CSE100

Advanced Data Structures

Lecture 3

(Based on Paul Kube course materials)
Lecture 3

- 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
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...
References in C++

- 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

```java
int a = 5;
int & b = a; // b is now an alias for a
```

- a: ______ 5
- b: ______

• 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:

```cpp
void swap(int& a, int& b) {
    int tmp = a; a = b; b = tmp;
}
```

```cpp
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:

```cpp
void swap(int* a, int* b) {
    int tmp = *a; *a = *b; *b = tmp;
}
```

```cpp
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:
    ```cpp
class const double TEMP = 98.6; // TEMP is a const double
    ( Equivalent: double const TEMP = 98.6; )
    ```
  - Declaring a pointer to be `const` means that the pointer can’t be used to change the value of anything it points to:
    ```cpp
    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’:
  
  ```
  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
                  // which means the same thing as...
  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 to 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...

```c
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...

```c
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...

```c
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...

```c
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

```c
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 and 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)`
Following our example...

```cpp
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).

```c
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: `nullptr` is the “null pointer” of any type
  
  - `nullptr` was introduced in the C++11 standard
  - 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.

```c
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 C++

• With a declaration like
  ```cpp
  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` or `std::array` 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:

```cpp
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
  ```cpp
  char s[] = {'h','e','l','l','o','\0'};
  ```
  or
  ```cpp
  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:
  ```c
  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:
  ```c
  #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:

```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:

```cpp
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):

```cpp
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
    // created internally, in this class
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:

```cpp
#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:

```cpp
#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 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:

```java
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:

```cpp
template <typename T>
class C {
public:
    C() : a(nullptr) { } // 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