COSC 2P95

Procedural Abstraction

Week 3

Brock University

Procedural Abstraction

We've already discussed how to arrange complex sets of actions (e.g. loops), but we haven't learned anything that would help us to develop clean and readable code.

- Most algorithms include steps that only need to a cursory understanding at that level of discourse
- Of course, eventually, everything must be completely and unambiguously defined
- We achieve this via *procedural abstraction*: defining single actions that actually represent more complicated sequences of actions

In some languages, we might call them *methods*. We won't be using that term *yet*.

Procedures, and functions, and methods, oh my!

Are we familiar with these terms? Do we know the difference between them?

- A *procedure* is a sequence of imperative instructions that can be collectively aliased (i.e. invoked) via a call to a single term or phrase
- A *function* is comparable to a procedure, but, upon completion, returns a value to the context that invoked it
- A *method* could be a function method, or a procedure method (often simply called *function* or *method*, respectively). What matters is that it's an action that's directly tied to an object/class
  - Once we get to OO, methods will sometimes also be known as *member functions*
  - Some schools of thought will dictate that *procedures* tied to objects should change state, while *functions* should ask questions of the objects

In C (and C++, outside of methods), it's not unheard of to simply refer to *functions*.

How do we start?

A procedure is pretty simple to write. All you need is:

```
[modifier] (return_type) name([parameters]) {
    statement_list
}
```

e.g.

```
void wokka() {
    int dealie=8;
    cout<<"Grobble grobble grobble!"<<endl;
}
```

Why `void`? Because that's how you say you're not returning anything!

Parameters

Parameters allow you to modify the behaviour of an action.

- Parameters declared within the procedure header are also known as *formal parameters*; when you invoke the procedure, the arguments you provide are the *actual parameters*
- The actual parameters must match the formal parameters
  - Of course, casting/coercion can help with this
  - Unlike with some languages, you can't omit any, or provide them out of sequence

Invoking procedures from other procedures

Can we call one procedure from within another?

- Well, obviously we can, but there's something special we need to worry about that may not be a concern for some other languages

We're going to need to try an example.

Whu... what?!?

There are two different kinds of compilers: one-pass, and two-pass.

- A two-pass compiler first goes through the source files, collecting tokens, and building up a table of references. It then does a second pass to ‘connect the dots’
- By comparison, a one-pass compiler only goes through each source file once, building up its tables as it goes. If it hasn’t seen a term by the time it’s needed, it simply doesn’t exist

So... what’s the solution?

Forward declarations

Of course, more often than not, you *could* simply ensure that you declared every procedure before you needed it. But what would happen if you had two procedures, `a` and `b`, with `a` calling `b`, and `b` calling `a`?

Clearly, we simply need to cut the cake before we serve it!

- We don't actually need both *definitions* simultaneously; we simply need to know their *signatures*
- The solution is a *forward declaration*
- All we need to do is to write the procedure's header, with a semicolon. We can provide the body later

Let's revisit the example.

Headers

The only problem is that, for larger source files, starting with a couple dozen prototypes can be a bit inelegant.

More importantly, once we get to using multiple files and linking, we won't want to have to copy all of them over to every connected file.

- We can write a *header file*, which is just a minimal source file that (if used appropriately) doesn't define any behaviours
- Primarily, we use them for function prototypes, but we can also define constants and types (once we get to that)
- We use a preprocessor directive (`#include filename.h`) to include it

Let's wait until the next slide for an example.

Functions

If a procedure has a `void` return type, then a function has... well, pretty much anything else

- We've already defined what a function is. All it takes is to change the return type, and include a `return` statement

Quick example time?

`return` statements

There are two odd points worth knowing about the `return` statement:

- If you neglect to include the `return` statement, even with a function that promised a particular return type, it'll still compile (and run)
  - However, there's no legitimate reason to do this, and the program won't be as readable
- You can include a `return` statement, even for a procedure
  - A common use for this is if you might want to preemptively leave a procedure, but don't want to indent the rest of the procedure for the *else*

(Yes, it's example time again)

Overloading functions

If you want to define the same behaviour more than once, with the only difference being the types, then you can *overload* the function names

- Just declare and define more than one copy of the same name, with only the signatures varying
- Be very careful with these. Coercion can already match with different signatures; ambiguous calls may not compile at all

(Are we tired of examples yet? Because... example time)

Speaking of returns...

You might sometimes need a function that can return more than one value.

- Tough

No multiple returns?

Unfortunately, C++ actually doesn't directly provide a mechanism for *multiple returns*.

That said, we have multiple alternatives available to us (of varying levels of usefulness):

- Once we learn how to create records (e.g. `struct`), you could have a single return that can contain multiple values inside
- C++ provides a tuple in its standard library for effectively the same use
- In theory, you could allocate an array, and provide it as a parameter to the function
  - The function would then modify the contents of the array to match the data to be returned
  - *Generally*, this isn't going to be practical
  - However, it's immensely common as a mechanism for providing a buffer (e.g. for receiving multiple bytes from an input stream)

What if we could simply use a parameter as a return?

Types of parameters

For programming languages in general, there are different potential types of parameters:

- In Parameters — for receiving arguments
- Out Parameters — allows a procedure to return values directly through the parameter
- In/Out Parameters — can be used either way

Ada, for examle, is very good for labelling these explicitly.

But, how does this help us in C++?

Values, pointers, and references

First, let's briefly discuss the difference between *pass-by-value* and *pass-by-reference*.

This is where we're particularly glad we're using C++ (even over C).

- A *pointer* is a special variable that contains an address as its stored value
  - Pointers can be *dereferenced* to access (or modify) the value referenced by the pointer
  - Other than still technically being a number (meaning you can perform arithmetic on it), pointers are mostly analogous to Java references
- C++ also provides actual *references*
  - A reference
- Both pointers and references require accessing the *address* of the corresponding variable

The two different techniques are best understood by demonstrating the difference.

Usage Suggestions

There's a school of thought that functions should only request values, and procedures should change state.

- A *side effect* occurs when a function changes the state of the system
  - Side effects are sometimes discouraged for functions, if it can make it harder to predict state just from reading the source
- An *idempotent* function is one that always produces the same result when it receives the same parameters
  - Idempotent functions *can* have side effects, so long as those side effects don't change the behaviour of successive calls
  - e.g. adding a value to an array at a specific position
  - On the other hand, a *push* into a stack is not

Variadic functions

Sometimes you may not know exactly how many parameters a function will have. Common examples include formatted print statements, or functions that select a single element from an arbitrary number of options (e.g. *max*).

- We're actually not going to get into this yet, because C++ introduced a better approach, but it relies on *templates* (which is a few weeks away)
- However, if you find yourself needing to do it in the meantime, you can always use C's *vararg* feature

Constants

We've already talked about symbolic constants (`#define`) and `const`, but there's still one more term: `constexpr`

- A constexpr defines a *compile-time* constant
- Technically you can define constant values, but there's little reason to choose that over a const
- constexpr can also be used to define simple functions, so long as they can be fully-fined at compile-time
- You may need to explicitly tell the compiler to use the C++11 standard (or higher)
    - e.g. `g++ -std=c++11 -o out file.cpp`

Example time? Example time.

Inline functions

Recall how function invocation works: it requires allocating memory for the call frame, jumping to a different place in the code, etc. It's slower than simply having the equivalent instructions in-place.

- Actually, you can do that. Prepending the `inline` keyword before a function acts as a hint to the compiler that it may explicitly expand the the function, as though its lines had been written directly
- Typically, it'll run slightly faster, and take up slightly more space

Scope revisited

Now that we've looked at functions and pointers, let's revisit scope and extent a bit.

Let's review a sample (which is technically different from an example).

Recursion

Thankfully, there are pretty much no special considerations for writing recursive solutions in C/C++.

However, we do remember what a recursive function is, and how it works, right? Because we'll need that later.

Arrays

We'll get to dynamic allocation and ragged arrays later, but there's no reason not to start using arrays.

- Allocate an array as such: `int name[size];`
- You can pre-initialize them: `int name[3]={val1,val2,val3};`
  - You don't need to specify the size when pre-initializing
- Be careful relying on the `sizeof` function for array sizes
- You can do interesting things by casting one array type into another
- When declaring a variable-length automatic array as a parameter, you can only leave the number of rows blank
- 2D (or more) arrays simply require additional square brackets

Note: Because of how arrays work, you can also reference them with a pointer of the base type.

Up to you if we want to look at this together.

Strings

There are two different major string options available:

- C used a char array (or char pointer)
- C++ defines a class

Both have their own string libraries.

C++ strings are pretty good, but, on occasion, you'll be forced to use the older C-style.

Command-line parameters

- Recall from the other week that the `main` function accepts two parameters: the number of command-line parameters, and a (C-style) string array containing the parameters
- The first (index 0) entry in the array will be the program name, including path
- Since they're strings, you'll need to do convert them if you want to accept numbers
  - You can use the `atoi` (or `atof`, or `atol`) function provided by the `cstdlib` library

Questions?

- Have we had enough examples? Because we can totally come up with more.