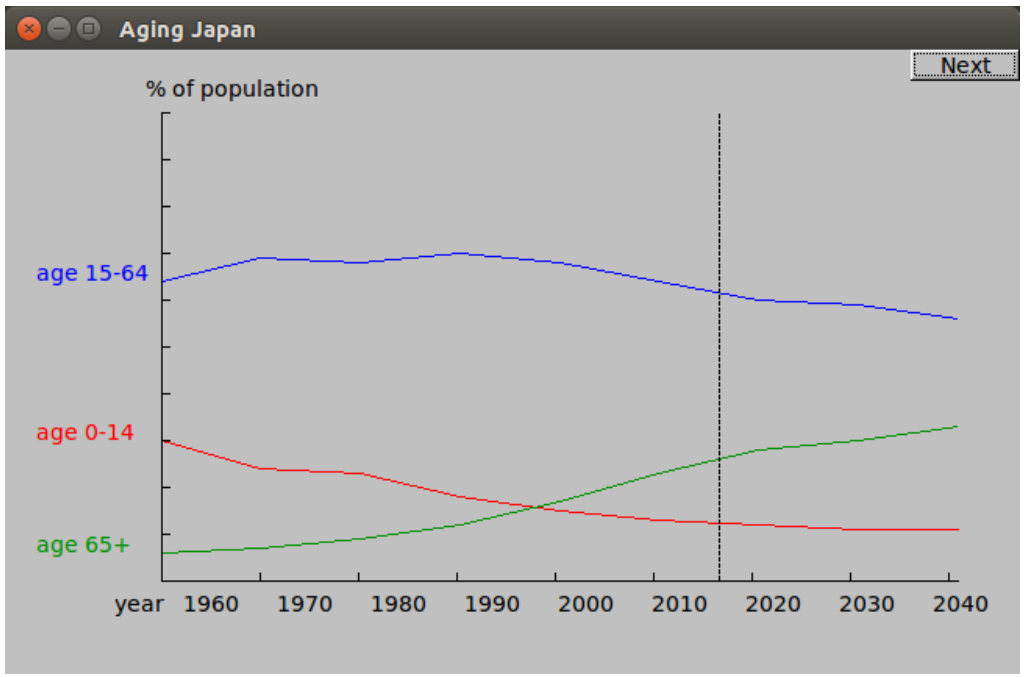

Figure: Graph that we will make

Parts needed for graphing data

- Reading a file of data
- Scaling data to fit the window
- Displaying the data
- Labelling the graph

Reading the file

- The data we are interested in plotting is in lines of the form:

```
( 1960 : 30 64 6 )
(1970 : 24 69 7 )
(1980 : 23 68 9 )
```

- Format gives year, followed by percent aged 0-14, aged 15-64, then aged 65+
- To keep code organized, often is useful to define a struct or class to read in data that is in a peculiar format

Reading the file

- Since our data is a type of distribution of ages, lets call the structure `Distribution`:

```
1  struct Distribution {
2    int year, young, middle, old;
3  };
4  // assume format: ( year : young middle old )
5  istream& operator>>(istream& is, Distribution& d){
6    char ch1 = 0; char ch2 = 0; char ch3 = 0;
7    Distribution dd;
8    if (is >> ch1 >> dd.year
9           >> ch2 >> dd.young >> dd.middle
10          >> dd.old >> ch3) {
11     if (ch1!= '(' || ch2!=':' || ch3!=')') {
12       is.clear(ios_base::failbit);
13       return is;
14     }
15   } else
16     return is;
17   d = dd;
18   return is;
19 }
```

Reading the file

- Use of `Distribution` struct splits code up into logical parts
- Wasn't strictly required – could have had one long sequential code read the data in then plot it
- But splitting the code into conceptual blocks helps make the code easier to debug
- Also makes it easier for someone else to read

Reading the file – read loop

```
string file_name = "japanese-age-data.txt";
ifstream ifs(file_name.c_str());
if (!ifs) error("can't open ",file_name);
//...
Distribution d;
while (ifs>>d) {
  if (d.year<base_year || end_year<d.year)
    error("year out of range");
  if (d.young+d.middle+d.old != 100)
    error("percentages don't add up");
  //...
}
```

Reading the file – notes

- Here we hard coded the filename into the program
- That's usually not a good idea for a long lived program
- Sometimes the code is meant to be a quick one off
- For longer lived code add extra code needed to make it more general
- Note that we check that the year is in some range
- Also check that percentages add up to 100%

Layout for the graph

- We will use `Open_polylines` to represent the data – one for each age group
- Need to label datasets – use colour and label somewhere along line
- Label x-axis with years
- Vertical line through 2016 to indicate the current year
- Getting placement of components of graph correct is fairly tricky and requires calculating several sizes and offsets
- Often a good idea to define several constants to help with various locations on the screen
- A quick sketch of the locations can help too!

A picture of the layout

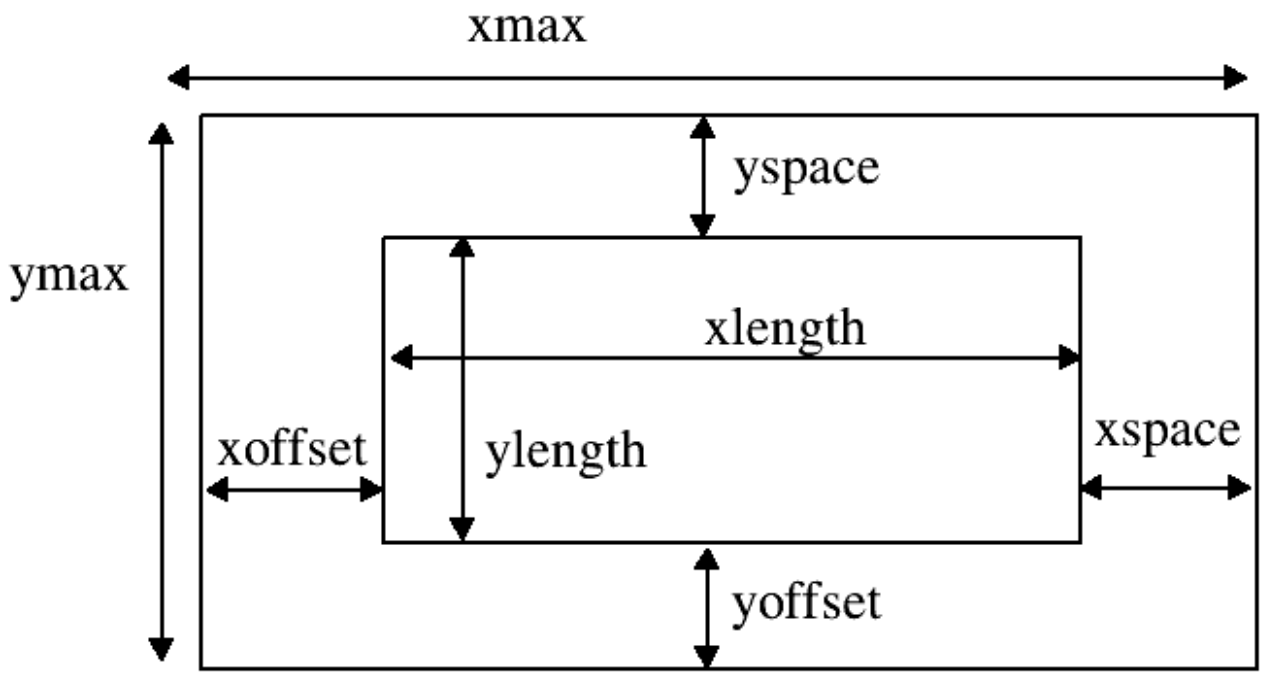

Figure: Sketch of offsets to locations on screen

Definitions of `const` values for layout

```
1  constexpr int xmax = 650; //window size
2  constexpr int ymax = 400;
3  // distance from left hand side of window to y axis
4  constexpr int xoffset = 100;
5  // distance from bottom of window to x axis
6  constexpr int yoffset = 60;
7  constexpr int xspace = 40;   // space beyond axis
8  constexpr int yspace = 40;
9  // length of axes
10 constexpr int xlength = xmax-xoffset-xspace;
11 constexpr int ylength = ymax-yoffset-yspace;
```

Scaling the data

```
constexpr int base_year = 1960;
constexpr int end_year  = 2040;
// scale of x values
constexpr double xscale =
  double(xlength) / (end_year-base_year);
// scale of y values
constexpr double yscale=double(ylength)/100;
```

- Converted `int` values into `double` to avoid rounding errors on scales for axes

`class` to do Scaling

- To scale a year onto screen, we subtract xoffset then scale by xscale:

```
1  // data value to coordinate conversion
2  class Scale {
3    int cbase;     // coordinate base
4    int vbase;     // base of values
5    double scale;
6  public:
7    Scale(int b, int vb, double s)
8      : cbase(b), vbase(vb), scale(s) { }
9    int operator()(int v) const {
10     return cbase + (v-vbase)*scale;
11   }
12 };
```

The scaling factor

- We define the scaling factors as:
  `Scale( int coordBase, int baseValue, doube scale )`

```
1 Scale xs{ xoffset, base_year, xscale };
2 Scale ys{ ymax-yoffset, 0, -yscale};
```

- `yscale` was made negative since the coordinates grow downwards, whereas graph increases upwards
- Then when we want coordinates on screen for `value` object `xs`:

```
int locx = xs( value );
```

- `locx` will then have result of `xoffset + (value-base_year)*xscale`

Building the graph

- Now we can build the graphs axes and line at 2016 as follows:

```
Simple_window win(Point{100,100}, xmax, ymax,
  "Aging Japan");
Axis x(Axis::x, Point{xoffset,ymax-yoffset},
  xlength, (end_year-base_year)/10,
  "year   1960      1970      1980      1990      "
  "2000      2010      2020      2030      2040");
x.label.move(-100,0);
Axis y( Axis::y, Point{xoffset,ymax-yoffset},
    ylength, 10,"% of population");
Line current_year( Point{xs(2016), ys(0)},
    Point{xs(2016), ys(100)});
current_year.set_style(Line_style::dash);
```

- Axes cross at `(1960,0)` at `xoffset, ymax-yoffset` on window

Notes on our axes

- Notches on x-axis are calculated to be every 10 years based on constants that where defined – if the data is updated this can easily be extended
- Notches on y-axis are every 10%
- Axis label string is a bit curious, two adjacent string literals were used:

```
"year    1960        1970        1980        1990        "
"2000        2010        2020        2030        2040"
```

- Adjacent string literals are concatenated by the compiler
- The label string is not very nice, since it had to be adjusted by hand to put the numbers next to the tick marks
- To do better we would have to build set of labels – one for each tick mark then build string for overall label
- Placing line at 2016 simplified by using scaling class

Open_polylines for the data

```
Open_polyline children;
Open_polyline adults;
Open_polyline aged;
Distribution d;
while ( ifs >> d ) {
  if ( d.year<base_year || end_year<d.year )
    error("year out of range");
  if ( d.young + d.middle + d.old != 100 )
    error("percentages don't add up");
  int x = xs( d.year );
  children.add( Point{ x, ys( d.young) } );
  adults.add( Point{x, ys( d.middle ) } );
  aged.add( Point(x, ys( d.old ) } );
}
```

Notes on `Open_polylines` for the data

- Using the `Scale` objects `xs` and `ys` make adding points to our polylines trivial
- Little classes like `Scale` are great for simplifying notation and avoiding having scaling done in multiple places
- Reduces possibility of making an error – since we only need to write the scaling once
- Increases readability of code

Adding labels to the data sets

- Now we add colored text to label the datasets:

```
Text children_label( Point{20,children.point(0).y},
    "age 0-14");
children.set_color(Color::red);
children_label.set_color(Color::red);
Text adults_label( Point{20,adults.point(0).y},
    "age 15-64");
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);
Text aged_label( Point{20,aged.point(0).y},
    "age 65+");
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
```

Attaching objects to the window

- Finally, remember to attach all the `Shape` objects to the window and start the GUI

```
1     win.attach(children);
2     win.attach(adults);
3     win.attach(aged);
4     win.attach(children_label);
5     win.attach(adults_label);
6     win.attach(aged_label);
7     win.attach(x);
8     win.attach(y);
9     win.attach(current_year);
10    win.wait_for_button();
```

Result of our efforts

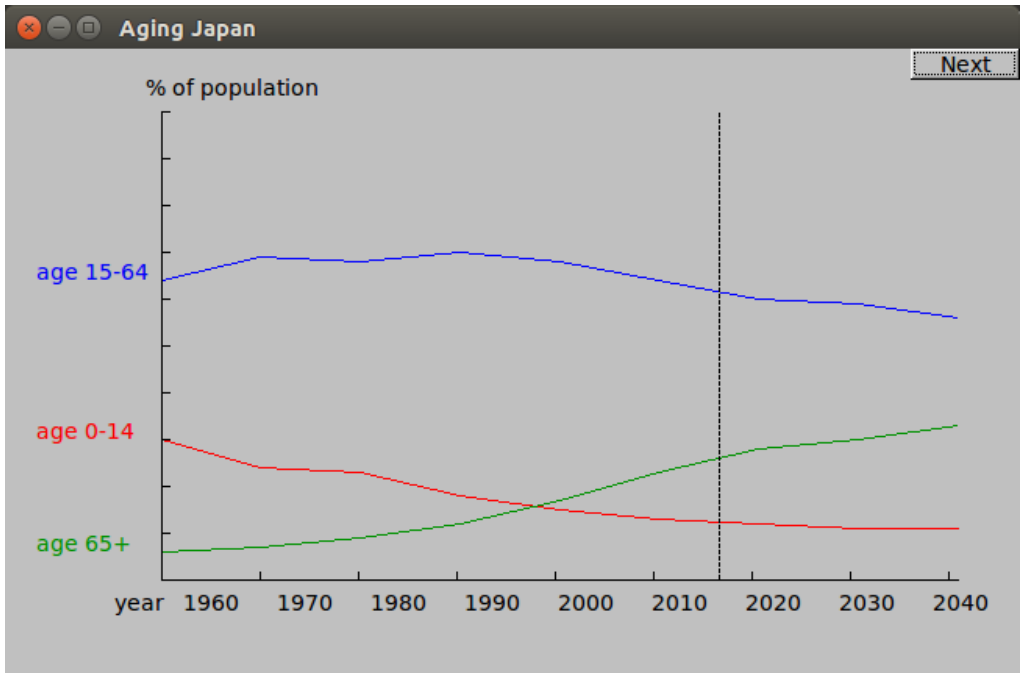

Figure: In case you forgot what we were plotting!

- Have a look at `plotJapan.cpp` and try running it.

Outline

Introduction to sequential containers [1]

Sequential container types	
`vector`	flexible sized array, with fast random access inserting, deleting other than at back may be slow
`deque`	double-ended queue, with fast random access fast insert/delete at front or back
`list`	doubly-linked list, supports only bidirectional sequential access. Fast insert/delete at any point
`forward_list`	singly-linked list, supports sequential access in one direction. Fast insert/delete at any point
`array`	fixed sized array, with fast random access cannot add or remove elements.
`string`	special container similar to `vector` that contains only characters.

[1] Notes on sequential containers are based on Ch.9 of S.Lippman, J. Lajoie, B. Moo, "C++ Primer," 5th edition.

Standard Template Library Container overview

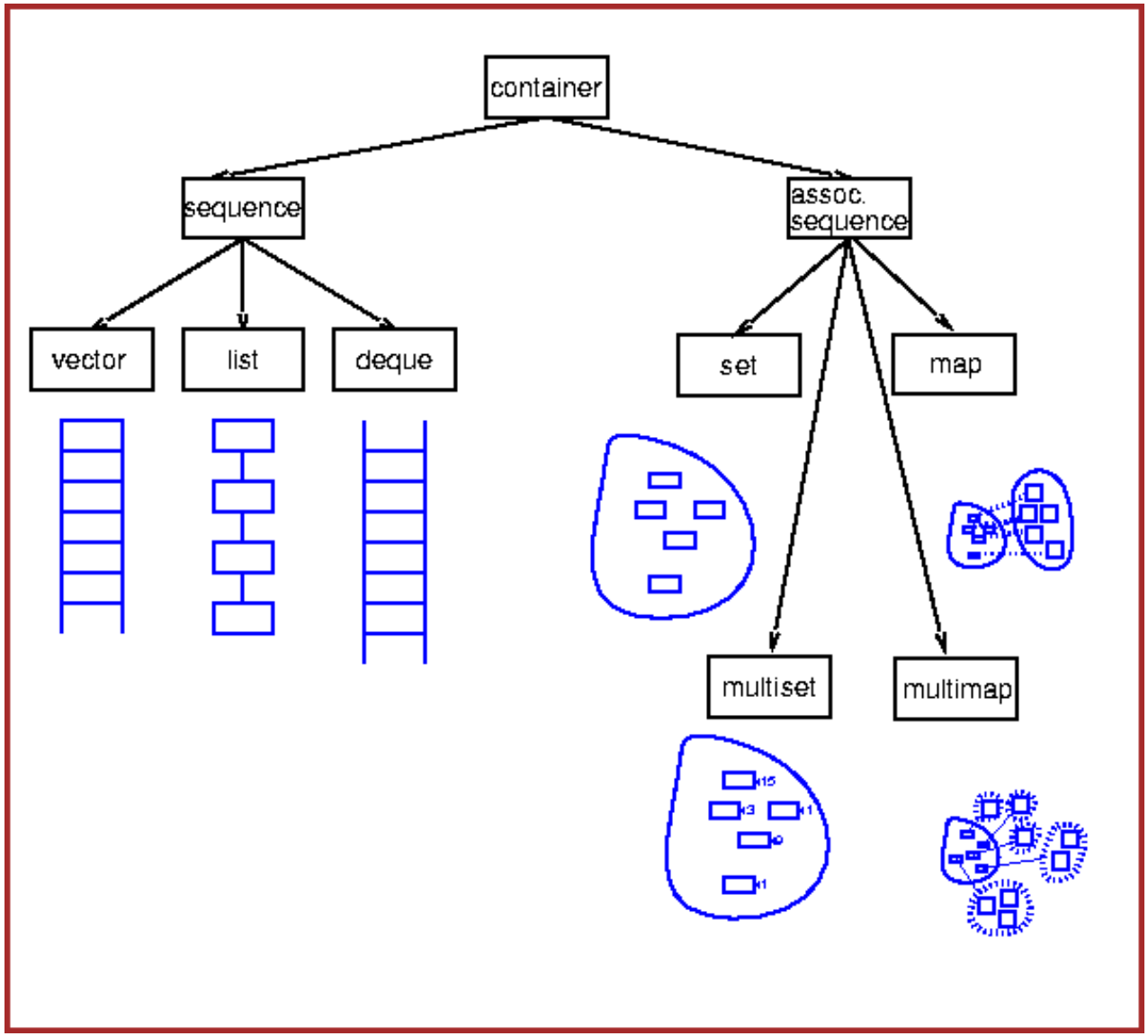

Figure: Image from
cs.brown.edu/~jak/proglang/cpp/stltut/tut.html

`list` and `forward_list`

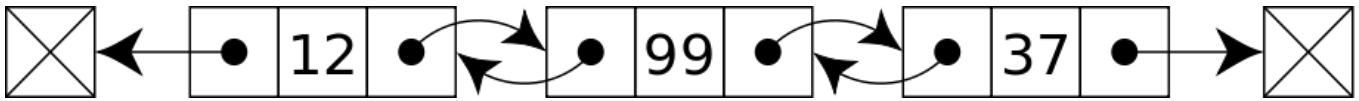

Figure: Doubly-linked list (`list`) Public domain images from `en.wikipedia.org/wiki/Linked_list`.

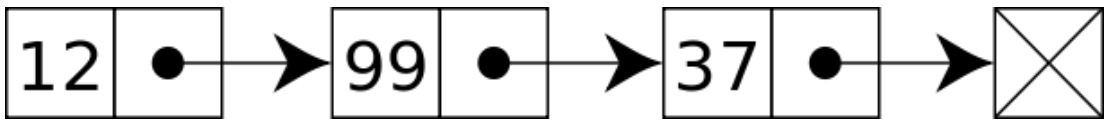

Figure: Singly-linked list (`forward_list`). Public domain images from `en.wikipedia.org/wiki/Linked_list`.

Notes on `vectors` and `lists`

- `string` and `vector` hold elements in contiguous memory:
  - Easy to compute address of an element from index
  - Adding or removing element in middle takes time – need to move all elements after the one added/removed
  - Sometimes may need to move all elements to a new block of memory to maintain continuity of memory
- `list` and `forward_list` make it fast to add or remove an element anywhere in the container
  - We cannot access an element by index, instead have to iterate through the container to get to a particular element

Notes on `deque` and others

- `deque` allows adding and removing elements from its beginning or end
- `deque` also has fast random access
- `deque` will be expensive to remove or add an element in its middle
- `array` is new to C++ 11 – is a safer way to use built-in arrays.
- `array` is a fixed size container, so has no operations to add, remove or resize
- `forward_list` is comparable to the best handwritten, singly linked list – that does not provide a size operation (as that would add overhead)
- Library containers are dramatically faster than in previous releases – and are designed to perform as well as most carefully crafted alternatives
- *Modern C++ programs should use the library containers rather than more primitive array structures*

Deciding which container to use

- Ordinarily use `vector` unless there is a good reason to prefer another container
- If you need random access to elements use a `vector` or `deque`
- If program needs to insert or delete elements in middle of the container, use a `list` or `forward_list`
- If the program needs to insert or delete elements at the front or back, but not middle, use a `deque`
- If the container only inserts elements in middle while reading input – and subsequently does random access:
  - Consider using `vector` and adding to end, followed by use of `sort` function
  - If you must insert in middle, consider using `list` for input, then copy into `vector`

Container libraries overview

- Each container is defined in a header file with the same name a the type, ie. `vector` is in the `vector` header, `list` is in the `list` header, etc.
- For most containers are class templates – as with `vector` need to supply the element type, for example:

```
1 list< Button > lbt; // lbt is a list holding
      Button objects
2 deque< double > ddd; // ddd is a deque holding
      double objects
```

- Most types can be used as the element type, including other container types:

```
1 vector< vector<string> > lines; //vector of vectors
```

- Older compilers may require a space between the angle brackets

Container element initializer

- Container may have requirements of the element type
- For example may take a size argument, or require a default constructor
- Example, suppose class `noDefault` does not have a default constructor:

```
1 vector< noDefault > v1( 10, init); // ok: element
      initializer supplied
2 vector< noDefault > v2( 10 ); // error: must
      supply element initializer
```

- Next several slides outline operations that can be done with container classes

Container type aliases and constructors

type aliases in the container classes	
`iterator`	type of the iterator for this container
`const_iterator`	iterator read but not change elements
`size_type`	unsigned integer type
`difference_type`	signed integer for diff between 2 iters
`value_type`	element type
`reference`	elements lvalue (`value_type &`)
`const_reference`	elements const lvalue
constructors for containers	
`C c;`	default constructor
`C c1( c2 );`	construct c1 as a copy of c2
`C c( beg, end );`	copy elements from range iterators *not valid for array*
`C c{a, b, c, ...};`	List initialize c

Container assignment, swap, and size

assignment and swap	
`c1 = c2;`	copy `c2` into `c1`
`c1 = {a,b,c,...};`	replace elements in `c1` with list
`a.swap( b );`	swap elements in `a` with those in `b`
`swap( a, b );`	equivalent to `a.swap(b);`
size	
`c.size();`	number of elements in `c`
	not valid for forward_list
`c.max_size();`	maximum elements `c` can hold
`c.empty();`	`false` if `c` has any elements

Container add/remove elements

add/remove elements	
None of these operations are valid for array	
Interface to these operations varies by container type	
`c.insert( args )`	copy elements by args into c
`c.emplace( inits )`	use inits to construct an element in c
`c.erase( args )`	remove element(s) specified by args
`c.clear()`	remove all elements from c (void return)
equality and relational operators	
`==, !=`	equality valid for all containers
`<, <=, >, >=`	relationals not all containers

Container iterators

Obtaining iterators	
`c.begin()`	return iterator to first element of `c`
`c.end()`	return iterator to one past last elem of `c`
`c.cbegin()`	return const iterator to first of `c`
`c.cend()`	return const iterator one past end of `c`
Iterators for reversible containers (not forward_list)	
`reverse_iterator`	iterator addressing elements in reverse
`const_reverse_iterator`	reverse iter that can't change elements
`c.rbegin()`	iterator for last element in `c`
`c.rend()`	iterator to one before first element in `c`
`c.crbegin()`	const iter – last element in `c`
`c.crend()`	const iter – one before 1st elem in `c`

Iterators

- An iterator range is given by a pair of iterators:
  - `begin()` is the first element
  - `end()` is the last element
- Range of elements is `[begin,end)`
- It is possible to reach `end` by repeatedly incrementing `begin`
- Note that compiler can't enforce above, so our programs must follow above conventions
- Above means we can safely write loops such as:

```
while ( begin != end ){
  * begin = val; // ok: range isn't empty so begin
      is an element
  ++ begin;      // advance iterator to next
      element
}
```

- Reverse iterator version:

```
while ( rend != rbegin ){
  * rend = val; // ok: range isn't empty so rend
      is an element
  --rend; // decrement iterator to previous element
}
```

Container type members

- If we need element type, can use `value_type`
- If we need reference to that type use `reference` or `const_reference`, and so on...
- To use these types, we name the class to which they are a member, for example:

```
1 // iter is an iterator type defined by
      list<string>:
2 list<string>::iterator iter;
3 // count is difference_type defined by vector<int>:
4 vector<int>::difference_type count;
```

- Note the use of the scope operator :: to say we want that member of the class
- Q: What should be used as index into `vector<int>`?
- Q: What type should be used to read elements in `list` of `strings`?

`begin` and `end` members

- The `begin` and `end` operations give iterators referring to the first and one past the last element of the container
- There are several versions of `begin` and `end` depending on whether you want `const` or reverse versions:

```
1 list<string> a = {"Einstein","Newton","Maxwell"};
2 auto it1 = a.begin(); // list<string>::iterator
3 auto it2 = a.rbegin(); //
      list<string>::reverse_iterator
4 auto it3 = a.cbegin(); //
      list<string>::const_iterator
5 auto it4 = a.crbegin(); //
      list<string>::const_reverse_iterator
```

- When write access is not needed, use `cbegin` and `cend`

Defining and intializing containers

```
1 list<string> physicists = {"Feynman","Pauli","Dirac"};
2 vector< const char* > theories =
      {"QED","QCD","Electroweak"};
3 list<string> nobels( physicists ); //ok types match
4 deque<string> authors( physicists ); //error container
      mis-match
5 vector< string > models( theories ); //error type
      mis-match
6 // below is ok, copies char* elements to string
7 forward_list<string> words( theories.begin(),
      theories.end() );
```

- Initialization of container using another container – types and element types must be identical
- Can copy elements to new compatible type using iterator intitialization
- Note use of list initialization, specifying elements for the container

Sequential container size

```
vector<int> ivec( 10, -1);  // ten int elements each -1
list<string> svec( 10, "hi"); // ten strings each "hi"
forward_list<int> ivec(10);   // 10 elements init to 0
deque<string> svec(10);       // ten empty strings
```

- Constructors taking a size are valid only for sequential containers – not for associative containers
- If element type does not have a default constructor, we must specify an explicit element intializer
- Q: How can you initialize a vector<double> from a list<int>?

Using array

- std library arrays have fixed size – need to specify size at declaration:

```
1  array< int , 43 > a;  // array of 42 ints
2  array< string , 10 > b; // array of 10 strings
3  // below: array type includes element type and size
4  array<int , 10>::size_type i;
5  // below error: array<int> is not a type (size
       required)
6  array<int >::size_type j;
7  array<int , 10> ia1; // 10 default-init ints
8  // list initialize array:
9  array<int , 10> ia2 = {0,1,2,3,4,5,6,7,8,9};
10 // below: first element is 42, rest are zero
11 array<int , 10> ia3 = {42};
12 // one advantage of array over built in array
13 // is that copy by assignment is allowed:
14 int digs[10] = {0,1,2,3,4,5,6,7,8,9};
15 // below error: no copy for built-in array
16 int cpy[10]=digs;
17 array<int,10> digits = {0,1,2,3,4,5,6,7,8,9};
18 array<int,10> copy = digits; // okay
```

Container assignment

- Assignment operators act on entire container
- `assign` method can be used for sequential container, options are:

`seq.assign(b,e);`	replace elements in `seq` with those in iterator range `b` up to `e`
`seq.assign(i1);`	replace elements in `seq` with those in initializer list `i1`
`seq.assign(n,t);`	replace elements in `seq` with `n` elements with value `t`

```
1 c1 = c2; // replace c1 with copy of contents of c2
2 c1 = { a, b, c }; // c1 is assigned three elements
3 list<string> names;
4 vector< const char* > cn;
5 // can convert const char * to string:
6 names.assign( cn.cbegin(), cn.cend() );
7 list<string> lshi(1); // one element empty string
8 lshi.assign(10,"Hi"); // ten elements,each is "Hi"
```

Container swap

```
1 vector<string> vs1(10); // vector of 10 empty strings
2 vector<string> vs2(24); // vector of 24 empty strings
3 swap( vs1, vs2 ); // vs1 now 24 elements, vs2 has 10
```

- After swap all pointers, references and iterators remain bound to the same element they denoted before swap
- Member and non-member version of swap exists – best to use non-member version

Relational operators

- Every container supports equality operators == and !=
- ordered containers also support (>,>=, <, <=)
- can only compare containers of same type, ie. vector<int> with vector<int>, vector<double> with vector<double>, etc.
  - If containers are same size and all elements are equal, then two containers are equal – otherwise unequal
  - If containers have different sizes but every element in the smaller one is equal to the corresponding element of larger one, smaller one is less than other
  - If neither container is an initial sub-sequence of other, then comparison depends on comparing the first unequal elements.

Relational operator examples

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
v1 < v2; // true, v1 and v2 differ at element [2]
v1 < v3; // false, v3 has fewer elements
v1 == v4; // true -- all elements are the same
v1 == v2; // false -- v2 has fewer elements than v1
```

Relational operator examples

- Relational operator to compare two containers only works if the operator is defined for the element type
- For example no relational operators were defined for `Button` objects:

```
1 vector< Button > buttonA, buttonB;
2 // error below: Button has no less-than operator
3 if ( buttonA < buttonB )
```

Adding elements to sequential containers

- Adding to anywhere but end of `vector` or `string`, or anywhere other than end or beginning of `deque` will require elements to be moved
- `push_back` appends element to the back of a container (not for `array` or `forward_list`)
- item "pushed back" is copied – no subsequent connection between the object pushed back, and the one now in the container
- Example:

```
1 // define container as vector<string>,
2 // list<string> or deque<string>
3 string word;
4 while ( cin >> word )
5   container.push_back( word );
```

- Subsequent changes to the element in container do not effect original object, and vice versa

push_front

- For `list`, `forward_list` and `deque` operation `push_front()` is most appropriate.
- Operation inserts a new element at the front of the container:

```
1 list<int> ilist;
2 for (size_t ix = 0 ; ix != 4 ; ++ix )
3   ilist.push_front( ix );
```

- note that `push_front` doesn't exist for `vector`

Adding an element at specified point

- Insert members are supported for `vector`, `deque`, `list` and `string`
- Insert takes iterator as first argument to indicate where to put elements (just before)
- For example:

```
1 // insert "Hello just before iter
2 // slist is list<string>
3 slist.insert( iter , "Hello" );
```

- inserts string "Hello" just before element `iter`
- It is legal to insert anywhere in a `vector`, `deque` or `string`, however it may be an expensive operation

Inserting a range of elements

- Arguments of **`insert`** after the initial iterator are analogous to container constructors
- Version taking specified number of identical elements at given iterator position:

```
svec.insert( svec.end(), 10, "Great!" );
```

- Versions taking pair of iterators or initializer list:

```
// already defined and filled list<string> slist
vector <string> v = {"Cabbibo", "Kobayashi",
    "Maskawa"};
slist.insert( slist.begin(), v.end()-2, v.end() );
slist.insert( slist.end(),
    {"Bardeen","Cooper","Schrieffer"} );
// error below: must not refer to same container we
// are inserting into
slist.insert( slist.begin(), slist.begin(),
    slist.end() );
```

Return value from insert

- `insert` returns an iterator to the first element that was inserted
- We can use value returned by `insert` to insert elements at position in container:

```
1 list<string> lst;
2 auto iter = lst.begin();
3 while ( cin >> word )
4   iter = lst.insert( iter, word );
```

- Above insert works same as `push_front`

Using the `emplace` operations

- `emplace`, `emplace_back` and `emplace_front` are analogous to the "push" operations
- Difference is `emplace` operations use arguments to construct element of type stored by container using type's constructor
- Example:

```
1 // example container holding Button objects
2 c.emplace_back( Point{0,0}, 0, 0, "flood",
      cb_flood );
3 c.emplace_back( iter, Point{0,0}, 0, 0, "fire",
      cb_flood );
4 // below is error, since push_back has no version
5 // taking five arguments:
6 c.push_back( Point{0,0}, 0, 0, "flood", cb_flood );
```

Accessing elements

- Sequential containers have a `front` member, and all except `forward_list` have `back` member:

```
1  // check that there are elements before
       dereferencing
2  // an iterator or calling front or back
3  if (!c.empty()) {
4    // val and val2 are copies of the value
5    // of the first element in c
6    auto val = *c.begin(), val2 = c.front();
7    // val3 and val4 are copies of the of the last
         element in c
8    auto last = c.end();
9    // below: can't decrement forward_list iterators
10   auto val3 = *(--last);
11   // below: not supported by forward_list
12   auto val4 = c.back();
13 }
```

Accessing elements

- `at` and subscript [] operator valid only for `string`, `vector`, `deque` and `array`
- operations below undefined if container `c` is `empty`

`c.back()`	get reference to last elem in `c`
`c.front()`	get reference to first elem in `c`
`c[n]`	get reference to element at index n
`c.at(n)`	returns reference to element at n throws an `out_of_range` exception

```
1 if (!c.empty()) {
2   c.front() = 42;      // assigns 42 to the first
        element in c
3   auto &v = c.back();// get a reference to the
        last element
4   v = 1024;            // changes the element in c
5   auto v2 = c.back();// v2 is not a reference;
        it's a copy of c.back()
6   v2 = 0;              // no change to the element
        in c
7 }
```

Erasing elements

`c.pop_back()`	remove last elem of `c`, return void
`c.pop_front()`	remove first elem of `c`, return void
`c.erase(p)`	remove elem at iterator `p`
	returns iterator of one after one erased
`c.erase(b,e)`	remove elements in range of iterators `b,e`
	returns iterator one after last one erased
`c.clear()`	remove all elements in `c`

Examples of erasing elements

```
// below ilist is a list<int> for example.
while (!ilist.empty()){
  // do something with current front of list
  process( ilist.front() );
  // done with that one, remove it
  ilist.pop_front();
}
list<int> lst={0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
// erase odd elements in list
while ( it != lst.end() ){
  if ( *it % 2 ){
    it = lst.erase(it);
  } else {
    ++it;
  }
}
```

How a vector grows

- `vector` elements are stored contiguously
- Need to consider what happens when we add an element
- If there is no room for new element, can't just add element at another memory location
- Instead allocate memory to hold the existing elements plus the new one
- If this happened each time we add an element, performance would be unacceptably slow
- Instead implementations typically allocate capacity beyond what is immediately needed
- Thus, there is no need to reallocate the container for each new element

Members to manage capacity

- `capacity()` operation tells us how many elements the container can hold before it must allocate more space.
- `reserve(n)` operation lets us tell the container how many elements it should be prepared to hold.
- `shrink_to_fit()` requests to reduce `capacity()` to equal `size()`

```
1  vector<int> ivec;
2  // size should be zero;
3  // capacity is implementation defined
4  cout << "ivec: size: " << ivec.size()
5       << " capacity: " << ivec.capacity() << endl;
6  // give ivec 24 elements
7  for (vector<int>::size_type ix = 0; ix != 24; ++ix)
8       ivec.push_back(ix);
9  // size should be 24; capacity will be >= 24 and is
       implementation defined
10 cout << "ivec: size: " << ivec.size()
11      << " capacity: " << ivec.capacity() << endl;
12 // result on my machine:
13 //ivec: size: 0 capacity: 0
14 //ivec: size: 24 capacity: 32
```

Additional string operations

- `string` has additional operations:
  - to interact with c-style character arrays,
  - to use indices in place of iterators
  - to convert strings into numeric types

Additional ways to construct strings

`string s(cp, n);`	`s` is copy of `n` char from array `cp`. array >= n char
`string s(s2, pos2);`	`s` is a copy of characters in string `s2` starting at index `pos2`
`string s(s2, pos2, len2);`	`s` is a copy of `len2` char from `s2` starting at index `pos2`

Examples constructing strings

```
1  // null-terminated array:
2  const char *cp = "Hello World!!!";
3  // not null terminated:
4  char noNull[] = {'H', 'i'};
5  // copy up to the null in cp; s1 == "Hello World!!!":
6  string s1(cp);
7  // copy two characters from no_null; s2 == "Hi":
8  string s2(noNull,2);
9  // undefined: noNull not null terminated:
10 string s3(noNull);
11 // copy 5 characters starting at cp[6]; s4 == "World":
12 string s4(cp + 6, 5)
13 // copy 5 characters starting at s1[6]; s5 == "World":
14 string s5(s1, 6, 5);
15 // copy from s1 [6] to end of s1; s6 == "World!!!":
16 string s6(s1, 6);
17 // ok, copies only to end of s1; s7 == "World!!!":
18 string s7(s1,6,20);
19 // throws an out_of_range exception:
20 string s8(s1, 16);
```

The `substr` operation

`s.substr(pos,n);`	return string containing `n` chars starting at `pos` [default: pos=0, n=size-1]

```
1 string s("hello world");
2 string s2 = s.substr(0, 5);   // s2 = hello
3 string s3 = s.substr(6);      // s3 = world
4 string s4 = s.substr(6, 11); // s3 = world
5 string s5 = s.substr(12);     // throws an out_of_range
    exception
```

assign, insert and erase for string

- string supports sequential container assignment operators and **assign**, **insert**, and **erase**
- also defines versions of these that use indices in place of iterators
- Example:

```
// insert five exclamation points at the end of s
s.insert(s.size(), 5, '!');
// erase the last five characters from s
s.erase(s.size() - 5, 5);
const char *cp = "Stately, plump Buck";
s.assign(cp, 7); //  s == "Stately"
s.insert(s.size(), cp + 7);
// now  s == "Stately, plump Buck"
```

append and replace for string

`s.insert(pos,args);`	insert characters specified by *args* before `pos` (index or iterator) returns iter or index to 1st inserted
`s.erase(pos,len);`	erase `len` characters starting at `pos` (index or iterator)
`s.assign(args);`	replace chars in `s` according to `args` returns a reference to `s`
`s.append(args);`	append `args` to `s`, return ref to `s`
`s.replace(range,args);`	remove range of chars from s replace with chars specified by args range is index+len or b+e iters
args is one of following forms	
`str`	the string str
`str, pos, len`	up to len char from str starting at pos
`cp, len`	up to len char from char array cp
`cp`	null terminated array pointed to by cp
`n, c`	n copies of character c
`b, e`	characters in range formed by iters b and e
initializer list	comma-separated list of chars in braces

args for replace, insert cont...

args depend on how range or pos is specified:

args can be	replace (pos, len, args)	replace (b,e,args)	insert (pos,args)	insert (iter,args)
str	yes	yes	yes	no
str, pos, len	yes	no	yes	no
cp, len	yes	yes	yes	no
cp	yes	yes	no	no
n, c	yes	yes	yes	yes
b2, e2	no	yes	no	yes
init-list	no	yes	no	yes

string search operations

search ops return index of char or npos if not found	
`s.find( args )`	find first occurrence of args in s
`s.rfind( args )`	find last occurrence os args in s
`s.find_first_of( args )`	find first occur of any char from args in s
`s.find_last_of( args )`	find last occur of any char from args in s
`s.find_first_not_of( args )`	find first char in s not in args
`s.find_lst_not_of( args )`	find last char in s not in args

args is one of	
c, pos	find char c starting at pos in s (pos default 0)
s2, pos	find string s2 starting at pos in s (pos default 0)
cp, pos	find null term c-array cp at pos in s (pos default 0)
cp, pos, n	find first n char in c-array cp at pos in is

string search operation examples

```
1  string name("AnnaBelle");
2  auto pos1 = name.find("Anna"); // pos1 == 0
3  string numbers("0123456789"), name("r2d2");
4  string dept("03714p3");
5  // below returns 5, index to the character 'p'
6  auto pp = dept.find_first_not_of(numbers);
7  string::size_type pos = 0;
8  // each iteration finds the next number in name
9  while ( (pos = name.find_first_of(numbers, pos) ) !=
       string::npos) {
10   cout << "found number at index: " << pos
11        << " element is " << name[pos] << endl;
12 }
13 // search backwards
14 string river("Mississippi");
15 auto first_pos = river.find("is");  // returns 1
16 auto last_pos = river.rfind("is");  // returns 4
```

Numeric conversion

- Strings often contain characters representing numbers
- C++ 11 supports several functions to convert between numeric data and strings:

```
int i = 42;
string s = to_string(i);  // converts the int i to
    its character
double d = stod(s);       // converts the string s
    to floating-point
string s2 = "pi = 3.14";
// convert the first substring in s that starts
    with a digit,  d = 3.14
d =
    stod(s2.substr(s2.find_first_of("+-.0123456789")));
```

Numeric conversions

Below `s` is string, `p` is `size_t` in which to put index of first non-numeric character in s (default 0 to not store index), and `b` is base (default 10).

`to_string(val);`	overloaded functions returning string representation of val (numeric types)
`stoi(s,p,b);`	string to int
`stol(s,p,b);`	string to long
`stoul(s,p,b);`	string to unsigned long
`stoll(s,p,b);`	string to long long
`stoull(s,p,b);`	string to unsigned long long
`stof(s,p);`	string to float
`stod(s,p);`	string to double
`stold(s,p);`	string to long double

Container adaptors

- `stack`, `queue`, and `priority_queue` are part of standard library defined as sequential container adaptors
- `stack` is first in, last out: `pop` elements in opposite order from what they were `pushed`
- `queue` is first in, first out: `pop` elements in same order they were `pushed`
- `priority_queue` is a queue that keeps elements in a quasi-sorted state, and `pop` elements in order of sorting
- **adaptor** is a general concept in library – there are container, iterator and function adaptors
- adaptor is mechanism to take existing container type and make it act like a different type

Operations of container adaptors

stack operations	
`s.pop()`	remove, but don't return top element of stack
`s.push(item)`	add element to top of stack with copy of item
`s.emplace(args)`	add element, using element init from args
`s.top()`	return, but don't remove top element of stack
queue and priority_queue operations	
`q.pop()`	remove front or highest priority element
`q.front()`	return but don't remove front element
`q.back()`	queue only, return don't remove back elem
`q.top()`	priority_queue only: return highest prior elem
`q.push(item)`	add element to end (or in priority order)
`q.emplace(args)`	as push but init type from args

`stack` adaptor

- `stack` adaptor takes a sequential container (other than array or forward list) and makes it operate as if it were a stack
- Example declaration of stack:

```
deque<int> deq;
stack<int> stk( deq );
```

- By default both `stack` and `queue` are implemented in terms of `deque`, and `priority_queue` on `vector`
- Can override default by providing second argument to adaptor:

```
// empty stack implemented on vector:
stack< string, vector<string> > str_stk;
// stack implemented on vector, initalized with
// string vector svec
stack< string, vector<string> > str_stk2( svec );
```

- `stack` is defined in same named header file

`stack` adaptor example

- Example use of stack:

```
1  stack<int> intStack;  // empty stack
2  // fill up the stack
3  for (size_t ix = 0; ix != 10; ++ix) {
4      intStack.push(ix);   // intStack holds 0 ... 9
           inclusive
5  }
6  // while there are still values in intStack
7  while (!intStack.empty()) {
8    int value = intStack.top();
9    // code that uses value 9 ... 0
10   // pop the top element, and repeat
11   intStack.pop();
12 }
```

Another `stack` example

- Reverse Polish notation calculator shows use of stack
- Notation is to push numbers to stack then apply operators on numbers at top of stack
- Example instead of writing $5 + (6 * 5 + 1)/3$ we write 5 6 5 * 1 + 3 / +
- also called Postfix operator
- Ie. we push the three numbers 5 6 5 to the stack then multiply last two numbers leaving 5 30 on the stack
- Next we push 1 on the stack then add last two numbers on stack, leaving 5 31 on the stack
- Next we push 3 on the stack, then divide last two numbers on stack leaving 5 and 10.3333 on the stack
- Finally we add the last two numbers on the stack leaving the answer 15.333 on the stack

Reverse polish calculator using stack

- To implement the stack the algorithm is:
  - Read a string from left to right (using istringstream)
  - If scanned input is a number push it to the stack
  - If the scanned input is an operator, pop two numbers from the stack and apply the operation
  - Repeat until string is scanned
  - Last element remaining on stack is answer

Code for stack calculator

```
1  stack<double> nums;      // stack to push/pop numbers
2  istringstream is( strin ); // input string to stream
3  token tok;     // location to put next token
4  // keep asking for another token until end of string.
5  while( getToken( is, tok ) ){
6    if ( tok.type == token::tkop ){ // token is operator
7      // make sure there are enough numbers on the stack
8      if ( nums.size() < 2 ) {
9        cerr << "Input error, operator doesn't have
             enough operands" << endl;
10       break;
11     }
12     double op1 = nums.top(); nums.pop();
13     double op2 = nums.top(); nums.pop();
14     nums.push( mathOp( op1, tok.oper, op2 ) );
15   } else { // token is a number
16     nums.push( tok.num );
17   }
18 }
```

Notes on stack calculator

- `token`, is a user defined type that holds either a number or an operator (as a char).

```
1 // simple structure to hold a token
2 // token is either a number or an operator
3 struct token {
4   enum token_type { tkop, tknum };
5   token_type type;
6   double num;
7   char oper;
8 };
```

Code to parse next token from an input stream

```
1  bool getToken( istream & is, token & t ){
2    static string ops{"+-*/"};
3    static string nums{"0123456789."};
4    char c; // skip white spaces
5    while ( is.peek() == ' ') is>>noskipws>>c;
6    is >> skipws; //return stream to default
7    if ( is.peek() == '\n' ) return false;
8    if ( ops.find( is.peek() ) != string::npos ) {
9      t.type = token::tkop; // operator
10     is >> t.oper;
11     if (!is) return false;
12     return true;
13   }
14   if ( nums.find( is.peek() ) != string::npos ) {
15     t.type = token::tknum; // number
16     is >> t.num;
17     if (!is) return false;
18     return true;
19   }
20   return false; // not number or op
21 }
```

Code to do the mathematical operation

- Need to check char used to represent the operation to decide which operation to perform on the two numbers.
- This is done in another helper function:

```
1  double mathOp( double op1, char oper, double op2 ){
2    switch ( oper ){
3    case '+':
4      return (op1 + op2);
5      break;
6    case '-':
7      return (op1 - op2);
8      break;
9    case '*':
10     return (op1 * op2);
11     break;
12   case '/':
13     return (op1 / op2);
14     break;
15   default:
16     cerr<<"invalid operator: "<< oper <<endl;
17     return 0.;
18   }
```

Try the calculator out!

- Have a look at `calcStack.cpp` which contains the code we have just reviewed, and try running it.

Week 11 Done!

- Reminder: Homework 8 is last homework is due Nov. 24 (17:00)
- Reminder: Final Project is due Dec. 1 (17:00)

My solution to adapt our function

- We can use a static variable to keep track of n

```
1  // wrapper for expo to store n for next call to
       expo
2  double expo1( const double x, const bool
       change=false, const int nn=1 ){
3    static int n = 1;
4    if ( change ) n = nn;
5    return expo( x, n );
6  }
7  // wrapper for expo1 to make single parameter
       approx
8  // to exponential(x)
9  double expo2( const double x ){
10   return expo1( x );
11 }
```

- Call `expo1( x, true, n)` to change value of n
- Call `expo2(x)` as single parameter function

approxExpo.cpp

```
for ( int n = 0 ;  n < 50 ; ++n ){
  ostringstream ss;
  ss << "Exp approximation to n=" << n;
  win.set_label( ss.str() );
  expo1( 1.0, true, n );
  Function e{ expo2, r_min, r_max, orig,
              200, x_scale, y_scale };
  win.attach(e);
  win.wait_for_button();
  win.detach(e);
}
```

- `win.detach` makes sure next iteration in loop doesn't have `win` try to draw Function that went out of scope