

CS11 Intro C++

FALL 2015-2016

LECTURE 6

C++ Strings

- ▶ C++ retains C notion of `char*` as a “string”
 - ▶ Array of `char` values, terminated with a 0 value (a.k.a. “the null character” or “NUL”)
- ▶ Useful C functions for string manipulation in `<cstring>` header (C++ name for `string.h`)
- ▶ Typically difficult / bug-prone to manipulate in complex ways
 - ▶ Have to manually allocate and reallocate space to hold string data
 - ▶ Can easily write past end of string (buffer overflows, exploits!)
 - ▶ Can easily forget to free memory used by C strings

C++ Strings (2)

- ▶ C++ also introduces a new `std::string` type
 - ▶ Resizable string that keeps data in heap memory
 - ▶ `#include <string>`
- ▶ Provides many features over `char*` strings
 - ▶ Ability to insert/delete/replace/extract substrings easily, without manual memory management
 - ▶ Can access, mutate, and traverse characters using array access, or using iterators (!!)
 - ▶ Can convert to and from `char*` strings easily
 - ▶ Supports stream IO with `>>` and `<<` operators
- ▶ Prefer `string` to `char*`, wherever possible!

C++ String Initialization

- ▶ C++ **string** objects can be initialized from other strings, or from **char*** values

```
string s1 = "green";    // Same as s1("green");  
string s2 = s1;          // Same as s2(s1);
```

- ▶ **s2** is an independent copy of **s1**

- ▶ Can also initialize to be a repeated character

```
string reps(5, 'a');    // reps == "aaaaa"
```

- ▶ Can also initialize to a substring

```
string s3(s1, 2, 2);    // s3 == "ee"
```

- ▶ First number is the position (zero-based)

- ▶ Second number is count

- ▶ Other constructor options too...

C++ String Assignment

- ▶ `string` supports assignment operator

```
string s1 = "orange";  
string s2 = "yellow";  
s2 = s1;  
s1 = "gray";
```

- ▶ Can also use `assign()` member-function

```
s2.assign(s1);  
s1.assign("gray");
```

- ▶ Strings do not share underlying storage

- ▶ Assignment makes a copy of what is assigned

String Comparison and Concatenation

- ▶ `string` supports comparison operators

- ▶ `== != < >` etc.
 - ▶ Case sensitive by default
 - ▶ Depends on locale!

- ▶ Use `+` or `+=` for concatenation

```
string title = "purple";
title = title + " people";
title += " eater";
```

- ▶ Can also append individual characters

```
title += 's';
```

- ▶ Can also use `append()` member-function

String Lengths and Indexes

- ▶ `length()` member-function reports number of characters in string

```
string color = "chartreuse";
cout << color << " has " << color.length()
    << " characters." << endl;
```

- ▶ `string` also has a `size()` member-function that does the same thing as `length()`
- ▶ Characters have indexes 0 to `length() - 1`
- ▶ `string::npos` indicates “invalid index”
- ▶ All strings have `length() < string::npos`

Individual Characters

- ▶ Individual character access with []

```
string word = "far";  
word[1] = 'o'; // now word == "for"
```

- ▶ Indexes aren't checked for validity
 - ▶ (Not a big issue, but just be aware of it)
- ▶ **string** also provides iterators for traversing characters (!!)
 - ▶ “Strings are collections of characters.”
 - ▶ **begin()** returns an iterator pointing to first char
 - ▶ **end()** returns an iterator pointing past last char
 - ▶ Can pass **string** iterators to STL algorithms

Finding Substrings

- ▶ Four versions of `find()` member function

```
size_type find(const string &, size_type start=0)
```

```
size_type find(const char *, size_type start,  
               size_type length)
```

```
size_type find(const char *, size_type start=0)
```

```
size_type find(char, size_type start=0)
```

- ▶ Returns index of match, or `string::npos` for no match

- ▶ The `rfind()` member function searches backwards through a `string`

- ▶ Provides same four versions, with appropriate default values for arguments

Variants of `find()`

- ▶ `find_first_of()`, `find_last_of()`
 - ▶ Unlike `find()`, matches if any character in argument appears in string
 - ▶ `find_first_of()` starts at beginning, goes forward
 - ▶ `find_last_of()` starts at end and goes backwards
- ▶ `find_first_not_of()`, `find_last_not_of()`
 - ▶ Finds first character not in argument
 - ▶ Again, can search from beginning, or from end

String Manipulation

- ▶ `substr()` extracts a substring
 - `substr(size_type start = 0,
size_type length = npos)`
 - ▶ Returns a new `string` object containing the substring
 - ▶ Note that default arguments just copy entire string
- ▶ `replace()` modifies a substring
 - ▶ Again, many versions of `replace()`
 - ▶ Some take iterators; some take start+length arguments
- ▶ `erase()` removes a substring
 - ▶ “Replace with nothing”
 - ▶ Can call with start+length; or with iterators

More String Manipulation!

- ▶ `append()` member functions allow for appending characters or strings
 - ▶ Can append a C++ `string`, a C `char*` string, or an individual `char` value
 - ▶ (Or you can use `+=` operator, as discussed earlier!)
- ▶ `insert()` member functions for inserting characters into a string
 - ▶ Can insert at a specific index
 - ▶ Can use an iterator to indicate location

Converting C++ Strings to char*

- ▶ Can convert strings to `char*` values
 - ▶ `c_str()` and `data()` both return a zero-terminated `char*`
 - ▶ (Before C++11, `data()` returned a non-terminated `char*` value, which wasn't generally very useful)

- ▶ Example:

```
string value = "orange";
printf("%s\n", value.c_str());
```

- ▶ Also, the `copy(...)` member function copies a string into a `char*` buffer the caller provides
 - ▶ Useful when the caller has their own buffer to use

c_str() Gotchas!

- ▶ Don't cache pointers returned by `c_str()` or `data()` member functions
 - ▶ May not have valid data after a non-`const` call to `string` member function
- ▶ Example:

```
string s = "hello";
char *p = s.c_str();
s += " world!"; // p no longer trustworthy
printf("%s\n", p);
```

- ▶ Last line may print garbage, or it may crash, or it may work fine! **Whatever it does, it's a bug!**

c_str() Gotchas! (2)

- ▶ Don't return `c_str()` or `data()` value from a `string` local variable!
 - ▶ Memory is managed by the `string` object
 - ▶ It goes away when the `string` variable goes out of scope!
- ▶ A (bad) example:

```
char * getUserName() {  
    string name;  
    cout << "Enter username: ";  
    cin >> name;  
    return name.c_str(); // BAD!  
}
```

Shape Classes

- ▶ Want to write classes to represent 2D shapes
 - ▶ Circle: radius
 - ▶ Triangle: base + height
 - ▶ Rectangle: width + height
- ▶ All these shapes have common features, but also special characteristics
- ▶ Common themes:
 - ▶ All shapes have an area
- ▶ Differences:
 - ▶ Shapes have different properties to describe them
 - ▶ Each shape has a different way to compute area

Shape Classes (2)

- ▶ One approach: one class to represent all kinds of shapes we want to support

```
class Shape {  
    string type; // what kind of shape is it?  
    float radius; // For circles only!  
    float height; // For triangles, rectangles  
    float base; // For triangles only!  
    float width; // For rectangles only!  
public:  
    Shape(const string &t); // Specify type  
    string getType() const;  
    float getArea() const;  
};
```

Shape Classes (3)

- ▶ **This approach has several drawbacks.**
- ▶ Not a clean abstraction!
 - ▶ Both data and operations for different shapes are mixed together
- ▶ Hard to understand what is going on
 - ▶ Very likely to lead to bugs
- ▶ Also wastes space
 - ▶ For each type of shape, some data members won't be used at all

C++ Class Inheritance

- ▶ C++ provides **class inheritance**
 - ▶ (as do most OOP languages)
- ▶ Base class provides generalized capabilities
 - ▶ A generic “shape” type specifying common features
- ▶ Derived classes inherit state/behavior of base class
 - ▶ All kinds of shapes provide the same general set of operations, e.g. `float getArea() const;`
- ▶ Derived classes also provide specialized capabilities
 - ▶ Each kind of shape can specify what fields it needs
 - ▶ Each kind of shape specifies how to compute its area

C++ Class Inheritance (2)

20

- ▶ Class inheritance models an “is-a” relationship
 - ▶ A circle “is a” shape
 - ▶ A rectangle “is a” shape
 - ▶ A triangle “is a” shape
- ▶ Terminology:
 - ▶ Base class, parent class, superclass
 - ▶ Derived class, child class, subclass

Shape Base-Class

```
class Shape {  
public:  
    enum class Type {  
        CIRCLE, RECTANGLE, TRIANGLE };  
private:  
    Type type;  
public:  
    Shape(Type t);  
    Shape::Type getType() const;  
    float getArea() const;  
};
```

- ▶ All shapes will have these common traits

Circle Subclass

```
class Circle : public Shape {  
    float radius;  
public:  
    Circle(float r);  
    float getArea() const;  
};
```

- ▶ `Circle` class inherits “shape type” member, and `getType()` / `getArea()` accessors, from `Shape`
- ▶ Can also specify that circles need a radius, and how to compute the area of a circle

Rectangle Subclass

```
class Rectangle : public Shape {  
    float width;  
    float height;  
public:  
    Rectangle(float w, float h);  
    float getArea() const;  
};
```

- ▶ Similarly, specify that rectangles need a width and a height

Initialization Details

- ▶ When a subclass of **Shape** is initialized, the **Shape** constructor must also be invoked
 - ▶ Reason: subclass constructor body might access or call superclass members
- ▶ If the parent class has a default constructor, this initialization will happen automatically
 - ▶ e.g. when a **Circle** is initialized, it would automatically call the **Shape** default constructor
- ▶ But, **Shape** doesn't have a default constructor
`Shape(Type t);`

Initialization Details (2)

- ▶ `Shape` doesn't have a default constructor

```
shape(Type t);
```

- ▶ `Circle` and `Rectangle` must use a **base-class initializer** to call the `Shape` constructor

```
circle::circle(float r) : shape(Type::CIRCLE) {  
    radius = r;  
}
```

- ▶ `circle` constructor specifies that `Shape` constructor should be called first, with `Type::CIRCLE` as the argument

- ▶ Once base-class initialization is complete, `circle` constructor can continue its work

Initialization Details (3)

- ▶ This is only required if the parent class doesn't have a default constructor

```
circle::circle(float r) : shape(Type::CIRCLE) {  
    radius = r;  
}
```

- ▶ If parent class has a default constructor, it will be called automatically when subclass is initialized
- ▶ (Also, subclass may simply not want to use the parent class' default constructor...)

Shape Areas

- ▶ Shape base-class doesn't have an area to compute

```
float Shape::getArea() const {  
    return 0; // do this for now...  
}
```

- ▶ Subclasses can override the parent-class version

```
float Circle::getArea() const {  
    return 3.14159 * radius * radius;  
}  
  
float Rectangle::getArea() const {  
    return width * height;  
}
```

Shape Areas (2)

- ▶ Example:

```
Rectangle r(8, 7);  
Circle c(5);
```

```
cout << "Rectangle area = "  
     << r.getArea() << endl;  
cout << "Circle area = "  
     << c.getArea() << endl;
```

- ▶ Prints:

```
Rectangle area = 56  
Circle area = 78.53...
```

Shape Areas (3)

- ▶ Add a helper function to print shape areas

```
void printArea(Shape &s) {  
    cout << "Shape area = "  
        << s.getArea() << endl;  
}  
  
Rectangle r(8, 7);  
Circle c(5);  
printArea(r);  
printArea(c);
```

- ▶ Prints:

```
Rectangle area = 0  
Circle area = 0
```

Which getArea()?

- ▶ By default, C++ uses the variable's type to figure out which function to call

```
void printArea(Shape &s) {  
    cout << "Shape area = "  
        << s.getArea() << endl;  
}
```

- ▶ The `getArea()` function is being called on a variable of type `Shape&...`
 - ▶ ...so the compiler calls `Shape::getArea()`, which returns 0
 - ▶ ...even when `s` actually refers to a subclass of `Shape`
- ▶ This is called **static dispatch**, since the compile-time type of the variable (aka the “static type”) is used

The `virtual` Keyword

- ▶ The `virtual` keyword tells C++ to choose the function to call, based on the type of the object that is referenced, rather than the variable's type
 - ▶ This is called **dynamic dispatch**, because it uses the “dynamic type” or the “runtime type” of the value
- ▶ In `Shape` class declaration:
`virtual float getArea() const;`
- ▶ Don't need to restate `virtual` on subclass version of function – the “virtualness” is inherited
- ▶ Why not just specify `virtual` on all parent-class functions?

How `virtual` Works

- ▶ When a member function is marked `virtual`, C++ stores extra information in the object
 - ▶ **The object itself** knows which version of the function to use
- ▶ Imposes a [typically very] small size overhead
- ▶ Calling virtual functions takes [very] slightly longer

type	CIRCLE
getArea	Circle::getArea
radius	5

type	RECTANGLE
getArea	Rectangle::getArea
width	8
height	7

How `virtual` Works (2)

- ▶ You should only mark member functions `virtual` when you intend them to be overridden
- ▶ Corollary: if a function isn't marked `virtual`, don't override it!

type	CIRCLE
getArea	Circle::getArea
radius	5

type	RECTANGLE
getArea	Rectangle::getArea
width	8
height	7

Class Hierarchies and Destructors

- ▶ Final example:

```
Shape *s1 = new Rectangle(8, 7);  
Shape *s2 = new Circle(5);  
delete s1;  
delete s2;
```

- ▶ Which destructor will this code call?

- ▶ Both lines will invoke the **Shape** destructor ☹

- ▶ The behavior of this code is undefined

- ▶ Typically results in a memory leak, or possibly a crash

Class Hierarchies and Destructors (2)

- ▶ Anytime you create a class hierarchy, you must make sure the base-class destructor is **virtual**

```
class Shape {  
    ...  
    virtual ~Shape() { }  
};
```

- ▶ If destructor doesn't need to do anything, still give it an empty body so the linker doesn't complain
- ▶ When you intend a class to be subclassed, give it a virtual destructor
- ▶ Corollary: if a class should not be subclassed, do not give it a virtual destructor!

Shape::getArea()

- ▶ This didn't really make sense:

```
float Shape::getArea() {  
    return 0;  
}
```

- ▶ “Shape” is too general a concept for us to define this function
 - ▶ We know that all shapes have an area...
 - ▶ ...but we need a specific kind of shape before we can actually compute the area!

Pure-Virtual Functions

- ▶ Can declare functions to be **pure-virtual**
- ▶ The function is only declared in the base-class, but it must be defined in subclasses

```
class Shape {  
    ...  
    virtual float getArea() const = 0;  
};
```

- ▶ The “= 0” part makes it pure-virtual
- ▶ Now, the **Shape** class cannot be instantiated
 - ▶ Part of its implementation is now missing!
 - ▶ Compiler will report an error

Abstract Classes

- ▶ Shape is now called an **abstract class**
 - ▶ It declares behaviors, but doesn't define all of them
- ▶ Can still create variables and arguments of type **Shape*** or **Shape&**
 - ▶ Variable can reference any object that provides everything that **Shape** declares...
- ▶ Allows us to create generic functions that can take any kind of Shape

```
Shape * findLargestArea(  
    const vector<Shape *> &shapes);
```

- ▶ Can pass in a vector of pointers to triangles, rectangles, and spheres

Object-Oriented Programming Concepts

- ▶ So far, we have seen:
 - ▶ **Encapsulation:** hiding and guarding internal state
 - ▶ **Abstraction:** presenting a simple, high-level interface
- ▶ Class hierarchies introduce two more:
 - ▶ **Inheritance:** a child class gets its parent class' members/behavior
 - ▶ ...and a child-class can customize parent-class behavior
 - ▶ **Polymorphism:** calling a function on an object can exhibit different behaviors, based on object's type
 - ▶ In C++, this requires the use of virtual functions