HIGH-LEVEL PROGRAMMING 2

Useful C++ ADTs by Prasanna Ghali

Abstraction and Encapsulation

- Abstraction: technique for reducing complexity that identifies which specific information should be visible [the *interface*] and which should be hidden [the *implementation*]
- Encapsulation: packaging technique 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 This Week

- 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++


```cpp
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

- **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

- 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 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>

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

```
#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);
}
```

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


```
// 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


```
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

- Can iterate thro' elements using range-`for` statement
- See *initializer-list.cpp* …

`<utility>`

- In `<utility>`, standard library provides a few “utility components” such as `std::pair` and `std::tuple`

Class `std::pair`

- 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, float> p2{PCI{'a', 1}, 1.21f};
  std::cout << std::get<0>(p2).first << ' '
            << p2.first.second << ' ' << p2.second << '\n';
}
```

Class `std::pair`

- `<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};
  // ERROR: i and d are lvalues!!!
  std::pair<int, double> p5 = std::make_pair(i, d); // ERROR
}
```

Class `std::pair`

- “Quick and dirty” data structure to combine two values into single value
- Used in many places in standard library [which we’ll look at later]
- Useful when we want to combine two pieces of data into single value but don’t want to both to define a structure to represent these data
- We can use `std::pair` to return two values from function

Class `std::pair`

- 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`

- Can rewrite function to return value of type `std::pair`: see *div-rem.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`

- Function to return both sum and average: see *sum-avg.cpp* ...

```
#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 but don't want to both to define a structure to represent these data
- We are not concerned with [`std::tuple`] this semester

`using` Keyword [1st Use]

- `using` *declaration* makes specific names declared in a namespace accessible without requiring namespace and `::` operator
- Hides previous meaning that name had *in outer scope*!!!

```
#include <iostream>

int cout {1};

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

`using` Keyword [2nd Use]

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

`using` Keyword [2nd Use]

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

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

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

```
// this is your source ...
using namespace Graphics;
using namespace AI;

// suppose you're authoring function baz
void baz() {
  // some other functions are called ...

  bar();      // do some AI stuff ...
  foo(10.1); // do some graphics stuff ...

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

using Keyword [2nd Use]

- 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 ...
using namespace Graphics;
using namespace AI;

// suppose you're authoring function baz
void baz() {
  // some other functions are called ...

  bar();      // do some AI stuff ...
  foo(10.1); // which foo?

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

`using` Keyword [2nd Use]

- `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]

- `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]

```
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]

- For C++ language technical reasons, always use `using` keyword to define type aliases
- Forget about keyword `typedef` in C++
- Use keyword typedef only in C code and for maintaining legacy code …

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` that provides much improved implementation of concept of string
- Why `std::string`?

C-Strings: The Good

- Simple and basic entity: makes use of `char` type and array structure
- 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++ class 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]

C++ 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 ...
```

C++ 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++ Strings: String Literals

- See here for more examples

```
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";
```

output of above code fragment:

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

C++ Strings: Input and Output

- `std::string` implements non-member overloads of `operator<<`, `operator>>`, and function `getline` to read line of text

C++ Strings: Assignment and Concatenation

- `std::string` overloads =, += and + operators

```
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';
```

C++ Strings: Comparisons

- `std::string` overloads entire gamut of comparison and relational operators

C++ Strings: Size and Capacity

- `std::string` provides variety of ways to query number of characters ...

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

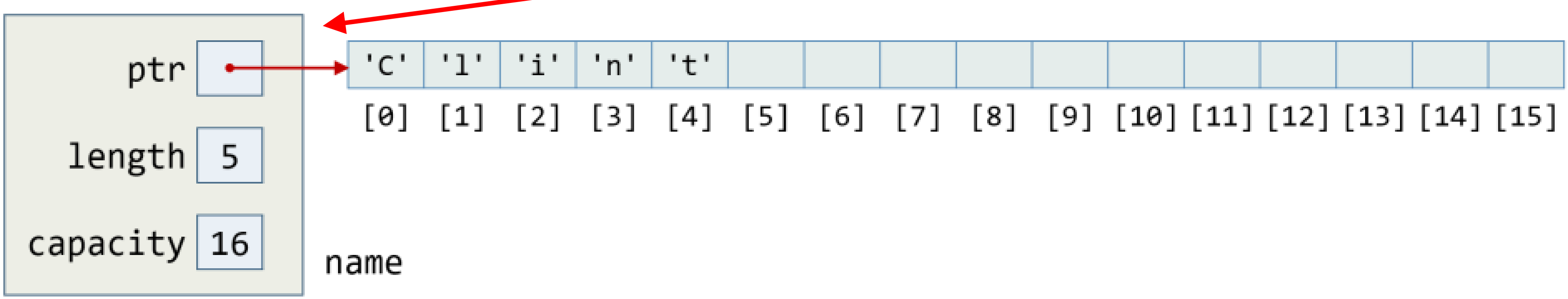
C++ Strings: Possible Memory Representation

- 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:



C++ Strings: Indexing

- 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';
```

C++ Strings: Indexing

- Indexing not recommended!!!

```
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;
}
```

C++ Strings: Ranging

- Instead, use range-`for` statement!!!

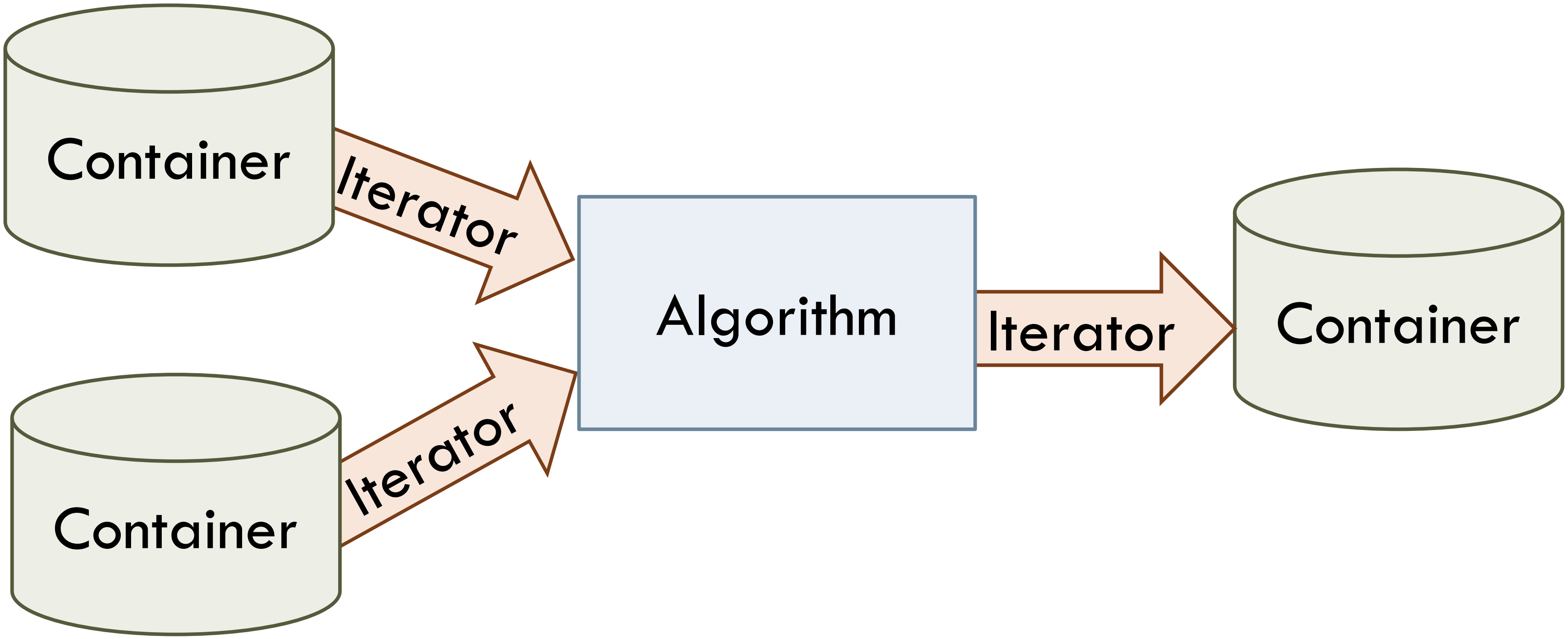
```
int vowels(std::string const& s) {
  int count{};
  for (char ch : s) {
    count = (ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u')
              ? count+1 : count;
  }
  return count;
}
```

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

Handout

- See handout on strings for more examples …

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



Handout

- See handout on vectors for more examples …