CSE100

Advanced Data Structures

Lecture 2

(Based on Paul Kube course materials)
An introduction to C++

- C++ is an object-oriented language based on the non-object-oriented C language
- C++ was initially developed by Bjarne Stroustrup at Bell Labs in the 1980’s
- Now it is an ISO standard and is one of the 3 most widely used programming languages in the world (along with Java and C)
- In CSE 100, we don’t assume that you have programmed in C++ or C before
- We do assume that you have programmed in Java before!
- C++ is superficially similar to Java, but with many differences that can make it a challenge to use
- Let’s start by showing some comparisons between Java and C++, and then look at specific C++ details
Simple classes and objects in Java

• Suppose you have a Java class defined this way:

```java
public class C {
    private int a;
    public void setA(int a) { this.a = a; }
    public int getA() { return a; }
}
```

• Now you could write a Java program like this:

```java
public static void main(String[] args) {
    C x; // declare x to be a pointer to a C object
    x = new C(); // create a C object, and make x point to it
    x.setA(5); // dereference x, and access a member
    System.out.println(x.getA());
}
```
Simple classes and objects in C++

- Defining ‘the same’ class in C++:
  ```cpp
class C {
  private:
    int a;
  public:
    void setA(int a) { this->a = a; }
    int getA() { return a; }
  }
  ```

- And writing ‘the same’ program:
  ```cpp
#include <iostream>
int main() {
  C* x;  // declare x to be a pointer to a C object
  x = new C();  // create a C object, and make x point to it
  x->setA(5);  // dereference x, and access a member
  // note: (*x).setA(5) is equivalent
  std::cout << x->getA() << std::endl;
}
```
Pictures of memory

• It is useful to abstract the essentials about the contents of memory by drawing simple pictures...

• Here is a picture of the contents of memory after those statements (either the Java or the C++ version) execute:

• The pointer variable $x$ points to the object, an instance of the class $C$, which contains an instance variable named $a$ with value 5:

```
  x:
  a: 5
```

• Always be aware of the difference between a pointer and what it points to!
  • (more on this later)
Differences between Java and C++

Note some differences shown in those examples

- **Class definitions:**
  - In a Java, a visibility specifier (public, protected, or private) is attached to each member declaration; if missing, the member has “package” visibility
  - In C++, a visibility specifier (public:, protected:, or private:) sets off a section of member declarations; if missing, the members have private visibility
  - In C++, but not Java, the body of the class definition should end with a semicolon ; after the closing brace }
  - Both C++ and Java will provide a public default constructor if you do not define any, but in C++ it is not guaranteed to initialize primitive member variables for you

- **Pointers:**
  - In Java, declaring a variable of class type creates a pointer that can point to an instance of that type (an object); in C++, you have to explicitly specify that the variable is a pointer by using * after the typename
  - In Java, the dot operator . dereferences a pointer and accesses a member of the object pointed to; in C++, the -> operator does that; or, you can use a combination of the dereference operator * and the member access operator . That is, these are equivalent in C++: x->getA() and (*x).getA()
Memory management in Java and C++

• In Java, all objects are created using `new`, and memory an object takes is automatically reclaimed by the garbage collector when it can no longer be accessed in your program:

```java
C x = new C(); // create an object
x = null;      // object is inaccessible; no problem, gc will free
```

• In C++, objects can be created using `new`, but memory for such objects is not automatically reclaimed:

```cpp
C* x = new C(); // create an object using new
x = nullptr;    // object is inaccessible;
                // problem!! it will not be freed
```

• That is a memory leak! In C++, an object created using `new` must be explicitly freed using `delete` for its memory to be reclaimed for later use in your program:

```cpp
C* x = new C(); // create an object using new
...             // do things with the object
delete x;      // free the object using delete
x = nullptr;   // make pointer null to avoid dangling pointer
```

• Note: some objects created in other ways in a C++ program are reclaimed automatically. Understanding the details of memory management is an important part of C++ programming.
Automatic, static, and dynamic data in Java and C++

- A running program typically has 3 kinds of data in it:
  - automatic
  - static
  - dynamic

- These kinds of data differ in:
  - what region of machine memory the data is stored in
  - when and how memory for the data is allocated ("created")
  - when and how memory for the data is deallocated ("destroyed")

- Both Java and C++ have basically these same 3 kinds of data, with some differences
  - (Note: these are kinds of data, i.e. variables. Executable instructions are stored somewhat differently.)
Automatic data

- Automatic data is called "automatic" because it is automatically created and destroyed when your program runs
  - Examples of automatic data in both Java and C++: formal parameters to functions, local variables in functions (if the variables are not declared static)
- Memory for automatic data is allocated on the runtime stack
- Memory for automatic data is allocated when execution reaches the scope of the variable’s declaration
- Memory for automatic data is deallocated when execution leaves that scope
Static data

- Static data is called "static" because it essentially exists during the entire execution of your program
  - Examples of static data in Java and C++: variables declared static (or, in C++, variables declared outside any function and class body)
- Memory for static data is allocated in the static data segment
- Memory for static data is allocated when your program starts, or when execution reaches the static data declaration for the first time
- Memory for static data is deallocated when your program exits
Dynamic data

- Dynamic data is called "dynamic" because it is allocated (and, in C/C++, destroyed) under explicit programmer control
  - Example of dynamic data in Java: *all* objects (instances of a class), including arrays. Dynamic data is created using the `new` operator.
  - Example of dynamic data in C++: any variables created using the `new` operator. In C++, any type variable can be created using `new`.
- Memory for dynamic data is allocated from the "heap"
- In Java, dynamic data exists from the time `new` is called until the object is reclaimed by the garbage collector.
- In C++, dynamic data exists from the time `new` is called until it is deallocated with the `delete` operator.
Example of dynamic objects in C++

Consider this C++ function. The object created with new is dynamic and must be explicitly destroyed with delete. Other variables (i, j, x) are automatic.

```cpp
int foo(int i) {
    int j = i + 1;
    C* x;
    x = new C();
    x->setA(j);
    j = x->getA();
    delete x;
    return j;
}
```
Example of automatic objects in C++

• Now consider this C++ function.
• Here \( x \) refers directly to an object, an instance of the class \( C \) (not a pointer!), created automatically on the stack, and initialized with \( C \)'s default constructor. It will automatically be destroyed when the function returns. Other variables \( (i,j) \) are also automatic.

```cpp
int foo(int i) {
  int j = i + 1;
  C x;
  x.setA(j);
  j = x.getA();
  return j;
}
```
• Note the use of the ordinary C++ member access operator .
• Note that Java does not permit creating objects as automatic data on the stack like this
Dynamic data bugs

• Automatic and static data is created and destroyed automatically. As a programmer, you should know how and when this happens, but you do not need to take steps to make it happen.

• Because the standard C++ runtime environment does not have a garbage collector, both allocation and deallocation of dynamic data must be done under programmer control.

• This leads to common bugs that can be very hard to track down.

• We’ll look at these types of bugs:
  • Inaccessible objects
  • Memory leaks
  • Dangling pointers
Inaccessible objects

The inaccessible object bug: changing the value of the only pointer to an object, so you can’t access the object anymore

```c
thing* p = new thing();
thing* q = new thing();
p = q;  // the first thing is now inaccessible
```
Memory leaks

Memory leaks: forgetting to delete dynamic data (often related to inaccessible objects)

- (these are hard to get in Java, because of its garbage collector)

```java
void foo() {
    double* q = new double(3.0);
    /* stuff, but no delete q */
}
...
for(i=0;i<1000000;i++) foo(); // massive memory leak!
```
Dangling pointers

A "dangling pointer" is a pointer variable that contains a non-null address that is no longer valid... the pointer isn’t null, but it isn’t pointing to a valid object either

• (Hard to get in Java, with no pointers to automatic variables, and good garbage collection of dynamic variables)

```cpp
int* bar() {
    int i;       // automatic variable, allocated on stack
    return &i;  // return pointer to automatic variable...
    // bad news! Stack frame is popped upon return
}
```

```
int* p;
int* q;
p = new int(99);
q = p;
delete p;    // q and p are now dangling pointers
p = nullptr;  // q is still dangling!
```
Writing, compiling, and running programs in Java and C++

• The traditional first example when learning a programming language is the simplest possible program that performs some output: a console application that says “Hello world!”
• Let’s look at and compare Java and C++ “Hello world” programs
• In Java:

```java
public class HelloWorld {
    public static void main (String args[]) {
        System.out.println("Hello World!");
    }
}
```
Compiling and running “Hello World” in Java

- In Java, all code must be inside the body of a class definition; in this case, there is just one class, a public class named `HelloWorld`.
- A public class definition has to appear in a file with the same name as the class, and a .java extension; in this case, `HelloWorld.java`.
- The public class `HelloWorld` contains a public static method named `main` that takes an array of Strings as argument:
  - That `main` method is the entry point when `HelloWorld` is run as a program.
- From the command line, compile a Java source code file using the `javac` compiler: `javac HelloWorld.java`
  - ... which will produce a file `HelloWorld.class` of Java Virtual Machine bytecodes.
- Run the program by running the `java` interpreter, telling it the name of the class you want to run as a program:
  - `java HelloWorld`
  - ... which will print to standard output: `Hello World!`
“Hello World” in C++

A simple version of a “Hello World” console application in C++:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```
Compiling and running “Hello World” in C++

- In C++, code can appear outside a class definition (as in that example)
- In C++, there is no requirement about the name of a source code file, but it is conventional to use a .cpp extension; for example, `hw.cpp`
- A C++ program must contain exactly one function named `main` which returns an `int`
  - That `main` function is the entry point of the program
- From the command line, compile a C++ source code file using the `g++` compiler: `g++ hw.cpp`
  - ... which will produce a file `a.out` of executable native machine code
- Run the program:
  - `a.out`
  - ... which will print to standard output:
    - `Hello World!`
Parts of the hello world program in C++

```cpp
#include <iostream>

int main () {

    // This #include directive tells the compiler to read in the contents of the
    // system header file `iostream` at this point. This file contains declarations
    // of variables and functions necessary for doing I/O in C++.

    std::cout << "Hello World!" << std::endl;

    // Send the string “Hello World!” to standard output, followed by end-of-line. << is
    // the bitwise left-shift operator, overloaded to implement stream I/O. The std:: namespace
    // qualifier (put using namespace std; near the top of the file to avoid it).

    return 0;

    // Since main is declared to return an int, we better do so. Returning from main
    // ends your program; it is traditional to return 0 if it is terminating normally, and to
    // return a nonzero value if it is terminating because of some error condition.
```
The C++ compiler

- We will use the GNU C++ compiler, `g++`
- This is a very good C/C++ compiler, freely available for many platforms
- (Other C++ compilers may differ in some features)
- If you have a .cpp file that contains a definition of main(), and that uses only functions defined in it or included from the standard libraries, you can just compile it as was shown, creating an executable file named a.out
- If you want another name for the resulting executable, you can use the `-o` flag, with the name you want. This will compile hw.c and create an executable named hello:
  ```
g++ hw.c -o hello
  ```
- If you want to include symbolic information in the executable file which is useful for debugging, add the `-g` flag:
  ```
g++ -g hw.c -o hello
  ```
- If your C++ program involves several files, you can compile them separately, and link them to produce an executable file. More on that later!
C++ primitive types

• The C++ language provides these basic types:
  • integer types (these are signed; put unsigned in front to specify unsigned):
    - char at least 8 bits (it’s exactly 8 bits on every platform I’m aware of)
    - short at least 16 bits
    - int at least as large as short (16 bits on DOS and Win 3.1, 32 bits elsewhere)
    - long at least 32 bits
    - bool 8 bits, and acts exactly like an integer type
  • floating-point types:
    - float at least 6 decimal digits of precision and magnitude $10^{-38}$ to $10^{38}$
    - double at least 10 decimal digits of precision and magnitude $10^{-38}$ to $10^{38}$
    - long double at least 10 decimal digits of precision and magnitude $10^{-38}$ to $10^{38}$
• You can also create types based on these basic types and on user-defined types:
  • pointers
  • classes, structures and unions
  • arrays
  • enumerations
The size of variables in C++

- C++ doesn’t specify the size of its types as precisely as Java does.
- For example, an `int` in a C++ program might be 16 bits, or 32 bits, or even 64 bits.
- C++ provides a way to tell how much memory a type takes: the `sizeof` operator.
- For any typename `T` (even a user-defined type), the expression `sizeof(T)` has an integer value equal to the number of bytes (not bits) that it takes to store a variable of type `T`.
- For any variable `x`, the expression `sizeof(x)` has an integer value equal to the number of bytes that it takes to store `x`.
- So, for example, on a typical UNIX system, in a C program with this declaration:
  ```c
  int num;
  ```
  what would be the values of these expressions?
  ```
  sizeof(int)
  sizeof(num)
  ```
Variable declaration statements in C++

- Variable declaration statements in C++ have a syntax basically similar to Java.
- However, they have very different semantics in some cases.
- For example, suppose C is the name of a class. Then:
  - In Java:
    ```
    C x;
    ```
    declares x to be of type C, but does not create an object that is an instance of the class C. (It creates a pointer that can point to such an object, but does not create any object.)
  - On the other hand, in C++:
    ```
    C x;
    ```
    declares x to be of type C, and creates an object that is an instance of the class C, using the default constructor of the class. x directly names this object; it is not a pointer.
  - To put it another way, C++ treats primitive types and classes similarly, whereas Java treats them differently. Compare the semantics of these declarations in C++ vs. Java:
    ```
    C x;
    int a;
    ```
Arithmetic and boolean operators in C++

- C++ has all the arithmetic operators that Java has, and they work similarly:
  
  + - * / % ++ --

- C++ has all the comparison and boolean operators that Java has, too:
  
  < > == >= <= && || !

  - However, these operators work a bit differently, because of how C++ deals with boolean values

  - One important feature of C++ operators is that they can be overloaded: you can write your own definitions of them
Iterative and conditional control constructs in C++

- C++ has all the iterative and conditional control constructs that Java has:
  
  if if-else while do-while for switch

- These work basically like their Java counterparts, except for some differences related to how C++ handles boolean expressions
Boolean expressions in C++

• C++ provides a primitive type `bool`, and literals `true`, `false`
• However, in every context these are equivalent to integral values where 0 means "false", and any nonzero value means "true"
• Assuming the following declarations...

```c++
int num = 3;
double x = 4.0;
```
• ... what are the values of these expressions?

```
num == 3 1
x > num 1
x > num && num == 3 1
!(x > num) && num == 3 0
x > num && num == 3 1
1 && 37 1
x > num < 0 0
```
Boolean expressions in C++, cont’d

• The fact that integer values are interpreted as booleans means you can write some terse (and maybe hard to understand) code in C++. For example, in the recursive version of factorial, you could write:

```c++
// precondition: n >= 0
int factorial(int n) {
    if(n) return (n * factorial(n-1));
    else return 1;
}
```

• It also means you can introduce hard-to-see bugs. What does the following version of factorial do (note this will compile in C++ but not Java)?

```c++
// precondition: n >= 0
int factorial(int n) {
    if(n=0) return 1;
    else return (n * factorial(n-1));
}
```
Pointers in C++

- In Java, an object is always referred to via a pointer variable that points to it; on the other hand, a primitive type value is always referred to directly, and never referred to via a pointer variable that points to it.
- In C++, you have a choice whether to refer to an instance of any type directly, or via a pointer.
- In particular, C++, you can create a pointer that points to any type you want (even another pointer), and manipulate the values stored in the pointers, and in the memory they point to, in absolute detail.
- This is an extremely powerful feature of C and C++, and it exposes some of the implementation details of pointers that are abstracted in Java.
Contents and addresses

- Every “memory cell” in a computer has two attributes:
  - a value (the contents of the cell)
  - the address of the cell
- In a typical machine, the smallest addressable cell contains 8 bits (one byte)
- A cell may also have a variable name (identifier) associated with it in a program
  - you can think of the variable name as an abstraction of the address

```c
int a= 5, b= -999; // create 2 integer variables
```

<table>
<thead>
<tr>
<th>address</th>
<th>memory cell</th>
<th>identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>512000</td>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>512004</td>
<td>-999</td>
<td>b</td>
</tr>
</tbody>
</table>
Pointers

• A pointer is a variable whose value is a memory address
• To declare and create a pointer variable in C++, use a declaration statement of the form
  
  `<typename> * <identifier>`
  
• This creates a pointer to a variable of type `<typename>`, and makes `<identifier>` the name of this pointer variable

For example:

  ```
  int * pt1;       // create a pointer-to-int named pt1
  int * pt1, * pt2; // create two pointer-to-ints, named pt1, pt2
  int * pt1, pt2;  // create one pointer-to-int, and one int!!!
  ```
Making pointers point

The “address-of” prefix operator & applied to a variable returns the memory address of the variable

```c
int a=5, b= -999;
int* pt1 = &a;  // initialize pt1 to point to a
```

<table>
<thead>
<tr>
<th>address</th>
<th>memory cell</th>
<th>identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>512000</td>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>512004</td>
<td>-999</td>
<td>b</td>
</tr>
<tr>
<td>512008</td>
<td>512000</td>
<td>pt1</td>
</tr>
</tbody>
</table>
An abstract picture of memory

- A useful abstract way to draw the contents of memory after those statements execute is shown here. The exact value of the address in the pointer is not shown. Instead, the fact that the pointer named \texttt{pt1} points to the variable named \texttt{a} is shown with an arrow:

\begin{verbatim}
int a=5, b= -999;
int* pt1 = &a;             // initialize pt1 to point to a
\end{verbatim}

Note: this picture is impossible in Java, which does not allow pointers to point directly to primitive type variables
Dereferencing pointers

• The “dereferencing” prefix operator * applied to a pointer value “goes to” the location in memory pointed to by the pointer

```cpp
int a = 5, b = -999;
int* pt1 = &a; // declare and initialize pt1 to point to a
cout << *pt1; // print the value of what pt1 points to: 5
*pt1 = 10; // assign a value to what pt1 points to
cout << a; // print the value of a, which is...
```

• In C++ you can have pointers to anything! For example you can create a pointer to a pointer to an int:

```cpp
int ** pp = &pt1; // declare and initialize pp to point to pt1
**pp = 20; // assign a value to what pp points to points to...
cout << a; // print the value of a, which is...
```
References in C++

• Besides having pointers to variables, C++ also permits creating *references* to existing variables
• References are especially useful for function parameters, in that they allow passing arguments to functions without copying the arguments
• (The same effect can be had with pointers, but the syntax of references is simpler.)
• It is important to understand the difference between pointers and references...
Next time

- References in C++
- The “const” qualifier
- Arrays, pointers and pointer arithmetic
- The interface/implementation distinction in C++
- C++ class templates
- “friend” visibility

Reading: Weiss Ch 1