CSCI251/CSCI851 Autumn-2020
Advanced Programming (**SGPb**)

C++ good practice:

Part B

On we go …

- The last set finished on page 130 so we will pick up from there…

Page 139: Compounding expressions

- Expressions with two or more operators …

```
6+3*4/2+2
```

- Two rules of thumb:
  - If you are unsure of how your expression will be evaluated based on precedence, use brackets to make sure the logic is what you want.

```
((6+((3*4)/2))+2)
```

  - If you change the value of an operand, don't use the operand elsewhere in the same expression.
    - Exception: When the subexpression changing the operand is itself the operand of another subexpression… as in `*++iter`.

- A point related to the first rule of thumb is given on page 146.
- Assignment has lower precedence than the relational operators.
- So parentheses (brackets) are usually needed around assignments in conditions.

Page 140: Overflow

- Values evaluated are sometimes outside of the range that can be stored.
- Example, if a `short` stores 16 bits.

```
short short_value = 32767;
short_value +=1;
cout << "short_value: " << short_value << endl;
```

The value wraps around so we get, on Banshee for example,

```
short_value: -32768
```

Page 144: `true` or `false`

- `true` and `false` are Boolean literals, the only values able to be held by `bool` variables.

```
cout << typeid(false).name() << endl;
```

```
bool
```

- Using them in comparisons

```
if (val == true)
```

is inappropriate if `val` is not a `bool`, and unnecessary if `val` is a `bool`. ☺

Pages 148-149: Prefix/postfix

- Prefix (++x) involves more work than postfix (x++), at least on the surface and probably usually.
- The latter has to store the unincremented value to return, the prefix doesn't since it doesn't use the unincremented value.
- For `ints` and pointers compilers can optimise away the cost, but not necessarily for general iterators.

- However, postfix does have it's uses.
- The statements …

```
cout << *iter << endl;
++iter;
```

- … can be replaced with …

```
cout << *iter++ << endl;
```

- This, once you are comfortable with the notation, is clearer and less error prone.

Page 151: Beware the nested conditionals

- The conditional operator …

```
cond ? expr1 : expr2;
```

- … can be used to concisely represent if-else logic.
- But, they can also make code unreadable.
- The textbook suggest “It’s a good idea to nest no more than two or three”.
  - I’d be wary about nesting them at all.

Pages 153-154: Bitwise operators

- We have come across left shift and right shift operations, << and >>.
  - There are also ~ (bitwise NOT), & (bitwise AND), ^ (bitwise XOR), and | (bitwise OR).
- The handling of the sign bit used in signed representations isn't guaranteed, so only use these operators with unsigned types.
- Possible to get bitwise and logical mixed up, so & vs && and | vs ||.

Pages 162-165: Casting

- Directly from the book: “Although necessary at times, casts are inherently dangerous constructs.”
- Casts are not always defined, and even if it works you may well lose data, or just generally treat something as a type it isn’t, and maybe wasn’t ever supposed to be.
- The book is particularly wary about the machine dependent `reinterpret_cast`.

Pages 172-173: Statement syntax

- It's not unusual to have stubs, empty functions or null statements, where you intend to add functionality later.
- BP: These should be commented, so anyone looking at your code can see the gap is deliberate.
- Additional null statements can be dangerous.

```
while (iter != svec.end()) ;
   ++iter
```

- The ; means the iterator increment isn't in the while loop. ☹
- Blocks aren't terminated by semicolons.

BP: Page 177: Auto Indentation

- A lot of editors and development environments help by using auto indentation to match the code meaning.
- The book says: Use them if they are available.
- Qualifier: Remember my preference that you use a text editor until you know a bit more about what is going on.
  - I want to think about the layers.

BP: Pages 178-182: Switch

- You may want several cases to map to the same result, you can use something like …

```
switch (ch)
{
      case ‘a’: case ‘e’: case ‘i’: case ‘o’: case ‘u’:
            ++vowelCount;
            break;
}
```

- Such omission of a `break` at the end of a `case` is fairly unusual though, so it’s a good idea to comment this.
- It’s good practice to include a `break` after the last label, in case an additional `case` is added.
- Include a `default` case even if it should never be reached, remember defensive programming.

Pages 183-186: Variable lifetimes

- Variables defined in a `while` condition, or in the body of a `while` loop, are created and described on each iteration.
  - This is likely to be expensive.
- Objects defined within a `for` loop header are only visible to the body of the `for` loop.
  - If we expect such objects to be of value afterwards, so they aren't simply counters, we should define them prior to the loop header.

BP Page 192: Don't use `goto` !

- It's possible to jump between sections of code using

```
goto label;
```

- For example …

```
goto end;
```

… jumps to …

```
end: return;
```

- Don't use `goto`.

Page 196: Exception safe code…

- Exceptions interrupt the normal flow of a program, meaning the state of objects may be invalid, incomplete, or resources may not be freed up.
- Code that correctly cleans things up is said to be exception safe, and the book has a spiel explaining that writing exception safe code is difficult.
  - See also Resource Aquisition is Initialisation (RAII).
- https://www.tomdalling.com/blog/software-design/resource-acquisition-is-initialisation-raii-explained/

Page 207: Functions in headers

- The header that declares a function should be included in the source file that defines that function.
- So in `function.cpp` we have.

```
#include "function.h"
```

Pages 210-211: Reference parameters

- C Programming:
  - Pointer parameters are often used to access objects outside a function.

```
void reset(int *ip){ … }    reset(&i)
```

- C++ Programming BP:
  - Use reference parameters instead.

```
void reset (int &i) { … }   reset(i)
```

- Reference parameters that aren't going to change inside a function should be made references to `const`.

```
bool isShorter (const string &s1, const string &s2)
```

Pages 215-217: Array size for functions

- Make sure that all uses of an array stay within the bounds of the array.
- This is critical for functions where the size may not be as readily available, and the book describes some methods for dealing with size.
  - An end marker, like in C-strings. Not so useful for ints ☹
  - Using standard library being, end type pointers.
  - Explicitly passing a size parameter.

Page 217: Array reference parameters

- In using array reference parameters for functions be careful with the syntax!

```
f(int &arr[10])    // Array of references.
f(int (&arr)[10])  // Reference to an array of 10 ints.
```

Page 219: Extras start at `argv[1]`

- The program name is stored in `argv[0]`, so the arguments to the program begin at `argv[1]`.

Pages 222-225: Return values …

- For functions with return type void, using

```
return expression;
```

- … is possible only if it's to return the result of a function that returns void.
- If functions are supposed to return a type, every return within that function must do so.
- If a function has a loop containing a return, there should still be a return statement afterwards, but many compilers won't detect this problem.

- Don’t return references or pointers to local variables, they are out of scope anywhere else.
  - When function end their storage is freed.

```
const string &manip()
{
     string ret;
     // Do something to ret.
if (!ret.empty())
     return ret;
else
     return "Empty";
}
```

Both of these return statements return undefined values.

We are okay with:

```
const string manip()
```

Page 226: C++11
List initialising return values

```
vector<string> process()
{
  if expected.empty())
        return {};
  else if (expected == actual)
    return {"functionX", "okay")
  else
    return {“functionX”, expected, actual};
}
```

Pages 229-230: C++11 Trailing return types

- It's possible, in C++11, to list the return type of a function after the parameter list.
- For example;

```
auto func(int i) -> int(*)[10];
```

- The textbook suggests this is most useful in cases, as above, where it's a fairly complex type, here a pointer to an array of ten `int`s.

- The textbook gives a similar example making use of `decltype`, which is somewhat similar to `auto`.

```
int odd[]={1,3,5,7,9};
int even[]={0,2,4,6,8};
decltype(odd) *arrPtr(int i)
{
     return (i % 2) ? &odd : &even;
}
```

- The function `arrPtr` returns a pointer to an array of five `int`s, as taken from `odd`.
- The * is needed since `decltype(odd)` gives an array, not a pointer to it.

```
int (*x)[5];
x=arrPtr(3);
cout << **x << *(*x+1) << *(*x+2) << endl;
```

A couple of notes on `main()`

- You cannot call `main()` from `main()`, or from any function potentially called from `main()`, so you cannot usefully call `main()`.

```
"temp2.cpp", line 14: Error: Cannot
have a recursive call of main().
```

(This is the error on Banshee)

- You cannot overload `main()`.

Page 233: When to overload…

- A more descriptive name is often better than using something generic to capture fairly different functionality.
  - Think from the point of view of a person using the functions you have written.
- The example given in the textbook relates to moving a cursor and while you could use default arguments on a general `move()` function, it's likely a `moveHome()`would be better.

```
myScreen.moveHome();
myScreen.move();
```

Page 234: Hiding & Local functions

- Overloading is when we use the same name with different parameters in the same scope.
- If we use the same name in an inner scope, the outer functions are hidden, even when they have different parameters.
- C++ checks for the name prior before to checking the type and if it finds a local one it won't look in outer scope.
- Generally local functions are a bad idea.

A bit more on hiding

- Here goes a set function within the class `Person`.

```
void Person::set(int age)
{
    this->age = age;
}
```

- This will produce a compiler warning about hiding, effectively the local age hides the class level age so you need the this pointer to deal with it.
- If you want to get rid of the warning, change the variable name `age`.
  - There are likely conventions on this.

Page 237: Default arguments

- Default arguments should normally be specified with the function declaration in an appropriate header.
  - Okay, why bring this up?
- Well, it's possible to add, but not change, defaults after the initial declaration …

```
string screen(size, size, char= ' ');     // Initial.
string screen(size, size, char='*');      // Not okay.
string screen(size=24, size=80, char);    // Okay.
```

- … but it's better not too!
- There is a note on page 243 that with default arguments calls to functions may appear to have fewer arguments that it actually does.

Page 239: Inline request

- To be inline means the code in the function is expanded into the code at the pointer of calling, rather than being set aside as a distinct function.
- Adding the keyword `inline` prior to the return type of a function makes a request to the compiler to make the function inline.

```
inline const string &
shorterString(const string &s1, const string &s2)
{
      return s1.size() <= s2.size() ? s1 : s2;
}
```

- But it is only a request, and may be ignored.
- On page 257 it is noted that functions defined in the class are implicitly inline.

Pages 239-240: `constexpr` return

- A `constexpr` function can be used in a constant function, but it isn't actually required to return a constant expression.
- So, we need to be careful how we define and use `constexpr` functions.

Page 240: `inline` & `constexpr` functions

- They can be defined multiple times, but each definition of the same function needs to match exactly.
- So, `inline` and `constexpr` functions are normally defined in headers.

Pages 245-246 : Casts and conversions

- Casts shouldn't be needed to call overloaded functions.
  - The textbook suggests such a need means the parameter sets are poorly defined.
- There is a warning that promotions and conversions among built-in types can give surprising and undesirable results in the context of function matching.

Pages 247-248: Pointers to functions

- This is a note on syntax.
- Consider that we have the following function:

```
bool lengthCompare(const string &, const string &);
```

- The function type is `bool(const string&, const string&)`.
- To allow `pf` to point at this function we need …

```
bool (*pf)(const string &, const string &);
```

- Without the brackets …

```
bool *pf(const string &, const string &);
```

- … would make `pf` a function returning a pointer to a `bool`.