EECS 560 DATA STRUCTURES

MODULE 0: THE FUNDAMENTALS

DISCLAIMER

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Data Structures and Algorithm Analysis in C++***, 4th edition, by Mark Allen Weiss.

ADT (ABSTRACT DATA TYPE)

- ADT is an mathematical abstraction of a data structure, containing the set of its objects (data) and operations (methods).

- Example:
  - “vector” is an ADT
  - Its object primarily contains an array
  - Its methods may include “retrieve(i)”, which returns the value of its ith element, and “size()”, which returns the number of elements in the vector

ADT CONT.

- The primary goal of this course is to gain a deep understanding of the implementational details of the major ADTs (both of the data and methods).
- The secondary goal is to analyze the efficiency for the methods of different ADTs.

- We will be working on:
  - Simple ADTs: list, queue, stack, and hash table
  - Tree-based ADTs: search trees, heaps, disjoint sets
  - Graph ADTs
  - Sorting algorithms

FUNDAMENTALS

- In this module we will be reviewing fundamentals for learning data structure
    - C++ features and general guidelines for implementing data structures
    - basic concepts in time complexity analysis

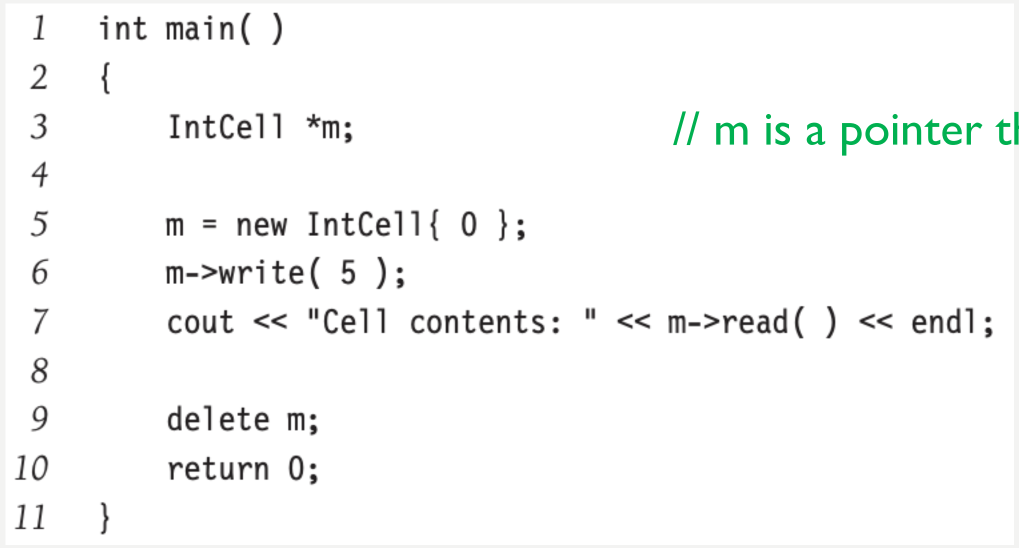
C++ FEATURES

- We will be discussing the following C++ features:
  - pointers and references
  - parameter passing
  - C++ objects
  - the big five: fundamental methods for data structure ADTs
  - template
  - function as parameter and function object

C++ FEATURES: POINTER

- Pointer is a variable that stores the (memory) address where another object resides.

- Assume we have an object named IntCell, we can use "*" to indicate that the variable is an object:



```
int main( )
{
    IntCell *m;

    m = new IntCell{ 0 };
    m->write( 5 );
    cout << "Cell contents: " << m->read( ) << endl;

    delete m;
    return 0;
}
```

// m is a pointer that points an address that stores the IntCell type

C++ FEATURES: POINTER

- Unlike many programming languages; C++ relies on the programmer to manage the memory usage for higher flexibility and efficiency (while being more challenging to program and debug)
  - memory allocation with **new**: e.g., int *x = new int;
  - memory collection with **delete**: e.g., delete x;

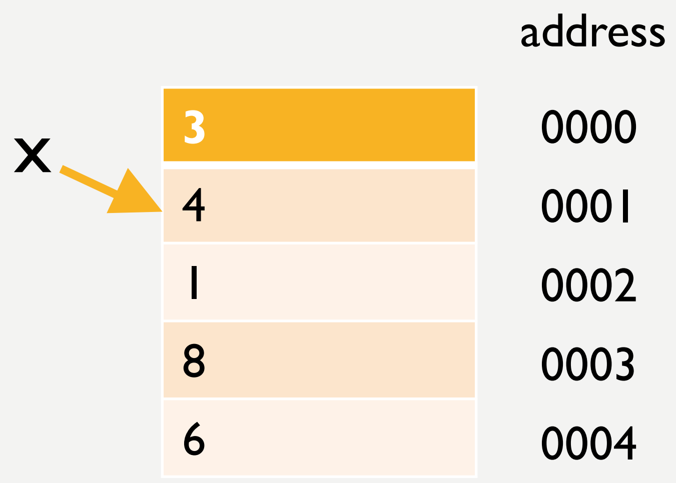
- Ideally, any address we have allocated must be reclaimed. However, in many cases this is not the case:
  - e.g., if we alter x by adding x = nullptr before reclaiming the address it points to, the program will lose the track of where x originally points to, and failed to reclaim the corresponding memory block
  - this is often called a **memory leak**, a common error found in many C++ programs

C++ FEATURES: POINTER

- Consider the example where x is a pointer for integer type:
  - You can access the data that x points to with "*"
  - for example: std::cout << *x << std::endl; will print out 4
  - note that std::cout << x << std::endl; will print out 0001

  - You can modify the data that x is pointing at with "*"
  - for example, *x=2 will replace the 4 in address 0001 with 2
  - note that x=2 will set x to point to address 0002

  - You can assign a pointer to another pointer
  - for example, y=x will assign y with a value of 0001; in this case both y and x are pointing to the address 0001
  - if we only want to assign the value x is pointer at to the address where y is pointer at, we should use *y=*x



C++ FEATURES: POINTER

- We can get the address of a variable using operator “&”, and assign it to a pointer:
  - e.g., int a = 6; int *x = &a;
  - now x will point to a

- If the pointer points to an object, we can access its member through operator “->”
  - e.g., IntCell *x = new IntCell; x->findMax();
  - x will be a pointer for the object IntCell, and we can use “->” to call IntCell’s member function findMax()

C++ FEATURES: REFERENCE

- References can be considered as alias of variables (lvalue) or constants (rvalue).
- Reference can be defined using the operator "&"
  - e.g., `int &x = a`;
  - in this case, we can use x interchangeably with a
- The major difference between reference and pointer:
  - reference **does not need a memory space to store**; it only takes up a small space in stack
  - pointer **does need a memory space to store**; it also takes up a small space in stack

C++ FEATURES: REFERENCE

- Some other important differences between pointer and reference:
  - pointer can be re-assigned, reference cannot
    - int *p = nullptr; p = &a; is valid; while int &p = a; p = b; is invalid
  - you can have multi-level indirection in pointer (such as pointer of pointer of … a type), but you can have only one-level indirection for reference
    - int *y = a; int **x = y; is valid; while the operator "&&" would mean a different thing (rvalue reference, will discussed in detail later) when declaring reference
  - pointer can be declared uninitialized, reference cannot
    - int *x; is valid; while int &x; is invalid

C++ FEATURES: REFERENCE

- More differences between pointer and reference:
  - pointer can be modified and used to iterate an array; reference cannot
    - int *p = a[0]; p++; is valid; while int &p = a[0]; p++; is invalid
  - pointer needs to be dereferenced when accessing the memory location it points to; reference does not need to be dereferenced
    - int *p = a; cout << *p; vs int &p = a; cout << p;
  - pointer can be stuffed into arrays, reference cannot
    - int **p = new int* [n]; is valid; while we cannot have an array of refereces
  - pointer cannot be bound to temporaries (rvalue), reference can
    - int *p = 12; is invalid; while int &&p = 12; is valid

C++ FEATURES: REFERENCE

- Reference can be further divided into reference to **lvalue** and reference to **rvalue**:
  - lvalue: regular variables, which are stored in some memory block; for example, int a, int *p, r[4] *etc.*
  - rvalue: temporary constants, which are not stored in memory block: for example, 12, “hello world” *etc.*
  - (since pointer needs to point to an address and rvalues do not have addresses, so there is not pointer to rvalue)
  - C++ allow reference to both lvalue (with “&”) and rvalue (with “&&”)
    - lvalue reference: int &p = a;
    - rvalue reference: int &&p = 12;

C++ FEATURES: REFERENCE

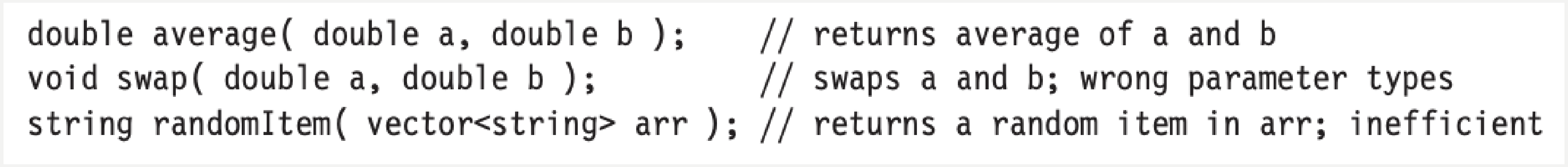
- How C++ handles lvalue reference and rvalue reference:
  - for lvalue reference, C++ will simply create an alias of the address as the reference (note that lvalues are already stored in memory); when using, we must first **copy** the constant value to a new address (and we have two copies of the data, one copy as temporary data, and another copy as stored in memory), and the reference can be created for the copy stored in memory.
  - for rvalue reference, C++ will first **move** the temporary value in memory and create a corresponding alias for it; (we only have one copy of the data) as a result this mechanism is faster when passing constant parameters (will be discussed in later slides).

C++ FEATURES: PARAMETER PASSING

- There are four major ways to pass parameters to C++ functions
  - call by value
  - call by constant reference
  - call by lvalue reference
  - call by rvalue reference

C++ FEATURES: PARAMETER PASSING

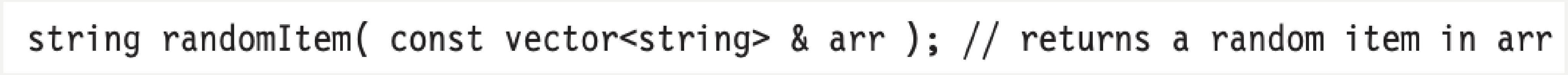
- Call by value:
  - needs to make a copy of the parameter before passing into the function (slow)
  - cannot be use to modify the parameters (non-mutable)



```
double average( double a, double b );    // returns average of a and b
void swap( double a, double b );         // swaps a and b; wrong parameter types
string randomItem( vector<string> arr ); // returns a random item in arr; inefficient
```

C++ FEATURES: PARAMETER PASSING

- Call by constant reference
  - can avoid copying the variables (fast)
  - keeps the variables unchanged (non-mutable)



```
string randomItem( const vector<string> & arr ); // returns a random item in arr
```

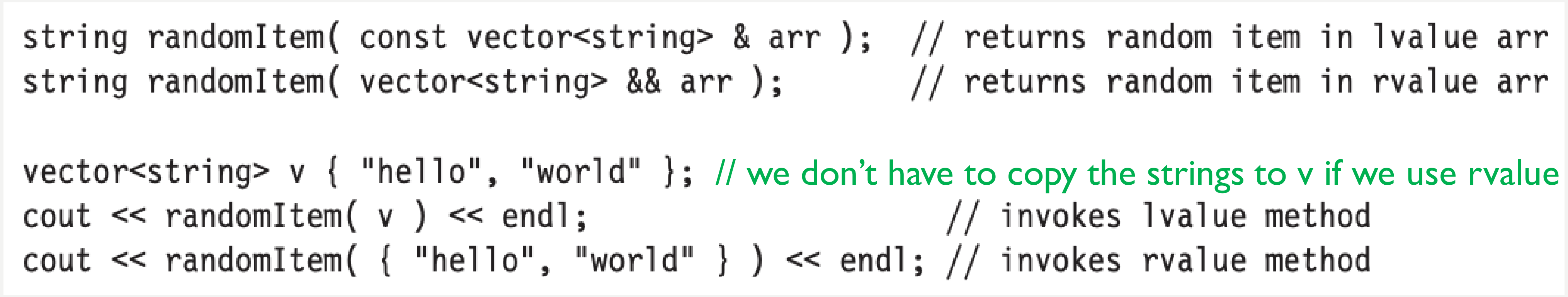
C++ FEATURES: PARAMETER PASSING

- Call by lvalue reference:
  - can avoid copying variables (fast)
  - allows modification of variables (mutable)
  - perhaps the mostly-used parameter passing mechanism

```
void swap( double & a, double & b );     // swaps a and b; correct parameter types
```

C++ FEATURES: PARAMETER PASSING

- Call by rvalue reference
  - allows passing of constants as parameters (which call by value can do but call by lvalue reference cannot)
  - can avoid copying the constants (which call by value cannot do but call by lvalue reference can)



```
string randomItem( const vector<string> & arr );  // returns random item in lvalue arr
string randomItem( vector<string> && arr );       // returns random item in rvalue arr

vector<string> v { "hello", "world" }; // we don't have to copy the strings to v if we use rvalue reference
cout << randomItem( v ) << endl;                        // invokes lvalue method
cout << randomItem( { "hello", "world" } ) << endl; // invokes rvalue method
```

C++ FEATURES: PARAMETER PASSING

- A note on call by passing pointer:
  - an older parameter passing style in the C age
  - in most cases, it has the same effect as call by lvalue reference
  - when to use call by passing pointer?
    - if you do need to take advantage of the features of pointer that reference cannot offer; e.g., looping through a vector, modify the pointer, allowing NULL pointer (uninitialized parameters) *etc*.
    - if you are working on legacy C code and wish to keep the coding style consistent

C++ FEATURES: OBJECT

- As an object-oriented programming language, the key components in C++ programs are objects.
- In this course, we will implement each data structure ADT as an object.

C++ FEATURES: OBJECT

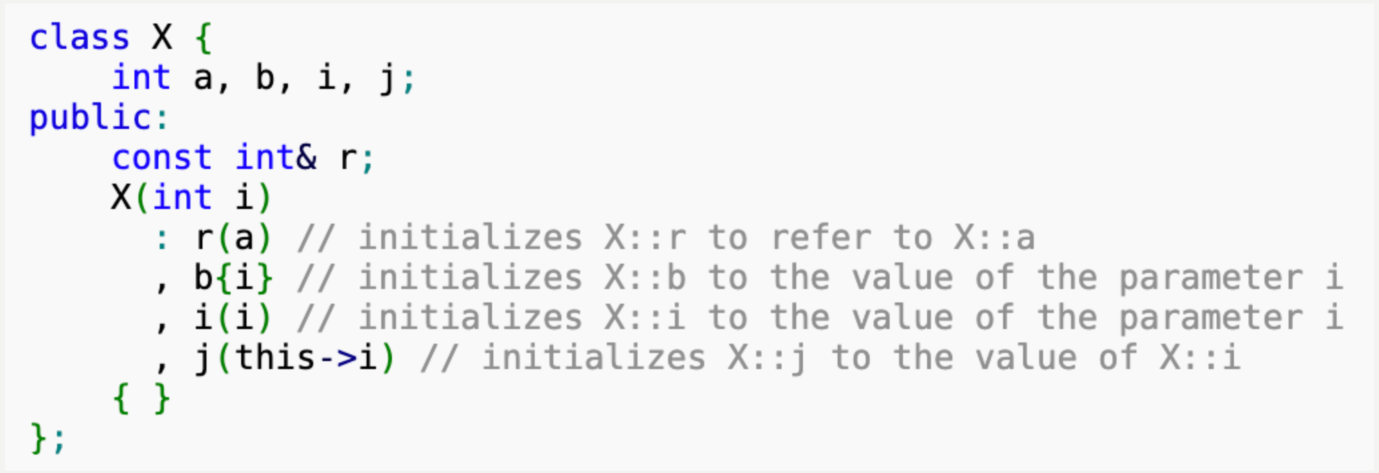
- Some coding practice:
  - Each object can contains **data** and **methods** (while does not need to contain both). For example, for a vector object, we can use an array as data the store the items of the vector, and implement a method that finds the maximum item from the vector. Data and methods are also called **members** of the object.
  - Both data and methods can be declared as either **private** or **public**. Private members are not directly accessible by methods that are not a member of the object, while public members are.
  - A C++ object is often split into two parts: interface and implementation. Interface defines the **signature** (data type, parameter types, and return type) of the components and is often written in a **.h** file (also called header file), while implementation actually implement the methods and is often written in a **.c**, **.cc**, or **.cpp** file.

C++ FEATURES: THE BIG FIVE

- For most of C++ object implementations (and every data structure ADT we will be implementing in this course), they all contain five methods that will provide a comprehensive and standard interface. These five methods are called "the big five":
  - Destructor: reclaim allocated memory to the object
  - Copy constructor: initialize the object from another object
  - Move constructor: initialize the object from rvalue
  - Copy assignment: overwrite the object using another object
  - Move assignment: overwrite the object using rvalue
  - The major difference between copy and move methods: copy methods will keep the original variable, while move will not. As a result, move methods are intended to be used with rvalue parameters.

C++ FEATURES: THE BIG FIVE

- Initialization list in C++ 11:
  - a colon followed by a list of comma-separated member initializers
  - the only way to initialize a const member in the object
  - the only way to initialize a member object if the member object has no zero-parameter constructor



```
class X {
    int a, b, i, j;
public:
    const int& r;
    X(int i)
      : r(a) // initializes X::r to refer to X::a
      , b{i} // initializes X::b to the value of the parameter i
      , i(i) // initializes X::i to the value of the parameter i
      , j(this->i) // initializes X::j to the value of X::i
    { }
};
```

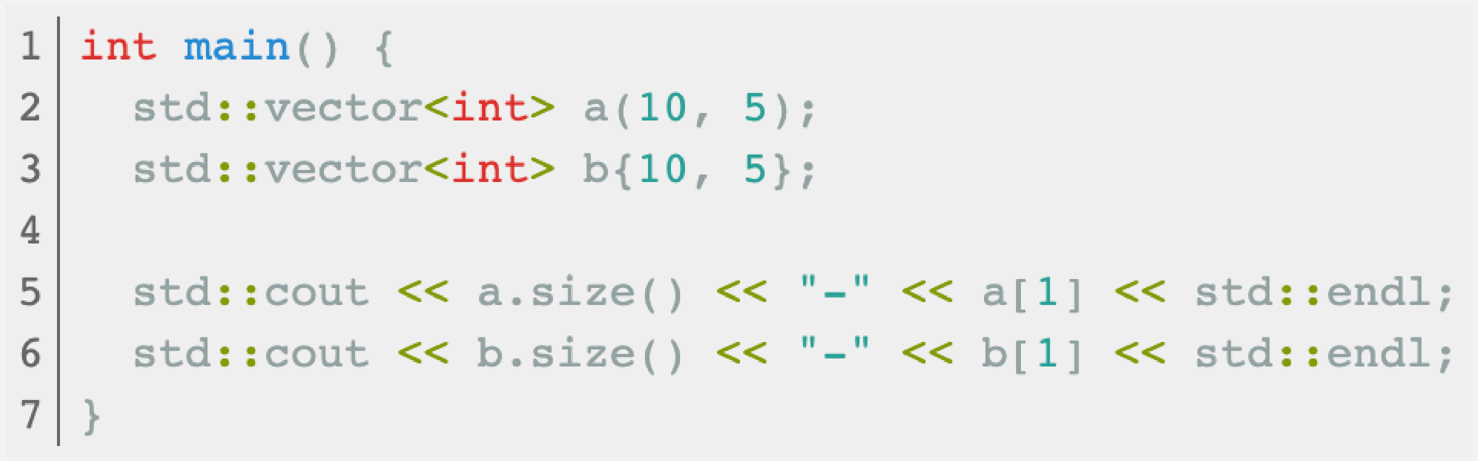
C++ FEATURES: THE BIG FIVE

- You may notice that some member functions are initialized with braces “{}” while the other ones are initialized with parenthesis “()”.

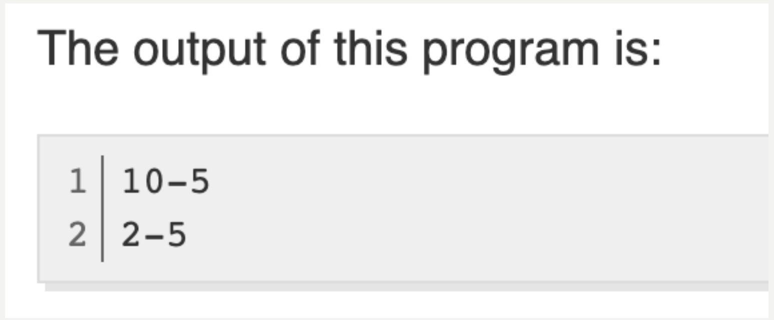
- Using braces “{}” is a recent style encouraged by C++ to consolidate initialization syntax. In many cases using braces and parenthesis are the same.

- However, in some cases they are different.
  - they may invoke different object constructors, depending how they are defined.
  - initialization with parenthesis “()” is considered as calling the constructor with **N parameters**
  - initialization with braces “{}” is considered as calling the constructor with **one list parameter**

C++ FEATURES: THE BIG FIVE



```cpp
int main() {
  std::vector<int> a(10, 5);
  std::vector<int> b{10, 5};

  std::cout << a.size() << "-" << a[1] << std::endl;
  std::cout << b.size() << "-" << b[1] << std::endl;
}
```



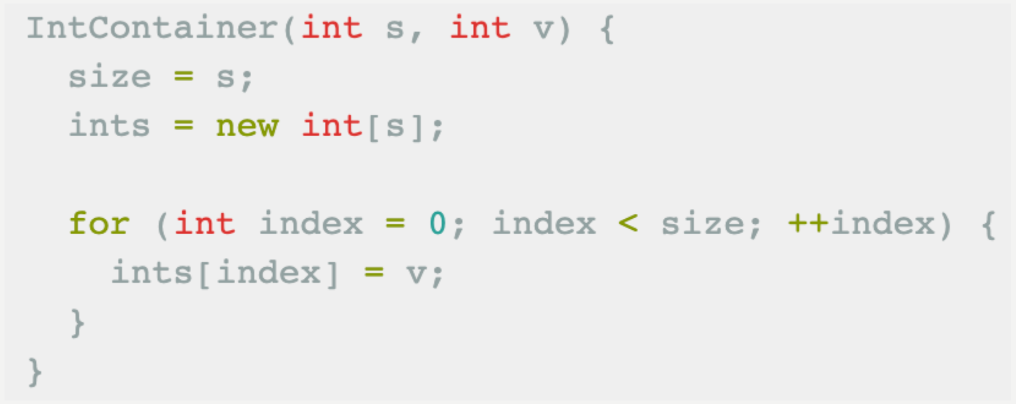
The output of this program is:

```
10-5
2-5
```

This is because std::vector defines different constructors for accepting two integer parameters and accepting one list parameter. The effect for the first call would be initializing a vector with ten fives, while the second call would be initializing a vector with a ten and a five.

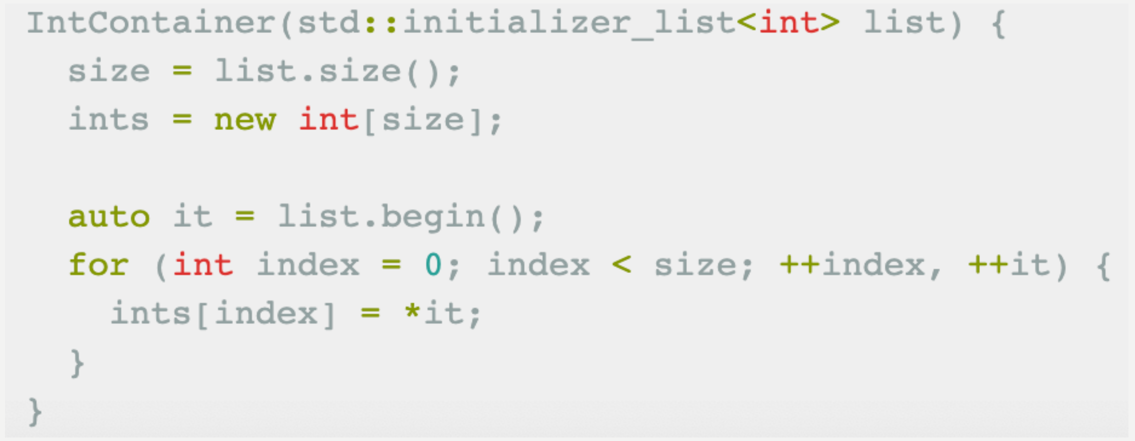
https://ncona.com/2019/01/variable-initialization-in-cpp/#:~:text=Initializer%20lists%20were%20introduced%20in,%7B%7D%20instead%20of%20parentheses%20()%20.

C++ FEATURES: THE BIG FIVE



```
IntContainer(int s, int v) {
  size = s;
  ints = new int[s];

  for (int index = 0; index < size; ++index) {
    ints[index] = v;
  }
}
```

constructor with two parameters



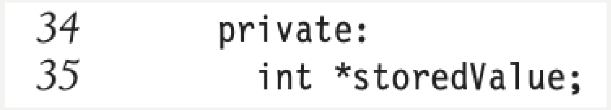
```
IntContainer(std::initializer_list<int> list) {
  size = list.size();
  ints = new int[size];

  auto it = list.begin();
  for (int index = 0; index < size; ++index, ++it) {
    ints[index] = *it;
  }
}
```

constructor with one list parameter

https://ncona.com/2019/01/variable-initialization-in-cpp/#:~:text=Initializer%20lists%20were%20introduced%20in,%7B%7D%20instead%20of%20parentheses%20()%20.

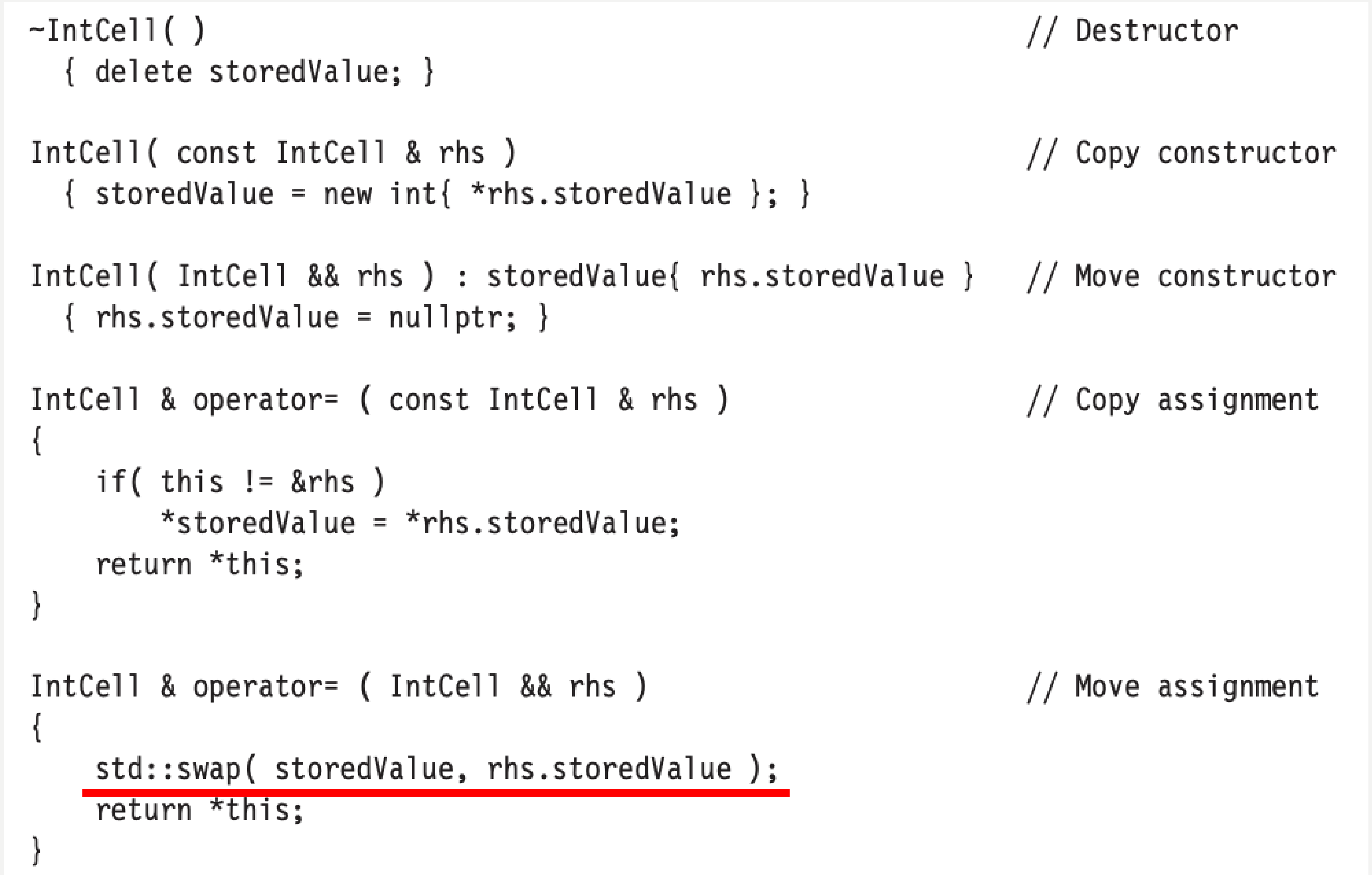
C++ FEATURES: THE BIG FIVE

data declaration



```
        private:
          int *storedValue;
```

the big five



```
~IntCell( )                                                    // Destructor
  { delete storedValue; }

IntCell( const IntCell & rhs )                                 // Copy constructor
  { storedValue = new int{ *rhs.storedValue }; }

IntCell( IntCell && rhs ) : storedValue{ rhs.storedValue }     // Move constructor
  { rhs.storedValue = nullptr; }

IntCell & operator= ( const IntCell & rhs )                    // Copy assignment
{
    if( this != &rhs )
        *storedValue = *rhs.storedValue;
    return *this;
}

IntCell & operator= ( IntCell && rhs )                         // Move assignment
{
    std::swap( storedValue, rhs.storedValue );
    return *this;
}
```

A more efficient swap via taking advantages of rvalue

C++ FEATURES: THE BIG FIVE

the less efficient way

```
void swap( vector<string> & x, vector<string> & y )
{
    vector<string> tmp = x;
    x = y;
    y = tmp;
}
```

three copies

the more efficient way

```
void swap( vector<string> & x, vector<string> & y )
{
    vector<string> tmp = std::move( x );
    x = std::move( y );
    y = std::move( tmp );
}
```

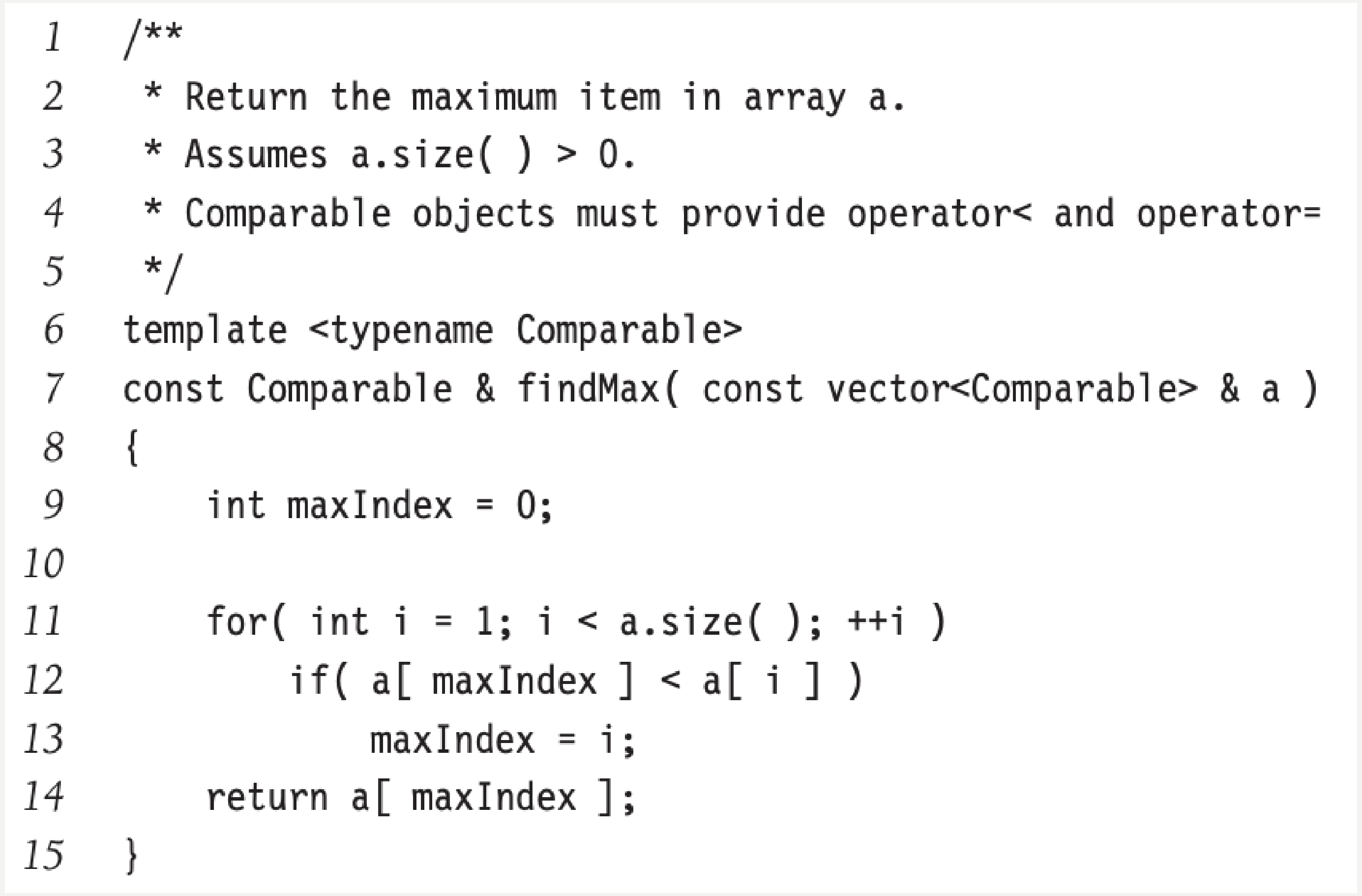
three reference changes:

- std::move() marks the variable as rvalue, which means that it can be moved without copying.
- the first line assigns tmp to label the data x
- the second line assigns x to label the data y
- the third line assigns y to label the data tmp

C++ FEATURES: TEMPLATE

- Template allows for a single implementation of the same behaviors on different data types:
  - function template
  - class template
- For example:
  - find the maximum number of an array of integers
  - find the maximum number of an array of floats
  - ……
- For most of the data structure ADTs, we would like to implement them as templates to make them more generic.

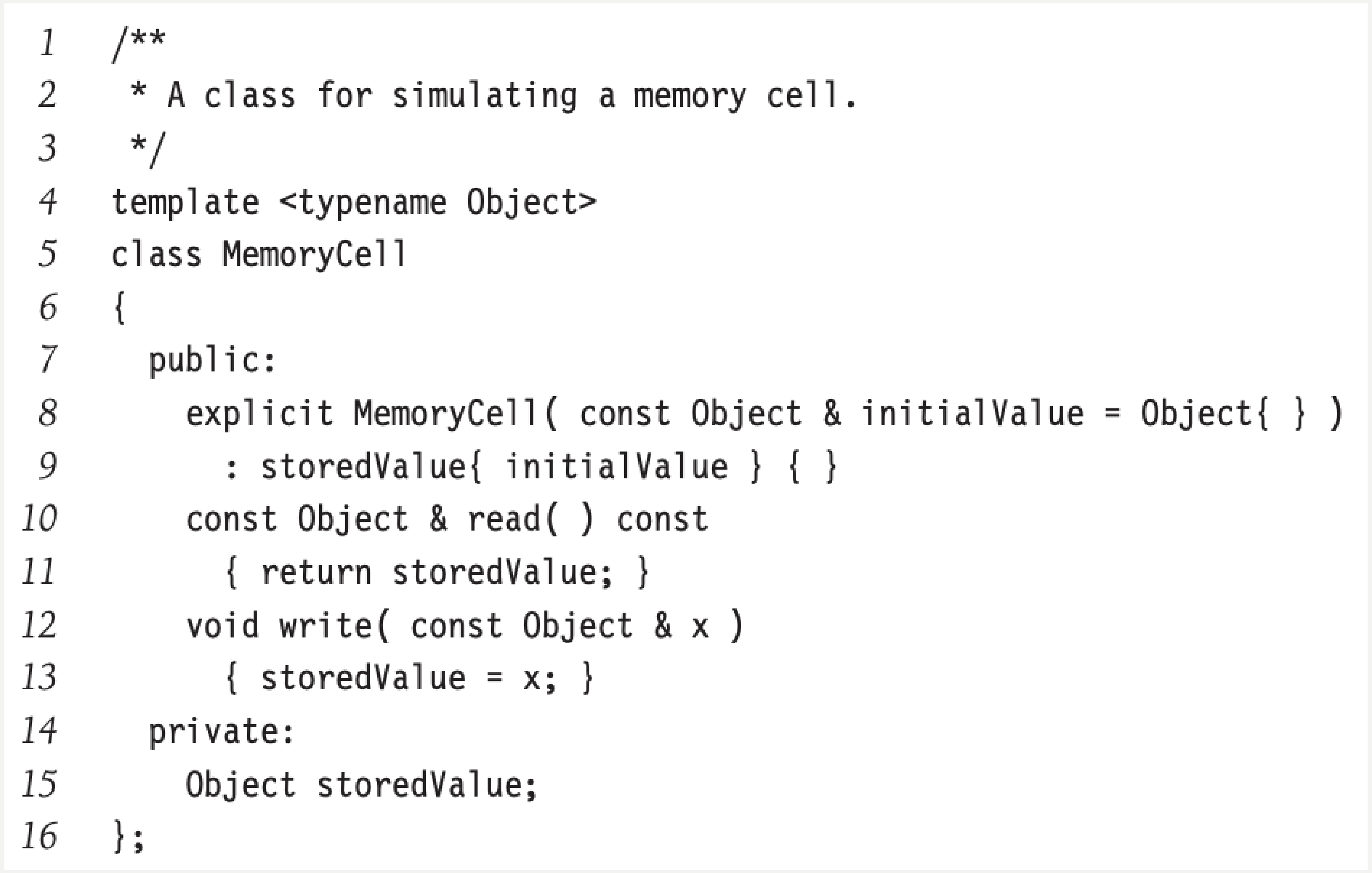
C++ FEATURE: TEMPLATE



```cpp
/**
 * Return the maximum item in array a.
 * Assumes a.size( ) > 0.
 * Comparable objects must provide operator< and operator=
 */
template <typename Comparable>
const Comparable & findMax( const vector<Comparable> & a )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); ++i )
        if( a[ maxIndex ] < a[ i ] )
            maxIndex = i;
    return a[ maxIndex ];
}
```

A function template

C++ FEATURE: TEMPLATE



```
/**
 * A class for simulating a memory cell.
 */
template <typename Object>
class MemoryCell
{
  public:
    explicit MemoryCell( const Object & initialValue = Object{ } )
      : storedValue{ initialValue } { }
    const Object & read( ) const
      { return storedValue; }
    void write( const Object & x )
      { storedValue = x; }
  private:
    Object storedValue;
};
```

A class template

C++ FEATURES: TEMPLATE

- Generic programming:
  - Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.
  - That is, the use of template

- Technically, the declaration and implementation of a template object must be put in the same file (.h). In other words, if you use template, then you will write all your code in the .h file and will not need the .cc file.
- Generic programming is more extensible, but also needs a bit more effort to maintain
- C++ STL adopted the generic programming style

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT

- Occasionally we may want to use different ways to handle the same data, and don't want to re-implement the function:
  - imagine a function that finds the largest rectangles among a set of them
  - we can compare them based on their areas
  - we can also compare them based on their perimeters

- One solution would be to write two comparison functions, and pass the comparison function as parameter. For example, findMax(rectangles, compareByArea) or findMax(rectangles, compareByPerimeter).

- A more recent style is to implement the functions as objects

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT

- An example based on std::sort():

```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

function definition

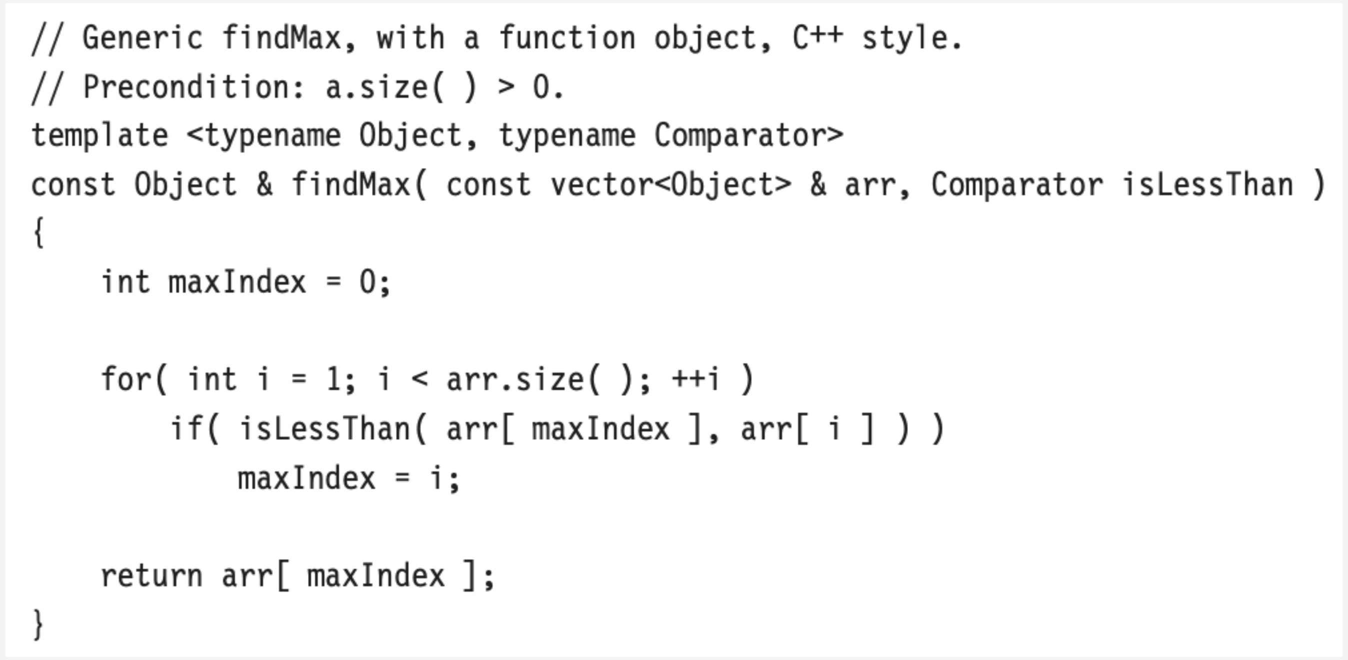
```
bool myfunction (int i,int j) { return (i<j); }
```

comparison function definition

```
// using function as comp
std::sort (myvector.begin()+4, myvector.end(), myfunction);
```

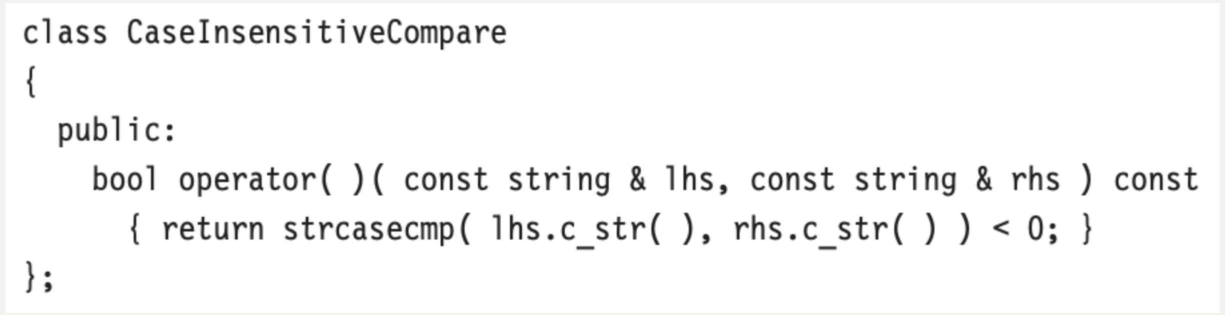
passing the comparison function as parameter

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT



```
// Generic findMax, with a function object, C++ style.
// Precondition: a.size( ) > 0.
template <typename Object, typename Comparator>
const Object & findMax( const vector<Object> & arr, Comparator isLessThan )
{
    int maxIndex = 0;

    for( int i = 1; i < arr.size( ); ++i )
        if( isLessThan( arr[ maxIndex ], arr[ i ] ) )
            maxIndex = i;

    return arr[ maxIndex ];
}
```

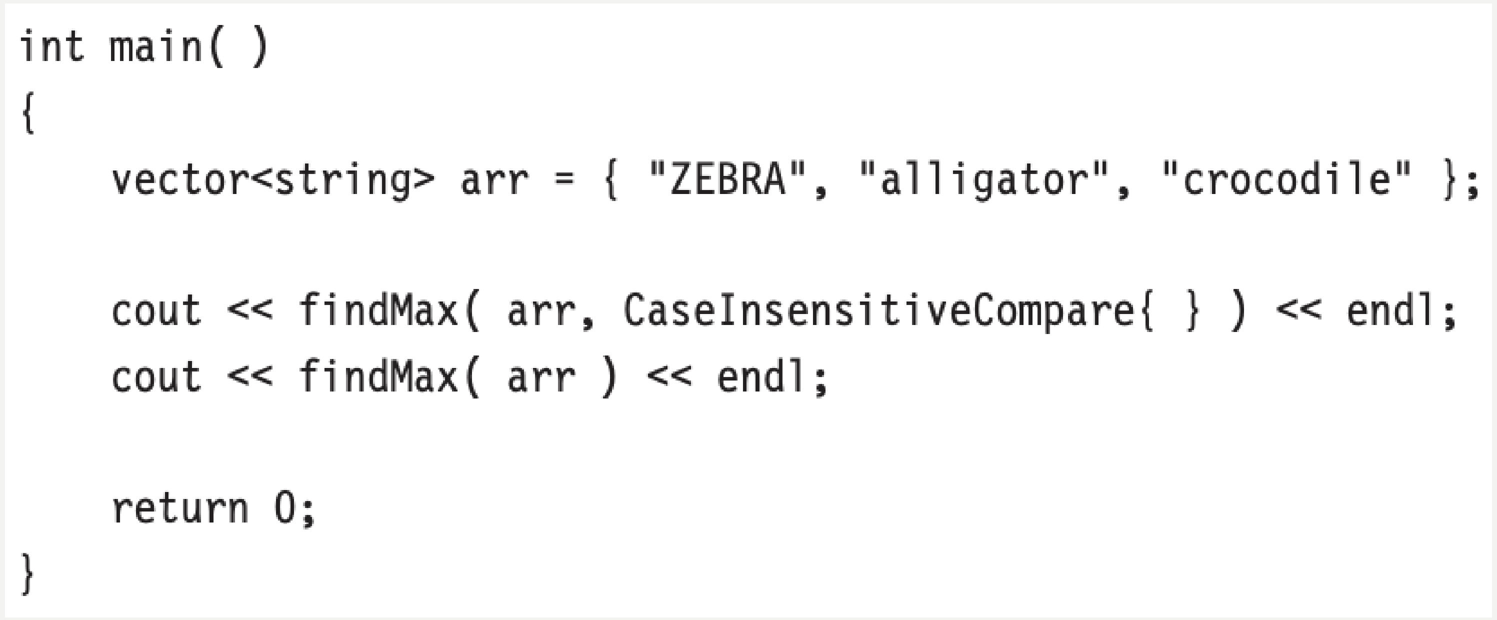
the findMax() function



```
class CaseInsensitiveCompare
{
  public:
    bool operator( )( const string & lhs, const string & rhs ) const
      { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
};
```

the comparison function object

C++ FEATURE: FUNCTION PARAMETER AND FUNCTION OBJECT



```
int main( )
{
    vector<string> arr = { "ZEBRA", "alligator", "crocodile" };

    cout << findMax( arr, CaseInsensitiveCompare{ } ) << endl;
    cout << findMax( arr ) << endl;

    return 0;
}
```

calling findMax() with function object as parameter

TIME COMPLEXITY ANALYSIS

- Time complexity analysis refers to the evaluation of the efficiency of the algorithm.

- Many complications exists:
  - primitive operations
  - speed of the machine
  - implementation (software and hardware)
  - input size (measured by the number of bits)
  - …

- As a result, we will be developing a model to eliminate the impact of these confounding factors.

TIME COMPLEXITY ANALYSIS

- Time complexity model:
  - the model is a function that converts the input size to the number of primitive steps
  - only count the number of primitive operations (e.g., addition is much faster than multiplication, but we count either operation as one primitive operation), which leads to the elimination of coefficient in our study
    - disregarding term coefficient: consider two terms $5x$ and $10x$
    - the algorithm having the second term as time complexity can run faster if it is assigned to a faster machine
  - focus on the cases where the input size is large, which leads to only focusing on the fastest-growing term of the function disregarding the specific constant coefficient
    - fastest-growing term: consider two terms: $0.001x^2$ and $1000x$
    - we see that the first term will eventually dominate when x grows sufficiently large

TIME COMPLEXITY ANALYSIS

- We can consider time complexity, **when multiplied by a large-enough constant**, as an upper bound of the running time with **arbitrarily large input size**.
  - consider a function that converts the size of input, n, to the number of primitive operations, T:
  - $T = 16n^2 + 8n + 17$; note that n is an integer and $n > 0$
  - by eliminating the coefficient and only focusing on the fastest-growing term, we get n^2
  - if we multiply n^2 with a large-enough constant, say 41 (=16+8+17), we will get $41n^2$, which is guarantee to be larger than T for arbitrarily large n
  - can we pick a slower-growing term as the complexity? say n? Note that no matter how large the constant we will multiply, it will fail to upper bound T with large n.
  - $999999n$ will fail to bound T if n is larger than 999999

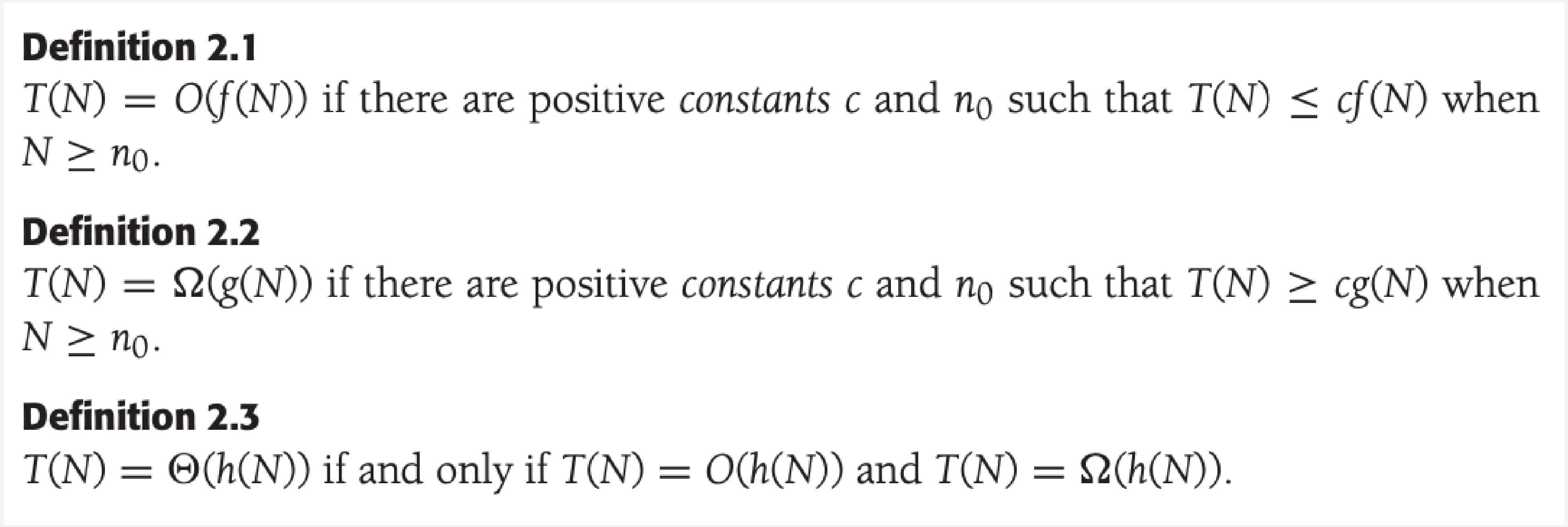
TIME COMPLEXITY ANALYSIS

- The above upper bound is most-frequently used in time complexity analysis.
- It measures (in a pessimistic point of view) how slower the program will run when the input size grows.
- We use big-O to denote such an upper bound:
  - in the previous example where $T = 16n^2 + 8n + 17$, we will write $T = O(n^2)$
  - we say "the running time of the program/algorithm is upper bounded by the order of n^2"
  - or, in many cases, we simply say that "the program/algorithm runs in n^2 time"

TIME COMPLEXITY ANALYSIS

- In addition to upper bound, we also have lower bound and tight bound.
- Lower bound measures the minimum amount of time required by the program.
- Tight bound is established when the upper bound and lower bound of the program are the same.

TIME COMPLEXITY ANALYSIS



Definition 2.1
$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Definition 2.2
$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

Definition 2.3
$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

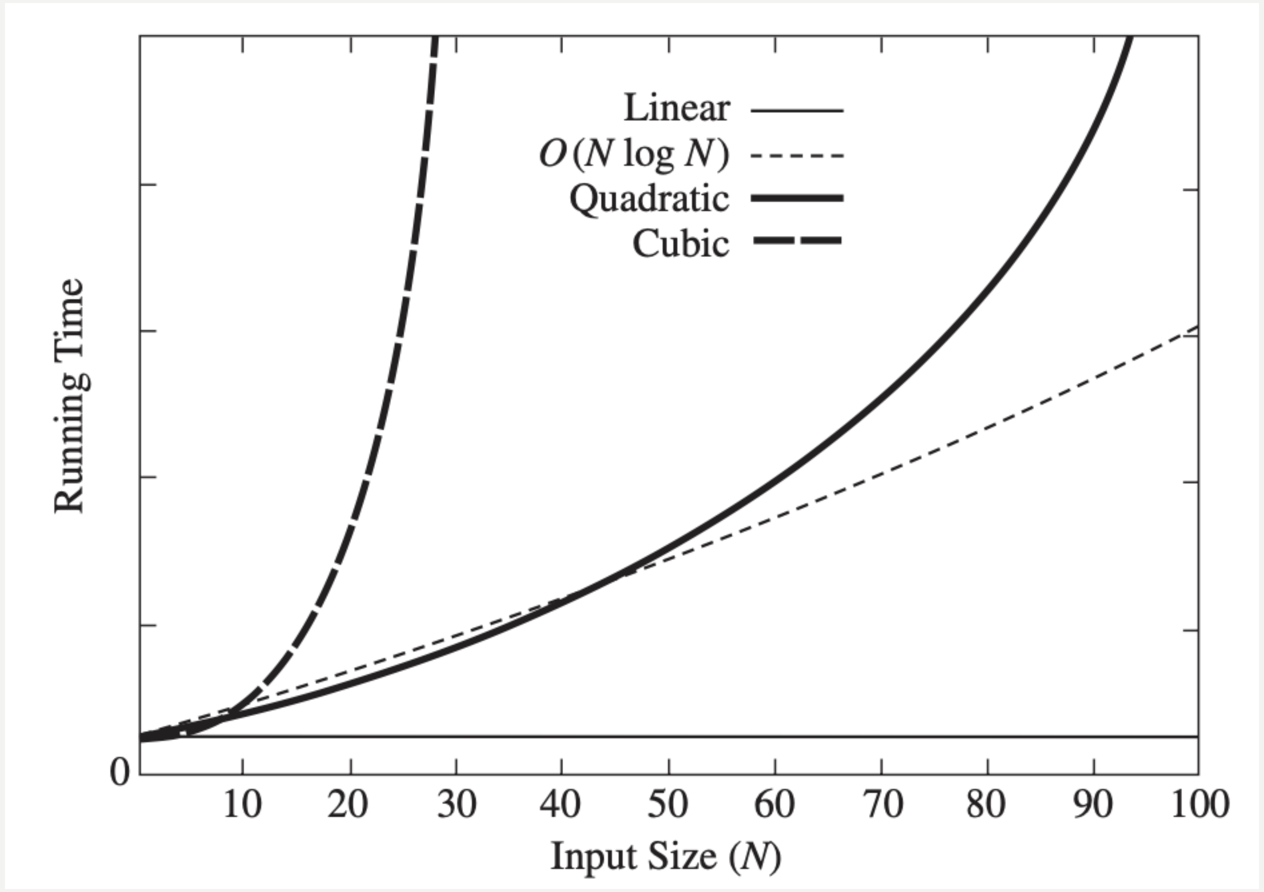
TIME COMPLEXITY ANALYSIS

- Four major classes of time complexity terms (from the fastest to the slowest):
  - constant
  - logarithm
  - polynomial
  - exponential

- Associated parameters (say base or exponent) does not matter between classes
  - $\log_2 n$ will be smaller than $n^{0.0001}$ as n gets larger

- We only consider the parameter when comparing terms within the same class
  - n^2 is faster than n^3

TIME COMPLEXITY ANALYSIS



TIME COMPLEXITY ANALYSIS

- The majority of algorithms we will discuss in the course will have a time complexity lower than exponential.

- In theory, a clear distinction is made between polynomial-time algorithms and exponential-time algorithms:
  - polynomial-time algorithms and faster algorithms are deemed **tractable**
  - exponential-time algorithms are deemed **intractable**

- More discussions regarding the issue in EECS 660 and EECS 764

SUMMARY

- C++ features
  - pointers and references
  - parameter passing
  - C++ objects
  - the big five: fundamental methods for data structure ADTs
  - template
  - function as parameter and function object

- Time complexity analysis
  - measures how efficient the algorithm is
  - function that converts the input size to the number of primitive operations
  - no constant coefficient, only the fastest-growing term
  - upper bound, lower bound, tight bound