Inheritance

William Hendrix

Lecture 19

Outline

- Review
- Inheritance
- Virtual functions
- Polymorphism
- Inheritance hierarchies

Complexity example

- Order of complexity can make a major difference for very large problem instances

- **Example:** creating a 1 billion element vector
  - If we allocate 1 at a time, we need to reallocate 1 billion times
  - If we allocate 1 million at a time, we need to reallocate 1000 times
    - Very wasteful for 10-element vectors
  - If we double the array size, we need ~30 reallocations
    - 2^{30} = 1,073,741,824
    - Vector is at least half full unless elements are being removed

- Moral of the story: $O(ln(n)) << O(n)$

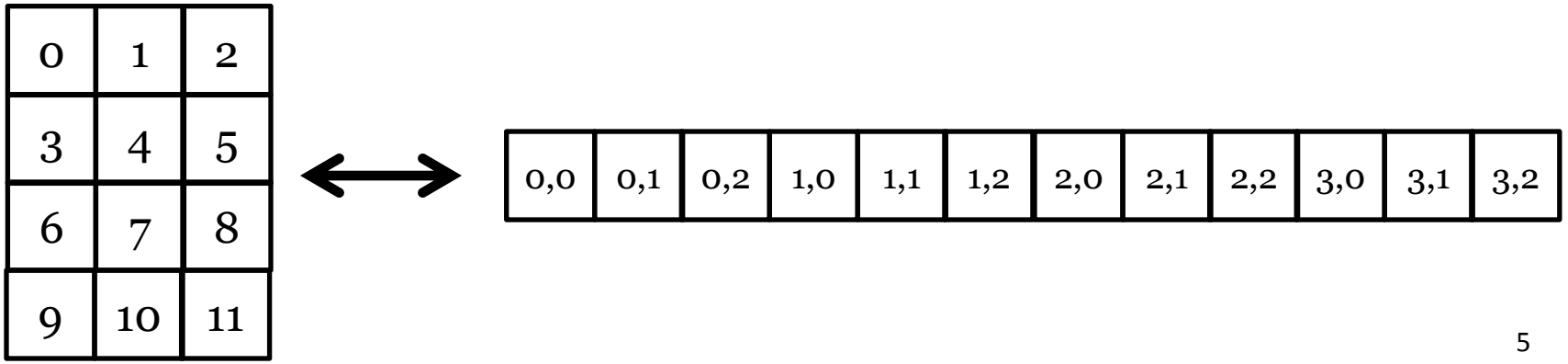
Complexity challenge

- Write a function that accepts a sorted array of integers, its size, and a target value, and outputs the indices `i` and `j` of the two values such that `arr[i] + arr[j]` is the closest to the target without going over. The function should run in linear time (*O(n)*) with respect to the array size.

- **Note:** if you calculate the sum of every pair, that will take $\binom{n}{2} = O(n^2)$ time.

2D array example

```
const int nrow = 4;
const int ncol = 3;
int stat[nrow][ncol];
int* dyn = new int[nrow*ncol];
for (int i = 0; i < nrow; i++)
  for (int j = 0; i < ncol; i++)
    stat[i][j] = dyn[i*ncol+j] = i*ncol+j;
if (memcmp(stat, dyn, i*j*sizeof(int)) == 0)
  cout << "The arrays are the same\n";
```



Nested loops

- Nested for loops are ideal for scanning 2D arrays

```
for (int row = 0; row < nrows; row++)
{
  int rowsum = 0;
  for (int col = 0; col < ncols; col++)
    rowsum += arr[i*ncols + j];
   cout << "Row " << row << ": " << rowsum << endl;
}
```

- Another useful construct: "triangular" loops

```
for (int i = 0; i < nrow; i++)
  for (int j = 0; j < i; j++)
    cout << "i * j = " << i * j << endl;
```

  - Avoids iterating through both `(i, j)` and `(j, i)`
  - Useful for symmetric matrices/computation

Inheritance

- Mechanism for extending or specializing classes
- Syntax

```
class NewClass : public OldClass
{
  //...
};
```

- All public members of `OldClass` will be accessible by `NewClass`
  - Only need to define "new" features of class (constructor, at least)
  - Private members will not be accessible, even within `NewClass` member functions
- **Important:** do not forget `public`, or inherited member functions will become `private`
- Member functions of `OldClass` can be *overridden* by redefining them in `NewClass`
  - Specifies new behavior for function
  - E.g., `void print()`

“Is-a” vs. “Has-a” inheritance

- “Has-a” inheritance
  - Add a class as a data member
  - Can’t call member functions of “inherited” class on objects of “inheriting” class
    - Public methods are still accessible inside “inheriting” class functions
- Conceptually: is `NewClass` a type of `OldClass`, or does it have/contain/reference an `OldClass` object?
- Example
  - A Computer contains a DiskDrive; it is not a type of DiskDrive
    - Computers may contain 0 or 2+ DiskDrives
  - A WebServer is a special type of Computer
    - Every WebServer is also a Computer and have the same fields and methods (`clock_speed`, `RAM`, `runProgram(char* file)`, etc.)
    - WebServers may have additional features, like `domainName` and `getWebpage(char* url)`

Inherited constructors/destructors

- Just like with "has-a" inheritance, default `OldClass` constructor always called before `NewClass` constructor
- Unlike "has-a" inheritance, we can't "reassign" superclass object
- Must use special constructor syntax:

```
NewClass::NewClass(args)
  : OldClass(args)
{
  //Initialize fields exclusive to NewClass
}
```

- Superclass destructor is called automatically after destructor finishes
  - Destructor should just clean up features exclusive to `NewClass`

Accessing superclass members

- Use scope resolution operator (`::`) to access superclass elements that are overridden

- Example: `void WebServer::print()`
  - A WebServer has a URL that it should output, but everything else should be printed like a Computer
    - **Design principle:** reuse code whenever possible
  - If we call `print()` inside `WebServer::print()`, this is a recursive call
  - To call Computer's print function, use `Computer::print()`:

```
void WebServer::print()
{
  Computer::print();
  cout << "Domain name:  " << domainName << endl;
}
```

Relationship to superclass

- Subclass stores all superclass data members and can invoke all member functions of the superclass
- Data members from superclass are guaranteed to occupy first bytes in subclass, in the same order
- Type casting from subclass to superclass is trivial
  - Pass a subclass object in place of superclass function parameter
  - Assign superclass object to be a subclass object
  - **Caveat:** object will call member functions from declared class, *not* the actual class
- An array of `OldClass*` can point to a mix of `OldClass` and `NewClass` objects (or any derived types)
- Data shearing
  - If we declare an array of `OldClass`, there won't be enough space for `NewClass` data members
  - Additional data members are overwritten by next array element
  - Always use pointers to mix superclass and subclass objects

Virtual functions

- Example

```
vector<Computer*> network;
network.push_back(&williams_pc);
network.push_back(&nans_pc);
network.push_back(&webserv);
for (int i = 0; i < 3; i++)
  network[i].print();
```

- Virtual functions are called based on actual class, not declared class
  - `virtual` destructor is useful if when using `delete[] arr;`
- Syntax: add `virtual` before return type in function definition (*not* prototype)
- Function must be declared as `virtual` in superclass and subclass
- Function binding determined at runtime
  - Checks actual type to determine which function to call
  - Incurs small overhead every time the function is called

Tonight

Lab 5 will be Monday, Feb 24