## A First Look at function

## Every function gets compiled at its first call only

## Nested function definition is illegal in c++

## 

## Forward declaration and definitions

## Rule: When fixing compilation error, always look for the first error occurred

## In a file one identifier can have one definition only

## In a program one function can have one definition only

## All Prototypes are declaration but not vice versa.

## Prototypes are stricter declaration, or declaration with types.

## 

## Rule: Use angled brackets to include header files that come with the compiler. Use double quotes to include any other header files.

## Rule: use the non .h version of a library if it exists, and access the functionality through the std namespace. If the non .h version does not exist, or you are creating your own headers, use the .h version

## Header files

* + 1. Always include header guards.
    2. Do not define variables in header files unless they are constants. Header files should generally only be used for declarations.
    3. Do not define functions in header files.
    4. Each header file should have a job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
    5. Give your header files the same name as the source files they’re associated with (e.g. grades.h goes with grades.cpp).
    6. Try to minimize the number of other header files you #include in your header files. Only #include what is necessary.
    7. Do not #include .cpp files.

## TBD

## Arrays and Pointers

## array cant be reassigned.

## While passing to a function as parameter,array decays to a pointer.

## Array name and pointer to array are not same

## int array[5] {1,2,3,4,5} -array is of int[5] type to compiler

## int \*ptr= array; //ptr is of int\*

## ptr = &array; //ptr is of int(\*)[5] type

## const char\* fn(), make return type exist , even after fn finishes.

## Two const char\* with same value can be optimised by compiler to be stored in single read only location.

## std::nullptr

## Null pointer passed as reference, compiler don't complain fn(int \*&ptr)

## If you pass it a non-char pointer, std::cout will simply print the contents of that pointer (the address that the pointer is holding). However, if you pass it an object of type char\* or const char\*, it will assume you’re intending to print a string. Consequently, instead of printing the pointer’s value, it will print the string being pointed to instead!

## Rule: Set deleted pointers to 0 (or nullptr in C++11) unless they are going out of scope immediately afterward.

## One often asked question of array delete[] is, “How does array delete know how much memory to delete?” The answer is that array new[] keeps track of how much memory was allocated to a variable, so that array delete[] can delete the proper amount. Unfortunately, this size/length isn’t accessible to the programmer.

## A dynamic array functions identically to a decayed fixed array, with the exception that the programmer is responsible for deallocating the dynamic array via the delete[] keyword.

## Just like a normal const variable, a const pointer must be initialized to a value upon declaration.

## A non-const pointer can be redirected to point to other addresses.

## A const pointer always points to the same address, and this address can not be changed.

## A pointer to a non-const value can change the value it is pointing to. These can not point to a const value.

## A pointer to a const value treats the value as const (even if it is not), and thus can not change the value it is pointing to.

## References:

## L-values are objects that have a defined memory address (such as variables), and persist beyond a single expression. r-values are temporary values that do not have a defined memory address, and only have expression scope. R-values include both the results of expressions (e.g. 2 + 3) and literals.

## Types:

## References to non-const values (typically just called “references”, or “non-const references”),

## References to const values (often called “const references”).

## C++11 added r-value references

## References must be initialised .

## References can not be reassigned.

## Best used as function parameter and to access nested data structure.

## One of the most annoying issues with C-style arrays is that in most cases they decay to pointers when evaluated. However, if a C-style array is passed by reference, this decaying does not happen.

## References are usually implemented internally by the compiler using pointers

## Unlike references to non-const values, which can only be initialized with non-const l-values, references to const values can be initialized with non-const l-value, const l-values, and r-values.

## References to r-values extend the lifetime of the referenced value.

## Rule: Pass non-pointer, non-fundamental data type variables (such as structs) by (const) reference.

## 

## Rule: Use references or const references for your element declaration in for-each loops for performance reasons.

## For-each doesn’t work with pointers to an array or dynamic array

## Rule: When using a pointer to access the value of a member, use operator-> instead of operator. (the . operator)

## Note that there is no such thing as a void reference. This is because a void reference would be of type void &, and would not know what type of value it referenced.

## A void pointer is a pointer that can point to any type of object, but does not know what type of object it points to. A void pointer must be explicitly cast into another type of pointer to be dereferenced. A null pointer is a pointer that does not point to an address. A void pointer can be a null pointer.

## Note that you can not set a pointer to a pointer directly to a value

## std::array

## std::array supports a second form of array element access (the at() function) that does bounds checking

## Rule: Always pass std::array by reference or const reference

## T

## T

## Chapter 7

* 1. Function parameters and Arguments
     1. On success “main ()” returns ‘0’ to operating system.
  2. Passing parameters by Value
     1. Good for fundamental data types and enumerators,
     2. Where parameters are not supposed to be altered.
     3. Non-const references can only reference non-const l-values (e.g. non-const variables), so a reference parameter cannot accept an argument that is a const l-value or an r-value (e.g. literals and the results of expressions).
     4. Using const is useful for several reasons:
        1. It enlists the compilers help in ensuring values that shouldn’t be changed aren’t changed, the compiler will throw an error if you try.
        2. It tells the programmer that the function won’t change the value of the argument. This can help with debugging.
        3. You can’t pass a const argument to a non-const reference parameter. Using const parameters ensures you can pass both non-const and const arguments to the function.
        4. Const references can accept any type of argument, including l-values, const l-values, and r-values.
     5. As a reminder, you can pass a C-style array by pointer by reference. This is useful if you need the ability for the function to change the array (e.g. for a sort function) or you need access to the array’s type information of a fixed array (to do sizeof() or a for-each loop). However, note that for this to work, you explicitly must define the array size in the parameter. If you want this to work with fixed arrays of any length, you can make the array length a template parameter
     6. References must be initialized, so there’s no worry about null values.
     7. void foo(int \*&ptr) // pass pointer by reference
     8. void printElements(int (&arr)[4])
     9. Always use const reference unless you are sure that you must update the parameter.
  3. Returning by Value, Reference and address or Tuple(c++11)
     1. Every C++ expression is either an lvalue or an rvalue. An lvalue refers to an object that persists beyond a single expression. You can think of an lvalue as an object that has a name. All variables, including nonmodifiable (const) variables, are lvalues. An rvalue is a temporary value that does not persist beyond the expression that uses it.
     2. int value = returnByReference(); // case A -- ok, treated as return by value
     3. int &ref = returnByValue(); // case B -- compile error since the value is an r-value, and an r-value can't bind to a non-const reference
     4. const int &cref = returnByValue(); // case C -- ok, the lifetime of the return value is extended to the lifetime of cref
  4. Inline Function
     1. One should avoid in line with global function in header files
     2. Function with extern linkage should not be defined in header files, However, inline functions are exempt from the rule that you can only have one definition per program, You can explicitly control the linkage of a symbol by using the extern and static keywords. If the linkage isn't specified, then the default linkage is extern for non-const symbols and static(internal) for const symbols.
  5. Function Overloading
     1. Function return type is not considered for its uniqueness
  6. Default Parameters
     1. Rule: If the function has a forward declaration, put the default parameters there. Otherwise, put them in the function definition.
     2. Default parameters don’t contribute in the unique ness of a function while function overloading.
     3. Default parameter declaration must be right bound.
     4. They can be defined once , either in declaration or definition.
  7. Function Pointers
     1. Default parameters won’t work for functions called through function pointers. Default parameters are resolved at compile-time (that is, if you don’t supply an argument for a defaulted parameter, the compiler substitutes one in for you when the code is compiled). However, function pointers are resolved at run-time.
     2. typedef bool (\*fnctPtr)(int, int) or
     3. fnctPrt = bool(\*)(int , int) // alias C++11
     4. C++ new type

#include <functional>

bool myFunction(int a, int b, std::function<bool(int, int)> fnctPntr)

* 1. The Stack and the heap
     1. Class of storage
        1. The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
        2. The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
        3. The data segment (also called the initialized data segment), where initialized global and static variables are stored.
        4. The heap, where dynamically allocated variables are allocated from.
        5. The call stack, where function parameters, local variables, and other function-related information are stored.
     2. Allocating memory on the heap is comparatively slow.
     3. Allocated memory stays allocated until it is specifically deallocated (beware memory leaks) or the application ends (at which point the OS should clean it up).
     4. Dynamically allocated memory must be accessed through a pointer. Dereferencing a pointer is slower than accessing a variable directly.
     5. Because the heap is a big pool of memory, large arrays, structures, or classes can be allocated here.
     6. Let’s examine in more detail how the call stack works. Here is the sequence of steps that takes place when a function is called:

1. The program encounters a function call.
2. A stack frame is constructed and pushed on the stack. The stack frame consists of:
   * The address of the instruction beyond the function call (called the **return address**). This is how the CPU remembers where to return to after the called function exits.
   * All function arguments.
   * Memory for any local variables.
   * Saved copies of any registers modified by the function that need to be restored when the function returns
3. The CPU jumps to the function’s start point.
4. The instructions inside of the function begin executing.

When the function terminates, the following steps happen:

1. Registers are restored from the call stack
2. The stack frame is popped off the stack. This frees the memory for all local variables and arguments.
3. The return value is handled.
4. The CPU resumes execution at the return address.
   * 1. All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
   1. Std::vector capacity and stack behavior
      1. Std::vector<int> array[5]; array = {1,2,3} // length = 3, capacity=5
      2. When a vector is resized, the vector may allocate more capacity than is needed. If, when, and how much additional capacity is allocated is left up to the compiler implementer.
   2. Recursion
      1. Rule: Generally favor iteration over recursion, except when recursion really makes sense*.*
   3. Handling errors, cerr and exit
      1. cerr is another mechanism that is meant specifically for printing error messages. **cerr** is an output stream (just like cout) that is defined in <iostream>. Typically, cerr writes the error messages on the screen (just like cout), but it can also be individually redirected to a file.
      2. assert(found && "Car could not be found in database"); //A C-style string always evaluates true
      3. The assert() function comes with a small performance cost that is incurred each time the assert condition is checked. Furthermore, asserts should (ideally) never be encountered in production code (because your code should already be thoroughly tested). Consequently, many developers prefer that asserts are only active in debug builds. C++ comes with a way to turn off asserts in production code: #define NDEBUG.
      4. Unlike assert, which operates at runtime, static\_assert is designed to operate at compile time, e.g. static\_assert(sizeof(int) == 4, "int must be 4 bytes");
   4. Command line argument
      1. **argc** is an integer parameter containing a count of the number of arguments passed to the program (think: argc = **arg**ument **c**ount). argc will always be at least 1, because the first argument is always the name of the program itself. Each command line argument the user provides will cause argc to increase by 1.
      2. std::stringstream convert(argv[1]);
   5. Ellipses
      1. Functions that use ellipsis must have at least one non-ellipsis parameter.
      2. return\_type function\_name(argument\_list, **...**)
      3. We have to include the cstdarg header. This header defines va\_list, va\_arg, va\_start, and va\_end.
      4. A **sentinel** is a special value that is used to terminate a loop when it is encountered. For example, with strings, the null terminator is used as a sentinel value.
      5. For those of you coming from C, this is what printf does!
   6. TBD.

## Basic Object-Oriented Programming

* 1. Welcome to object-oriented programming

In traditional sense object is a memory space. In OOPS realm, along with

being a memory space, it also has associated variables and methods.

* 1. Classes and class members
     1. A class defaults its member to private where as a struct defaults to public.
     2. Class object is passed implicitly while calling any method of class.
  2. Access specifier
     1. One nuance of C++ that is often missed or misunderstood is that access control works on a per-class basis, not a per-object basis. This means that when a function has access to the private members of a class, it can access the private members of *any* object of that class type that it can see.
  3. Access functions and encapsulation(Information hiding)
     1. Private attributes can only be accessed by another class member only
     2. Rule: In general make member variable private and member methods

Public.

* + 1. Benefit:

Encapsulated classes are easier to use and reduce the complexity of your programs.

Encapsulated classes are easier to change.

Encapsulated classes are easier to debug.

* + 1. Access functions typically come in two flavors: getters and setters. **Getters** are functions that return the value of a private member variable. **Setters** are functions that set the value of a private member variable.
    2. Getters should usually return by value or const reference, not non-const reference
  1. Construtor
     1. Rule: Use member initializer lists to initialize your class member variables instead of assignment*.*
     2. Even the const member variables can be initialized in member initialization list.
     3. Copy, Direct and Uniform initialization, are the three types of initialization available in C++.
     4. Variables in the initializer list are not initialized in the order that they are specified in the initializer list. Instead, they are initialized in the **order in which they are declared in the class**.
  2. Overlapping and delegating constructor
     1. Constructors *are* allowed to call non-constructor functions in the class.
     2. Delegating constructor: Make sure you’re calling the constructor from the member initializer list, not in the body of the constructor.
  3. Destructor
     1. Under the RAII (Resource Acquisition Is Initialization) paradigm, objects holding resources should not be dynamically allocated. This is because destructors are only called when an object is destroyed. For objects allocated on the stack, this happens automatically when the object goes out of scope, so there’s no need to worry about a resource eventually getting cleaned up. However, for dynamically allocated objects, the user is responsible for deletion -- if the user forgets to do that, then the destructor will not be called, and the memory for both the class object and the resource being managed will be leaked!
     2. Note that if you use the exit() function, your program will terminate and no destructors will be called. Be wary if you’re relying on your destructors to do necessary cleanup work
  4. Hidden this pointer
     1. Because \*this is just a function parameter, it doesn’t add any memory usage to your class (just to the member function call, since that parameter goes on the stack while the function is executing).
     2. One can chain member functions by returning “this” pointer e.g. cout print multiple string by same method.
  5. Class code and header files
     1. Types (which include classes), are exempt from the part of the one-definition rule that says you can only have one definition per program. Therefore, there isn’t an issue #including class definitions into multiple code files.
     2. Member functions defined inside the class definition are considered implicitly inline. Inline functions are exempt from the one definition per program part of the one-definition rule.
     3. Default parameters for member functions should be declared in the class definition (in the header file), where they can be seen by whomever #includes the header.
  6. Const Class object and member function
     1. Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed.
     2. It turns out that const class objects can only explicitly call *const* member functions. A **const member function** is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object).
     3. To make a const member function, we simply append the const keyword to the function prototype, after the parameter list, but before the function body
     4. For member functions defined outside of the class definition, the const keyword must be used on both the function prototype in the class definition and on the function definition

Int foo::func() const

* + 1. Const constructors are not allowed.
    2. We can overload const and non const functions.
  1. Static member variable
     1. The static keyword has another meaning when applied to global variables -- it gives them internal linkage (which restricts them from being seen/used outside of the file they are defined in). Because global variables are typically avoided, the static keyword is not often used in this capacity.
     2. When we declare a static member variable inside a class, we’re telling the compiler about the existence of a static member variable, but not actually defining it (much like a forward declaration). Because static member variables are not part of the individual class objects (they are treated similarly to global variables, and get initialized when the program starts), you must explicitly define the static member outside of the class, in the global scope.
     3. When the static member is a const integral type (which includes char and bool) or a const enum, the static member can be initialized inside the class definition itself.
     4. In C++11, “constexpr” , all types to be defined and initialized in class.

 static constexpr double s\_value = 2.2;

* + 1. A class can have a static self-object or static self-pointer or self-pointer itself, but can’t have an automatic self-object. As static member doesn’t contribute to the size of objects and pointer is of fix size, irrespective of the field, pointer is pointing. However, in case of an automatic object compiler can’t decide the size of class with object inside.
  1. Static member function
     1. Public static member variables can be accessed using class scope
     2. Private static member access need public static member function.
     3. Accessing static member function don’t have an \*this attachment, they don’t need an object to be created for access.
  2. Friend functions and classes
     1. A friend function or class is a function or class that can access the private members of another class as though it were a member of that class. This allows the friend or class to work intimately with the other class, without making the other class expose its private members (e.g. via access functions).
     2. Friending is uncommonly used when two or more classes need to work together in an intimate way, or much more commonly, when defining overloading operators (which we’ll cover in chapter 9).
     3. Note that making a specific member function a friend requires the full definition for the class of the member function to have been seen first.
  3. Anonymous Objects
     1. In C++, anonymous objects are primarily used either to pass or return values without having to create lots of temporary variables to do so. Memory allocated dynamically is also done so anonymously (which is why its address must be assigned to a pointer, otherwise we’d have no way to refer to it).
     2. However, it is worth noting that anonymous objects are treated as rvalues (not lvalues, which have an address). This means anonymous objects can only be passed or returned by value or const reference. Otherwise, a named variable must be used instead.
  4. Nested types in classes
     1. The nested class does not have any special access to the “this” pointer of the enclosing class.
  5. Timing your code
     1. #include <chrono> // for std::chrono functions – Good thing to try.
  6. TBD
  7. TBD
  8. TBD

## Operator overloading

## Introduction

## Almost any existing operator in C++ can be overloaded. The exceptions are: conditional (?:), sizeof, scope (::), member selector (.), and member pointer selector (.\*).

## All operators keep their default precedence and associativity (regardless of what they’re used for) and this can not be changed.

## When overloading operators, it’s best to keep the function of the operators as close to the original intent of the operators as possible.

## Overloading operator using friend/normal function

## Prefer overloading operators as normal functions instead of friends if it’s possible to do so without adding additional functions*.*

## Overloading operator using member function

* + 1. If you’re overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
    2. If you’re overloading a unary operator, do so as a member function.
    3. If you’re overloading a binary operator that modifies its left operand (e.g. operator+=), do so as a member function if you can.
    4. If you’re overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function or friend function.

## TBD

## **An introduction to object relationship (part-of, has-a, uses-a)**

## Compositoin

To qualify as a **composition**, an object and a part must have the following relationship:

* + 1. The part (member) is part of the object (class)
    2. The part (member) can only belong to one object (class) at a time
    3. The part (member) has its existence managed by the object (class)
    4. The part (member) does not know about the existence of the object (class)

## Compositions are unidirectional in nature.

## TBD

## Aggregation

To qualify as an **aggregation**, a whole object and its parts must have the following relationship:

* + 1. The part (member) is part of the object (class)
    2. The part (member) can belong to more than one object (class) at a time
    3. The part (member) does *not* have its existence managed by the object (class)
    4. The part (member) does not know about the existence of the object (class)

## Like a composition, an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship. However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts.

## Aggregation model is “has-a” relationship

## TBD

## Association

To qualify as an **association**, an object and another object must have the following relationship:

* + 1. The associated object (member) is otherwise unrelated to the object (class)
    2. The associated object (member) can belong to more than one object (class) at a time
    3. The associated object (member) does *not* have its existence managed by the object (class)

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Composition** | **Aggregation** | **Association** |
| Relationship type | Whole/part | Whole/part | Otherwise unrelated |
| Members can belong to multiple classes | No | Yes | Yes |
| Members existence managed by class | Yes | No | No |
| Directionality | Unidirectional | Unidirectional | Unidirectional or bidirectional |
| Relationship verb | Part-of | Has-a | Uses-a |

* + 1. The associated object (member) may or may not know about the existence of the object (class)
    2. Association can be defined as “uses-a” relation.

## TBD

## Dependencies

## Dependencies are generally not at class/object level

## Container Classes

## Std::Initializer list

## TBD

## Introduction to inheritance (is-a)

## Order of construction

## C++ allocates memory for Base, then calls Base’s default constructor to do the initialization.

## When C++ constructs derived objects, it does so in phases. First, the most-base class (at the top of the inheritance tree) is constructed first. Then each child class is constructed in order, until the most-child class (at the bottom of the inheritance tree) is constructed last.

## TBD

## Constructor and initialization of Derived Classes

Here’s what actually happens when derived is instantiated:

* + 1. Memory for derived is set aside (enough for both the Base and Derived portions)
    2. The appropriate Derived constructor is called
    3. **The Base object is constructed first using the appropriate Base constructor**. If no base constructor is specified, the default constructor will be used.
    4. The initialization list initializes variables
    5. The body of the constructor executes
    6. Control is returned to the caller

## Constructors can only call constructors from their immediate parent/base class.

## Destructor call sequence is opposite to constructor, it starts from most derived and goes upto most base.

## TBD

## Inheritance and Class specifier

|  |  |  |  |
| --- | --- | --- | --- |
| **Access specifier in base class** | **Access specifier when inherited publicly** | **Access specifier when inherited privately** | **Access specifier when inherited protectedly** |
| Public | Public | Private | Protected |
| Private | Inaccessible | Inaccessible | Inaccessible |
| Protected | Protected | Private | Protected |

## When members are inherited, the access specifier for an inherited member may be changed (in the derived class only) depending on the type of inheritance used, as above.

* + 1. Because Derived is a Base, Derived has access to stuff in Base. However, Base does not have access to anything in Derived.

## TBD

## Calling Inherited function and overriding its behavior

## When a member function is called with a derived class object, the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the parent classes. It uses the first one it finds.

## When you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as private in the base class can be redefined as public in the derived class, or vice-versa!

## Use resolution operator “::” to call any specifics of Base class in derived.

## out << static\_cast<Base>(d); To call friend function of Base as overloaded operator here “<<”

## Because a Derived is-a Base, we can static\_cast our Derived object into a Base, so that the appropriate version of operator<< that uses a Base is called.

## C++ gives us the ability to change an inherited member’s access specifier in the derived class. This is done by using a *using declaration* to identify the (scoped) base class member that is having its access changed in the derived class, under the new access specifier.

## TBD

## TBD

## Virtual Function

## A base class pointer/reference have visibility to its member only, same is true for derived class object/pointer/reference.

## Even if you assign derived class object to base class, base class always have visibility to its local members only.

## A **virtual function** is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class. This capability is known as **polymorphism**. A derived function is considered a match if it has the same signature (name, parameter types, and whether it is const) and return type as the base version of the function. Such functions are called **overrides**.

## Never call virtual functions from constructors or destructors. As in case of constructor, derived object would not have been created by then and in case of destructor, derived objects would be destroyed by then.

## Return\_type func\_name const override{} //must use override for virtual function.

## There is one special case in which a derived class virtual function override can have a different return type than the base class and still be considered a matching override. If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class. These are called **covariant return types**.

## We then call b->getThis(). Variable b is a Base pointer to a Derived object. Base::getThis() is virtual function, so this calls Derived::getThis(). Although Derived::getThis() returns a Derived\*, because base version of the function returns a Base\*, the returned Derived\* is downcast to a Base\*. And thus, Base::printType() is called.

In other words, in the above example, you only get a Derived\* if you call getThis() with an object that is typed as a Derived object in the first place.

## TBD.

## Destructors

## There may be cases where you want a Base pointer to a Derived object to call Base::getName() instead of Derived::getName(). To do so, simply use the scope resolution operator

## You should **always** make your destructors virtual if you’re dealing with inheritance.

## Early (static) and Late(Dynamic) binding

## Direct function calls can be resolved using a process known as early binding. **Early binding** (also called static binding) means the compiler (or linker) is able to directly associate the identifier name (such as a function or variable name) with a machine address.

## In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as **late binding** (or dynamic binding). In C++, one way to get late binding is to use function pointers.

## TBD

## Pure virtual function , abstract class and interface claass

* + 1. C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
    2. To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

## First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can not be instantiated! Second, any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

## It turns out that we can define pure virtual functions that have also bodies. When providing a body for a pure virtual function, the body must be provided separately (not inline). if the derived class is happy with the default implementation provided by the base class, it can simply call the base class implementation directly using scope specifier.

## An **interface class** is a class that has no member variables, and where *all* of the functions are pure virtual!

## Abstract classes still have virtual tables, as these can still be used if you have a pointer or reference to the abstract class. The virtual table entry for a pure virtual function will generally either contain a null pointer, or point to a generic function that prints an error (sometimes this function is named \_\_purecall) if no override is provided.

## Virtual Base Class

## In famous diamond problem of inheritance, while creating object of most derived class, most base class get created twice. To avoid this, one should use “**virtual base class**”. To share a base class, simply insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only one base object that is shared.

## In above case only the most derived class create the most base class. Constructors of the intermittent parent classes are ignored. Intermittent parent class constructors are used, for creating their individual object. These intermittent classes get access of base class members using one virtual table maintained by compiler.

## First, virtual base classes are always created before non-virtual base classes, which ensures all bases get created before their derived classes.

## Object Slicing

## If derived object assigned to reference or pointer base class, it is typed as base and worked accordingly. But if directly a derived object get assigned to base object only the base part get assigned and hence sliced.

## The elements of std::vector must be assignable, whereas references can’t be reassigned (only initialized).

## std::vector<std::reference\_wrapper<Base> > v; // our vector is a vector of std::reference\_wrapper wrapped Base (not Base&)

## Frankenobject:

int main()

{

Derived d1(5);

Derived d2(6);

Base &b = d2;

b = d1; // this line is problematic

return 0;

}

you’ll discover that d2 now has the Base portion of d1 and the Derived portion of d2.

## **Dynamic Casting (Down casting)**

* + 1. *Rule: Always ensure your dynamic casts actually succeeded by checking for a null pointer result*.
    2. Note that Because dynamic\_cast does some consistency checking at runtime (to ensure the conversion can be made), use of dynamic\_cast does incur a performance penalty.
    3. Also note that there are several cases where downcasting using dynamic\_cast will not work:  
       1) With protected or private inheritance.  
       2) For classes that do not declare or inherit any virtual functions (and thus don’t have a virtual table).  
       3) In certain cases involving virtual base classes

## Dynamic cast work good with pointers as well with references. Because C++ does not have a “null reference”, dynamic\_cast can’t return a null reference upon failure. Instead, if the dynamic\_cast of a reference fails, an exception of type std::bad\_cast is thrown.

## In general, using a virtual function *should* be preferred over downcasting.

## Only member functions can be virtualized -- this makes sense, since only classes can inherit from other classes, and there’s no way to override a function that lives outside of a class (you can overload non-member functions, but not override them).

## TBD

## Templates

## Function Templates

## In C++, **function templates** are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**. Once we have created a function using these placeholder types, we have effectively created a “function stencil”.

## To create a template type parameter, use either the keyword *typename* or *class*. There is no difference between the two keywords in this context, so which you use is up to you. Note that if you use the class keyword, the type passed in does not actually have to be a class (it can be a fundamental variable, pointer, or anything else that matches).

## Template Classes

## A template is not a class or a function -- it is a stencil used to create classes or functions.

## With non-template classes, the common procedure is to put the class definition in a header file, and the member function definitions in a similarly named code file. In this way, the source for the class is compiled as a separate project file. However, with templates, this does not work. Gives linking error.

## In order for the compiler to use a template, it must see both the template definition (not just a declaration) and the template type used to instantiate the template.

## Template non type parameter

## Function Template specialization

## We can use a **function template specialization** (sometimes called a full or explicit function template specialization) to create a specialized version of the print() function for type double. This is extremely simple: simply define the specialized function (if the function is a member function, do so outside of the class definition), replacing the template type with the type you wish to redefine the function for.

## Class Template specialization (Similar as above)

## Partial Template specialization

## Note that as of C++14, partial template specialization can only be used with classes, not template functions (functions must be fully specialized).

## Partial template specialization allows us to specialize classes (but not individual functions!) where some, but not all, of the template parameters have been explicitly defined.

## TBD

## Partial template specialization for pointers

## The fully specialized version will take precedence over the partially specialized version.

## TBD

## Exceptions

## Constructors don’t have any return type, but can have reference parameter as return argument.

## Basic Exception handling

## Use “try, throw, catch”

## Exceptions of fundamental types can be caught by value, but exceptions of non-fundamental types should be caught by const reference to avoid making an unnecessary copy.

## Once the exception was handled, the program continued as normal after the catch blocks.

## Note that the compiler will not perform implicit conversions or promotions when matching exceptions with catch blocks!

## a catch block may throw another exception. Because the catch block is outside of the try block, the newly thrown exception in this case is not handled by the preceding try block -- it’s handled by the next enclosing try block.

## TBD

## Exceptions, functions and Stack unwinding

## Uncaught exceptions, catch-all handlers, and exception specifiers

## When main() terminates with an unhandled exception, the operating system will generally notify you that an unhandled exception error has occurred. How it does this depends on the operating system

## A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch.

* + 1. “catch(...) {}” …one can windup its main with this .
    2. TBD

## Exception, Classes and inheritance

## Constructors are another area of classes in which exceptions can be very useful. If a constructor fails, simply throw an exception to indicate the object failed to create. The object’s construction is aborted and its destructor is never executed (note: this means your exception handler should handle any necessary cleanup).

## *Note that exception handlers should catch class exception objects by reference instead of by value. This prevents the compiler from making a copy of the exception, which can be expensive when the exception is a class object, and prevents object slicing when dealing with derived exception classes (which we’ll talk about in a moment). Catching exceptions by pointer should generally be avoided unless you have a specific reason to do so.*

## Objects of type Base will not match the Derived handler (Derived is-a Base, but Base is not a Derived)

## Handlers for derived exception classes should be listed before those for base classes.

## Use std::exception for all 21 types of predefined exceptions.

## TBD

## Re-throwing exceptions

## only exceptions thrown within a try block are eligible to be caught. This means that an exception thrown within a catch block will not be caught by the catch block it’s in. Instead, it will be propagated up the stack to the caller.

## Use “throw;” to re-throw.

## Function Try blocks

## Function try blocks are designed to allow you to establish an exception handler around the body of an entire function, rather than around a block of code.

## With function-level try blocks, you must throw or rethrow an exception. If you do not explicitly throw a new exception, or rethrow the current exception (using the throw keyword by itself), the exception will be implicitly rethrown up the stack.

## When construction of an object fails, the destructor of the class is not called. Consequently, you may be tempted to use a function try block as a way to clean up a class that had partially allocated resources before failing. However, referring to members of the failed object is considered undefined behavior since the object is “dead” before the catch block executes. This means that you can’t use function try to clean up after a class.

## TBD

## Exception dangers and downsides

## std::unique\_ptr that can be used for this purpose. std::unique\_ptr is a template class that holds a pointer, and deallocates it when it goes out of scope.

## Exceptions should never be thrown in destructors.The problem occurs when an exception is thrown from a destructor during the stack unwinding process. If that happens, the compiler is put in a situation where it doesn’t know whether to continue the stack unwinding process or handle the new exception. The end result is that your program will be terminated immediately.

## TBD

## TBD

## Move Semantics and smart pointers

## Intro to smart pointers and move semantics

## Such a class is called a smart pointer. A Smart pointer is a composition class that is designed to manage dynamically allocated memory and ensure that memory gets deleted when the smart pointer object goes out of scope. (Relatedly, built-in pointers are sometimes called “dumb pointers” because they can’t clean up after themselves).

## TBD

## R Value references

## L-VAlue

L-value reference Can be initialized with Can modify

Modifiable l-values Yes Yes

Non-modifiable l-values No No

R-values No No

L-value reference to const Can be initialized with Can modify

Modifiable l-values Yes No

Non-modifiable l-values Yes No

R-values Yes No

## R-Values

## R-value reference Can be initialized with Can modify

Modifiable l-values No No

Non-modifiable l-values No No

R-values Yes Yes

R-value reference to const Can be initialized with Can modify

Modifiable l-values No No

Non-modifiable l-values No No

R-values Yes No

## TBD

## Move Constructors and move assignments

## Rule: If you want a move constructor and move assignment that do moves, you’ll need to write them yourself.

## The C++ specification has a special rule that says automatic objects returned from a function by value can be moved even if they are l-values. This makes sense, since res was going to be destroyed at the end of the function anyway! We might as well steal its resources instead of making an expensive and unnecessary copy.

## std::move, converts lvalue to rvalue.

## std::unique\_ptr

## Rule: Favor std::array, std::vector, or std::string over a smart pointer managing a fixed array, dynamic array, or C-style string

## std::unique\_ptr has a cast to bool that returns true if the std::unique\_ptr is managing a resource.

## Rule: use std::make\_unique() instead of creating std::unique\_ptr and using new yourself

## In general, you should not return std::unique\_ptr by pointer (ever) or reference (unless you have a specific compelling reason to).

## There are two easy ways to misuse

## std::unique\_ptrs, both of which are easily avoided. First, don’t let multiple classes manage the same resource.

## Second, don’t manually delete the resource out from underneath the std::unique\_ptr.

## TBD

## std::shared\_ptr

## Rule: Always make a copy of an existing std::shared\_ptr if you need more than one std::shared\_ptr pointing to the same resource.

## Use std::make\_shared for creating shared pointer.

## A std::unique\_ptr can be converted into a std::shared\_ptr via a special std::shared\_ptr constructor that accepts a std::unique\_ptr r-value. The contents of the std::unique\_ptr will be moved to the std::shared\_ptr.

## In C++14 and earlier, std::shared\_ptr does not have proper support for managing arrays, and should not be used to manage a C-style array. As of C++17, std::share\_ptr does have support for arrays. However, as of C++17, std::make\_shared is still lacking proper support for arrays, and should not be used to create shared arrays. This will likely be addressed in C++20.

## TBD

## Circular dependency issues with std::shared\_ptr, and std::weak\_ptr

## A Circular reference (also called a cyclical reference or a cycle) is a series of references where each object references the next, and the last object references back to the first, causing a referential loop. The references do not need to be actual C++ references -- they can be pointers, unique IDs, or any other means of identifying specific objects.

In the context of shared pointers, the references will be pointers.

## auto ptr1 = std::make\_shared<Resource>();

ptr1->m\_ptr = ptr1;

## std::shared\_ptr can be used when you need multiple smart pointers that can co-own a resource. The resource will be deallocated when the last std::shared\_ptr goes out of scope. std::weak\_ptr can be used when you want a smart pointer that can see and use a shared resource, but does not participate in the ownership of that resource.

## TBD

## TBD

## The Standard Template Library

## The STL

## STL container overview:

## As of C++11, the STL contains 6 sequence containers: std::vector, std::deque, std::array, std::list, std::forward\_list, and std::basic\_string.

## String

## Associative container: set, multiset, map, multimap,

## Container adapter : stack, queue, priority queue

## TBD

## STL Iterator overview

## all containers provide (at least) two types of iterators:

## container::iterator provides a read/write iterator

## container::const\_iterator provides a read-only iterator

* + 1. begin() - first element
    2. end() - Just after the last element.

## STL algorithms overview

## sort, reverse, find, min\_element, max\_element and many more.

## TBD

## std::string

## std::string and std::wstring

## These are the two classes that you will actually use. std::string is used for standard ascii (utf-8) strings. std::wstring is used for wide-character/unicode (utf-16) strings. There is no built-in class for utf-32 strings (though you should be able to extend your own from basic\_string<> if you need one).

## TBD

## std::string construction and destruction

## There are many string construction methods supported apart from construction with number i.e. std::string uString(3)

## One need to use std::ostringstream to construct string using numbers.

## std:: string length and capacity:

## use length() or size() for getting length of string.

## Capacity is always greater or equal to length, compiler keeps so to avoid frequent reallocation.

## TBD

## Input and Output (I/O)

## in -- an istream\_withassign class tied to the standard input (typically the keyboard)

## cout -- an ostream\_withassign class tied to the standard output (typically the monitor)

## cerr -- an ostream\_withassign class tied to the standard error (typically the monitor), providing unbuffered output

## clog -- an ostream\_withassign class tied to the standard error (typically the monitor), providing buffered output

## Input with istream

## cin >>uString , cin>>setw(10)>>uString

## cin.get(), cin.getline() , std::string::getline(cin,sString)

## Output with ostream and ios

## There are two ways to change the formatting options: flags, and manipulators. You can think of flags as boolean variables that can be turned on and off. Manipulators are objects placed in a stream that affect the way things are input and output.

## To switch a flag on, use the setf() function,

## std::cout.setf(std::showpos); // turn on the std::showpos flag

## use unsetf() for the opposite

## Flags live in the ios class, manipulators live in the std namespace, and the member functions live in the ostream class.

## TBD

## Stream Classes for strings

## use class “stringstream os” there are many associated utility functions.

## Stream states and input validation

## The ios\_base class contains several state flags that are used to signal various conditions that may occur when using streams

## TBD

## Basic File I/O

## ifstream, ofstream, fstream

## TBD

## Random File I/O

## Random file access is done by manipulating the file pointer using either seekg() function (for input) and seekp() function (for output). In case you are wondering, the g stands for “get” and the p for “put”. For some types of streams, seekg() (changing the read position) and seekp() (changing the write position) operate independently -- however, with file streams, the read and write position are always identical, so seekg and seekp can be used interchangeably.

## Two other useful functions are tellg() and tellp(), which return the absolute position of the file pointer. This can be used to determine the size of a file

## The fstream class is capable of both reading and writing a file at the same time -- almost! The big caveat , with lot of riders :)

## Rule: Do not write addresses to files. The variables that were originally at those addresses may be at different addresses when you read their values back in from disk, and the addresses will be invalid.

## TBD

* C++ General coding guidelines :
  + Forward declaration saves compilation time, as include causes more files to be opened and compiled.