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

C++ good practice:

Part A

Global variables …

- Don’t use global variables unless it’s something that does need to be known everywhere, and then it should almost certainly be const.
- If you have global variables, you need to remember they are there so don’t re-use the names.
  - It’s pretty much like having all of the global variables passed as an argument to every function you write everywhere.

Good practice?

- The textbook provides a fair few warnings and good practices associated with programming in C++.
- We are going to look at some of these.
- Some verbatim, others paraphrased.

Page 5: Compiler options

- Use the available compiler options to identify potential problems.

```
CC +w2 …
g++ -Wall …
```

Page 7: Flush debugging

- Using some sort of print statements while debugging can be quite helpful.
  - To work out where your program reaches for example.
- If you do this, make sure you flush the stream, that is output with `endl`.
- If you don't and the program crashes there may be content in the buffer and that may mislead people as to where the program crashes.

Page 19: Consistency in style

```
int main(){

}
```

```
int main()
{

}
```

- It probably doesn't matter which style you use, but try to be consistent.
- The lecture notes are not ☹.

Page 34: Which Arithmetic Type?

- Use an unsigned type if you know values cannot be negative.
- Use `int` for integer arithmetic, `long long` if they larger, because long is too often the same as `int`.
- Don't use char or bool in arithmetic expressions, just for holding characters or truth values.
- Double for floating-point, float usually lacks enough precision.
  - The precision of long double is usually unnecessary and there is often significant run-time cost.

Page 36: Undefined and implementation-defined behaviour

- Avoid undefined expressions.
  - They might run but may well be platform/compiler/run dependent.
- Implementation-defined relates to things like the size of an `int`, or other variables, that a program assumes rather than dealing with dynamically.

Page 37: Avoid mixing signed and unsigned types …

- This can have surprising results.
- The textbook gives an example of `a*b` where a is -1 and b is 1.
  - If both are `int`s, we get -1.
  - If a is an `int` and b is an `unsigned`, we have a machine dependent expression.

```
int a=-1;
unsigned int b=1;
cout << a*b << endl;
$ ./a.out
4294967295
```

BP Page 40: Literal types

- Using a long literal?
  - Use the uppercase L to avoid confusion between the lowercase letter l and the digit 1.
  - For example: 11ULL.

Pages 43-45: Some initialisation notes

- Initialisation is not assignment:
  - Initialisation happens when a variable is given a value when it is created.
  - Assignment obliterates an object's current value and replaces that value with a new one.
- Uninitialised objects of built-in type defined inside a function body have undefined value.
  - Objects of class type that we do not explicitly initialise have a value that is defined by the class.
- Unitialised variables cause run-time problems.
  - You might crash, that's probably good because running with an unreliable result is likely to make things worse in the long run.

Page 45:
Define once, declare many times.

- Variables must be defined exactly once.
- They can be declared many times.

BP Page 47: Naming …

- Naming conventions are most useful when they are followed consistently.

Pages 48-49: Defining variables

- It's usually a good idea to define objects/variables near where you first use them.
  - This improves readability.
  - It is often easier to give it a useful initial value.
- Almost always a bad idea to define a local variable with the same name as a global variable the function uses or might use.

Pages 52, 54: Problems with pointers.

- Debugging problems due to pointer errors trouble even experienced programmers.
- Advise: Initialise all pointers.
  - As with general variables you may get a crash, or worse due to wrong interpretations.
  - The suggestion in the book is to initialise the pointer after setting up the object it will point to, or initialise the pointer to `nullptr` or zero.

Pages 57-63: Compounding

- No single right way to define pointers or references, but be consistent.

```
int* p1;
int *p1;
```

- It can be easier to understand complicated pointer or reference declarations if you read them from right to left.
- There are no `const` references.
  - References aren't objects and cannot be constant themselves in the typical sense.
    - References only ever refer to one object so in some sense are all constant.

- A pointer to `const` isn't about whether the object pointed to is `const`, but about whether or not you are allowed to change the object through that pointer.
  - The object itself might change.

Page 66: `constexpr` variables

- Generally, it is a good idea to use `constexpr` for variables that you intend to use as constant expressions.
  - The constexpr specifier declares it's possible to evaluate the value of the function or variable at compile time.

Pages 76-77: Headers

- If a header is updated, the source files using the header much be recompiled.
- Preprocessor variables don’t follow C++ scoping rules and in particular must be unique throughout the program.
- Headers should have guards, whether or not they are included by other headers.

Page 84: Library efficiency

- In addition to specifying the operations that the library types provide, the standard also imposes efficiency requirements on implementors.
- As a result, library types are efficient enough for general use.

Page 87: `getline` newline dropping

- The newline causes `getline` to return is discarded and not stored in the string.

Page 91: C headers vs C++ headers

- Headers in C have names of the form `name.h`.
- The corresponding file headers in C++ are instead named `cname`.
  - The c is used to indicate they are part of the C library but without the .h means they are part of the standard library.

Page 95: Subscripts are unchecked

- Using subscripts?
  - Make sure they are in range.
  - For string for example:
    - Subscript `>= 0`, and the `< size()` of the `string`.
    - The suggestion is to use a variable of type `string::size_type` since that is unsigned so at least deals with the first of those two conditions.
  - Possibility of buffer overflows.

`vector`

- Page 97: Not a type in itself, it's a template.
- Page 101: Grow efficiently.
  - Usually best to define an empty vector and add values as they become known at runtime, rather than to define a `vector` at it's expected size.
    - Unless all the elements need the same value.

range `for`

- Page 101: The body of a range `for` must not change the size of the sequence over which it is iterating.
  - This is true for iterators generally.
- Page 105: Range `for` is a good way to ensure subscripts are in range to avoid subscripting problems.

Iterators

- Page 106: The container member function `end( )` returns the off-the-end iterator, it's not referring to the last element but one position past the last.
- Page 107: The off-the-end iterator cannot be incremented or decremented.
  - It's the same behaviour as for a pointer one past the end of a built in array.

- Page 109: Iterator may refer to:
  - The concept: Uniform container interation.
  - The type: Defined for each container class.
  - An object as an iterator: An instance of the type.
- Page 110: Loops using iterators shouldn’t add elements to the accessed container.

Arrays

- Page 113: If you don't know how many elements you need, use a `vector`.
- If you do know how many elements you need, you may get better run-time performance from an array.

Pages 122-123: C-Style character strings

- For most applications, in addition to being safer, it is also more efficient to use the library `strings` rather than C-style strings.
- Those pages describe how the operations differ.

Page 125: More on C-strings, pointers/arrays

- The function c_str() is used to extract a c-string from a String.

- Generally, avoid pointers and arrays!
  - Quite error prone → low-level manipulations and tricky syntax.
- Modern C++ should use `vectors` and iterators.

Pages 125-130: Multidimensional arrays

- No such thing in C++ ☺
  - What we see as a multiple dimensional array is actually an array of arrays.
- Using range `for` across a multidimensional array, the loop control variable for all but the innermost array must be references.
  - This makes sure the variable types are correct.
- Warning on arrays and pointers …

```
int *ip[4];      // array of pointers to int.
int (*ip)[4];    // pointer to an array of 4 ints
```

```
int ia[3][4];
size_t count = 0;   // More general …
for (auto &row : ia)
   for (auto &col : row)
      {
           col = count;
           ++count;
      }

for (const auto & row : ia)
    for (auto col : row)
       cout << col << endl;
```