Managing Memory (and low level Data Structures)

Lectures 22, 23

Hartmut Kaiser

hkaiser@cct.lsu.edu

http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Programming Principle of the Day

• Avoid Premature Optimization
  • Don’t even think about optimization unless your code is working, but slower than you want. Only then should you start thinking about optimizing, and then only with the aid of empirical data.
  • "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" - Donald Knuth.

http://en.wikipedia.org/wiki/Program_optimization
Low Level Data Structures

• We were using the standard library data structures – containers
  • How are these built?
    • Low level language facilities and data structures
    • Closer to computer hardware (in semantics and abstraction level)
  • Why do we need to know how are these built?
    • Useful techniques, applicable in other contexts
    • More dangerous, require solid understanding
    • Sometimes absolute performance matters
Pointers and Arrays

- Array is a kind of container, however it’s less powerful and more dangerous
- Pointers are kind of random access iterators for accessing elements of arrays
- Pointers and Arrays are the most primitive data structures available in C/C++
  - Closely connected concepts, inseparable in real world usage
**Pointers**

• A pointer is a value representing the *address* of an object
  • Every distinct object has a distinct address denoting the place in memory it lives
  • If it’s possible to access an object it’s possible to retrieve its address

• For instance:

```
x    // if ‘x’ is an object
&x   // then ‘&x’ is the address of this object
p    // if ‘p’ is the address of an object
*p   // then ‘*p’ is the object itself
```
**Pointers**

- The ‘&’ is the address-of operator
  - Distinct from defining reference types!

- The ‘*’ is the dereference operator
  - Same as for any other iterator as well

- If ‘p’ contains the address of ‘x’ we say that ‘the pointer p points to x’

- Pointers are built-in data types which need to be initialized in order to be meaningful
Pointers

- Initialize to zero means ‘point to no object’
  - Null pointer (special value, as no object has this address)
  - C++11: `nullptr`

- Pointers have types!
  - The address of an object of type T is ‘pointer to T’
  - Written as: `T*`

- For instance:

  ```
  int x;        // object of type int
  int *p;      // pointer to an int, *p has type int
  int* p;      // pointer to an int, p has type int*
  ```
Pointers

• A small (but full) example:

```cpp
int main()
{
    int x = 5;

    // p points to x
    int* p = &x;
    cout << "x = " << x << endl;

    // change the value of x through p
    *p = 6;
    cout << "x = " << x << endl;
    return 0;
}
```

![Diagram showing variable x and pointer p connecting to it]
Pointers

• Think of pointers to be iterators

• They point to the single object stored in a ‘virtual’ (non-existent) container
Arrays

- Part of the language rather than standard library
  - Hold sequence of elements of the same type
  - Size must be known at compile time
  - No member functions, no embedded typedefs
    - i.e. no size_type member, use std::size_t instead

3 dimensional point:

```cpp
double coords[3];
```

// or
```cpp
std::size_t const ndim = 3;
double coords[ndim];
```
Arrays

• Fundamental relationship between arrays and pointers
  • Name of the array is interpreted as a pointer to its first element:
    \[
    \text{*coords} = 1.5; \quad // \text{set first element in \textit{coords} to 1.5}
    \]
**Pointer Arithmetic**

- Pointer is a random-access iterator
  - If ‘p’ points to the mth element of an array, then
    - ‘p+n’ points to the (m+n)th element of the array
    - ‘p-n’ points to the (m-n)th element of the array

- Further, as first element of array has number ‘0’
  - coords+1 points to the second, and
  - coords+2 points to the third element
  - coords+3 points to the first element after the last

- Possible to use standard algorithms with arrays:

```cpp
vector<double> v;
copy(coords, coords + ndim, back_inserter(v));
```
**Pointer Arithmetic**

- Possible to initialize containers from arrays:
  \[
  \text{vector<\texttt{double}> v(coords, coords + ndim);}
  \]

- More generally, wherever we used v.begin() and v.end(), we can use a and a+n (a: array, n: size)

- If ‘p’ and ‘q’ are pointers
  - Then p-q is the distance of the two pointers, which is the number of elements in between
  - Further (p – q) + q == p
**Indexing**

- If ‘p’ points to the mth element of an array, then p[n] is the (m+n)th element, not its address.

- Consequently, if ’a’ is an array, and ‘n’ an integer, then a[n] refers to the nth element inside the array ‘a’.

- More formally, if ‘p’ is a pointer, and ‘n’ an integer, then p[n] is equivalent to *(p+n)

- In C++ indexing is not a property of arrays, but a corollary to the properties of pointers and arrays and the fact that pointers are random access iterators.
Array Initialization

• Historically, arrays can be initialized easily:

```cpp
int const month_lengths[] = {
  // we will deal elsewhere with leap years
  31, 28, 31, 30, 31, 30,
  31, 31, 30, 31, 30, 31
};
```

• No size specified, it’s automatically calculated

• If size is specified, missing elements are set to zero (value-initialized)

• C++11 allows the same syntax for containers

```cpp
std::vector<int> const month_lengths = {
  // we will deal elsewhere with leap years
  31, 28, 31, 30, 31, 30,
  31, 31, 30, 31, 30, 31
};
```
String Literals Revisited

• String literals are character arrays with a trailing zero byte

• These are equivalent:

```c
char const hello[] = { 'H', 'e', 'l', 'l', 'o', '\0' };  
"Hello"
```

• Null character is appended to be able to locate the end of the literal

• Library has special functions dealing with ‘C’ strings (string literals)
String Literals Revisited

• Find the length of a string literal (‘C’ string): strlen

    // Example implementation of standard-library function
    size_t strlen(char const* p)
    {
        size_t size = 0;
        while (*p++ != '\0')
            ++size;
        return size;
    }

• Counting bytes (characters) excluding the null character
String Literals Revisited

• Variable hello and literal “Hello” are equivalent:
  ```cpp
  string s(hello);
  string s("Hello");
  • All will construct a string instance ‘s’ holding “Hello”

• Pointers are iterators:
  ```cpp
  string s(hello, hello + strlen(hello));
  ```
Pointers and References

• Think of a reference as an automatically dereferenced pointer
  • Or as “an alternative name for an object”
  • A reference must be initialized
  • The value of a reference cannot be changed after initialization

```c
int x = 7;
int y = 8;
int* p = &x;    *p = 9;
p = &y; // ok

int& r = x;     r = 10;
r = &y; // error (and so is all other attempts to
        // change what r refers to)
```
Arrays of Character Pointers

- String literal is a convenient way of writing the address of the first character of a null-terminated string.
- Arrays can be initialized conveniently.
- Show how to initialize an array of character pointers from a sequence of string literals.

Grading (again *sigh*):

<table>
<thead>
<tr>
<th>If the grade is at least</th>
<th>97</th>
<th>94</th>
<th>90</th>
<th>87</th>
<th>84</th>
<th>80</th>
<th>77</th>
<th>74</th>
<th>70</th>
<th>60</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Then the letter grade is</td>
<td>A+</td>
<td>A</td>
<td>A-</td>
<td>B+</td>
<td>B</td>
<td>B-</td>
<td>C+</td>
<td>C</td>
<td>C-</td>
<td>D</td>
<td>F</td>
</tr>
</tbody>
</table>

If the grade is at least 97, the letter grade is A+; otherwise, use the corresponding letter grade based on the table.
Arrays of Character Pointers

```c
string letter_grade(double grade)
{
    // range posts for numeric grades
    static double const numbers[] = {
        97, 94, 90, 87, 84, 80, 77, 74, 70, 60, 0
    };

    // names for the letter grades
    static char const* const letters[] = {
    };

    // compute the number of grades given the size of the array
    // and the size of a single element
    static size_t const ngrades = sizeof(numbers)/sizeof(numbers[0]);

    // given a numeric grade, find and return the associated letter grade
    for (size_t i = 0; i < ngrades; ++i) {
        if (grade >= numbers[i])
            return letters[i];
    }
    return "?\?\?";
}
```
Arguments to main()

- Command line arguments are optionally passed to main
  - Alternative prototype for main():
    ```
    int main(int argc, char** argv);
    ```
    - argc: number of arguments
    - argv: pointer to an array of character pointers, one argument each
    - At least one argument: the name of the executable itself, thus argc >= 1
Arguments to main()

• Let’s assume our executable is called ‘say’
• Invoking it as
  
  `say Hello, world`

• Should print: Hello, world
Arguments to main()

```cpp
int main(int argc, char** argv)
{
    // if there are arguments, write them
    if (argc > 1) {
        int i; // declare i outside the for because we need
        // it after the loop finishes
        // write all but the last entry and a space
        for (i = 1; i < argc-1; ++i)
            cout << argv[i] << " "; // argv[i] is a char*
        cout << argv[i] << endl; // write the last entry
        // but not a space
    }
    return 0;
}
```
Multiple Input Files

- Print the content of all files given on command line to console:

```c
int main(int argc, char** argv)
{
    int fail_count = 0;
    // for each file in the input list
    for (int i = 1; i < argc; ++i) {
        ifstream in(argv[i]);
        // if it exists, write its contents, otherwise
        // generate an error message
        if (in) {
            string s;
            while (getline(in, s))
                cout << s << endl;
        } else {
            cerr << "cannot open file " << argv[i] << endl;
            ++fail_count;
        }
    }
    return fail_count;
}
```
The Computer’s Memory

- As a program sees it
  - Local variables “live on the stack”
  - Global variables are “static data”
  - The executable code are in “the code section”
Three Kinds of Memory Management

• *Automatic* memory management
  • Local variables
    • Allocated at the point of the definition
    • De-allocated at the end of the surrounding scope
  • Memory becomes invalid after that point:
    // this function deliberately yields an invalid pointer.
    // it is intended as a negative example—don't do this!
    int* invalid_pointer()
    {
      int x;
      return &x;  // instant disaster!
    }
Three Kinds of Memory Management

- *Static* memory management
  - Memory allocated once
    - Either at program startup (global variables)
    - Or when first encountered (function-static variables)
  - De-allocated at program termination:
    ```c
    // This function is completely legitimate.
    int* pointer_to_static()
    {
        static int x;
        return &x;
    }
    
    • Always returns pointer to same object
Three Kinds of Memory Management

• *Dynamic* memory management
  • Allocate an instance of T with ‘new T’
  • De-allocate an existing instance pointed to by ‘p’ with ‘delete p’:

    ```
    int* p = new int(42);  // allocate int, initialize to 42
    ++*p;                 // *p is now 43, same as ++(*p)
    delete p;             // delete int pointed to by p
    ```

• Another example:

    ```
    int* pointer_to_dynamic()
    {
        return new int(0);
    }
    ```
Allocating an Array

• Arrays of type T are dynamically allocated using ‘new T[n]’, where n is the number of allocated elements

• De-allocation of an array pointed to by p is done using ‘delete [] p’

• ‘n’ can be zero! (Why?)

```cpp
T* p = new T[n];
vector<T> v(p, p + n);
delete[] p;
```

• Only way to create a dynamically sized array (remember, static array has to have size known at compile time)
A Problem: Memory Leak

```cpp
double* calc(int result_size, int max) {
    double* p = new double[max];  // allocate another max doubles
    // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // ... use p to calculate results to be put in result ...
    return result;
}

double* r = calc(200,100);  // oops! We “forgot” to give the memory
    // allocated for p back to the free store
```

- Lack of de-allocation (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks
A Problem: Memory Leak

double* calc(int result_size, int max)
{
    int* p = new double[max];  // allocate max doubles
    // i.e., get max doubles
    from
    // the free store
    double* result = new double[result_size];
    // ... use p to calculate results to be put in result ...
    delete[] p;  // de-allocate (free) that array
    // i.e., give the array back to the
    // free store
    return result;
}

double* r = calc(200,100);
// use r
delete [] r;  // easy to forget
Memory Leaks

• A program that needs to run "forever" can't afford any memory leaks
  • An operating system is an example of a program that "runs forever"

• If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  • Trick question: not enough data to answer, but about 130,000 calls

• All memory is returned to the system at the end of the program
  • If you run using an operating system (Windows, Unix, whatever)

• Program that runs to completion with predictable memory usage may leak without causing problems
  • i.e., memory leaks aren't "good/bad" but they can be a problem in specific circumstances
Memory Leaks

• Another way to get a memory leak

```c
void f()
{
    double* p = new double[27];
    // …
    p = new double[42];
    // …
    delete[] p;
}

// 1st array (of 27 doubles) leaked
```
Memory Leaks

- How do we systematically and simply avoid memory leaks?
  - Don't mess directly with new and delete
    - Use vector, etc.
  - Or use a garbage collector
    - A garbage collector is a program that keeps track of all of your allocations and returns unused free-store allocated memory to the free store (not covered in this course; see http://www.research.att.com/~bs/C++.html)
  - Unfortunately, even a garbage collector doesn’t prevent all leaks
  - Use RAII, see next lecture