Week 11 : Graphical User Interfaces and Sequential Containers

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 11

Outline

Outline

User interfaces

- We will consider common case of interface to a computer with a keyboard, mouse and screen
- Three main choices for interface:
  - *Console input and output*: We have used this via <iostream> using `cin` and `cout`. It is probably the most commonly used method when solving technical problems.
  - *GUI library*: User interaction by manipulating objects on the screen (pointing, clicking, dragging, dropping, hovering, etc.). This is what we will learn this evening.
  - *Web browser interface*: Markup language where communication between program and screen is again textual. Browser is a GUI application that translates some of the text into graphical elements.

The "Next" button

- We have been using the `Simple_window` class that includes a "Next" button – how does that work?
- Each time we call `win.wait_for_button()` we look at objects on screen until we hit button to get output from the next part of the program
- This seems similar to `cout` to the screen, then wait for input via `cin >> var;`
- Of course implementation is quite different – GUI keeps track of mouse actions (clicking etc.)
- When program wants an action it must:
  - Tell GUI what to look for (eg. "Clicked next button")
  - Tell GUI what to do when someone does that
  - Wait until GUI detects action that program is interested in
- GUI does not just return to our program, it responds to clicking buttons, re-sizing windows, etc.
- We need a way to tell GUI what to do when particular button is clicked

A simple window

- We can ask the GUI to call a function when "something interesting" happens, such as clicking a particular button
- These functions are called "callback functions"
- To do that we must:
  - Define a button
  - Get it displayed
  - Define a function for the GUI to call
  - Tell the GUI about that button and that function
  - Wait for the GUI to call our function

Simple_window's button

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h,
                  const string& title );
    // simple event loop call
    void wait_for_button();
  private:
    // the "Next" button
    Button next_button;
    // implementation detail:
    bool button_pushed;
    // callback for next_button
    static void cb_next(Address,
                        Address pw);
    // action to be done when next_button
    //  is pressed:
    void next();
};
```

Notes on `Simple_window`'s button

- We see that `Simple_window` inherits from `Graph_lib::Window`
- Our button is initialized in the constructor:

```
Simple_window::Simple_window(Point xy,
        int w, int h, const string& title) :
    Window{xy,w,h,title},
    next_button{ Point{x_max()-70,0},
                 70, 20, "Next", cb_next},
    button_pushed{ false }
{
  attach(next_button);
}
```

- We see that location, size and title are passed on to `Window` to deal with
- The `next_button` is then initialized with: a location `Point{x_max-70,0}` , a size (70, 20), a label "Next", and a callback function `cb_next`
- Finally the `next_button` is attached to the window

Notes on `Simple_window`'s button

- The `button_pushed` Boolean is a detail – it is used to keep track of whether the button was pressed since the last time `next()` was called
- In fact the only public part of the implementation is the constructor, and the `wait_for_button()`
- The private method `cb_next()` is what we want the GUI to call when the "Next" button is clicked
- We used the `cb_` prefix to indicate that it is meant to be a callback function
- Recall that program runs on top of several layers (Our program -> GUI interface code -> `fltk` -> Operating system -> Device driver)
- A click detected by mouse driver has to cause `cb_next()` to be called
- We give `Simple_window` the address of `cb_next()` which gets passed down through the software layers so that it can get called

Notes on `cb_next()`

- Type required for callback function is chosen so that is usable from the lowest level of programming, including C and assembler
- A callback returns no value, takes two addresses as args
- Declare a C++ member function that obeys those rules as:

```
1 // callback for next next_button
2 static void cb_next( Address, Address pw);
```

- The keyword `static` makes sure `cb_next()` can be called as an ordinary function
- Ie. it is not a C++ member function invoked for a specific object
- It would have been nice to be able to use a member function, but that's not what we get when talking to lower level code
- The `Address` arguments specify that the function takes two addresses of "somethings" in memory
- First Address we don't need so it is not given a name
- Second Address is of the window containing that "Widget"

Notes on `cb_next()`

- We can use Address as follows:

```
1 void Simple_window::cb_next(Address, Address pw)
2 // call Simple_window::next() for the window
3 // located at pw
4 {
5     reference_to<Simple_window>(pw).next();
6 }
```

- The `reference_to<Simple_window>(pw)` tells the compiler that the address in `pw` is to be treated as an address of a `Simple_window`
- From the reference, we call the member function `next()` – out of the system-dependent code as quick as possible
- Purpose of keeping `next()` separate from `cb_next()` was to keep system-dependent part separate from our user code
- `next()` function then performs the desired action

Review of steps to get to our `next()` function

- Steps to get our `next()` function called:
  1. Define our `Simple_window`
  2. `Simple_window` registers its `next_button` with GUI system
  3. We click the image of the `next_button` and GUI calls `cb_next()`
  4. `cb_next()` converts low-level system info into a call to our `next()` function for our window
  5. `next()` performs whatever action we want done in response to the button click
- Technique deals with all kinds of interactions – not just our button push
- Window can have many buttons, and program many windows
- Understanding how `next()` is called – we understand how to deal with every action in a program with GUI interface

A wait loop

- `next()` button in this case stops execution of program by setting `button_pushed` to true
- Ie. we had called:

```
win.wait_for_button();
```

- Which calls `fltk`'s version of `wait()` that takes care of everything until the "Next" button is pressed:

```
void Simple_window::wait_for_button()
// modified event loop:
// handle all events (as per default),
// quit when button_pushed becomes true
// this allows graphics without control inversion
{
    while (!button_pushed) Fl::wait();
    button_pushed = false;
    Fl::redraw();
}
```

A lambda expression as a callback

- For each button we define two functions: one to map from system's notion of a callback, and one to do our desired action
- For example, we eliminate `cb_next()` and rewrite the constructor as:

```
1  Simple_window::Simple_window( Point xy,
2    int w, int h, cost string& title ) :
3    Window{ xy, w, h, title },
4    next_button{ Point{x_max()-70,0}, 70, 20, "Next",
5        [](Address,Address pw){
6          reference_to<Simple_window>(pw).next(); } },
7    button_pushed{false}
8  {
9    attach( next_button );
10 }
```

Button

- We define a `Button` as:

```
1 struct Button : Widget {
2     Button(Point xy, int w, int h, const string&
          label, Callback cb)
3         : Widget{ xy, w, h, label, cb} {}
4     void attach(Window&);
5 };
```

- The `Button` is a `Widget` with a location `xy`, a size `(w,y)` and a text label tt label
- A *Widget* is the technical term for a control.
- We use widgets to define the forms of interaction with a GUI

Widget

- Our `Widget` interface class looks like:

```
typedef void(*Callback)(Address, Address);
class Widget {
  public:
    Widget( Point xy, int w, int h,
            const string& s, Callback cb);
    virtual void move( int dx, int dy );
    virtual void hide();
    virtual void show();
    virtual void attach( Window& ) = 0;
    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;
  protected:
    Window* own;  // Window containing Widget
    Fl_Widget* pw;// FLTK Widget
};
```

Notes on `Widget`

- A `Widget` starts out visible, but can be made invisible, or visible again using the methods `hide()` and `show()`
- Like a `Shape` we can move a `Widget` using the `move()` method
- A `Widget` needs to be attached to a `Window` before it is used
- Note that `attach()` was declared as a "pure virtual" function
- That makes `Widget` a base class – representing a window control
- Each class derived from `Widget` must define its `attach()` method with what it means for it to be attached to `Window`
- `Widget::attach()` is what is called by `Window::attach()` as part of its implementation
- Result is that `Window` then knows about the `Window` and the `Widget` knows which `Window` it is in

Notes on `Widget`

- Definition of `Widget` and classes depending on it such as `Button`, `Menu`, etc. are in GUI.h
- Note that `Window` doesn't know what kind of `Widget` it deals with
- Similarly `Widget` doesn't know what kind of `Window` it deals with
- Note that some data members were left accessible
- The `Window * own` and `Fl_Widget * pw` members are strictly for the implementation, so are declared as `protected`

Notes on `Button`

- Recall the `Button` interface is declared as:

```
1 struct Button : Widget {
2     Button( Point xy, int w, int h,
3             const string& label, Callback cb)
4         : Widget(xy,w,h,label,cb){ }
5     void attach(Window&);
6 };
```

- the `attach()` function contains the relatively messy `fltk` code, which we won't look at for now
- note that placing a `Shape` over a button doesn't affect the button's s ability to function
- The shape of the button is left beyond our control, other than its location and size

In_box

- Two widgets are provided to get text from the GUI into and out of our program. Consider the `In_box` interface:

```
struct In_box : Widget {
  In_box(Point xy, int w, int h, const string& s)
    :Widget(xy,w,h,s,0) { }
  int get_int();
  string get_string();
  void attach(Window& win);
};
```

- The `In_box` accepts text typed into it.
- We get the text as a string with `get_string()`
- Or we can get an integer using `get_int()`
- Note that you can get a float by using stringstream to parse the string

Out_box

- A widget for presenting some message a user is the `Out_box`, with interface:

```
1 struct Out_box : Widget {
2   Out_box(Point xy, int w, int h, const string& s)
3     :Widget(xy,w,h,s,0) { }
4   void put(int);
5   void put(const string&);
6   void attach(Window& win);
7 };
```

- Similar to `In_box` it provides methods for output of an int (`put(int)`) and for output of a string (`put( const string &)`)

Menu

- A widget for a vector of buttons as a `Menu` has interface:

```
struct Menu : Widget {
  enum Kind { horizontal, vertical };
  Menu(Point xy, int w, int h, Kind kk, const
      string& label);
  Vector_ref<Button> selection;
  Kind k;
  int offset;
  int attach(Button& b);  // Menu does not delete
     &b
  int attach(Button* p);  // Menu deletes p
  void show(); // show all buttons
  void hide(); // hide all buttons
  void move(int dx, int dy); // move all buttons
  void attach(Window& win); // attach all buttons
};
```

- The menu is simply a vector of buttons laid out in a line
- You provide the top left corner, a width and height to resize buttons as they are added to the menu
- Note this is not a “pop up” menu

Example

- Lets build a simple window that allows a user to display a sequence of lines (`Open_polyline`) specified by user input of coordinate pairs
- The window will look like:

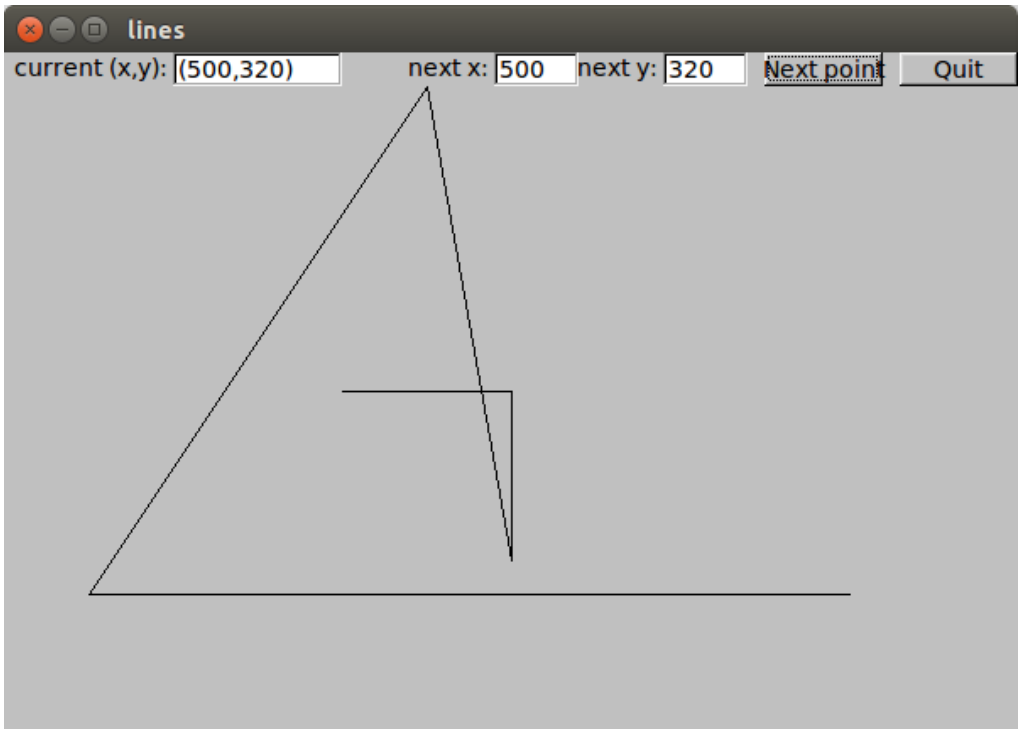


Figure : First example of using Widgets

Example

- Initially “current(x,y)” box is empty, and program waits for user to enter coord pair
- User enters values in “next x”, “next y” boxes and clicks “Next point” button to add the point
- Once done starting point appears in the “current(x,y)” box
- Each new coordinate pair entered draws a line from “current(x,y)” to the pair entered, and updates “current(x,y)”
- When the user is done they click a “Quit” button.

Class to represent window for drawing lines

```
struct Lines_window : Graph_lib::Window {
  Lines_window( Point xy, int w, int h,
                const string& title );
  Open_polyline lines;
private:
  // add (next_x,next_y) to lines
  Button next_button;
  Button quit_button;
  In_box next_x;
  In_box next_y;
  Out_box xy_out;
  // callback for next_button
  void next();
  // callback for quit_button
  void quit();
};
```

Notes on `Lines_window` class

- Line is represented by `Open_polyline`
- `Button`s and boxes are declared
- Each button has a member function implementing the desired action is defined `next()` and `quit()`
- Rather than having `cb_next()` and `cb_quit()` we use lambdas in the constructor.

Constructor for `Lines_window` class

```
Lines_window::Lines_window( Point xy, int w, int h,
                            const string& title ) :
  Window{xy, w, h, title},
  next_button{ Point{x_max()-150,0}, 70, 20,
    "Next point", [](Address, Address pw){
       reference_to<Lines_window>(pw).next(); } },
  quit_button{ Point{x_max()-70,0}, 70, 20, "Quit",
     [](Address,Address pw){
       reference_to<Lines_window>(pw).quit(); } },
  next_x{ Point{x_max()-310,0}, 50, 20, "next x:"},
  next_y{ Point{x_max()-210,0}, 50, 20, "next y:"},
  xy_out{ Point(100,0), 100, 20, "current (x,y):"}
{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    attach(lines);
}
```

Lines_window::quit() and ::next()

- "Quit" button deletes the window
- This is done by call to `fltk`'s `hide()` function to delete the window from view

```
1  void Lines_window::quit(){
2    hide();
3  }
```

- `next()` button does real work of reading coord pair, updating `Open_polyline`, and updating `Out_box`:

```
1  void Lines_window::next(){
2    int x = next_x.get_int();
3    int y = next_y.get_int();
4    lines.add(Point(x,y));
5    // update current position readout:
6    stringstream ss;
7    ss << '(' << x << ',' << y << ')';
8    xy_out.put(ss.str());
9    redraw();
10 }
```

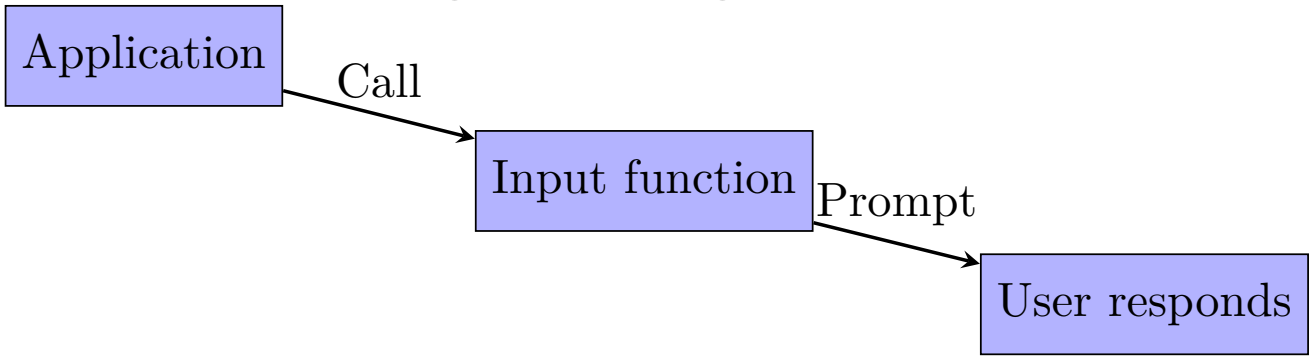
`main` function for `lineDrawing.cpp`

- Body of main is very short – we make our window and call `gui_main()` to hand control over to the `GUI`

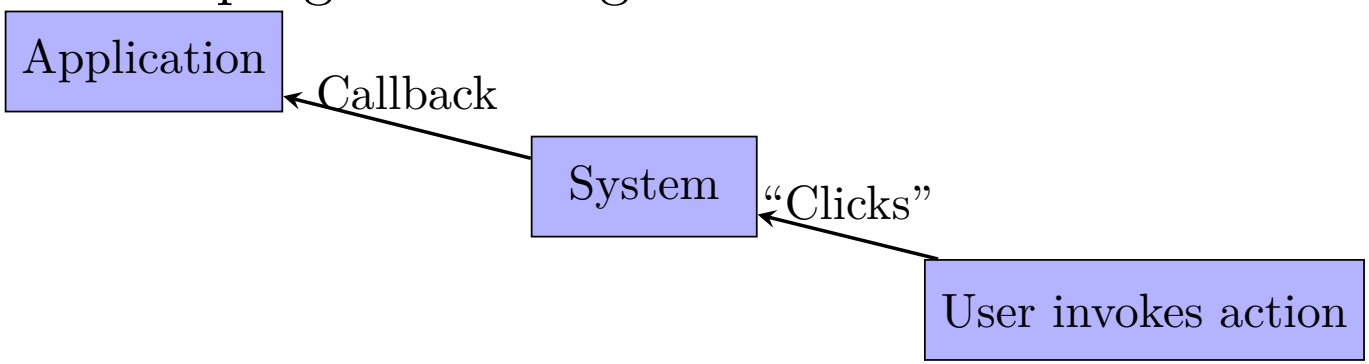
```
1  int main()
2  try {
3      Lines_window
           win(Point(100,100),600,400,"lines");
4      return gui_main();
5  }
6  catch(exception& e) {
7      cerr << "exception: " << e.what() << '\n';
8      return 1;
9  }
10 catch (...) {
11     cerr << "Some exception\n";
12     return 2;
13 }
```

Control inversion

- Conventional program is organized like this:



- A “GUI program” is organized like this:



Control inversion

- Implication of control inversion is that order of execution is determined by user interaction
- Makes systematic testing harder
- When coding GUI take extra care by making incremental improvements with testing in between
- Function invoked by a callback can do anything
- For now we'll just consider simplest case in which we hold data in our window

Adding a menu

- Lets provide a menu to change the color of all of the lines that are added:

```
1  struct Lines_window : Window {
2    //...
3    Menu color_menu;
4    static void cb_red( Address, Address);
5    static void cb_blue( Address, Address);
6    static void cb_black( Address, Address);
7    void red_pressed(){ change(Color::red); }
8    void blue_pressed(){ change(Color::blue); }
9    void black_pressed(){ change(Color::black); }
10   void change(Color c){ lines.set_color(c); }
11   //...
12 }
```

Initializing menu callbacks

- We also need to set up the callback in the constructor:

```
Line_window::Line_window( Point xy, int w, int h,
  const string& title ) :
  //...
  color_menu{ Point{x_max()-70,40},
    70, 20, Menu::vertical, "color" }
{
  //...
  color_menu.attach( new Button{ Point{0,0},
    0, 0, "red", cb_red } );
  color_menu.attach( new Button{ Point{0,0},
    0, 0, "blue", cb_blue } );
  color_menu.attach( new Button{ Point{0,0},
    0, 0, "black", cb_black } );
  attach( color_menu );
}
```

- Buttons are attached to menu using `attach()`
- `Menu::attach()` sets size and location of buttons and attaches them to the window

Result of adding menu

Figure : `Lines_window` with color menu

Making our menu a pop open menu

- Lets add a “color menu” button that when pressed opens the color menu to choose one of the colors
- To do that we add another Button, and “pressed” functions to adjust visibility of that button and the color menu
- Here is what we add:

```
1  struct Lines_window : Window {
2    //... below all in private:
3    Button menu_button;
4    void hide_menu() {
5      color_menu.hide();  menu_button.show(); }
6    void menu_pressed() {
7      menu_button.hide(); color_menu.show(); }
8    void red_pressed() {
9      change(Color::red); hide_menu(); }
10   void blue_pressed() {
11     change(Color::blue); hide_menu(); }
12   void black_pressed() {
13     change(Color::black); hide_menu(); }
14   void menu_pressed() {
15     menu_button.hide(); color_menu.show(); }
16   //...
```

Updated constructor

```
1  Lines_window::Lines_window(Point xy, int w, int h,
       const string& title)
2    : Window{ xy, w, h, title},
3  //...
4    menu_button(Point(x_max()-80,30), 80, 20,
5       "color menu", cb_menu),
6  {
7    //...
8    attach(color_menu);
9    color_menu.hide();
10   attach(menu_button);
11   //...
12 }
```

- The initializers should be in the same order as the data member definitions
- Compilers will likely give a warning if they are out of order

Result of making popup menu

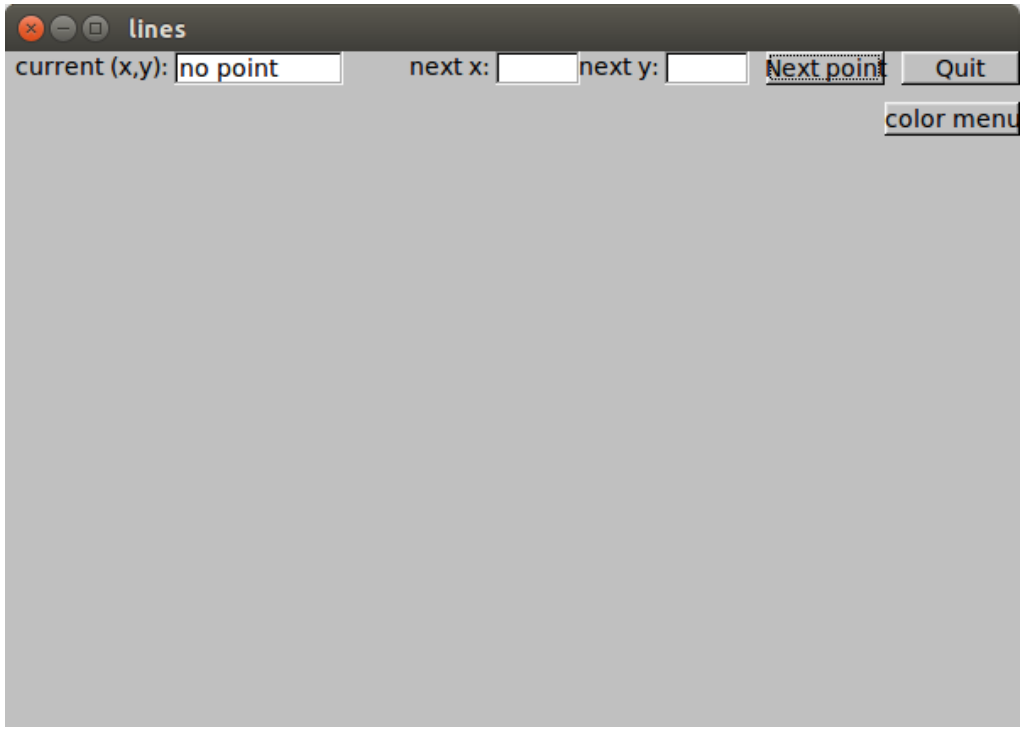


Figure : `Lines_window` with popup menu

Debugging GUI code

- Can be quite frustrating before first shapes and widgets start appearing in window
- Try this main():

```
1 int main(){
2   Lines_window{ Point{100,100}, 600, 400, "lines"
        };
3   return gui_main();
4 }
```

- Find the error! Try it even if you do spot the error.

Debugging GUI code

- How do you find errors in GUI programs?
  - Using well tried program parts
  - Simplifying all new code and slowly "growing" program
  - Carefully looking over the code line by line
  - Checking all linker settings
  - Comparing to code in an already working program
  - Explaining the code to a friend
- It is harder to trace execution of code – since it no longer goes in order
- Can also try using a “debugger” program
- Your code is not only one trying to interact with the screen

Debugging code

- Common problems to look for include:
  - Make sure object isn't hidden behind another object
  - Make sure `Shape` or `Widget` is attached, and that it doesn't go out of scope
- Most common problem is having object go out of scope after being attached, for example:

```
1  void disaster_fillmenu( Menu& m ){
2    Point zero{0,0};
3    Button b1{ zero, 0, 0, "flood", cb_flood );
4    Button b2{ zero, 0, 0, "fire", cb_fire );
5    m.attach( b1 );
6    m.attach( b2 );
7  }
8  int main(){
9    Menu disasters {Point{100,100},
10     60, 20, Menu::horizontal, "disasters" };
11   disaster_fillmenu( disasters );
12   win.attach( disasters };
13   return 0;
14 }
```

What is wrong with previous example code?

- All buttons were local to `disaster_fillmenu` function
- Attaching buttons to menu doesn't change scope of buttons
- One solution is to use `new` to allocate new memory for the objects – but then we lose the pointer to that location:

```
1 void disaster_fillmenu( Menu &m ){
2   Point zero{0,0};
3   m.attach( new Button{zero, 0, 0, "flood",
        cb_flood } );
4   m.attach( new Button{zero, 0, 0, "fire", cb_fire
        } );
5 }
```

- Another solution is to have a `Vector_ref< Button >` that is in a scope outside of `disaster_fillmenu` to hold the Buttons

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

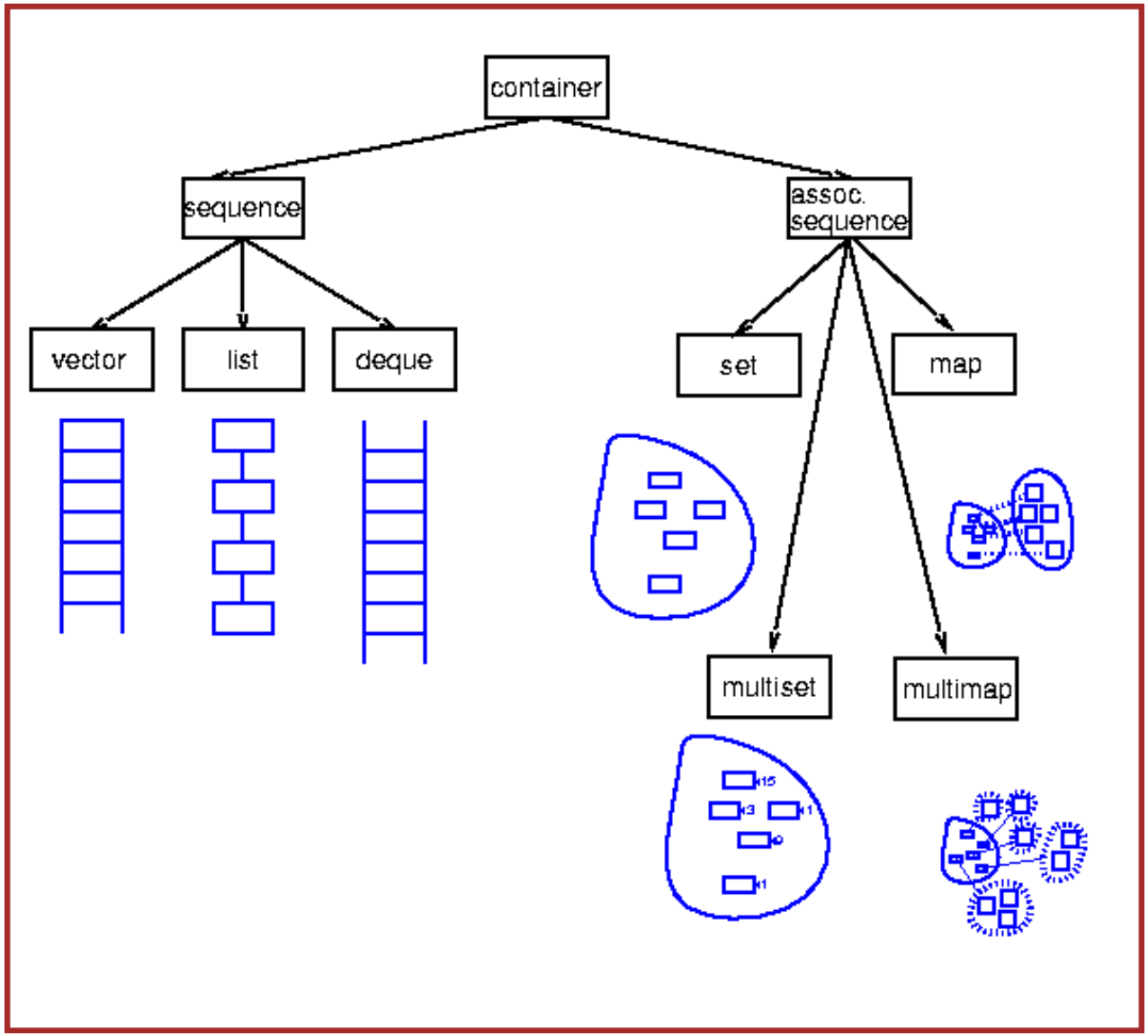


Figure : Image from
cs.brown.edu/~jak/proglang/cpp/stltut/tut.html

`list` and `forward_list`

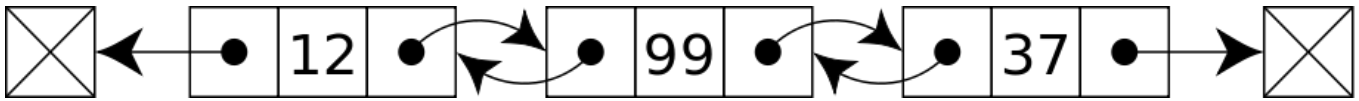


Figure : Doubly-linked list (`list`) Public domain images from `en.wikipedia.org/wiki/Linked_list`.

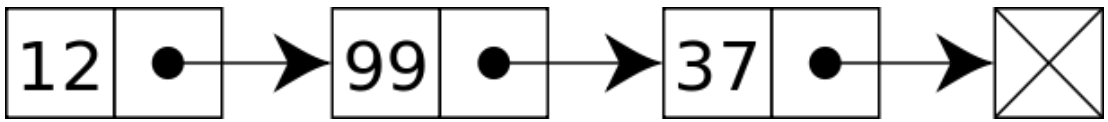


Figure : Singly-linked list (`forward_list`). Public domain images from `en.wikipedia.org/wiki/Linked_list`.

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:

```
1 while ( begin != end ){
2   * begin = val; // ok: range isn't empty so begin
        is an element
3   ++ begin;      // advance iterator to next
        element
4 }
```

- Reverse iterator version:

```
1 while ( rend != rbegin ){
2   * rend = val; // ok: range isn't empty so rend
        is an element
3   --rend; // decrement iterator to previous element
4 }
```

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:

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

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

```
// null-terminated array:
const char *cp = "Hello World!!!";
// not null terminated:
char noNull[] = {'H', 'i'};
// copy up to the null in cp; s1 == "Hello World!!!":
string s1(cp);
// copy two characters from no_null; s2 == "Hi":
string s2(noNull,2);
// undefined: noNull not null terminated:
string s3(noNull);
// copy 5 characters starting at cp[6]; s4 == "World":
string s4(cp + 6, 5)
// copy 5 characters starting at s1[6]; s5 == "World":
string s5(s1, 6, 5);
// copy from s1 [6] to end of s1; s6 == "World!!!":
string s6(s1, 6);
// ok, copies only to end of s1; s7 == "World!!!":
string s7(s1,6,20);
// throws an out_of_range exception:
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:

```
1 // insert five exclamation points at the end of s
2 s.insert(s.size(), 5, '!');
3 // erase the last five characters from s
4 s.erase(s.size() - 5, 5);
5 const char *cp = "Stately, plump Buck";
6 s.assign(cp, 7); //  s == "Stately"
7 s.insert(s.size(), cp + 7);
8 // 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. 25 (17:00)
- Reminder: Final Project is due Nov. 29 (17:00)