# COSC 2430 Spring 2021

# Pointers and Arrays

## 1. Pointers

### 1.1. Memory regions

In C++, each variable is stored in a region of memory. The size of the region depends on the type of the variable. For example, a `char` variable takes 1 byte, an `int` typically takes 4 bytes, and so on. To see the of the memory regions needed for various types, you can try the following

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

int main()
{
    cout << "sizeof(char)     = " << sizeof(char) << endl;
    cout << "sizeof(char&)    = " << sizeof(char&) << endl;
    cout << "sizeof(int)      = " << sizeof(int) << endl;
    cout << "sizeof(int&)     = " << sizeof(int&) << endl;
    cout << "sizeof(long int) = " << sizeof(long int) << endl;
    cout << "sizeof(bool)     = " << sizeof(bool) << endl;
    cout << "sizeof(float)    = " << sizeof(float) << endl;
    cout << "sizeof(double)   = " << sizeof(double) << endl;
    cout << "sizeof(string)   = " << sizeof(string) << endl;
    return 0;
}
```

An `int`, for example, is stored in 4 consecutive bytes somewhere in memory. When the variable is changed, or if its value needs to be accessed, (some of) these four bytes will change their values, or their values will be read. We can, in fact, print out the address of a variable using the "address of" operator `&`.

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

int main()
{
    int a = 12345;
    // the following prints something like 0x7fff6425d7c4
    cout << "Address of a is at: " << &a << endl;
    return 0;
}
```

**A side note**: what might be a little surprising is that `sizeof(string)` is fixed, even though a string can be very long. You can test how long a string can be in your computer's architecture by

```
cout << "max_size: " << string().max_size() << "\n";
```

I'm using a 64-bit machine, and I got

```
max_size: 4611686018427387897
```

## 1.2. Pointers

Pointers are extremely powerful! Too powerful to the point that some other languages such as Java deliberately "hide" underlying pointer operations from the programmers. In this "beginner" lecture we won't be able to discuss every possible use/abuse of pointers. A *pointer* is a variable that holds a memory address. Hence, in a 32-bit architecture a pointer is 4 bytes long, and in a 64-bit architecture a pointer is 8 bytes long. Typically, we want a pointer to "point to" the starting address of a memory region holding a particular data type. And thus, we often speak of "a pointer to `int`", "a pointer to `string`", and so forth. For example,

```
int *i_ptr;    // i_ptr is a pointer to an integer
char *c_ptr;   // c_ptr is a pointer to a character
string *s_ptr; // s_ptr is a pointer to a string
```

How do I know the number of bytes a pointer variable occupies? Easy:

```
cout << "sizeof(i_ptr)    = " << sizeof(char*) << endl;
cout << "sizeof(char*)    = " << sizeof(char*) << endl;
cout << "sizeof(int*)     = " << sizeof(int*) << endl;
cout << "sizeof(string*)  = " << sizeof(string*) << endl;
```

If we define a pointer as above without initializing it, some compilers will assign the zero value to them, some will leave garbage values. To make a pointer points to the memory region holding some data type, we can use the "address of" operator **&**

```
int x = 10;
int *i_ptr;

i_ptr = &x;
// now i_ptr points to the first byte of the 4 bytes long x.
```

We can access the actual variable pointed to by the pointer using the *dereferencing* operator *. Continuing with the above code, we can do

```
cout << "x = " << x << endl;      // this prints x = 10
cout << "x = " << *i_ptr << endl; // this also prints x = 10
*i_ptr = 20;
cout << "x = " << x << endl;      // this prints x = 20
cout << "x = " << *i_ptr << endl; // this also prints x = 20
```

Thus, in a sense `*i_ptr` is a reference to `x`. But, unlike references, pointers can be reassigned to refer to a different location. Again, continuing with the above example, we can do

```
int y = 30;
i_ptr = &y;
cout << "y = " << *i_ptr << endl; // this prints y = 30
*i_ptr = 40
cout << "y = " << y << endl;      // this prints y = 40
```

Since dereferencing a pointer gives us a reference to the actual value, *we can attain the "pass-by-reference" effect by passing pointers by value*! For example:

```
// pt.cpp: testing pointers
#include <iostream>

using namespace std;

void my_swap(int *a, int *b) { int temp=*a; *a=*b; *b=temp; }

int main() {
    int x = 1, y=9;
    my_swap(&x, &y);
    cout << "x = " << x << endl; // x = 9
    cout << "y = " << y << endl; // y = 1
    return 0;
}
```

Note that the pointers `a,b` are passed by value: they store the addresses of where `x` and `y` stay in the main body. Inside the `my_swap` function we did not change the values of the pointers, we changed the values of what the pointers point to! You should read the above sentence again: "we can attain pass-by-reference effect by passing pointers by value". Also note that we cannot pass `x,y` to `my_swap()`, we have to pass the addresses of `x` and `y`.

> ***GPH: If we do indeed intend for a function to modify its argument(s), it is often a good programming habit to declare the arguments as pointers to the objects we want to modify***. *The reason is that a user of your function (a fellow programmer) will be forced to put the "address of" operator* `&` *in front before passing an argument to the function — and as s/he does so s/he fully understands that there might be modifications to the value referred to by the pointer.*

**Pointers to objects**

We will often use pointers to objects, e.g., pointers to strings. To access a data member or a function member (i.e. method) of an object, we can use the syntax `(*obj_ptr).member`, but we can also use the syntax `obj->member`. For example,

```
#include <iostream>
#include <string>
using namespace std;  // BAD PRACTICE

void print_reversed_sentence(const string*);

int main()
{
    string line;

    while (true) {
        cout << "> ";
        if (getline(cin, line)) {
            print_reversed_sentence(&line);
        } else {
            cout << "Ctrl-Z/D pressed, Bye Bye" << endl;
            return 0;
        }
    }
    return 0;
}

// this is clumsy, just wanted to illustrate pointer to object
void print_reversed_sentence(const string* s_ptr)
{
    int start;
    int end = s_ptr->length()-1;
    for (start = s_ptr->length()-1; start>=0; start--) {
        if ( ( (*s_ptr)[start] == ' ') && (start < end) ) {
            cout << s_ptr->substr(start+1,end-start) << ' ';
            end = start-1;
        }
    }

    if (start < end)
        cout << s_ptr->substr(start+1,end-start+1) << endl;
}
```

Note that `s_ptr->[start]` does not compile! This is an exception. Note also that the pointer `s_ptr` was passed as a **pointer to a constant object**, which means we cannot modify the object by doing things like `*s_ptr = "new string"`.

**We can have pointers to pointers (to pointers to pointers ad infinitum)**

For example, we might want to swap two pointers (instead of swaping the values pointed to by the pointers). For example:

```
#include <iostream>

using namespace std; // BAD PRACTICE

void my_swap(string **a, string **b)
{
    string* temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    string first("David");
    string last("Blaine");
    string* p1 = &first;
    string* p2 = &last;

    my_swap(&p1, &p2);

    cout << "p1 points to " << *p1 << endl; // "Blaine"
    cout << "p2 points to " << *p2 << endl; // "David"
    return 0;
}
```

## 2. Arrays

rrays we are discussing here are C-style arrays. In the newest C++ standard (C++11) //en.wikipedia.org/wiki/C%2B%2B11) there is also a `std::array` class (http://en.cppreference.com/w/cpp/container/array), which is a light weight wrapper class for C-style arrays.

### 2.1. Defining arrays

For a given type `T`, we can use `T arr[s]` to define an array `arr` of size `s` elements of type `T`; the ISO C++ standard (http://www.open-std.org/jtc1/sc22/wg21/docs/standards) requires that `s` has to be a constant expression. For example,

```
// array_test.cpp
#include <iostream>
using namespace std; // BAD PRACTICE

int main()
{
    const size_t s = 5;
    int A[s];
    int B[5] = {1, 2, 3, 4, 5};
    int C[] = {1, 2, 3, 4, 5};  // the same as saying int C[5] ...

    for (size_t i=0; i<s; i++) {
        A[i] = i*i;
        B[i] += A[i];
        C[i] += B[i];
    }

    for (size_t i=0; i<s; i++)
        cout << C[i] << ' ';

    cout << endl;
    return 0;
}
```

What is iffy is that some compiler (such as our `g++` version 4 or more) allows for an extension (http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Variable-Length.html#Variable-Length) of C++ where defining an array with variable length size is OK.

```
int s = 5;
int A[s];   // should not be allowed, but OK with g++ 4.x
```

However, doing so is not guarantted to work for all C++ compilers; hence, it is not recommended. To make the compiler more conforming to the standards, we can ask it to be more "pedantic" with the `-pedantic` option:

```
g++ -pedantic -ansi test_array.cpp

...
error: ISO C++ forbids variable-size array ‘A’
```

If you want to use variable-length arrays, use `vector`. We introduce `vector` type below.

## 2.2. Arrays and C-style strings

A C-style string (http://www.cprogramming.com/tutorial/lesson9.html) is a character array terminated by a `'\0'` byte (also called the NULL byte). (Some people call this the most expensive one-byte mistake (http://queue.acm.org/detail.cfm?id=2010365). But things are always "obvious" in hind-sight. We might just want to read the reasoning behind this decision (http://cm.bell-labs.com/cm/cs/who/dmr/chist.html) directly from Dennis Richie.) For example,

```
char name[] = "David";         // name has 6 elements, the last is implicitly '
cout << sizeof(name) << endl; // prints 6

int i=0;
while (name[i] != '\0')
    cout << name[i++];
cout << endl;
cout << name << endl;

char name[5] = "David";        // compilation error
```

We can also define the character array element-wise explicitly

```
#include <iostream>
using namespace std; // BAD PRACTICE

int main()
{
    char name[] = "David";
    char another[] = { 'D', 'a', 'v', 'i', 'd', '\0' };

    cout << name << endl;
    cout << another << endl;

    return 0;
}
```

— xpression `"David"` is called a string literal (http://msdn.microsoft.com/en-rary/69ze775t(v=vs.80).aspx), which is an array of characters as described above. Long string ls can be written using the following syntax:

```
char name[] = "This is a very long "
              "name and thus won't fit "
              "on a line";
cout << name; // get "This is a very long name and thus won't fit in a line"
```

We certainly can have a character array with many `'\0'` characters in the middle. We might even want them for some purposes. But, be careful when you print such an array out to the screen (for debugging, or something), in that case only the letters up to the first occurrence of `'\0'` are printed. Don't be surprised!

```
char sa[] = "Only up to here\0The rest can still be printed";
cout << sa << endl;

for (i=0; i< sizeof(sa); i++) {
    if (sa[i] != '\0') cout << sa[i];
    else cout << "[NULL CHAR]";
}
cout << endl;
```

Similarly, if you assign a string literal to a `string` object, the string object will be initialized with characters before the first NULL character.

```
string str_obj = sa;
cout << str_obj;     // prints "Only up to here"
```

*Why do we care about C-style strings?* C-style strings do not give any explicit method for enlarging, bound-checking, concatenating, finding substrings, and many other convenient member functions that a C++ string offers. Well, precisely because a C++ string is powerful, it takes more computational resources to operate them. In areas that involve system programming (such as when you take CSE 421 — OS — or CSE 489 — Networking), we often need very fine control of every single byte. In such cases operating on "low-level" C-strings and arrays are necessary. In fact, the `main()` function, and some file IO functions take C-style strings as arguments (see below).

### 2.3. Multidimensional arrays

A multidimensional array can be defined and initialized in the natural way:

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

int main()
{
    const int m=2;
    const int n=3;
    int A[m][n] = { {1, 2, 3}, {4, 5, 6} };

    // initialization must have bounds for all dimensions, except the first
    int B[][n] = { {10, 20, 30}, {40, 50, 60} };
    int C[m][n];
    int i, j;

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            C[i][j] = A[i][j] + B[i][j];

    for (i=0; i<m; i++) {
        for (j=0; j<n; j++) {
            cout << setw(2) << C[i][j] << ' ';
        }
        cout << endl;
    }

    return 0;
}
```

## 3. Arrays and Pointers

Pointers and arrays in C++ are closely related. Let me first try to list ways in which they are similar.

- First, an array's name can be used as a pointer to the first element of the array. For example

```
int a[5]   = {1,2,3,4,5};
int* i_ptr = &a[0]; // i_ptr points to a[0]
i_ptr      = a;     // i_ptr points to a[0], equivalent to the above line
*i_ptr     = 10;    // now a[0] == 10
```

- Second, we can navigate an array using *pointer arithmetic*. The idea is as follows. An array is simply a contiguous block of memory that holds the elements in the array. For example, if `sizeof(int)==4` then `int a[5]` is a contiguous block of 20 bytes in memory. The element `a[0]` occupies the first 4 bytes, `a[1]` occupies the next 4 bytes, and so on. Now, when we have a pointer `i_ptr` pointing to an `int`, the expression `i_ptr + 1` is a pointer pointing to the next integer after the integer that `i_ptr` points to. Thus, if `i_ptr == a` then `i_ptr+1` points to `a[1]`. We can test this pointer arithmetic by

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

int main()
{
    int a = 123;
    int* int_ptr = &a;
    cout << "+0:" << int_ptr << endl;
    cout << "+1:" << int_ptr+1 << endl;
    cout << "+2:" << int_ptr+2 << endl;

    return 0;
}
```

The program prints something like

```
+0:0x7fff6a9387c4
+1:0x7fff6a9387c8
+2:0x7fff6a9387cc
```

You can see that `int_ptr+1` is 4 more than `int_ptr`. In general, if `ptr` is a pointer to `T`, then `ptr+n` has integral value equal to `ptr+n*sizeof(T)`, where `n` is an arbitrary integer (could be negative). We thus can apply the arithmetic operators `++, --, +, -, +=, -=` to pointers (and integers). We can substract two pointers to the same type: that returns the number of elements of the type can be stored between the two pointers. But we cannot add, multiply, or divide two pointers. Those operations are meaningless. With pointer arithmetic, navigating an array can be done as follows.

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

int main()
{
    int A[5] = {1, 2, 3, 4, 5};
    int i;
    int* int_ptr;

    // we can traverse A like this
    for (i=0; i<5; i++)
        cout << A[i] << ' ';
    cout << endl;

    // or like this
    for (int_ptr=A; int_ptr != A+5; int_ptr++)
        cout << *int_ptr << ' ';
    cout << endl;

    return 0;
}
```

- Pointers and arrays can pretty much be used interchangeably in argument passing:

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

void ps1(char* s)
{
    while (*s != '\0') { cout << *s; s++; }
}

void ps2(char s[])
{
    while (*s != '\0') { cout << *s; s++; }
}

int main()
{
    char* s1 = new char[7]; // create dynamically an array of 7 chars
    char s2[] = "abcde\n";
    int i=0;
    while (s2[i] != '\0') { s1[i] = s2[i]; i++; }

    ps1(s1); ps1(s2); // valid
    ps2(s1); ps2(s2); // also valid

    delete [] s1; // release the mem. space allocated for the array, s1 is
    return 0;
}
```

Note the use of the new and delete operators for dynamic memory managment (http://www.cplusplus.com/doc/tutorial/dynamic/).

Next, here are some ways in which arrays are different from pointers.

- The array name can be thought of as a *constant pointer*.

```
1  int a[5];
2  int* i_ptr;
3  i_ptr = a; // perfectly fine!
4  a = i_ptr; // compilation error!
```

- When we define an array, its size has to be given. There's certainly no notion of "array size" when we use pointers.

## 4. Command line arguments

There are at least four valid prototypes for the main functions.

```
1  int main()
2  int main(void)
3  int main(int argc, char **argv)
4  int main(int argc, char *argv[]) // argv is an array of pointers to char*
```

We have been using the first. The last two are pretty much the same: `argc` counts the number of command line parameters plus 1. For example, if our program is named `myprog` and was invoked with `myprog one another`, then `argc==3` and `argv` is a multidimensional array of characters where `argv = ["myprog", "one", "another", 0]`: `argv[0] == "myprog"`, `argv[1] == "one"`, `argv[2] == "another"`, and `argv[3] == 0`. For example, we can write a program that takes two file names and the content of one (text) file to the other as follows.

```
// mycopy.cpp
#include <iostream>
#include <cassert>  // for the assert() macro
#include <cstdlib>  // for exit()
#include <fstream>  // for file IO

using namespace std; // BAD PRACTICE

int main(int argc, char* argv[])
{
    if (argc != 3) {
        cerr << "Usage: mycopy from_file to_file\n";
        exit(1); // this indicates an "on error" exit status
    }

    assert(argv[3] == 0); // exit the program if this is not the case, need -g

    cout << "Copying " << argv[1] << " to " << argv[2] << endl;

    string line;
    ifstream ifs(argv[1]); // this *MUST* take C-style string arg
    ofstream ofs(argv[2]); // this *MUST* take C-style string arg

    if (ifs && ofs) {
        while (getline(ifs, line)) {
            ofs << line;
            if (!ifs.eof()) ofs << '\n';
        }
    } else {
        cerr << "Failed to open " << argv[1] << " or " << argv[2] << endl;
        exit(1);
    }
    return 0;
}
```

Note the use of fstream objects (http://en.cppreference.com/w/cpp/io/basic_fstream), which are similar to the `iostream` objects but characters are read and writen to files. Also, note the use of the `assert()` macro (http://stackoverflow.com/questions/1571340/what-is-the-assert-function), which assert that some expression is true. The program will quit otherwise. The use of assertions is only done in the "debugging" mode when we are developing our product. For `assert()` to take effect, we must compile the program with the `-g` flag:

```
g++ -g mycopy.cpp
```

It is a good idea to put the `-pedantic -ansi -g` flags in your `Makefile`.

## 5. A few more pointer examples

The first example uses a pointer to pointer to print out the address of a variable. If your machine is a 32-bit machine, replace 8 by 4.

```cpp
// an example of pointer to pointer
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int a = 123;
    int* int_ptr = &a;
    uint8_t* byte_ptr = reinterpret_cast<uint8_t*>(&int_ptr);
    cout << "Address of a is at: " << &a << endl;
    cout << "Address of a is at: " << int_ptr << endl;


    cout << "Address of a is at: 0x";
    for (int i=8; i>0; i--) {
        cout << hex << setfill('0') << setw(2) << (int) byte_ptr[i-1];
    }
    cout << endl;

    return 0;
}
```

The second example illustrate two ways in which we can "return" an array from a function. We will write a program which reads a given file and prints out the character frequencies: for each alphabetic letter we print the number of occurrences of the character. (There are 26 characters, case-insensitive. Their ASCII codes are `'a' == 97`, ..., `'z' = 122`.

```cpp
// freq.cpp
// ~~~~~~~~
// author: hqn
// - read a file, get a vector of character frequency
// = this is only to illustrate some concepts, we will have a
//   much better solution using map
#include <iostream>
#include <fstream>
#include <cassert> // for assert() macro
#include <cstdlib> // for exit()
#include <cctype>  // for tolower()

using namespace std; // BAD PRACTICE

const int NO_CHARS = 26;

void compute_char_freqs1(ifstream&, int**);

// read characters from ifs, record frequencies in the newly
// allocated array pointed to by the returned pointer
int* compute_char_freqs2(ifstream&);

//  print 26 character frequencies
void print_freqs(int*);

int main(int argc, char* argv[])
{
```

```
28      if (argc != 2) {
29          cerr << "Usage: freq filename\n";
30          exit(1);
31      }
32      ifstream ifs;
33      ifs.open(argv[1]); // this is another way of explicitly openning the file
34
35      if (!ifs) { // failed to open the file
36          cerr << "Failed to open " << argv[1] << " for reading\n";
37          exit(1);
38      } else {
39          int* freq_array;
40          // compute_char_freqs1(ifs, &freq_array);
41          freq_array = compute_char_freqs2(ifs);
42          print_freqs(freq_array);
43          delete[] freq_array; // important!
44          ifs.close();
45      }
46
47      return 0;
48  }
49
50
51  void print_freqs(int* fa)
52  {
53      for (int i=0; i<NO_CHARS; i++) {
54          if (fa[i] != 0)
55              cout << "[" << char(i+'a') << ", " << fa[i] << "] ";
56      }
57      cout << endl;
58  }
59
60
61  int* compute_char_freqs2(ifstream &ifs)
62  {
63      int* fa = new int[NO_CHARS]; // frequency array
64      assert(fa != 0);
65
66      int i;
67      for (i=0; i<NO_CHARS; i++) fa[i] = 0;
68
69      char c;
70      while (!ifs.eof()) {
71          ifs.get(c);
72          if (isalpha(c)) {
73              c = tolower(c);
74              fa[c-'a']++;
75          }
76
77      }
78      return fa;
79  }
80
81  /**
82   * ------------------------------------------------------------------
83   *  read characters from ifs, record frequencies in the newly allocated array
84   *  pointed to by
```

```
 *  Question: compute_char_freqs2() seems to be cleaner, when do we want to
 *  use this method in compute_char_freqs1()?
 * ------------------------------------------------------------------------
 */
void compute_char_freqs1(ifstream &ifs, int** ret)
{
    int* fa = new int[NO_CHARS]; // frequency array
    assert(fa != 0);

    int i;
    for (i=0; i<NO_CHARS; i++) fa[i] = 0;

    char c;
    while (!ifs.eof()) {
        ifs.get(c);
        if (isalpha(c)) {
            c = tolower(c);
            fa[c-'a']++;
        }

    }
    *ret = fa;
}
```