C++ is Evolving

C++ is an evolving language. A committee of the ISO (International Organization for Standards) ratifies proposed changes to C++. New standards have been released every few years. In this handout we give a brief introduction to some additional features that were added to the C++11 standard. The topics included here serve as an introduction for more advanced topics in computer programming and computer science. Consult a more advanced textbook or the ISO C++ Standard online at https://isocpp.org/ if you wish to dive deeper into these topics.

std::array

The standard container ***array*** is included in the `<array>` library and allows you to use a vector-like notation for random access into a fixed-size sequence of elements. Essentially, the container allows you to safely access array elements like a vector but with the performance and minimal storage requirements of a regular array.

The following code shows how to create an array of six integers while initializing the first three elements. The remaining three elements are automatically initialized to zero, so we don't have the problem of unknown uninitialized values like we do with normal arrays.

// The `std::array`

```
#include <iostream>
#include <array>

using std::cout;
using std::endl;
using std::array;

int main()
{
	// The array is allocated to hold six integers.
	// The first three are set to 10, 20, and 30 while
	// the remainder are set to 0.
	array<int,6> a = {10, 20, 30};

	cout << "The size of the array: " << a.size() << endl;
	cout << "The element at index 1: " << a[1] << endl;
	cout << "Setting a[4] to 100" << endl;
	a[4] = 100;
	cout << "Outputting all elements of the array: " << endl;
	for (int element : a)
		cout << " " << element << endl;
}
```

Sample Dialogue

```
The size of the array: 6
The element at index 1: 20
Setting a[4] to 100.
Array contains:
```

```
10
20
30
0
100
0
```

Just like a vector but unlike a standard array, we can retrieve the size of the array using the `size()` function. We can also read and set the contents of the array using the traditional `[]` notation. Attempts to read a value out of range returns 0 and attempts to set a value out of range has no effect. Note that indices 3 and 5 in the array get set to the default value of 0.

We can use the same techniques that are available to vectors. For example, if we include `<algorithm>` then we can sort the array within the ranges specified by the iterators `begin()` and `end()`.

```
std::sort(a.begin(), a.end());
cout << "After sort, array contains: " << endl;
for (int element : a)
   cout << element << endl;
```

The output is the array in sorted order:

```
After sort, array contains:
0
0
10
20
30
100
```

Regular Expressions

A full treatise of regular expressions is beyond the scope of this lecture, but a summary of regular expressions and some examples in C++ are described here.

Note that some compilers do not support the C++11 regular expression library so check your compiler to see if the `<regex>` library is supported. For this class, Visual Studio DOES support the regex library. The installed version of g++ on must be 4.9 or higher for regex support. The installed version on uaa-transformer.duckdns.org is 4.9.4. Upgrading is non-trivial on some machines due to an ecosystem of programs that depend on versions of libraries in which new library versions break old versions of other programs. This makes it difficult to have old and new versions of software side-by-side. To get around this problem, the SCL (Software Collections Library) allows enabling of a versioned set of tools.

For those familiar with regular expressions, the new C++11 standard supports the Javascript and POSIX formats.

Formally, a regular expression provides a way to describe a language from the class of regular languages. For our purposes we'll think of a regular expression as a way to describe a pattern that can be used to match a sequence of text. For example, we could use a regular expression to see if a string of text contains a date in the MM-DD-YYYY format. Without regular expressions we would have to write code

ourselves to process the text, which could be difficult for complicated patterns. A summary of basic regular expressions are below.

Regular Expression	**Meaning**
Letter or digit	The same letter or digit. For example, the regular expression `a` matches the text `a`, and the regular expression `abc123` matches the text `abc123`.
.	Matches any single character
|	Union or logical OR
R?	The regular expression `R` appears 0 or 1 time
R+	The regular expression `R` repeats consecutively 1 or more times
R*	The regular expression `R` repeats consecutively 0 or more times
R{n}	The regular expression `R` repeats consecutively n times
R{n,m}	The regular expression `R` repeats consecutively n to m times
^	Beginning of the text
$	End of the text
[list of elements]	Match any of the elements. For example, `[abcd]` would match `a`, `b`, `c`, or `d`.
[element1-elementN]	Match any of the elements in the range. For example, `[a-zA-Z]` would match any uppercase or lowercase letter.
()	Precedence and expression grouping

Here are examples of some simple regular expressions:

Description	**Regular Expression**
Three a's followed by three b's	`aaabbb` or `a{3}b{3}`
Any sequence of zero or more a's	`a*`
One or more a's followed by any sequence of b's	`a+b*`
The rules for an identifier, i.e., a letter or underscore followed by any sequence of letters, digits, or underscores	`[a-zA-Z_]+[a-zA-Z0-9_]*`

The C++11 regex library includes many useful character classes. Some of them are listed in the following table.

Regular Expression	**Meaning**
\d	A single digit
\D	A non-digit
\s	A whitespace character (e.g., tab, newline, space)
\w	A word character

We can utilize these classes to simplify our patterns. For example, if we would like to match two consecutive words then the regular expression of `\w+\s\w+` will match any sequence of 1 or more word characters, followed by a whitespace, followed by any sequence of 1 or more word characters.

To match regular expressions in C++11 include the `<regex>` library. The `regex` class is part of the `std` namespace and takes a pattern as input. The regex class has the functions `regex_match` to exactly match a pattern to a string, `regex_search` to look for occurrences of patterns in a string, and `regex_replace` to replace matches in the string with a format string.

The following code illustrates `regex_match` to determine if `text1` or `text2` matches the pattern of two words separated by whitespace. Note that since we need to include a literal \ in the pattern, the C++11 literal string format becomes very useful to simplify the pattern string. Otherwise we would need two \\'s to represent a single \ since \ is the escape character.

// Regular Expression Matching

```
#include <iostream>
#include <regex>
#include <string>

using std::cout;
using std::getline;
using std::cin;
using std::endl;
using std::string;
using std::regex;

int main()
{
	// A phone number in the format xxx-xxx-xxxx
	// The R denotes a literal string rather than
	// escape the \ character.
	string phonePattern = R"(\d{3}-\d{3}-\d{4})";
	// A pattern with two words separated by whitespace
	string twoWordPattern = R"(\w+\s\w+)";
	regex regPhone(phonePattern);
	regex regTwoWord(twoWordPattern);

	string s;
	cout << "Enter a string to test the phone pattern." << endl;
	getline(cin, s);
	if (regex_match(s, regPhone))
		cout << s << " matches " << phonePattern << endl;
	else
		cout << s << " doesn't match " << phonePattern << endl;

	cout << endl;
	cout << "Enter a string to test the two word pattern." << endl;
	getline(cin, s);
	if (regex_match(s, regTwoWord))
		cout << s << " matches " << twoWordPattern << endl;
	else
		cout << s << " doesn't match " << twoWordPattern << endl;
}
```

Example execution:

```
Enter a string to test the phone pattern.
907-867-5309
907-867-5309 matches \d{3}-\d{3}-\d{4}

Enter a string to test the two word pattern.
word up
word up matches \w+\s\w+


Enter a string to test the phone pattern.
867-5309
867-5309 doesn't match \d{3}-\d{3}-\d{4}

Enter a string to test the two word pattern.
oneword
oneword doesn't match \w+\s\w+
```

What if we wanted the first digit of the area code to be a 9?

As a further example of the phone number pattern, let's see how we can combine regular expressions to match phone numbers in either of these formats:

- (999) 999-9999
- 999-999-9999
- 999 999 9999

We need to match the first group of three digits. To match exactly three digits we can use `\d` for a digit and `{3}` for exactly three digit:

```
\d{3}
```

To account for the parenthesis we can allow an optional left and right parenthesis. We have to use the escape character in front of the parenthesis otherwise the parenthesis will be interpreted as grouping for precedence. The `?` after the `\(` matches zero or one left parenthesis and the `?` after the `\)` matches zero or one right parenthesis. The regular expression so far for the first three digits with or without parenthesis is:

```
\(?\d{3}\)?
```

A dash or whitespace separates the first group of digits from the next group of three digits. We can match the dash or whitespace with the regular expression `(-|\s)` which becomes the following when added to the end of our regular expression:

```
\(?\d{3}\)?(-|\s)
```

Next we repeat a group of exactly three digits:

```
\(?\d{3}\)?(-|\s)\d{3}
```

Finally we have a dash or whitespace and exactly four digits:

```
\(?\d{3}\)?(-|\s)\d{3}(-|\s)\d{4}
```

The following code snippet outputs “Phone number found” since `regex_search` returns true if it finds a match to the regular expression anywhere in the target string:

```
string text = "Call me at (907) 867-5309";
string pattern = R"(\(?\d{3}\)?(-|\s)\d{3}(-|\s)\d{4})";
regex reg(pattern);

if (regex_search(text, reg))
      cout << "Phone number found" << endl;
```

Finally, if you wish to find all occurrences that match a regular expression, then you can use a regular expression iterator. The class `sregex_iterator` is used to iterate though all matches of the regular expression within a target string. The class `regex_iterator` is used for a C-style string. An example is shown below in which all phone numbers within the string are displayed. The constructor for the iterator takes the regular expression and references to the beginning and end of the string. Note that by default `end_iterator` is initialized to an ending condition that we can use for `cur_iterator`.

```
string text = "Call me at my desk phone (907) 867-5309 " +
              "or my cell phone 907-350-3491.";
string pattern = R"(\(?\d{3}\)?(-|\s)\d{3}(-|\s)\d{4})";
regex reg(pattern);

sregex_iterator cur_iterator(text.begin(), text.end(), reg);
sregex_iterator end_iterator;
while (cur_iterator != end_iterator)
{
      cout << cur_iterator->str() << endl;
      cur_iterator++;
}
```

Sample Dialogue

```
(907) 867-5309
907-350-3491
```

Smart Pointers

We have described the benefits of pointers but also illustrated the pitfalls if memory management is not performed correctly. Dangling pointers or memory leaks can result in errors that are difficult to find. C++11 includes a new class named `shared_ptr` that simplifies memory management and sharing of objects in memory.

The `shared_ptr` class is a template that is a wrapper around an object allocated from the freestore. The wrapper uses **reference counting** to track how many other pointers reference the object. The counter starts at zero. The counter is incremented each time a new variable references the object. Similarly, the counter is decremented each time a variable no longer references the object. In other words, the counter is decremented when the variable is deleted or reassigned. If the counter reaches zero then the object can be safely deleted and the allocated memory returned to the freestore. This is all performed automatically, which frees the programmer from having to write his or her own memory management code!

As an example, consider the following code which implements a simple linked list of the `Node` class. The class simply stores an integer. The code is written using the "old" format of linking classes via pointer and does not explicitly free the memory that is allocated in the `listTest` function. This means that the program has a memory leak when execution returns to the `main` function. This could cause memory problems if the program did not immediately exit.

```
// Linked list of a simple Node class using traditional pointers.
// Note that this version has a memory leak when execution returns to
// main.
#include <iostream>
using std::cout;
using std::endl;

// A simple Node class. A full-featured class would have
// several more functions.
class Node
{
private:
	int num;
	Node *next;
public:
	Node();
	~Node();
	Node(int num, Node *nextPtr);
	int getNum();
	Node* getNext();
	void setNext(Node *nextPtr);
};

Node::Node() : num(0), next(nullptr)
{ }

Node::Node(int numVal, Node *nextPtr) : num(numVal), next(nextPtr)
{ }

Node::~Node()
{
	cout << "Deleting " << num << endl;
}

int Node::getNum()
{
```

```
	return num;
}

Node* Node::getNext()
{
	return next;
}

void Node::setNext(Node *nextPtr)
{
	next = nextPtr;
}

void listTest()
{
	// Create a linked list with 10->20->30
	Node *root = new Node(10, nullptr);
	root->setNext(new Node(20, nullptr));
	root->getNext()->setNext(new Node(30, nullptr));

	// Output the list
	Node *temp;
	temp = root;
	while (temp != nullptr)
	{
		cout << temp->getNum() << endl;
		temp = temp->getNext();
	}
}

int main()
{
	listTest();
}
```

Sample Dialogue

```
10
20
30
```

Note that despite the existence of a destructor for the `Node` class, the destructor is never called. This is because we never delete each node. The memory allocated in `listTest` is never freed so we have a memory leak in `main`. This is not really a problem since the program immediately exits (at which point memory is reclaimed) but if there were further processing after the call to `listTest` then we may encounter memory problems.

Next, consider the same program written with the `shared_ptr` class. We must include the `<memory>` library. Every occurrence of a pointer to the `Node` class is replaces with `shared_ptr<Node>` instead.

```
// Linked list of a simple Node class using smart pointers.
// There is no memory leak since ths shared_ptr class
// handles reference counting and memory deallocation.
```

```
#include <iostream>
#include <memory>
using std::cout;
using std::endl;
using std::shared_ptr;

// Class modified to use shared_ptr of Nodes.
class Node
{
private:
	int num;
	shared_ptr<Node> next;
public:
	Node();
	~Node();
	Node(int num, shared_ptr<Node> nextPtr);
	int getNum();
	shared_ptr<Node> getNext();
	void setNext(shared_ptr<Node> nextPtr);
};

Node::Node() : num(0), next(nullptr)
{ }

Node::~Node()
{
	cout << "Deleting " << num << endl;
}

Node::Node(int numVal, shared_ptr<Node> nextPtr) : num(numVal),
next(nextPtr)
{ }

int Node::getNum()
{
	return num;
}

shared_ptr<Node> Node::getNext()
{
	return next;
}

void Node::setNext(shared_ptr<Node> nextPtr)
{
	next = nextPtr;
}

void listTest()
{
	shared_ptr<Node> root(new Node(10, nullptr));
	shared_ptr<Node> next1(new Node(20, nullptr));
	shared_ptr<Node> next2;
```

```
        // After a shared_ptr is declared we can set it
        // using the reset function
        next2.reset(new Node(30, nullptr));
        // Link the nodes together
        root->setNext(next1);
        next1->setNext(next2);

        // Output the list
        shared_ptr<Node> temp;
        temp = root;
        while (temp != nullptr)
        {
              cout << temp->getNum() << endl;
              temp = temp->getNext();
        }
}

int main()
{
        listTest();
        cout << "Exiting program." << endl;
}
```

Sample Dialogue

```
10
20
30
Deleting 10
Deleting 20
Deleting 30
Exiting program.
```

Note that the linked list is automatically deallocated for us by the `shared_ptr` class when the variables go out of scope in the `listTest` function. This is done after the call to `listTest` exits, as indicated by the messages output by the `Node` destructor before the program exits.

As a further example, consider what would happen if there is a global variable that references the second item in the linked list. In this case the `shared_ptr` class will not delete the remainder of the items in the list when the `listTest` function exits. This is because the nodes are only deleted when there are no references to them. Note that the use of the global variable is not considered a good programming practice, but is shown here only to illustrate the concept of reference counting.

Additional global variable:

```
shared_ptr<Node> global_reference;
```

Modified code in `listTest`:

```
void listTest()
{
        shared_ptr<Node> root(new Node(10, nullptr));
        shared_ptr<Node> next1(new Node(20, nullptr));
        shared_ptr<Node> next2;
```

```
        // After a shared_ptr is declared we can set it
        // using the reset function
        next2.reset(new Node(30, nullptr));
        // Link the nodes together
        root->setNext(next1);
        next1->setNext(next2);

        // Output the list
        shared_ptr<Node> temp;
        temp = root;
        while (temp != nullptr)
        {
              cout << temp->getNum() << endl;
              temp = temp->getNext();
        }
        // The line below creates a reference to the second item
        // in the linked list
        global_reference = root->getNext();
}
```

Sample Dialogue

```
10
20
30
Deleting 10
Exiting program.
Deleting 20
Deleting 30
```

The big difference is that only the first node is deleted when the `listTest` function exits because it has no references. The remaining two nodes still have references due to the global variable. However, when the program finally exits, even these nodes go out of scope and memory is deallocated.

You should be aware that the `shared_ptr` class does not solve all of your problems. There is a problem if you make a circular list of references, in which case the reference count will never reach 0 and memory will not be reclaimed. To solve this problem, C++11 includes an additional class named `weak_ptr` in which case an object will be destroyed if a `weak_ptr` is the only reference to it. As long as at least one of your links is connected by a `weak_ptr` then the entire circular list will eventually be deallocated.

C++11 also includes a class named `unique_ptr` that cannot be assigned to any other pointer. Older versions of C++ supported a class named `auto_ptr` but it has been deprecated in C++11.