Week 5

Classes

Outline for This Week

- Classes
  - What they are
  - Defining a class
  - Class vs struct
  - Members and methods
  - Scope and classes
  - Defining methods
  - this
  - Initializing class data
  - Access modifiers and encapsulation
  - Classes and Structured Exception Handling
  - Classes and arrays
  - Revisiting call by value vs call by reference
  - Classes and pointers – and dynamic memory allocation
  - Improving the linked list

Classes

- A *class* is user-defined data type that includes characteristics (data) and behavior (functionality).
- Classes are very similar to structures:
  - The difference is the default access level of class members (*public* for struct, *private* for a class)
- When the new data type is used in a declaration, the class is said to be *instantiated* or *instanced*
  - an *object* or *instance* of the class is created.
- Classes have data members and method functions
- Access to members is accomplished via the dot (.) or arrow (->)

Defining a Class

- Define a class is just like a struct, except use the keyword *class*
- Unlike a struct, classes can restrict access to members.
- The keyword *public* must be used for unrestricted access.

```
class date {
public:
  int month;
  int day;
  int year;
};

void main()
{
  date today;
  today.month = 3;
}
```

See class1.cpp

Method Functions

- Functionality can be built-in to the class along with the data. Functions in a class are called *member functions* or *method functions*.

```
class date {
public:
  int month;
  int day;
  int year;
  void set(int m=1, int d=1, int y=2016);
  void display();
};
```


Defining a Method Function

- Method functions can be defined inline within the class definition or external from the class definition.

```
class date {
public:
  int month; int day; int year;
  void set(int m=1, int d=1, int y=2016);
  void display() { cout << month << '/' << year;}
};


void date::set(int m, int d, int y)
{
  month = m; day = d; year = y;
}
```

See class2.cpp

Notes

- All class data is in scope within any method function of the class.
- Caution: Naming method function parameters the same as class member names will cause ambiguity
- "this" is a pointer to the class instance and can be used to avoid ambiguity

```
void date::set(int month, int d, int y)
{  month = month;    // No! Ambiguous
   this->month = month; //Ok
}
```

See class3.cpp

Using Class Methods

- Notice the following:
  - The function names can be simplified to "set" and "display" vs. setDate, etc. since the type of object is implied.
  - The date object is no longer specified as a parameter
- Access the member function the same way you would access member data (using the . or -> notation).

```
date d;
d.set(6,12,1950);
d.display();
```

- The object that is used by *set* and *display* is the one that invokes the function.

```
d.display(); // 'd' is displayed
```

Example: Using a Method Function

- Notice that the <object name>.function() form is analogous to the noun-verb sentence structure.

```
int main()
{
  date christmas;        // Instance date
  christmas.set(12, 25); // call set()
  christmas.display();   // call display()
  return 0;
}
```

- **Do not** call the function using the data type name:

```
date.set();  // illegal.  Date is not an object
```


Initializing the Class

- An initializer list can be used for classes with only data members (called Aggregate classes)

```
class person {
public:
    int id;
    string name;
    double pay;
};
person p = { 101, "Pete", 60 };
```

- A constructor can also be used for initializing a class.
- It is the same as other member functions *except*:
  - it cannot be called explicitly
  - it **cannot return a value** and has no type specified
  - it **must** have the same name as the class

Constructor

- The constructor
  - *Can* have default parameter values
  - *Can* be overloaded
  - *Can* be defined inline or external from the class
  - *Can* call any other method function in the class
  - *Can* contain any functionality that is appropriate to the application.
- It can be used to
  - Initialize the data members of the class
  - Allocate memory required by the class
  - Open a file needed by the class
- If no constructor is defined, default initialization will occur

Default Constructor

- A constructor without parameters is called the *default constructor.*

```
class date {
public:
   int month; int day;  int year;
   date();          //constructor
 };

date::date()
 {
   month=3; day=15; year=2016;
 }
```

See class4.cpp

Using initializer notation

```
date::date(): month(3), day(15), year(2016)
{ }
```

- Note: initialization will happen in the order the members are defined in the class.
- This is significant if you are using one member to initialize another

```
class p{
  int x;  int y;
  p(): y(3), x(y) {}   //No, x will init first
};
```

See class5.cpp

"Calling" the Constructor

- When we instantiate the class, the new object will automatically be initialized by a call to the constructor:

```
date d;
d.display();  // output: 3/15/2016
```

- The constructor can call other functions defied in the class

```
date::date()
{
  set(3,15,1998);
}
```

- Notice: no need to specify the object when calling the set() function.

Overloading the Constructor

- Overload the constructor and add parameters allow the parameterized initialization:

```
class date {
public:
  int month; int day; int year;
  date();
  date(int month,int day,int year);
  void set(int month=1, int day=1, int year=2016);
  void display();
};
date::date(int month,int day,int year)
{    set (month, day, year); }
```


Provide arguments to constructor when an object is instanced

```
void main()
{
   . . .
  date christmas(12,25); //   12/25/2016
  date d;                // 3/15/2016
   . . .
}
```

See class6.cpp

Special Case: Constructor with one Parameter

```
person(string name)
{
    this->name = name;
}

person p("jim");

person p = "jim";
```

See class7.cpp

Data Hiding and Abstraction

- One of the advantages to a C++ class is that it allows true abstraction: the ability to represent an object completely via its interface – the methods.
- This means that the data for the class must be "hidden".
- Using data hiding, the class designer can insure that the data member values are never corrupted- only the class functions will have access to them.
- And, some method functions may be reserved for "internal" use internal only.
- Data hiding is accomplished using the keyword, **"private"**
  - By default, all data and function members are private.
- "protected" also affects restricted data access.

Private vs. Public

	private	public
access from method functions of the class	✓	✓
access via an object of the class		✓

Example

```
class example {
   int a;         // private by default
public:
   float b;
   example();     // at least one constructor is public
private:
   void someFunction() { a = 5; }
};
void main()
{
   example object1;        // create an example object
   object1.a = 7;          // illegal, it's private
   object1.b = 2.3;        // ok, it's public
   object1.someFunction();  // illegal- it's private
}
```

Accessing Private Members

- Useful to create “set”, “get” functions for all private data members.

```
class date {
private:
  int month;  int day;   int year;
public:
  date();
  void setDay(int day) {this->day = day; }
  int  getDay() {return day;}
};
```

See class8.cpp

Nested Classes

- A class type can be used within another class:

```
class person {
public:
 person();
 person(char *name,
        int bmonth, int bday, int byear);
 string name;
 date birthday;
}
person sam;
```

- When a person is instanced, a date and a string are also instanced
- *Question: How would we display sam's birthday?*
  - *Answer:* sam.birthday.display();

Nested Classes - Initialization

The birthday object can be initialized in two ways:

1. Explicitly set each data member in the constructor. Note that the nested class must have a default constructor defined.

```
person(char *qname,
       int month, int day, int year)
{
   birthday.day = day;
   birthday.month = month;
   birthday.year = year;
      . . .
```

2. (Better) Initialize by calling the date constructor directly in the person constructor header.

```
person(char *qname,
       int month, int day, int year):
                birthday(month, day, year)
```


Dynamically Allocating Memory for an Object

- Given our date class from the previous slide- allocate space for a date using new
-
```
date *d = new date;
date *d = new date();
date *d = new date(1,1,2017);
```

- When using new with the default constructor, the () are optional
- However, the results are not equal.
  `new date();`
  - this will zero initialize the class if no default constructor is present.

See class-new.cpp

Access members with a “new” object

- Since you will need a pointer as a handle to the object, pointer notation is in order.

```
date d = new date();
d->display();
```

- You will still need to deallocate memory with delete

```
delete d;
```


Arrays and Classes

- Array notation is the same as for structures
- The issue is how to initialize all of the instances
- The following examples are based on the “person” class below:

```
class person {
private:
 string name;
 date birthday;
public:
 person();
 person(char *name);
 person(char *name, int bmonth, int bday, int byear);
 set(char *name, int bmonth, int bday, int byear);
}
```

Creating Arrays of Classes

- Option 1: Use the default constructor.

```
person employees[2];
```

- Option 2: Use an initializer list

```
person employees[2] = {
    person("Tom",3,2,1940),
    person("Sam",4,2,1940)  };
```

- Option 3: Simplified notation if the class has a constructor with a single parameter

```
person employees[2] = {"bob", "hal" };
```


Creating Arrays of Classes Dynamically Allocated

- Option 4:

```
person *employees[2];
employees[0] = new person("sam", 2,3, 1990);
// ...
```

- Option 5:

```
person *employees[2] = {
new person("sam", 2,3, 1990),
new person("hal", 4,5, 1990) };
```

- Option 6:

```
person *employees = new employees[2];
employees[0]->set("sam", 2,3, 1990);
```

Deallocating the Memory

- What is the difference in the structure of the memory allocated between option 4/5 and option 6?
  - Option 6 allocates a block of two contiguous objects whereas option 4/5 allocates two individual blocks.
- *For option 4 or 5:*

```
delete employees[0];
delete employees[1];
```

- *For option 6*

```
delete [] employees;
```

See class10.cpp

Functions and Classes

- Classes get passed to a function using “call by value”
- That means a copy of the class is created
  - I.e., a new object is instanced
- This is not always desirable
- Better practice to use a reference or const reference

```
void callByRef (myClass &m)
void callByRef (const myClass &m)
```

Classes and Structured Exception Handling

- A custom exception class can both define a data type and augment with more information about the exception.

```
class bad_date
{
   int month; int day;
   public:
   bad_date(int month, int day) :month(month),
   day(day)
       { }
   void error(){ cout << "Invalid date: " << endl
   << "Month: " << month << "  Day: " << day << endl;
   }
};
```

See class11.cpp

Continued ..

```
void set(int month, int day, int year)
{
   if (month > 12 || day > 31)
       throw(bad_date(month, day));
   this->month = month;
   this->day = day;
   this->year = year;
}

try {
       d.set(12, 40, 2016);
}
catch (bad_date d)
{  d.error();  }
```

Organizing Your C++ Program

- In C++, it is the convention to use one header file (.h) for each class (or set of classes with a similar purpose).
- This header file contains class definitions and any associated constructs such as typedef's, enum's etc.
- A corresponding cpp file will contain class method function definitions.
- For example
  - list.h would contain the definition for a linked list class
  - list.cpp would have the corresponding code including #include's for libraries as well as the header file for the class.
- An additional set of code files contains main() and other useful functions, along with corresponding header files.
- Sample Application:

main.cpp	main.h
functions.cpp	
class1.cpp	class1.h
class2.cpp	class2.h

The Linked List - revisited

- Adding functionality to the list element provides a cleaner interface and also allows the program to throw exceptions or do other error processing as needed.
- Functionality for a List Element class:
  - Get or set contained data
  - Get or set what the element is linked to
- Functionality for a Linked List class
  - Initialize the list
  - Append to the list
  - Check if list is empty
  - Move through the list
  - Get contained data

The linked list - revisited

```
class ListElement
{
 private:
   int _data;
   ListElement * _next;
 public:
   ListElement() : _data(0), _next(nullptr) {}
   ListElement(int val): _data(val) {}
   int data() { return _data; }
   void set(int val) { _data = val; }
   void link(ListElement *e) { _next = e; }
   ListElement *next() { return _next; }
};
```


A Linked List Class

```
class LinkedList
{
private:
   ListElement *head;
   ListElement *tail;
   ListElement *current;
public:
   LinkedList(): head(nullptr),
                 tail(nullptr),
                 current(nullptr) {}
   void append(ListElement *e);
   void moveFirst();
   bool moveNext();
   bool empty();
   int data();
};
```

See linked_list_class

Week 6

Advanced Topics in Classes

Outline for This Week

- The destructor
- copy constructor
- move constructor
- = default
- Class declarations
- static members
- const members and mutable members
- Returning *this
- typedef member
- friend's
- Explicit inline methods
- Adding stream functionality
- Overloaded operators
- Conversions and explicit

Dynamically Allocated Data Members

- A class may contain a data member that is a pointer to a dynamically allocated object.
  Why?
  - May not know the size of an object until run-time.
  - Smaller to store a pointer.
  - Provides another method by which to initialize a nested class.

```
class SmartArray{
public:
   int *data;
   SmartArray(int size) {
      data = new int[size];
                . . .
   }
};
```

→ Question: when does the memory get deallocated?

Answer: In the Destructor

- The constructor is called at the beginning of the object's lifetime.
- A destructor is an optional method called implicitly when the lifetime of the object ends
- It is used to "clean up" before an object is destroyed
  - deallocation of memory used by an object
  - close files
- The name of the destructor is created using the name of the class preceded by the character: ~
  - Example: `~date();`
    `date::~date() { . . . }`
- Destructors have no parameters and no return value.
- Destructors are called in the *opposite* order from the constructors.

Adding a Destructor

```
class SmartArray{
public:
  int *data;
  SmartArray(int size) {
     data = new int[size];
                . . .
  }
  ~SmartArray() {
      //  other clean up code perhaps
      delete[] data;
  }
};
```


Method Behaviors

	Method Function	*Constructor*	*Destructor*
Has return value	✓		
Has parameters	✓	✓	
Can be called explicitly	✓		
Called implicitly when class object is instantiated (memory is allocated)		✓	
Called implicitly when class object is destroyed (memory is deallocated)			✓
Can be overloaded	✓	✓	
Can have default parameter values	✓	✓	
Can *call* another function from the class	✓	✓	✓
Can *be called* by another function of the class	✓		
Has access to values of all data members of the class	✓	✓	✓

See destructor.cpp

Types of Constructors

- Auto synthesized
- Default
- Copy
- Move
- Conversion
- Other

Copy Constructor

```
myClass::myClass(const myClass &m);
```

- The purpose of the copy constructor is to make an accurate clone of the class while maintaining pointer and other integrity – *when the class is constructed*.
- The resources/data of the object to be copied remain intact.
- The copy constructor will be automatically synthesized if one is not available.
- The assignment operator does not invoke the copy constructor:

```
myClass m1, m2;
m2 = m1;     // copy constructor not called
```

When is the copy constructor used?

```
class myClass;
void foo(myClass m);

myClass m1;
// all use the copy constructor
myClass m2(m1);
myClass m2 = m1;
myClass m3 = myClass(m1);
myClass *m4 = new myClass(m1);
foo (m1);
```

See copy-cons.cpp

Move Constructor

```
myClass::myClass(myClass&& m);
```

- A move constructor indicates that resources will be depleted from one object to populate another.
  - But there is no requirement to move anything
- Invoke the move constructor using std::move()

See move-cons.cpp

= default and = delete

- Use an "= default" declaration with a constructor to indicate that the default action should be used.
- Use an "= delete" declaration with a constructor to indicate that the default constructor should *not* be generated
- No body is created in either case

Class Declarations

- A class can be "declared" without being fully defined.
  - `class myClass;`
- Use this when a full definition would create a forward reference problem.

```
class B;
class A {
   B * b;
};
class B {
   A * a;
};
```

static members

- Similar to static local variables in a function, a static member has an extended lifetime.
  - Good for data that needs to persist across all instances of an object.
- Specifically, its lifetime is not bound to the class, but to the program.
- Static data members must be defined and initialized prior to instantiation
  - The class definition provides the declaration only
- Memory for static members is allocated per class, not per object.
- Static method functions can be called without using an instance.

Dealing with static Members

- Methods declared as static can access static members – even without a class instance
- Methods declared as static cannot access non-static members
- Non-static members can also access static members.
- Constructors cannot be static

```
class myClass {
    static int n; //static member (also private)
    int x;
public:
   myClass() { x = 0; n++; }
   int getn() { return n; };
   static void setn(int p) { n = p; }
   int getx() { return x; }
   void setx(int p) { x = p; }
};
```

See static.cpp

Classes and typedef

- A class can declare a typedef which then becomes scoped to the class.
- A private typedef will be inaccessible outside of the class.

```
class myClass {
public:
   typedef int * intptr;
   intptr p;
};
int main()
{
   int x = 10;
   myClass::intptr p2;
   myClass m1;
   m1.p = &x;
   p2 = &x;
   cout << *m1.p << " " << *p2 << endl;
}
```

See typedef.cpp

const and mutable

- Declare a member as *const* to indicate it cannot be changed
  - const data members cannot be changed and must be initialized in the constructor
- Declare a method as *const* to indicate that it will not change the data in the class
  - A const instance of the class can only use const methods.
- Declare a data member as *mutable* to indicate that it can be changed – even in a const method
- Note that const is a valid criteria for overloading a function
  - I.e., A const method function can overload a non-const method function
- The const qualifier is placed after the method or data member.

const Class Members

```
class myClass {
private:
  int x;
  const int y;
  mutable int z;
public:
  myClass(): y(5) { x = 3;}
  void foo() const
  {
     x = 3;  // NO!
     z = 3;  // OK
   }
  void foo();
};
```

See const.cpp

Members returning *this

- Return a reference to the class to “chain” several methods together.

```
MyClass m1;
m1.read().add(3).display();
```

See return_this.cpp

Friends

- Friend functions and classes have access private members of another class.
- Friend status must be granted within a class. I.e., a class picks its friends.

```
class MyClass {
 private:
    int x;
 public:
   MyClass() { x = 10;}
  friend class AnotherClass;
  friend void friendFunction(MyClass &m);
 };
```

See friends.cpp

Overloading an Operator

- You may define a function that *overloads* an operator
  - The operands are derived from the function parameters
  - One of the parameters is normally a user defined type (such as a class or enum)
  - When used with a class, these functions may require access to private members. They can be designated as friend functions
  - This can "extend" the functionality of library classes
- For a binary operator, the first parameter is the left side operand. The second parameter is the right side operand.

```
return-type operator op (operand1, operand2);
```

Overloading Operators - Examples

```
ostream & operator << (ostream & out, const MyClass &m);
void operator += (MyClass &m, int n);
```

- "Call" the operator functions

```
MyClass m1;
m1 += 17;        //uses += operator redefinition
cout << m1 << endl;
```

- The usual rules of overloading still apply.
  - For example, this already exists and can't be overloaded:
    `ostream & operator << (ostream & out, int);`
- Rules of operator precedence apply
- The functionality of the operator does not have to be consistent with its "normal" usage.

See ov_op_fcn.cpp

Week 7

Overloaded Operators and Conversions

Outline for This Week

- Overloaded Operators
- Review: Using non-member functions
- Using Member functions
- Overloading arithmetic operators
- Overloading comparison operators
- Overloading unary operators
- Overloading special operators: member access. function call, subscript
- Conversion operators

Overloading an Operator

- You may define a function that *overloads* an operator
  - The operands are derived from the function parameters
  - One of the parameters is normally a user defined type (such as a class or enum)
  - When used with a class, these functions may require access to private members. They can be designated as friend functions
  - This can "extend" the functionality of library classes
- For a binary operator, the first parameter is the left side operand. The second parameter is the right side operand.

```
return-type operator op (operand1, operand2);
```


Overloading Operators - Examples

```
ostream & operator << (ostream & out, const MyClass &m);
void operator += (MyClass &m, int n);
```

- "Call" the operator functions

```
MyClass m1;
m1 += 17;        //uses += operator redefinition
cout << m1 << endl;
```

- The usual rules of overloading still apply.
  - For example, this already exists and can't be overloaded: `ostream & operator << (ostream & out, int);`
- Rules of operator precedence apply
- The functionality of the operator does not have to be consistent with its "normal" usage.

See ov_op_fcn.cpp

Overloaded Operators in a Class

- Classes can also redefine or *overload* operators to change the operator behavior in the context of the class
- This is done using a method function of the class.
- The first (left side) operand is the class object.
- The second (right side) operand is the parameter to the overloaded operator function.
- The overloaded operator must remain consistent with normal usage: i.e., one operand if it is a unary operator, two if it is a binary operator

Creating an Overloaded Operator Method

```
<return-type> <class>::operator op (<type> op2);
```

- Defining the operator method:

```
class MyClass {
      .   .   .
    void operator * (string s);
};
```

- Calling the operator method:

```
MyClass m, *ptr_m;
ptr_m = &m;  string str;
m * str;
*ptr_m * str;
m.operator *(str);
ptr_m->operator *(str);
```

See number1.cpp

Notes

- Overloaded operators for binary operators have one parameter – representing the second operand
- The class object is the first operand.

```
Number n1(2); int x, y = 10;
x = n1 + y;  //ok
x = y + n1;  //NO!
```

- Beware of precedence and shortcut rules
- Do not overload: :: . ?: , &, && ||
- Returning a class reference: when "class-type &" is set as the return type, we can "chain" several operations

```
Number n, m, w;
n = m = w;
```


Member vs Non-member Implementation

- If you want to also allow the class object to be the 2nd operand, use an external operator function instead of creating a member function:

```
int operator + (int, const Number &);
x = y + n1;       //now it's ok
```

- Some operators can only be implemented as member functions: = [] () ->
  - Additionally, it is a good idea to have the following as members: assignment ++ --

See number2.cpp

Overloading the Operator Method

- The usual rules for overloading apply

```
class Number
{
   int *data;
   public:
   Number(int);
   Number();
   ~Number();
   void set (int);
   bool isDefined();
   void show();
   int operator + (int);
   int operator + (const Number &n);
};
```

See number3.cpp

Comparison & Equality Operators

- It's helpful to have an equality operator and comparison operators defined to facilitate sorting, etc with the object
- Return a boolean
- The details of the comparison are class specific
- This is a good candidate to be a const method
- Format:

```
bool <class>::operator op (<type> op2);
```

See ovcomp.cpp

Overloading a Unary Operator

- Do not include parameters
- Use `operator ++ (int)` to indicate postfix notation
- Call the postfix "version" of the operator by placing the operator after the operand or calling it explicitly with an argument

```
class Number;
Number n;
n++;
n.operator ++(0);
```

See number4.cpp, number5.cpp number6.cpp number7.cpp

Overloading new

- The parameter must be of type `size_t`
- Can use *malloc* to do the memory allocation
- Often used to increase performance through a caching or block allocation algorithm.
- `void *operator new(size_t t);`

See number8.cpp

Overloading []

- Overloading the [] operator is generally used to get access to an element of a container.
- The parameter generally indicates size and will be of type size_t
- Return a reference to the element being returned to allow it to be an l-value (i.e., on the left side of an assignment)
- This is often a good candidate as a const method

See ovsubscript.cpp

Overloading -> (Member Access)

- Overload the -> operator to indicate dereference of a member.
- Can only be created as a member function
- More about these when we talk about containers

```
o->m = 10 ;
```

Is same as ..

```
(o.operator->())->m = 10
```

Overloading the function call operator

- Overload the () operator to create a “command” class
- Pass the “function” parameters as the parameter list
- This can make a class “look like” a function.

```
class AbsVal {
public:
   int operator() (int n) const
   {
      return n < 0 ? -n : n;
   }
};

AbsVal absVal;
cout << absVal(-10) << endl;
```

See ovfunction.cpp

Conversions

- Explicit conversions can be done via the constructor.
  - `Number(int n);`
- Implicit conversions can be done using a conversion operator.
  - `operator int()`
- There is no return value specified.
- Use the keyword: explicit to only allow explicit conversion (i.e., using a cast)
- When a conversion operator is created, it may reduce the need for other overloaded operators.

See number9.cpp

Week 8

Lexicon of Challenging C+ Concepts

Or

The Great 8

Outline for This Week

- Conversions and Casting
- enum's
- Initialization
- Memory Management
- Pointers
- References
- Strings and Character arrays
- Vectors vs Arrays
- Plus: Miscellaneous topics
  auto, arguments to main, constexpr

Conversions and Casting

- Implicit
  - Occurs automatically as long as a conversion is defined and there is no loss of data.
  - There are several implicit conversions defined for intrinsic types.
    - Example:
      Given - double d; int n=2;
      d=n; //will convert implicitly
    - Example:
      Given -MyClass1 m1; MyClass2 m2;
      m1 = m2; // will convert implicitly if a conversion is defined from m2 to m1
- Explicit
  - Requires a cast: static_cast<type>()
  - A cast can also be used for implicit conversions for clarity

enum

- enum‘s quickly create several symbolic constants with integral values
- They can be resolved using the :: operator- but not necessary
- They can be placed within a namespace necessitating resolution
- They are NOT an array!
- The “value” of each item is a number, NOT text.

Initialization

- In C++ items are not initialized even if they are created with 'new'.
- The value prior to initialization is *undetermined*
  - In many cases it will be "true" !
- This the same for arrays.
  - int arr[4]={}; or int arr[4]={0};
    initializes to all zeros
  - Note:
    int arr[10] = {1,2,3,4};
    will contain 1,2,3,4 followed by 6 zeroes
  - Special case for char arrays: char abc[]="some text";
- The compiler will try to prevent access to uninitialized data, but you can't count on it.

Initializing Objects

- If a class or struct does not have a constructor, the default constructor will be auto-synthesized.
- Values are *still* not zeroed out.
- Using constructors:
  - As a minimum, a "default constructor" can provide some initialization "seamlessly"
  - A parameterized constructor will facilitate custom initialization of an object
  - A "copy constructor" will initialize an object from another object of the same type
  - A "conversion constructor" will initialize an object from another object of a different type

Memory Management

- Local variables are created on the stack
- Dynamically allocated memory is created in the heap
- Memory is a limited resource
  - Memory “leaks” can cause a program to run out of memory
- Leaks can occur from lack of delete or deleting a single location when delete[] was needed.
- Pitfalls with allocated memory:
  - “Losing” a pointer
  - Access beyond the allocated block
- An allocated block can be accessed with a pointer or using array notation.

Pointers

- A pointer is a variable or constant that stores the address of another variable or constant.
- Use the & (address of) operator to point to a variable/constant.
  - It can also be used to point to another pointer.
- The name of an array is already a pointer – specifically, it is a pointer constant.
  - Given - int array[10];
    array *and* &array[0]
    are equivalent
- Pointer arithmetic can be used to move a pointer – i.e., make it point elsewhere.

Pointers - *continued*

- Pointers to pointers
  - These come up most often as part of a function call as a way to change the pointer itself within the function.
  - foo(int *p); *vs.* foo(int **p);
- Pointers to functions
  - This can be a way to parameterize an action
  - foo(int (*f)())
    the parameter is a pointer to a function that returns an int

References

- A reference is a special case of a variable that literally references another variable/constant.
- It does occupy space in memory, but you cannot see it's address or value because it is implemented to look the same as the item it is referencing. The space used is that of pointer
- References are commonly used as a way to pass an item to a function
  - The item needs to change in the function
  - You want to avoid copying the item when it is passed to the function.
    - In this case, a const & will prevent the item from being changed within the function.

Strings and Character arrays

- A character array can be used as text
- Built-in library functions handle these arrays as long as they are "null terminated" (end in a 0)
  - same as the character '\0'
- Characters are expressed 'a'
- Character array is expressed as "a"
  - This is actually two characters
- A character array is NOT an object/class. Therefore the usual operator overloading is not available
  - Assignment (=) and equals (==) operators will NOT work
- However, there are several functions that can help.
  - For some compilers, you may need to include <cstring>

Character Array Functions

cout <<	output text
cin >>	Input text until whitespace
strcpy(s1, s2)	s1=s2, returns s1
strcmp(s1,s2)	compare s1 to s2. returns 0 (equal), 1, -1 -1: the first char not matching has a lower value in s1 1: the first char not matching has a higher value in s1
strlen(s)	# of characters in s (not including '\0')
strcat(s1, s2)	s1 = s1 + s2
strchr(s, c)	find character c in s. result is the position or -1
strstr (s1, s2)	find string s2 in s1. returns position or -1
strtok(s, delimiter)	split a string into tokens
atoi(s)/atof(s)	"convert" a char array to a number

- All of these assume char * for "string" parameters. The underlying array must have already been created with sufficient size.

String's

- string is part of the standard library

Operators
Given: string s1, s2;

s1 == s2	is s1 equal to s2
s1 != s2	is s1 different from s2
s1 > s2	is s1 alphabetically greater than s2
s1 >= s2	is s1 alphabetically greater than or equal to s2
s1 < s2	is s1 alphabetically less than s2
s1 <= s2	is s1 alphabetically less than or equal to s2
s1 + s2	concatenate s1 with s2
s1 += s2	concatenate s1 with s2, store the result in s1

string – useful methods

str.insert(pos, s)	insert string s or char * s at position *pos*
str.append(s)	append string s or char * s to str
str.find(s)	find first occurrence of string s or char * s in str also, rfind
str.find_first_of(s, pos)	find first occurrence of any character in string s or char * s in str starting at *pos* (defaults to 0) str.find_last_of(s, pos), str.find_not_of(s, pos)
str.replace(pos, howmany, s)	replace characters in str from position, *pos* for *howmany* characters by char *s or string s
str.erase(pos, howmany)	erase string *str* from *pos* for *howmany* chars
str.substr(pos, howmany)	substring from position *pos* for *howmany* chars
str.size()	number of characters in str – also str.length()
str.empty()	true if there are no characters in str
str.resize()	make str larger/smaller -also, str.capacity()
str.compare(s)	compare str to char *s or string s, result: 0, -1, 1
c_str()	returns the 'C' style string (char *) equivalent

Vectors vs Arrays

- A vector is a class in the standard template library.
  - Need to specify the underlying type.
  - The size of the vector is dynamic
  - Memory allocated is contiguous
- A vector appears very similar to an array – but it is an object – think of a smart array
- An array is a locally allocated block of memory.
- Similar to a character array, it is not an object so there are no operators that automatically work with it.

Creating arrays vs vectors

- int array[10];
- int array[10] = {}; // all 0
- vector<int> v(10); // all 0
- vector<int> v(10, 1); // all 1
- vector<int> v(int *ptr1, int *ptr2); // uses ptr1 to ptr 2
- vector<int> v = vec; //initialized from another vector

Miscellaneous Topics

- auto
  - Use auto within a declaration when the type is evident
    auto x = 10;
- arguments to main: argc, argv
  - int argc is the argument count
  - char **argv is an array of pointers
  - use at the command line
- constexpr
  - indicates that an object or method/functionas fit for use in constant expressions.
  - may be evaluated at run-time
  - similar to const