Lecture 2

- An introduction to C++
- Comparisons of Java to C++
- Basic C++ programming
- C++ primitive types and operators
- Arrays, pointers and pointer arithmetic
- The interface/implementation distinction in C++
- C++ class templates

Reading: Weiss Ch 1

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:

```
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:

```
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++:

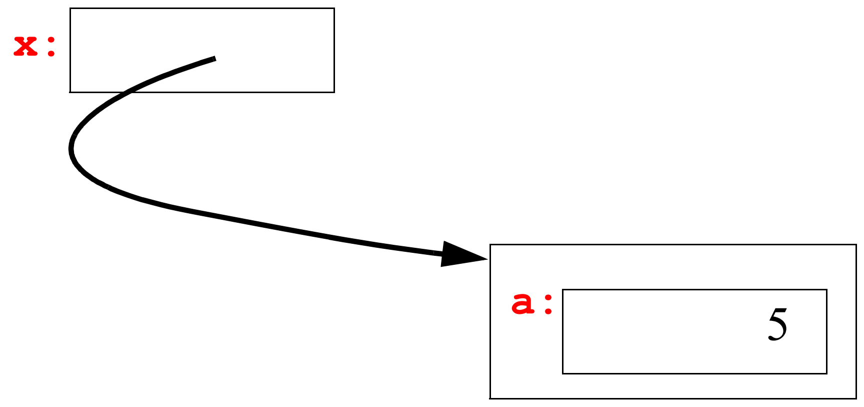
```
class C {
   private:
    int a;
   public:
    void setA(int a) { this->a = a; }
    int getA() { return a; }
};
```

- And writing ‘the same’ program:

```
#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 (Java or C++) 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:



    - Always be aware of the difference between a pointer and what it points to!

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 `*`
  - 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: `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:

```
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:

```
C* x = new C();  // create an object using new
x = 0;  // 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

```
C* x = new C();   // create an object using new
...               // do things with the object
delete x;         // free the object using delete
x = 0;            // 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 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
- 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 it is allocated with the **`new`** operator 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.

```
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 the default constructor. It will automatically be destroyed when the function returns. Other variables (`i`,`j`) are also automatic.

```
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 automatically 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

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

```
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 automatic garbage collection)

```
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 = 0;                 // q is still dangling!
```

Writing, compiling, and running programs in Java and C++

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

```
#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++

```
#include <iostream>
```

- 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++.

```
int main () {
```

- To run, every C++ program must contain a definition of a function named “main”. When the program starts, the operating system calls this function. main() is declared to return an int value. Here, we are not using any arguments to main, but you can, if you want to process command line arguments.

```
    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::` is a 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:
  `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...

```
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
1  && 37                                  1
x > num > 0                               1
```

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:

```
// 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)?

```
// 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

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

address	*memory cell*	*identifier*
512000	5	`a`
512004	-999	`b`

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

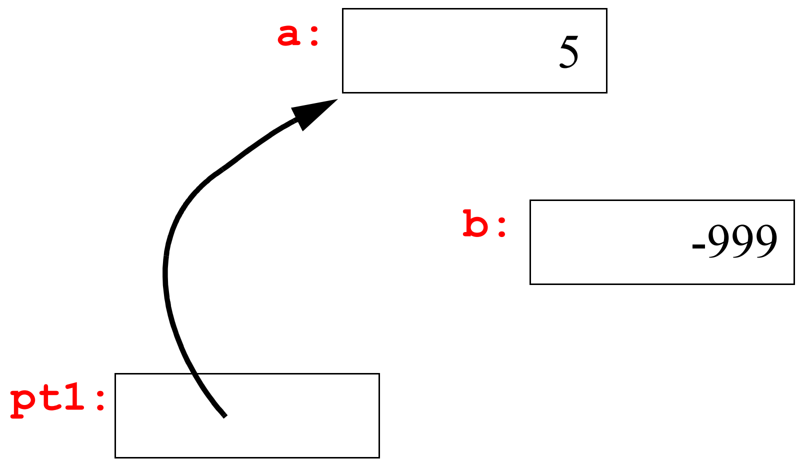
```
int a=5, b= -999;

int* pt1 = &a;    // initialize pt1 to point to a
```

address	*memory cell*	*identifier*
512000	5	`a`
512004	-999	`b`
512008	**512000**	`pt1`

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 **pt1** points to the variable named **a** is shown with an arrow:

```
int a=5, b= -999;

int* pt1 = &a;    // initialize pt1 to point to a
```



- 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

```
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:

```
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
- To declare and create and initialize a reference variable in C++, use a declaration statement of the form

```
<typename> & <identifier> = <existing_variable>
```

- This creates a reference to `<existing_variable>`, which must be of type `<typename>`, and makes `<identifier>` the name of this reference
- A reference is an *alias*: it provides another name for an existing variable

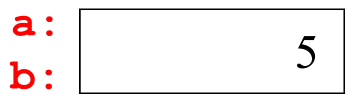
- For example:

```
int a = 5;

int & b = a;     // b and a now both refer to the same variable

b = 6;

std::cout << a << std::endl;   // prints 6
```

An abstract memory picture of references

- A reference is an alias: it sets up another name for an existing variable

```
int a = 5;

int & b = a;    // b is now an alias for a
```



- Note: this picture is impossible in Java, which does not allow multiple names for the same variable
- (Java does allow multiple pointer variables to point to the same object, but that is not the same thing at all!)

Reference function parameters

- References are most useful in application to function parameters, where they permit true pass-by-reference semantics, avoiding any argument copying
- For example, you can easily write a true swap function using reference parameters:

```
void swap(int& a, int& b) {
   int tmp = a;  a = b;  b = tmp;
}
------------
int x = 5, y = 10;
swap(x,y);
std::cout << x << ", " << y << std::endl; // prints 10, 5
```

- Note that the same effect can be had by using pointers, but with messier syntax:

```
void swap(int* a, int* b) {
   int tmp = *a;  *a = *b;  *b = tmp;
}
------------
int x = 5, y = 10;
swap(&x,&y);
std::cout << x << ", " << y << std::endl; // prints 10, 5
```

The const label in C++

- C++ permits specifying variables (including formal parameters to functions), and member functions of classes, as `const`
- This can be used in several ways. Examples:
  - Declaring a variable to be const means that its value cannot change; therefore it must be given a value in its declaration:

    ```
    const double TEMP = 98.6;  // TEMP is a const double
    ```

    (Equivalent: `double const TEMP = 98.6;`)

  - Declaring a pointer to be a pointer to a const means that the pointer can't be used to change the value of anything it points to:

    ```
    int const * ptr;   // ptr is a pointer to a const int
    ```

    (Equivalent: `const int * ptr;`)

- (More on const-labeled parameters and member functions later.)

Reading C++ type declarations

- C++ type declarations can be more easily understood when read ‘backwards’, that is right to left, keeping in mind that `*` means ‘pointer to’ and `&` means ‘reference to’:
- Examples:

```
int * p        // p is a pointer an int
int const * p // p is a pointer to a const int
               // which means the same thing as...
const int * p // p is a pointer to an int that is const
int * const p // p is a const pointer to an int
int const * const p // p is a const pointer to a const int
                     // which means the same thing as...
const int * const p // p is a const pointer to an int that is const
int & p        // p is a reference an int
int const & p // p is a reference to a const int
               // which means the same thing as...
const int & p // p is a reference to an int that is const
```

- Note that `*` and `&` mean something different if they occur in expressions instead of type declarations:
  - In expressions, these are unary prefix operators!
  - `*` is the pointer dereference operator, and `&` is the address-of operator

Pointers to const, and const pointers

- Creating a pointer to a const...

```
const int c = 3;   // create a const int variable
                   // equivalent:  int const c = 3;
const int * p;     // create a pointer to a const int
                   // equivalent:  int const * p;
p = &c;            // make p point to c

*p = 4;            // ERROR: p is pointer to const
```

- Creating a const pointer...

```
int d = 5;        // create an int variable

int * const q = &d;  // create and initialize a const ptr to int

*q  = 9;          // okay... changing what q points to

q = &d;           // ERROR:  q itself is const!
```

Pointers to consts and const pointers...

```
const int c = 3;  // create a const int variable
int d = 5;        // create an (ordinary, non-const) int variable

const int * p;     // create a pointer to a const int
int * const q;     // create a const pointer to an int
int * r;           // create a pointer to an int

p = &c;            // okay:  p is pointer to const
p = &d;            // okay:  increasing “constness”
*p = 9;            // ERROR:  can’t use p to change a value
d = 9;             // okay

r = &d;            // okay:  r is pointer to an (ordinary) int
r = &c;            // ERROR:  decreasing “constness”

q = &c;            // ERROR: can’t assign to const pointer!

const int * const s = &c;  // create and initialize
                          // a const ptr to a const int
```

References to const, and const references

- Creating a reference to a const...

```
const int c = 3;      // create a const int variable
                      // equivalent:  int const c = 3;
const int & r = c;    // create and initialize reference to it
                      // equivalent:  int const & r = c;
r = 4;                // ERROR: r is reference to const
```

- Creating a const reference...
- Actually, all references are const, in that the alias is established when the reference is created, and cannot be changed later

```
int d = 5, e = 6;         // create int variables

int & s = d;        // create and initialize a reference to d

s = e;              // just changes the value of what s refers to
                    // d, s are still aliases of the same variable
                    // which now has value 6
```

Pointer arithmetic in C++

- These arithmetic operators are defined on pointer arguments:
  `++, --, +, -, +=, -=`
- A pointer variable stores an integer number, which is a memory address
- However, arithmetic operations on a pointer variable do not necessarily change that number in the way you would expect
- For example, if `p` is a pointer, `p++` does not necessarily make the number stored in `p` one larger than it was before
  - ... it makes the number stored in `p` larger by an amount equal to `sizeof(*p)`

Pointer arithmetic, cont'd

- Following our example...

```
int a=5, b= -999;
int* pt1 = &a;     // pt1 contains address 512000

pt1++;             // now pt1 contains address 512004... why?

*pt1 = 10;         // assign a value to what pt1 points to

cout << b;         // print the value of b, which is...

pt1 = pt1 + 1;     // now pt1 contains address 512008

cout << *pt1;      // print the value at this address

pt1 = pt1 - 2;     // now pt1 contains address 512000 again
```

Pointer arithmetic, cont’d again

- When you add an integer to a pointer in C++, the result is of the same type as the pointer, and the increment is the integer times the size of the object the pointer is declared to point to
- When you subtract one pointer from another in C++, the result is of integer type, and is equal to the number of objects of type the pointers are declared to point to that would fit between where the two pointers point (fractional part discarded)
- Adding two pointers is illegal (and it doesn’t make any sense to do it)

```
double* p1;
double* p2;

p1 - 3        // the address of the 3rd “double” cell in memory
              // before the one p1 points to... this is a
              // double* valued expression!

p1 - p2       // the number of double-sized cells in memory
              // between the one p2 points to and the one p1
              // points to... this is an int-valued expression!

p1 + p2      // illegal
```

Other operators and pointers...

- These relational operators are defined for pointer arguments:

  `==, <, <=, >, >=`

- Pointer literal constants: `0` is the "null pointer" of any type
  - with `#include <cstddef>`, you can use the identifier `NULL` instead of `0`, though some C++ style standards disapprove of that
  - note that in a boolean context, the null pointer is interpreted as "false", and any other pointer value is interpreted as "true"

- Unary `!` operator:

  For any pointer expression `p`,

  `!p` is 1 if `p` is the null pointer; else 0

Arrays and pointers in C and C++

- In C or C++, an array name acts like a const pointer to the first element of the array
- C arrays are always contiguous in memory: second element is right after the first, etc.

```
int a[99];   // create an array of 99 ints
int* p;      // create a pointer-to-int

p = a;       // makes p points to the first element of a

p = & a[0];  // this also makes p point to first element of a

a = p;       // ERROR:  can't assign to an array name, it's const

p[3]         // array subscripts work on pointers...
             // this is exactly the same as *(p + 3)

*(a + 3)     // and pointer arithmetic works on array names...
             // this is exactly the same as a[3]

             // which is also exactly the same as  3[a]
             // (believe it or not!)
```

Pitfalls with arrays and pointers in in C++

- With a declaration like

  ```
  int a[100];
  ```

  expressions like `a[-33]` or `a[101]` or `*(a + 10000)` are very dubious, to say the least

- However, these are not compiler errors, and may not even give you runtime errors (unless they involve accessing protected memory)

- The power of pointer arithmetic can hurt you, if you're not careful...

- For these reasons, though C-style arrays are available in C++, it is advisable to use data structures from the Standard Template Library (STL) such as `std::vector` instead of basic arrays

Strings as arrays of chars in C++

- As in C, an array of char elements with an element `'\0'` used to mark the end of the string can be used as a string in C++
    - `'\0'` is the null character, an 8-bit integer with value 0
    - to contain a string, a char array must contain a null character
    - the null character is used to determine the end of the string contained in the array
- A null-terminated array of char can be initialized with a string literal:

```
char s[] = "hello";
```

- this creates an array `s` with 6 elements, not 5!
- the null character is included, as the last element of `s`
- this is equivalent to

```
char s[] = {'h','e','l','l','o','\0'};
```

or

```
char s[6] = "hello";
```

Dealing with strings in C++

- Since in C++ a null-terminated char array (like any array) does not keep information about its own length, extra checking must be done to make sure all accesses to the string are within bounds

- This checking is tedious to do, but if not done, leads to serious bugs and exploitable code
  - Most software security problems are due to unchecked array bounds violations (buffer overflows) in C and C++ programs!

- Again, in C++, it is advisable to use data structures from the Standard Template Library (STL): instead of null-terminated char arrays, use **`std::string`** for strings!

Command line arguments

- If you want your main function to deal with command line arguments, declare it like this:

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

or

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

- Now when main is called, `argc` will be the number of command line arguments (the arg count) and `argv` will be an array of pointers to the null-terminated arguments (the arg vector)
- (Actually, in C++, unlike Java, the name of the command is included as the first element of this array, so `argc` is always at least 1)
- So, here is a simple program that just prints the command line arguments:

```
#include <iostream>
int main(int argc, char* argv[]) {
   int i;
   for(i=1;i<argc;i++) {
      cout << argv[i] << endl;
   }
   return 0;
}
```

More on class definitions in Java and C++

- As we have seen, basic class declaration and definition in C++ is similar to Java
- Let's look now in more detail, with emphasis on constructors, destructors, and using `const`
- Consider this simple example in Java:

```
public class C {

   public C() { a = 33; }                   // default ctor

   public C(int _a) { a = _a; }             // another ctor

   public int getA() { return a; }          // accessor

   public void setA(int _a) { a = _a; }  // mutator

   private int a;                           // instance variable

}
```

Constructor definitions in C++

- In C++ that example could be:

```
class C {
public:
   C() { a = 33; }                         // default ctor
   C(int _a) { a = _a; }                   // another ctor
   int getA() const { return a; }          // accessor
   void setA(int const & _a) { a = _a; }  // mutator
private:
   int a;                                 // member variable
}; // <-- Note the semicolon!!
```

C++ constructor initializer lists

- An alternative, using constructor initializer lists (this is often preferred because it can avoid multiple constructions of the same member variable):

```
class C {

public:

   C() : a(33) { }                          // default ctor

   C(int _a) : a(_a) { }                    // another ctor

   int getA() const { return a; }           // accessor

   void setA(int const & _a) { a = _a; }    // mutator

private:

   int a;                                  // member variable

};  // <-- Note the semicolon!!
```

Destructor definitions in C++

- When a C++ object is deallocated, storage for all its member variables will be freed
- However, what if one or more of its member variables are pointers to dynamic data? That dynamic data must also be deallocated if it will no longer be used
- In C++, this is the job of a *destructor* function
- A destructor is defined as a function that looks like a constructor with no arguments, preceded by a **~**
- The destructor for an object is called when that object is deallocated, and it should be written to do all necessary memory deallocation beyond freeing the object’s instance variables themselves
- Rule of thumb: If an object creates dynamic data accessible via its member variables, that object should have a destructor that deletes that dynamic data
- That simple class we’ve been using as an example doesn’t create any dynamic data, so let’s define one that does, and define a destructor for it

A class with a destructor

```
class C {

public:
   C() : a(new int(33)) { }
   C(int _a) : a(new int(_a)) { }

   ~C()  {  delete a;  }                  // destructor


   int getA() const { return *a; }        // accessor

   void setA(int const & _a) {            // mutator
      *a = _a;
   }

private:

   int * a;          // member variable, pointer to dynamic data

};
```

Implementing modules in C++

- A *module* is a cohesive software unit that provides some services for its clients
- A well-designed module will have an abstraction barrier: the implementation is hidden behind an interface
- The client interacts with things "exported" or "exposed" through the module’s interface
- When implementing a module in C++, try to follow this convention:
  - the interface is specified in a *header file* that has ".hpp" extension
  - the implementation goes in (one or more) files that have ".cpp" extension
  - the implementation files (as well as any application files that intend to use the module) must #include the module’s header file
  - Note: if the .hpp file is not a standard system header file, put its name in quotes instead of corner brackets:

```
#include "employee.hpp"
```

Interface files vs. implementation files in C++

- To the extent possible, interface specifications go in the module's ".hpp" header file. These include:
  - declarations and definitions of exported constants
  - declarations of exported global variables
  - exported type declarations: typedefs, enums, class declarations
  - function declarations (function prototypes, not definitions!)
  - comments about the interface
  - relevant preprocessor directives (such as #includes)

- Put *definitions* of variables and functions in one or more ".cpp" files
  - function definitions
  - global variable definitions (that actually allocate storage for the variable)

- A C++ application or module that uses a module should **`#include`** the module's .hpp file to get access to the interface specification (this also goes for the module's implementation file itself, to keep the interface consistent!)

Separation of interface and implementation in C++

- Using our example, the header file c.hpp might look like this:

```
#ifndef C_HPP
#define C_HPP

class C {
   public:
      C();                       // default ctor
      C(int _a);                 // another ctor
      int getA() const;          // accessor
      void setA(int const & _a);         // mutator
   private:
      int a;                     // member variable
   };

#endif
```

- Note the use of preprocessor directive “header guard” to prevent multiple includes of this header file... Every header file should have these, with an identifier that is unique for each file

Separation of interface and implementation in C++

- Then the implementation file c.cpp might look like this:

```
#include "c.hpp"

C::C() : a(33) { }                      // default ctor

C::C(int _a) : a(_a) { }                // another ctor

int C::getA() const { return a; }       // accessor method

void C::setA(int const & _a) { a = _a; }        // mutator method
```

- Note the use of the scope resolution operator `::` to specify that these are things in the `C` class that are being defined

Separate compilation and linking in C++

- The source code files for a module can be compiled into an "object" file , with .o extension
- (One or more .o files can be archived into a single library file; see man ar)
- A client program wishing to use the module must then `#include` the module's .hpp header file, be compiled and linked to the module's .o file or library
- How to do that?

Separate compilation and linking with g++

- To compile a C++ source code file and produce an object file (not a runnable program) you specify the "`-c`" flag to the compiler

- Suppose there is a graphics module, implemented in a file `graphics.cpp`. It doesn't contain a definition of main(), so it can't produce a runnable program, but we we can compile it and produce an object file:

```
g++ -c graphics.cpp
```
compiles module graphics.cpp, producing object file graphics.o

- Then a client program that uses the graphics module (and #includes graphics.hpp), and that defines main() can be written and also separately compiled:

```
g++ -c client.cpp
```
compiles client.cpp, producing object file client.o

- To produce a runnable program, you need to link together all the needed object files. The C compiler can perform this linking step. At the same time, you can use the "-o" flag to give the result a different name than a.out:

```
g++ client.o graphics.o -o cmd
```
links object files, producing executable cmd

Java generics and C++ templates

- A Java class can be made generic. Using our example:

```
public class C<T>  {
      public C() { a = null; }             // default ctor
      public C(T _a) { a = _a; }           // another ctor
      public T getA() { return a; }        // accessor method
      public void setA(T _a) { a = _a; }  // mutator method
      private T a;                         // instance variable
   }
```

- A similar effect can be had using C++ templates...

A C++ class template

- In C++, that would look like:

```
template <typename T>
class C {

public:

   C() : a(0) {  }                    // default ctor

   C(T* _a) : a(_a) {  }              // another ctor

   T* getA() const { return a; }      // accessor member function

   void setA(T* _a) { a = _a; }       // mutator member function

private:

   T* a;                              // member variable

};
```

- This class does not have a destructor. Does it need one? Why or why not?

C++ templates and separate compilation

- From the point of good modular software design, it is desirable to put interface declarations in .hpp header files, implementation definitions in .cpp files, separately compile, and link to create an application
- However with the current state of C++ compilers, this is not possible when using class templates!
- With current compilers, the complete definition of a class template must be visible to client code when the client code is compiled
- So, what is often done is:
  - put the entire class template definition in a .hpp file
  - `#include` that .hpp file in every client implementation file that requires it
- This situation is less than ideal, but there is no better solution at the present time

Next time

- Binary search trees
- Toward a binary search tree implementation using C++ templates
- C++ iterators and the binary search tree successor function
- Binary search tree average cost analysis

Reading: Weiss, Ch 4