Queens College, CUNY, Department of Computer Science
Object-Oriented Programming in C++
CSCI 211/611
Summer 2018
Instructor: Dr. Sateesh Mane

due date Friday, July 13, 2018, 11.59 pm

Homework: Pointers

- **Experience with other classes has demonstrated that in many cases the source of difficulty is not the mathematics or the programming.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

- **Consult your lab instructor for assistance.**
- **You may also contact me directly, but I cannot promise a prompt response.**
- Please submit your inquiry via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS211 or CS611):

     ```
     StudentId_first_last_CS211_hw_pointers.zip
     StudentId_first_last_CS611_hw_pointers.zip
     ```

  2. The archive should contain one "text file" named "hw_pointers.[txt/docx/pdf]" and one cpp file per question named "Q1.cpp" and "Q2.cpp" etc.
  3. Note that not all questions may require a cpp file.

General information

- You should include the following header files, to run the programs below.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cmath>
```

- If you require additional header files to do your work, feel free to include them.
- **Include the list of all header files you use, in your solution for each question.**
- The questions below do not require complicated mathematical calculations.
- If for any reason you require help with mathematical calculations, **ask the lab instructor or the lecturer.**

Q1 Pointers, dereferencing, etc.

- Write a program as follows and run it and print the outputs.
- Instantiate some variables and arrays and pointers as follows.

```
int main()
{
  int i = 3;
  int j = 4;

  int *p1 = &i;
  int *p2 = &j;

  //  see below

  return 0;
}
```

- **Print the values of i and `*p1`.** They should both be equal to 3.
- **Print the values of j and `*p2`.** They should both be equal to 4.
- **Next execute the following statement.**

```
*p1 = *p2;            // equivalent to i = j
```

- The above statement is equivalent to $i = j$, because `p1` points to i and `p2` points to j.

$$p_1 \longrightarrow i$$
$$p_2 \longrightarrow j$$

  1. Hence `*p1` is i and `*p2` is j, so we end up with $i = j$.
  2. **Print the values of i, j, `*p1` and `*p2`.**
  3. You should obtain $i = 4$, $j = 4$, `*p1` equals 4 and `*p2` equals 4.

- **Next execute the follosing statements.**

```
*p1 = j*2;            // equivalent to i = j*2
*p2 = j+3;            // equivalent to j = j+3
```

- As before, `p1` points to i and `p2` points to j, so `*p1` is i and `*p2` is j.

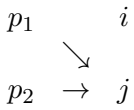
  1. Hence the first statement is equivalent to $i = j * 2$.
  2. The second statement is equivalent to $j = j + 3$.
  3. **Print the values of i, j, `*p1` and `*p2`.**
  4. You should obtain $i = 8$, $j = 7$, `*p1` equals 8 and `*p2` equals 7.

- **See next page(s).**

- **Next do something different.**

```
p1 = p2;
```

- **The above statement copies the memory address (not the values of i and j).**



1. **Both pointers point to j**, i.e. `p1 = &j` and `p2 = &j`.
2. **Nothing points to i anymore.**
3. **Print the values of i, j, `*p1` and `*p2`.**
4. You should obtain $i = 8$ and $j = 7$ (no change from above).
5. Because both pointers point to j, therefore `*p1` equals 7 and `*p2` equals 7.

Q2 Pointers, mix & match

- Write a program as follows and compile it.
- Instantiate some variables and arrays and pointers as follows.

```
int main()
{
  int i = 3;
  int j = 4;
  int a[] = {5, 6, 7};
  double x = 8.9;

  int *p1 = &i;
  int *p2 = &j;
  double *pd = NULL;

  //  see below

  return 0;
}
```

- **Write the following statement.**

```
pd = &x;
```

  1. **Compile the program.** It should compile successfully.
  2. Print the values of x and `*pd`. They are both equal to 8.9.

- **Next write the following statement.**

```
pd = &j;              // compiler error
```

  1. **This causes a compiler error.**
  2. **A pointer to `double` cannot point to the address of `int`.**

- *Comment out (or delete) the above statement.*
- **Next write the following statement.**

```
pd = p2;              // compiler error
```

  1. **This causes a compiler error.**
  2. **A pointer to `double` cannot copy the memory address from a pointer of `int`.**

- **In general, we cannot mix and match pointers of different data types.**

Q3 Pointers and arrays

- Write a program as follows and run it and print the outputs.
- Instantiate some variables and arrays and pointers as follows.

```
int main()
{
  int n = 3;
  int i = 3;
  int a[] = {5, 6, 7};
  int *p1 = NULL;

  //  see below

  return 0;
}
```

- **Write the following statement.**

```
p1 = a;
```

  1. **Compile the program.** It should compile successfully.
  2. A pointer to `int` can point to an array of type `int`.
  3. In C++, an array is implemented as a pointer to a block of memory.

- **Run a loop and print the values of `a[i]` and `p1[i]`.**

```
for (i = 0; i < n; ++i)  // etc
```

- The values of `a[i]` and `p1[i]` are equal.
- **Write the following statement.**

```
i = p1[2];
```

  1. **Print the values of `i`, `a[2]` and `p1[2]`.**
  2. Because `p[2]` is the same as `a[2]`, all the values are equal (to 7).

- **Run a loop and perform the following calculation.**

```
for (i = 0; i < n; ++i)
  p1[i] = -2 * p1[i];
```

  1. **Print the values of `a[i]` and `p[i]` for $0 \leq i < n$.**
  2. Because `p[i]` is the same as `a[i]`, the above calculation changes the value of `a[i]`.
  3. The values of `a[i]` and `p[i]` are equal to $-10, -12, -14$, for $i = 0, 1, 2$.

Q4 Pointers in function calls

- Write the following program and compile and run it.

```
// header files

void swap(int *u, int *v)                                                    // #1
{
  cout << "(b)  "  << (*u) << "   " << (*v) << endl;

  double tmp = *u;                 // save temporary value (dereference) // #2
  *u = *v;                         // copy dereference                   // #3
  *v = tmp;                        // assign dereference                 // #4

  cout << "(c)  "  << (*u) << "   " << (*v) << endl;                       // #5
}

int main()
{
  int i = 3;
  int j = 4;

  cout << "(a)  "  << i << "   " << j << endl;
  swap(&i, &j);                                                              // #6
  cout << "(d)  "  << i << "   " << j << endl;

  return 0;
}
```

- It was explained in the lectures how the above function swaps the values of two variables in the main program. (The example in the lectures employed pointers to `double`.)
- Run the program and print the outputs and note the values in each line of output.

1. **Explain the notation "`&i`" and "`&j`" in statement #6 in the main program.**
2. **#1: Explain the memory addresses to which "`u`" and "`v`" point in statement #1 in the function.**
3. **#2: What is the value of "`tmp`" in statement #2 (and why)?**
4. **#3: What variables are operated on in the statement "`*v`" to "`*u`" in statement #3?**
5. **#3: What number is copied from "`*v`" to "`*u`" in statement #3?**
6. **#4: What number is copied from "`tmp`" to "`*v`" in statement #4?**
7. **#5: What are the values of `i` and `j` (in the main program) when statement #5 is executed?**

Q5 Pointers and arrays in function calls

- Write the following program and compile and run it and print outputs.

```
// header files

void array_dbl(int n, int *u)                          // #1
{
  for (int i = 0; i < n; ++i) {
    u[i] = 2*u[i];
  }
}

int main()
{
  int n = 3;
  int a[] = {5, 6, 7};
  int *p1 = a;
  // #2                                                // #2
  array_dbl(n, a);
  // #3                                                // #3
  array_dbl(n, p1);
  // #4                                                // #4
  return 0;
}
```

- **Print the values of the array elements `a[i]`, $0 \leq i < n$ at location #2.**
- **#1: Explain the memory addresses to which "u" points in statement #1 in the function for the first function call.**
  1. Note that `u[i]` in the function is the same as `a[i]` in the main program.
  2. **Print the values of the array elements `a[i]`, $0 \leq i < n$ at location #3.**
  3. The value of `a[i]` should be multiplied by a factor of 2 from the previous output.
- **#1: Explain the memory addresses to which "u" points in statement #1 in the function for the second function call.**
  1. Note that `u[i]` in the function is the same as `p1[i]` in the main program, therefore the same as `a[i]` in the main program.
  2. **Print the values of the array elements `a[i]`, $0 \leq i < n$ at location #4.**
  3. The value of `a[i]` should again be multiplied by a factor of 2 from the previous output.
  4. Therefore the value of `a[i]` will be multiplied by a factor of 4 from its original value.

Q6 Pointers as return values of functions

- **The return value of a function can be a pointer.**
- Modify the input to the function in the lecture to a `const` vector.

  const double* `get_const_element(const vector<double> &v, int n);`

- **Explain why the function return value must be changed to a `const` pointer.**
- **Modify the main program to run correctly with the new function.**

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

const double* get_const_element(const vector<double> &v, int n)   // etc

int main()
{
  int len = 5;
  vector<double> v;

  for (int i = 0; i < len; ++i)
    v.push_back( i + 1.2 );

  for (int i = -1; i < len+2; ++i) {
    //  get_const_element(v, i);               // call the function correctly
    if return value == NULL                    // write this line correctly
      cout << "null, i = " << i << endl;
    else
      cout << *(return value) << endl;         // write this line correctly
  }

  return 0;
}
```

Q7 Dynamic memory for single item

- Write a program as follows and run it and print the outputs.
- Instantiate some variables and arrays and pointers as follows.

```
int main()
{
  double x = 8.9;
  double *pd = new double;                  // dynamic memory

  //  see below

  return 0;
}
```

- **Write the following statement.**

```
*pd = 10.2;
```

- *The dynamically allocated variable has no name. We access it only as* `*pd`.
- **Write a comparison test as follows.**

```
if (x < (*pd)) // etc
```

1. If `true`, print "`x < (*pd) true`" also the values of `x` and `*pd`.
2. If `false`, print "`x < (*pd) false`" also the values of `x` and `*pd`.
3. **It is also possible to write the test without the parentheses.**

   ```
   if (x < *pd) // etc
   ```

4. This is legal but can be confusing and lead to bugs.

- **Write the test in the opposite order.**

```
if (*pd > x) // etc
```

- The true/false outcome should be the same as for the first comparison test.
- **Release the memory for `pd` by calling operator `delete`.**

```
delete pd;
```

- **See next page(s).**

- Dynamically allocate memory for `pd` a second time by calling operator `new`.
  1. **Allocate the memory as follows.**

     ```
     pd = new int;                                    // compiler error
     ```

  2. **Observe that this causes a compiler error.**
  3. We cannot allocate memory of type "`new int`" to a pointer to `double`.
- ***Comment out the previous line.*** **Allocate fresh memoey correctly as follows.**

  ```
  pd = new double;
  ```

- **Write the following statement.**

  ```
  *pd = -6.7;
  ```

- *The variable* `*pd` *in this statement is different from the previous variable (recall* `*pd = 10.2`*).*
- The previous variable has been deleted and is out of scope.
- **Write another comparison test as follows.**

  ```
  if (x < (*pd)) // etc
  ```

  1. If `true`, print "`x < (*pd) true`" also the values of `x` and `*pd`.
  2. If `false`, print "`x < (*pd) false`" also the values of `x` and `*pd`.
- **Release the memory for `pd` by calling operator `delete`.**

Q8 Dynamic memory for array

- Write a program as follows and run it and print the outputs.
- **Write a function with the following signature.**

```
void sum_array(int j, const double *a, double &sum);
```

- Inside the function, write a loop to calculate the value of `sum` to the following value.

```
sum = a[0] + ... + a[j]
```

- Write a main program and instantiate variables and arrays and pointers as follows.

```
int main()
{
  int n = 5;
  double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
  double *pa = NULL;
  double *pb = NULL;

  //  see below
  return 0;
}
```

- **Dynamically allocate memory as follows.**

```
pa = new double[n];
pb = new double[n];
```

- **Important:**
  Explain why the following allocation statement compiles but does not do what we want.

```
pa = pb = new double[n];
```

- **See next page(s).**

- We set the value of `pa[i]` as follows, for $0 \leq i < n$.

$$\begin{aligned}
\texttt{pa}[0] &= d[0] \\
\texttt{pa}[1] &= d[0] + d[1] \\
\texttt{pa}[2] &= d[0] + d[1] + d[2] \\
\texttt{pa}[3] &= d[0] + d[1] + d[2] + d[3] \\
\texttt{pa}[4] &= d[0] + d[1] + d[2] + d[3] + d[4]
\end{aligned}$$

- **Write nested loops as follows.**

```
for (int i = 0; i < n; ++i) {
  pa[i] = 0;
  for (int j = 0; j <= i; ++j) {
    pa[i] = pa[i] + d[j];
  }
}
```

- Note that the value of `pa[i]` is the sum of `d[j]` for $0 \leq j \leq i$.
- *Make sure you understand how the nested loops work.*
- **Set the elements of `pb` to the same values but using a function call.**

```
  for (int i = 0; i < n; ++i)
    sum_array(i, d, pb[i]);          // set array element using function call
}
```

- Notice that `sum` in the function call is a reference to element `pb[i]`.
  1. The element `pb[i]` is a variable of type `double` and "`sum`" is a reference to `double`.
  2. Hence `sum` can be bound to `pb[i]`.
- **Print the values of `d[i]`, `pa[i]` and `pb[i]` for $0 \leq i < n$.**
- **Release the memory for `pa` and `pb` by calling the correct version of operator `delete`.**

Q9 Pointers and math operations (not for examination)

- **This question is of lower priority.**
- **There will be no questions on "pointers in math calculatons" in exams.**
- Write a program as follows and run it and print the outputs.
- Instantiate some variables and arrays and pointers as follows.

```
int main()
{
  int i = 6;
  int j = 3;
  int k = 0;
  int *p1 = &i;
  int *p2 = &j;

  //  see below

  return 0;
}
```

- **Write the following statements and run the program.**

```
k = (*p1) + (*p2);        // equivalent to k = i + j
cout << k << endl;

k = (*p1) - (*p2);        // equivalent to k = i - j
cout << k << endl;

k = (*p1) * (*p2);        // equivalent to k = i * j
cout << k << endl;

k = (*p1) / (*p2);        // equivalent to k = i / j
cout << k << endl;
```

- **Print the value of k in each case.**
- The above is how expressions involving pointers are written in mathematical calculations.
- *They can be very confusing to read.*

- **See next page(s).**

- *What is worse, the parentheses around* `(*p1)` *and* `(*p2)` *are optional.*
- **The following statements are legal in C++**
- **Write the following statements and run the program.**

```
k = *p1 + *p2;              // equivalent to k = i + j
cout << k << endl;

k = *p1 - *p2;              // equivalent to k = i - j
cout << k << endl;

k = *p1 * *p2;              // equivalent to k = i * j
cout << k << endl;

k = *p1 / *p2;              // equivalent to k = i / j
cout << k << endl;
```

- **Print the value of k in each case. The outputs are the same as before.**
- ***Code such as the above may be legal, but is very dangerous.***
- **I strongly advise against it.**
- It is difficult to read and understand and it is very easy to make mistakes.
- It costs nothing to write parentheses `(*p1)` and `(*p2)` to clarify the meanings of the symbols.
- **There will be no exam questions with nasty/confusing expressions such as `*p1 + *p2` or `*p1 - *p2` or `*p1 * *p2` or `*p1 / *p2`.**
- **Points will be deducted for students who submit code such as the above for solutions to exams/projects, *even if the code compiles and runs correctly.***