COP 3331
OBJECT ORIENTED DESIGN
SPRING 2017

WEEK 4: POINTERS AND MORE ON CLASSES

SCHINNEL SMALL

UNIVERSITY OF SOUTH FLORIDA

POINTER RECAP

WHAT'S A POINTER AGAIN?

- Recall: A simple variable is a named space in memory that stores a value consistent with its type

- A *pointer variable* (or simply, pointer) is a named space in memory that stores a memory address
  - e.g. `int *ptr;`

- The data type referenced in the declaration is used to specify the type of data the pointer points to

POINTER EXAMPLE

```
#include <iostream>
using namespace std;
int main()
{
   int x = 25;           // int variable
   int *ptr = nullptr;   // Pointer variable, can point to an int
   ptr = &x;       // Store the address of x in ptr

   // Use both x and ptr to display the value in x.
   cout << "Here is the value in x, printed twice:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x

   // Assign 100 to the location pointed to by ptr. (in other words, x)
   *ptr = 100;

   // Use both x and ptr to display the value in x.
   cout << "Once again, here is the value in x:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x
}
```

Program Output

```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```

THE NULL POINTER

- Recall: A null pointer is a pointer that points to *nothing*
- Traditionally we would express this by writing
  - `ptr = 0;`
  - `ptr = NULL;`
- C++ 11 introduced the `nullptr` (see previous example, that is functionally equivalent to the syntax above
- 0 is the only integer that can be directly assigned to a pointer (without casting it as a pointer type)
  - There's a `reinterpret_cast` for that

POINTERS

- Tip: When declaring a pointer, be sure to use the * as an operator for *each* pointer
  - `int *p, q;  // only p is a pointer here`
  - `int *p, *q; // declares p and q as pointers`

- You can use one pointer to manipulate several variables, but this must be done carefully!
  - e.g.

```
ptr = &x; *ptr += 100;
ptr = &y; *ptr += 100;
ptr = &z; *ptr += 100;
```

POINTERS AND FUNCTIONS

POINTERS AND FUNCTIONS

- Recall: you can pass arguments to a function
  - by value
  - by reference

- Passing by value allows a copy of the value to be transferred
  - The change of value in the function does not affect the variable used in the function call

- Passing by reference allows several variables to share the same memory address
  - The change in value in the function affects the variable used in the function call

- You can pass arguments to a a function by reference using a pointer
  - A pointer to the variable is passed by value (copied)
  - The called function accesses the variable by dereferencing the pointer, which is passing by reference)

POINTERS IN FUNCTIONS EXAMPLE

```
#include <iostream>
using namespace std;
void cubeByReference(int*); // prototype

int main()
{
   int number{5};

   cout << "The original value of number is " << number;
   cubeByReference(&number); // pass number address to cubeByReference
   cout << "\nThe new value of number is " << number << endl;
}

// calculate cube of *nPtr; modifies variable number in main
void cubeByReference(int* nPtr)
{
   *nPtr = *nPtr * *nPtr * *nPtr;   // cube *nPtr
}
```

POINTERS AND ARRAYS

POINTERS AND ARRAYS

- Recall: an array is a contiguous group of memory spaces

- An array requires a *base address* from which the ordered memory spaces can begin
  - The base address is the address of position 0

- When we declare an array we are storing the base address of its structure and determining the number of memory spaces needed after the base address (i.e. it's size)

- In other words, we are creating an entity that <u>stores a memory address, that does not change</u> while the array is utilized
  - Or, put another way, an array is a constant pointer

POINTERS AND ARRAYS

- Since an array is technically a pointer, this allows us to use pointer and array notation interchangeably
- Remember:
  `x[index]` is equivalent to `*(x + index)`
- This means that we can dereference an array with * and also that pointers can be used as array names

POINTERS AND ARRAY EXAMPLE

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int NUM_COINS = 5;
   double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
   double *doublePtr;    // Pointer to a double
   int count;            // Array index

   // Assign the address of the coins array to doublePtr.
   doublePtr = coins;

   // Display the contents of the coins array. Use subscripts with the pointer!
   cout << "Here are the values in the coins array:\n";
   for (count = 0; count < NUM_COINS; count++)
      cout << doublePtr[count] << " ";

   // Display the contents of the array again,
   //but this time use pointer notation with the array name!
   cout << "\nAnd here they are again:\n";
   for (count = 0; count < NUM_COINS; count++)
      cout << *(coins + count) << " ";
   cout << endl;
}
```

POINTERS AND ARRAYS

- Remember, as an array name is a constant pointer, you will not be able to change the base address

- Example: Consider the following declarations
```
double readings[20], totals[20];
double *dptr = nullptr;
```

- These statements are legal:
```
dptr = readings; // Make dptr point to readings.
dptr = totals; // Make  dptr point to totals.
```

- But these are illegal:
```
readings = totals; // ILLEGAL! Cannot change readings.
totals =    dptr; // ILLEGAL! Cannot change totals.
```

POINTER ARITHMETIC

POINTER ARITHMETIC

- Addition and subtraction can be performed on a pointer

Operation	Example
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;   // points at 11`

POINTERS AND CONSTANTS

POINTERS AND CONSTANTS

- To store the address of a constant in a pointer, then we need to store it in a *pointer-to-const*

- The syntax for a pointer-to-const is shown below:

The asterisk indicates that `rates` is a pointer.

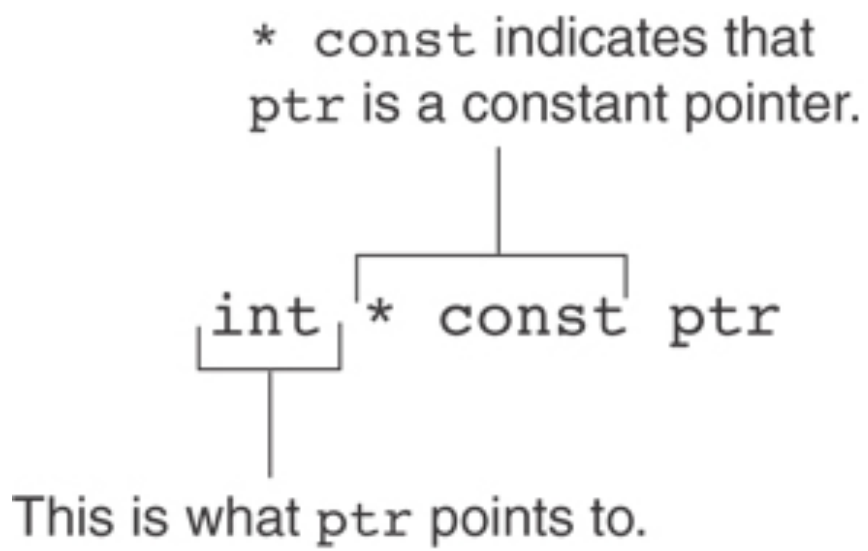
```
const double *rates
```

This is what `rates` points to.

POINTERS AND CONSTANTS

- A *constant pointer* is a pointer that is initialized with an address, and cannot point to anything else

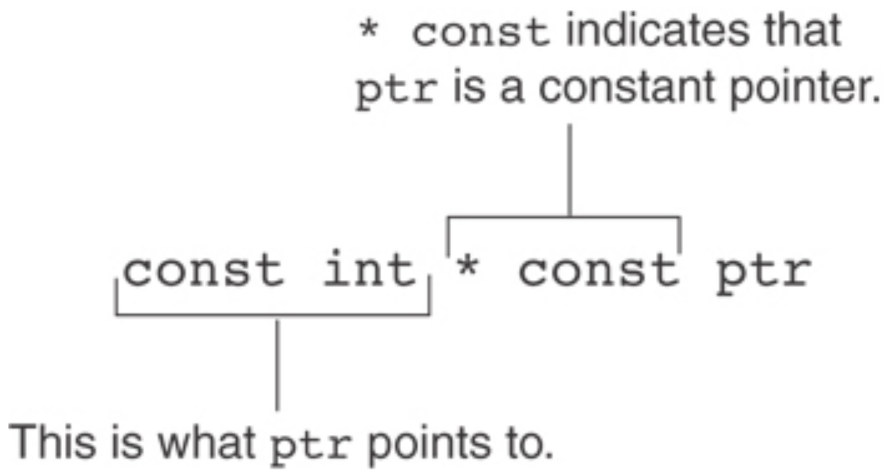
- The syntax for a constant pointer is shown below:



POINTERS AND CONSTANTS

- A *constant pointer to a constant is*:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- The syntax for a constant pointer to a constant is



DYNAMIC MEMORY ALLOCATION

DYNAMIC MEMORY ALLOCATION

- Pointers can be used to allocate storage for a variable during program execution

- A variable created using this option does not have a name, so it must be referenced using the pointer

- You use the `new` operator to allocate memory (this returns address of the memory location)

  e.g.
  ```
  double *dptr = nullptr;
  dptr = new double;
  ```

DYNAMIC MEMORY ALLOCATION

- The new operator can also be used to allocate a dynamic array…

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- You can then use the array or pointer notation to access the array

```
for(i = 0; i < SIZE; i++)
   *arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)
   *(arrayptr + i) = i * i;
```

DYNAMIC MEMORY ALLOCATION

- To free the dynamic memory for a variable:

```
delete fptr;
```

- To free the dynamic memory for an array, use [ ]:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory

DYNAMIC MEMORY ALLOCATION

- C++ 11 introduced smart pointers to dynamically allocate and free the memory after you are done with it

- The `unique_ptr` is one of the smart pointers available.

- You must include the memory header file to use it.

```
#include <memory>
```

- Syntax:

```
unique_ptr<int> ptr( new int );
```

MEMORY LEAK

- If you do not use a smart pointer, or the delete operator, you run the risk of dealing with memory leak

- Memory leak is caused when memory that has been allocated with a pointer cannot be freed

e.g.
```
int *p;
p = new int;
*p = 45;
p = new int;
*p = 66;
```

TWO DIMENSIONAL DYNAMIC ARRAY

- You can also create two dimensional and multi-dimensional dynamic arrays using the new operator
  - Other methods exist, but we will discuss only one method for now

- Concept:
  - Create a *pointer to a pointer*
  - Use the new operator to create an array of pointers
  - Use the new operator again (on each pointer) to create an array of values

TWO DIMENSIONAL DYNAMIC ARRAY

- Create a pointer to a pointer with the syntax

```
datatype ** identifier;
```

  – e.g. `int ** ptr;`

- Use the pointer to pointer to create an array of pointers

```
ptr = new int* [rows];
```

  – Note the use of the * to indicate the the arrays will contain pointers

TWO DIMENSIONAL DYNAMIC ARRAY

- Now that you have an array of pointers, you can use array notation to create dynamic arrays of values

```
for (int r = 0; r < rows; r++)
    ptr [r] = new int [c];
```

- You can pass the dynamic 2d array to functions by creating additional pointer to pointers in the function header

TWO DIMENSIONAL DYNAMIC ARRAY

```
// This program creates a dynamic 2d array. It also uses functions
// to fill the array and print its contents.
#include <iostream>
#include <iomanip>

using namespace std;

void fill(int **p, int rowSize, int columnSize);
void print(int **p, int rowSize, int columnSize);

int main()
{
    int **ptr2table;         //pointer to fill table

    int rows;
    int columns;

    //Get the size of the table from the user
    cout << "Enter the number of rows and columns: ";
    cin >> rows >> columns;
    cout << endl;
```

TWO DIMENSIONAL DYNAMIC ARRAY

```
    //Create the rows of the table; this is the array of pointers
    ptr2table = new int* [rows];

    //Create the columns of the ; this will be the values
    for (int r = 0; r < rows; r++)
        ptr2table[r] = new int[columns];

    //Insert elements into board
    fill(ptr2table, rows, columns);

    cout << "Here is the table:" << endl;

    //Output the elements of board
    print(ptr2table, rows, columns);

    return 0;
}
```

TWO DIMENSIONAL DYNAMIC ARRAY

```
void fill(int **p, int rowSize, int columnSize)
{
    for (int row = 0; row < rowSize; row++)
    {
        cout << "Enter " << columnSize << " number(s) for row "
        << "number " << row << ": ";
        for (int col = 0; col < columnSize; col++)
            cin >> p[row][col];
        cout << endl;
    }
}

void print(int **p, int rowSize, int columnSize)
{
    for (int row = 0; row < rowSize; row++)
    {
        for (int col = 0; col < columnSize; col++)
            cout << setw(5) << p[row][col];
        cout << endl;
    }
}
```

MORE ON CLASSES AND OBJECTS

THE INCLUDE GUARD

- When your main program file has an `#include` directive for a header file, there's a possibility that the header file will have an `#include` directive for a second header file

- If the main file also has an `#include` directive for the second header file, then the preprocessor will include the second header file twice

- You can use an include guard to prevent this
  – It prevents the header file from accidentally being included more than once

THE INCLUDE GUARD & DEFINE DIRECTIVE

- The syntax for an include guard is
  `#ifndef CONSTANT … #endif`
- ifndef means "if not defined"
- The constant represents a version of the class that has already been loaded
- If the constant is not defined, then we use the `#define` directive to define it
- The `#endif` directive is used to enclose the definition of the class (from the `#ifndef` directive)
  - In other words, if not defined, create the class enclosed between the directive

INCLUDE GUARD EXAMPLE

```
// Specification file for the Rectangle class.
#ifndef RECTANGLE_H
#define RECTANGLE_H

// Rectangle class declaration.

class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};

#endif
```

STATIC CLASS MEMBERS

STATIC CLASS MEMBERS

- When we instantiate an object, each object has its own copies of the class's instance variables

- If a member variable is declared `static`, all instances of the class has access to that variable
  - Remember the `static` modifier 'preserves' the memory space

- If a member function is declared `static`, it may be called without any instances of the class being defined

STATIC MEMBER FUNCTIONS

- A function that is a static member of a class can’t access any non static data in its class
- So, why would it be useful?
  - Even though static member variables are declare din a class, the are actually defined outside the class declaration
  - This allows for the existence of the variable before any instance of the class are created

STATIC MEMBER FUNCTIONS

- Why is is useful? (cont'd)
  - A class's static member functions can be called before any instances of the class are created
  - This means that the class's static member functions can access the class's static member variables before any instances of the class are defined
  - This allows us to create very specialized setup routines for class objects

STATIC CLASS VARIABLE EXAMPLE

```
// Tree class
class Tree
{
private:
   static int objectCount;     // Static member variable.
public:
   // Constructor
   Tree()
      { objectCount++; }

   // Accessor function for objectCount
   int getObjectCount() const
      { return objectCount; }
};

// Definition of the static member variable, written
// outside the class.
int Tree::objectCount = 0;
```

STATIC CLASS VARIABLE EXAMPLE

```
// This program demonstrates a static member variable.
#include <iostream>
#include "Tree.h"
using namespace std;

int main()
{
   // Define three Tree objects.
   Tree oak;
   Tree elm;
   Tree pine;

   // Display the number of Tree objects we have.
   cout << "We have " << pine.getObjectCount()
        << " trees in our program!\n";
   return 0;
}
```

Program Output

```
We have 3 trees in our program!
```

FULL STATIC EXAMPLE

```
#ifndef BUDGET_H
#define BUDGET_H

// Budget class declaration
class Budget
{
private:
   static double corpBudget;  // Static member variable
   double divisionBudget;     // Instance member variable
public:
   Budget()
      { divisionBudget = 0; }

   void addBudget(double b)
      { divisionBudget += b;
        corpBudget += b; }

   double getDivisionBudget() const
      { return divisionBudget; }

   double getCorpBudget() const
      { return corpBudget; }

   static void mainOffice(double);  // Static member function
};

#endif
```

FULL STATIC EXAMPLE

```
#include "Budget.h"

// Definition of corpBudget static member variable
double Budget::corpBudget = 0;

void Budget::mainOffice(double moffice)
{
   corpBudget += moffice;
}
```

FULL STATIC EXAMPLE

```
// This program demonstrates a static member function.
#include <iostream>
#include <iomanip>
#include "Budget.h"
using namespace std;

int main()
{
   int count;                        // Loop counter
   double mainOfficeRequest;         // Main office budget request
   const int NUM_DIVISIONS = 4;      // Number of divisions

   // Get the main office's budget request.
   // Note that no instances of the Budget class have been defined.
   cout << "Enter the main office's budget request: ";
   cin >> mainOfficeRequest;
   Budget::mainOffice(mainOfficeRequest);

   Budget divisions[NUM_DIVISIONS]; // An array of Budget objects.
```

FULL STATIC EXAMPLE

```
// Get the budget requests for each division.
   for (count = 0; count < NUM_DIVISIONS; count++)
   {
      double budgetAmount;
      cout << "Enter the budget request for division ";
      cout << (count + 1) << ": ";
      cin >> budgetAmount;
      divisions[count].addBudget(budgetAmount);
   }

   // Display the budget requests and the corporate budget.
   cout << fixed << showpoint << setprecision(2);
   cout << "\nHere are the division budget requests:\n";
   for (count = 0; count < NUM_DIVISIONS; count++)
   {
      cout << "\tDivision " << (count + 1) << "\t$ ";
      cout << divisions[count].getDivisionBudget() << endl;
   }
   cout << "\tTotal Budget Requests:\t$ ";
   cout << divisions[0].getCorpBudget() << endl;
}
```

ANNOUNCEMENTS

ANNOUNCEMENTS

- No assignment this week