C++ Comeback Tour - Taken from the tour at: <https://isocpp.org/tour>

Chapter 1 The Basics:

Intro:-

This chapter informally introduces:

* the notation of C++,
* C++’s model of memory and computation,
* and the basic mechanism for oransising code into a program.

These are the language facilities supporting the styles most often seen in C and sometimes called procedural programming. C++ is a compiled language - meaning for the program to run, it’s source text has to be processed by a compiler, which produces object files, which are combined by a linker, yielding an executable program. A C++ program typically consists of many source files. And an executable is platform specific, as in it isn’t portable. In regards to C++’s portability it is usually referring to the portability of the source code.

The ISO C++ standard defines two kind of entities:

* Core language features - such as built in types, loops and while statements
* Standard library components - such as containers and I/O operations

Standard library is provided with most if not all implementations of C++.

C++ is a statically typed language - that is every entity (e.g. object, value, name, and expression) must be known to the compiler at the time of use. The type of an object determines the set of operations that are applicable to it.

A minimal C++ program is as follows:

Int main(){} //minimal c++ program

It is a function called main, which takes no arguments and does nothing. The {} express grouping within C++, here they indicate the start and end of the function body. The double // refers to a comment, till the end of the current line.

Every C++ program must have exactly one global function called main. The int value returned by main(), if any, is the program’s return value to ‘the system’, \*

\*if no value is returned it will receive a successful compilation signal

A non zero value from the main() indicates failure, and not all operating systems and/or execution environments make use of this return value.

Typically a program produces some output, the following is the classic hello world example:

#include <iostream>

Int main(){ std::cout << “Hello, World!\n”; } //prints hello world

The include line instructs the compiler to include the declaration of the standard stream I/O facilities as found in iostream. Without this declaration, the expression would make no sense to the compiler. The operator << (”put too”), writes it’s second argument onto it’s first. In this case the string literal, “Hello, World!\n”, is written onto the standard output stream std::cout. In a string literal a \ followed by another character, denotes a ‘special character’; in this case the newline special character. std:: it is in the standard library namespace.

Essentially all executable code is placed in function and called directly or indirectly from the main global function.

Types, Variables, and Arithmetic:-

Every name and every expression has a type that determines the operations that may be performed on it.

I.e. int inch; //specifies that inch is a int, and that inch is an integer variable

A declaration - is a statement that introduces a name to the program. It specifies a type for the named entity.

* A type defined a set of possible valued and a set of operations (for an object)
* An object is some memory that holds a value of some type
* A value is a set of bits interpreted according to a type
* A variable is a named object

Some of C++’s built in types are: bool, char, int, float, double; each fundamental type has a fixed size that determines the range of values that can be stored in them (for integers) or the precision and the range of these values (for floating point). A char is of natural size to hold a character on a given machine (typically 8 bit bytes), and the size of the other inbuilt types are quoted as multiples of the size of char. The implementation dependent and can be obtained by the use of the sizeof operator.

The arithmetic operator can be used in appropriate combination for these types; x+y; // plus, +x //unary plus, x-y; // minus, -x //unary minus, x\*y // multiply, x/y //divide, x%y //modulus (remainer) for ints. As well as the comparison operators; x==y //equals, x!=y //not equals, x<y //less than, x<=y //less than or equal, x>y //greater than, x =>y //greater than or equal.

In assignment and arithmetic operations, C++ performs any meaningful conversions between builtin types so they can be mixed freely.

I.e.

double d = 2.2; //init double

int i = 7; //init int

d = d+ i; //assign sum to d

i = d\*i; //assign product to i, truncating the double to an int

There are an varieties of ways to the notation for expressing initialization, such as the = form that is universal form based on on a curly brace delimited initializers list;

I.e. double d1 = 2.2; //init double

double d2{2.2}; //init double

complex<double> z = 2.2; //init complex double

complex<double> z = {d1,d2}; //= is optional

complex<double> z{d1,d2};

The = form is traditional and dates back to C, if in doubt use {} - list form, If nothing else it prevents conversion and loss of data.

I.e. int i1 = 7.2; //truncates to 7

int i1{7.2} //does not compile not an int

A constant can’t be left uninitialised and rarely should an variable be left uninitialised; as a rule of thumb introduce the name when you have a reasonable value available for it. User Defined Types (such as strings, vectors, Matrix, Orcs), can be defined to be implicitly initialized. When defining a variable, there is no need to state it’s type explicitly, when it can be deduced from the initialiser.

I.e. auto b = true; // bool

With the auto keyword, we use the = syntax as no type conversions involved might cause a problem. We use auto where we don't have a specific reason to mention the type explicitly.

For example:

* The definition is in a large scope, type visibility for the reader
* We want to be explicit about a variables range or precision i.e. a long int, not an int

Auto is used to reduce the redundancy of long variable names, typically found in generic programming. In addition to conventional arithmetic and logical operators, C++ offers a range of more specific to operations for modifying variables such as; x+=y; //plusequal, ++x; prescript, x++; //postscript

Constants:

C++ supports two types of immutability; const refers to a promise not to change this value. This is used primarily to specify interfaces, so that the data can be passed to functions without fear of it being modified; the promise is enforced by the compiler. Constexpr, refers to be evaluated at compile time. This is used primarily to specify constants, to allow placement of data in memory where it is unlikely to be corrupted and typically for performance.

I.e. double sum(const vector&); // sum will not modify its argument

const int dmv = 17; // dmv is a named constant

constexpr double max1 = 1.4∗square(dmv); // OK if square(17) is a constant expression const double max2 = 1.4∗square(dmv); // OK, may be evaluated at run time

vector v { 1.2, 3.4, 4.5 }; // v is not a constant const double s1 = sum(v); // OK: evaluated at run time

constexpr double s2 = sum(v); // error : sum(v) not constant expression

For a function to be useful in a constant expression, that is, an expression that will be evaluated by the compiler, it must be defined as constexpr;

I.e. constexpr double square(double x) { return x; }

To be a constexpr a function must be rather simple and just return a statement computing a value. A constexpr can be used with non constant arguments, but when that is done the result is not a constant expression. We allow constexpr function to be called with non constant expression arguments in the context we do not require constant expressions. This avoids the need to rewrite the same code twice once for the constant expressions and one for the variable.

In a few contexts, constant expressions are required by language rules (such as array bounds, case labels, and some templates). Other times the compile time evaluation is of importance for performance reasons.

Tests and Loops:

C++ provides the conventional sets of statements for expressing selection and looping.

I.e. //…

cout << “do you want to proceed (y or n)?\n”;

char answer = o;

cin >> answer;

If (answer == y) return true;

return false;

To match the << operator we have the >> (put too) operator, it is utilising the standard input stream. The type of the right hand of the >> operand, determines what input is accepted and it’s right hand operand is the target of the input. Sometimes the if statement is better replaced with an, switch statement

I.e. //…

switch (answer){

case ‘y’: return true;

case ‘n’: return false;

default: return false;

}

A switch statement test as value against a set of constants. The case constants must be distinct, and if no match the default case is selected. The programmer doesn't have to define a default, if no action is taken if the value doesn’t match a case.

Programs are rarely written without loops, a while would be added to give the user several tries

I.e. int tries = 0;

char answer = o;

while(tries < 4){

cout << “do you want to proceed(y or n)?\n”;

cin >> answer;

//…

default: trie++;

}

Pointers, arrays and loops:

An array of elements of the type char, can be defined like so:

I.e. char v[6];

Similarly a pointer like so:

I.e. char\* p;

In declarations [] means the array of and \* means the pointer of; All array’s have 0 as their lower bounds, so v has 6 elements 0-5. The size of the array must be an constantexpression.

A pointer variable holds the address of an appropriate object:

I.e. char \*p = &v[0]; //assign memory of v[0] to p

char x = \*p; //deference p

In an expression \* means the contents of or is called the dereferencing operator.

For loops:

A raw for loop, doesn’t account for array bounds must be done by the programmer and has a variation of uses and notations:

I.e. for (auto i = 0; i < 10; ++i)

for (; i < 10; ++i)

for (; ;)

There is a simpler and more versatile variation however as of C++ 11, the ranged for loop has been a thing.

I.e. for (auto x : v) std::cout << x << “\n”;

The range for loop can be read as for every element of v, from the first to the last place a copy in x and print it. It can be used for any sequence of elements, and if not wanting it to copy the values from v into variable x, and instead refer to them it can be altered.

I.e. for (auto& x : v) std::cout << x << “\n”;

The & symbol is a reference in a declaration, where a reference is similar to a pointer but with no need to dereference it. When used in declarations, operators (such as \*, and []), are called declarator operators. A pointer must always be pointing to something, to avoid dangling pointers and unexpected behaviors. When not pointing to something it should be assigned a nullptr, there is only one nullptr shared by all pointer types. It is often wise to check the pointer points to something like so:

I.e. if (p == nullptr) return false;

It is possible to point the element to the next element with ++ and the previous if there is one with --;

User defined types:

We call the types that can be built out of the fundamental types, the const modifier and the declarator operator are all defined as builtin types. They efficiently and directly reflect the capabilities of the hardware. They don’t provide programmers with high level facilities to conveniently write advanced applications. Instead C++ augments built in types and operations with a sophisticated set of abstraction mechanisms out of which they build such high level facilities.

The abstraction mechanisms are primarily designed to let programmers to implement their own types. Types built from built in types and abstraction mechanisms are called user defined types. They are referred to as classes and enumerations.

Structs -

The first step in building a new type is often to organised the elements if needed into a data structure or struct

i.e. struct Vector{

Int i\_size; //num elements

double \* elem; //pointer to elements

};

Adding an method to construct this would like something like:

I.e. void vector\_init(vector& v, int s){

v.elem = new double[s];

v.i\_size = s;

}

Vector is passed by reference to allow modification, rather than passed by value which would make a copy and only change the copy. The new operator allocates memory on the heap, and is referred to as dynamic memory.

I.e. . //accesses through reference

-> //accesses through pointer

An array is passed by pointer and isn’t passed by value unlike other built in types, as it sends then memory address to the first element of the array.

Classes:

Having data specified separately from the operations on it has advantages, such as the ability to use the data in arbitrary ways. But to simulate a real type, it has to have tighter connection between representation and the operations needed for the user defined type. Typically representation is inaccessible to the user, so it's easier to use, guarantees consistency of the data and allows us to later improve representation.

To do that we have to distinguish between the interface to a type( to be used by all) and it’s implementation (which has access to the otherwise inaccessible data to the user). The language mechanism for that is a class. It is defined as a set of members which can be data, function, or type members. The interface is designed with public members of the class and private members of the class are accessible only through that interface.

I.e. class Vector {

public:

Vector(int s) :elem{new double[s]}, sz{s} { } // construct a Vector

double& operator[](int i) { return elem[i]; } // element access: subscripting

int size() { return sz; }

private:

double∗ elem; // pointer to the elements int sz; // the number of elements };

}

With a variable amount of data, the object is always the same but the size of the vector could differ from vector to vector. This is the basic technique involved in free store allocation (such as the free store allocated by new). The private elements are inaccessible except through public methods.

A function with the same name as it’s class is called a constructor, so the vector\_init can now be replaced with a constructor. The constructor describes how it needs to be constructed, in the above case it needs an int parameter. There is no error handling employed here which is bad, and no give back mechanism for the array; this would leave the memory inaccessible a destructor is one way around this.

Enumerations:

In addition to classes, C++ supports a simple form of user defined types for which we can communicate the values.

I.e. enum class Color{red,green,blue};

enum class TrafficLight{green,yellow,red};

Note that enumerators(i.e. red) are in scope of their enum class, so that they can be used repeatedly in different enum classes without confusion;

I.e. Color’s red isn’t the same as TrafficLights red

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error prone then it would be if the symbolic had been used over enumerators. It's a user defined type so operators can be defined for it.

I.e. TrafficLights operator++(TrafficLight& t){

switch(t){

case TrafficLights::Green: return t = TrafficLights::Green;

case TrafficLights::Yellow: return t = TrafficLights::Yellow;

case TrafficLights::Red: return t = TrafficLights::Red;

}

}

TrafficLights next = ++light;

By default it only has assignment =, initialiser ==, and comparisons i.e. <, defined for it. The class after enum specifies that an enumerator is strongly typed and that the enums are in scope. Being different types it presents misuse of constants.

I.e. Color x = red; //not specified which red

Color y = TrafficLights::Red; //wrong red

Color z = Color::Red; //ok

Similarly we cannot implicitly mix Color and int

int i = Color::Red; //not an int

Color c = 7; //not an color

To allow this, remove the class in front of the enum and it will be what's called a plain enum.

Modularity-

A C++ program consists of many separately developed parts, such as, functions, user defined types, class hierarchies and templates. The key to managing this is to clearly define the interactions among these parts. The first and most important distinction is between the interface to a part and its implementation. At the language levels, C++ represents interfaces by declarations. A declaration specifies all that's needed to use a function or a type.

I.e. double sqrt(dbl);

Class A {

public:

A(){}

private:

Int ia;

};

The point is their declarations are elsewhere, the function bodies specifically or potential too.

Separate compilation:

C++ supports the notion of this where a user sees only declarations of types and functions used. The definitions of these types and functions are in separate source files and compiled separately. This can be used to organise the program into a set of semi independent fragments. The declaration placed in the header file i.e. Vector.h and the source and access granted to the header given to the source file Vector.cpp.

To ensure consistency the source includes the header. It allows the code to be compiled separately. It isn't a language issue but how best to take advantage of a particular language. The best approach is to maximise modularity, represent modularity logically through the language features and exploit the modularity physically through the files for effective separate compilation.

Namespaces:

Are a mechanism along with functions and classes for expressing that some declarations belong together and that their names shouldn’t clash with other names. By putting code into namespaces, it avoids conflicts with other names and it is used by referencing the library in front: Std:min or using a using directive: using namespace std; But this includes the whole namespace rather than the required elements. It is primarily used to organise larger components such as libraries, simplifying composition of a program of separate components.

Error Handling:

C++ provides a few features to help, the major tool being the type system itself. Avoiding building applications from built in types and statements, we build more types that are appropriate for our application and algorithms. Simplifying our programming and limiting opportunities for mistakes and increasing the compilers chances to catch such errors. C++ is designed for elegant and efficient abstraction. One issue of this modularity and abstraction (in particular libraries) is the point where runtimes errors are detected is seperate from where it can be handled.

The importance of error handling is prevalent especially as the program grows in size and scope.

Exceptions:

Are a type of error handling, for example a vector operator[]{} can detect out of range attempt and throw an out\_of\_range exception:

I.e. double& Vector::operator[](int){

if (i < 0 || or size <= i) throw out\_of\_range{ Vector::operator[]}

Return elem[i]; }

The throw transfer control to a handler for exceptions, out\_of\_range is utilised in some functions that directly or indirectly. To do that it will unwind the function call stack as needed to get back to the context of the caller.

I.e. try{

v[v.size()] = 7; //try to access beyond the range

}

catch(out\_of\_range) {

//handle error

}

Try blocks contain code we are interested in handling exceptions for, here the assignment will fail. Therefore the catch clause out\_of\_range will be entered, this is where basic cleanup would be enacted and a rethrow of the exception would take place. The use of exception handling makes error handling simpler and more readable and systematic.

Invariant:

The use of exceptions for out\_of\_range, is a type of function checking it’s argument and refusing to operate because of a precondition that didn’t hold. Programmers should consider a function's preconditions and if feasible test them. However, operator[]() operates on objects of type vector and nothing it doesn’t make sense unless the members of vector does not contain reasonable data within them. For instance we said that elem points to an array of sizeof x and but only in the comments.

Where such a statement of what is assumed to be true for a class, this is called a class invariant or simply an invariant. It is a logical condition to ensure the correct working of the class, it defines the valid states for an object. It must hold when an object is created and must be preserved under all operations of the class.

It is the job of the constructor to establish the invariant for it’s class(So that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit. The earlier example of vector did this partially, by properly initializing the vector but failing to check for an invalid argument.

I.e. Vector v(-27);

A statement like above is likely to cause chaos, so:

Vector::Vector(int s){

if(s<0 throw length\_error{};

Elem = new double[s];

Size = s;

}

Using standard libraries length\_error to report a non positive number of elements, this is commonly used with the stl, so an example:

Void test(){

Vector v(-27);

}

catch(std::length\_error){

//negative number

}

catch(std::bad\_alloc){

//bad memory allocation

}

Classes can be designed to utilise them to carry arbitrary information from a point where an error is detected to a point where it can be handled. Often a function has no way to complete it’s assigned test after an exception is throw. So the term handling again refers to doing minimal local clear and rethrowing the exception.

The notion of invariants is central to the design of classes and exceptions serve a similar role in the design of functions. They help us understand precisely what we want and forces us to be specific. It underlies C++ notions of resource management supported by constructors and deconstructors.

Static Assertions:

Exception report runtime errors, but if an error can be found at compile time then that’s preferable. We can do simple checks on other properties that are known at compile time and report failures that are sent to the compiler messages.

I.e. static\_assert( 4<= sizeof(int), “integers are too small”);

This will write integers that are too small if an int is less than 4 bytes in size, meaning the 4 <= sizeof(int) precondition doesn’t hold; this is an example of an static assertion. The static\_assert mechanism can be used for anything that can be expressed in terms of an constant expression:

I.e. constexpr double c = 2.99792.458;

void f(double speed){

const double localmax = 160\*60\*60;

static\_assert(speed < c, “can’t go that fast”); //won’t work not an constexpr\*

static\_assert(speed < localMax, “can’t go that fast”); //will work is an constexpr\*\*

\*speed has to be constant

\*\*both constant forming an constant expression so all good

In general static\_assert(T, S), T is the condition and S is what is printed if the condition isn’t met, it’s most important use is as parameter checking in generic programming.

Chapter 2 Abstract Mechanisms:

Introduction:

This chapter gives an idea of C++’s support for abstraction and resource management without going into extensive detail. It informally presents ways to define and use user defined types. Presenting its basic properties, and implementation techniques and language facilities used for concrete classes, abstract classes and class hierarchies. It introduces templates as a mechanism for parameterizing types and algorithms with (other) types and algorithms. Computation of user defined and built in types are represented as functions, sometimes generalized to template functions and function objects. These are the language facilities supporting the programming style known as object orientated and generic programming.

Classes:

The central language feature of c++ is a class. Which is a user defined type provided to represent a concept in the code of a program. Whenever our design for a program has a useful concept, idea, entity, etc., we try to represent it as a classe in the program so that the idea is there in code rather than in our heads, in a design document or in some comments. Programs built out of a well designed set of classes are far easier to understand and get right then on build solely from inbuilt types.

Essentially all language facilities beyond the fundamental types, operators and statements exist to help define better classes or use them more conveniently. By better meaning more elegant and efficient, easier to use, maintain, read and reason about. Most programming techniques rely on the design and the implementation of specific kinds of classes. With needs and tastes varying immensely, consequently the support for classes is extensive. Here the basic support for:

* Concrete classes
* Abstract classes
* Classes in hierarchies

Will be discussed here as most classes will be one of the above types, or a combination of several techniques from each in makeup.

Concrete Classes:

The basic idea is that it acts like a built in type. The defining characteristic of a concrete type is that its representation is part of its definition. That allows an implementation to be optimally efficient in time and space. In particular it allows us to:

* Place objects of concrete types on the stack, in statically allocated memory, and in other objects
* Refer to objects directly(not just through pointers and references)
* Initialise object immediately and completely using constructors
* Copy and move objects with constructors

This representation can be private(as it is for the vector) and accessible only through member functions but it is present. Therefore if the representation changes in any significant way the user must recompile. This is the price of having concrete types act like builtin types. For types that don’t change often and where local variables provide much needed clarity and efficiency, this is often acceptable even ideal. To increase flexibility a concrete type can keep major parts of its representation on the free store(heap) and access them through the part stored in the class object itself.

This is the way vector and string are implemented in the stl; they can be considered resource handles with carefully crafted interfaces.

An arithmetic type:

A classical user defined type is a complex number, below is a simplified version of the stl’s variation:

class complex {

double re, im; // representation: two doubles

public: complex(double r, double i) :re{r}, im{i} {} // complex from two scalars

complex(double r) :re{r}, im{0} {} // complex from one scalar

complex() :re{0}, im{0} {} // default complex: {0,0}

double real() const { return re; }

void real(double d) { re=d; }

double imag() const { return im; }

void imag(double d) { im=i; }

complex operator+=(complex z) { return {re+=z.re, im+=z.im}; }

complex operator−=(complex z) { return {re−=z.re, im−=z.im}; }

complex operator∗=(complex); // defined out-of-class somewhere complex operator/=(complex); // defined out-of-class somewhere

};

The above class only shows operations requiring access to the representation. Complex has a simple and conventional representation and a lot of conventional operators. In addition to the logical demand, complex must also be efficient or it will remain unused. This implies that simple operation must be inlined. That is simple operations (such as constructors, +, imag()), must be implemented without function calls in the generated machine code. Functions defined in a class are inlined by default.

An industrial strength complex such as found in the stl, would be carefully implemented to do the appropriate inlining. A constructor can be evoked without an argument and this type of constructor is called the default constructor. As in Complex() is the class complex’s default constructor. By defining a default constructor it eliminates the possibility of uninitialised variables of that type. Note the const specifier on the right hand side on the return real and imaginary parts, used to ensure that a function may not modify the object from which it was evoked.

Other useful operators can be defined separately from the class definition if so desired. It can be used like so:

Complex a{2.3};

Complex b{1/a};

Complex c{a+2\*Complex{1.2,3}};

//..

if (c != b) c = (b/a) + 2 \* b;

The compiler converts operators involving complex number into the appropriate function calls; i.e. c ! b == operator!(c,b). User defined operators or operator overloading should be used cautiously and conventionally. The syntax is fixed by the language, so you can’t define a / unary. Also it is not possible to change the builtin types operators.

A Container:

Is an object holding a collection of elements, so we call a type like a vector, a container because it is a type of container object. As defined earlier our vector isn’t an unreasonable container of doubles or data; it is simple to understand, establishes a useful invariant, provides range checked access and provides size() to allow us to iterate over elements. However it has a major flaw, it allocates elements using new, but never deallocates them. This is a bad idea C++ does define an interface for garbage collection, but it is not guaranteed that one is available or will run to make unused memory available for new objects.

In fact in some environments you can’t use a garbage collector and sometimes you prefer more detailed control of construction and deconstruction for logical reasons. Needing a mechanism to ensure that the memory allocated by the constructor is deallocated, is the job of the mechanism called the destructor.

class Vector {

private:

double∗ elem; // elem points to an array of sz doubles

int sz;

public:

Vector(int s) :elem{new double[s]}, sz{s} // constructor: acquire resources{

for (int i=0;i<s; ++i) elem[i]=0; // initialize elements }

˜Vector() { delete[] elem; } // destructor: release resources

double& operator[](int i);

int size() const;

};

The name of a destructor is the complement operator ~, followed by the name of the class; it is the complement to the constructor. The constructor allocates some memory on the free store(heap or dynamic store) using the new operator. This is all without the user intervention by the user of vector. The use of vectors to create and use vectors much as they would variables of built in types.

void fct(int x) {

Vector v(x); // use v

{

Vector v2(2∗x); // use v and v2

} // v2 is destroyed here // use v

} // v is destroyed here

Obeying the same rules for naming, scoping, allocation, lifetime and so on as does a built in type. The constructor/destructor combination is the basis of many elegant techniques and is in particular use for most c++ general resource management techniques. The constructor allocates the elements and initializes the vector members appropriately. The destructor deallocates the elements. This handle to data model is very commonly used to manage data that can vary in size during its lifetime of an object.

The technique of acquiring resources in the constructor and releasing them in the destructor is known as Resource Acquisition Is Initialism or RAII, it allows us to eliminate naked new operations to avoid allocations in general code and keep them buried inside the implementation of well behaved abstractions. The same with naked delete operations both should be avoided, it makes for far less error prone code and for easier to keep resources free of memory leaks.

Initialising containers:

A container exists to hold elements so obviously we need convenient ways to get elements into a container. We can hald this like previously done within the vector, but typically more elegant methods are preferred, such as:

* Intialiser lists constructors, initialize with a list of elements
* pushback - add a new element at the end

class Vector {

// ...

Vector(std::initializer\_list); // initialize with a list

// ...

void push\_back(double); // add element at end increasing the size by one

// ...

};

The push\_back() is particularly useful for input of arbitrary numbers of elements.

For example:

Vector read(istream& is) {

Vector v;

for (double d; is>>d;) v.push\_back(d);

return v;

}

The input loop is terminated by the end-of-file character or a formatting error. Until then each member is read and emplaced to the vector at the end, v size is the number of elements read. A for-statement is used here over the while-statement to keep the scope of d limited to the lop. The std::intialiser\_list used to define the initializer constructor is a stl type known to the compiler. When we use {}-list such as {1,2,3,4}, the cimike creates an object of type intialiser\_list to give to the program. So:

Vector v1 = {1,2,3,4,5}; // v1 has 5 elements

Vector v2 = { 1.23, 3.45, 6.7, 8 }; // v2 has 4 elements

Vector’s initializer-list constructor might be defined like this:

Vector::Vector(std::initializer\_list lst) // initialize with a list :elem{new double[lst.size()]}, sz{lst.size()} {

copy(lst.beg in(),lst.end(),elem); // copy from lst into elem

}

Abstract types:

Types such as container/vector are concrete types because their representation is part of their definition; they resemble builtin types. In contrast, abstract types, is a type that completely insulates a user from the implementation details. To do that we must decouple the interface from the representation and give up genuine local variables. Since we don’t know anything about the representation of an abstract type, we must allocate objects on the free store( heap) and access through references or pointers.

So a container as an abstract version of a vector, could look like so:

class Container {

public:

virtual double& operator[](int) = 0; // pure virtual function

virtual int size() const = 0; // const member function

virtual ˜Container() {} // destructor (§3.2.1.2)

};

This class is a pure interface to specific containers defined later. The word virtual, means may be redefined later in the class derived from this one. A class derived from container provides an implementation for the container interface. The curious =0;, syntax says it is a pure virtual; that is a class derived from the container must define the function. Thus instantiating a container isn’t possible, it can only serve as the interface to a class that implements operator[]() and size() functions.

A class with a pure virtual function is known as an abstract class. It can be used like so:

void use(Container& c)

{

const int sz = c.size();

for (int i=0; i<sz; ++i)

cout << c[i] << ’\n’;

}

It uses the interface with complete ignorance of the implementation details. A class that provides the interfaces to a variety of classes is often called a polymorphic type. As is common for abstract classes, it doesn’t have a constructor, as it has nothing to initialise. A virtual destructor, common again within abstract classes as they tend to be manipulated through references or pointers and someone destroying it through a pointer has no idea what resources are owned by its implementation.

Not surprisingly, the implementation could use everything from the class vector:

class Vector\_container : public Container {// Vector\_container implements Container

Vector v;

public:

Vector\_container(int s) : v(s) { } // Vector of s elements

˜Vector\_container() {}

double& operator[](int i) { retur n v[i]; }

int size() const { return v.size(); }

};

The public can be read as derived from or is a subtype of. The above class is said to be derived from the class container, and the container is the base class of vector. Or the superclass(container) and the subclass(vector). The derived class is said to inherit members from its base class, the use of base and derived classes is commonly referred to as inheritance. The members operator[]() and size() are said to be overridden the corresponding members in the base class.

The destructor overrides the base classes destructor, note that the member destructor(~vector()) is implicitly invoked by the classes destructor. For a function like use(containers) to use a container in complete ignorance of the implementation details, some other functions will have to make an object on which it can operate. Since use() doesn’t know about vector\_container it only knows of container interface it will work just as well for different implementations of container.

It knows only the interface defined by the container, so use needs be recompiled if the implementation of vector container changes or a brand new class derived from container is used.

Virtual functions:

Consider again the use of container:

void use(Container& c)

{

const int sz = c.size();

for (int i=0; i<sz; ++i)

cout << c[i] << ’\n’;

}

void h()

{

List\_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

use(lc);

}

void g()

{

Vector\_container vc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

use(vc);

}

How is the call c[i] in use() resolved to the right operator[]()? When use() is called from h(), from a vector container it must call Vector\_Container::operator[](), and when use is called from g() from a list container it must List\_Container::operator[](). To achieve this the resolution is to have a container object contain information to allow it to select the right function to call at run time. The usual implementation technique is to convert the name of a virtual function into an index into a table of pointers to functions.

The table is either called the virtual function table, or the virtual lookup table, or simply by vtbl. Each class with virtual functions has its own vtbl identifying all of its virtual functions. The functions in the vtbl allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. The implementation of the caller needs to only know the location of the pointer to the vtbl in a container and the index used for each virtual function.

This virtual call mechanism can be made almost as efficient as the normal function call mechanism (within 25%). It’s space overhead is one pointer in each object of class with virtual functions, plus one vtbl for each such class.

Class Hierarchies:

The container example is a very simple example of a class hierarchy. Which is a set of classes ordered in a lattice created by derivation (e.g. public). We use class hierarchy to represent concepts that have hierarchical relationships such as a fire engine is a kind of truck which is a kind of vehicle. And a smiley face is a kind of circle which is a kind of shape, huge hierarchies, with hundreds of classes that are both deep and wide are common.

In visual representations arrows represent inheritance relationships and in code it could look like something below:

class Shape {

public:

virtual Point center() const =0; // pure virtual

virtual void move(Point to) =0;

virtual void draw() const = 0; // draw on current "Canvas"

virtual void rotate(int angle) = 0;

virtual ˜Shape() {}

// …

};

This an abstract class, as far as representation goes nothing except the location of the pointer to the vtbl is common for every shape. Given this to write a general function manipulating vector of pointers of shape, could look like so:

void rotate\_all(vector<Shape∗>& v, int angle) // rotate v’s elements by the angle

{

for (auto p : v)

p−>rotate(angle);

}

To define a particular shape, we must say that it is a shape and specify its particular properties including its virtual functions.

class Circle : public Shape {

private:

Point x; // center

int r; // radius

public:

Circle(Point p, int rr); // constructor

Point center() const { retur n x; }

void move(Point to) { x=to; }

void draw() const;

void rotate(int) {} // nice simple algorithm

};

So far for the shape and circle, the example shows nothing new on the earlier example container and vector; but we can take this a step further if we choose.

class Smiley : public Circle { // use the circle as base for a face

private:

vector<Shape∗> eyes; // usually two eyes

Shape∗ mouth;

public:

Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

// …

˜Smiley()

{

delete mouth;

for(auto p : eyes) delete p;

}

void move(Point to);

void draw() const;

void rotate(int);

void add\_eye(Shape∗ s) { eyes.push\_back(s); }

void set\_mouth(Shape∗ s);

virtual void wink(int i); // wink eye number i

};

The push\_back member function add’s its arguments to the vector(here, eyes) increasing the vector by one. We can not define Smiley::draw() using calls ot smiley’s base and member draw():

void Smiley::draw()

{

Circle::draw();

for (auto p : eyes) p−>draw();

mouth−>draw();

}

Note here that the smiley keeps its eyes in a stl vector and deletes them in it’s destructor. Shape's destructor is virtual and smileys destructor overrides it. A virtual destructor is essential for an abstract class because an object of a derived class may be deleted through a pointer to a base class. Then, the virtual function call mechanism ensures that the proper destructor is called. The destructor implicitly invokes the destructor of it’s base and it’s members.

In this simplified example, it is the programmers task to place the eyes and mouth appropriately within the circle representing the face. We can add data members operperations or both as we define new classes by derivation. Giving great flexibility with corresponding opportunities for confusion and poor design. A class hierarchy offers two kinds of benefits:

* Interface inheritance - an object of a derived class can be used whenever an object of a base class is required. That is the base class act as an interface for the derived class. The container/shape example is a classic example of abstract base classes.
* Implementation inheritance - a base class provides functions or data that simplifies the implementation of derived classes. Smiley use of circles constructor and circle draw are examples of this. Such base classes often have data members and constructors.

Concrete classes:

Especially those with small representation, are much like built in types. However with classes in a hierarchie; we tend to allocate them on the free store(heap) using new and access them through references or pointers. I.e. Consider a function to read data describing shapes from an input stream and constructs the appropriate shape object:

enum class Kind { circle, triangle, smiley };

Shape∗ read\_shape(istream& is) // read shape descriptions from input stream is

{

// read shape header from is and find its Kind k

switch (k) {

case Kind::circle:

// read circle data {Point,int} into p and r

return new Circle{p,r};

case Kind::triangle:

// read triangle data {Point,Point,Point} into p1, p2, and p3

return new Triangle{p1,p2,p3};

case Kind::smiley:

// read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1 ,e2, and m

Smiley∗ ps = new Smiley{p,r};

ps−>add\_eye(e1);

ps−>add\_eye(e2);

ps−>set\_mouth(m);

return ps;

}

}

This example is simplified, especially where error handling is concerned, but it demonstrates that the user has no idea which kind of shapes it manipulates. It can be compiled once and used for new shapes added to the program. There's no pointer to the shape outside the user so it must also be released within. Using the delete operator relying on the abstract class destructor because it's virtual it invokes the most derived class’s destructor. This is paramount as the derived class may have cleanup to do.

Two other issues that may arise to a good eyed experience programmer are:

* User may of failed to capture read\_shape return value thus forgetting to delete it
* Owner of the container of shapes may forget to delete the objects pointed to

In that sense functions returning a pointer to an object allocated on the free store(heap) are dangerous! One solution is the usage of smart pointers over raw pointers:

unique\_ptr<Shape> read\_shape(istream& is) // read shape descriptions from input stream is

{

// read shape header from is and find its Kind k

switch (k) {

case Kind::circle:

// read circle data {Point,int} into p and r

return unique\_ptr<Sha pe>{new Circle{p,r}}; // §5.2.1

// …

}

void user()

{

std::vector<unique\_ptr<Shape>> v;

while (cin)

v.push\_back(read\_sha pe(cin));

draw\_all(v); // call draw() for each element

rotate\_all(v,45); // call rorate(45) for each element

} // all Shapes implicitly destroyed

Now the unique\_ptr or smart pointer will deallocate itself when required. For this to work users would need all appropriate functions to accept the vector<unique\_ptr<shape>>. This can be tedious and an alternative will be discussed later.

Copy and move:

By default objects can be copied; this is true for user defined types as well as builtin ones. The default of the meaning of copy is a memberwise copy or copy each member type of copy, shown below:

complex z1 {1,2};

complex z2 {z1}; // copy initialization

complex z3;

z3 = z2; // copy assignment

Z1, z2, and z3 are all the same values as both assignment and initialization is copied for both members. When designing a class consideration should go into how it may be copied. For simple concrete types memberwise is usually sufficient, but for more

complex concrete types such as vectors it's not the right semantic and for abstract classes it rarely is either.

Copying Containers:

When a class is a resource handle, that is it is responsible for an object accessed through a pointer the result memberwise copy can be a disaster. It would violate the resource handlers invariant for a vector it would leave the copy referring to the same elements as the original, like so:

Vector v1(4);

Vector v2 = v1;

v1[0] = 2; // v2[0] is now also 2!

v2[1] = 3; // v1[1] is now also 3!

The destructor in vector is a strong hint that the default copy semantic is wrong and the compiler should have warned against it. A better copy semantic needs to be define, this requires the usage of:

* A copy constructor
* A copy assignment

class Vector {

Private:

// elem points to an array of sz doubles

double∗ elem;

int sz;

Public:

Vector(int s); // constructor: establish invariant, acquire resources

˜Vector() { delete[] elem; } // destructor: release resources

Vector(const Vector& a); // copy constructor

Vector& operator=(const Vector& a); // copy assignment

double& operator[](int i);

const double& operator[](int i) const;

int size() const;

};

A suitable definition of a vector copy for a container to simple copy the elements can then be performed, like so:

Vector::Vector(const Vector& a) // copy constructor

:sz(a.sz)

{

elem = new double[sz];

for (int i=0; i<sz; ++i)

elem[i] = a.elem[i];

}

Of course we need a copy assignment to go with this:

Vector& Vector::operator=(const Vector& a) // copy assignment

{

double∗ p = new double[a.sz];

for (int i=0; i<a.sz; ++i)

p[i] = a.elem[i];

delete[] elem; // delete old elements

elem = p;

sz = a.sz;

return ∗this;

}

The name this is predefined in every member function and points to the object for which the member function is called. A copy constructor and copy assignment for a class x, are typically declared to take an argument of type const x&.

Moving containers:

We can define copying but it can be expensive for things like large containers, consider the following bit of code:

Vector operator+(const Vector& a, const Vector& b)

{

if (a.size()!=b.size())

throw Vector\_size\_misma tch{};

Vector res(a.size());

for (int i=0; i<a.size(); ++i)

res[i]=a[i]+b[i];

return res;

}

Returning from a + involves copying the result out of the local variable res and into some place where the caller can access it. Using + like so:

void f(const Vector& x, const Vector& y, const Vector& z)

{

Vector r;

// …

r = x+y+z;

// ...

}

That would mean copying a vector at least twice, for each use of the + operator. The copy is never used again after the copy, we didn't want to copy but get the result out of a function. So we wanted to move the vector rather than copy it, fortunately we can state this intention, like so:

class Vector {

Private:

// elem points to an array of sz doubles

double∗ elem;

int sz;

Public:

// …

Vector(const Vector& a); // copy constr uctor

Vector& operator=(const Vector& a); // copy assignment

Vector(Vector&& a); // move constr uctor

Vector& operator=(Vector&& a); // move assignment

// ...

};

Given that the compiler will choose the move operator to implement the transfer of the return value out of the function. This means now the vectors are moved instead of being copied. The move constructor could be defined, like so:

Vector::Vector(Vector&& a)

{

elem = a.elem; // "grab the elements" from a

sz = a.sz;

a.elem = nullptr; // now a has no elements

a.sz = 0;

}

The && means a rvalue reference and is a reference to which we can bind an rvalue. The rvalue is intended to complement the lvalue which roughly translates to that which can appear on the left hand of the assignment. So an rvalue is to a first approximation - a value that you can’t assign to, such as an integer returned by a function call and a rvalue reference is a reference to something that nobody else can assign to it. An *lvalue* (*locator value*) represents an object that occupies some identifiable location in memory (i.e. has an address).

*rvalues* are defined by exclusion, by saying that every expression is either an *lvalue* or an *rvalue*. Therefore, from the above definition of *lvalue*, an *rvalue* is an expression that *does not* represent an object occupying some identifiable location in memory.

A move constructor doesn’t take a const argument, as move constructor is supposed to remove the value from it’s argument.

A move assignment is defined similarly, a move operation is used when a rvalue reference is used as an initialiser or as the right hand of an assignment. After a move an object should be in a state where a destructor is allowed to be run. Typically we should also allow assignment to a moved from object. In cases where the programmer knows that a value will not be used again, but can’t expect the compiler to be smart enough a programmer can be specific, like so:

Vector f()

{

Vector x(1000);

Vector y(1000);

Vector z(1000);

// …

z = x; // we get a copy

y = std::move(x); // we get a move

// …

return z; // we get a move

};

This stl move returns an rvalue reference to it’s argument, we use it just before the return. By the time the z is destroyed, it too has been moved from(by the return) so that like x it holds no elements.

Resource management:

By defining constructors, copy, move and destructors, we can provide control over the lifetime of a contained resource (such as an element of a container). In particular, move allows objects to move simply and cheaply from one scope to another. That way we can move objects that we cannot or would not want to copy out of scope. Consider a stl thread representing a concurrent activity and a vector of a million doubles. We can’t copy the former and don’t want to copy the latter:

std::vector<thread> my\_threads;

Vector init()

{

thread t {heartbeat}; // run heartbeat concurrently (on its own thread)

my\_threads .push\_back(move(t)); // move t into my\_threads

Vector<double> vec;

// ... fill vec …

return vec; // move res out of run()

}

auto v = init(); // start heartbeat and initialize v

This makes resource handlers, such as vectors and thread an alternative to pointers in many cases. In fact stl smart pointers such as unique\_ptr are themselves resource handlers.

Preventing copy and move

Using the default copy or move for a class in a hierarchy is typically disastrous. Given only a pointer to a base, we simply don’t know what members the derived classes have so we can’t copy them. So it is best to delete the default copy and move operations, so as to eliminate their default definitions, like so:

class Shape {

Public:

Shape(const Shape&) =delete; // no copy operations

Shape& operator=(const Shape&) =delete;

Shape(Shape&&) =delete; // no move operations

Shape& operator=(Shape&&) =delete;

˜Sha pe();

// …

}

Now any attempt to copy will be caught by the compiler. If a need for copy in a class hierarchy, it is preferred to use a clone function of some kind. In case move or copy are accidentally left in, there is no harm done. As a move isn’t implicitly created where the user has declared a destructor. This can be a good reason to explicitly define a destructor where the compiler would have done implicitly anyway. A base class is one example of a class where we wouldn’t want to copy. A resource handle can’t be copied just by copying its members generally, these are things you’d do well to remember.

Templates:

A template is a class or a function that we parametrize with a set of types of values. We use templates to represent concepts that are best understood as something very general from which we can generate specific types and functions by specifying arguments, such as the type of a double.

Parameterized types:

We can generate our vector of doubles type to a vector of anything type by making it into a template and replacing specific type of double with a parameter, like so:

template<typename T>

class Vector {

Private:

T∗ elem; // elem points to an array of sz elements of type T

int sz;

Public:

Vector(int s); // constructor: establish invariant, acquire resources

˜Vector() { delete[] elem; } // destructor: release resources

// copy and move operations

T& operator[](int i);

const T& operator[](int i) const;

int size() const { return sz; }

}

The template<typename t> prefix makes T as a parameter of the declaration it prefixes. It is C++’s version of a mathematical for T or more precisely for all types T. A member function could look, like so:

template<typename T>

Vector<T>::Vector(int s)

{

if (s<0) throw Neg ative\_size{};

elem = new T[s];

sz = s;

}

template<typename T>

const T& Vector<T>::opera tor[](int i) const

{

if (i<0 || size()<=i) throw out\_of\_range{"Vector::operator[]"};

return elem[i];

}

Given this vector can be define like so:

Vector<char> vc(200); // vector of 200 character s

Vector<string> vs(17); // vector of 17 integers

Vector<list<int>> vli(45); // vector of 45 lists of integers

The .. in the vector<list>> terminates the nested template arguments; it is not a misplaced input operator. It is not necessary to place a space between the two >s.

The we can use vector, like so:

void f(const Vector<string>& vs) // Vector of some strings

{

for (int i = 0; i<vs.size(); ++i))

cout << vs[i] << ’\n’;

}

If we also want to use the range-for loops for our vector, we must define suitable begin() and end(), like so:

template<typename T>

T∗ beg in(Vector<T>& x)

{

return &x[0]; // pointer to first element

}

template<typename T>

T∗ end(Vector<T>& x)

{

return x.beg in()+x.size(); // pointer to one-past-last element

}

Given those, we can write:

void f2(const Vector<string>& vs) // Vector of some strings

{

for (auto s : vs)

cout << s << ’\n’;

}

Similarly, we can define other associative arrays, etc., as templates, they are a compile time mechanism, so their use incurs no runtime overhead in comparison to hand written code.

Function templates:

Templates have many more uses than simply parameterizing a container with an element type. In particular they are extensively used for parameterization of both type and algorithms in the stl, for example we can write a function that calculates the sum of the element values of any container, like so:

template<typename Container, typename Value>

Value sum(const Container& c, Value v)

{

for (auto x : c) v+=x;

return v;

}

The value template argument and the function argument v are there to allow the caller to specify the type and initial value of the accumulator(the variable in which to accumulate the sum):

void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)

{

int x = sum(vi,0); // the sum of a vector (add ints)

double d = sum(vi,0.0); // the sum of a vector (add doubles)

double dd = sum(ld,0.0); // the sum of a list of doubles

auto z = sum(vc,complex<double>{}); // the sum of a vector of complex<double>

}

The point of adding ints in a double would be to graceful handle a number larger than the largest int. Note how the type of the template argument for sum<T,V> are deduced from the function arguments. This sum() is a simplified version of the stl’s accumulate().

Function objects ( functors)

A useful type of template function is a function object sometimes known as a functor, for example:

template<typename T>

class Less\_than {

const T& val; // value to compare against

public:

Less\_than(const T& v) :val(v) { }

bool operator()(const T& x) const { return x<val; } // call operator

};

The function call operator implements the function call or application operator (). We can define named variables of type lessthan for argument types, like so:

Less\_than<int> lti {42}; // will compare to 42 (using <)

Less\_than<str ing> lts {"Backus"}; // will compare to "Backus" (using <)

We call them such objects like a function, like so:

void fct(int n, const string & s)

{

bool b1 = lti(n); // true if n<42

bool b2 = lts(s); // true if s<"Backus"

// ...

}

They are widely used as arguments to algorithms for example, like so:

template<typename C, typename P>

int count(const C& c, P pred)

{

int cnt = 0;

for (const auto& x : c)

if (pred(x)) ++cnt;

return cnt;

}

A predicate is something that we can invoke to true or false, for example:

void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)

{

cout << "number of values less than " << x

<< ": " << count(vec,Less\_than<int>{x})

<< ’\n’;

cout << "number of values less than " << s

<< ": " << count(lst,Less\_than<string>{s})

<< ’\n’;

}

There, the lessthan<int>{x} constructs an object for which the call operator compares to the int called x; and similar for the string too. The beauty of functors is they carry the value to be compared against within them. No need to write a separate function for each value(and type) and no need for global variables to hold values. Also, for example a simple function object like lessthan inlining as simple as that call to the function is more efficient than the indirect function call. The ability to carry data plus their efficiency makes functors particularly useful as an argument to an algorithm.

Functors that are used to specify the meaning of a key operation of a general algorithm(such as for less than for count()) we often refer to them as policy objects. We have to define lessthan separately from its use this could be seen as inconvenient. Consequently there is notation for implicitly creating functors., like so:

void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)

{

cout << "number of values less than " << x

<< ": " << count(vec,[&](int a){ retur n a<x; })

<< ’\n’;

cout << "number of values less than " << s

<< ": " << count(lst,[&](const string& a){ retur n a<s; })

<< ’\n’;

}

The notation [&](int a){ return a < x; } is called a lambda expression or lambda function. It performs exactly like the lessthan functor. The [&] is a capture list specifying that local names such as x will be passed by reference. Had we wanted to capture only we could have stated this like so [&x]. Had we wanted to give the generated object a copy of x we could have also said so, like so [=x]. To capture nothing would look, like so [], capture all by local reference is [&] and by value is [=]. Using lambdas are more convenient and terse but also obscure.

For non-trivial actions more than a simple expression that is, naming the operations as to clearly state its purpose and for ease of reusability in the program. In writing the operations of the vector of pointer and unique pointers they’re can be many variations required to perform correctly. With functors specifically lambda they can help by allowing us to separate the traversal of the container from the specification of what is to be done with each element. First we need a functor that applies an operation pointed to by a element of a container of pointers, like so:

template<class C, class Oper>

void for\_all(C& c, Oper op) // assume that C is a container of pointers

{

for (auto& x : c) op(∗x); // pass op() a reference to each element pointed to

}

Now we can write a version of user() from without writing a set of \_all functions:

void user()

{

vector<unique\_ptr<Shape>> v;

while (cin)

v.push\_back(read\_sha pe(cin));

for\_all(v,[](Shape& s){ s.draw(); }); // draw\_all()

for\_all(v,[](Shape& s){ s.rota te(45); }); // rotate\_all(45)

}

I pass a reference to Shape to a lambda so that the lambdas don’t have to care exactly how the objects are stored in the container. In particular, those for\_all() calls would still work if I changed v to a vector<Shape∗>.

Variadic templates:

A template can be defined to accept an arbitrary number of arguments, this technique is typically called a variadic template, shown below:

template<typename T, typename... Tail>

void f(T head, Tail... tail)

{

g(head); // do something to head

f(tail...); // try again with tail

}

void f() { } // do nothing

Key to implementing a variadic template is to note that when you pass a list of arguments to it you can separate the first argument or the head from the rest or the tail. Then we perform operations on the first(head) and then recursively call f() with the rest of the arguments(the tail). The ellipses, … , is used to indicate the rest of a list. Eventually the tail will become empty and we need a separate function to deal with this inevitability.

We can call this f() like this:

int main()

{

cout << "first: ";

f(1,2.2,"hello");

cout << "\nsecond: "

f(0.2,’c’,"yuck!",0,1,2);

cout << "\n";

}

This would call f(1,2.2, “hello”), which will call f(2.2, “hello”) which will call f(“hello”), which will call f(). What might g() do? In a real program it will do whatever we wanted done to each argument. For example, we could make it writes its argument to the output, like so:

template<typename T>

void g(T x)

{

cout << x << " ";

}

Given that, the output will be:

First: 1 2.2 hello

Second: 0.2, c yuck! 0 1 2

It seems that f() is a simple variant of printf() printing arbitrary lists or values - implemented in three lines of code plus their surrounding declarations. The strength of variadic templates, sometimes just called variadic is that they can accept any arguments you care to give them. Their weakness is that the type checking of the interface is possibly an elaborate template program.

Aliases:

Surprisingly often, it is useful to introduce a synonym for a type or a template, for example the standard header <cstddef> contains a definition of the alias size\_T, maybe:

using size\_t = unsigned int;

The actual type name size\_t is implementation dependent, so in another implementation size\_t maybe un unsigned long. Having the alisa size\_t allows the programmer to write portable code. It is very common for parameterized type to provide an alias for types related to their template arguments, like so:

template<typename T>

class Vector {

public:

using value\_type = T;

// ...

}

In fact, every stl container provides value\_type as the name of their value type. This allows us to write code that will work for every container that follows these conventions. For example we could have:

template<typename C>

using Element\_type = typename C::value\_type;

template<typename Container>

void algo(Container& c)

{

Vector<Element\_type<Container>> vec; // keep results here

// …

}

The aliasing mechanism can be used to define a new template by binding some or all the templates arguments, for example:

template<typename Ke y, typename Value>

class Map {

// …

};

template<typename Value>

using String\_map = Map<string,Value>;

String\_map<int> m; // m is a Map<string,int>

Chapter 3 Containers and algorithms:

Introduction:-

No significant program is written in just bare programming language, first a set of supporting libraries is developed. These then form the basis for further work, most programs are tedious to write in bare language, whereas just about any task can be rendered simple by the use of good libraries. A brief presentation of the useful standard library types, such as string, ostream, vector, map(this chapter) and unique\_ptr, thred, regex and complex in the ext as well as the most conventional ways of using them.

The standard library facilities described are part of every complete C++ implementation. In addition to the standard C++ library, most implementations offer “graphical user interface” systems (GUIs), web interfaces, database interfaces, etc. Similarly, most application development environments provide foundation libraries for corporate or industrial standard development and/or execution environments. These are not described; the intent is to provide a self contained description of C++ as defined by the standard and to keep the examples portable, except where specifically noted.

Standard-library overview:

The facilities provided by the standard library can be classified like this:

* Basic run time language support (e.g. for allocation and run time type information)
* The c standard library ( with very minor modification to minimize the violations of type system)
* Strings and I/O streams (with support for international character sets and localisation), I/O stream is an extensive framework to which users can add their own streams, buffering strategies and character sets.
* A framework of containers (such as vector, list and map) and algorithms (such as find(), sort(), and merge()). This framework is conventionally called the stl, it is extensible so that users can easily add their own containers and algorithms
* Support for numerical computation (such as standard mathematical functions, complex numbers vectors with arithmetic operations, and random number generators
* Support for regular expression matching
* Support for concurrent processing, including threads and locks; the concurrency support is foundational so that users can add support for new models of concurrency libraries.
* Utilities to support template metaprogramming, stl-style generic programing
* Smart pointer for resources management and an interface to garbage collectors
* Special purpose containers such as array, bitset, and tuple

The main criterion for including a class in the library was that it would somehow be used by almost every C++ programming (novices and experts alike), that it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be easy to learn (relative to the inherent complexity of the task performed). Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

The standard library headers and namespace:

Every standard library facility is provided through sme standard header, for example:

#include <string>

#include <list>

This makes the standard string and list available.

The standards library is defined in the namespace called std, to use standard library facilities, the std:: prefix can be used:

std::string s {"Four legs Good; two legs Baaad!"};

std::list<std::string> slogans {"War is peace", "Freedom is Slavery", "Ignorance is Strength"};

For simplicity, the std:: is rarely used throughout the examples. Neither will their #include necessary for their successful compilation explicitly. T comple and run the program fragments here, you must include the appropriate headers and make the names declared accessible. For example:

#include<string> // make the standard string facilities accessible

using namespace std; // make std names available without std:: prefix

string s {"C++ is a general−purpose programming language"}; // ok: string is std::string

It is generally poor taste to dump every name from a namespace into global space.

Strings:-

The stl provides a string type to complement string literals. The string type provides a variety of useful operations, such as concatenation. For example:

string compose(const string& name, const string& domain)

{

return name + ’@’ + domain;

}

auto addr = compose("dmr","bell−labs.com");

Here addr is initialized to the character sequence [dmr@bell-labs.com](mailto:dmr@bell-labs.com). Addition of strings means concatenation. You can concatenate a string, a string literal and a c style string, or a character to a string. The standard spring has a move constructor so returning even long strings by value is efficient. In many applications, the most common form of concatenation is adding something to the end of a string. This is directly supported by the += operation, like so:

oid m2(string& s1, string& s2)

{

s1 = s1 + ’\n’; // append newline

s2 += ’\n’; // append newline

}

The two methods of adding to the end of a string are semantically equivalent, but the latter is more explicit about what it does and more concise and potentially more efficiently implemented. A string is mutable. In addition to the = and +=, subscripting(using[]) and substring operations are supported. An example of substring manipulating is shown below:

str ing name = "Niels Stroustrup";

void m3()

{

string s = name.substr(6,10); // s = "Stroustr up"

name .replace(0,5,"nicholas"); // name becomes "nicholas Stroustrup"

name[0] = ’N’; // name becomes "Nicholas Stroustrup"

}

The substr() operation returns a string that is a copy of the substring indicated by its argument. The first argument is an index into the string(its position), and the second argument is the length of the desired substring. Since indexing starts from 0, s gets the value Stroustrup. The replace() operation replaces a substring with a value. In this case, the substring starting with - with length 5 is Niels; which is replaced by Nicholas. Thus, the finale value of the name is Nicholas Stroustrup. Note that the replacement string need not be the same size as the substring that it is replacing.

Naturally, strings can be compared against each and against string literals, shown in the example below:

string incantation;

void respond(const string& answer)

{

if (answer == incantation) {

// perform magic

}

else if (answer == "yes") {

// ...

}

// ...

}

Stream I/O

The standard library provides formatted character input and output through the use of the iostream library. The input operations are typed and extensible to handle user defined types. This gives a brief overview of the use of iostream. Other forms of user interactions such as graphical I/O, are handled through libraries that are not part of the ISO standard and therefore are not described here.

Output:

The I/O stream library defines output for every built in type. Further it is easy to define output of a user defined type. The operator << (put to) is used as an output operator on objects of type ostream; cout is the standard output stream and cerr is the stand stream for reporting errors. By default, values written to cout are converted to a sequence of characters. For example, to output the decimal number 10, we can write.

void f()

{

cout << 10;

}

This places the character 1 followed by the character 0 on the standard output stream.

Equivalently, we could write:

void g()

{

int i {10};

cout << i;

}

Output of different types can be combined in the obvious way:

void h(int i)

{

cout << "the value of i is ";

cout << i;

cout << ’\n’;

}

For h(10), the output will be, the value of i is 10. People soon tire of repeating the name of the output stream when outputting several related items. Fortunately, the result of an output expression can itself be used for further output. For example:

void h2(int i)

{

cout << "the value of i is " << i << ’\n’;

}

This h2() produces the same output as h(). A character constant is a character enclosed in single quotes. Note that a character is output as a character rather than as a numerical value. For example:

void k()

{

int b = ’b’; // note: char implicitly converted to int

char c = ’c’;

cout << ’a’ << b << c;

}

The integer value of the character ’b’ is 98 (in the ASCII encoding used on the C++ implementation that I used), so this will output a98c

Input:

The standard library offes istreams for input. Like with ostreams, istreams, deal with character string representations of built in types and can easily be extended to cope with user defined types. The operator >> (get from) is used as an input operator; cin is the standard input stream. The type of the right hand operand of >> determines what input is accepted and what is the target of the input operation, like so:

oid f()

{

int i;

cin >> i; // read an integer into i

double d;

cin >> d; // read a double-precision floating-point number into d

}

This reads a number, such as 1234, from the standard input into the integer variable i and a floating point number such as 12.34e5, into the double precision floating point variable d.

String I/O

Often we want to read a sequence of characters. A convenient way of doing tha tis to read into a string, for example:

int main()

{

string str;

cout << "Please enter your name\n";

cin >> str;

cout << "Hello, " << str << "!\n";

}

If you type in Eric the response is Hello , Eric!

You can read a whole line (including the terminating newline character) using the getline function, like so:

int main()

{

cout << "Please enter your name\n";

string str;

getline(cin,str);

cout << "Hello, " << str << "!\n";

}

With this program, the input Eric Bloodaxe yields the desired output: Hello , Eric Bloodaxe! The newline that terminated the line is discarded, so cin is ready for the next input line. The standard strings have the nice property of expanding to hold what you put in them; there's no need to pre calculate their maximum size. So, if you enter a couple of megabyte os semicolons, the program will echo pages of semicolons back at you.

I/O of user defined types:

In addition to the I/O of built-in types like that of standard strings, the istream library allows programmers to define I/O for their own types. For example, consider a simple type Entry that we might use to represent entries in a telephone book.

struct Entry {

string name;

int number;

};

We can define a simple output operator to write an Entry using a {"name",number} format similar to the one we use for initialization in code:

ostream& operator<<(ostream& os, const Entry& e)

{

return os << "{\"" << e.name << "\", " << e.number << "\"}";

}

A user defined output operator takes its output stream (by reference) as its first argument and returns it as it’s result. The corresponding input operator is more complicated because it has to check for correct formatting and deal with error checking.

istream& operator>>(istream& is, Entry& e)

// read { "name" , number } pair. Note: for matted with { " " , and }

{

char c, c2;

if (is>>c && c==’{’ && is>>c2 && c2==’"’) { // start with a { "

string name; // the default value of a string is the empty string: ""

while (is.get(c) && c!=’"’) // anything before a " is part of the name

name+=c;

if (is>>c && c==’,’) {

int number = 0;

if (is>>number>>c && c==’}’) { // read the number and a }

e = {name,number}; // assign to the entry

return is;} }

}

is .setf(ios\_base::failbit); // register the failure in the stream

return is;

}

An input operation returns a reference to its istream which can be used to test if the operation was successful or not. For example, when used as condition cin>>c means, did we succeed at reading from cin into c? The (is>c) skips whitespace by default but is.get(c) does not so that this Entry-input operator ignores(skips) whitespace outside the name string, but not within it. For example:

{ "John Marwood Cleese" , 123456 }

{"Michael Edward Palin",987654}

We can read such a pair of values from input into an Entry like this:

for (Entry ee; cin>>ee; ) // read from cin into ee

cout << ee << ’\n’; // write ee to cout

Containers:

Much of computing involves the creating collecting og values and then manipulating such collections. Reading characters into a string and printing out the string is a simple example. A class with the main purpose of holding objects is commonly called a container. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program. To help illustrate the stl containers, we will consider a simple program for keeping names and telephone numbers.

This is the kind of program for which different approaches appear simple and obvious to people of different backgrounds. The Entry class from earlier can be used to hold a simple phone book entry. Here, the example will deliberately ignore many real world complexities, such as the fact that many phone numbers do not have a simple representation as a 32 bit int.

Vectors:

The most useful stl container is that of the vector, it is a sequence of elements of a given type. These elements are then stored contiguously in memory, we can initialize a vector with a set of values of its element type:

vector<Entry> phone\_book = {

{"David Hume",123456},

{"Karl Popper",234567},

{"Bertrand Arthur William Russell",345678}

};

Elements can be accessed through subscripting:

void print\_book(vector<Entry>& book)

{

f or (int i = 0; i!=book.size(); ++i)

cout << book[i] << ’\n’;

}

As usual, indexing starts at 0 so that book[0] holds the entry for David Hume. The vector

member function size() gives the number of elements. The elements of a vector (obviously) constitute a range, so we can use the simpler

range-for loop:

void print\_book(vector<Entry>& book)

{

for (const auto& x : book) // for "auto" see §2.2.2

cout << x << ’\n’;

}

When we define a vector, we give it an initial size (initial number of elements):

vector<int> v1 = {1, 2, 3, 4 }; // size is 4

vector<string> v2; // size is 0;

vector<Shape∗> v3(23); // size is 23; initial element value: nullptr

vector<double> v4(32,9.9); // size is 32; initial element value: 9.9

An explicit size is enclosed in ordinary parentheses, e.g. (23), and by default the elements are initialised to the element type’s default value(e.g. Nullptr for pointer and 0 for numbers). If you don’t want the default value, you can specify one as a second argument 9.9 for the 32 elements of v4. The initial size can be changed, one of the most useful operations of the vector is it’s push\_back(), which adds a new lament at the end of a vector, increasing its size by 1, for example:

for(Entry e; cin>>e)

phone\_book.push\_back(e);

This reads Entrys front the standard input into the phobe\_book until either the end of input(end of file) is reached or the input operation encounters a format error. The standard library vector is implemented so that growing a vector by repeated push\_back is efficient. A vector is a single object that can be assigned to, for example:

Void f(vector<Entry>& v){

vector<Entry> v2 = phone\_book;

v = v2

//…

}

Assigning a vector involves copying its elements. Thus, after the initialization and the assignment in f(), v and v2 each holds a separate copy of every entry in the phone book. When a vector holds many elements, such an innocent looking assignment and initialisation can be prohibitively expensive. Where copying is undesirable, references or pointers or move operation should be utilised.

Elements

Like all stl containers, vector is a container of elements of some type T; that is a vector<T>. Just about any type qualifies as an element type: builtin numeric types(such as char, int, and double), user defined types(such as string, Entry, list<int>, and Matrix<double,2>) and pointers (sch as const char\*,sshape\*, double\*). When you insert a new element, its value is copied into the container.

For example, when you put an integer with the value 7 into a container, the resulting element really has the value 7. The element is not a reference to a pointer to some object containing 7. This makes for nice compact containers with fast access. This is beneficial to those who care about memory sizes and run time performance.

Range checking:

The stl vector does not guarantee range checking, for example:

vector<Entry> phone\_book(1000);

int i = phone\_book[2001].number; // 2001 is out of range

That initialisation is likely to place some random value in i rather than give an error. This is undesirable and out of range errors are commonplace problems. Consequently, I often use a simple range-checking adaption of vector, like so:

template<typename T>

class Vec : public std::vector<T> {

pub lic:

using vector<T>::vector; // use the constructors from vector

// (under the name Vec);

T& operator[](int i) { return vector<T>::at(i); } // range-checked

const T& operator[](int i) const { return vector<T>::at(i); } // range-checked

// for const objects;

};

The above vec inherits everything from the vector except the subscript operations that it redefines to do some range checking operations. The at() operation is a vector subscript operation that throws an exception of the type out\_of\_range if its arguments is out of the vectors range. An out\_of\_range access will throw and exception that the user can catch, like so:

void f(Vec<Entry>& book)

{

try {

book[book.siz e()] = {"Joe",999999}; // will throw an exception

}

catch (out\_of\_range) {

cout << "range error\n";

} }

The exception will be thrown and then caught, if the user does not catch an exception

the program will terminate in a well defined manner rathan than preceding or failing in an undefined manner. One way to minimize surprise from uncaught exceptions to use a main() a try-block as its body.

int main()

try {

// your code

}

catch (out\_of\_range) {

cerr << "range error\n";

}

catch (...) {

cerr << "unknown exception thrown\n";

}

This provides default exception handlers so that if we fail to catch some exception, an error message is printed on the standard error diagnostic output stream cerr. Some implementations save you the bother of defining vec (or equivalent) by providing a range checked version of vector(e.g. As a compiler option).

List:

The standard library offers a doubly linked list called list; We use a list for sequences where we want to insert and delete elements without moving other elements. Insertion and deletion of the phone book entries could be common, so a list could be appropriate for representing a simple phone book. For example:

list<Entry> phone\_book = {

{"David Hume",123456},

{"Karl Popper",234567},

{"Bertrand Arthur William Russell",345678}

};

When we use a linked list, we tend not to access elements using subscript the way we commonly do for vectors. Instead, we might search the list looking for an element with a given non-zero value. To do this we take advantage of the list is a sequence as described:

int get\_number(const string& s)

{ for (const auto& x : phone\_book)

if (x.name==s)

return x.number;

return 0; // use 0 to represent "number not found"

}

The search starts at the beginning of the list and processes until either it has found s or the end is reached. Sometimes we need to identify an element in a list. For example, we may want to delete it or insert a new entry before it. To do that we use an iterator; a list iterator identifies an element of a list and can be used to iterate through a list. Every standard library container provides the functions begin() and end(), which returns an iterator to the first and to one past the last element , respectively. Using iterators explicitly, we can less elegantly - write the get\_number() function, like so:

int get\_number(const string& s)

{

for (auto p = phone\_book.begin(); p!=phone\_book.end(); ++p)

if (p−>name==s)

return p−>number;

return 0; // use 0 to represent "number not found"

}

In fact, this is roughly the way the terset and less error prone range-for loop is implemented by the compiler. Given an iterator p, the \*p is the element to which it refers, ++p advances p to refer to the next element, and when p refers to a cass with a member m then p−>m is equivalent to (\*p).m; Adding elements to the list and removing elements from a list is easy:

void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)

{

phone\_book.insert(p,ee); // add ee before the element referred to by p

phone\_book.erase(q); // remove the element referred to by q

}

Note that these list examples could be written identically using vectors and surprisingly unless you understand the machine architecture, it will generally perform better; with a smaller vector than with a small list. When all we want is a sequence of elements, we have a choice between using a vector and a list. Unless you have a reason not to use a vector, a vector performs better for traversal(e.g find() and count()) and for sorting and searching (e.g. sort(), binary search()).

Map:

Writing code to look up a name in a list (name, numbers) pairs is quite tedious. In addition, a linear search is inefficient for all but the shortest lists. The standard library offers a search tree called map, in other contexts, a map is known as an associative array or a dictionary. It is implemented as a balanced binary tree. The standard library map is a container of pairs of values optimised for lookup. For example:

map<string,int> phone\_book {

{"David Hume",123456},

{"Karl Popper",234567},

{"Bertrand Arthur William Russell",345678}

};

When indexed by a value of its first type (called a key) a map returns the corresponding value of the second type( called the value or the mapped type). For example:

int get\_number(const string& s){

return phone\_book[s];

}

In other words, subscripting a map is essentially the lookup we get\_number(). If a key isn't found, it is entered into the map with a default value for its value. The default value for an integer is type is 0; the value I just happened to choose represents an invalid telephone number. If we wanted to avoid entering invalid numbers into our phone book, we could use find() and insert() instead of [];

Unordered\_map

The cost of a map loop uses O(log(n)) where n is the number of elements in the map. That’s pretty good, for example, for a map with 1,000,000 elements, we perform only about 20 comparisons and indirections to find an element. However in many cases we can do better by using a hashed look up rather than comparison using an ordering function such as <. The standard library hased containers are referred to as unordered because they don’t require an ordering function.

For example, we can use an unordered\_map from <unordered\_map> to implement our phonebook:

unordered\_map<string,int> phone\_book {

{"David Hume",123456},

{"Karl Popper",234567},

{"Bertrand Arthur William Russell",345678}

};

Like for a map, we can subscript an unordered\_map:

int get\_number(const string& s)

{

return phone\_book[s];

}

The standard-library unordered\_map provides a default hash function for strings. If necessary, you can provide your own.

Container Overview:

A map, a list, and a vector can each be used to represent our phobe\_book example. However each has its own strengths and weaknesses. For example, subscripting and traversing a vector is cheap and easy. On the other hand, vector elements are moved when we insert or remove elements; a list has the exact opposite properties. A map resembles a list of (key,value) pairs except that is optimized for finding values based on keys. Not that a vector is usually more efficient than a list for short sequences of small elements.

With this in mind it is recommended to use stl vector as the default type for sequences of elements; needing a good reason to choose another. The stl provides some of the most general and useful container types to allow the programmers to select a container that best serves the need of the application.

Standard container summary:

* vector<T> A variable-sized vector
* list<T> A doubly-linked list
* f orward\_list<T> A singly-linked list
* set<T> A set
* multiset<T> A set in which a value can occur many times
* map<K,V> An associative array
* multimap<K,V> A map in which a key can occur many times
* unordered\_map<K,V> A map using a hashed lookup
* unordered\_m ultimap<K,V> A multimap using a hashed lookup
* unordered\_set<T> A set using a hashed lookup
* unordered\_m ultiset<T> A multiset using a hashed lookup

The unordered containers are optimised for lookup with a key(often a string); in other words they are implemented using hash tables. The containers defined in the namespace std and presented in headers, <vector>, <list>, <map>, etc. As well as this they present container adapters queue<t>, stack<t>, deque<t> and priority\_queue<t>. The stl also provides some more specialised container-like types, such as fixed sized array array<T,N> and bitset<N>.

The standard containers and their basic operations are defined to be similar from a notational point of view. Furthermore, the meaning of the operations are equivalent for the various containers. Basic operations apply to every kind of container for which they make sense and can be efficiently implemented. For example:

* begin() and end() give iterators to the first and one-beyond the last elements,
* push\_back() can be used (efficiently) to add elements to the end of a vector as well as for a list
* size() returns the number of elements.

This notational and semantic uniformity enables programmers to provide new containers types that can be used in a very similar manner to the standard ones. The range checked, vector is one such example of that. The uniformity of the container interfaces also allows us to specify algorithms independently of individual containers.

Algorithms:

A data structure, such as a list or a vector is not very useful on it’s own. To use one, we need operations for basic actions such as adding and removal of elements. Furthermore, we rarely just store objects in a container; we may sort them, print them, extract subsets, remove elements, search for objects, etc. Consequently the standard library provides the most common algorithms for containers in addition to providing the most common container types.

For example the following sorts a vector and places a copy of each unique element on a list:

bool operator<(const Entry& x, const Entry& y) // less than

{

return x.name<y.name; // order Entrys by their Names

}

void f(vector<Entry>& vec, list<Entry>& lst)

{

sort(vec.begin(),vec.end()); // use < for order

unique\_cop y(vec.begin(),vec.end(),lst.begin()); // don’t copy adjacent equal elements

}

The standard algorithms are expressed in terms of sequences of elements. A sequence is represented by a pair of iterators specifying the first element and the one beyond the last element. In this example sort(), sorts the sequence from ve.begin() to ve.end() - which just happens to be all the elements of the vector. For writing, you will need only specify the first element to be written. If more than one element is written, the elements following that initial element will be overwritten.

Thus to avoid any errors lst, must have at least elements as there are unique values in vec. If we wanted to place the unique element in a new container, we could have written:

list<Entry> f(vector<Entry>& vec)

{

list<Entry> res;

sort(vec.begin(),vec.end());

unique\_copy(vec.begin(),vec.end(),back\_inserter(res)); // append to res

return res;

}

A back\_inserter() adds an element at the end of a container, extending the container to make room for them. Thus the standard caintiner plus back\_inserters() eliminate the need to use error-prone, explicit C style memory management using realloc(). The standard library list has a move constructor that makes return res by value efficient even for a list of a thousand elements. If you were to find the pair of iterators style of code to be tedious, then you can define the container version of the algorithms and write sort(ve).

Use of iterators:

When you first encounter a container, a few iterators referring to useful elements can be obtained; begin() and end() are the best example of this. In addition, many algorithms return iterators.

For example, the standard algorithm find looks for a value in a sequence and then returns an iterator to the element found, like so:

bool has\_c(const string& s, char c) // does s contain the character c?

{

auto p = find(s.begin(),s.end(),c);

if (p!=s.end())

return true;

else

return false;

}

Note that find returns end() to indicate ‘‘not found.’’ An equivalent, shorter, definition of

has\_c() is:

bool has\_c(const string& s, char c) // does s contain the character c?

{

return find(s.begin(),s.end(),c)!=s.end();

}

A more interesting exercise would be to find the location of all occurrences of a character in a string. We can return the set of occurrences as a vector of string iterators. Assuming that we would like to modify the location found, we pass non-const string:

vector<string::iterator> find\_all(string& s, char c) // find all occurrences of c in s

{

vector<string::iterator> res;

for (auto p = s.begin(); p!=s.end(); ++p)

if (∗p==c)

res .push\_back(p);

return res;

}

We iterate through the string using a conventional loop, moving the iterator p forward one element at a time using ++ and looking at the elements using the dereference operator ∗.

We could test find\_all() like this:

void test()

{

string m {"Mary had a little lamb"};

for (auto p : find\_all(m,’a’))

if (∗p!=’a’)

cerr << "a bug!\n";

}

Iterators and standard algorithms will work equivalently on every standard container for which their use makes sense. So we could generalize find\_all():

template<typename C, typename V>

vector<typename C::iterator> find\_all(C& c, V v) // find all occurrences of v in c

{

vector<typename C::iterator> res;

for (auto p = c.begin(); p!=c.end(); ++p)

if (∗p==v)

res .push\_back(p);

return res;

}

The typename is needed to inform the compiler that C’s iterator is supposed to be a type and not a value of some type, let’s say the integer 7. We can hide this implementation detail by introducing a type alias for the iterator, like so:

template<typename T>

using Iterator<T> = typename T::iterator;

template<typename C, typename V>

vector<Iterator<C>> find\_all(C& c, V v) // find all occurrences of v in c

{

vector<Iterator<C>> res;

for (auto p = c.begin(); p!=c.end(); ++p)

if (∗p==v)

res .push\_back(p);

return res;

}

We can now write:

void test()

{

string m {"Mary had a little lamb"};

for (auto p : find\_all(m,’a’)) // p is a string::iterator

if (∗p!=’a’)

cerr << "string bug!\n";

list<double> ld {1.1, 2.2, 3.3, 1.1};

for (auto p : find\_all(ld,1.1))

if (∗p!=1.1)

cerr << "list bug!\n";

vector<string> vs { "red", "blue", "green", "green", "orange", "green" };

for (auto p : find\_all(vs,"green"))

if (∗p!="g reen")

cerr << "vector bug!\n";

for (auto p : find\_all(vs,"green"))

∗p = "vert";

// ...

}

Iterators are used to separate algorithms and containers. An algorithm operates on it’s data through iterators and knows nothing about the container in which the elements are stored. Conversely, a container knows nothing about the algorithms operating on its element; all it does is supply the iterators upon request, the result is very general and flexible software.

Iterator types:

So what are iterators really? Any particular iterator is an object of some type. There are however, many different iterator types, because an iterator needs to hold the information necessary for doing its job for a particular container type. These iterator types can be as different as the containers and the specialised need they serve. For example, a vector’s iterators could be an ordinary pointer, because a pointer is a quite reasonable way of referring to an element of a vector.

Alternatively a vector iterator could be implemented as a pointer to the vector plus an index; using such an iterator would allow range checking. A list iterator must be something more complicated than a simple pointer to an element because an element of a list in general does not know where the next element of that list is. List iterators might be a pointer to a link. What is common for all iterators is their semantics and the naming of their operations.

For example, applying ++ to any iterator yields an iterator that refers to the next element, and similarly the \* yield the element to which the iterator refers to. In fact any object that objects a few simple rules like these is an iterator. Furthermore, users rarely need to know the type of a specific iterator each container knows its iterator types and makes them available under the conventional names iterator and const\_iterator.

Stream iterators:

Iterators are a general and useful concept for dealing with sequences of elements in containers. However, containers are not the only place where we find sequences of elements. For example, an input stream produces a sequence of values and we write a sequence of values to an output stream. Consequently, the notions of iterators can be usefully applied to that if input and output.

To make an ostream\_iterator, we need to specify which stream it will be used for and the type of object written to it. For example we can define an iterator that refers to the standard output stream, cout:

ostream\_iterator<string> oo {cout}

The effect of assigning to \*oo is to write the assigned value to cout, like so:

int main()

{

∗oo = "Hello, "; // meaning cout<<"Hello, "

++oo;

∗oo = "world!\n"; // meaning cout<<"world!\n"

}

This is yet another way of writing the canonical message to standard output. The ++oo is done to mimic writing into an array through a pointer.

Similarly, an istream\_iterators is something that allows us to treat an input stream as a read-only container. Again we must specify the stream to be used and the type of values expected:

istream\_iterator<string> ii{cin};

The input iterators are used in pairs representing a sequence, so we must provide an istream\_iterator to indicate the end of the input. This is the default istream\_iteraror:

istream\_iterator<string> eos{}

Typically , istream\_iterators, and ostream\_iterators are not used directly. Instead they are provided as arguments to algorithms. For example we can write a simple program to read a file, sort the words read, eliminate duplicates and write the result to another file, like so:

int main()

{

str ing from, to;

cin >> from >> to; // get source and target file names

ifstream is {from}; // input stream for file "from"

istream\_iter ator<string> ii {is}; // input iterator for stream

istream\_iter ator<string> eos {}; // input sentinel

ofstream os{to}; // output stream for file "to"

ostream\_iter ator<string> oo {os,"\n"}; // output iterator for stream

vector<string> b {ii,eos}; // b is a vector initialized from input [ii:eos)

sor t(b.begin(),b.end()); // sort the buffer

unique\_cop y(b.begin(),b.end(),oo); // copy buffer to output, discard replicated //values

return !is.eof() || !os; // return error state

}

An istream is an istream that can be attached to a file and an ostream is an ostream that can be attached to a file. The ostream\_iterator’s second argument is used to delimit output values.

Actually the program is longer than it needs to be. We read the string into vector then we sort() them, and then we write out eliminating duplicates. A more elegant solution for this problem is not to store the duplicates at all. This can be done by keeping the strings in a set, which does not keep duplicates and keeps its elements in order. That way, we could replace the two lines using a vector with one using a set to replace unique\_copy(), with the simpler copy.

set<string<b{ii,eos}; //collect strings from input

copy(b.begin(), b.end(), oo); //copy buffer to output

We used the names ii, eos, and oo only once after their definition se we could further reduce the size of the program, like so:

int main()

{

string from, to;

cin >> from >> to; // get source and target file names

ifstream is {from}; // input stream for file "from"

ofstream os {to}; // output stream for file "to"

set<string> b {istream\_iterator<string>{is},istream\_iterator<string>{}}; // readinput

copy(b.begin(),b.end(),ostream\_iterator<string>{os,"\n"}); // copy to output

return !is.eof() || !os; // return error state (§2.2.1, §37.3)

}

It is a matter of taste and experience whether or not this last simplification improves readability. If your tastes lean toward the very terse, you can further eliminate the name os.

Predicate:

In the example above, the algorithms have simply built in the action to be done for each element of a sequence. However we often want to make that action a parameter to the algorithm. For example the find algorithm provides a convenient way of looking for a specific value. A more general variant looks for an element that fulfills a specified requirement or a predicate. For example, we might want to search a map for the first value larger than 42.

A map allows us to access its elements as sequence of (key,value) pairs, so we can search a map<string,int>’s sequences for a pair<const string, int> where the int is greater than 42, like so:

void f(map<string,int>& m)

{

auto p = find\_if(m.begin(),m.end(),Greater\_than{42});

// …

}

Here, Greater\_than is a function object (§3.4.3) holding the value (42) to be compared against:

struct Greater\_than {

int val;

Greater\_than(int v) : val{v} { }

bool operator()(const pair<string,int>& r) { return r.second>val; }

};

Alternatively, we could use a lambda expression:

int cxx = count\_if(m.begin(), m.end(),[](const pair<string,int>& r) { return r.second>42; }

Algorithm overview:

What is an algorithm? A general definition is a finite set of rules which gives a sequence of operations for solving a specific set of problems. It has 5 important features:

1. Finiteness
2. Definiteness
3. Input
4. Output
5. effectiveness

In the context of the c++ standard library, an algorithm is defined in the namespaces std and presented in the <algorithm> header. These standard library algorithms all take sequences as inputs.

Container algorithms:

A sequence is defined by a pair of iterators [begin:end). This is general and flexible, but most often, we apply an algorithm to a sequence that is the contents of a container. For example:

sor t(v.begin(),v.end());

Why don’t we just say sor t(v)? We can easily provide that shorthand:

namespace Estd {

using namespace std;

template<class C>

void sort(C& c)

{

sort(c.begin(),c.end());

}

template<class C, class Pred>

void sort(C& c, Pred p)

{

sor t(c.begin(),c.end(),p);

}

// ...

}

I put the container versions of sor t() (and other algorithms) into their own namespace Estd(‘‘extended std’’) to avoid interfering with other programmers’ uses of and extensions to

std.

Concurrency and utilities-

Introduction:

From an end users perspective, the ideal standard library would provide components directly supporting essentially every need. For a given application domain, a huge commercial library can come close to that ideal. For a given application domain a huge commercial library can come close to that ideal. However that is not what the C++ standard library is trying to do. A manageable, universally available library cannot be everything to everybody.

Instead the C++ standard library aims to provide components that are useful to most people in most application areas. That is, it aims to serve the intersection of all needs rather than their union. In addition, the support for a few widely important application areas, such as mathematical computation and text manipulation, have crept in.

Resource management:

One of the key tasks of any nontrivial program is to manage resources. A resource is something that must be acquired later(explicitly or implicitly) released. Examples are memory, locks, sockets, thread handles, file handles. For a long running program failing to release a resource in a timely manner(a leak) can cause serious performance degradation and possibly even a miserable crash. Even for short programs, a leak can become an embarrassment, say by a resource shortage increasing the run time by orders of magnitude.

The standard library components are designed not to leak resources. To do this they rely on basic language support for resource management using constructor/destructor pairs to ensure that a resource does not outlive an object responsible for it. The use of a constructor/destructor pair in vector to manage the lifetime of it’s elements is an example and all standard library containers are implemented in similar manners. Importantly this approach interacts correctly with error handling using exceptions, for example, the technique is used for the standard library lock classes:

mutex m; //used to protect access to shared data

//..

void f(){

lock\_guard<mutex> lck{m}; acquire the mutex m;

//..manipulate shared data…

}

A thread will not proceed until lck’s constructor has acquired its mutex, m, the corresponding destructor releases the resources. So in this example, lock\_guard’s destructor releases the mutex when the thread of control leaves f() (through a return, by falling off the end of the function or throwing an exception). This is an application of the resources acquisition is initialization technique (RAII); this technique is fundamental to the idiomatic handling of resources in C++. Containers (such as vector and map), string and iostream manage their resources (such as file handles and buffers) similarly.

unique\_ptr and shared\_ptr

The examples so far take care of objects defined in a scope, releasing the resources they acquire at the exit from the scope, but what about objects allocated on the heap or the free store. In <memory>, the standard library provides two smart pointers to help manage objects on the free store:

* unique\_ptr to represent unique ownership
* Shared\_ptr to represent shared ownership

The most basic use of one of these smart pointers is to prevent memory leaks caused by careless programming:

void f(int i, int j) // X\* vs. unique\_ptr<X>

{

X∗ p = new X; // allocate a new X

unique\_ptr<X> sp {new X};// allocate a new X and give its pointer to unique\_ptr

// ...

if (i<99) throw Z{}; // may throw an exception

if (j<77) return; // may return "early"

p−>do\_something(); // may throw an exception

sp−>do\_something(); // may throw an exception

// ...

delete p; // destroy \*p

}

Here we forgot to delete p if i < 99 or if j < 77. On the other hand, unique\_ptr ensures that its abject is properly destroyed whichever way we exit f() (by throwing an exception, by executing return or by falling off the end). In this simple case, we could have solved the problem simply by not using a pointer and not using new:

Void f(int i, int j)//use a local variable

{

X x;

//…

}

Unfortunately, overuse of new (and of pointers and references) seems to be an increasing problem. However, when you really need the semantics of pointer, unique\_ptr is a very light weight mechanism with no space or tie overhead compared to correct use of built in pointer.

It further uses include passing free-store or heap allocated object in and out of functions, like so:

unique\_ptr<X>make\_X(int i)

//make an x immediately give it to a unique\_ptr

{

//check i, etc.

return unique\_ptr<X>{new X{i}};

}

A unique\_ptr is a handle to an individual object (or an array in much the same way that a vector is handled to a sequence of objects. Both control the lifetime of other objects (using RAII) and both rely on move semantics to make return simple and efficient.

The shared\_ptr is similar to unique\_ptr except that shared\_ptrs are copied rather than moved. The shared\_ptrs for an object share ownership of an object and that object is destroyed when the last of its shared\_ptrs is destroyed. For example:

void f(shared\_ptr<fstream>);

void g(shared\_ptr<fstream>);

void h(shared\_ptr<fstream>);

void user(const string& name, ios\_base::openmode mode)

{

shared\_ptr<fstream> fp {new fstream(name,mode)};

if (!∗fp) throw No\_file{}; // make sure the file was properly opened

f(fp);

g(fp);

h(fp);

// ...

}

Now, the file opener by fp’s constructor will be closed by the last function to explicitly or implicitly) destroy a copy of fp. Note the f(), g(), or h() may spawn a task holding a copy of fp or in some other way store a copy that outlives user(). Thus, shared\_ptr provides a form of garbage collection that respects the destructor based resource management of the memory managed objects. This is neither cost free nor exorbitantly expensive, but does make the lifetime of the shared hard to predict.

Use shared\_ptr only if you actually need shared ownership. Given unique\_ptr and shared\_ptr, we can implement a complete no naked new policy for many programs. However these smart pointers are still conceptually pointers and therefore only my second choice for resource management - after containers and other types that manage their resource at a higher conceptual level. In particular shared\_ptrs do not in themselves provide any rules for which of their owners can read and/ or write the shared object.

Data races and other forms of confusion are not addressed simply by eliminating the resource management issues. Where do we use smart pointers rather than resource handles with operations design specifically for the resource?

Unsurprisingly the answer is when we need pointer semantic, so:

* When we share an object, we need pointers (or references) to refer to the shared object, so shared\_ptr becomes the obvious choice unless there is an obvious single owner.
* When we refer to a polymorphic object, we need a pointer (or a reference) because we don’t know the exact type of the object or even it’s size), so unique\_ptr becomes the obvious choice.
* A shared polymorphic object typically requires shared\_ptrs

We do not need to use a pointer to return a collection of objects from a function; a container that is a resource handle will do that simply and efficiently.

Concurrency:

Concurrency refers to the execution of several tasks simultaneously, it is widely used to improve throughput (by using several processors for a single computation) or to improve responsiveness (by allowing one part of a program to progress while another is waiting for a response). All modern programming languages provide support for this. The support provided by the C++ standard library is a portable and type safe variant of what has been used in C++ for more than 20 years and is almost universally supported by modern hardware.

The standard library support is primarily aimed at supporting systems level concurrency rather than directly providing sophisticated higher level concurrency models; those can be supplied as libraries built using the standard library facilities. The standard library directly supports concurrent execution of multiple threads in a single address space. To allow that, C++ provides a suitable memory model, and a set of atomic operations. However most users will see concurrency only in terms of the standard library and libraries built on top of that.

This section briefly gives examples of the main standard library concurrency support facilities: threads, mutexes, lock() operations, packaged\_tasks and futures. These are built directly upon what operating systems offer and do not incur performance penalties compared with those.

Tasks and Threads:

We call a computation that can potentially be executed concurrently with other computations a task. A thread is the system level representation of a task in a program. A task to be executed concurrently with another task is launched by constructing a std::thread ( found in <thread>) with the task as it’s argument. A task is a function or a functor:

void f(); // function

struct F { // function object

void operator()(); // F’s call operator (§3.4.3)

};

void user()

{

thread t1 {f}; // f() executes in separate thread

thread t2 {F()}; // F()() executes in separate thread

t1.join(); // wait for t1

t2.join(); // wait for t2

}

The join()s ensure that we don’t exit user() until the threads have completed. To “join” means to “wait for the thread to terminate.”

Threads of a program share a single address space. In this, threads differ from processes, which generally do not directly share data. In this, threads differ from processes, which generally do not directly share data. Since threads share an address space, they can communicate through shared objects. Such communication is typically controlled by locks or other mechanisms to prevent data races (uncontrolled concurrent access to a variable).

Programming concurrent tasks can be very tricky. Consider the possible implement of f an F for instance:

void f() { cout << "Hello "; }

struct F { void operator()() { cout << "Parallel World!\n"; } };

This is an example of a bad error. Here f() and F() each use the object cout, without any form of synchronization. The resulting output would be unpredictable and could vary between different executions of the program because the order of executing of the individual operations in the two tasks is not defined. The program may crash because cout was corrupted or produce “odd” output, such as:

PaHerallllelo World!

When defining tasks of a concurrent program, our aim is to keep tasks completely separate except where they communicate in simple and obvious ways. The simplest way of thinking of a concurrent task is as a function that happens to run concurrently with its caller. For that to work, we just have to pass arguments, get a result back, and make sure that there is no use of shared data in between (no data races).

Passing arguments:

Typically, a task needs data to work upon. We can easily pass data(or pointer or references to the data) as arguments. Consider:

void f(vector<double>& v); // function do something with v

struct F { // function object: do something with v

vector<double>& v;

F(vector<double>& vv) :v{vv} { }

void operator()(); // application operator;

};

int main(){

vector<double> some\_vec {1,2,3,4,5,6,7,8,9};

vector<double> vec2 {10,11,12,13,14};

thread t1 {f,some\_vec}; // f(some\_vec) executes in a separate thread

thread t2 {F{vec2}}; // F(vec2)() executes in a separate thread

t1.join();

t2.join();

}

Obviously, F{vec2} saves a reference to the argument vector in F. F can now use that array and hopefully no other task accesses vec2 while F is executing. Passing vec2 by value would eliminate that risk. The initialisation with {f,some\_vec} uses a thread variadic template constructor that can accept an arbitrary sequence of arguments. The compiler checks that the first argument can be involved given the following arguments and builds the necessary function object to pass to the thread.

Thus, if F::operator()() and f() perform the same algorithm, the handling of the two tasks are roughly equivalent: in both cases, a function object is constructed for the thread to execute.

Returning Results:

In the example in the previous section, I pass the arguments by non-const reference. I only do that if I expect the task to modify the value of the data referred to. That’s a somewhat sneaky but non uncommon method of returning a result. A less obscure technique is to pass the input data by const reference and to pass the location of a place to deposit the result as a separate argument.

void f(const vector<double>& v, double∗ res); // take input from v; place result in \*res

class F {

public:

F(const vector<double>& vv, double∗ p) :v{vv}, res{p} { }

void operator()(); // place result in \*res

private:

const vector<double>& v; // source of input

double∗ res; // target for output

};

int main()

{

vector<double> some\_vec;

vector<double> vec2;

// …

double res1;

double res2;

thread t1 {f,some\_vec,&res1}; // f(some\_vec,&res1) executes in a separate thread

thread t2 {F{vec2,&res2}}; // F{vec2,&res2}() executes in a separate thread

t1.join();

t2.join();

cout << res1 << ’ ’ << res2 << ’\n’;

}

The returning of results through arguments, isn’t very elegant so this topic will be continued further on in the chapter.

Sharing data:

Sometimes tasks need to share data. In that case, the access has to be synchronised so that at most one task at a time has access. Experienced programmers will recognise this as a simplification (e.g., there is no problem with many tasks simultaneously reading immutable data), but consider how to ensure that at most one task at a time has access to a given set of objects. The fundamental element of the solution is a mutex, a mutual exclusive object. A thread acquires a mutex using a lock() operation:

m utex m; // controlling mutex

int sh; // shared data

void f()

{

lock\_guard<mutex> lck {m}; // acquire mutex

sh += 7; // manipulate shared data

} // release mutex implicitly

The lock\_guard’s constructor acquires the mutex (through a call m.lock()). If another thread has already acquired the mutex, the thread waits(blocks) until the other thread completes it’s access. Once a thread has completed it’s access to the shared data, the lock\_guard releases the mutex (with a call m.lock()). The mutual exclusion and locking facilities are found in <mutex>.

The correspondence between the shared data and a mutex is conventional: the programmer simply has to know which mutex is supposed to correspond to which data. Obviously, this is error prone, and equally obviously we try to make the correspondence clear through various language means. For example:

class Record {

public:

mutex rm;

//..

};

It doesn't take a genius to guess that for a record called rec, rec.rm is a mutex that you are supposed to acquire before accessing the other data of rec, throg a comment or a better anime might have helped a reader.

It is not uncommon to need to simultaneously access several resources to perform some actions. This can lead to deadlock. For example, if thread1 acquires the mutex1 and then tries to acquire mutex2 while thread2 tries to acquire mutex2 and then tries to acquire mutex1, then neither task will ever proceed further.

The standard library offers help in the form of an operation for acquiring several locks simultaneously:

void f()

{

// ...

lock\_guard<m utex> lck1 {m1,defer\_lock}; // defer\_lock: don’t yet try to acquire themutex

lock\_guard<m utex> lck2 {m2,defer\_lock};

lock\_guard<m utex> lck3 {m3,defer\_lock};

// ...

lock(lc k1,lc k2,lc k3); // acquire all three locks

// ... manipulate shared data ...

} // implicitly release all mutexes

This lock() will only proceed after acquiring all it’s mutex arguments and will never block (go to sleep while holding a mutex. The destructors for the individual lock\_guards ensure that the mutexes are released when a thread leaves the scope.

Communicating through shared data is pretty low level. In particular, the programmer has to devise ways of knowing what work has and has not been done by various tasks. In that regard, use of shared data is inferior to the notion of call and return. On the other hand, some people are convinced that sharing must be more efficient than copying arguments and returns. That can indeed be so when large amounts of data are involved, but locking and unlocking are relatively expensive operations.

On the other hand, modern machines are very good at copying data, especially compact data, such as vector elements. So don’t choose shared data for communication because of efficiency without thought and preferably not without measurements.

Waiting for events:

Sometime, a thread needs to wait for some kind of external event, such as another thread completing a task or a certain amount of time having passed. The simplest event is simply time passing. Consider the following:

using namespace std::chrono; // see §35.2

auto t0 = high\_resolution\_clock::now();

this\_thread::sleep\_for(milliseconds{20});

auto t1 = high\_resolution\_clock::now();

cout << nanoseconds(t1−t0).count() << " nanoseconds passed\n";

Note that I didn't even have to launch a thread; by default, this\_thread refers to the one and only thread. The time facilities are found within <chrono>.

The basic support for communicating using external events is provided by condition\_variables found in <condition\_variables>. A condition\_variable is a mechanism allowing one thread to wait for another. In particular, it allows a thread to wait for some condition (often called an event) to occur as the result of work done by other threads. Consider the classical example of two threads communicating by passing messages through a queue.

For simplicity, I declare the queue and the mechanism for avoiding race conditions on that queue global to the producer and consumer:

class Message { // object to be communicated

// ...

};

queue<Message> mqueue; // the queue of messages

condition\_variable mcond; // the variable communicating events

mutex mmutex; // the locking mechanism

The types queue, condition\_variable, and m utex are provided by the standard library.

The consumer() reads and processes Messages:

void consumer()

{

while(tr ue) {

unique\_loc k<mutex> lck{mmutex}; // acquire mmutex

mcond.wait(lck); // release lck and wait;

// re-acquire lck upon wakeup

auto m = mqueue.top(); // get the message

mqueue .pop();

lck.unlock(); // release lck

// ... process m ...

} }

Here, I explicitly protect the operations on the queue and on the condition\_variable with a unique\_loc k on the mutex. Waiting on condition\_variable releases its lock argument until the

wait is over (so that the queue is non-empty) and then reacquires it.

The corresponding producer looks like this:

v oid producer()

{

while(true) {

Message m;

// ... fill the message ...

unique\_loc k<mutex> lck {mmutex}; // protect operations

mqueue .push(m);

mcond.notify\_one(); // notify

} // release lock (at end of scope)

}

Using condition\_variables supports many forms of elegant and efficient sharing, but can be rather tricky.

Communicating tasks:

The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks ( work to potentially be done concurrently) rather than directly at the lower level of threads and locks:

* future and promise for returning a value from a task spawned on a separate thread
* packaged\_task to help launch tasks and connect up the mechanisms for returning a result.
* async() for launching of a task in a manner very similar to calling a function

These facilities are found within the <future>.

future and promise:

The important point about future and promise is that they enable a transfer of a value between two tasks without explicit use of a lock; the system implements the transfers efficiently. The basic idea is simple: when a task wants to pass a value to another, it puts the value into a promise. Somehow, the implementation makes that value appear in the corresponding future, from which it can be read(typically by the launcher of the task).

If we have future<X> called fx, we can get() a value of type X from it:

X v fx.get(); //if necessary, wait for the value to get computed

If the value isn’t there yet, our thread is blocked until it arrives. If the value couldn’t be computed, get() might throw an exception (from the system or transmitted from the task from which we were trying to get() the value). The main purpose of a promise is to provide simple put operations (called set\_value() and set\_exception()) to match future’s get(). If you have a promise and need to send a result of type X to a future, you can do one of two things: pass a value or pass an exception. For example:

void f(promise<X>& px) // a task: place the result in px

{

// …

tr y {

X res;

// ... compute a value for res …

px.set\_value(res);

}

catch (...) { // oops: couldn’t compute res

// pass the exception to the future’s thread:

px.set\_e xception(current\_exception());

} }

The current\_exception() refers to the caught exception. To deal with an exception transmitted through a future, the caller of get() must be prepared to catch it somewhere.

For example:

void g(future<X>& fx) // a task: get the result from fx

{

// ...

tr y {

X v = fx.get(); // if necessary, wait for the value to get computed

// ... use v ...

}

catch (...) { // oops: someone couldn’t compute v

// ... handle error ...

} }

Packaged\_task

How do we get a future into the task that needs a result and the corresponding promise into the thread that should produce the result? The packaged\_taks is provided to simplify setting up tasks connected with futures and promises to be run on threads. A packaged\_task provides wrapper code to put the return value or exception from the task into a promise. If you ask it, the packaged\_taks wil give you the corresponding future. For example, we can set up two tasks to each add half of the elements of a vector<double> using the standard library accumulate():

double accum(double∗ beg, double ∗ end, double init)

// compute the sum of [beg:end) starting with the initial value init;

{

return accumulate(beg,end,init);

}

double comp2(vector<double>& v)

{

using Task\_type = double(double∗,double∗,double); // type of task

packaged\_task<Task\_type> pt0 {accum}; // package the task (i.e., accum)

packaged\_task<Task\_type> pt1 {accum};

future<double> f0 {pt0.get\_future()}; // get hold of pt0’s future

future<double> f1 {pt1.get\_future()}; // get hold of pt1’s future

double∗ first = &v[0];

thread t1 {move(pt0),first,first+v.size()/2,0}; // start a thread for pt0

thread t2 {move(pt1),first+v.size()/2,first+v.size(),0}; // star t a thread for pt1

// ...

return f0.get()+f1.get(); // get the results

}

The packaged\_task template takes the type of the task as it’s template argument ( here Task\_type), an alias for double(double\*,double\*, double)) and the task as its constructor argument (here accum). The move() operations are needed because packaged\_ tasks cannot be copied.

Please note the absence of explicit mention of locks in this code: we are able to concentrate on tasks to be done, rather than on the mechanisms used to manage their communication. The two tasks will be run on separate threads and thus potentially in parallel.

async()

The line of thinking we have pursued in this chapter is one I believe to be the simplest yet still among the most powerful: treat a task as a function that may happen to run concurrently with other task. It is far from the only model supported by the C++ standard library, but it serves well for a wide range of needs. More subtle and tricky models e.g., styles of programming relying on shared memory, can be used as needed. The standard library function async() provides a very simple way of executing a task asynchronously:

double comp4(vector<double>& v)

// spawn many tasks if v is large enough

{

if (v.size()<10000) return accum(v.begin(),v.end(),0.0);

auto v0 = &v[0];

auto sz = v.size();

auto f0 = async(accum,v0,v0+sz/4,0.0); // first quarter

auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0); // second quarter

auto f2 = async(accum,v0+sz/2,v0+sz∗3/4,0.0); // third quarter

auto f3 = async(accum,v0+sz∗3/4,v0+sz,0.0); // fourth quarter

return f0.get()+f1.get()+f2.get()+f3.get(); // collect and combine the results

}

Basically, async() separates the call part of a function call from the get result part and separates both from the actual execution of the task. Using async(), you don’t have to think about threads and locks. Instead, you think just in terms of tasks that potentially compute their results asynchronously. There is an obvious limitation.

Don’t even think of using async() for tasks that share resources needing locking - with async() you don’t even know how many threads will be used because that’s up to async() to decide based on what it knows about the system resources available at the time of a call. For example, async() may check whether any idle cores (processors) are available before deciding how many threads to use.

Please note that async() is not just a mechanism specialised for parallel computation for increased performance. For example, it can also be used to spawn a task for getting information from a user, leaving the main program active with something else.

Small utility components:

Not all standard library components come as part of obviously labelled facilities, such as containers or I/O. This section gives a few examples of small, widely useful components:

* clock and duration for measuring time
* Type functions, such as iterator\_traits and is\_arithmetic, for gaining information about types
* pair and tuple for representing small potentially heterogeneous sets of values

The point here is that a function or a type need not be complicated or closely tied to a mass of other functions and types to be useful. Such library components mostly act as building blocks for more powerful library facilities, including other components of the standard library.

Time

The standard library provides facilities for dealing with time. For example here is the basic way of time something:

using namespace std::chrono; // see §35.2

auto t0 = high\_resolution\_clock::now();

do\_work();

auto t1 = high\_resolution\_clock::now();

cout << duration\_cast<milliseconds>(t1−t0).count() << "msec\n";

The clock returns a time\_point(a point in time). Subtracting to time\_points gives a duration(a period of time). Various clocks give their results in various units of time (the clock used here is measured in nanoseconds), so it is usually a good idea to convert a duration into a known unit. That’s what duration\_cast does. The standard library facilities for dealing with time are found in the sub namespace std::chrono in <chrono>.

Type functions:

A type function is a function that is evaluated at compile time given a type as its argument or returning a type. The standard library provides a variety of type functions to help library implementers and programmers in general to write code that takes advantage of aspects of the language, the standard library, and code in general.

For numerical types, numeric\_limits from <limits> presents useful information, for example:

Constexpr float min = numerical\_limits<float>::min(); //smallest positive float

Similarly , information about sizes can be extracted by the built in sizeof operator, for example:

Constexpr int szi = sizeof(int); // the number of byes in an int

Such type functions are part of C++’s mechanisms for compile time computation that allow tighter type checking and better performance that would otherwise have been possible. Use of such features is often called metaprogramming or (when templates are involved) template metaprogramming. Here, I just present two facilities provided by the standard library: iterator\_traits and type predicates.

Iterator\_traits:

The standard library sort() takes a pair of iterators supposed to define a sequence. Furthermore, these iterators must offer random access to that sequence, that is they must be random access iterators. Some containers, such as forward\_list, do not offer that. In particular, a forward\_list is a simply linked list so subscripting would be expensive and there is no reasonable way to refer back to previous elements. However, like most containers, forward\_list offers forward iterators that can be used to traverse the sequence by algorithms and for statements.

The standard library proves a mechanism, iterator\_traits that allows us to check which kind of iterator is supported. Given that, we can improve range sort() from earlier to accept either a vector or a forward\_list. For example:

void test(vector<string>& v, forward\_list<int>& lst)

{

sort(v); // sort the vector

sort(lst); // sort the singly-linked list

}

The techniques needed to make that work are generally useful. First, I write two helper functions that take an extra argument indicating whether they are to be used for random-access iterators or forward iterators. The version for random access iterator is trivial:

template<typename Ran> // for random-access iterators

void sort\_helper(Ran beg, Ran end, random\_access\_iterator\_tag)

// we can subscript into [beg:end)

{

sor t(beg,end); // just sort it

}

The version for forward iterators is almost as simple; just copy the list into a vector, sort,

and copy back again:

template<typename For> // for forward iterators

void sort\_helper(For beg, For end, forward\_iterator\_tag)

// we can traverse [beg:end)

{

vector<decltype(∗beg)> v {beg,end}; // initialize a vector from [beg:end)

sort(v.begin(),v.end());

copy(v.begin(),v.end(),beg); // copy the elements back

}

The decltype() is a built-in type function that returns the declared type of it’s argument. Thus, v is vector<X> where X is the element type of the input sequence. The real type magic is in the selection of helper functions:

template<class C)

void sort(C& c)

{

using Iter = Iterator\_type<C>;

sort\_helper(c.begin(),c.end(),Iterator\_category<Iter>{});

}

Here there are two type functions: Iterator\_type<C> returns the iterator type of C (that is C::iterator\_ and then iterator\_category<iter>{}{ constructs a tag value indication the kind of iterator provided:

* Std::random\_access\_iterator\_tag if C’s iterator supports random access
* Std::forward\_iterator\_tag if C’s iterator supports forward iteration

Given that, we can select between the two sorting algorithms at compile time. This technique is called tag dispatch is one of the several used in the standard library and elsewhere to improve flexibility and performance. The standard library support for techniques for using iterators, such as tag dispatch comes in the form of a simple class template iterator\_traits from <iterator>. This allows simple definition of type functions used in sort():

template<typename C>

using Iterator\_type = typename C::iterator; // C’s iterator type

template<typename Iter>

using Iterator\_category = typename std::iterator\_traits<Iter>::iterator\_category; // Iter’s category

If you don’t want to know what kind of ‘‘compile-time type magic’’ is used to provide the standard library features, you are free to ignore facilities such as iterator\_traits. But then you can’t use the techniques they support to improve your own code.

Type predicates:

A standard library type predicate is a simple type function that answers a fundamental question about types. For example:

bool b1 = Is\_Arithemetic<int>(); //yes, int is an arithmetic type

bool b2 = Is\_Arithemetic<string>(); //no, string isn’t an arithmetic type

These predicates are found in <type\_traits>, other examples are is\_class, is\_pod, is\_literal\_type, has\_virtual\_destructor, and is\_base\_of. They are most useful when we write templates. For example:

template<typename Scalar>

class complex {

Scalar re , im;

public:

static\_assert(Is\_arithmetic<Scalar>(), "only support complex of arithmetic types");

// …

};

To improve readability compared to using the standard library directly, I defined a type

function:

template<typename T>

constexpr bool Is\_arithmetic()

{

return std::is\_arithmetic<T>::value ;

}

Older programs use ::value directly instead of (), but I consider that quite ugly and it

exposes implementation details.

Pair and Tuple:

Often, we need some data that is just data; that is, a collection of values, rather than an object of a class with a well defined semantics and an invariant for its value. In such cases, we could define a simple struct with an appropriate set of appropriately named members. alternatively , we could let the standard library write the definition for us. For example, the standard library algorithm equal\_range returns a pair of iterators specifying a sub sequence meeting a predicate:

template<typename Forward\_iterator, typename T, typename Compare>

pair<Forward\_iterator,Forward\_iterator>

equal\_range(Forward\_iterator first, Forward\_iterator last, const T& val, Compare cmp);

Given a sorted sequence [first:last), equal\_range() will return the pair representing the subsequence that matches the predicate cmp. We can use that to search in a sorted sequence of records:

void f(const vector<Record>& v)

{

// assume that v is sorted on its "name" field

auto er = equal\_range(v.begin(),v.end(), "Reg",

[](const Record& r1, const Record& r2) { return r1.name==r2.name;}

);

for (auto p = er.first; p!=er.second; ++p) // print all equal records

cout << ∗p; // assume that << is defined for Record

}

The first member of a pair is called first and the second member is called second. This naming is not particularly creative and may look a bit off at first, but such consistent naming is a boon when we want to write generic code.

The standard library pair (from <utility>) is quite frequently used in the standard library and elsewhere. A pair provides operators, such as =, ==, <, it it’s elements do. The make\_pair() function makes it easy to create a pair without explicitly memento its type. For example:

void f(vector<string>&v){

auto pp = make\_pair(v.begin(),2); // pp ia a pair <vector<string>::interator,int>

}

If you need more than two elements or less, you can use tuple (also from <utility>). A tuple is heterogeneous sequence of elements; for example:

tuple<string,int,double> t2("Sild",123, 3.14); // the type is explicitly specified

auto t = make\_tuple(string("Herring"),10, 1.23); // the type is deduced

// t is a tuple<string,int,double>

string s = get<0>(t); // get first element of tuple

int x = get<1>(t);

double d = get<2>(t);

The elements of a tuple are numbered (starting with zero), rather than named the way elements of pairs are (first and second). To get compile-time selection of elements, we must unfortunately use the ugly get<1>(t), rather than get(t,1) or t[1]. Like pairs, tuples can be assigned and compared if their elements can be. A pair is common in interface because often we want to return more than one value, such as result and indicator of the quality of that result. It is less common to need three or more parts as a result, so tuples are more often found in the implementations of generic algorithms.

Regular expressions:

Regular expressions are a powerful tool for text processing. They provide a simply and tersleyt described patterns in text (e.g., a U.S. ZIP code such as TX 77845, or an ISO-style date, such as 2009-06-07) and to efficiently find such patterns in text. In <regex>, the standard library provides support for regular expressions in the form of the std::regex class and it’s supporting functions. To give a taste of the regex library, let us define and print a pattern, like so:

regex pat(R”(\w{2}\s\*\d{5}(-\d{4})?)”; //ZIP code pattern XXddddd-dddd and variants

cout << “pattern: ” << pat << “\n”;

The regular expression specifies a pattern starting with two letters /w{2} optionally followed by some space \s\* followed by five digits \d{5} and optionally followed by a dash and four digits -\d{4}. To express the pattern, I used a raw string literal starting with a R”( and terminated by a )”. This allows backslashes and quotes to be represented in the string without the use of special notation.

The simplest way of using a pattern is to search for it in a stream:

int lineno = 0;

for (string line; getline(cin,line);) { // read into line buffer

++lineno;

smatch matches; // matched strings go here

if (regex\_search(line,matches,pat)) // search for pat in line

cout << lineno << ": " << matches[0] << ’\n’;

}

The regex\_search(line,match,pat) searches the line for anything that matches the regular expression stored in pat and if it finds any matches, it stores them in matches. If no match was found in the regex search will return false. The matches variable is of type smatch. The s stands for sub and an smatch is a vector of sub matches. The first element, here matches[0], is the complete match.

Math:

C++ wasn’t designed primarily with numerical computation in mind. However, C++ is heavily used for numerical computation and the standard library reflects that.

Mathematical functions and algorithms:

In <cmath>, we find the usual mathematical functions, such as sqrt(), log(), and sin() for arguments of type float, double, and long double. Their complex number versions are found in <complex>.

In <numeric> we find a small set of generalized numerical algorithms, such as

accumulate(). For example:

list<double> lst {1, 2, 3, 4, 5, 6 , 9999.99999};

auto s = accumulate(lst.begin(),lst.end(),0.0);

cout << s << ’\n’;

These algorithms work for every standard-library sequence and can have operations supplied as arguments.

Complex numbers:

The standard library supports a family of complex number types along the lines of the complex class described previously. To support complex numbers where scalars are single precision floating point numbers(float), double precision floating point numbers (double), etc, the standard library complex is a template:

template<typename Scalar>

class complex {

pub lic:

complex(const Scalar& re ={}, const Scalar& im ={});

// …

};

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers; for example:

v oid f(complex<float> fl, complex<double> db)

{

complex<long double> ld {fl+sqrt(db)};

db += fl∗3;

fl = pow(1/fl,2);

// ...

}

The sqrt() and pow() (exponentiation) functions are among the usual mathematical functions defined in <complex>.

Random numbers:

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of applications areas is reflected in the wide selection of random number generators provided by the standard library <random>. A random number generator consists of two parts:

* An engined that produces a sequence of random or pseudo-random values
* A distribution that maps those values into a mathematical distribution in a range.

Examples of distributions are uniform\_int\_distribution ( where all integers produced are equally likely), normal distribution (the bell curve), and exponential\_distribution (exponential growth); each for some specified range. For example:

using my\_engine = default\_random\_engine; // type of engine

using my\_distribution = uniform\_int\_distribution<>; // type of distribution

my\_engine re {}; // the default engine

my\_distribution one\_to\_six {1,6}; // distribution that maps to the ints 1..6

auto dice = bind(one\_to\_six,re); // make a generator

int x = dice(); // roll the dice: x becomes a value in [1:6]

The standard library function bind() makes a function object that will invoke it’s first argument (here, one\_to\_six) given it’s second argument (here,re) as its argument. Thus a call to dice() is the equivalent to a call one\_to\_six(re).

Thanks to its uncompromising attention to generality and performance one expert has deemed the standard library random number component “what every random number library wants to be when it grows up”. However, it can hardly be deemed novice friendly. The using statements makes what is being done a bit more obvious. It could instead of been written like so:

auto dice = bind(uniform\_int\_distribution<>{1,6}, default\_random\_engine{});

Which version is the more readable depends entirely on the context and the reader. For novices of any background the fully general interface to the random number library can be a serious obstacle. A simple uniform random number generator is often sufficient to get started. For example:

Rand\_int rnd {1,10}; // make a random number generator for [1:10]

int x = rnd(); // x is a number in [1:10]

So, how could we get that> we have to get something like dice() inside a class Rand\_int:

class Rand\_int {

pub lic:

Rand\_int(int low, int high) :dist{low,high} { }

int operator()() { return r(); }

private:

def ault\_random\_engine re;

unif orm\_int\_distribution<> dist;

auto r = bind(dist,re);

};

That definition is still expert level, but the use of Rand\_int() is manageable in the first week of a C++ course for novices. For example:

int main()

{

Rand\_int rnd {0,9}; // make a uniform random number generator

vector<int> mn(10); // make a vector of size 10

f or (int i=0; i!=500; ++i)

++mn[r nd()]; // fill mn with the frequencies of numbers [0:9]

or (int i = 0; i!=mn.size(); ++i) { // write out a bar graph

cout << i << ’\t’;

for (int j=0; j!=mn[i]; ++j) cout << ’∗’;

cout << endl;

} }

The output is a (reassuringly boring) uniform distribution (with reasonable statistical variation):

0 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

1 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

2 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

3 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

4 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

5 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

6 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

7 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

8 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

There is no standard graphics library for C++, so used here is “ASCII graphics.” Obviously, there are lots of open source and commercial graphics and GUI libraries for C++, but in this book I’ll restrict myself to ISO standard facilities.

Vector Arithmetic:

The vector described previously was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to vector

would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides

(in <valarray>) a vector-like template, called valarray, that is less general and more amenable

to optimization for numerical computation:

template<typename T>

class valarray {

// …

};

The usual arithmetic operations and the most common mathematical functions are supported for v alarrays. For example:

void f(valarray<double>& a1, valarray<double>& a2)

{

valarray<double> a = a1∗3.14+a2/a1; // numeric array operators \*, +, /, and =

a2 += a1∗3.14;

a = abs(a);

double d = a2[7];

// …

}

Numerical Limits:

In <limits>, the standard library provides classes that describe the properties of built-in

types – such as the maximum exponent of a float or the number of bytes in an int; For example, we can assert that a char is signed:

static\_asser t(numeric\_limits<char>::is\_signed,"unsigned characters!");

static\_asser t(100000<numeric\_limits<int>::max(),"small ints!");

Note that the second assert (only) works because n umer ic\_limits<int>::max() is a constexpr function.