**MSCF**

**Financial Computing I**

**Homework 1, for Lectures 1, 2, and 3**

***Due At 3:30 pm Thursday, Nov. 2, Before Lecture***

***You will lose 1 point every 10 minutes after that time***

1. **new/delete and Memory Leak (15 points)**

In this little program, we allocate very large arrays in a loop, but fail to **delete[]** them. The first time through the loop, all is well: we allocate a large array-of-**int**, a large array-of-**double**, and a large array-of-**char**. But the second time through, we allocate new large arrays of **int**, **double**, and **char**, and *lose track* of where the first ones were allocated. ***We have created a memory leak!*** This happens over and over, each time through the loop.

**// File: hw1.1.cpp**

**// Author(s):**

**#include <iostream>**

**#include <iomanip>**

**using namespace std;**

**int main()**

**{**

**for (int i(1); i <= 1000000; ++i) { // 1 million times**

**int \*pi = // "array" of int**

**new int[10000000]; // 10 million ints**

**double \*pd = // "array" of double**

**new double[10000000]; // 10 million doubles**

**char \*pc = // "array" of char**

**new char[10000000]; // 10 million chars**

**cout << "i: " << setw(7) << i << '\n';**

**// ... but no delete[] statements!**

**// Memory leak!**

**}**

**}**

1. Create a new, empty Win32 Console Application project named **hw1.1**. Download the file **hw1.1.cpp** and Add it as an Existing Item. ***Or***, add a New, empty source file named **hw1.1.cpp**, and copy-and-paste the program above into **hw1.1.cpp**. Now, compile and test. ***How many times*** can you go through the loop before the program crashes, or is aborted by Visual Studio? ***Make a note of this in a comment in your source code.***
2. Add the necessary **delete[]** statements in the **for** loop, so that each allocated array is properly deallocated. Compile and test. The program should no longer crash/abort. (You can simply close the output window, after you have gone well beyond the place where the program crashed/aborted before. You don’t need to wait for all one million iterations through the loop!) ***Change*** the **for** loop limit from **1000000** to **10**; compile and test.
3. Below the **for** loop, add code that:
   1. Declares a pointer variable of appropriate type, and dynamically allocates an **int** object initialized with **-123**
   2. Then displays the *value* of this object
   3. Then displays the *address* of this object (that is, where the int object is located in memory)
   4. And finally deallocates this object.
4. What is the value of the pointer from part (c) above, after you have deleted the object it points to? What happens if you delete this pointer again? What happens if you assign the value **1234** to the location the pointer is pointing to? If you don’t observe that anything happens, what could that mean? Comment out this code you have written for part (d) after your testing; compile and test again to confirm that all is well … or at least that all appears to be well!
5. Below the code you have written so far, add this code (or “uncomment” this code if you used the downloaded **hw1.1.cpp** file):

**for (int i(0); i < 100; ++i) {**

**cout << "before allocation " << i << "\n";**

**char \*pc = new char[500000000];**

**cout << "after allocation -- ";**

**if (pc == nullptr)**

**cout << "new returned nullptr\n";**

**else**

**cout << "new did NOT return nullptr\n";**

**}**

Compile and test. What happens and why?

1. Modify the code in part (e) so that **new** will ***not*** throw an exception (and crash your program) when it is unable to allocate the amount of memory requested. Compile and test. What happens now?
2. C++14 allows single quotes to be used as separators within numeric literals, which may improve readability. Each single quote is simply ignored. For example, **10'000'000** and **1'0'0'0000'0** are both equivalent to **10000000**. Try this out in your code; compile and test.
3. **struct, Arrays of struct, Pointers to struct (70 points)**

You must at all times maintain a simultaneous awareness of the *value* of an expression (or at least the possible value) and the *data type* of that expression. The data type of an expression determines how the value can be used in the code. (Recall **What You Should Know** on slide 25, Programming Prep Lecture 7.)

Arrays, structures, and pointers connecting among them are foundational to competent C++ programming. Here are the facts that we (should) know:

1) If **X** is an object in memory of type **T**, then

The value of **&X** is the address of object **X**

The data type of expression **&X** is *pointer-to* **T**

2) If **T** is a data type and **P** is of type *pointer-to* **T**, then

The value of **\*P** is the value of the **T**-type object at address **P**

The data type of expression **\*P** is **T**

3) If **T** is a data type and **P** is of type *pointer-to-***T**, and **I** is an integer, then

The value of **P + I** is the address of the object of type **T** that is **I \* sizeof T**

bytes further along in memory from **P**

***Unless*** **T** is **void**: pointer arithmetic is illegal on *pointer-to* **void**

**I** may be positive, negative, or zero

The data type of expression **P + I** is *pointer-to* **T**

4) If **T** is a data type, **N** is a positive integer, and **A** is an *array-of-***N** **T** then

**A** is equivalent to **&A[0]**

***Except*** in a few cases: **sizeof A**; range **for** loop; **&A**; **begin(A)**, **end(A)**

The value of **A** is the address of the initial element of array **A**

The data type of **A** can be treated as *pointer-to* **T**

5) If **P** is of type *pointer-to-***T**, and **I** is an integer, then **P[I]** is equivalent to

**\*(P + I)**

The value of **P[I]** is the value of the **T**-type object at address **P + I**

The data type of expression **P[I]** is **T**

6) If **S** is a **struct** type with a member named **mem** of type **M**, and **X** is an object

of type **S**, then **X.mem** is the **mem** member of object **X**

The value of **X.mem** is the value of the **M**-type **mem** member of **X**

The data type of expression **X.mem** is **M**

7) If **S** is a **struct** type with a member named **mem** of type **M**, and **P** is of type

*pointer-to* **S**, then **(\*P).mem** is equivalent to **P->mem**

The value of **(\*P).mem == P->mem** is the value of the **M**-type **mem**

member of **\*P**

The data type of expression **(\*P).mem == P->mem** is **M**

***Notice*** that **[]** and **->** (on the right) have highest precedence, **&** and **\*** (on the left) have lower precedence, and **+** has lowest precedence.

In effect, these are “axioms of algebra” for dealing with arrays and array elements, structures and structure members, and pointers and pointed-to objects. You don’t need to be able to write these rules down, but you ***do*** need to be able to apply these rules almost automatically, as you are able to do arithmetic and algebra automatically. (No interviewer is going to be too impressed if you know a concept but struggle to come up with the notation that implements the concept.)

This program creates, displays, and modifies a portfolio of stocks:

**// File: hw1.2.cpp**

**// Author(s):**

**#include <iostream>**

**#include <iomanip>**

**using namespace std;**

**struct Stock {**

**char ticker[8]; // leave some extra space**

**int id; // my firm's ID for this stock**

**double price;**

**};**

**struct Portfolio {**

**Stock \*stocks; // the "array" of Stocks**

**int \*num\_owned; // can sell short, so int**

**size\_t nstocks; // number of stocks, >= 0**

**};**

**void put\_Stock(Stock \*ps) {**

**cout << setw(8) << ps->ticker << setw(4) << (\*ps).id**

**<< setw(10) << ps[0].price;**

**}**

**int main()**

**{**

**Stock apple{ "AAPL", 0, 162.45 },**

**vmware{ "VMW", 1, 120.08 },**

**ibm{ "IBM", 2, 153.72 },**

**alcoa{ "AA", 3, 47.71 },**

**ford{ "F", 4, 12.07 };**

**Portfolio p1{ new Stock[3]{ apple, ibm, ford},**

**new int[3]{ 20, 10, -5 },**

**3 };**

**cout << "Portfolio:\n";**

**for (int i(0); i < p1.nstocks; ++i) {**

**put\_Stock(p1.stocks + i);**

**cout << setw(10) << p1.num\_owned[i] << '\n';**

**}**

**\*(p1.num\_owned + 2) -= 10;**

**cout << "\nModified portfolio:\n";**

**for (int i(0); i < p1.nstocks; ++i) {**

**put\_Stock(p1.stocks + i);**

**cout << setw(10) << p1.num\_owned[i] << '\n';**

**}**

**delete[] p1.stocks;**

**delete[] p1.num\_owned;**

**}**

1. Create a new, empty Win32 Console Application project named **hw1.2**. Download the file **hw1.2.cpp** and Add it as an Existing Item. ***Or***, add a New, empty source file named **hw1.2.cpp**, and copy-and-paste the program above into **hw1.2.cpp**. Compile and test; the program should work fine.
2. Add four or five more individual **Stock** objects of your choice to the program, and add three or four more of the **Stock** objects to the end of the initialization of the **Portfolio** object **p1**. Compile and test. If necessary, fix any bugs you may have introduced.
3. If a **Stock** object is large, then passing a copy of a **Stock** object as an argument to a function can be expensive. What is the *data type* of the argument passed into the **put\_Stock** function? (Add a comment with the answer to this in your code.)
4. The first time that **put\_Stock** is called in the first for loop, what *value* is passed as the argument to **put\_Stock**? (***Hint:*** When you can’t determine a specific answer, like 12.75, give the most accurate answer you can, like: “the value of the **ford** object”, “the address of the **vmware** object”, “a value in the possible range for **int** objects”, or “undefined”.)
5. Define a function named **put\_Portfolio\_and\_value**, that receives a *pointer-to* **Portfolio** as its argument, and then displays a nicely formatted table like this:

Stock ID Price Position Value

AAPL 0 162.45 20 3245.00

IBM 2 153.72 10 1537.20

Portfolio Total: 4782.20

1. Draw a diagram that shows how the **Stock** objects and their members and the **Portfolio** object and its members are organized. Label each object (and structure member) with its name and data type. Which objects are on the stack? In the static data area? On the heap? (Either scan or photograph your diagram so you can include a diagram file in your homework submission.)
2. What is the *value* of **apple.ticker**? Add the answer in a comment in your code. (Obviously you can compile and run this code to get a specific answer on your system, but give an English descriptive answer: this is what you will need to do on the quiz.)
3. What is the *data type* of **apple.ticker**?
4. What is are the *value* and *data type* of **apple.ticker[3]**?
5. What are the *value* and *data type* of **p1.stocks + 2**?
6. What are the *value* and *data type* of **sizeof ford.ticker**?
7. What are the *value* and *data type* of **sizeof vmware.price**?
8. What are the *value* and *data type* of **sizeof p1.num\_owned**?
9. What are the *value* and *data type* of **\*p1**? (Yes, this is a “trick” question, of a kind you might be asked at an interview.)
10. What are the *value* and *data type* of **(p1.stocks + 1)->name[3]**?
11. What are the *value* and *data type* of **(**\***p1.stocks). name + 3**?
12. What are the *value* and *data type* of **p1.stocks[2].id**?
13. What are the *value* and *data type* of **(\*&p1).num\_owned[1] - 3**?

Immediately following the declaration and initialization of p1, suppose we add these statements:

**Portfolio \*pp(&p1 + 2);**

**Stock \*ppf{p1.stocks + 1};**

1. (Example question: you don’t need to include an answer to this in your code) Write an expression using **pp** that yields the **price** of Apple stock:

(Example answers)

**pp[-2].stocks[0].price**

**(pp - 2)->stocks[0].price**

**pp[-2].stocks->price**

**(pp - 2)->stocks->price**

**(\*(pp - 2)).stocks[0].price**

**(\*(\*(pp - 2)).stocks).price**

(Some of these are common in code; others are uncommon; but all mean the

same thing.)

1. Write an expression (in a comment in your code) using **ppf** and **[]** that yields the **id** of Ford stock. Write a different but equivalent expression.
2. Write an expression using **ppf** and **->** that yields the **id** of IBM stock. Write a different but equivalent expression.
3. Write *six* different but equivalent expressions using **p1** that yield the *address* of the **'P'** character in Apple’s **ticker**.

This is the conclusion of what you need to turn more in with your homework. In preparation for the Lecture 2 Quiz, select any component of the data in this program, then see how many equivalent notations you can write that yield that component or the *address* of that component. Try this exercise with several components (including individual **char** objects within the **ticker**s). Make sure you can do examples involving P[i] vs. \*(P + i) and P->mem vs. (\*P).mem notations. You can, of course, code and test all your answers to check whether you are correct!

1. **The Inside-Out, Right-Then-Left Rule (15 points)**

You must be able to:

1. Create or look at a diagram representing an organization of data, and effortlessly write down an English description of the components, and the C++ declarations of the components.
2. Read an English description of the data type of an object, and effortlessly write down a diagram of the components of the object, and the C++ declaration of the data type of the object.
3. See a C++ declaration of an object, and effortlessly draw the diagram and write the English description.

Startlingly many professional (that is, paid) C++ programmers cannot do these things effortlessly … or much at all. You do not want to be one of these ignorant, not-very-productive hackers. On the Lecture 3 Quiz, you will need to answer 9 of 10 of these kinds of questions correctly.

**Example Questions** (you don’t need to turn these in):

1. Write the C++ declaration of an object named **X**, whose data type is *array-of-*5 *pointers-to* **Splot**. (Notice that it does not matter what a **Splot** is. It might be an **int**, or it might be a **CorporateEmpire**.)

**Splot \*X[5];**

1. Write the C++ declaration of an object named **X**, whose data type is *pointer-to* *array-of-7* *array-of-3* *pointer-to* *pointer-to* **Banjo**.

**Banjo \*\*(\*X)[7][3];**

1. Write the English description of this declaration:

**Moose \*(\*\*(\*X[3][9])[27])[4];**

**X** is an *array-of-*3 *array-of-*9 *pointer-to* *array-of-*27 *pointer-to* *pointer-to* *array-of-*4 *pointer-to* **Moose**

1. Given this diagram, give the English description and the C++ declaration for the object **X**:

**X**

Donut

Donut

Donut

**X** is an *array-of-*3 *pointer-to* *pointer-to* **Donut**.

**Donut \*\*X[3];**

**Required Questions** (type your answers in a text file named hw1.3.txt [or the like] and submit this file with the rest of your homework.

1. Write the English description of this C++ declaration of object **X**:

**Zebra \*\*\*(\*X[3][9][7])[14];**

1. Write the English description of this C++ declaration of object **X**:

**Stump (\*\*\*X[3])[12][8];**

1. Write the English description of this C++ declaration of object **X**:

**Toaster \*\*(\*(\*\*(\*X)[7])[12][5])[128][51];**

1. Write the C++ declaration of the object **X** whose data type is *array-of-*9 *pointer-to* *pointer-to* *pointer-to* *array-of-*64 *pointer-to* *pointer-to* *pointer-to* **Wombat**.
2. Write the C++ declaration of the object **X** whose data type is array-of-3 array-of-4 pointer-to array-of-17 array-of-2 pointer-to pointer-to array-of-9 **Laptop**.
3. Write the C++ declaration of the object **X** whose data type is pointer-to pointer-to array-of-13 pointer-to pointer-to array-of 5 array-of-6 pointer-to pointer-to **Bag**.

***REMEMBER*** to put all authors’ names into each of your source code files.Put your **hw1.1.cpp, hw1.2.cpp**, and **hw1.3.txt** files into a **.zip** archive and upload to the course web site.