HIGH-LEVEL PROGRAMMING 2

Useful C++ ADTs by Prasanna Ghali

Abstraction and Encapsulation

- Abstraction: design technique for reducing complexity that identifies which specific information should be visible [the *interface*] and which should be hidden [the *implementation*]
- Encapsulation: packaging technique provided by programming language to hide implementation and to make visible interface

Abstract Data Types

- Interface created using data abstraction and encapsulation that defines high-level data type and operations on values of that type
- Clients only need to worry about how to use ADT interface and not on ADT implementation

Plan For Today

- Look at useful ADTs provided by C++ standard library such as: `std::array`, `std::initializer_list`, `std::pair`, `std::string`, `std::vector`, …

Static Array Usage in C And C++


```
double average(double arr[], std::size_t size) {
  if (!size) { // avoid later division by 0
    return 0.;
  }
  double sum {};
  for (std::size_t i{}; i < size; ++i) {
    sum += arr[i];
  }
  return sum/size;
}

int main() {
  int const MAX_STUDENTS {5};
  double grades[MAX_STUDENTS] {11.1, 22.2, 33.3, 44.4, 55.5};
  std::cout << "Average: " << average(grades, MAX_STUDENTS);
}
```

Problems With Static Arrays (1/3)

- **No runtime boundary checking occurs when reading from and writing to array!!!**
  - Reading/writing past array bounds results in undefined behaviour
  - C/C++ compilers don't provide help in detecting reading/writing past array bounds

```
double grades[5] {11.1, 22.2, 33.3, 44.4, 55.5};
int i = 5000;
// this call to crashes the program on my PC ...
std::cout << "grades[" << i << "]: " << grades[i] << "\n";
```

Problems With Static Arrays (2/3)

- Another insidious error …

insidious because some other variable is inadvertently getting updated …

```
double grades[5] {11.1, 22.2, 33.3, 44.4, 55.5};
int i = -5;
// writing to and reading from outside array ...
grades[i] = 66.6;
```

Problems With Static Arrays (3/3)

- Comparison of static arrays doesn't work as expected!!!

```
double mine[]   {1.1, 2.2, 3.3, 4.4};
double yours[]  {1.1, 2.2, 3.3, 4.4};
double theirs[] {1.1, 1.1, 3.3, 4.4};

if (mine == yours)   { std::cout << "mine == yours\n"; }
if (mine != theirs)  { std::cout << "mine != theirs\n"; }
if (mine > yours)    { std::cout << "mine > yours\n"; }
```

In an expression, name of static array decays to pointer to first array element

Problems With Dynamic Arrays

- Things are worse when we don’t know array size at compile-time!!!

```
int student_count;
// read input file to determine number of students ...
// allocate appropriate memory on free store ...
int *grades = new [student_count];

// read grades from input file into dynamic array ...
// process grades ...

// don't forget to return memory back to free store ...
```

Pointers Are Error Prone

- Dereferencing uninitialized pointers
- Dereferencing `nullptr`s
- Reading uninitialized objects that are dynamically allocated
- Failing to `delete` [or `delete[]`] allocated memory causing memory leak
- Calling `delete` rather than `delete[]` and vice versa
- Accessing `delete`d memory
- Double `delete`ing dynamically allocated objects
- Premature deletion causes dangling pointers
- Off-by-one array subscripting

Modern C++ Alternatives

- In modern C++, best to avoid C-style arrays!!!
- Instead use C++ standard library functionality: `std::array`, `std::initializer_list`, `std::string`, `std::vector`, …

It is *easy to use these types correctly* to avoid problems related to C-style arrays and C-strings and *hard to use these types incorrectly* to make mistakes common to C-style arrays and C-strings

std::array<T,N>: Usage (1/2)

- Modern C++ provides C++ standard library type `std::array<T,N>` as replacement for static C-style arrays!!!

```
#include <array>

std::array<int, 4> a0;     // elements uninitialized
std::array<int, 4> a1 {}; // elements initialized to 0
std::array<int, 4> a2 {2, 4, 6, 8};
a0.fill(11);   // set all elements to 11
int x = a0[2]; // operator[] is overloaded
a0[2] *= 2;
std::cout << a0.size() << '\n'; // prints 4

int sa[] {2, 4, 6, 8};
std::cout << std::size(sa) << ' ' << std::size(a0) << '\n';
```

std::array<T,N>: Usage (2/2)


```
std::array<double, 5> da{1.1, 2.2, 3.3, 4.4, 5.5};

double total{};
for (size_t i{}; i < da.size(); ++ i) {
  total += da[i];
}
```

```
// range for is better than indices ...
std::array<double, 5> da{1.1, 2.2, 3.3, 4.4, 5.5};

double total{};
for (double val : da) {
  total += val;
}
```

Passing array<> (1/4)

- Passing static arrays [by value] to functions is troublesome!!!

```
double sum(double const arr[], size_t N) {
  double total{};
  for (size_t i{}; i < N; ++i) {
    total += arr[i];
  }
  return total;
}

double sad[5] {1.1, 2.2, 3.3, 4.4, 5.5};
std::cout << sum(sad, 5UL);
```

Passing array<> (2/4)

- Passing static arrays by reference to functions is better!!!

```
double sum(double const (&arr)[5]) {
  double total{};
  for (size_t i{}; i < 5; ++i) {
    total += arr[i];
  }
  return total;
}

double sad[5] {1.1, 2.2, 3.3, 4.4, 5.5};
std::cout << sum(sad);
```

Passing `array<>` (3/4)

- Passing static arrays by reference to functions is better!!!
- Range `for` statement is better!!!

```
double sum(double const (&arr)[5]) {
  double total{};
  for (double val : arr) {
    total += val;
  }
  return total;
}

double sad[5] {1.1, 2.2, 3.3, 4.4, 5.5};
std::cout << sum(sad);
```

Passing array<> (4/4)

- Passing array<>s to functions is simpler than passing static arrays and as efficient

```
#include <limits>
#include <array>

int largest(std::array<int, 1'000'000> const& value) {
  int large_val {std::numeric_limits<int>::min()};
  for (int x : value) {
    large_val = (x > large_val) ? x : large_val;
  }
  return large_val;
}

int main() {
  std::array<int, 1'000'000> big_array;
  // fill big_array with values ...
  int largest_value = largest(big_array);
}
```

Operations on `array<>`s As a Whole (1/2)

- Can't compare static arrays without using loops

```
// compiles but doesn't not as intended!!!
double mine[]   {1.1, 2.2, 3.3, 4.4};
double yours[]  {1.1, 2.2, 3.3, 4.4};
double theirs[] { 1.1, 1.1, 3.3, 4.4};

if (mine == yours)  { std::cout << "mine and yours are equal\n"; }
if (mine != theirs) { std::cout << "mine and theirs are equal\n"; }
if (mine > yours)   { std::cout << "mine is greater than yours\n"; }
```

Operations on array<>s As a Whole (2/2)

- array<> containers can be compared using any of comparison operators as long as containers are of same size and store elements of same type!!!

```
std::array<double,4> mine   {1.1, 2.2, 3.3, 4.4};
std::array<double,4> yours  {1.1, 2.2, 3.3, 4.4};
std::array<double,4> theirs {1.1, 1.1, 3.3, 4.4};

// lexicographic comparison!!!
if (mine == yours)  { std::cout << "mine and yours are equal\n"; }
if (mine != theirs) { std::cout << "mine and theirs are equal\n"; }
if (mine > yours)   { std::cout << "mine is greater than yours\n"; }
```

array<>s Simplify Use Of Multidimensional Arrays (1/2)


```
std::ostream&
operator<<(std::ostream& os, std::array<int,3> const& row);

std::ostream& operator<<(std::ostream& os,
          std::array<std::array<int,3>,4> const& mat);

// define 4 X 3 2d array ...
std::array<std::array<int,3>, 4> mat43 = {
  std::array<int,3>{1, 2, 3}, std::array<int,3>{4, 5, 6},
  std::array<int,3>{7, 8, 9}, std::array<int,3>{10, 11, 12}
};

// print 2d matrix of size 4 X 3 ...
std::cout << "4 x 3 matrix is:\n" << mat43 << '\n';
```

array<>s Simplify Use Of Multidimensional Arrays (2/2)


```
std::ostream&
operator<<(std::ostream& os, std::array<int,3> const& row) {
  for (int x : row) {
    std::cout << std::setw(2) << x << ' ';
  }
  return os;
}

std::ostream& operator<<(std::ostream& os,
                std::array<std::array<int,3>,4> const& mat) {
  for (std::array<int,3> const& col : mat) {
    std::cout << col << '\n';
  }
  return os;
}
```

`using` Keyword [1st Use]

- `using` *declaration* makes specific names declared in a namespace accessible without requiring namespace and `::` operator

```
#include <iostream>

int cout {1};

int main() {
  using std::cout; // using declaration
  cout << "hello world: " << ::cout << "\n";
}
```

This `using` declaration hides previous meaning that name `cout` had in outer scope!!!

`using` Keyword [2nd Use] (1/4)

- `using` *directive* makes ~~specific~~ all names declared in a namespace accessible without requiring namespace and `::` operator
- Worst possible C++ feature!!!
- Why?

`using` Keyword [2nd Use] (2/4)

- Suppose you've following situation [where everything works as expected]

```cpp
// from graphics.hpp ...
namespace Graphics {
  void foo(int);
  // other stuff ...
}

// from ai.hpp ...
namespace AI {
  void bar();
  // other stuff ...
}
```

```cpp
// this is your source ...

#include "graphics.hpp"
#include "ai.hpp"

using namespace Graphics;
using namespace AI;

// suppose you're authoring function baz
void baz() {
  bar();      // do some AI stuff ...
  foo(10.1); // do some graphics stuff ...

  // some other functions are called ...
}
```

using Keyword [2nd Use] (3/4)

- Now, interface in `namespace AI` is expanded [causing your code to insidiously go wrong]

```
// from graphics.hpp ...
namespace Graphics {
  void foo(int);
  // other stuff ...
}

// from ai.hpp ...
namespace AI {
  void bar();
  void foo(double);
  // other stuff ...
}
```

```
// this is your source ...

#include "graphics.hpp"
#include "ai.hpp"

using namespace Graphics;
using namespace AI;

// suppose you're authoring function baz
void baz() {
  bar();      // do some AI stuff ...
  foo(10.1); // which foo?

  // some other functions are called ...
}
```

`using` Keyword [2nd Use] (4/4)

- `using` *directive* makes ~~specific~~ all names declared in a namespace accessible without requiring namespace and `::` operator
- Simply assume this feature doesn't exist and you'll never have any trouble with it!!!

`using` Keyword [3rd Use] (1/3)

- `using` *alias declaration* introduces a C style `typedef` name

any built-in or derived type

```
typedef    existing-type    IDENTIFIER;
```

keyword

new name for existing type

any built-in or derived type

```
using      IDENTIFIER = existing-type;
```

keyword

new name for existing type

using Keyword [3rd Use] (2/3)

```
int incr(int val) {
  return val+1;
}

typedef int INTC;
typedef int (*PFC)(int);
typedef int (&RFC)(int);

using INTCPP = int;
using PFCPP  = int(*)(int);
using RFCPP  = int(&)(int);
```

```
int main() {
  INTC   x {-1};
  PFC    pfc_incr = incr;
  RFC    rfc_incr = incr;

  INTCPP y {-1};
  PFCPP  pfcpp_incr = incr;
  RFCPP  rfcpp_incr = incr;

  std::cout << pfc_incr(++x) << '\n';
  std::cout << rfc_incr(++x) << '\n';
  std::cout << pfcpp_incr(++y) << '\n';
  std::cout << rfcpp_incr(++y) << '\n';
}
```

`using` Keyword [3rd Use] (3/3)

- Always use `using` keyword to define type aliases in C++ source files
- Forget about keyword `typedef` in C++
- Use keyword `typedef` only in C code and for maintaining legacy code …

Type Aliasing: Example


```
// simplify declarations for 4 X 3 2d array of ints
using COL3  = std::array<int, 3>;
using MAT43 = std::array<COL3, 4>;

std::ostream& operator<<(std::ostream& os, COL3 const& row) {
  for (int x : row) { std::cout << std::setw(2) << x << ' '; }
  return os;
}

std::ostream& operator<<(std::ostream& os, MAT43 const& mat) {
  for (COL3 const& col : mat) { std::cout << col << '\n'; }
  return os;
}

MAT43 mat43 {COL3{1,2,3}, COL3{4,5,6}, COL3{7,8,9}, COL3{0,1,2} };
std::cout << "4 x 3 matrix is:\n" << mat43 << '\n';
```

`array<>`: Final Word

- Stop using static arrays and prefers `array<>`s
- Everything you do with static arrays can now be done easily, safely, and flexibly with `array<>`s!!!
- Microsoft has good reference [page]

Initializer Lists

- `initializer_list<>` is standard library type that represents array of values of specified type but without array interface!!!
- Useful when you want to define function that takes an *unknown number* of values of *same type*

initializer_list<> Operations (1/2)


```
// default initializer: empty list of elements of type T
std::initializer_list<T> lst;

// lst2 has as many elements as there are initializers;
// elements are copies of corresponding initializers
// Elements in list are immutable and hence const
std::initializer_list<T> lst2 {a, b, c, ...};

// lst3 doesn't make copies of elements of lst2!!!
// Instead, both lst2 and lst3 share elements
std::initializer_list<T> lst3(lst2);

// same as above: both lst and lst2 share elements
lst = lst2;

// cannot use braces ...
std::initializer_list<T> lst4 {lst2}; // ERROR
```

initializer_list<> Operations (2/2)


```
std::initializer_list<T> lst {a, b, c, ...};

lst.size(); // number of elements in list lst

// member function returns pointer to 1st element in lst
lst.begin();
// global function returns pointer to 1st element in lst
std::begin(lst);

// member function returns pointer to one past last element
lst.end();
// global function returns pointer to one past last element
std::end(lst);
```

Initializer Lists

- Subscript operator is not overloaded for `std::initializer_list<>`
- Can iterate thro' elements using range `for` statement
- See *initializer-list.cpp* …
- Microsoft has good reference page

<utility>

- In `<utility>` library, standard library provides a few "utility components" such as `std::pair<>` and `std::tuple<>`
- For more information, see this reference for `pair<>` here and this reference for `tuple<>`

Class `std::pair` (1/6)

- "Quick and dirty" data structure to combine two values into single value
- Useful because we don't need to define a structure to represent two values
- Used in many places in standard library [which we'll look at later]

Class `std::pair` (2/6)

- Class `std::pair` treats two values of arbitrary types as single unit

```
#include <utility>

int main() {
  // make a pair of C-string and double:
  std::pair<char const*, double> p1{"pi", 3.14};
  std::cout << p1.first << ' ' << p1.second << '\n';
  std::cout << std::get<0>(p1) << ' ' << std::get<1>(p1) << '\n';

  // make a pair of a pair<char,int> and double ...
  using PCI = std::pair<char, int>;
  std::pair<PCI, double> p2{PCI{'a', 1}, 1.21};
  std::cout << std::get<0>(p2).first << ' '
            << p2.first.second << ' ' << p2.second << '\n';
}
```

Class `std::pair` (3/6)

- `<utility>` provides convenience function `std::make_pair` to make pairs from *values* without writing types explicitly
- See *pair-intro.cpp* ...

```
int main() {
  std::pair<int, double> p4 = std::make_pair(12, 12.123);
  std::cout << std::get<0>(p4) << ' ' << std::get<1>(p4) << '\n';

  int i{12};
  double d{12.123};
  std::pair<int, double> p5 = std::make_pair(i, d); // ok
}
```

Class `std::pair` (4/6)

- Usual way to return more than two values is to either use two "in/out" reference parameters or one "in/out" reference parameter and function return value

```
void divide_remainder(int dividend, int divisor, int& q, int& r) {
  q = dividend/divisor;
  r = dividend - divisor*q;
}

int main() {
  int q, r;
  divide_remainder(7, 3, q, r);
  std::cout << "quotient: "  << q << " | "
            << "remainder: " << r << "\n";
}
```

Class `std::pair` (5/6)

- Can rewrite function to return value of type `std::pair`: see *pair-divrem.cpp* …

```
#include <utility>

std::pair<int, int> divide_remainder(int dividend, int divisor) {
  int q = dividend/divisor, r = dividend-divisor*q;
  return std::pair<int, int>{q, r};
}

int main() {
  std::pair<int, int> result = divide_remainder(7, 3);
  std::cout << "quotient: "  << result.first << " | "
            << "remainder: " << result.second << "\n";
  std::cout << "quotient: "  << std::get<0>(result) << " | "
            << "remainder: " << std::get<1>(result) << "\n";
}
```

Class `std::pair` (6/6)

- Function to return both sum and average

```
#include <utility>

// return both sum and average as std::pair<int, double> value ...
std::pair<int, double> sum_avg(std::initializer_list<int> values) {
  if (!values.size()) {
    return std::make_pair(0, 0.0);
  }

  int sum {};
  for (int x : values) { sum += x; }
  double average = static_cast<double>(sum)/values.size();

  return std::pair<int, double>{sum, average};
}
```

Class `std::tuple`

- Another "quick and dirty" data structure to combine ~~two~~ multiple values into single value
- Useful when we want to combine ~~two~~ multiple pieces of data into single value without defining structure to represent these data
- We are not concerned with `std::tuple` this semester

C-Style Strings In C++

- String is array [sequence] of `char`s
- C-style string is an array of `char`s terminated by null character `'\0'`
- C++ inherits C-string functions from C and they're declared in `<cstring>`
- C++ standard library has type `std::string`
- Why `std::string`?

C-Strings: The Good

- Simple and basic entity: makes use of `char` type and array concept
- Lightweight: minimal memory requirements
- Low level: Can be easily manipulated and copied
- If you're C programmer, why learn anything else?

C-Strings: The Bad

- Not a first class data type: cannot write intuitive expressions similar to built-in types
- Low level 1: susceptible to hard to find memory and security bugs
- Low level 2: programmers must have knowledge of underlying representation

C++ Strings

- Programs dealing with text are simpler if they use C++ standard library type `std::string` because `std::string` is implemented as *abstract data type*
  - We don’t know nor care about implementation
  - Instead, we only care about interface [behavior]
  - And, there is very large interface!!!

std::strings: Include <string>

```
#include <string>

// now you can use objects of type std::string

#include <cstring>

// avoid including <cstring> so that you
// don’t use legacy C mechanisms unless
// you’re writing code for both C and C++ ...
```

std::strings: Definition

- Lots of constructors to initialize string objects

Constructor	**Examples**
`string name;`	`string s0;` or `string s0{};`
`string name(str-literal);` `string name(str-literal, n);`	`string s1 = {"Bart Simpson"};` `string s2 {"Bart Simpson", 4};`

`string name(cstr-variable);`
`string name = cstr-variable;`
`string name{cstr-variable};`
`string name(cstr-variable, n);`
	`char const *ps {"Hello World"};`
	`string s3(ps);`
	`string s3 = ps;`
	`string s3{ps};`
	`string s4(ps, 5); // "Hello"`

`string name(str-variable);`
`string name = str-variable;`
`string name{str-variable};`
	`string s5(s1);`
	`string s5 = s1;`
	`string s5{s1};`

`string name(str-variable, pos, n);`	`string s6(s1, 5, 3); // "Sim"`
`string name(n, ch);`	`string s7(5, '+'); // "+++++"`

C++ String Literals

- See here for more examples

`R"(...)"` notation represents *raw string literal* with everything between delimiters becoming part of string

```
std::string s1{"a b c"},
            s2{"a \"b\" c"},
            s3{ R"("a ""b """c)"};
std::cout << s1 << "\n";
std::cout << s2 << "\n";
std::cout << s3 << "\n";
```

```
a b c
a "b" c
"a ""b """c
```

output of above code fragment:

std::string Type Members

- All containers including `std::string` define several types

Type Aliases	Examples
`value_type`	Element type: `char`
`size_type`	Unsigned integral type big enough to hold size of largest possible container: `size_t`
`difference_type`	Signed integral type big enough to hold size distance between two iterators [pointers]
`iterator`	Type of iterator [pointer] to elements in container: `char*`
`const_iterator`	Iterator [pointer] type that can read but not change its elements: `char const*`
`reference`	Element's lvalue type: `char&`
`const_reference`	Element's `const` lvalue type: `char const&`

Basic std::string Operations (1/2)

Expression	Meaning
`os << s`	Writes `s` into output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns number of characters in `s`.
`s[n]`	Element's lvalue type: `char&`

Basic `std::string` Operations (2/2)

Expression	**Meaning**
`s1 + s2`	Returns a `string` that is concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	Evaluates `true` if `string`s `s1` and `s2` contain exactly same characters; otherwise returns `false`.
`s1 != s2`	Inverse of `s1 == s2`.
`<, <=, >, >=`	Comparisons are case-sensitive and use lexicographic [dictionary] ordering.

std::strings: Input and Output (1/2)

- Non-member overloads of `operator<<` and `operator>>` write and read string, respectively

```
// read an unknown number of strings
std::string word;
while (std::cin >> word) {
  std::cout << word << '\n';
}
```

std::strings: Input and Output (2/2)

- Non-member function getline reads line of text

```
// read unknown number of lines separated
// by '\n' until end-of-file
std::string line;
while (std::getline (std::cin, line)) {
  std::cout << line << '\n';
}
```

```
// read unknown number of tokens separated
// by ':' until end-of-file
std::string token;
while (std::getline (std::cin, token, ':')) {
  std::cout << token << '\n';
}
```

std::strings: Assignment and Concatenation

- std::string overloads =, += and + operators for character, C-string, std::string arguments

```
std::string s1{"Lisa"};
s1 += " ";
s1 += "Simpson"; // "Lisa Simpson"
std::cout << s1 << '\n';
std::string s2{"is a fictional character"};
s1 = s1 + ' ' + s2;
std::cout << s1 << '\n';
```

std::strings: Size and Capacity (1/2)

- std::string provides variety of member functions to query number of characters
  - size and length are equivalent and return current number of characters
  - empty is shortcut to check whether number of characters is zero
  - max_size returns maximum number of characters string can contain
  - capacity returns number of characters string may contain without reallocating memory

```
std::string s{"Lisa"};
std::cout << s.empty()    << '\n';
std::cout << s.length()   << '\n';
std::cout << s.size()     << '\n';
std::cout << s.max_size() << '\n';
std::cout << s.capacity() << '\n';
```

output of code fragment:

```
0
4
4
15
4611686018427387903
```

std::strings: Size and Capacity (2/2)

- Since reallocation takes time, member function `reserve` is provided to avoid reallocations!!!

```
std::ifstream ifs{"file.txt"};
std::string line;
// process line-by-line the text in file ...
while (std::getline(ifs, line)) {
  // process line of text ...
}
```

```
// more efficient at run-time ...
std::ifstream ifs{"file.txt"};
std::string line;
line.reserve(1024);
// process line-by-line the text in file ...
while (std::getline(ifs, line)) {
  // process line of text ...
}
```

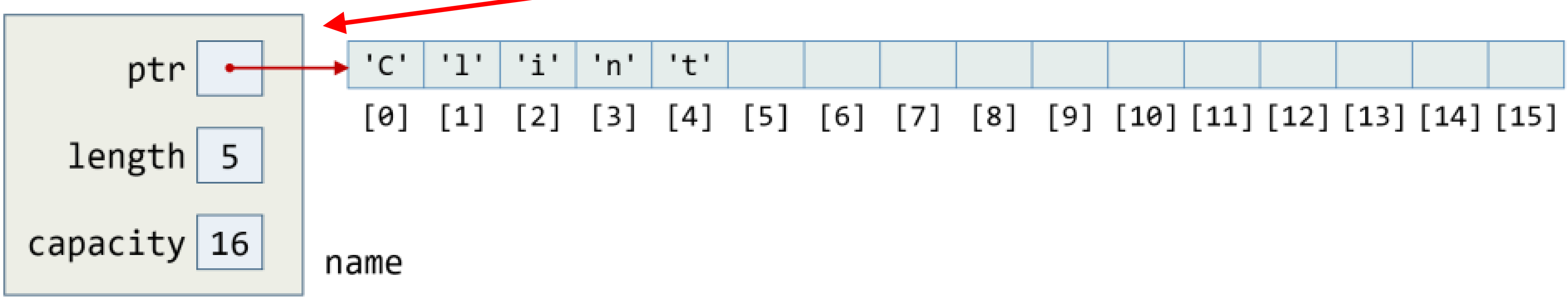
`std::string`s: Possible Memory Representation

- Possible definition of class `std::string` ...

```
class string {
private:
  char *ptr;
  size_t length;
  size_t capacity;
public:
  // rich interface ...
};
```

```
std::string name{"Clint"};
```

name's memory representation:



std::strings: Indexing (1/2)

- std::string overloads subscript operator [no runtime index check] and member function at [throws exception when argument is out-of-range]

```
std::string s{"abcdef"};

std::cout << s[2] << '\n'; // 'c'
std::cout << s[s.length()-1] << '\n'; // 'f'
std::cout << s[s.length()*2] << '\n'; // undefined behavior

std::cout << s.at(3) << '\n'; // 'd'
// std::out_of_range exception thrown
std::cout << s.at(s.length()*2) << '\n';
```

std::strings: Indexing (2/2)

- Indexing not recommended when entire std::string is accessed!!!

```
int vowels(std::string const& s) {
  int count{};
  for (std::string::size_type i{}; i < s.length(); ++i) {
    count = (s[i]=='a' || s[i]=='e' || s[i]=='i'
             || s[i]=='o' || s[i]=='u') ? count+1 : count;
  }
  return count;
}
```

```
std::string capitalize(std::string s) {
  for (std::string::size_type i{}; i < s.length(); ++i) {
    s[i] = (s[i] >= 'a' && s[i] <= 'z') ? s[i]-'a'+'A' : s[i];
  }
  return s;
}
```

std::strings: Ranging

- Use range `for` statement to access all elements!!!

```
int vowels(std::string const& s) {
  int count{};
  for (char ch : s) {
    count = (ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u')
            ? count+1 : count;
  }
  return count;
}
```

```
std::string capitalize(std::string s) {
  for (char& ch : s) {
    ch = (ch >= 'a' && ch <= 'z') ? ch-'a'+'A' : ch;
  }
  return s;
}
```

std::strings: More Element Access Methods


```
std::string s{"abcde"};
s.front();         // evaluates to s[0]
s.front() = 'A'; // char& to s[0]
s.back() ;         // evaluates to s[4]
s.back() = 'Z';  // char& to s[4]

s = "";   // assign the empty string
s.front(); // ERROR: undefined behavior
s.back();  // ERROR: undefined behavior
s.front() = 'Z'; // ERROR
s.back()  = 'Z'; // ERROR
```

`std::string`s: Comparisons (1/4)

- `std::string` overloads entire gamut of comparison and relational operators
- Equality operators `==` and `!=` test whether two `string`s are equal or unequal, respectively
  - Two `string`s are equal if they're same length and contain same characters

```
std::string s1{"hello"}, s2{s1};
s1 == s2 // evaluates true
s1 != s2 // evaluates false
```

std::strings: Comparisons (2/4)

- Overloaded operators <, <=, >, and >= compare strings *lexicographically*:
  - If two strings have different lengths and if every characters in shorter string is equal to corresponding character of longer string, then shorter string is less than longer string
  - If any characters at corresponding position in two strings differ, then result of string comparison is result of comparing first character at which strings differ

std::strings: Comparisons (3/4)

- Overloaded operators <, <=, >, and >= compare strings *lexicographically*

```
std::string("aaaa") == std::string("bbbb") // false
"aaaa" > std::string("bbbb") // false
std::string("aaa") < "bbbb"  // true
```

std::strings: Comparisons (4/4)

- Substrings can be compared using member function `compare`

```
std::string s{"abcd"}, s2{"dbca"};

s.compare("abcd") // returns 0
s.compare(s2)      // returns value < 0 [s is less]
s.compare("ab")    // returns value > 0 [s is greater]
s.compare(s)       // returns 0 [s is equal to s]
s.compare(0,2,s,2,2)   // ["ab" is less than "cd"]
s.compare(1,2,"bcx",2) // ["bc" equal to "bc"]
```

std::string: Substrings

- Use member function substr to extract substring

```
std::string s {"interchangeability"};

s.substr()                 // returns s
s.substr(11)               // returns string("ability")
s.substr(5, 6)             // returns string("change")
s.substr(s.find('c')) // returns string("changeability")
```

`std::string`s: Searching & Finding (1/3)

- Using member functions of `std::string`s, you can search
  - For a single character, character sequence [substring], or one of a set of characters
  - Forward and backward
  - By starting from any position at beginning or inside string

std::strings: Searching & Finding (2/3)

Search operations return index of desired character or npos if not found	
`s.find(args)`	Find 1st occurrence of *args* in s
`s.rfind(args)`	Find last occurrence of *args* in s
`s.find_first_of(args)`	Find 1st occurrence of any character from *args* in s
`s.find_last_of(args)`	Find last occurrence of any character from *args* in s
`s.find_first_not_of(args)`	Find 1st character in s not in *args*
`s.find_last_not_of(args)`	Find last character in s not in *args*

args must be one of	
`c, pos`	Look for character c starting at position pos in s. pos defaults to 0.
`s2, pos`	Look for string s2 starting at position pos in s. pos defaults to 0.
`cp, pos`	Look for C-string cp starting at position pos in s. pos defaults to 0.
`cp, pos, n`	Look for 1st n characters in C-string cp starting at position pos in s. No default for pos or n.

std::strings: Searching & Finding (3/3)


```
std::string s {"Mississippi"};

s.find("si")    // returns 3 [1st substring "si"]
s.find("si", 4) // returns 6 [1st substring "si" starting from s[4]]
s.rfind("si")   // returns 6 [last substring "si"]
s.find_first_of("si") // returns 6 [1st char 's' or 'i']
s.find_last_of("si")  // returns 10 [last char 's' or 'i']
s.find_first_not_of("si") // returns 0 [1st char neither 's' nor 'i']
s.find_last_not_of("si") // returns 9 [last char neither 's' nor 'i']
s.find("sssi")              // returns string::npos
```

std::string: Value npos (1/2)

- If search function fails, it returns `std::string::npos`

```
std::string s {"Mississippi"};
std::string::size_type idx = s.find("sss");
if (idx == std::string::npos) {
  std::cout << "\"ssi\" not found!!!\n";
}
```

std::string: Value npos (2/2)

- Be very careful when using string::npos!!!
  - Library uses type size_t [largest possible unsigned type] as size_type
  - Library uses maximum value of its type for value of npos [which is -1]
  - -1 in unsigned long is not equivalent to -1 in unsigned int!!!

```
std::string s{"Mississippi"};
unsigned int idx = s.find("sss");
// -1 in unsigned int is not same as -1 in size_t!!!
if (idx == std::string::npos) {
  std::cout << "ssi not found !!!\n";
}
```

std::strings: Inserting Characters (1/3)

- Several possibilities for removing all characters in string:

```
std::string s {"Mississippi"};

s = "";     // assign the empty string
s.clear(); // clear contents
s.erase(); // erase all characters
```

std::strings: Inserting Characters (2/3)

- Can use operator+=, append, and push_back member function to append characters:

```
std::string s, s2, s3{"Bart"};

s += s3;     // append "Bart"
s += " Sim"; // append C-string
s += 'p';                 // append single character
s += {'s', 'o', 'n'}; // append initializer_list<char>

s2.append(s3);        // append "Bart"
s2.append(s, 4, 4); // append " Sim"
s2.append(s, 8, std::string::npos); // append "pson"
s2.push_back(' '); // append ' '
s2.append(3, 'z'); // append three characters: 'z' 'z' 'z'
```

std::strings: Inserting Characters (3/3)

- Can use `insert` member functions to insert characters
  - These functions require index of character in string after which new characters are inserted

```
std::string const cs{"Sim"};
std::string s{"Bart"};

s.insert(s.size(), "-");    // s: Bart-
s.insert(s.size(), cs);     // s: Bart-Sim
s.insert(s.size(), "pson"); // s: Bart-Simpson
```

std::strings: Removing Characters

- Can use member functions erase and pop_back to remove characters and replace to replace characters

```
std::string s{"i18n"}; // s: i18n

// s: internationalization
s.replace(1, 2, "nternationalization");
s.erase(13); // s: international
s.erase(7, 5); // s: internal
s.pop_back();  // s: interna
s.replace(0, 2, "ex"); // s: externa
```

`std::string`s: Numeric Conversions (1/2)

- Non-member convenience functions exist to convert `std::string`s into numeric values or to convert numeric values to `std::string`s which:
  - Skip leading whitespaces
  - Return index of first character after last processed character
  - Allow specifying number base to use for integral values
  - Might throw `std::invalid_argument` if no conversion is possible or `std::out_of_range` if converted value is outside range of values for return value

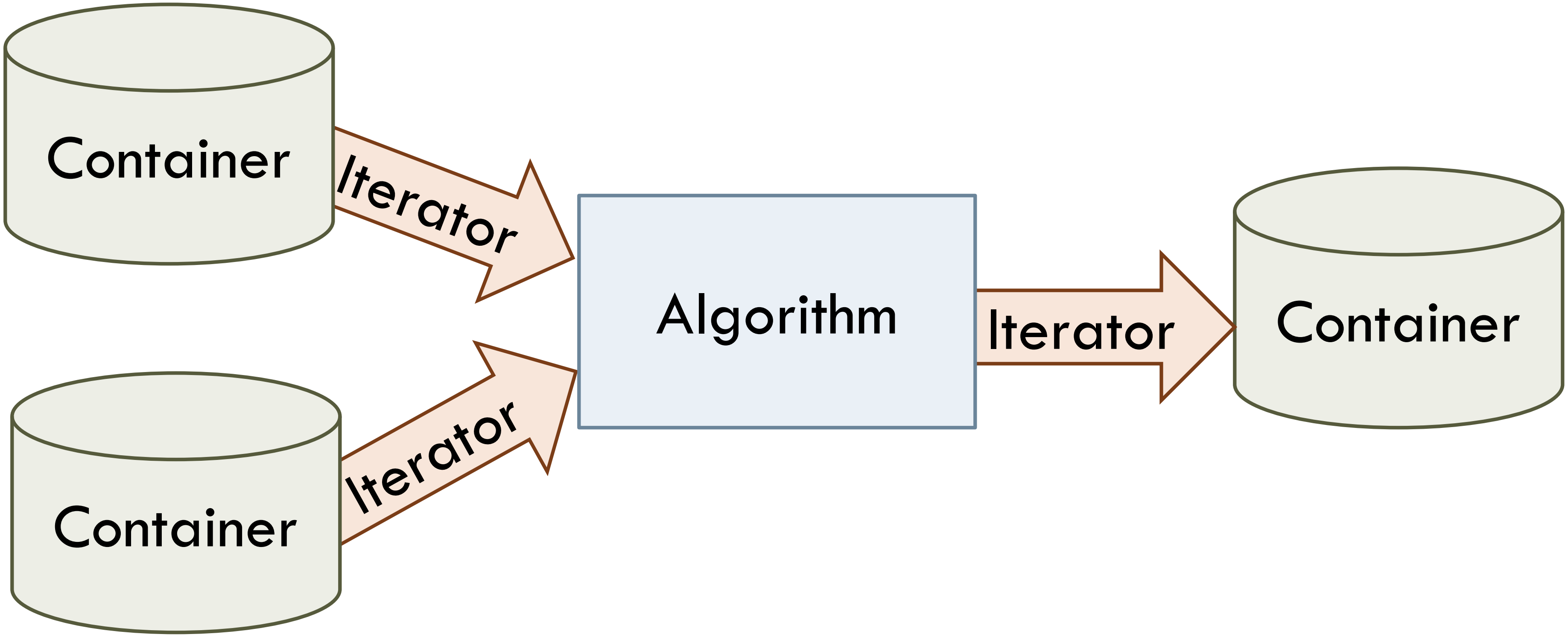
std::strings: Numeric Conversions (2/2)

Function	Effect
`stoi(str, idx=nullptr, base=10)`	Converts *str* to an int
`stol(str, idx=nullptr, base=10)`	Converts *str* to a long
`stoul(str, idx=nullptr, base=10)`	Converts *str* to an unsigned long
`stoll(str, idx=nullptr, base=10)`	Converts *str* to a long long
`stoull(str, idx=nullptr, base=10)`	Converts *str* to an unsigned long long
`stof(str, idx=nullptr)`	Converts *str* to a float
`stod(str, idx=nullptr)`	Converts *str* to a double
`stold(str, idx=nullptr)`	Converts *str* to a long double
`to_string(val)`	Converts *val* to a std::string

More Examples

- See code and handout on strings for more examples …

Introduction to Standard Template Library Through `std::vector<T>`



Handout

- See handout on vectors for more examples …