Today’s topics: continuing the introduction to C++, again assuming you already know Java
- pointers and references
- functions and parameter passing
- templates
- introduction to the standard library (STL)
- STL strings

Goals:
- sufficient coverage for first project
- introduce main topics; read details on your own

Next week: objects, classes, inheritance

I. Pointers

Pointers are useful constructs for building data structures
- objects may contain references to other objects
- a data member of an object instance can be a pointer to another instance

Pointer Types in C++

To declare a variable as a pointer, use a * in the type name:
- string s; // s is a string object
- string *sp; // sp is a pointer to a string

Note the placement of the space. Syntactically, the following are identical, but the first form is the idiom:
- string *sp;
- string* sp;

Assigning Values to Pointers

To assign a value to a pointer, use the “address of” operator
- int i = 4;
- int *pi = &i; // p points to i

You can also assign one pointer value to another
- int *p = &i;
- int *q;
- ...
- q = p; // p and q both point to i

Using the Heap

A third way to assign to a pointer is to create an item on the heap
- int *p = new int;
- new is an operator that takes a type as an operand
- It allocates an instance of that type on the heap and returns the address of the new item
Null Pointer

- C++ uses the constant 0 as the null pointer
  
  ```cpp
  int *p = 0;  // initialize to null
  ```
- 0 is not a valid address -- the O/S never stores any object at address 0
- new returns 0 if it could not allocate the requested item
  
  ```cpp
  if (p = new int) {
      ...
  }
  ```

Dereferencing Pointers

- If a pointer has been assigned a value (i.e. it points at something) you can dereference the pointer by using * as a unary operator
  
  ```cpp
  int n = 26;
  int *p = &n;
  cout << (*p + 2);  // prints 28
  *p += 1;  // sets n to 27
  ```

Arrays and Pointers

- When you define an array in C++ you are actually defining a pointer
  
  ```cpp
  int a[10];  // a is an array of 10 ints
  int *pa = a;  // a is also a pointer
  ```
- Allocating an array on the heap:
  
  ```cpp
  int *pb = new int[10];
  for (int i = 0; i < 10; i++)
      pb[i] = i;
  ```
  
  pb[i] means "the address i items past where pb points"

Pointer Arithmetic

- You can add or subtract constants
  
  ```cpp
  int *a = new int[10];
  int *b = a + 2;
  ```
- The compiler does automatic scaling
  
  ```cpp
  string *sa = new string[10];
  string *sb = sa + 2;
  ```

Pointer Operations

- Other things you can do with pointers:
  
  ```cpp
  char *s;
  char *t;
  if (s == t) { ... }  // same address?
  if (s != t) { ... }  // different address?
  int n = s-t;  // how far apart?
  ```
  
  But not
  
  ```cpp
  if (s < t) { ... }  // unsafe
  s = s + t;  // meaningless
  ```
Deallocating Heap Items

✓ When you are done using an item previously created by calling new, make sure you deallocate it
  string *sp = new string;
  delete sp;
✓ The call to delete frees the space used by the item pointed to by sp
✓ Be careful! Do not use sp after calling delete. Idiom:
  delete sp;
  sp = 0;

Deallocating Arrays

✓ Since the compiler can’t always distinguish between pointers to single items and pointers to arrays, use delete[] to deallocate arrays
  int *a = new int[100];
  delete[] a;
  a = 0;
  // If you create it with [], deallocate it with delete[]

References

✓ C++ pointers, and the unary operators * and &, trace their roots back to C
✓ C++ has an additional reference type that is also based on the concept of pointing to another item
✓ A reference is a pointer that
  - must be assigned when it is created
  - can not be changed once it is assigned

Reference Declarations

✓ Use the & symbol when declaring reference variables
  int i;
  int &j = i;
✓ Note:
  - the type of j is “reference to an int”
  - j is assigned a value when it is defined
  - the value given to j is i, not &i (the address of i)
✓ References are mostly used when declaring function parameters and return types (next topic)

II. Functions

✓ As in C and Java, subroutines ("procedures") are called functions
✓ In a function declaration, the header defines the name, return type, and parameter list
  double square(double x) {}  
  string append(string s1, string s2) {} 
  void insert(string s, string t, int &i) {} 

Return Type

✓ Functions can return values of any type
✓ Use a return statement in the body of a function to pass a value back to the calling context:
  double square(double x) {
    return x*x;
  }
  double x = 13.5;
  double z = square(x);
✓ The type of the return expression must match the type of the function
Void Functions

- If a function does not return a value -- i.e. it is called for the side effects it produces -- it's type should be declared as void and it should not return a value
- void printIfPositive(int x) {
  if (x < 0)
    return; // note no expression
} 
- printIfPositive(-16);

Ignoring the Return Value

- As in C and Java, it is OK for the caller to ignore the return value (i.e. expressions are valid statements)
- Example: insert might return the new string:

```
string insert(string s, string t, int i) {
  // insert t into s at position i
  return s;
}
```
- string greeting = "Hello, world";
  insert(greeting, "cruel ", 7);

Implied Type

- For compatibility with C, early C++ standards allowed programmers to leave off the return type in a function declaration
- insert(string s, string t, int i) {
  // insert t into s at location i
} 
- The implied type of these functions is int, not void
- This is a syntax error:
- string s = insert("Hello, world","cruel ",7);
- The compiler thinks you want to assign an integer value to a string variable

Style

- Always declare a type for a function
- If the function will not return values, declare it to be a void function
- Preferred layout:
  ```
  double power(double x, int n) {
    // body indented w.r.t. header
  }
  ```
- All parameters on one line
- Note open brace on the next line

Parameters

- As in C and Java, all parameters are passed by value
  ```
  double square(double x) {
    return x*x;
  }
  ```
  copy the value of x, pass it to square; returns 16
  ```
  double y = square(4);
  ```

Call By Reference

- It is possible to use pointers to implement the effects of "call by reference"
  ```
  void inc(int *i) {
    *i += 1;
  }
  ```
  ```
  int n = 3;
  ```
  ```
  inc(&n);
  ```
  copy << 'n' << n << endl;
  ```
  ```
  Note the procedure is passed a pointer to n
  ```
  ```
  Prints "n = 4"
Reference Parameters

✓ A better way to implement call by reference in C++ is to define the parameter as a reference type

```cpp
void inc(int &i) {
    i += 1;
}
```

```cpp
int n = 3;
inc(n);
```

✓ Note:
- the type of parameter `i` is `&i`
- do not “dereference” `i` in the body of `inc`
- do not pass the address of `n` in the call to `inc`

References Are Efficient

✓ An advantage of passing parameters by reference is that for large objects it is much more efficient to pass the address than to copy the object

```cpp
string lookup(string s, table t) {
    // lookup can’t call any method of the table class that might modify t
}
```

✓ Not an issue in Java: all objects are stored on the heap and identified by references, so no need for a `&` notation

References Are Security Risks

✓ A disadvantage of passing parameters by reference is that objects passed to a procedure might be modified

```cpp
string lookup(string s, table t) {
    // erase s? add a new item?
}
```

References as Return Types

✓ It is also possible to return references from functions

```cpp
int larger(int &a, int &b) {
    return (a > b) ? a : b;
}
```

```
int i = 5;
int j = 10;
larger(i, j) = 0;
cout << "i = " << i << " j = " << j << endl;
```

Constants

✓ C++ allows you to flag variables as constants

```cpp
string lookup(string s, const table t) {
    // lookup can’t call any method of the table class that might modify t
}
```

✓ More on `const` declarations in a later lecture

Default Parameters

✓ When declaring a function, you can give default values for the parameters

```cpp
string greeting(string s = “world”) {
    return “Hello, “ + s;
}
```

```cpp
cout << greeting() << endl;
```

```
Hello, world
cout << greeting(“class”) << endl;
Hello, class
```
Overloading Function Names

- C++, like Java, allows you to overload function names
- The compiler distinguishes between functions by the combination of arguments
- Example: assume Circle and Square are classes
double area(Circle c) {
  return c.radius * c.radius * pi;
}
double area(Square s) {
  return s.edge * s.edge;
}
cout << "area: " << area(Circle(3)) << endl;

Overloading Operators

- C++ goes one step further than Java
- It allows you to overload built-in operators, as well
- Suppose you are defining a class for sets
- You can define a new meaning of the binary * operator so it invokes a method that computes the intersection of two sets
  Set s1, s2, s3;
  s1.insert("hello");
  --
  s3 = s1 * s2;

Overloading Operators (cont'd)

- You cannot define new symbols -- you can only add new meanings to symbols the compiler already knows
- int n = 4!; // sorry, I isn't a postfix operator
- You cannot alter the precedence of operators
- Very useful for output: overload the << operator
cout << "intersection = " << s3 << endl;

* This feature is too often abused -- restrain yourself

III. Templates

- C++ is a strongly typed language
- This makes it difficult to write functions that can handle a variety of different types of parameters
- Example: find the larger of two numbers
  int i, j;
  double x, y;
  int n = max(i,j);
  double z = max(x,y);

Generic Functions via Overloading

- You could write two different max functions
  int max(int a, int b) {
    return (a > b) ? a : b;
  }
double max(double a, double b) {
  return (a > b) ? a : b;
}
- But this is a pain, especially if you have lots of types
  string max(string a, string b) {
    return (a > b) ? a : b;
  }

Generic Functions via Runtime Checks

- If you could determine the type of an item at runtime, you could write one function
- Example, in Java:
  Object max(Object a, Object b) {
    if ((a instanceof Integer) &&
      (b instanceof Integer)) {
      //
    } else {
      Object result = a;
      //
    }
- But C++ is obsessed with having the compiler generate efficient code by knowing types at compile time
Template Definition

✓ The solution: *templates*

✓ Write one piece of code which has a *type* as a parameter

```cpp
template<class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

✓ Now `max` is a general rule for creating several different functions

✓ The compiler generates as many actual functions as the program needs

Why Templates in CIS 399?

✓ We will not use templates directly this term

✓ You are encouraged to learn more about them, but they are not necessary for our projects

But:

✓ The C++ Standard Library is based on templates

✓ STL = Standard Template Library

✓ Template syntax is used to declare library objects, and you need to know more about templates if you are going to write more complex programs

IV. Introduction to STL

✓ The C++ Standard Template Library provides most of the same container classes as the Java API
  - strings
  - lists
  - sets
  - maps

✓ It also has some very powerful abstractions used in *generic programming*
  - iterators
  - *common algorithms* (find, etc)

STL Headers

✓ The STL has separate headers for the various classes, so you may need to include several different headers

✓ STL header files do not have a .h extension

```cpp
#include <iostream.h>
#include <iostream>
```

✓ Do not mix STL and old-style header files

✓ If you use any STL classes, use all new headers

Namespace

✓ Another new feature of C++ used by the STL is the idea of a *namespace*

✓ Namespaces allow you to break very large applications into smaller pieces

✓ Example: a large application might use several libraries
  - X-window system
  - Graphics libraries
  - Networking libraries

✓ Without namespaces, if you use a library, all the names from the library become part of your program when you include the library header

Namespace std

✓ The STL uses a namespace called `std`

✓ You have two choices when you write programs that use the STL

(1) Make your program part of `std` as well

```cpp
#include <iostream>
using namespace std;
```

(2) Better: import the names you need from `std`

```cpp
#include <iostream>
using std::cout;
```
Aside: Name Conventions

A common convention for identifiers in C++ is
- use lower case letters to start variable and function names
  - i, j, max
- use upper case letters in the middle of long names
  - countDifferences
- start class names with upper case letters
  - Protein, GeneticCode
- use all upper case for constants
  - HAMPTON, MITOCHONDRIAL
- STL class names use all lower case
  - string, list, map, ...

To Be Continued

We'll spend a lot more time on the STL later in the course
For this week's project, you only need to know about the
string class...

V. Strings

To use strings, include the header file and import the class
name:
- #include <string>
  using std::string;

STL string objects are not arrays of chars

You cannot pass string objects directly to the old C and
C++ string library functions strlen, strpos, etc.

String Constructors

There are several ways to create string variables
- string s;
  an initially empty local variable
- string *ps;
  a pointer, initially null
- string *ps = new string;
  ps now points to an empty string on the heap
- string a("Hello");
  specifies the initial contents of the string
- string *ps = new string("Hello");
  allocate and initialize

String Methods

Read the STL documentation for a complete list of the
methods of the string class.
Some of the more useful ones for the first project (assume
a and t are strings):
- a.length();
  the number of characters currently in a
- a + t;
  the concatenation of a and t
- a += t;
  append t to the end of a

String Literals

The C++ compiler treats string literals as null-terminated
arrays of characters
This is the old C/C++ technique for representing strings
The string class methods accept string literals and
characters as arguments
- a = "Hello";
  initialize a string
- a += ", world";
  append a literal string
- a += 'I';
  append a character
Strings and Functions

Since you can assign a literal string to a string object, you can pass a literal string to a function that expects a string argument:

```cpp
string repeat(string s, int n) {
    string res;
    for (int i = 0; i < n; i++)
        res += s;
    return res;
}
```

```cpp
cout << repeat("ha", 3) << endl;
```

Output: `hahaha`

A Final Word on the STL

STL objects are copied when they are assigned:

```cpp
string s = "hi";
string t = s;    // t is a copy of s
s += "t";
cout << "s = " << s << " t = " << t << endl;
```

Compare this to Java, in which `s` and `t` are references to the same object:

Challenge: count the number of string objects created by the call to `repeat` on the previous slide.

Next...

Classes in C++