COP 3331
OBJECT ORIENTED DESIGN
SUMMER 2017

WEEK 2 – WEDNESDAY (MAY 24TH): MORE ON CLASSES, POINTERS

UNIVERSITY OF SOUTH FLORIDA

RECAP: CONSTRUCTOR FUNCTION

```
#include <string>
using namespace std;

class Account
{
private:
    string name;

public:
    Account(string n);    // constructor prototype
    void setName(string accountName);
    string getName() const;
};

// This constructor will initialize the account name.
Account::Account(string n)
{
    name = n;
}

// mutator function
void Account::setName(string accountName)
{    name = accountName;   }

// accessor function
string Account::getName() const
{    return name;    }
```

With this constructor, the only line the changes in the driver program is the object declaration line:

```
Account myAccount{"Johnny Appleseed"};
```

EXPLICIT KEYWORD IN IN-LINE CONSTRUCTORS

- A constructor with a single non-default parameter is called *converting constructor/function* since such a constructor allows conversion of an argument type to an object of a class.

- Compiler uses such converting functions for *implicit* conversion in the code.

- The explicit specifier avoids this implicit conversion.

CONSTRUCTOR EXAMPLE INSIDE CLASS DEFINITION

- Standard syntax:

```
Account(string n)
 {
     name = n;
 }
```

With explicit keyword:

```
explicit Account (string n):
name {n}
{
   // Empty body
}
```

- Applies only to constructors defined with one parameter

EXAMPLE OF EXPLICIT CONSTRUCTOR

- Wihtout explicit:

```
Account(string n)
 {
      name = n;
 }
```

With explicit keyword:

```
explicit Account (string n):
name {n}
{
   // Empty body
}
```

```
int main ()
{
Account myaccount {"James"};

if (myaccount == "James" )
     cout << "Hi";
};
```

✓

```
int main ()
{
Account myaccount {"James"};

if (myaccount == "James" )
     cout << "Hi";

};
```

✗

EXAMPLE OF EXPLICIT CONSTRUCTOR

- Standard syntax:

```
 Account(string n)
  {
      name = n;
  }
```

With explicit keyword:

```
explicit Account (string n):
name {n}
{
    // Empty body
}
```

```
int main ()
{
Account myaccount = {"James"};

};
```

✓

```
int main ()
{
Account myaccount = {"James"};

};
```

✗

CLASS EXAMPLE WITH MORE SPECIFIC FUNCTIONS

```
#include <string>
using namespace std;

class Account
{
private:
    string name;
    int balance{0};  // data member with default initial value

public:
    // constructor
    Account(string n, int initBal) : name{n}
    {
        // could have written name = n here instead of appending it
        // to the function header

        // validate that the initBal is greater than 0; if not,
        // data member balance keeps its default initial value of 0
        if (initBal > 0)
        {                      // if the initBal is valid
            balance = initBal; // assign it to data member balance
        }

    }
```

CLASS EXAMPLE WITH MORE SPECIFIC FUNCTIONS

```
    // function that deposits (adds) only a valid amount to the balance
    void deposit(int depositAmount)
    {
        if (depositAmount > 0)  // if the depositAmount is valid
        {
            balance = balance + depositAmount; // add it to the balance
        }
    }

    // accessor function that returns the account balance
    int getBalance() const
    {   return balance;     }

    // mutator
    void setName(string accountName)
    {   name = accountName;     }

    // accessor
    string getName() const
    {   return name;    }

};
```

DRIVER PROGRAM FOR PREVIOUS EXAMPLE

```
#include <iostream>
#include "Account.h"
using namespace std;

int main()
{
    Account account1{"Jane Green", 50};
    Account account2{"John Blue", -7};

    // display initial balance of each object
    cout << "account1: " << account1.getName() << " balance is $"
    << account1.getBalance();
    cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance();

    cout << "\n\nEnter deposit amount for account1: "; // prompt

    int depositAmount;
    cin >> depositAmount; // obtain user input
    cout << "adding " << depositAmount << " to account1 balance";
    account1.deposit(depositAmount);  // add to account1's balance
```

DRIVER PROGRAM FOR PREVIOUS EXAMPLE

```
    // display balances
    cout << "\n\naccount1: " << account1.getName() << " balance is $"
    << account1.getBalance();
    cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance();

    cout << "\n\nEnter deposit amount for account2: "; // prompt
    cin >> depositAmount; // obtain user input
    cout << "adding " << depositAmount << " to account2 balance";
    account2.deposit(depositAmount); // add to account2 balance

    // display balances
    cout << "\n\naccount1: " << account1.getName() << " balance is $"
    << account1.getBalance();
    cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance() << endl;
}
```

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

Program Output

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

```
#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
}
```

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:
  - 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 stores a memory address, that does not change 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 pointer 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`

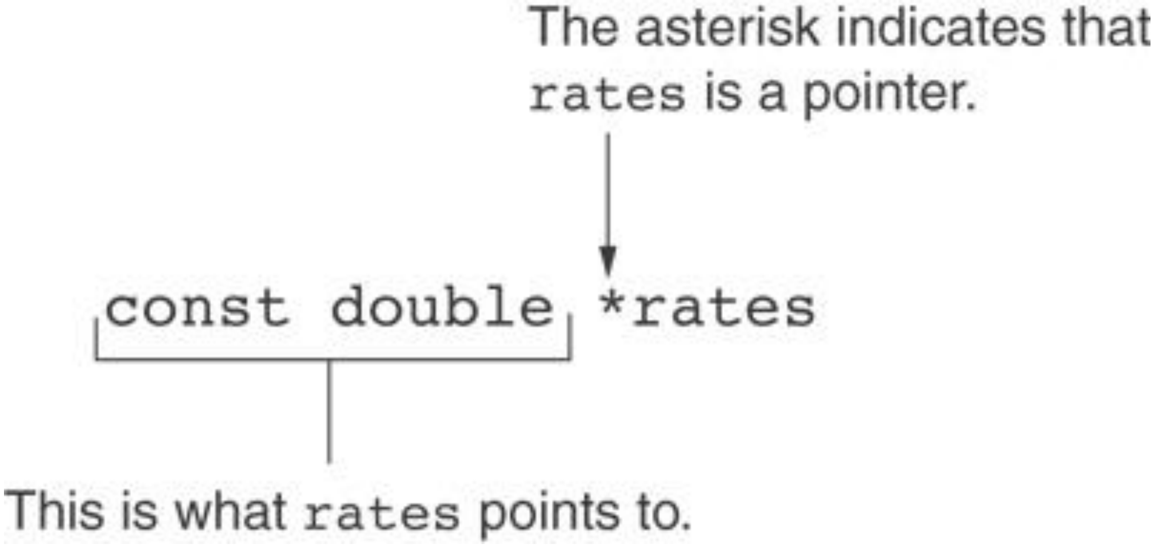
`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;   // points at 11`

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.

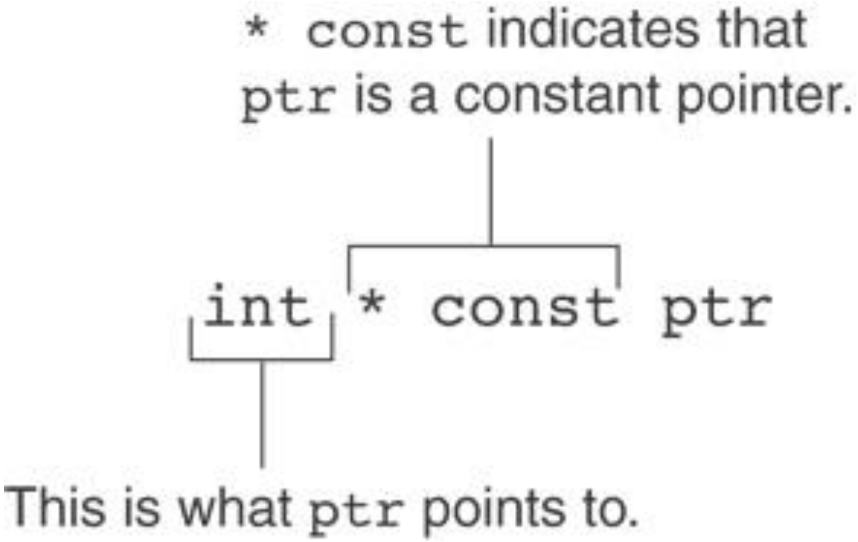
```
const double *rates
```

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:



* const indicates that ptr is a constant pointer.

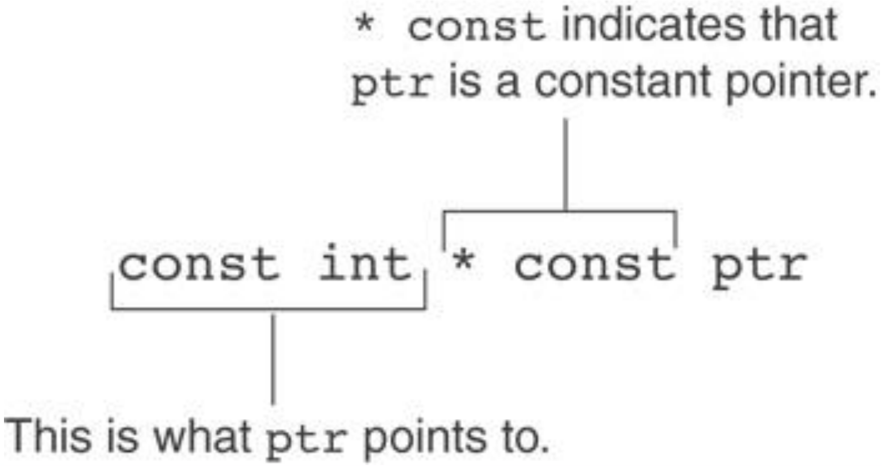
```
int * const ptr
```

This is what ptr points to.

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



* const indicates that ptr is a constant pointer.

```
const int * const ptr
```

This is what ptr points to.

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;

double *arrayptr;
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 dptr;
```

- 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 table; 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;
    }
}
```