ICSI 403
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 04a – Introduction to C++ Programming, part 3

J Marques de Carvalho

Last Time (1)

- Parameters and overloading
  - Pass-by-value (default) vs pass-by-reference
  - Passing *any* parameter by reference (using &)
  - Overloading
  - Default Parameters
  - Preprocessor Directives (`#define`, `#if`, etc.)
  - Assertions

Last Time (2)

- Arrays
  - Array declarations, initialization
    - Declared at compile-time with fixed (constant) size
    - Created at run-time with variable size (chapter 10)
  - Arrays as parameters
    - Array name = pointer to start of array (reference)
      - Not an object, so no `.length` parameter; no ragged arrays
    - `const` arrays can't be changed in a called function
    - Individual elements passed by value (by default)
    - Multidimensional arrays and parameter lists
      - Must explicitly state all sizes except for first dimension

Last Time (3)

- Pointers
  - “Pointer” ≡ “Address of something”
    - “Reference variables on steroids”
  - Declaring pointers
    - `int *a;`
  - Getting addresses of things with `&` operator
  - *Dereferencing a pointer* (`*p`) to get value *pointed to*
  - Pointer statement examples
  - Pointer Arithmetic
  - Pointers as data types
  - Dynamic memory allocation (`new` returns a pointer)
  - `NULL` Pointers

Structures and Classes

- One of the things that may make C/C++ look like a colossal step backwards is the structure, or `struct`
- A `struct` is like the data part of a class with no methods
- The `struct` came from C (pre-C++), but was motivated by data records

Structures and Classes

- A structure for a bank Certificate of Deposit might consist of the certificate number, the balance, the interest rate, and the term:


```
struct CDAccount
{
    long   certNumber;
    double balance;
    double interestRate;
    int    termInMonths;
};
```

Now `CDAccount` is a data type, just like `int`

Structures and Classes

- We can also create instance(s) of the structure on the same statement in which we define the structure:

```
struct CDAccount
{
    long   certNumber;
    double balance;
    double interestRate;
    int    termInMonths;
} account1, account2;
```

Structures and Classes

- Or we can declare them later:

```
struct CDAccount
{
    long   certNumber;
    double balance;
    double interestRate;
    int    termInMonths;
};
...
CDAccount account1, account2;
```

Structures and Classes

- We refer to structure members with a dot (.) between the structure name and the member's name:

```
struct CDAccount
{
   long   certNumber;
   double balance;
   double interestRate;
   int    termInMonths;
} account1, account2;

account1.certNumber = 8675309;
account1.balance = 10000.00;
account1.interestRate = 0.015;
account1.termInMonths = 30;
```

Structures and Classes

- We can use pointers and the `new` operator to create structures on the fly (you should be thinking linked list nodes here):

```
struct CDAccount
{
   long   certNumber;
   double balance;
   double interestRate;
   int    termInMonths;
};

CDAccount *acct1, *acct2;

acct1 = new CDAccount;
// now we have a new account record
```

Structures and Classes

- When we use a pointer to point to a struct, we don't use the dot syntax; we use an explicit pointer operator:

```
struct CDAccount
{
    long   certNumber;
    double balance;
    double interestRate;
    int    termInMonths;
};
CDAccount *acct1, *acct2;
acct1 = new CDAccount;

acct1->certNumber = 8675309;
acct1->balance = 10000.00;
```

Syntactically, `acct1->balance` is equivalent to `(*acct1).balance`, but the former is *often* used. You may *see* the latter as well.

Structures and Classes

- The previous slide said:
  - Syntactically, `acct1->balance` is equivalent to `(*acct1).balance`, …
  - Be clear in that `(*acct1).balance` is not the same as `*acct1.balance`.
  - The former uses the variable acct1 as a pointer to locate a structure that has a member called balance, and then gets the value of the balance member
  - The latter is equivalent to `*(acct1.balance)`, the value that the pointer `balance` in the declared structure `acct1` (`acct1` isn't a pointer in this case) refers to. It's unlikely that `balance` is a pointer, so the compiler would probably detect this issue for you

Structures and Classes

- When we use a pointer to point to a struct, we don't use the dot syntax; we use an explicit pointer operator:

```
struct CDAccount
{
    long   certNumber;
    double balance;
    double interestRate;
    int    termInMonths;
} acct1;
CDAccount *acct2;
acct2 = new CDAccount;

acct1.certNumber = 8675309;
acct2->certNumber = acct1.certNumber;
```

Structures and Classes

- What, exactly, do we call "->"?

"member selection" operator

- I tend to read it in code as "points to"; that's shorthand for:
  - (pointer variable left of "->") "points to" a structure that has a member called (identifier right of "->")
  - `acct2->certNumber` ≡ "acct2 points to certnumber"

Structures and Classes

- Structure definitions are typically global (outside of all functions), so that all functions can use the structure
- Two different `struct`s can have members with the same name (a CD `struct` can have a member called `balance`, and a savings account `struct` can also have a `balance` member
  - Obviously, the two `balances` would be different entities
  - There's no ambiguity, because the `struct` name would differentiate:
    - `CD.balance` vs `SavAcct.balance`

Structures and Classes

- Structures may be nested:

```
struct XYLocation
{
    double x;
    double y;
};

struct triangle
{
    XYLocation point1;
    XYLocation point2;
    XYLocation point3;
};
```

Structures and Classes

- Structures create a data type, so functions can return a `struct`.
- Assigning one `struct` to another copies the *contents* from one to the other
  - A `struct`'s name *isn't* a reference variable
    - Unless we use a pointer to a `struct`, which *is* a reference variable (explicitly)

Structures and Classes

- Classes:
- The whole “classes are templates (or patterns or blueprints) from which we instantiate objects” thing is exactly as it is in Java
- How they’re structured (in the source file) really is a colossal step backwards:
  - The code for a class’s methods doesn’t have to be contained in the class definition!
  - Classes don’t have to be in their own files

Structures and Classes

- In the class definition, individual members (data / functions) aren't declared as `public` or `private` on a line-by-line basis; they're grouped:

```
class DayOfYear
{
   public:
      void output();
      int month;
      int day;
}
```

Structures and Classes

- If you have a mix of `public` and `private` members (and you usually will), use a `public:` section, followed by a `private:` section
- You can have as many `public:` and `private:` sections as you want, but it gets confusing to go back and forth; so, it's best to group all the `public` items together, and all the `private` ones together

Structures and Classes

- The :: operator is called the *scope resolution operator*, and is similar to the dot operator
  - The . operator works between object & member
  - The :: operator works between class name & member
    - The :: operator has **the** highest precedence in C++

Structures and Classes

- We can put the member functions for a class within the class defintion.
  - If we put the class's member function definitions inside the class definition, we don't need the :: operator (or the class name before it)

Structures and Classes

- Doing so creates an inline function – the code for the function is inserted whenever the function is called.
  - It also violates the separation of the code (the implementation) from the interface (the class definition)

Constructors

- Constructors in C++ work like they do in Java
- The constructor for a class is a function with the same name as the class
  - If the constructor function is defined outside the class definition, then its header will start `classname::classname`
    - They basically work just like in Java (default constructors, etc.)

Enough C++ For Now

- Enough of C++!
- The good news is that C++ and Java *are* very similar
- Java borrowed heavily from the C++ syntax, and it shows
- Java also cleaned up a lot of the object-oriented implementation messiness

Enough C++ For Now

- We've covered enough of the differences (and similarities) between C++ and Java to get you started.
- You'll discover more of C++'s idiosyncrasies as the semester evolves