COSC 2P95

Records, Organization, and Allocation

Week 5

Brock University

Organization of data

Though we've already seen how to declare variables, arrays, etc., there's still one major piece missing: records.

- Whether it's for the sake of cleaner algorithms, or just to help us organize our data, the ability to arrange multiple disparate elements into a heterogeneous record is pretty much mandatory
- To that end, we also wish to be able to explicitly *define new types*

But first, let's look at something simple.

Enumeration

Suppose you want to represent a state, chosen from a finite number of clearly-defined possible states.

Further suppose that you have an innate label for each of these states.

- e.g. Days of the week, common colours, etc.
- Of course, one could argue that booleans fall under this category, but we already have those covered

How could we represent such a concept?

Enumeration

A bad idea

We *could* go ahead and just use `ints` (or `shorts`, etc.), but that doesn't help us remember the bounds, and doesn't afford us any mnemonics.

- Instead, what we'd like to do is to define an explicit *enumeration* type
- That is, we'd like to be able to define a set of labels, and define a type that could hold such a label

Unscoped enumerations

First, let's look at the C-style enumeration: the `enum`. In C++, we'll also call it an *unscoped enumeration*.

- We use the `enum` keyword, followed by a label for a type to represent this category of enumeration, and then a {} block, inside of which we have a comma-separated list of all the possible selections for the enumeration
- It will create integral constants to represent the possible states (for this reason, it's good practice to choose all-uppercase names for the options)

I swear this really is easy. Let's just take a look at an example.

Scoped enumerations

So that was neat, until it turned terrible. One issue is with *scope* (or the lack thereof). As a convention, it's advisable to define `enums` to have terms that all start with the same prefix, as a de facto scope.

However, we have a better alternative: *scoped enumerations* (i.e. enumeration classes)

- We only have to tweak how we declare and use them slightly
- Note that `enum classes` were only introduced in C++11

This is worth checking out:
`http://en.cppreference.com/w/cpp/language/enum`

Example time!

Variant records and overlapping

Consider two possible scenarios:

- You know that a single field will store a value, but that field could be one of two mutually exclusive labels
  - e.g. fahrenheit or celsius; radius or width; etc.
- You want to hold one type of data, but might want to treat it as another type
  - e.g. a single 32-bit integer vs 4 bytes for a colour

We can address both of these concepts with the same feature: `unions`.

Probably best to just demonstrate (with an example)!

Record structures

I think it's finally time to discuss *records*: the concept of a single data type that holds multiple *fields* of different types. Unions don't work, because a unit still only holds a single thing at a time.

- A `struct` is a defined structure that holds multiple pieces of data; each accessible in the same fashion as with unions
- Though less flexible (sizewise) than most other data structure, they're still immensely valuable
- Primarily, all you need do is define them as with unions, but use the `struct` keyword

Let's look at an example, but we'll also see an issue that we'll need to come back to in a moment.

Defining new types

Suppose you find yourself needing to type out a long type (this'll be more likely once we get to OO and templates), or simply want to be able to change how a type is named.

- You can use a `typedef` to define a type
  - `typedef int grobble;`
  - You'd define new numeric types (typically ending with `_t`) when you want to use a single keyword, but may need to provide different underlying types depending on the architecture (e.g. providing `int` or `short`, depending on how many bits wide each is)
  - This used to be far more significant in vanilla C, because the typedef for `struct` and `union` is implicit with C++
- C++11 also provide a *type alias*
  - e.g. instead of `typedef double measured_t;`, you'd say: `using measured_t=double;`

Contemplation of unions and structs

As mentioned, the syntax for defining structs and unions is basically the same. However, the resulting types (and usage) is significantly different.

- We do still need to address using structs for arguments and assignments; they're handled a bit differently from the equivalent in some other languages
- Technically, we could address the problem with either pointers or references, though one works more smoothly than the other

It's definitely important to understand how *dereferencing* works, particularly when dealing with struct pointers.

Example time?

Final thought on structs and pointers

Though this is probably easy to guess, you *can* use pointers as members of structs. One particularly useful trick is to include a pointer to the same struct type, so you can create *linked lists* and *trees*.

Note that, if you ever need to do something like this in plain C, you'll have a slightly trickier time with your declarations (and will likely need to use typedefs).

We put the *fun* in *functions*!

Consider the case of knowing that you'd need to use one of four mathematical functions, but won't know which until runtime. Even worse,you'll need to be using it repeatedly (e.g. to fill in an array). Is there a way to avoid needing to use conditionals?

- You can actually declare a *function pointer*, which is like a variable, except for a function
  - You need to include the *signature* of what you expect to be pointing to
- We can also provide functions as arguments!

Also fun: structs can have proper *member functions* and *constructors*, but let's discuss that next week.

Example time for the function pointers, though?

Void pointers

It won't be long before we're learning better, C++-specific, tools, but *void pointers* are still an interesting leftover relic from C.

- Like any other pointer, it simply contains an address
- The benefit comes from not needing to set an explicit (useful) type
    - For the most part, all a void pointer can do is refer to an address; nothing more
    - Before actually using the referenced resource, you then *cast* the pointer back to a more specific pointer type (and then probably dereference said pointer)

Basically, it acts as a de facto *generic pointer type*.

On the subject of casting...

Besides our classic casts, we've also been occasionally seeing a new style of cast:

- `static_cast` is an explicit conversion cast
- `dynamic_cast` may be convenient once we get to OO, and need to downcast references/pointers to classes
- `reinterpret_cast` will sometimes work where the static cast won't; good for questionable type manipulation

`http://en.cppreference.com/w/cpp/language/reinterpret_cast`
`http://en.cppreference.com/w/cpp/language/const_cast`
`http://en.cppreference.com/w/cpp/language/static_cast`
`http://en.cppreference.com/w/cpp/language/dynamic_cast`

The call stack

Just as a quick reminder, do we remember what *the stack* is, and how it works?

Allocating heap memory

By now, we're comfortable making use of space on the stack. Allocation/deallocation are taken care of for us, and there aren't any screwy pointers unless we want there to be.

But what do we do when we need to allocate space for a variable/structure, but don't really have an inkling as to how many, or how large, until runtime?

- We *dynamically allocate* memory on the heap
  - This means we'll also eventually need to free that memory back up (i.e. explicitly deallocate)
- C used `malloc` to allocate, and `free` to deallocate
- C++ uses `new` and `delete`

Either way, this is good for arrays, as well as growing data structures (e.g. from structs).

`http://www.learncpp.com/cpp-tutorial/6-9a-dynamically-allocating-arrays/`

Final heap memory concerns

There are just a couple final considerations to keep in mind:

- When freeing up memory, make sure to get *everything* you allocated
  - Otherwise, you could end up with a *memory leak*
- Also beware of *dangling pointers*

Templates

Remember back when we learned about function *overloading*?

- If not, it was when we demonstrated that you can have two functions with the same name, but different signatures
  - But who wants to have to keep rewriting functions to match every type that could ever come up?

We do have a solution: *templates*.

```
template <typename Type>
Type max(Type a, Type b) {
     return b>a?b:a;
}
```

Then, we just use it as normal!

Data structure example

- How are we fixed for time? Do we have time to write a data structure example?

Questions?

Comments?

- Catchy tunes?