CS250 Intro to CS II

Spring 2014

Chapter 9 - Arrays, Pointers, Dynamic Memory
Chapter 14.4 Copy Constructors

- Reading: pp. 500-528
- Good Problems to Work: p.506 9.1, 9.3, 9.4, 9.5, 9.6, 9.7

- Reading: pp. 812-818
- Good Problems to Work: p. 858 6, 7, 8, 9, 10, 11, 18

Arrays and Pointers

- Array names can be used as constant pointers
- Pointers can be used as array names

```
short numbers[] = {5, 10, 15, 20, 25};

cout << "numbers[0] = " << *numbers << endl;
cout << "numbers[1] = " << *(numbers + 1) << endl;
cout << "numbers[2] = " << numbers[2] << endl;
```

Problem

- Consider the following C++ segment

```
const int SIZE = 8;
int numbers[] = {5, 10, 15, 20, 25, 30, 35, 40};
int *pNumbers, sum = 0;
```

- Write the C++ code using only pointer notation that will print the sum of the values found in the array numbers

Pointer Arithmetic

- Some mathematical operations can be performed on pointers
  a) ++ and -- can be used with pointer variables
  b) an integer may be added or subtracted from a pointer variable
  c) a pointer may be added or subtracted from another pointer

If the integer pointer variable pInt is at location 1000, what is the value of pInt after pInt++; is executed?

Pointers and Functions

- What are the two ways of passing arguments into functions?

- Write two functions **`square1`** and **`square2`** that will calculate and return the square of an integer.
  - **`square1`** should accept the argument passed by value,
  - **`square2`** should accept the argument passed by reference.

Pointers as Function Arguments

- A pointer can be a formal function parameter
- Much like a reference variable, the formal function parameter has access to the actual argument
- The address of the actual argument is passed to the formal argument

Pointers as Function Arguments

```
void square3 (int *pNum)
{
  *pNum *= *pNum;
}
```

- What would a function call to the above function look like?

Pointers to Constants

- A pointer to a constant means that the compiler will not allow us to change the data that the pointer points to.

```
void printArray (const int *pNumbers)
{
}
```

Constant Pointers

- A constant pointer means that the compiler will not allow us to change the actual pointer value BUT we can change the data that the pointer points to.

```
void printArray (int * const pNumbers)
{
}
```

Constant Pointers to Constants

- A constant pointer to a constant means the compiler will not allow us to change the actual pointer value OR the data that the pointer points to.

```
void printArray (const int * const pNumbers)
{
}
```

Problem

Using pointer notation, write a C++ function printCharacters that will accept a character array and the size of the array. The function will print each element of the array on a separate line.

Dynamic Memory Allocation

- Variables can be created and destroyed while a program is running
- **new** is used to dynamically allocate space from the heap. A pointer to the allocated space is returned
- **delete** is used to free dynamically allocated space

Using new and delete

```
int *pInt;

pInt = new int;

*pInt = 5;

cout << *pInt << endl;

delete pInt;
```

Pointers to Arrays

- We can dynamically create space for an array

```
int *pAges, sum = 0;
pAges = new int[100];
for (int i = 0; i < 100; ++i)
{
  *(pAges + i) = i; // or pAges[i] = i;
}
delete [] pAges;
```

NULL Pointer

- A null pointer contains the address 0
- The address 0 is an unusable address

```
pAges = new int[100];
if (NULL == pAges)
{
  cout << "Memory Allocation Error\n";
  exit (EXIT_FAILURE);
}
```

- Only use delete with pointers that were used with new

Memberwise Assignment

Consider the following C++ code:

```
Rectangle cBox1 (10.0, 5.0), cBox2;
```

What is the meaning of:

```
cBox2 = cBox1;
```

Destructors

- The opposite of constructors
- Have the same name as the class, with a ~ in front of it
- Called whenever an object is destroyed
- A destructor has no arguments and or return value
- Only one destructor allowed!
- No need for us to explicitly declare a destructor unless there are pointer variables in the class

Constructor/Destructor Example

```
class Test
{
  public:
    Test(int);
    ~Test();

  private:
    int mId;
};

Test::Test(int i)
{
  mId = i;
  cout << "constructor for " << mId << " is called\n";
}
Test::~Test()
{
  cout << "destructor for " << mId << " is called\n";
}
```

What is the output?

```
void funct();

int main()
{
  Test cTest1(1);
  funct();
  Test cTest3(3);

  return EXIT_SUCCESS;
}

void funct()
{
  Test cTest2(2);
}
```

Copy Constructor

- A copy constructor is a special constructor called when a new object is created and initialized with the data from another object
- Most times the default memberwise assignment is OK. When is this not the case?

class Person Interface

```
#ifndef PERSON_H
#define PERSON_H

class Person
{
  public:
    Person (char * = NULL, unsigned short = 0);
    Person (const Person &);
    ~Person ();
    const char *getName () const;
    int getAge () const;

 private:
    char *mpszName;
    unsigned short mAge;
};

#endif
```

class Person Implementation

```
#include "Person.h"
#include <iostream>
using namespace std;
// Constructor
Person::Person (char *pszName, int age)
{
  if (NULL != pszName)
  {
    int nameLength = strlen (pszName);
    mpszName = new char[nameLength + 1];
    strncpy_s (mpszName, nameLength + 1,
               pszName, nameLength + 1);
    mAge = age;
  }
}
```

class Person Implementation

```
// Copy Constructor used to initialize an object
// being created
Person::Person (const Person &rcPerson)
{
  if (NULL != rcPerson.mpszName)
  {
    int nameLength = strlen (rcPerson.mpszName);
    mpszName = new char[nameLength + 1];
    strncpy_s (mpszName, nameLength + 1,
    rcPerson.mpszName, nameLength + 1);
    mAge = rcPerson.mAge;
  }
}
```

class Person Implementation

```
Person::~Person ()
{
  delete [] mpszName;
}

const char *Person::getName () const
{
  return mpszName;
}

unsigned short Person::getAge () const
{
return mAge;
}
```

Person Driver

```
#include "Person.h"
#include <iostream>

using namespace std;

int main ()
{
Person cPerson ("John Smith", 18);

cout << cPerson.getName () << " is "
     << cPerson.getAge () << " years old." << endl;

return EXIT_SUCCESS;
}
```

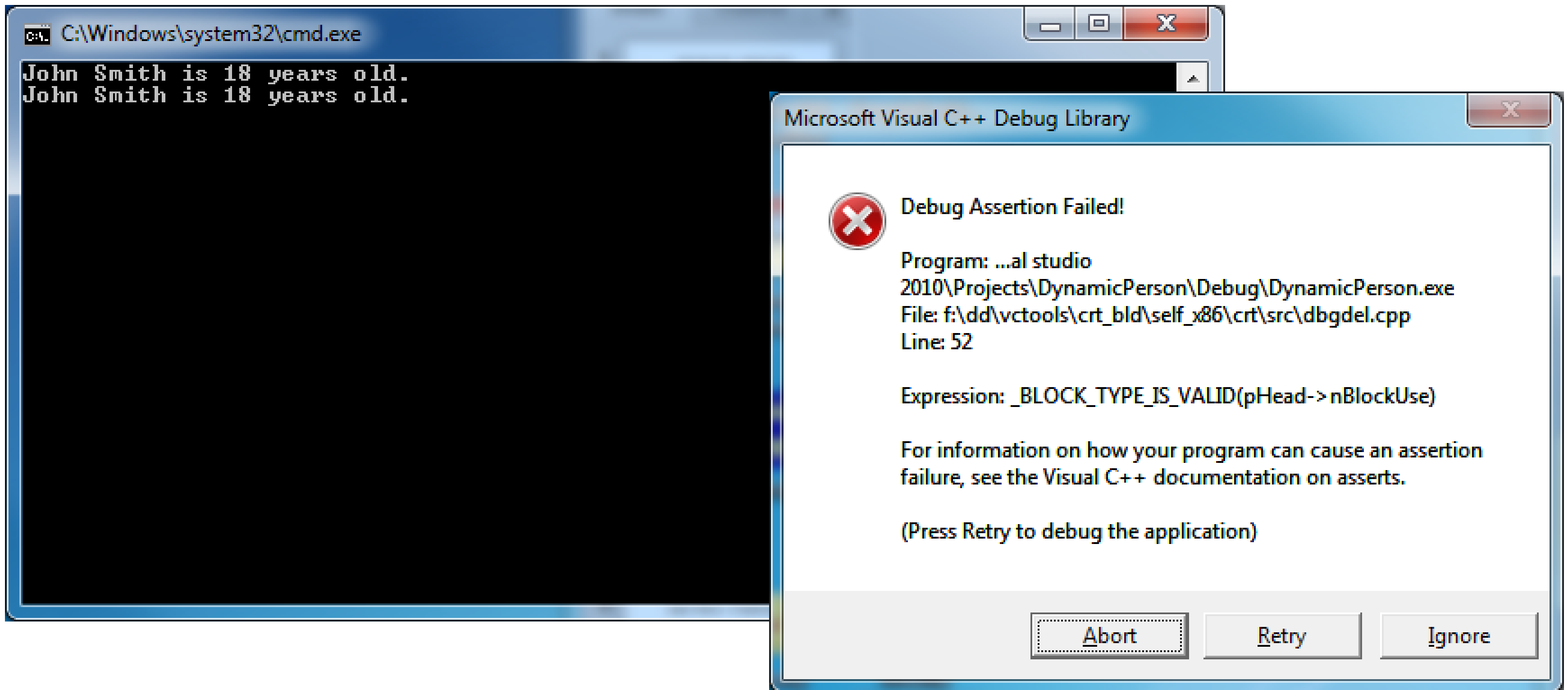
What happens?

- If we add the following code before the return, what happens?

```
cTempPerson = cPerson;

cout << cTempPerson.getName () << " is "
     << cTempPerson.getAge () << " years old."
     << endl;
```

Results … Why?



Problem Still Exists

- What is the difference?

```
Person cTempPerson = cPerson;
```

```
cTempPerson = cPerson
```

- What is the solution?
- Grab the CopyConstructor Solution in CS250 Public and let's make sure we understand this concept