Review 2

CS A250 – C++ Programming II

Objectives

- Items that will be reviewed:
  - Pointers and dynamic data
  - Memory management
    - De-allocating memory
    - Dangling pointer problem
  - Classes
    - Principles of OOP
    - Separate compilation
    - Constructor
    - The const modifier for member functions
    - Destructor
  - The STL string class

Pointers

- **Computer memory** is divided into numbered locations (**bytes**)
  - **Variables** are implemented as a sequence of adjacent bytes
- A **pointer** is a **memory address** of a **variable**
  - Specifies ***where*** the variable is located (where the variable starts)
- You have already used **pointers**:
  - Passing by reference passes the address of a variable, not the actual value

Pointers (cont.)

- **Pointers** are "typed"
  - You can store a pointer in a variable, but a pointer is **not** a type (**`int`**, **`double`**, etc.)
    - A pointer *points to* an **`int`**, **`double`**, etc.
- The **dereference operator** (**`*`**) indicates a pointer

```
int *p;
```

  - **`p`** is a pointer that can point to an **`int`**
    - Cannot point (refer) to anything else
  - **`p`** will contain the address of where the integer is located

Common Error

- Two ways to place the **dereference operator**

`int *p` same as `int* p`

- **Careful!** Whether you write

`int* p1, p2;` **OR** `int *p1, p2;`

  - You are declaring a ***pointer*** **`p1`** and a ***variable*** **`p2`**
  - Each pointer needs to have its own **dereference operator**

Addresses and Numbers

- A **pointer** is an **address**
- An **address** is an **integer**
- A **pointer** is **NOT** an **integer**!
  - This is **abstraction**
- C++ forces **pointers** be used as **addresses**
  - <u>Cannot</u> be used as numbers
  - Even though a pointer ***is*** a number

The & Operator

- The " *address of* " operator **&**
- Also used to specify **call-by-reference parameter**

```
int *p1, *p2, v1, v2;

p1 = &v1;
```

Sets pointer variable **p1** to "point to" int variable **v1**

- Operator **&**
  - Determines " address of " variable
- Read like:
  - "**p1** equals address of **v1**" **OR**
  - "**p1** points to **v1**"

POINTING TO… (CONT.)

- Given the following:

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- **Two ways** to refer to **v1** now:
  - Variable **v1** itself

```
cout << v1;
```

  - Via pointer **p1**

```
cout << *p1;
```

Pointer Assignments

- Pointer variables can be "**assigned**"

```
int *p1, *p2;

p1 = p2;
```

  - Assigns one pointer to another
  - "Make **p1** point to where **p2** points“

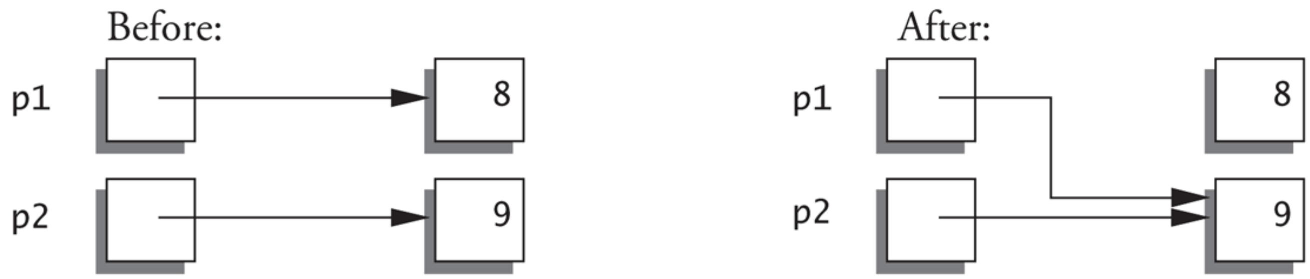
- Do **not** confuse with

```
*p1 = *p2;
```

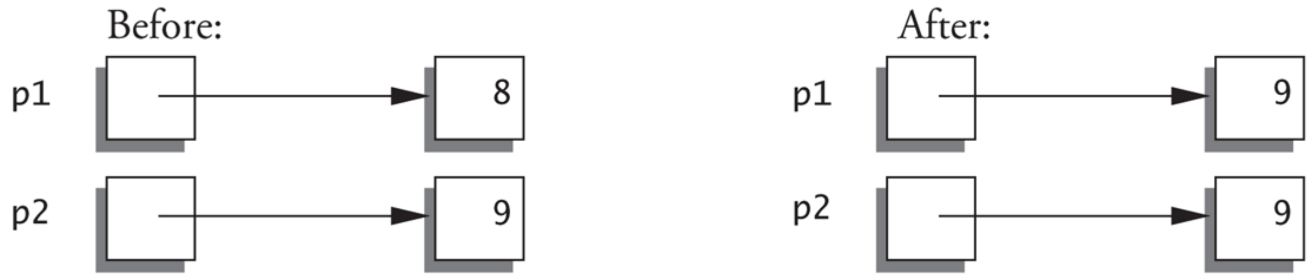
  - Assigns "value pointed to" by **p1**, to
    "value pointed to" by **p2**

Pointer Assignments

```
p1 = p2;
```




```
*p1 = *p2;
```



Another Example

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v


cout << v;
cout << p1;
cout << p2;
cout << *p1;
cout << *p2;
cout << &p1;
```

What is the output?

Another Example

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v


cout << v;             5 (value of v)
cout << p1;
cout << p2;
cout << *p1;
cout << *p2;
cout << &p1;
```

Another Example

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v


cout << v;              5 (value of v)
cout << p1;             0066FDE0 (address of v)
cout << p2;
cout << *p1;
cout << *p2;
cout << &p1;
```

ANOTHER EXAMPLE

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v


cout << v;             5 (value of v)
cout << p1;            0066FDE0 (address of v)
cout << p2;            0066FDE0 (address of v)
cout << *p1;
cout << *p2;
cout << &p1;
```

Another Example

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v


cout << v;             5 (value of v)
cout << p1;            0066FDE0 (address of v)
cout << p2;            0066FDE0 (address of v)
cout << *p1;           5 (value p1 is pointing to)
cout << *p2;
cout << &p1;
```

Another Example

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v

cout << v;          5 (value of v)
cout << p1;         0066FDE0 (address of v)
cout << p2;         0066FDE0 (address of v)
cout << *p1;        5 (value p1 is pointing to)
cout << *p2;        5 (value p2 is pointing to)
cout << &p1;
```

Another Example

```
int *p1, *p2, v; //declare two pointers and one variable
v = 3;           //variable is equal to 3
p1 = &v;         //pointer holds the address of the var
p2 = p1;         //set p2 to point to v as well
*p1 = 5;         //overwrite the value pointed by p1
                 //   with the value of v

cout << v;          5 (value of v)
cout << p1;         0066FDE0 (address of v)
cout << p2;         0066FDE0 (address of v)
cout << *p1;        5 (value p1 is pointing to)
cout << *p2;        5 (value p2 is pointing to)
cout << &p1;        0034FDF8 (address of p1)
```

Pointers and Functions

- **Pointers** are full-fledged types
  - Can be used just like other types
- Can be function **parameters**
- Can be ***returned*** from functions

```
int* someFunction(int* p);
```

  - This function declaration
    - Has a "*pointer to an int*" parameter
    - Returns a "*pointer to an int*" variable

THE **new** OPERATOR

- Since pointers can refer to variables…
  - No "real" need to have a standard **identifier**
- Can ***dynamically*** allocate variables
  - Operator **new** creates variables
    - No identifiers to refer to them
    - Just a pointer!

```
int *p1;
p1 = new int;
```

- Creates new "*nameless*" variable, and assigns **p1** to "point to" it
- Can access with ***p1** (use just like an ordinary variable)

THE `new` OPERATOR (CONT.)

- **Now, the question is:**
  - **Why** would you declare a **dynamic variable**?

THE `new` OPERATOR (CONT.)

- **Now, the question is:**
  - **Why** would you declare a **dynamic variable**?
  - Because, once you are done using a dynamic variable, you can actually **reclaim the memory** used by it.
  - Dynamic variables are store in a section of **memory** that we can actually manipulate, the **heap**.

Memory Management

- The **heap**
  - Also called "**freestore**"
  - Reserved for ***dynamically-allocated variables***

  - All **new** dynamic variables consume memory in **freestore**
    - If too many → could use all **freestore** memory
    - Future "new" operations will fail if heap is "full"

Memory Management (cont.)

- Because you have control of the **heap**

1. You need to reclaim the memory not used anymore

AND

2. You need to make sure the pointer does not point to that section of memory any longer

De-allocating Memory

- Always **de-allocate** ***dynamic memory***
  - When no longer needed
  - To restore memory in the **heap**

```
int *p = new int;

// other code...

delete p;
p = NULL;
```

De-allocating Memory

- Always **de-allocate** ***dynamic memory***
  - When no longer needed
  - To restore memory in the **heap**

```
int *p = new int;

// other code...

delete p;
p = NULL;
```

De-allocates dynamic memory pointed by pointer **p**

Pointer **p** is not pointing to any section of memory any longer

Dangling Pointers

- The expression

```
delete p;
```

  - Destroys **dynamic memory**
  - But **p** still points there!
    - Pointer **p** is now a "**dangling pointer**"
  - If **p** is then "**dereferenced**" (***p**)
    - We can have unpredictable results!
- **Avoid** dangling pointers
  - Assign pointer to **NULL** after deleting the variable

```
delete p;
p = NULL;
```

Dynamic Arrays

- Limitations of **static arrays**
  - Must **specify capacity** *first* → can be a waste of memory
  - May not know until program runs
- **Dynamic arrays**
  - Capacity **not** specified at programming time
  - Determined while program is ***running***
  - Use **`new`** operator

```
double *a = new double[10];
```

Deleting Dynamic Arrays

- Always **de-allocate** ***dynamic arrays***
  - When no longer needed
  - To restore memory in the heap

```
int *a = new int[10];

// other code...

delete [] a;  // note the brackets [] !!!
a = NULL;
```

  - De-allocates dynamic memory "pointed to by pointer **a**"
    - By adding `[]` we are specifying it is an array

Deleting Dynamic Arrays

- Always **de-allocate** ***dynamic arrays***
  - When no longer needed
  - To restore memory in the heap

```
int *a = new int[10];

// other code...

delete [] a;
a = NULL;
```

De-allocates dynamic memory pointed to by pointer **a**"

By adding `[]` we are specifying it is an **array**

Null the pointer so that it does not point to any section in memory any longer.

Classes

CLASSES

- Integral to **object-oriented programming** (OOP)
  - In C++ **variables** of ***class type*** are **objects**
  - Objects contain ***data*** and ***operations***

- **Principles of OOP**
  - **Information Hiding**
    - Details of how **operations** work not known to "user" of class
  - **Data Abstraction**
    - Details of how **data** is manipulated within **Abstract Data Type** (**ADT**) and class not known to "user"
  - **Encapsulation**
    - Bring together **data** and **operations**

Separate Compilation

- With separate **interface** and **compilation**:
  - "User" of class does not need to see details of how class is implemented
  - "User" only needs **rules** (**interface**) for the class
    - In **C++** → **public member functions** and **associated comments**
  - **Implementation** (**compilation**) of class ***hidden***
    - Member function definitions elsewhere

File: ccp_separate_compilation.pdf

Constructors

- ***Key*** principle of **OOP**
- **Initialize objects**
  - Initialize some **or** all **member variables**
  - Other actions possible as well
    - Validate data to ensure appropriate data is assigned to class private member variables
- Must be in ***public*** section of the class
- Can **overload** constructors just like other functions
- **Default constructor**
  - Constructor w/ **no** arguments
  - Should always be defined

The `const` Modifier on Functions

- When to make a **member *function* `const`**?
  - When the **function** does **not** modify any **member variables**

```
class MyClass
{
public:
    ...
    void print( ) const;
    ...
private:
    int myInt;
};
```

Function **print** does **not** modify the **member variable** **myInt**
➔ Make the function **const**

```
void MyClass::print( ) const
{
    cout << myInt << endl;
}
```

THE `const` MODIFIER ON FUNCTIONS (CONT.)

- Trying to make a function **const** when the function **modifies** any of the **member variables**, will result in an **error**

```
class MyClass
{
public:
     ...
     void add(int value);
     ...
private:
     int myInt;
};
```

Function **add** will **modify** the **member variable** myInt
➔ **Cannot** make the function **const**

```
void MyClass::add(int value)
{
     myInt += value;
}
```

Destructor

- **Destructor**
  - Automatically called when object is out of scope
  - Default version removes ***only*** ordinary variables, ***not*** dynamic variables
- If **pointers** are private member data
  - Then you are creating **dynamic variables** (**new**)
  - Need to de-allocate dynamic data (**delete**)
  - Need to **NULL** pointer
    - This last step is not really necessary, because the pointer is an ordinary variable and will be destroyed automatically, BUT it is good practice

Classes - Example

- **File:** Cpp_separate_compilation
- **Project:** Employee Class

THE STL `string` CLASS

THE STL `string` CLASS

- Defined in library

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

- String variables and expressions
  - Treated much like simple types
- Can ***assign***, ***compare***, ***add (concatenate)***

```
string str1, str2, str3;
str3 = str1 + str2;       // Concatenation
str3 = "Hello Mom!"       // Assignment
```

THE STL `string` CLASS (CONT.)

- When creating a string, you are using the default constructor of the STL string class
  - No need to initialize the string to an empty string
  - The default constructor already does that

```
string str = "";   // this is redundant

string str;        // str is already an
                   // empty string
```

End Review 2

(no exercise)