Coding Practices

PIC 10B, UCLA

Coding Practices

Coding is not just about “getting the job done”. This sort of mentality can lead to code that is difficult to read or debug, inefficient, and prone to error.

A language like C++ can be compiled on many different computer architectures, but not all code is valid across platforms. For this reason, it is important to develop good coding practices so the code can be compiled across platforms, the resulting code is easy to debug and works reliably; and to program with efficiency in mind: code might work fine on a Windows environment with 1000 **std::string variables**, but how would that code work with a company like Google with millions or billions of **std::string variables**, perhaps also with Unix-based compilers?

Coding Practices

There is also one other nice “feature” of C++: the language is always evolving. New features and syntax are always being added; while code from ancient C++ versions will work on most modern compilers, many of the added functionalities are worth studying and learning because they can shorten or simplify otherwise complex procedures, while maintaining readability.

The C++ Standard

The way compilers work is governed by the *International Standard ISO - Programming Language C++*. Compilers and IDEs like Visual Studio, g++, etc., must adhere to the rules in this *very, very, very detailed and* technical document.

In this course some excerpts will come from the C++ Standard, generally quoted in italics and distinguished by the highly technical legalese.

The C++ Standard

Think of **"the Standard"** like "the law": it governs what compilers are required to do and how code should behave.

Without knowing the rules of the language, programmers may make bad assumptions and adopt poor coding practices that make code unsafe to use.

Remark: just as not all people are law-abiding, some compilers do not fully adhere to the Standard. But most do a pretty good job once a feature has been around for a while.

Notes on Behaviour

Some items to be aware of in terms of how code behaves:

- **Specified by the Standard:** rules the Standard explicitly states.

  "The sizeof operator yields the number of bytes in the object representation of its operand [...]. sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 1."

  So on every compiler, a **char** must take up 1 byte.

  "The type of a floating literal is double unless explicitly specified by a suffix."

  So **3.14** is seen as a **double** on all compilers, never **float** or **long double**.

Notes on Behaviour

- **Implementation-Defined**: cases for which each compiler can give its own implementation but must document it. For example

  "There are three floating-point types float, double, and long double. [...] The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double. The value representation of floating-point types is implementation-defined."

  The number of bytes a **double** occupies can vary across compilers.

  "The type size_t is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object."

  size_t could translate to different data types on different compilers.

Notes on Behaviour

- **Unspecified:** situations where the Standard makes no absolute rulings for behaviour (but may provide a list of permissible behaviours):

  "Evaluating a string-literal results in a string literal object with static storage duration [...] Whether all string literals are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a string-literal yield the same or a different object is unspecified."

  String literals get a spot in memory but in the simple code below:

  ```
  std::cout << "hi"<< "hi";
  ```

  it is unspecified as to whether the program will in fact store two separate string literals or just one big block with "hihi".

Notes on Behaviour

- **Undefined behaviour:** the most deadly. Behaviour can vary radically across compilers and no guidelines or specifications are provided. A program could fail to compile, generate a warning, crash mid-program, carry on and work normally, corrupt the computer's memory, etc.

  "Except that any class member declared mutable can be modified, any attempt to modify a const object during its lifetime results in undefined behavior."

  Objects marked const may be changed, but it is dangerous!

  "If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined."

```
int i = std::numeric_limits<int>::max(); // i is INT_MAX
i = i+1; // signed integer overflow, value of i could be anything
```

Review: Code Readability and Robustness

- Comments should be detailed and appear before each block of related code. Single line comments begin with // and multi-line comments begin with /* and end with */.
- Variables should be given clear names.
- Control flow structures should each be commented.
- Variable should be made **const** when they are constant for robustness.
- Control flow structures should always have braces **{ }** for readability and robustness.

Review: Code Readability and Robustness

```
const size_t out_layer_index = layers.size()-1; // index of the output layer

size_t sample_index = 0; // the sample we are using

for( auto i : indices_used ){ // for each index used

  // start the derivatives of cost with respect to activations dCda values to all be 0
  dCdas = std::deque< matrix<T> > ( indices_used.size(),
     matrix<T>( training_data[0].output.size(1), training_data[0].output.size(2) ) );

  // the loop here computes the dCdas for each layer from the final to the first, using back propagation

  for( size_t j = out_layer_index; j != static_cast<size_t>(-1); --j ){ // go from end to start

    if( j ==  out_layer_index ){ // if at the end, special case

        auto y = training_data[i].output; // the correct value
        auto a = avalues[sample_index][layers.size()]; // predicted

        // derivative of cost with respect to activation normalized by scale and batch size
        dCdas[j] = manager->policies.dCda(y,a).tr() / ( manager->policies.get_norm() * indices_used.size() );
    }
    else{ // then somewhere in the middle of the chain

      // compute the layer derivative and multiply it by the rest on the left
      dCdas[j] = dCdas[j+1] * manager->policies.dada_hidden( zvalues[sample_index][j+1],
         layers[j+1].get_weights() );

    }
  }
```

Review: Code Readability and Robustness

- Function names should be descriptive.
- Functions should be documented according to what they do, their inputs (if any), and outputs (if any), with the documentation beginning with /** and ending with */ - this is part of the Doxygen Style for comments and is an industry standard

Review: Code Readability and Robustness

```
/**
returns the minimum of two entries, treating both as size types
@tparam T the first data type
@tparam S the second data type
@param _first the first value
@param _second the second value
@return the minimum of the two sizes
*/
template<typename T, typename S>
size_t min_size(const T& _first, const S& _second){
  size_t first = static_cast<size_t>(_first), second = static_cast<size_t>(_second);
  return min(first,second); // call the min function accepting size_t's
}
```

Review: size_t

The variable type **size_t** is an unsigned integer type, used *especially for indexing containers and their sizes.*

When we talk about containers, we mean C-style arrays or classes that store elements like: **std::vector**s, **std::string**s, **std::set**s.

Different compilers and computer architectures may allow for different maximum sizes of their containers.

The **size_t** variable type chooses an unsigned integer type that is guaranteed to be able to access the largest possible index of a container, based on the architecture.

It is *not* always same thing as an **unsigned int** or an **unsigned long**, etc. Using **size_t** also has a semantic meaning of referring to a size.

Review: size_t

The return type of the **size** member function of **std::vector** is type **size_t**, not **int**!

```
std::vector<int> v;
// a bunch of push_backs...

size_t vsize = v.size(); // safe
for( size_t i = 0; i < vsize; ++i) { /* stuff */ } // safe

int vsize2 = v.size(); // unsafe: int might overflow
unsigned int vsize3 = v.size(); // unsafe: unsigned int might overflow

for(int i=0; i < v.size(); ++i) {/* stuff */} // unsafe: int might overflow
```

Review: class and struct

A **class** and **struct** are both classes; when declared by **class**, the default access level is **private** and when declared as **struct**, the default access level is **public**.

The default access level remains until a **public** or **private** (or other) access modifier is used.

Review: class and struct

In access restrictions of public vs private, the two classes are the same:

```
class X {
    int a() const; // private by default
public:
    int b() const; // public due to specifier
    int c() const; // also public
private:
    int d() const; // private due to specifier
};

struct X {
    int b() const; // public by default
private:
    int a() const; // private due to specifier
    int d() const; // also private
public:
    int c() const; // public due to specifier
};
```

Review: class and struct

Semantically, there can be a difference.

Usually we use **struct**s as "simple" classes: they only store data and lack member functions or else they only have member functions but no data.

Then we use **class**es for more "sophisticated" classes, such as an **Employee** class or a **Car** class where it is important to encapsulate the data and provide an interface through member functions, etc.

Review: Const-Correctness

If a class member function does not modify the class object member variables, it should be declared as **const**. Such functions are called accessor functions.

Mutator functions may change the class member variables and are not declared as const.

```
class A {
public:
    A(); // has a default constructor
    int get_a() const { return a; }
    void double_a() { a *= 2; }
private:
    int a;
};
```

Review: Const-Correctness

If a function does not modify a reference argument, the argument should be declared **const** - can also apply to arguments passed by value.

```
void foo( const A& someA) { std::cout << someA.get_a(); }
```

Adhering to const-correctness allows for easier debugging because if a variable value is totally wrong, the “list of suspects” is drastically reduced since some const functions are innocent.

Failure to systematically follow const-correctness leads to compiler errors. No, the way to fix them correctly is not to just remove a bunch of const-keywords :-)

Review: Header vs CPP Files

Header files are often used to declare functions, variables, and provide class interfaces.

CPP files are often used to define those functions, classes, and variables, and make use of them.

Review: Header vs CPP Files

To guard against multiple definitions of a class, all header files should begin with:

```
#ifndef _SOME_NAME_
#define _SOME_NAME_
```

and end with

```
#endif
```

Multiple declarations are allowed, but it is often an error, usually a linker error, to define something more than once.

Using **#pragma once** is a dangerous substitute for the #ifndef stuff: it has undefined behavior across compilers. It should be avoided.

Review: Header vs CPP Files

Cat.h (tells us everything about the class and functions)

```
#ifndef _CAT_
#define _CAT_
struct cat {
    /**
    makes the cat meow
    /*
    void meow() const;
};

/**
makes a cat talk
@param _cat the cat that will talk
*/
void talk(const cat& _cat);

#endif
```

Review: Header vs CPP Files

Cat.cpp (defines the functions)

```
#include<iostream>
#include "Cat.h"

void cat::meow() const { std::cout << "meow"; }

void talk(const cat& _cat) { _cat.meow(); }
```

Review: Header vs CPP Files

main.cpp (where we use the cat class)

```
#include "Cat.h"

int main() {
    cat c;
    talk(c);
    return 0;
}
```

Review: Usings and Namespaces I

One should not write naked statements like

```
using namepace std;
using std::cout;
// etc.
```

inside of a header file because all CPP or H-files that include such a header file will be forced to conform to the standard namespace.

This can lead to naming conflicts.

Review: Usings and Namespaces II

The user-defined **string** class conflicts with **std::string** below because the standard namespace has been imposed.

Header.h

```
#ifndef _HEADER_
#define _HEADER_
#include<string>
using namespace std; // evil being done here
#endif
```

main.cpp

```
#include "Header.h"
class string { }; // a user's string class...
int main() {
    string s; // ERROR: 'string' is ambiguous
    return 0;
}
```

Review: Passing by Reference vs by Value

When a parameter is passed by value, as below, a copy is made for use inside the function.

```
void foo(int, double); // the int and double are both copies

/* the string and vector are both copies and the string copy cannot be
modified in the function */

void boo(const std::string, std::vector<char>);
```

This can be costly if class objects are being copied. It is also true that changing a copy does not change the original.

Review: Passing by Reference vs by Value

Below, both parameters are passed by reference. The **int** can be modified and this will also modify the **int** passed to the function; the **std::string** cannot be modified.

```
void moo(const std::string&, int&);
```

Review: Passing by Reference vs by Value

Class objects should be passed by reference (most of the time): if they are to be modified, they should be a regular reference; if they are not going to change, they should be a reference to const.

Generally, fundamental types should be passed by reference if they are to be modified and otherwise by value: the size of a reference (4 or 8 bytes, usually) is roughly the same as the fundamental type itself.

Useful Convention: Passing by Reference vs by Value

Remark: a popular style guideline is to separate a function's arguments into two groups: arguments that are passed by value or reference to const, and arguments that are passed by reference. For example:

```
int fun(int value, const std::string& const_ref,
    const std::set<double>& const_ref2,
     int& by_ref, std::string& by_ref2 );
```

With this grouping, it is easier to see that when calling

```
int i = fun(7, string1, someset, j, string2);
```

that the 7 is copied, **string1** will not be modified, **someset** will not be modified, and that **j** and **string2** could be modified...

Speed Tip: Variable Definition Placement

Useful fact: in C++, program execution speeds up if variables are defined near where they are first used!

One should avoid defining a variable and then using it for the first time many lines of code later.

```
// Efficient
int x = 42;
std::cout << x;

// Inefficient
int y = 42;
// many lines of code not using y at all
// ...
std::cout << y; // first use of y
```

Review: std::endl vs '\n'

Besides **'\n'** being a **char** and **std::endl** being a function (or pointer to a function), there is a fundamental difference between the two.

With **std::endl**, a new line character is printed *and the output stream buffer is flushed (ensuring everything is printed to the console)*. This overrides some optimizations a compiler may use to make the code run faster by only flushing at optimal times.

For faster code, unless flushing "right then and there" is really important **'\n'** is preferred.

The **std::cout** buffer is automatically flushed before user commands are read through **std::cin** because the two streams are **tied**.

Review: Never Use system("pause")

system("pause") is evil: it is not defined across platforms and can lead to undefined behaviour. If you really need a pause, add something like std::cin.get();

Avoid Global Variables

Global variables, i.e., variables defined above **int main()** in some fashion, should be avoided because their names can clutter up the namespace and lead to glitches in code.

Avoid Global Variables

```
#include<iostream>
int i = 42; // global variable defined here

int main(){
    int j = 0;
    while(i < 100) { // woops, we meant to write j...
        std::cout << j++;
    }
    return 0;
}
```

We get an infinite loop. If only we didn't make **i** global, we'd have a compiler error instead...

Review: Constructor Initializer Lists

Before the body of a constructor is executed, all of its member variables are initialized **in the order they are declared** *within the class*.

Initialization of member variables is based upon construction parameters given by the programmer in the **constructor initializer list**, or a **default initialization** if the programmer does not specify how the construction is to take place.

The green part is the constructor initializer list:

Constructor(arguments) : / stuff */ { /* body */ }*

References, const members, and class object member variables without default constructors must be constructor initialized; it is also more efficient in general to constructor initialize.

Review: Constructor Initializer Lists

```
struct Bar { // default access to struct is public
    Bar(int theInt, size_t theSize, const std::string& theString) :
        i(theInt),
        v(theSize, '*'),
        isqrd(i*i) {
        s = theSring;
        j=7;
    }

    // its member variables
    int i, j, isqrd;
    std::string s;
    std::vector<char> v;
};
```

Review: Constructor Initializer Lists

Without compiler optimization (happens behind the scenes), the code

```
Bar b(9.8, 100, "PIC 10B");
```

means:

- first, **i** is set to 9 (**double** implicitly made into **int**);
- then, **j** is created but has some random unassigned value;
- then **isqrd** is set to i*i (81);
- then, **s** is default constructed and set to "";
- then **v** is set to a **std::vector<char>** storing 100 *'s.
- After this, **s** is replaced by "PIC 10B"
- and **j** is overwritten to be 7.

Review: Constructor Initializer Lists

A more efficient implementation:

```
struct Bar {
   Bar(int theInt, size_t theSize, const std::string& theString) : i(theInt),
      j(7), isqrd(i*i), s(theString), v(theSize, '*')     { }

   int i, j, isqrd;
   std::string s;
   std::vector<char> v;
};
```

Everything is done in the constructor initializer list. In general, as much as possible should be done in the constructor initializer list before using the constructor body to do further initializations.

The best practice is to initialize member variables in the initializer list in the same order those member variables are listed in the class.

Review: iterators, std::begin, std::end, etc.

With the **<iterator>** header, one unlocks functions such as

std::begin, returning an **iterator** pointing to the first element of a container (or a **const_iterator** if the container is **const**);

std::end, returning an iterator pointing just past the last element of a container (or a **const_iterator** if the container is **const**);

std::cbegin, returning a **const_iterator** (cannot modify the container elements) pointing to the first element of a container; and

std::cend, returning a **const_iterator** (cannot modify the container elements) pointing just past the last element of a container.

Review: iterators, std::begin, std::end, etc.

There are also **std::rbegin**, returning a **reverse_iterator** pointing to the last element of a container (or a **const_reverse_iterator** if the container is **const**);

std::rend, returning a **reverse_iterator** pointing just before the first element of a container (or a **const_reverse_iterator** if the container is **const**);

std::crbegin, returning a **const_reverse iterator** (cannot modify the container elements) pointing to the last element of a container;

and **std::crend**, returning a **const_reverse iterator** (cannot modify the container elements) pointing just before the first element of a container.

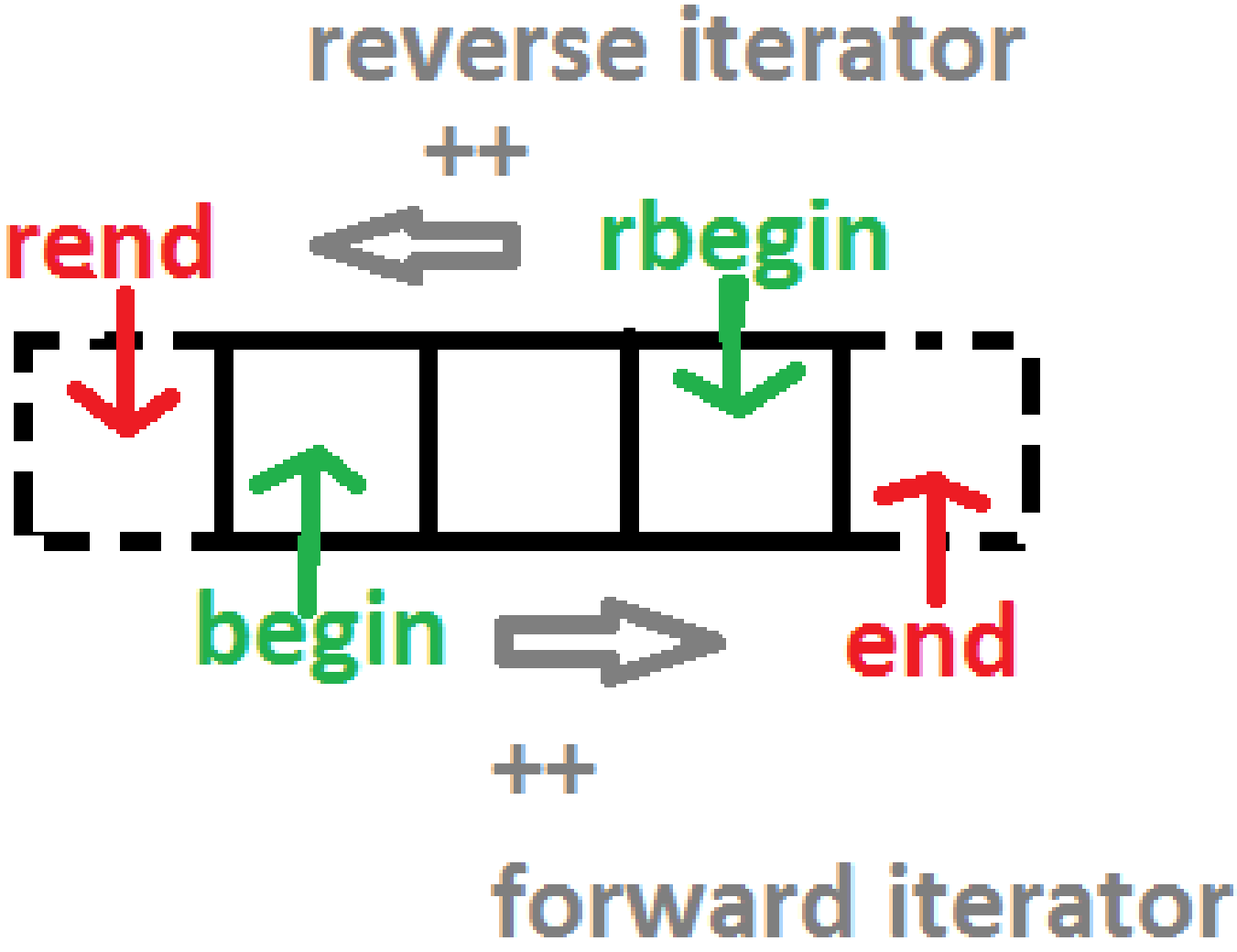
Review: iterators, std::begin, std::end, etc.

Although C-style arrays do not have member functions, these functions can still return pointers/iterators to the first element, just past the end, etc. for C-style arrays.

Remark 1: the support of **std::begin** and **std::end** began with C++11; in C++17, **std::rbegin**, etc., were added.

Remark 2: there are many versions of C++ out there. One of the biggest "revolutions" took place with C++11. Then came C++14. C++17 is the current most up-to-date version, but C++20 is already being planned.

Review: iterators, std::begin, std::end, etc.



Review: iterators, std::begin, std::end, etc.

```
// vec stores "one", "four", "eight"
std::vector<std::string> vec{"one", "four", "eight"};

// iter "points to" the "one"
std::vector<int>::const_iterator iter = std::cbegin(vec);
```

Review: iterators, std::begin, std::end, etc.

Reminder: putting a ***** before an iterator/pointer "dererences" it, giving the pointed-to object. Putting a **->** accesses class members if the iterators points to a class object.

```
std::cout << *iter << '\n'; // *iter is the first string
std::cout << iter->size() << '\n'; // called size function of that string
++iter; // now it "points to" the "four"
std::cout << *iter;

one
3
four

// *iter = "nine"; ERROR: cannot modify elements with const_iterator!
```

Review: Range-Based For-Loop and ‘auto’ Keyword

The **auto** keyword automatically detects the return type of the right-hand side and declares the variable type.

```
auto iter = std::cbegin(vec);

// same as:
//     std::vector<std::string>::const_iterator iter = std::cbegin(vec);

int i = 42;
auto& j = i; // same as:     int& j = i;
const int k = 12;
auto& L = k; // same as:     const int& L = k;
auto m = k; // same as:     int m = k;
const auto n = k; // same as:     const int n = k;
```

Subtleties apply when the **&&** symbol is used with **auto**. More on this much later.

Review: Range-Based For-Loop and ‘auto’ Keyword

Unless we add the **&** symbol, **auto** will not make a reference.

```
std::string s("hi");

auto t = s; // t is a copy of s

auto& u = s; // u is a reference to s

const auto& v = s; // v is a reference to const for s
```

Review: Range-Based For-Loop and ‘auto’ Keyword I

The following two codes are equivalent:

```
// assume words is an std::set<std::string>

// start iterator at beginning, moving through the set
for( auto itr = std::cbegin(words); itr != std::cend(words); ++itr) {
    // for each string, print its size
    std::cout << itr->size() << '\n';
}

// a RANGE BASED for loop
for( const auto & s : words ) { // for each string s
    std::cout << s.size() << '\n'; // print its size
}
```

Review: Range-Based For-Loop and 'auto' Keyword II

This is because the preceding snippet is "expanded" as

```
// past-end
auto end = std::end(words);

for( auto itr = std::begin(words) ; itr != end; ++itr) { // move through set

    // let s reference the iterator's string as const
    const std::string& s = *itr;

    // print the size of the string
    std::cout << s.size() << '\n';
}
```

Review: Range-Based For-Loop and 'auto' Keyword III

Note the correspondence. The [] indicate optional additional syntax.

```
for([const] auto [&] x : container) {
    // stuff with x
}

auto end = std::end(container);
for(auto itr = std::begin(container); iter != end; ++itr) {
    [const] auto [&] x = *iter;
    // stuff with x
}
```

Review: Avoid C-style Casts

Casting should be done explicitly by invoking the proper type of cast. In many cases this will be **static_cast**. For example,

```
int x = 13;
std::cout << static_cast<double>(x)/2;
```

is preferable to

```
std::cout << ((double)(x))/2;
```

The reason is that the C-style cast used directly above includes other types of casts:

reinterpret_cast,
const_cast, in addition to
static_cast.

It is better to be explicit about the type of cast that is intended.

Review: nullptr for a Pointer

The value **nullptr** is a **pointer literal**. When one needs a pointer that points to null, it is the most appropriate choice.

The **std::rand** function should be seeded as:

```
std::srand(std::time(nullptr));
```

The **std::time** member function accepts a **pointer** to a **time_t** variable.

NULL is actually just a pre-processor macro that is assigned either the value **0** (int) or **0L** (long int). Using **NULL** or **0** when a pointer is expected makes code harder to read and is more prone to error.

Review: Direct vs Copy (Indirect) Initialization

Although the end result of the lines of code below are the same, they are not, strictly speaking, the same.

```
std::string awesome = std::string("cat"); // copy initialization
std::string awesome = "cat"; // copy initialization

std::string awesome("cat"); // direct initialization
```

The former two are **copy initializations** that may result in temporary objects being constructed before making **awesome**. The latter directly constructs a **std::string** object called **awesome** from the string literal "cat".

In the absence of compiler optimization (which does often happen), direct initialization is always more efficient than copy initialization.

Review: Direct vs Indirect Initialization

Remark: when a function accepts arguments by value or returns by value, copy-initialization is done.

```
void bar(int i, std::string s) {
    std::cout << i << " " << s;
}

std::string quuz() {
    return "hi";
}
```

When **bar** is called, the local variables **i** and **s** are copy-initialized from the input arguments.

The output **std::string** from **quuz** is copy initialized from the value in its return statement.

Review: Functions and Classes with Default Arguments

Functions can be assigned default arguments from right to left. These default values are only used if the function is called without specifying these values. When a function is called, parameters are passed left to right.

```
/* suppose this is defined somewhere */
void foo(int i, int j = 4, int k = 5);

/* function foo used below */
foo(3); // same as foo(3, 4, 5);
foo(3, 2); // same as foo(3, 2, 5);
foo(1,8,6); // same as foo(1,8,6);
```

Review: Functions and Classes with Default Arguments

Remark: in giving the definitions, we do not specify the default arguments again if they were already specified in a preceding declaration. Below, we define the **foo** function:

```
void foo(int i, int j, int k){ // no default values given here
    std::cout << i + j + k;
}
```

Review: Functions and Classes with Default Arguments

Before the body of a constructor is run, any member variables that are not initialized in the consructor initializer list will be set by their default values if any are provided:

```
struct PIC10B {
    PIC10B() : funLevel(INT_MAX) { } // constructor

    int funLevel = 1000;
    std::string language = "C++";
};
```

Review: Functions and Classes with Default Arguments

The value of **funLevel** is set to **INT_MAX** as part of a constructor initializer so 1000 is unused.

But **language** is not given a value in the constructor initialization and because **language** is not initialized but is given a default value of **"C++"**, it is given this value prior to entering the empty constructor body.

Review: Return Type Efficiency

When returning class objects, it is best to return them as references unless the reference is to a local variable that may no longer exist...

```
class S {
private:
    std::string msg = "hello";
public:
    // no copying done: just reference the value
    const std::string& get_msg() const { return msg; }
};

// ...
S s;
std::cout <<s.get_msg(); // directly references member, no copy!
```

Review: Return Type Efficiency

```
std::string& not_good() {
    std::string local("not good"); // being made inside this function!

    // local ceases to exist but there is a reference to it!
    return local;
}

// ...
std::string& danger = not_good();

// UNDEFINED BEHAVIOUR: reference to destroyed variable
std::cout <<danger;
```

Typedefs

The **using** and **typedef** keywords can be used to define abbreviated variable types and give semantic meaning.

```
// so "money" means "double"
using money = double;
money change_due = 3.36;

// so "mark_pair" means std::pair<std::string, std::string>
typedef std::pair<std::string, std::string> mark_pair;
mark_pair joe( "Joe Bruin", "A+");
```

constexpr

Sometimes a variable can be set at **compile time** rather than **run time**.

The resulting program can be sped up with these modifications. The **constexpr** keyword allows for this sort of optimization.

If a variable or function is marked as **constexpr** then the it *can be set/run during compilation*. This will only happen if we request it from the compiler by using **constexpr**.

constexpr I

```
/**
Function computes the factorial of its input
@param n the value to compute n! for
@return n factorial = n(n-1)...(2)(1) or 1 if n=0
*/
constexpr int factorial(int n) {
    int nfact = 1;
    // if n==0, nfact stays as 1
    for(int i=1; i <= n; ++i) { // multiply 1 thru n
        nfact *= i;
    }
    return nfact;
}
// to be continued ...
```

constexpr II

```
int main() {
    constexpr int x = factorial(3); // done at compile time, 6
    constexpr int y = factorial(x); // done at compile time, 720

    const int a = 4; // done at run time: no constexpr
    int b = factorial(a); // done at run time

    // constexpr int c = factorial(a); ERROR: a not contexpr

    return y; // returns 720
}
```

If we look at the *assembly language instructions* from the program above, we find that the compiler already has determined that 720 should be returned from **main**, even before the program runs!!!

constexpr

There are many rules with **constexpr** but here are a few useful ideas to understand the preceding example:

- A **constexpr** variable can only be initialized from a *literal type* or a **constexpr** value returned from a function.
- A function marked **constexpr** must return a literal (or void).
- A function marked **constexpr** can be used with non-**constexpr** inputs, but the output is not **constexpr** and cannot be used to initialize a **constexpr** variable.
- A function marked **constexpr** cannot use non-literal variables.

constexpr

And for technical reasons to be explained later, a function that returns a **constexpr** type should be defined within a header file (or above main as shown).

A **"literal type"** is, for simplicity right now (some important cases are blatantly ignored), a fundamental type like **int**, **bool**, **double**, etc.; or a class that only has fundamental type member variables.

```
constexpr size_t nope() {
    std::string s; // not literal-type, will not work!
    return s.size();
}
```

Subtlety: Passing by Reference vs by Value I

Sometimes it can be okay to pass by value *if making a copy is unavoidable*. It can make the code look cleaner and, *depending on compiler optimizations*, there could even be a modest efficiency gain in directly passing by value rather than making a reference and then a local copy.

Often the two runtimes are nearly identical so we mostly do it for nicer looking code.

Subtlety: Passing by Reference vs by Value II

Consider a **Vector** class to represent a point (x, y) in $\mathbb{R}^2$.

```
struct Vector {
   float x = 0, y = 0;
};
```

Consider a function **add** that makes a new **Vector** whose **x** and **y** values are the corresponding sum of the **x** and **y** values of its inputs.

```
Vector a; // a==(0,0)
a.x = 3; // a==(3,0)
a.y = 4; // a==(3,4)
Vector b; // b==(0,0)
b.x = 10; // b==(10,0)
Vector c = add(a,b); // so c == (13,4)
```

Subtlety: Passing by Reference vs by Value III

```
// notice that v is a copy of the first argument
Vector add(Vector v, const Vector& w) {
    v.x += w.x;
    v.y += w.y;
    return v;
}
```

The sum of the two vectors means creating a new vector whose **x**-value is the sum of the two **x** values; likewise with the **y**-value.

Neither input is modified: a copy is made of the first argument and the second is passed as a reference to const.

Subtlety: Passing by Reference vs by Value I

The code below is more to write:

```
Vector add(const Vector& v, const Vector& w) {
    Vector copy(v); // make a copy of v
    copy.x += w.x;
    copy.y += w.y;
    return copy;
}
```

We created a new local variable *and made a reference*.
With the previous implementation, we *just made a copy* directly.

Braced Initialization

Since C++11, there is now braced initialization. There are some delicate rules to it, but it makes some tempting syntax valid. Consider:

```
struct X {
    X() { }
    X(int,double) { }
};
```

Then we use **X**:

```
X x(); // ERROR: should have written "X x;" to default construct, but...
X x{}; // okay, default constructs x
X x2{3,7.6}; // same as: "X x(3,7.6);"

X foo() {
    return { }; // same as "return X();" or "return X{};"
}
```

Braced Initialization

It can also be used with default values in classes.

```
struct Y {
    X x{4,11.1};
};
```

Braced Initialization

Warning: braced initialization can be a bit dangerous for data types that also accept lists of values to store...

```
std::vector<int> v {10,10}; // stores { 10, 10}, not ten 10's ...

std::vector<int> w(10,10); // stores ten 10's
```

Trailing Return Syntax

A C++ function can be written with a more mathematical notation.
Consider a function that squares its input argument...

$$f : \mathbb{R} \to \mathbb{R}$$
$$x \mapsto x^2$$

With "old school" notation, we define:

```
double f(double x) { return x*x; }
```

But we can now write:

```
auto f(double x) -> double { return x*x; }
```

Trailing Return Syntax

As a declaration, the syntax is:

auto function_name(list of arguments) -> output_type;

As a definition, the syntax is:

auto function_name(list of arguments) -> output_type { body; }

So, we could also write:

```
auto main() -> int { return 0; }
```

This syntax can also be quite useful with templates (to come).

Aside: Error in using NULL in place of nullptr

Consider overloading the **time** function:

```
void time(time_t seconds); // our own version
time_t time(time_t * ptr); // the std::time function of <ctime>
```

Assuming work in the standard namespace, calling

```
time(NULL);
```

would be ambiguous: NULL (**int** or **long int**) could be converted to **time_t** or to **time_t *** and neither instance of the overloaded function is a direct match.

Summary

- The C++ Standard governs how the language should behave.
- Proper code is readable, well-documented, and minimizes ambiguities and possible bugs.
- **size_t** is to be used for sizes and indices.
- Const correctness guards against bugs.
- There are range for loops, and many iterator functions.
- **auto** keyword helpful when return types are messy.
- Header files declare; cpp files define/implement.
- Efficiency is important: objects should (usually) be passed by reference or reference to const.
- Reference return values may be optimal for efficiency at times.
- Constructor initializer lists should be used instead of in-body initialization whenever possible.

Summary

- Direct initialization is better than copy initialization.
- The type of casting used should be clear.
- **using** and **typedef** can make notation simpler.
- Functions and classes can have **default arguments**.
- **constexpr** variables are set at compile time.
- **braced initalization** can construct objects similar to parentheses in listing arguments separated by commas but also applies to default construction.
- Functions can be defined with a **trailing return type**.

Exercises

1. Write and document a function **print_vec** accepting a vector storing **int**s and a boolean value, defaulted to true. When the boolean is set to true, it prints the elements of the vector in order; when the boolean is set to false, it prints the elements of the vector in the reverse order. When
   - printing in order, use a range-for loop;
   - printing in reverse, use iterators.
2. Write a **Person** class that stores a name and age, has a single constructor to initialize the name and age, has accessor functions to retrieve the name and age, has a mutator function to increase the age. Make the class const-correct and efficient.
3. Write a constexpr function, **with_tax**, that will compute the price of an item with tax (add ten percent).
4. Why is **size_t** important?