MEMORIAL UNIVERSITY
Faculty of Engineering and Applied Science

Lab Handout
ENGI3891

14,21 Oct 2015

Lab 2: Pointers

1 Purpose and outcomes

This lab will help you explore pointers and arrays. After completing this lab, you should:

1. understand C-style strings and their use in command-line arguments,
2. be familiar with four memory regions,
3. be comfortable using pointers to refer to values and
4. have an visceral appreciation of common pointer errors.

2 Procedure

Follow the instructions given below **on a Windows computer with MinGW**: Mac OS X, BSD and Linux may give very different results. Make a note of `<<output>>` and questions as usual.

2.1 Create a workspace

Create a directory for this lab, then fill it with some C++ source files using a text editor (Notepad++ is installed on the lab computers and is a sensible enough choice). You do not need to submit these files with your lab report.

Listing 1: lab2.h

```
#ifndef LAB2_H
#define LAB2_H

#include <ostream>

void exploreCodeSegment(std::ostream&);
void exploreDataSegment(std::ostream&);
void exploreHeap(std::ostream&);
void exploreStack(std::ostream&);

#endif
```

Listing 2: code.cpp

```
#include "lab2.h"

void exploreCodeSegment(std::ostream& out)
{
}
```

Listing 3: data.cpp

```
#include "lab2.h"

void exploreDataSegment(std::ostream& out)
{
}
```

Listing 4: heap.cpp

```
#include "lab2.h"

void exploreHeap(std::ostream& out)
{
}
```

Listing 5: stack.cpp

```
#include "lab2.h"

void exploreStack(std::ostream& out)
{
}
```

Listing 6: main.cpp

```
#include "lab2.h"

#include <iostream>
#include <string>

using namespace std;


int main(int argc, char *argv[])
{
        cout << "ENGI3891 Lab 2: pointers\n\n";
        return 0;
}
```

Listing 7: Makefile

```
CXXFLAGS=-std=c++11 -Wall

pointer-demo: main.o code.o data.o heap.o stack.o
        ${CXX} $^ -o $@

clean:
        rm -f *.o *.exe pointer-demo
```

2.2 Compile with `make`

`make` is a utility that automates the building of software, using the information contained in a `Makefile`. Instead of compiling and linking with the usual g++ commands, use `make` command to build the project, then run it:

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo
<<output>>
```

2.3 Handle command-line arguments

The `main` function takes two parameters, `argc` and `argv`. These parameters are part of how the operating system communicates with your program: when you run the command `g++ hello.cpp -o hello.exe`, the OS puts the strings `"g++"`, `"hello.cpp"`, `"-o"` and `"hello.exe"` into an array called `argv`, and the length of that array is `argc`.

The type of `argv` is `char*[]`, an array of pointers to `char`. Each pointer in `argv`, e.g., `argv[1]`, points to a *C-style string*, which is just an array of characters in memory with a `NULL` character (`'0'`) at the end (see figure 1).

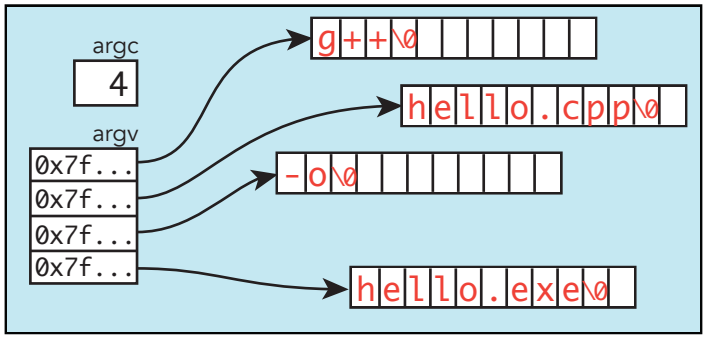

Figure 1: The `argv` array in memory.

We can print out a C-style string using the insertion operator, just like a C++ string:

```
std::ostream& operator << (std::ostream&, const char *);
```

Using this operator, modify `main.cpp` to print out the arguments passed to the `pointer-demo` program:

Listing 8: Rudimentary command-line argument handling

```
@@ -9,5 +9,12 @@ using namespace std;
 int main(int argc, char *argv[])
 {
        cout << "ENGI3891 Lab 2: pointers\n\n";
+
+       cout << "Received " << argc << " command-line arguments:\n";
+       for (int i = 0; i < argc; i++)
+               cout << "  " << i << ": '" << argv[i] << "'\n";
+
+       cout << "\n";
+
        return 0;
 }
```

Test `pointer-demo` with several invocations of command-line arguments:

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo
<<output>>
H:\engi3891\lab2>pointer-demo test
<<output>>
H:\engi3891\lab2>pointer-demo test1 test2
<<output>>
H:\engi3891\lab2>pointer-demo this is a test sentence
<<output>>
H:\engi3891\lab2>pointer-demo -I foo -x bar -z baz -o wibble
<<output>>
```

Finally, modify `main.cpp` to call our various demo functions based on command-line arguments:

Listing 9: More useful command-line argument handling

```
@@ -16,5 +16,25 @@ int main(int argc, char *argv[])

        cout << "\n";

+       for (int i = 1; i < argc; i++)
+       {
+               const string argument = argv[i];
+
+               if (argument == "code")
+                       exploreCodeSegment(cout);
+
+               else if (argument == "data")
+                       exploreDataSegment(cout);
+
+               else if (argument == "heap")
+                       exploreHeap(cout);
+
+               else if (argument == "stack")
+                       exploreStack(cout);
+
+               else
+                       cerr << "Invalid argument: '" << argument << "'\n";
+       }
+
        return 0;
 }
```

These functions don't do anything useful yet, as we have defined them to be empty (we have *stubbed out* the functions with the simplest possible implementation that will compile). Still, you can can run them:

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo stack data code heap invalid
<<output>>
```

3 Explore the data segment

Global variables are stored in the *data segment*, which is memory set aside when the operating system loads your program. Modify `data.cpp` to include global variables and print their addresses:

Listing 10: Data segment exploration

```
@@ -1,6 +1,16 @@
 #include "lab2.h"

+int globalInt1 = 42;
+int globalInt2 = 43;

 void exploreDataSegment(std::ostream& out)
 {
+       out
+               << "=====================================================\n"
+               << "Data segment (\"static store\"):\n"
+               << "=====================================================\n"
+               << "Address of globalInt1:  " << &globalInt1 << "\n"
+               << "Address of globalInt2:  " << &globalInt2 << "\n"
+               << "\n"
+               ;
 }
```

Re-compile the program and run it:

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo data
<<output>>
```

Question: *What is the difference (in bytes) between the addresses of* `globalInt1` *and* `globalInt2`*?*

Run `nm` to determine the offsets of these variables inside of the `pointer-demo.exe` executable file:

```
H:\engi3891\lab2>nm pointer-demo.exe | grep global | c++filt
<<output>>
```

Question: *How do these offsets compare with the addresses the variables occupy in memory?*

4 Explore the stack

Local variables are defined inside a function/method; they are so named because they are *local* to a particular invocation of that function or method. These variables are stored on the *stack*. Modify `stack.cpp` to print out some information about the stack and then run it:

Listing 11: Stack address exploration

```
@@ -1,6 +1,32 @@
 #include "lab2.h"


+void anotherFunction(std::ostream& out);
+
+
 void exploreStack(std::ostream& out)
 {
+       int a, b, c;
+
+       out
+               << "&a:   " << &a << "\n"
+               << "&b:   " << &b << "\n"
+               << "&c:   " << &c << "\n"
+               << "\n"
+               ;
+
+       anotherFunction(out);
+}
+
+
+void anotherFunction(std::ostream& out)
+{
+       int x, y, z;
+
+       out
+               << "&x:   " << &x << "\n"
+               << "&y:   " << &y << "\n"
+               << "&z:   " << &z << "\n"
+               << "\n"
+               ;
 }
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo stack
<<output>>
```

Question: *How far apart are* `a` *and* `b` *(or* `b` *and* `c`*,* `x` *and* `y`*, etc.)?*

Question: *How large is the gap between the* `a,b,c` *group of stack variables and the* `x,y,z` *group?*

4.1 Larger stack values

We can also put larger values on the stack. For instance, we can define a structure type for representing measurements of important atmostpheric properties, and we can instantiate an array of measurements on the stack:

Listing 12: Putting more data on the stack

```
@@ -4,14 +4,22 @@
 void anotherFunction(std::ostream& out);


+struct AtmosphericDataPoint
+{
+       double x, y, z, temperature, pressure;
+};
+
+
 void exploreStack(std::ostream& out)
 {
        int a, b, c;
+       AtmosphericDataPoint data[10];

        out
                << "&a:   " << &a << "\n"
                << "&b:   " << &b << "\n"
                << "&c:   " << &c << "\n"
+               << "data: " << data << "\n"
                << "\n"
                ;
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo stack
<<output>>
```

Question: *What is the new gap between* `a,b,c` *and* `x,y,z`*?*

Question: *Therefore, how large must* `data` *be?*

We can confirm these measurements by asking the compiler how large it thinks `data` ought to be:

Listing 13: Using `sizeof()` on the stack

```
@@ -20,6 +20,8 @@ void exploreStack(std::ostream& out)
                << "&b:   " << &b << "\n"
                << "&c:   " << &c << "\n"
                << "data: " << data << "\n"
+               << "size of data:     " << sizeof(data) << "\n"
+               << "size of element:  " << sizeof(data[0]) << "\n"
                << "\n"
                ;
```

```
H:\engi3891\lab2>make
```

```
<<output>>
H:\engi3891\lab2>pointer-demo stack
<<output>>
```

Question: *How large does the compiler (via* `sizeof`*) report that* `data` *is?*

4.2 Dangling pointers

Now, modify `stack.cpp` to return a pointer to a stack variable from a function:

Listing 14: Using `sizeof()` on the stack

```
@@ -10,9 +10,15 @@ struct AtmosphericDataPoint
 };


+int* address_of(int x)
+{
+       return &x;
+}
+
+
 void exploreStack(std::ostream& out)
 {
-       int a, b, c;
+       int a = 1, b, c;
        AtmosphericDataPoint data[10];

        out
@@ -26,6 +32,19 @@ void exploreStack(std::ostream& out)
                ;

        anotherFunction(out);
+
+       int *aptr = address_of(a);
+
+       out
+               << "&a               " << &a << "\n"
+               << "address_of(a):   " << aptr << "\n"
+               << "a:               " << a << "\n"
+               << "*address_of(a): " << *aptr << "\n"
+               << "\n";
+
+       anotherFunction(out);
+
+       out << "*address_of(a): " << *aptr << "\n";
 }
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo stack
<<output>>
```

Question: *Does* `address_of(a)` *return the same value as* `&a`*?*

Question: *Where does* `aptr` *point?*

Question: *Why does dereferencing* `aptr` *yield different values on lines 42 and 47 of* `stack.cpp`*?*

4.3 Addresses of references

Modify `stack.cpp` once more so that the `address_of` function has a reference to an integer as a parameter rather than a copy of one:

Listing 15: Using `sizeof()` on the stack

```
@@ -10,7 +10,7 @@ struct AtmosphericDataPoint
 };


-int* address_of(int x)
+int* address_of(int& x)
 {
        return &x;
 }
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo stack
<<output>>
```

Question: *How do* `&a` *and* `aptr` *compare now?*

Question: *Why does dereferencing* `aptr` *now yield the same result both times it occurs?*

5 Explore the heap

Modify `heap.cpp` to allocate a matrix of double-precision floating-point numbers and print its address:

Listing 16: Heap exploration

```
@@ -3,4 +3,12 @@

 void exploreHeap(std::ostream& out)
 {
+       constexpr size_t MatrixSize = 10000;
+       double *matrix = new double[MatrixSize * MatrixSize];
+
+       out
+               << "matrix:             " << matrix << "\n"
+               << "sizeof(matrix):     " << sizeof(matrix) << "\n"
+               << "sizeof(matrix[0]):  " << sizeof(matrix[0]) << "\n"
+               ;
 }
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo heap
<<output>>
```

Question: *How large is an element of* `matrix`*?*

Question: *Why is* `sizeof(matrix)` *so small?*

Modify `heap.cpp` to de-allocate the memory it used for the matrix:

Listing 17: Deleting heap memory

```
@@ -11,4 +11,10 @@ void exploreHeap(std::ostream& out)
                << "sizeof(matrix):     " << sizeof(matrix) << "\n"
                << "sizeof(matrix[0]):  " << sizeof(matrix[0]) << "\n"
                ;
+
+       delete [] matrix;
+
+       // do some more work
+
+       double x = *matrix;
 }
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo heap
<<output>>
```

Question: *Oops, we dereferenced de-allocated memory! What is the result? What happens if we double-delete instead?*

6 Explore the code segment

Modify `code.cpp` to print out the addresses of the `exploreDataSegment` and `exploreHeap` functions:

Listing 18: Code segment exploration

```
@@ -3,4 +3,12 @@

 void exploreCodeSegment(std::ostream& out)
 {
+       void *dataFunction = reinterpret_cast<void*>(exploreDataSegment);
+       void *heapFunction = reinterpret_cast<void*>(exploreHeap);
+
+       out
+               << "exploreDataSegment:  " << dataFunction << "\n"
+               << "exploreHeap:         " << heapFunction << "\n"
+               << "\n"
+               ;
 }
```

```
H:\engi3891\lab2>make
<<output>>
H:\engi3891\lab2>pointer-demo code
<<output>>
```

Run `nm` to determine the offsets of these functions inside of the `pointer-demo.exe` executable file:

```
H:\engi3891\lab2>nm pointer-demo.exe | grep explore | c++filt
<<output>>
```

Question: *How do these offsets compare with the addresses the variables occupy in memory?*

7 Postamble

Lab reports are due one week after the lab, at the beginning of that day's lecture. Unlike the last lab, you will submit this week's lab in one partner's assignments directory on the Subversion repository: you have no further need of the `engi3891/lab0/groupXX` directories. Since there will only be one copy per lab group, please ensure that you have the following line at the very top of your report:

```
# Lab report 1: <<username>> (<<student id>>) and <<username>> (<<student id>>)
```