Table of Contents

1. Introduction

Developed by Bjarne Stroustrop around 1979.

Intended to be C with Object Orientation (without sacrificing performance or low level manipulation).

Notable Features

- Virtual functions
- Operator overloading
- Multiple inheritance
- Exception handling
- Generics (known as Templates)

2. C++ Design Goals

C was made with efficiency, portability and independence of IDEs in mind.

Follows the **zero-overhead principle** (what you don't use you don't pay with respect to computational resources).

- Statically typed, general-purpose language.
- Multiple programming styles
  - Procedural Programming
  - Data Abstraction

- Object-Oriented Programming
- Generic Programming

3. C++ Program Structure

3.1. Header (.h)

Declaration of code elements.

- **Class structure** - methods and members.
- **Function signature** - name, variables, return variables.

```
// Pre-processor directives that guard against
// re-declaration should fib.h be included
// again elsewhere.
#ifndef _fib_h
#define _fib_h
// A function DECLARATION,
// no function body.
int fib(int n);
#endif // matches the #ifndef
```

3.2. Source (.cpp)

Implementation of functions and class methods.

- Compiled into **binary** (**.o**) object files

```
// Pre-processor directive including the DECLARATION
// of the fib function.
#include "fib.h"
// Function DEFINITION
int fib(int n)
{
if(n <= 2)
{ return 1 };
return fib(n-1) + fib(n-2);
}

#include <iostream> // Include I/O Stream library headers.
// Include the DECLARATION of the fib function.
int main(void)
{
int x;
// Output to standard output
std::cout << "Enter an integer: " << std::endl;
// Read in an integer from standard input
std::cin >> x;
// Output the result
std::cout << "fib(" << x << ") is " << fib(x) << std::endl;
```

4. C++ Compilation

Compile source (**.cpp**) files to create binary code.

```
# Compile fib.cpp containing int fib(int n) function
# -c create binary object fib.o from fib.cpp
```

```
g++ fib.cpp -c
# fibdriver.cpp knows how to call the int fib(int n) function
# because #include "fib.h"
g++ fibdriver.cpp -c
```

We then *link* binary objects (**.o**) to create an executable.

```
g++ fibdriver.o fib.o -o fib
```

4.1. C++ Libraries

C++ collects binary code into *libraries*.

Unix libraries reside in **/usr/lib** and **/usr/local/lib**

- **.so** (shared object) dynamic library.
- **.a** (archive) static library.

Binaries conform to **Application Binary Interface (ABI)**

- Compiled for specific architectures. Intel, AMD, MIPS etc.
- Faster than byte-code, although JIT.

Declaration *header file* is required by the compiler to interpret the library. This tells the compiler how to call a given function.

4.2. The Make Utility

make automates the process of dependency checks in projects. This is extra useful when a project has multiple source files and headers.

A **Makefile** depicts a collection of dependency rules.

```
target: dependencies
       action
```

4.2.1. Makefile Example

```
# This is a Makefile comment
CC=g++
# the compiler
LIBS=-lm -lX
# the libraries we will ref
# Need object files file1.o and file2.o to create exe proggy
proggy: file1.o file2.o
$(CC) file1.o file2.o -o proggy $(LIBS)
# Need file1.cpp and file1.h to create object file file1.o
file1.o: file1.cpp file1.h
$(CC) file1.cpp -c
# Need file2.cpp and file2.h to create object file file2.o
file2.o: file2.cpp file2.h
$(CC) file2.cpp -c
# other rules; invoked by make clean etc
clean:
@rm -f *.o
install:
@mv proggy ~/bin
```

5. C++ Pre-processor

The **pre-processor** modifies source code prior to compilation. These *directives* are introduced by **#**

Common Uses

```
#include <filename>    // include files
#define MY_VALUE 1     // define macros
#pragma once           // set compiler behaviour
```

Used to *optimize*, *target platforms* and *compile* certain parts of code.

5.1. Include Files

The **#include** directive "*inserts*" the indicated file at the point of the **#include**. It is a textual insertion which modifies the current file before compilation.

Include Convention

```
#include <filename>
#include "filename"
```

1. Searches default dirs (**/usr/include**)
2. Searches explicit include dirs (*-I/usr/local/matlib/include)

5.2. Pre-processor Macros

Macros are defined with *#define* and must preserve C++ syntax.

Can be a function or constant and should be defined with Upper-case names.

```
#define MYINT 22
#define MYSQR(x) ((x) * (x))
```

5.2.1. Conditional Macro Expansion

#if, #ifdef, #ifndef

```
#if MYVAL==4 // define f() for 4
string f(void) { return string("four"); }
#elif MYVAL==3 // define f() for 3
string f(void) { return string("three"); }
#else // define default f()
string f(void) { return string("fruit"); }
#endif
```

Platform Specific Code

```
#ifdef _USING_WINDOWS // windows specific code
string overlord(void) { return string("gates"); }
#elif _USING_MACOS //macos specific code
string overlord(void) { return string("jobs"); }
#endif
```

5.3. Avoiding Multiple File Inclusion

```
// Header file name: dog.h
#ifndef _DOG_H
#define _DOG_H
```

```
// stuff to include goes here, function declarations
void do_bark(dogtype dog);
#endif // Matches #ifndef _DOG_H
```

5.4. Macro String Operations

Various string manipulation operations can be declared through the pre-processor.

```
#define STR1    "A"
#define STR2    "J"
#define STR3 STR1 STR2     // STR3 now compiled to "AJ"
```

6. C++ Types

Three Kinds

- Simple
- Aggregate
- Class

6.1. Simple Types:

- **char** - 8-bit integer value.
- **int** - standard system integer.
- **float** - system single precision float.
- **double** - system double precision float.
- **short/long/long long** - short (half)/long (double) integer.

Signed vs Unsigned

- **bool** - 8 bit integer value. 0 converts to *false* and non-zero value converts to *true*.
- **unsigned char c** - u_char may also be defined.
- **std::size_t** - standard unsigned integral type.

Simple Type Sizes are System

```
cout << "System long size=" << sizeof(long) << " bytes.";
```

6.2. Integral Types

Type	int	long	long long
16-bit OS	16-bit	32-bit	n/a
32-bit OS	32-bit	32-bit	64-bit
64-bit OS	32-bit	64-bit	64-bit

Modern Alternative to *sizeof()

```
#include <limits>
```

```
...
std::cout << "long size="
        << std::numeric_limits<long>::digits;
```

cstdint header provides *int32_t*, *int64_t*, *uint32_t*, *uint64_t* types.

6.3. Aggregate Types

6.3.1. Structures

Groups data into a **record**.

Ancestor of the *class*.

Notes

- All data and methods are **public**.
- Helps with backward compatibility with C.
- Use a class if you really want a class!

Declaration

```
struct DataEntry{
   int IdNumber;
   char name[40];
   char address[300];
}; // REMEMBER THE SEMI-COLON
```

We can now use **DataEntry**

```
DataEntry d1;
cout << "Name is: " << d1.name << endl;
d1.IdNumber = 1048576;
```

Singleton Structure (no **struct** name)

```
struct { int a; } s1;
```

Shallow Copy

Shallow Copy (byte-by-byte) with Assignment Operator=

```
DataEntry d1 = d2;
```

(shallow copy means pointers will not be accessed)

6.3.2. Enumerations

A set of named integer constants.

Declaration

```
enum name {label_1, ..., label_n};
```

First integer will be zero, increment for each label.

Specific Mapping

```
enum DaysOfWeek {Sun=1, Mon, Tues, Wed, Thur, Fri, Sat};
```

DaysOfWeek is now a valid time.

```
DaysOfWeek dd;
if (dd == Fri) cout << "It's Friday!" << endl;
```

Enumeration type is **not int**.

Enumeration scope is global or class.

When using a class enum outside of class

```
MyClass::DaysOfWeek x = MyClass::Sun;
```

6.4. Class Types

6.4.1. Declaration

C++ separates method code from class declaration.

Declared with *class* keyword within header file (**.h**).

```
#ifndef PERSON_H
#define PERSON_H
#include <string>

class person{
private:             // private members
   std::string n;
public:              // public members
   person(std::string name); // constructor
   void set_name(std::string name); // setter
};  // NB! semi-colon
```

6.4.2. Implementation

Implement methods in source file (**.cpp**).

```
#include "person.h" //incl class declaration
// implementation of methods

person::person(std::string name) : n(name) {}  // constructor
void person::set_name(std::string name) { n = name; } // member function
```

:: (scope operator) and class name associates *declaration* + *definition*

7. C++ Variables

7.1. Variable Initialises

Simple Variables

```
float a = 0.4534534e-10;      //simple vars
int b[5] = { 0, 1, 2, 3, 4 }; //arrays
```

Structure (field by field)

```
struct Name { char a; int numbers[3]; float t; };
Name tt = {'A', {1,2,3}, 0.5};
```

Brackets (multi-value fields)

```
int myarray[3][3][2] = {
  { {1,2}, {3,4}, {5,6} },
  { {7,8}, {9,10}, {11,12} },
  { {13,14}, {15,16}, {17,18} }
};
```

Only works for **Plain Old Data (POD)**

Class types initialised with a constructor.

7.2. Variable Qualifiers

Qualifier Name	Qualifier Description
extern	variable defined outside current scope
static	variable bound to class
const	value cannot be changed after initialisation
register	suggests that compiler use CPU registers to store variable
volatile	variable protected from compiler optimisations

7.3. Type Conversion

C++ is a **strong type checker**.

Automatic Casts Expressions/assignments, function params, class.

```
int i = 1;
std::printf(i);
```

Explicit

```
float x = 4.0f
int i = (int)(x) + 2;  // old style
int j = int(x) + 2;    // new style
```

Old style required for **unsigned char** and **long long**.

Type conversion may be unsafe so the compiler attempts to limit it.

eg. **int** to **short** and back to **int** loses 2 bytes of data.

7.4. Type Definitions

Creates a **new type name** from an old one. Allows for simpler code. Also obeys the scoping principles.

Examples

```
// type u_char is an unsigned char
typedef unsigned char u_char;
```

```
std::vector<float> vec;
typedef std::vector<float>::const_iterator it;
it i = vec.begin(); // I'm not typing that out again
```

7.5. Overflow Errors

Operations can result in variables exceeding their maximum values which won't result in a compile or runtime error. Be careful to check these logical errors.

Recall a **char** can take on values in the range **-128** to **127**

```
char c = 0;
cout << int(c) << "\n";
c += 100;                   // Still Good
cout << int(c) << "\n";
c += 100;                   // Woops we're overflowin
cout << int(c) << "\n"
```

8. C++ I/O

C and C++ IO are based on *streams*, which are sequences of bytes flowing *in* and *out* of programs.

Input Operations Data bytes flow from an input source (e.g. keyboard, file, network, etc…) in the program.

Output Operations Data bytes flow from the program to an output sink (e.g. console, file, network, etc…)

8.1. iostream library

Stream support is provided in the **iostream** library

This library *overloads* the << operator for *stream insertion* and the >> operator for *stream extraction*.

Provides standard stream objects **cout** (console output) and **cin** (console input)

I/O TYPE	SCOPE	OBJECT
Console	global	cout, cerr, clog, cin
File	instantiated	ofstream, ifstream
Memory	instantiated	ostringstream, istringstream

8.1.1. I/O Code EXAMPLEs

1. Console

```
#include <iostream>
std::string s; double d;
std::cout << "Hello world " << s << ' ' << d << std::endl;
std::cin >> s >> std::ws /* consume ws */ >> d; // Read string+double
```

2. File

```
#include <fstream>
std::ofstream out("output.txt"); std::ifstream in("input.txt");
out << "Hello world " << s << ' ' << d << std::endl;
in >> s >> std::ws /* consume ws */ >> d; // Read string+double
```

3. Memory

```
#include <sstream>
std::ostringstream oss; std::istringstream iss("FooBar 1.234");
oss << "Hello world " << s << ' ' << d << std::endl;
iss >> s >> std::ws >> d;              // Get string+double from iss
std::cout << oss.str() << std::endl; // Print the oss' string.
```

8.1.2. I/O Stream Hierarchy

ofstream + **ostringstream** inherit from **ostream**.

This means base class **ostream &** can bind to **ofstream*/*ostringstream** variables.

8.1.3. I/O Stream Operators

- << appends data to stream object.
- >> removes data from stream object.

Can be *overloaded* for custom types.

```
class point { public: int x; int y; };
```

overload stream output << operator **ostream** is base class

```
ostream & operator<<(ostream & out, const point & p)
   { out << p.x << ' ' << p.y; return out; }
```

overload stream input >> operator **istream** is base class

```
istream & operator>>(istream & in, point & p)
   { in >> p.x >> std::ws /* consume ws */ >> p.y; return in; }
```

streams can now output and input point

```
point origin;
cin >> origin; // >> overloaded for point
cout << origin; // << overloaded for point
```

9. Console I/O

Done via methods and overloaded operators ( )

```
int i; float f;
// Add a string to cout
cout << "Enter an integer and a float: ";
cin >> i >> ws >> f; // Remove an int and a float from cin
```

- **cout** - writes to stdout
- **cin** - reads from stdin
- **cerr** - is mapped to stderr (no redirection)
- **Manipulators** - affects behaviour of stream

Special Characters

SPECIAL CHARACTER	USAGE
`\n`	newline
`\t`	tab
`\NNN`	print char with octal
`\xNNN`	print char with hex
`\a`	beeps

Examples

```
cout << "the end-of-line manip" << endl;
cout << "write numbers as hex" << hex << 45;
cin >> a >> ws >> b; // ws consumes whitespace
// #include <iomanip> for setprecision
cout << scientific << setprecision(8) << 1.342355;
```

We can also write our own manipulators.

1. Reading Console Data

   >> operator is *overloaded* to support different data types.

   Example

```
string mystring;
cin >> mystring; // Reads till whitespace
float f;
cin >> f; // Read in a float. e.g. try "1.45e-8"
```

   Other Methods

```
int i;
cin >> hex >> i; // Read in hex. Try 0xAB
cin >> octal >> i; // Read in octal. Try 054
```

2. Reading Lines of Console Data

```
#include <string>
#include <iostream>
// Needed for string objects
// Need for cout and cin
void main(void) {
std::string s;
std::getline(std::cin, s,'\n'); // Reads till '\n'.
std::cout << s << std::endl; // Output the input
}
```

   Why its Powerful

```
int i; float f;
// Assume user types "4 4.2 hello world"
cin >> i >> f; // Consumes the "4 4.2"
cin >> i >> f; // Tries to consume "hello world". FAILS
```

3. Reading from cin

```
    vector<string> items;
    string s;
    while (!cin.eof()) {
      cin >> s >> ws;
      items.push_back(s);
    }
```

9.1. Memory-based I/O

Formats strings or read from strings in memory.

Useful for converting non-string types to string.

Use **stringstream** normally but:

- **strstream** for old **char *** strings. deprecated + buggy.
- **istringstream/ostringstream** depending on input/output.

<u>Example</u>

```
#include <sstream>
int i1, i2; float f1; string str;
string input = "hello 1 2 2.3";
istringstream is(input);
is >> str >> i1 >> i2 >> f1;
```

9.2. File-based I/O

two basic types: input and output file streams.

simple file I/O: **fstream**, **ifstream**, **ofstream**

<u>Example</u>

```
#include <fstream>
ifstream myfile;
myfile.open("file.dat");
if (!myfile)
  { cerr << "File open failed!"; }
```

or

```
ifstream myfile("file.dat");
myfile.close(); // close the file
```

always close files (with destructors is fine)

9.3. Text File I/O

overloading operators (<<, >>) available; derived from **ios**

<u>EXAMPLE</u>

```
int i;
while (!myfile.eof()) {
  myfile >> i >> ws;
  cout << "The next data item is " << i << endl;
}
```

type of input determined by variable and white spaces separates data items (use **ws**)

9.4. Binary File I/O

Example

```
#include <iostream>
#include <fstream>

const int ARRAY_SZ=40;
unsigned char array[ARRAY_SZ];
ifstream myfile("binary.dat", ios::binary);
while (!myfile.eof()) {
  int n = myfile.read(array, ARRAY_SIZE);
  cout << "Data: ";
  for(int i=0; i<n; ++i) { cout << array[i]; }
  cout << endl;
}

outfile.write(array, ARRAY_SIZE); // Binary output
```

10. Scope

What variable/function is a label bound to at a point?

```
class A { int a=1; }; // a in scope of class A
void func1(void) { int a=2; } // a in scope of func1
void func2(void) { int a=3; } // a in scope of func2
while(true) { int a=4; } // a in the scope of code block
```

RULE OF THUMB: Pairs of **{}** define new scope.

Variables defined at **global** scope are visible everywhere.

10.1. Local Variables

A variable defined in an enclosed scope hides one in outer scope.

```
int i = 10; // global/outer scope
for (int j = 0; j < 5; j++)
{ int i = j*2; cout << i << endl; }
```

You can access global scope variables using scope operator, **::**

```
int i = 10; // defined OUTSIDE code/class
for (int j = 0; j < 5; j++)
  { int i = j*2; cout << ::i << endl;}
```

10.2. Lifetime

In C++, lifetime of automatic variables is bounded to scope. Once the scope is left, the variables defined within it are destroyed.

10.3. Namespaces

Large projects may result in name clashes. We solve this through a namespace.

- Labels in the namespace are "prefixed" with the namespace.

- No duplicate definitions within namespace.
- Use scope resolution operator **::** to refer to namespace labels.

```
namespace project { int p1; }
namespace projectx { float p1; }
cout << project::p1 << projectx::p1;
```

It's almost like a virtual directory.

11. C++11 New Features

11.1. New Typing Constructs

- **auto** - deduces type of variable from expression on rhs.

```
std::vector<float> vec;
// std::vector<float>::iterator i = vec.begin()
auto i = vec.begin();
```

- **decltype** - deduce type of variable from supplied expression.

```
int an_int;
// Figure out the type of this
// variable at compile time.
decltype(an_int) another_int = 5;
```

11.2. Initializer Lists

Extends Initializer Lists to work with non-POD constructs.

```
std::vector<int> a = { 0, 1, 2, 3, 4 };
```

Constructor takes a **std::initializer_list<type>** variable.

```
class MyList {
  public:
    MyList(const std::initializer_list<int> & rhs)
    {
      for(auto i = rhs.begin(); i != rhs.end(); ++i)
      { /* construct MyList */ }
    }
}
```

12. C++ Program Memory

C++ allows programmers to directly access, manipulate and manage memory.

Program Memory

Level 1	Code
Level 2	Global variables/ Static data
Level 3	Freestore or Heap
Level 4	Stack

Local variables are allocated to the **stack**

Dynamic variables are allocated to the **heap**

13. Pointers

Copying large object is slow. *Pointers* allow us to avoid extra copies

- Copy object's memory address around.
- 2^{16} = 64KB. 16-bit integer for memory address range.

Array Analogy A program's memory can be compared to a huge array of bytes. Index into the array is a pointer. Usually written in *hex*: 0XFFFFFFA0 (32-bit address)

13.1. Declaring Pointers

A **pointer** variable holds a **memory address**

```
type * ptrname;
```

Initializing to memory address with **&**

```
int a = 5;
int * ptr = &a;
```

Initialize default ptr value to **nullptr**

```
// keyword in C++11
int * p1 = nullptr;
// C++03 MACRO (0x0)
int * p2 = NULL;
```

13.2. Dereferencing Pointers

Value at address obtained using * operator

```
int * ptr = 0xFFFFFFA0; // Assign random address. BAD!
int i = *ptr; // Dereference to inspect value at random address.
cout << "int value @ address "
     << static_cast<void *>(ptr) << "=" << i << endl;
*ptr = 5;      // Dereference to set value.
```

Dereferencing members of Struct/Class Pointers

Class:

```
class binary_tree_node
{
public:
  float node_func(void) { return f; }
  binary_tree_node * left, * right;
  float f;
};
binary_tree_node n; binary_tree_node * node = &n;
(*node).f = 1.0f;
```

or even prettier

```
node->f = 1.0f;
node->left = node->right = NULL;
```

```
cout << node->node_func() << endl;
```

13.3. *Types of Pointers*

13.3.1. Generic Pointers

Can point to any type, but cannot be directly dereferenced, one must cast *explicitly*.

```
// assume this returns a pointer
void * ptr = GetAddress();
float * fptr = static_cast<float *>(ptr);
```

Functions can both receive and return void pointers. Don't do it though because it can get confusing. All pointers have the same size anyway (Architecture dependent).

Max addressable memory for 32-bits is 4GB.

13.3.2. Function Pointers

The function **name** is a pointer to code in memory.

```
// Declare a Function Pointer type, binfuncptr
typedef int (*binfuncptr)(int,int);
// Functions matching binfuncptr’s signature
int add(int a, int b) { return a+b; }
int subtract(int a, int b) { return a-b; }
// Applies function ptrs of type binfuncptr
int apply(binfuncptr ptr, int a, int b) { return ptr(a, b); }

int main(void) {
  cout << apply(add, 5, 3) << endl;
  cout << apply(subtract, 5, 3) << endl;
  int (*fptr)(int,int) = add; // No typedef
  fptr(5,3);
}
```

13.4. *Pointer Arithmetic*

Access array of memory using pointer arithmetic.

```
char a[4] = {’D’,’O’,’G’,’E’};
char * ptr = a; // Arrays are
for(int i=0; i<4; i++) // pointers!
  { cout << *(ptr+i) << endl; }
```

Step size is deduced from type.

```
for(char * p=a; p != a+4; ++p)
  { cout << *p << endl; }
```

Array indexing is still a lot nicer.

```
for(int i=0; i<4; ++i)
  { cout << ptr[i] << endl; }
```

13.5. *Pointer Indirection*

Pointers to pointers

```
// pointer to a char pointer
char * cptr = nullptr;
char ** ccptr = &cptr;
char *** cccptr = &ccptr;
```

add a * for every level of indirection

Example

```
char ch = 'a';
char * cptr = &ch;
char ** ccptr = &cptr;
char *** cccptr = &ccptr;

// What does this print?
cout << *cptr << **ccptr
  << ***cccptr << endl;

*cptr = 'b'; **ccptr = 'c';
***cccptr = 'd';

// What does this print?
cout << ch << endl;
```

13.6. Pointers as Function Arguments

Old style of reference passing. Pass reference instead of copying variable into function argument.

```
big_class * func(big_class * object, node_type ** node)
{
  // Access the rather large object
  dostuffwith(object->large_value);
  // Change the address of the supplied pointer!
  *node = object->node;
  return object;
}
```

Avoids copy of large value but is fairly legacy.

14. References

A reference serves as an alternative name for the object with which it has been initialised.

```
int x = 10;  // 'plain' int
int& r = x;  // reference
```

- **x** is an ordinary *int*
- **r** is a reference, initialised with **x**, no new memory is allocated

15. Functions

15.1. Pass-by-value

```
int add(int a, int b)
```

Two new local variables a and b are created and are assigned the incoming values. They are destroyed once the function completes.

15.2. Pass-by-reference

```
int add(int& a, int& b)
```

&a creates a reference to an int. It becomes an alias to the existing integer variables/values.

15.3. Overloading

Multiple function signatures with the same name.

```
double add (int a, int b)
double add(double a, double b)
```

16. Dynamic Memory Allocation

C++ does **not** have a garbage collector.

Dynamic variables (created using **new**) allocated/deallocated by programmer (Local or automatic variables managed by the runtime environment using scope).

Address of Dynamic Memory stored in a pointer.

```
// Allocate and deallocate single object
myobject * myobjptr = new myobject;
delete myobjptr;
// Allocate and deallocate array of objects
int * intptr = new int [30];
delete [] intptr; // NB! Brackets
```

- **new** - invokes constructors
- **delete** - invokes destructors

16.1. Stack vs. the Heap

Dynamic Memory is acquired from the **heap**.

Local Variables and Arguments live on the **stack**.

16.2. Dynamic Arrays

We can also create dynamic arrays with **new**

```
// 2 rows, different column sizes for each row
int rows = 2; int cols[2] = { 3, 2 };
float ** array = new float*[rows]; // Allocate array of float *'s
for (int i=0; i<rows; ++i) {
  // Allocate float array, current row determines size
  array[i] = new float[cols[i]];
  // Initialise the array with float values
  for(int j=0; j<cols[i]; ++j)
    { array[i][j] = float(i*j+1); }
  }
}

if (array[k][l] == 2.0 ) { /* do stuff */ }

for(int i=0; i<rows; ++i)
  { delete [] array[i]; } // Delete the inner arrays
```

```
delete [] array;
// Delete the outer array
```

17. Resource Acquisition is Initialisation (RAII)

C++ Memory Model pairs *Object Construction* and *Destruction* within **same scope**.

Important for guaranteeing *exception safety*.

17.1. Automated Pointer Management

Dynamically Allocated Memory isn't managed by the RAII paradigm.

<u>Problem</u>

```
int main(void) {
  student * ptr = new student;
  if(!ptr->invoke(1))
    throw dark_lord_exception();
  delete ptr; // doesn't get called if throw occurs!
}
```

<u>Solution</u> (encapsulate and guarantee pointer release)

```
class student_ptr {
private: student * ptr;
public:
  student_ptr(student * p) : ptr(p) {}
  ~student_ptr(void) { delete ptr; }
};
```

17.1.1. unique_ptr

Wraps a pointer in **automatic** variable.

Automatically deletes pointer when it leaves scope.

Zero extra overhead

```
#include <memory>
int main(void) {
  std::unique_ptr<student> ptr(new student);
  if(!ptr->invoke(1)) // Exact same pointer semantics
    throw dark_lord_exception();
} // Allocated pointer automatically cleaned up
```

1. <u>Usage Patterns</u>

   - Acquire allocated memory, obtain raw pointer, release

   ```
   std::unique_ptr<int> A(new int(10));
   int * ptr = A.get() // Return raw pointer
   A.release();        // Releases (deletes) held pointer
   ```

   - Exchange for new pointer

   ```
   std::unique_ptr<int> B(new int(20));
   B.reset(new int(30)); // Release held pointer, replace with new
   ```

   - Acquire allocated memory array, use subscript

   ```
   std::unique_ptr<int []> C(new int[10]);
   ```

```
std::cout << C[5];   // Subscript operator for arrays
```

2. <u>Unique Ownership</u>

   Unique pointers cannot be copied, only **moved**. The copy operator= is deleted.

```
std::unique_ptr<int> lhs(new int(10)); // lhs.get() != nullptr;
std::unique_ptr<int> rhs(new int(20)); // rhs.get() != nullptr;
lhs = std::move(rhs);  // Can't lhs = rhs;
// lhs.get() != nullptr && *lhs == 20;
// rhs.get() == nullptr;
```

   - **lhs**'s pointer is released (deleted).
   - **rhs**'s pointer is copied to lhs.
   - **rhs**'s pointer is NULLED.

   Only **one** unique_ptr can be *responsible* for a pointer.

17.1.2. shared_ptr

- When **shared_ptr** is *copied/copy constructed*, ref **count** incremented.
- When **shared_ptr** is *destroyed*, ref count decremented.

If **count** reaches 0, managed pointer *deleted*.

Extra overhead from *pointer indirection* and *count maintenance*. Use when lifetime of allocated object is uncertain.

1. shared_ptr cycles

   Shared pointers can result in **cycles**.

```
class node {
public:
std::shared_ptr<node> next;
};
...
{
shared_ptr<node> A = make_shared<node>(); // Count of 1
shared_ptr<node> B = make_shared<node>(); // Count of 1
shared_ptr<node> C = make_shared<node>(); // Count of 1
A->next = B; B->next = C; C->next = A; // Cycle on last =
  // Counts of A, B and C are now 2
} // Destructors of A, B, and C called. BUT
  // Internal shared_ptr counts are now 1,
  // The object should not really exist
```

17.1.3. weak_ptr

weak_ptr's point to **shared_ptr**'s but they don't increment/decrement the count.

<u>USE</u>

- to break cycles.
- create links.
- point to allocated memory without asking for responsibility.

```
shared_ptr<node> A = make_shared<node>();  // Count is 1
```

```
weak_ptr<node> B(A);                          // Count is 1
if(shared_ptr<node> C = B.lock()) {
  // Count is now 2. Use shared_ptr.
  // Count decremented when block closes (RAII)
}
// Count is back to 1
```

New Solution

```
class node {
public:
std::weak_ptr<node> next;
};
...
{
  shared_ptr<node> A = make_shared<node>();
  shared_ptr<node> B = make_shared<node>();
  shared_ptr<node> C = make_shared<node>();
  A->next = B; B->next = C; C->next = A; //
  // Counts of A, B and C are 1
} // no leaks when shared_ptr's leave scope
```

Now you can:

```
C->next = shared_ptr<node>(); // Set to nullptr
for(shared_ptr<node> head=A; ;head=head->next.lock()) {
  /* Do stuff and set the quit variable at some point */
  if(head->next.expired()) break;
}
```

Usage Hint

- **unique_ptr** for mandating sole responsibility of held pointer.
- **shared_ptr** for mandating shared responsibility for shared pointer.

18. Values and Reference Semantics

Value Semantics	The **value** of the object is important, not the **identity**
Reference Semantics	The **identity** of the object is important, not the **value**

C++ has both **value and reference** semantics while Java only has reference.

18.1. L-Values and R-Values

l-values persist.

r-values do not persist (think RAII).

```
Matric multiply(Matrix lhs, Matrix rhs)
  { return lhs * rhs; }
Matrix A, B, C, D;
A = B + C + D;
B = Matrix(1.0, 2.0, 3.0, 4.0)
C = multiply(A,B)
```

L-Values **A**, **B**, and **C**

R-Values **B + C + D**, **Matrix(1.0,2.0,3.0,4.0)**, **multiply(C,D)**

18.2. L-Value & vs R-Value References

l-value references (&) bind to named variables.

```
Matrix A;
Matrix & Aref = A;
```

r-value references (&&) bind to unnamed, temporary variable.

```
Matrix multiply(Matrix lhs, Matrix rhs)
  { return lhs * rhs; }
Matrix B, C, D;
// Binds to unnamed temporary holding result of B + C + D;
Matrix && A = B + C + D;
// Binds to result return value of multiply.
Matrix && A = multiply(B, C);
```

We can *move///steal* these variable's **values** before their destruction.

18.3. Values vs Reference Semantics

18.4. Java vs C++

Java has *simple* and *object* types. To the java compiler: an object is a reference! Syntax is exactly the same for simple types.

```
class A { // Java code
   // Takes reference
   // to c and value of i
public void f(C c, int i)
    { c.invoke(i); }
}
//...
A a = new A(); // Allocate
C c = new C(); // on heap
int i = 50;
a.f(c,i);

class A { // C++ code
  // Takes *reference*
  // to c and *value* of i
public: void f(C & c, int i)
    { c.invoke(i); }
};
//...
A a;
C c;
int i = 50;
a.f(c,i);
```

C++ syntax implies value semantics **by default**.

Java syntax implies reference semantics (except for simple types).

19. Implementing RAII and Value Semantics

19.1. Six Special Member Functions

1. Default Constructor
2. Copy Constructor
3. Move Constructor
4. Copy Assignment Operator
5. Move Assignment Operator
6. Destructor

The compiler creates these even if you don't.

You can explicitly for **defaults**

```
class student {
public:
  student(void) = default; // Default constructor
  student(const student & rhs) = default; // Copy Constructor
  student(student && rhs) = default;
  student & operator=(const student & rhs) = default; // Move Constructor
  student & operator=(student && rhs) = default; // Copy and Move Assignment
Operators
  ~student(void) = default; // Destructor

  std::string name; // Name
  std::vector<std::string> potions; // Vector of potions
};
```

Or disallow them with **delete**.

```
student(void) = delete;
```

19.2. Defined Behaviour

1. Creation
2. Copying (deep)
3. Moving
4. Cleanup

19.3. Rule of Five

One should manually implement these functions if the class manages special resource.

<u>If we define for one of these:</u>

1. Copy or Move Constructor
2. Copy or Move Assignment Operator
3. Destructor

Then we should probably define all five.

19.3.1. Default Constructor

Sensibly construct an object with no arguments.

```
class student {
public:           // Initialiser list
  student(void) : name("Harold Potter"), wand(acquire_want())
    { set_charges(wand, 100); }      // Constructor body

  std::string name;                  // Name
  int wand;                          // Unique Wand
};
```

This important for arrays which use default constructors.

```
student harry;   // Default constructor called
student h[3];    // Default constructor called 3 times
student hogs[3] = { student(), student(), student() };
```

Always try use initialiser list instead of a constructor body in constructors.

```
: name("Harold Potter"), wand(acquire_wand())                // YES
  { name = "Harold Potter"; wand = acquire_wand(); }          // RATHER NOT
```

=

```
class student {
  public student(void)
    { name = new String(); name = new String("Harry"); }
}
```

C++ *auto* vars must be constructed while Java can have nulled refs

Member vars always constructed in initialiser list

We can also supply default arguments to this constructor and functions in general.

```
student(int charges=50) : name("Harold Potter"),
                          wand(acquire_wand())
  { set_charges(wand, charges); }       // Constructor body
```

19.3.2. Destructor

Release resources managed by an object.

Invoked at end of scope {} (*deterministic* cleanup)

Java **finalize** is similar but non-deterministic.

```
class student {
public:
  ~student(void) {
    if(wand != -1) {  // Release if not null
      set_charges(wand, 0);      // Empty ammo
      release_wand(wand);        // Release it
    }
  }
  std::string name;              // Name
  int wand;                      // Unique wand
};
```

RAII will automatically call destructors of name, freeing the memory.

But the wand value (resource) must be manually released.

1. RAII Destructor Comparison

```
{
   student potter, malfoy;
   int wand = -1;                        // Start out empty
   wand = acquire_wand();                // Explicitly acquire
   set_charges(wand, 100);
   potter.zap(wand, malfoy);             // Use
   set_wand_charges(wand, get_charges(wand)-1);
   if (wand != -1) release_wand(wand);  // Explicitly release
 }
```

vs **wand resource wrapped** by class

```
{
   student potter, malfoy; // wand acquired in def constructor
   potter.zap(malfoy);
}                          // wand released by potter destructor
```

We achieve **RAII** functionality for wand resource by wrapping in a class.

The **Class has responsibility** for the resource, **Zero-overhead**.

19.3.3. Copy Constructor

- Constructs by copying another object.

```
student harry1;            // Default constructor called
student harry2 = harry1;   // Copy constructor invoked
student harry3(harr2);     // Alternate Copy Constructor syntax
```

- Takes one argument, **constant L-value ref** to object of same type:

```
class student {
public:
  student(const student & rhs) : name(rhs.name), wand(/* ? */)
    { set_charges(wand, get_charges(/* ? */)) }

  std::string name;             // Name
  int wand;                     // Unique Wands
```

19.3.4. Move Constructor

Constructs by moving **resources** from another objects, **rhs**. **Rhs** is useually temporary and about to be destroyed.

- Must leave **rhs** in a **destructable** state

```
student old_harry;    // Default constructor called
student new_harry = std::move(old_harry);  // Obtain r-val ref to l-val
so move kicks in
```

- One argument, **r-value ref** to object of same type

```
class student {
public:
  student(student && rhs) : name(std::move(rhs.name)),    // Move
constructor
      wand(rhs.wand),  // std::move(int) just copies anyway.
  { rhs.wand=-1; }         // We've taken rhs' wand so the destructor
won't try and release
```

```
  std::string name;       // Name
  int wand;               // Unique wand
```

19.3.5. Copy Assignment Operator

Copies contents of cone object to another (releasing existing resources).

Overloads the = operator which does different things for each type.

- Differentiate from Copy Constructor

```
  student h1;              // Default constructor called
  student h2 = h1;         // Copy constructor invoked
  student h3(h2);          // alternative Copy Constructor syntax
  h1 = h3;                 // Copy Assignment Operator invoked
```

A **combination of destructor + copy constructor**.

- One argument, **constant L-value ref** to class type

```
  student & operator=(const student & rhs) {
    if(this != &rhs) { // Optimisation, ignore for now
      name = rhs.name;            // Defer to copy operator=
      int new_wand = acquire_wand();                            // Acquire
      set_charges(new_wand, get_charges(rhs.wand));             // Acquire
      if(wand != -1) release_wand(wand);                        // Release
      wand = new_wand;                                          // Assign
    }
    return *this;  // Return a reference to the current object.
  }
```

19.3.6. Move Assignment Operator

Moves contents of one object to another and **releases the existing resources**.

Also overloads the = operator.

Differentiate from the Copy Constructor

```
student h1;           // Default constructor called
student h2 = h1;      // Copy constructor invoked
student h3(h2);       // alternative Copy Constructor syntax
h1 = std::move(h3)    // Move Assignment Operator invoked
```

A combination of destructor + move constructor.

One argument, **r-value ref** to Class Type

```
student & operator=(student && rhs) {
  if(this != &rhs) {  // Optimisation, ignore for now
    name = std::move(rhs.name);           // Defer to move operator=
    set_charges(wand, 0);                 // RELEASE held resource
    if(wand != -1) release_wand(wand);    // RELEASE held resource
    wand = rhs.wand;                      // Take rhs' resource
    rhs.wand = -1;                        // Make rhs' resource null/empty
  }
return *this; // Return a reference to the current object
}
```

19.4. Const Correctness

- **const** keyword specifies whether a variable may be modified.

```
const student potter("Harry");
```

- **const** methods will work on both **const** and **non-const** objects + refs.

```
class student {
public:
  std::string name;

  void set(const std::string & n) { name = n; };
  std::string get(void) const { return name; };
}
```

then,

```
potter.set("Hermione");             // Compiler complains
std::string name = potter.get();    // Succeeds
```

- const references are often used to specify read only access to objects.

```
std::ostream & operator<<(std::ostream & out, const student & s) {
  out << s.get();
  s.set("Hermione");     // Const problemo
  return out;
}
```

contrast with

```
std::istream & operator>>(std::istream & in, student & s) {
  std::string name;
  in >> name;
  s.set(name);      // This will work since s is not a const
  return in;
}
```

This defines a sort of **read/write interface** on your class methods.

19.5. Function Arguments

- Pass by **constant L-value ref** if only read.

```
void print_names(const student & s) { cout << s.get() << endl; }
```

- Pass by **L-value ref** if you modify arg for some reason.

```
void hermione_it(student & s) { s.set("Hermione"); }
```

- Pass by **value** if you're going to make a copy anyway.

```
void duplicate(const student & s)
{ student new_s = s; /* do stuff */ }
...
void duplicate(student s) { /* do stuff */ }
```

Pass by **value** may become the standard way of doing things since *move semantics + copy elision* eliminate unnecessary copies.

19.6. Function Return Values

- Constructing a new object, return by **value**.

```
student harry_factory(void) {
  student harry("H. Potter"); harry.set_wand_charges(500);
  return harry;
}
student h = harry_factory();
```

Compiler optimises implied copy away, or *move constructs* h.

- Can return **ref** to a ref argument.

```
std::ostream & operator<<(std::ostream & out, const student & s)
  { out << s.get(); return out; }
```

- Can return **const ref** to class members.

```
class student {
  std::string name;
  const std::string & get_name(void) const { return name; }
};
```

20. Containers and Iterators

20.1. Containers

Containers hold elements (objects/simple) **TEMPLATED**.

```
#include <vector>     // resizable array, with random access
vector<int> V(3);     // create with 3 default ints, else empty

#include <list>       // linked list. O(n) access
list<Animal> A;       // empty list of Animals

#include <set>        // holds unique values
set<int> S;           // red-black tree O(log n) access

#include <map>        // ordered associative mapping: key -> data
map<string,int> M;    // red-black tree. O(log n) access
M["zebras"] = 3;      // associate string with int

#include <unordered_set>
unordered_set<int> S;           // set backed with hash table

#include <unordered_map>
unordered_map<string, int> m;    // assoc map backed with hash table
```

Operator overloading is very useful for clean code.

20.2. Iterators

Containers are **heavyweight** and may contain lots of data. **Iterators** are **lightweight** and carry references to a data element within a container.

- This allows us to easily move between container elements.

```
vector<int> data = { 6, 8, 2, 4, 0 };
for(vector<int>::const_iterator i = data.begin(); i != data.end(); i++)
  { cout << *i << endl; }
for(auto const & ref : data)
  { cout << ref << endl; }
```

20.2.1. Types

Name	Description
vector::iterator	stores a pointer
list::iterator	pointer to node object which has next and previous pointers
set::iterator	pointers to left and right children, pointer to node parent probably

20.2.2. Iterator Access

Iterators obtained via container *begin()* and *end()* methods. *End()* is iterator pointing at **logical container end**. This doesn't referance data buf rather identifies when we've iterated through all container data.

```
for(vector<int>::const_iterator i=v.begin(); i!=v.end(); ++i)
```

- Move **forwards with ++i** and **backwards with –i**.
- Move **multiples with std::advance(i, n)**.
- *i dereferences the interator to gain access to container elements.

```
int value = *i;     // Notice pointer
*i = 6;             // Semantics again
```

- **operator==** and **operator!=** overloaded.

```
vector<int>::const_iterator i=v.begin();
vector<int>::const_iterator j=v.begin();
i == j; i != j;
```

21. Nested Classes

Define classes within the namespace of another class. Similar to **Java static inner classes only**.

```
class outer {
public:
  class nested {
  public:
    void print(const outer & rhs) const
      { cout << "Secret is " << rhs.secret << endl; }
  };
private:
  string secret;
};
```

- Nested class can access outer class's private members.
- Outer class must be passed into the inner class for this to happen.

22. Operator Overloading

Overloading operators allows for cleaner code.

- Any class member function can be overloaded (except destructor).
- Operators can be overloaded (given a new interpretation).
- **(), [], new, delete** and also be overloaded.

22.1. Associativity

Redefined operators retain precedence and associativity.

- **operator<<, operator+** -> Left Associative

```
// (cout << a) << b;
// operator<<(operator<<(cout, a), b);
cout << a << b;
// (a + b) + c;
// operator+(operator+(a, b), c);
a + b + c;
```

- **operator=, operator+=** -> Right Associative

```
// a = (b = c);
// operator=(a, operator+=(b, c));
a = b = c;
// a += (b += c);
// operator+=(a, operator+=(b, c));
a += b += c;
```

22.2. Standalone Functions

Can overload via standalone functions.

```
Matrix & operator+=(Matrix & lhs, const Matrix & rhs)
  { /* implement lhs += rhs */; return lhs; }
Matrix operator+(const Matrix & lhs, const Matrix & rhs)
  { Matrix result = lhs; result += rhs; return result; }

Matrix A, B, C;
C = A + B;    // calls operator+(A,B)
C += A;       // calls operator+=(C,A)
```

operator+ and **operator+=** need access to Matrix internals, it **must friend them**.

22.3. Class Member Functions

Can overload via class member functions.

```
Matrix & Matrix::operator+=(const Matrix & rhs)
  { /* implement *this += rhs; */ return *this; }
Matrix Matrix::operator+(const Matrix & rhs) const
  { Matrix result = *this; result += rhs; return result; }

Matrix A, B, C;
C = A + B;   // calls A.operator+(B)
C += A;      // calls C.operator+=(A)
```

- **operator+** is *const* because object is not modified.
- **operator+=** is *non-const* because object is modified and reference returned.
- automatic access to class internals.

- object is **always the lhs** argument (compared to standalone functions).

22.4. Contextuality and Unary Overloads

Operand Types determine which overloaded operator is called.

- Matrix Scalar Multiplication and Matrix Product

```
Matrix operator*(double lhs, const Matrix & rhs); // prefix
Matrix operator*(const Matrix & lhs, double rhs); // postfix
Matrix operator*(const Matrix & lhs, const Matrix & rhs);
```

- Unary (Single Component) Operator overloads take no arguments

```
Matrix Matrix::operator-(void) const;  // Unary Negation
  { Matrix result = *this; /* negate result */; return result; }
// contrast with Binary Difference
Matrix Matrix::operator-(const Matrix & rhs) const;
```

22.5. Efficiency

Update operators (+=) are generally faster than standard operators.

```
Matrix & operator+=(Matrix & lhs, const Matrix & rhs)
  { /* implement lhs += rhs */; return lhs; }
Matrix operator+(const Matrix & lhs, const Matrix & rhs)
  { Matrix result = lhs; result += rhs; return result; }
```

- **operator+=** modifies in place and returns reference.
- **operator+** creates new objet to hold result.
- **temporaries** to hold intermediate result of **operator+**.

```
Matrix A, B, C, D;
A = B + C + D;     // creates 2 temps, 1 copy.
```

vs.

```
Matrix A, B, C, D;
A = D;   // 1 Copy
A += C; A += B;
```

22.6. R-value References

chaining **operator+** creates **temps** holding intermediate results.

```
Matrix A, B, C, D;
A = B + C + D;        // Might create 2 temps. 1 copy assignment.
```

can optimise with move semantics (r-value refs)

```
Matrix operator+(Matrix && lhs, const Matrix & rhs)
  { lhs += rhs ; return std::move(lhs); }
Matrix operator+(const Matrix & lhs, Matrix && rhs)
  { rhs += lhs ; return std::move(rhs); }
```

moves avoid object creation and copying

```
A = B + C + D;     // create 1 temp. 2 moves.
```

22.7. Parenthesis

Overloading parenthesis.

```
double  & Matrix::operator()(int i, int j)
  { return data[i*width + j]; }

Matrix A(2,2);  // constructor
A(0,0) = 0.0;   // parenthesis operator
A(0,1) = A(1,0) = 1.0; // named object, A
```

- **operator()** can take many arguments as is used to define **functors** or function objects.

22.8. Array Subscript

The array subscript **operator[]** takes one argument only;

```
char & charbuf::operator[](int index)
  { return a[index]; }
```

We are able to chain multiple values in complex objects though

```
// Assume internal array of matrixrow objects
matrixrow & Matrix::operator[](int row)
  { return rows[row]; }
// Assume internal array of row data
double & matrixrow::operator[](int col_index)
  { return data[col_index]; }
...
Matrix A;
cout << A[row][col] << endl;
A[row][col] = 1.0;
```

23. Friend Functions and Classes

Object-Orientation requires strict encapsulation of data. C++ however allows cetain non-member functions to access class internals. This provides a way to get around limitations (speed and overloading).

A function or class may be a **friend** of another. This keyword indicates permission is granted by the class. Friend functions are **not** inherited.

23.1. Friend Classes

```
class X {
private:
  friend class BestFriendForever
  std::string secret;
};

class BestFriendForever {
public:
  void gossip(const X & x) {
    cout << "OMG, You'll never believe what I heard about X: "
       << x.secret << endl;
  }
};
```

All member functions of BestFriendForEver can access X's members.

23.2. Friend Functions

```
class X {
public:
  friend void press(void);  // not a class member
private:
  int mybuttons;  // class member
};
void press(const X & x) { ++x.mybuttons; }
```

Key to note that the function definition **has no friend keyword**.

23.3. Stream Operators

Naive attempt

```
ostream & ostream::operator<<(const Matrix & rhs)
  { *this << /* rhs members */; return *this; }
```

Need a standalone friend function

```
class Matrix { // Allow operator<< access to Matrix's privates
  friend ostream & operator<<(ostream & os, const Matrix & M);
}
...
ostream & operator<<(ostream & os, const Matrix & M)
  { os << /* M's privates */; return os; }
...
cout << A << B;  // operator(operator<<(cout, A), B);
```

23.4. Symmetric Operators

Friend functions allows us to define symmetric overloading operators.

```
class Matrix {
public:
  Matrix operator*(double c) const { /* postfix multiply */ };
  // Declare prefix multiply function a friend of Matrix
  friend Matrix operator*(double c, const Matrix & A);
};
...
Matrix operator*(double c, const Matrix & A)
  { /* implement prefix multiply, access A's private members */ }
...
Matrix A, B, C;
double fact = 3.1;
```

operator<< and **operator**>> usually standalone friend functions too.

24. C++ Inheritance

Classes can sub-class existing *parent* or *base* classes. This encourages code re-use.

- C++ has no special keyword (**extends** from Java) and no **super** keyword.
- C++ supports multiple inheritance with multiple parents

```
class A { /* implement */ };
class B { /* implement */ };
// C is a sub-class of A and B
```

```
class C : public A, public B { /* implement */ };
```

24.1. Composition vs Inheritance

24.1.1. Inheritance

If you need to *extend/enhance* class functionality and you want the same basic interface, then **inherit**.

```
class Base {
  int x, y;
public:
  void function1(void);
  void function2(void);
};

class Derived : public Base {
  int z;
public:
  void function3(void);
};
```

We say that *Derived class* **IS-A** *Base object*.

- Wherever we used a **Base object**, we can *always* use a **Derived object**.
- Can redefine inherited methods.
- Can add new function and member variables.

24.1.2. Composition

Compose to *ACCESS* another class's functionality.

```
class Base {
  int x;
public:
  void bfunction(void);
  void setival(int);
};

class NewClass {
  int z;
  Base bobject;
public:
  void set_base_data(int a) { boject.setival(a); }
};
```

- NewClass **HAS-A** Base object within it.
- NewClass isn't required to conform to Base's interface.
- NewClass doesn't have to be extended.

24.2. Static and Dynamic Polymorphism

24.2.1. Static Polymorphism

Resolved at compile-time.

- C++ uses **Static Polimorphism** by *default*.
- C++ can redefine functions in *Derived* classes.
- C++'s zero-overhead principle in action.

24.2.2. Dynamic Polymorphism

Resolved at run-time.

- Java used Dynamic Polymorphism by *default*.
- Explicitly introduced in C++ with **virtual** keyword.
- Also need **reference semantics** (pointers/references).
- C++ Pointers + References to Base objects work similarly to Java refs.

24.2.3. Static vs Dynamic Polymorphism

Static Polymorphism

```
class Base {
public:
  void print(void)
    { cout << "Base" << endl };
};
class Derived : public Base {
public:
  void print(void)
    { cout << "Derived" << endl };
};

Base * b = new Base;
Derived * d = new Derived;
b->print();  // Output "Base"
b->print();  // OUtput "Derived"
```

Dynamic Polymorphism

```
class Base {
public
  virtual void print(void)
    { cout << "Base" << endl };
};
class Derived : public Base {
public:
  virtual void print(void) override
    { cout << "Derived" << endl };
};
// Upcast new object pointers to base class
Base * b = dynamic_cast<Base *>(new Base);
Base * d = dynamic_cast<Base *>(new Derived);
b->print();  // Output "Base"
d->print();  // Output "Derived"
```

24.2.4. Cast Operators

- **static_cast** performs casting at compile time.

```
double value = 1.45;
double remainder = value - static_cast<int>(value);
```

- **dynamic_cast** casts to non-equivalent type using run-time check. Use to *upcast* and *downcast* between polymorphic types.

```
Base * b = dynamic_cast<Derived *>(new Derived);
```

Dynamic Cast Failes

Casted Type	Action
pointer	returns *nullptr*
reference	throws *std::bad_cast*

24.3. Constructors for Inherited Classes

A child class has to correctly initialise its parent (this is done using the initialiser list).

```
class Base {
public:
  int x, y;
  Base(int x, int y) : x(x), y(y) {}
};
class Derived : public Base {
public:
  int z;
  Derived(int x, int y, int z) : Base(x,y), z(z) {}
};
```

24.4. Accessing Base Members and Functions

Inheritance can hide (*override*) base class variables (*functions*).

uses

operator

```
class Base {
public:
  int aaa;
};
class Derived : public Base {
public:
  int aaa;
  void print(void) {
    cout << aaa << Base::aaa << endl;
  }
};
```

Special member functions and friends are **not** inherited.

24.5. Access Control

- **private** members are not inherited.
- **protected** members are inherited, but not visible outside class.

24.5.1. Access Declarations

C++ has 3 levels of access control which can modify inherited access. This overrides the inheritance access spec using **access declaration**.

Level	Public	Protected	Private
public	remain public	remain protected	remain private
protected	become protected	remain protected	remain private
private	become private	become private	remain private

```
class Base {
protected:
  int vprot;
public:
  int prot;
};

class Derived : public Base {
protected:
  Base::prot;  // access declaration prot noew protected;
};
```

Java provides public inheritance only.

24.6. Virtual Functions and Dynamic Binding

Virtual (class) functions allow dynamic binding (run-time binding). Java supports dynamic binding exclusively; C++ usually binds statically.

- **syntax**: *virtual ret_type FuncName(args);*

A **virtual function** *must have the same signature* in every sub-class. You do not need a virtual keyword in sub-classes though.

Constructors *cannot* be **virtual**. Destructors *should* be **virtual** for a **dynamically polymorphic** class.

```
class Base {
  virtual ~Base(void) { /* implement */ }
};

class Derived {
  virtual ~Derived(void) { /* implement */ }
};

Base * p = dynamic_cast<Base *>(new Derived);
delete p;  // Calls Derived's destructor.
```

24.6.1. Dynamic Binding

```
class Base {
public
  // Use dynamic binding for this function
  virtual void call(void)
    { cout << "Base" << endl; }
};

class Derived1 : public Base {
public:
  //Redefine it here
  virtual void call(void) override
    { cout << "Derived1" << endl; }
};

class Derived2 : public Base {
public:
  // And redefine it here
  virtual void call(void) override
    { cout << "Derived2" << endl; }
};

int main(void)
{
  Base * ptr[3] = { new Base, new Derived, new Derived2 };
  for(int i=0; i<3; ++i){
    // Calls correct version
    ptr[i]->call();
    delete ptr[i];
  }
  return 0;
}
```

24.7. Virtual Function Table

Virtual functions supported by **virtual function table**. Each class with **virtual** functions are backed by function pointer array. **Pointer** points to *most derived function* version. Base class gets a *hidden pointer* (**vptr**) to the virtual function table. **Vptr** gets set depending on *Derived class*. So extra indirection and typecasting introduces performance overhead.

24.8. Multiple Inheritance

Inherits from several base classes. Can inherit from multiple base classes. Java uses **interfaces** but constrained.

```
class Derived : public BaseOne, public BaseTwo
{
public:  // Constructor calls each Base Constructor
  Derived(...) : BaseOne(...), BaseTwo(...) {}
};
```

24.8.1. Ambiguity

Base classes define members/vars with same signature, we use scope resolution.

```
class BaseOne {
public:
```

```
  ostream & print(ostream & out) { out << "BaseOne"; return out; }
};
class BaseTwo {
public:
  ostream & print(ostream & out) { out << "BaseTwo"; return out; }
};
class Derived : public BaseOne, public BaseTwo {
public:
  ostream & orate(ostream & out) {
    out << print(out); return out;  // Ambiguous
    out << BaseOne::print(out); return out;
  }
};
```

24.8.2. Multiple Copies of Base

```
class Animal {  // Common Base
  public:
    virtual void flap();
};
class Mammal : public Animal {
  public:
    virtual void breathe();
};
class WingedAnimal : public Animal {
  public:
    virtual void flap();
};

// A bat is a winged mammal
class Bat : public Mammal, public WingedAnimal {  // Join
};
Bat bat;
bat.eat();  // eat() derived through Mammal, or WingedAnimal?
Animal * a = dynamic_cast<Animal *>(new Bat);  // Ambiguous base
```

1. Virtual Inheritance

   Use **virtual** inheritance to solves the two versions of Animals vptr.

   ```
   class Animal {...};
   class Mammal : public virtual Animal {...};
   class WingedAnimal : public virtual Animal {...};
   class Bat : public Mammal, public WingedAnimal {...};
   ```

24.9. Overide Keyword

Mathod to tell the compiler that we're trying to **override** a function.

```
class Base {
public:
  virtual void f(int arg) {}
};

class Derived : public Base {
public:
  virtual void f(float arg) override {}
};
```

Also tells compiler to complain if no override happens.

24.10. Final Keyword

Similar to *Java keyword*.

- *Prevent* further **inheritance of classes**.

```
class Base final {};
class Derived : public Base {};  // fails
```

- *Prevent* further **inheritance of functions**.

```
class Base {
  virtual void f(void) final;
};

class Derived : public Base {
  virtual void f(void);  // fails
};
```

24.11. Abstract Classes

Cannot be instantiated.

- contains 1+ *pure virtual functions* (PVG) - (Java: abstract functions)
- **syntax**: *virtual ret_type FunctionName(args) = 0;*
- can contain **non-abstract members**.
- a sub-class *must* implement **all PVF** to be instantiable.
- if some PVG are not implemented, the *sub-class* is abstract.

24.12. Static Keyword

24.12.1. Static Class Member Variables

static class variables are associated with the *type* but not with a *single instance* of that type.

```
class buffer
{
public:
  int N;
  float * a;
  const static int DEFAULT_SIZE = 10;
  const static std::string NAME;

public
  buffer(int size=DEFAULT_SIZE) : N(size), a(new float[N] {}
};
// Have to initialise non-integral static members in .cpp file
const std::string buffer::NAME = "Harry";
```

Use scope operator **::** on type to access externally.

```
buffer::NAME = "Draco";
```

24.12.2. Static Class Member Functions

static class member functions associated with the type but not with a *single instance* of that type.

```
class A
{
public:
  static int add(int lhs, int rhs) { return lhs + rhs; }
};
```

Use scope operator **::** on type to access externally.

25. C++ Templates

C++ support for **Generic Programming**. These are *Classes/Algorithms* written using *to-be-specified-later* types.

```
template <typename T>  // Parameterise buffer with some type T
class buffer {
private:
  T * a;               // pointer to a T
  int _size;           // buffer size
public:
  buffer(int size) _size(size), a(new T[size])  {}
  ~buffer(void)  { delete [] a; }
  T & operator[](int index) { return a[index]; }
};
```

Type **specified** upon *instantiation/invocation*.

```
buffer<int> int_buffer(10);        // buffer of 10 ints please
buffer<myobj> myobj_buffer(20);    // buffer of 20 myobjs please
```

- More powerful than Java/C# Generics.
- **Turing Complete** (Can simulate a computer).
- **Template Metaprogramming** (write compile time programs).
- **Standard Template Library (STL)** defines *algorithms* and *containers* using templates (macros can do similar things).

25.1. Templates Advantages

- Write the generic code once, use many times.
- Static code evaluated at compile time.
- Produces aggressively optimised and inlined binaries.
- Templated code completely defined in header (.h) files.
- Type-safe, Macros are not.
- Static polymorphism + templates can solve many dynamic polymorphism problems.

25.2. Template Disadvantages

- Code bloat.
- Templated code completely defined in header (.h) files.
  - Long compile times
  - No code separation

- No information hiding
- Binaries may be more difficult to debug.
- Nasty compiler errors.

25.3. Template Code Organisation

Compiler only knows the *form* of templated code, it doesn't know what types will be supplied so it can't produce binaries yet. Everything must be declared in the header (.h) file.

```
template <typename T> class buffer {
  T * a; int _size;
public:  // Can declare + implement within class
  buffer(int size) : a(new T[size]), _size(size) {} // constructor
  ~buffer(void) { delete [] a; }                    // destructor
  T & operator[](int index);  // Just declare operator[] in class
};
// Implementation of operator[] later in the .h
template <typename T> T & buffer<T>::operator[](int index)
  { return a[index]; }
// Standalone function
template <typename T> T mymax(const T & lhs, const T & rhs)
  { return lhs < rhs ? lhs : rhs; }
```

25.4. Template Declarations and Parameters

- **One Template Parameter**

```
template <typename T> class buffer {};
```

- **Multiple Template Parameters**

```
template <typename Key, typename Data, typename Compare, typename Alloc>
map {}
```

- **Default Template Parameters**

```
template<typename CharT,
     typename Traits = std::char_traits<CharT>,
     typename Allocator = std::allocator<CharT>
> class basic_string {};

typedef basic_string<char> string;
typedef basic_string<char32_t> u32_string;
```

If CharT had a default parameter we could

```
basic_string<> basic;
```

25.5. Expression Parameters

- Object Pointer/Reference.
- Function Pointer/Reference.
- Class Member Function Pointer/Reference.
- **integral** types.

```
template <typename T, int Size>
```

```
class buffer {
  T * a;
public:
  buffer(void) : a(new T[Size]) {} // constructor
  ~buffer(void) { delete [] a; }  // destructor
  T & operator[](int index) { return a[index] };
};
```

25.6. Template Specialisation

Templated class defines behaviour for a **set** of types.

```
template <typename T, int Size>
class buffer {  // Buffer defined for all types
  T * a;
public:
  buffer(void) : a(new T[Size}) {}
  ~buffer(void) { delete [] a; }
  T & operator[](int index) { return a[index] };
};
```

We may want to customise class definition for:

- A range of types
- A particular type

25.6.1. Class Template Specialisation

```
// General case
template <int depth> class Fib { public:
  static const unsigned long value = Fib<depth-1>::value + Fib<depth-2>::value;
};
// Specialise class template for iteration 0;
template <> class Fib<0> { public:
  static const unsigned long value = 1;
};
// Specialise class template for iteration 1;
template <> class Fib<1> { public:
  static const unsigned long value = 1;
};

std::cout << Fib<12>::value << std::endl;
```

25.6.2. Partial Template Specialisation

Class with many template parameters can be **Partially Specialised**.

```
// Most general form
template <typename T, int Size> class buffer {
private:
  T * a;
public
  buffer(void) : a(new T[Size]) {}
};
// Partially specialise buffer to handle bools differently
// Pack them into ints for example
template <int Size> class buffer<bool, Size> {
private:
  int * a;
```

```
public:
  buffer(void) : a(new int[Size/sizeof(int) + 1]) {}
};
```

25.6.3. Function Template Specialisation

Template definition of class/functions provide general versions for all types.

```
// Just cast value's memory address to a long
template <typename T>
long hash_function(const T & value)
  { return long(&value); }
```

However, we may want to **specialise** for a particular type.

```
template <>
long hash_function<std::string>(const std::string & value)
  { long value; /* hash each character */ return value; }
```

Function templates have to be fully specialised.

```
myobj obj; hash_function(obj);  // Uses most general version
string s; hash_function(s);     // Uses version specialised for string
```

Avoid specifying template parameters. Compiler figures them out from arguments.

```
string s; hash_function<std::string>(s); // Forces use of string specialisation
```

Rule of Thumb - Compiler selects the most specific specialisation.

25.7. Trait Classes

A templated class that characterises a type.

- **std::numeric_limits** is a good example.

```
template <typename T> class numeric_limits {
  typedef T value_type;             // Store the type we're creating
  const static bool is_signed;      // Is this type signed?
  const static bool is_integer;     // Is it integral?
  const static bool has_infinity;   // Does it have an idea of inifinity
  const static int digits;          // How many radix digits does it have?
};
std::numeric_limits<int>::is_signed = true;
std::numeric_limits<int>::has_infinity = false;
std::numeric_limits<float>::has_infinity = true;
std::numeric_limits<float>::value_type v = 10.0f;
```

- We can also **specialise for ints and floats**

```
template <> class numeric_limits<int> {
  typedef int value_type;
  const static bool is_signed = true;       // ints are signed
  const static bool is_integer = true;      // ints are integral
  const static bool has_infinity = false;   // ints can't do infinity
  const static int digits = 31              // 31 bits
};
template <> class numeric_limits<float> {
  typedef float value_type;
  const static bool is_signed = true;       // ints are signed
  const static bool is_integer = true;      // floats aren't integral
  const static bool has_infinity = false;   // float have infinity
```

```
  const static int digits = 31              // 24 bits
};
std::numeric_limits<float>::has_infinity == true;
std::numeric_limits<int>::has_infinity == false;
```

25.8. Dependent Typenames

```
template <typename T>
class some_class {
  typedef std::vector<T>::iterator iterator_type;
  std::vector<T>::iterator i;
};
```

std::vector<T>::iterator depends on type T, which hasn't been supplied yet. The iterator could be a static variable/function.

- prepending with **typename** tells compiler its type.

```
template <typename T>
class some_class {
  typedef typename std::vector<T>::iterator iterator_type;
  typename std::vector<T>::iterator i;
};
```

25.9. Template Coding

- Lots of type propagation.
- New types build from old.
- Everything evaluated at compile time.
- By contrast, Polymorphic OO strategy involes
  - Dynamic Polymorphism
  - Explicit Interfaces/Abstract Classes
  - Requires Run-Time Type Checking
  - Performance Overhead

25.10. Template Concepts

Done for nicer error message and specifying constraints on template types.

25.10.1. Concept Checking

```
template <typename T>
struct LessThanComparable {
  void constraints(void)
    { bool result = a < b; }
  T a, b;
};
```

This is then instantiated in code requiring functionality.

```
template <typename T>
const T & min(const T & lhs, const T & rhs)
{
```

```
                          // This is optimised out, but the compiler
  LessThanComparable<T> dummy;     // considers this type, and complains
                          // if < not supported

  return lhs < rhs ? lhs : rhs;
}
```

25.10.2. C++14 Template Concepts

Explicitly state type requirements as language construct.

```
template <LessThanComparable T>
const T & min(const T & lhs, const T & rhs)
{
  return lhs < rhs ? lhs : rhs;
}

template <typename T> requires LessThanComparable<T>
const T & min(const T & lhs, const T & rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

26. The Standard Template Library (STL)

Templated Containers, Iterators accessing Container Data, Templated Algorithms operating on Iterator Ranges, Template Concepts and algortihms independent of Container design.

```
#include <vector>
#include <algorithm>

std::vector<float> data = { 1, 2, 3, 4, 5 };
std::vector<float> result(data.size());
std::transform(data.begin(), data.end(), result.begin(), [](float v) { return
v*3 } );
```

26.1. Containers and Iterators

Containers hold elements (objects/simple);

- Code is stamped out for each type. Inlined and efficient.

```
pair<int, char>            // struct holding two elements
vector<int> V;             // resizable array
list<int> L;               // linked list
set<int> S;                // tree holding unique elements
map<string, int> M;        // tree doing associate map
stack<int> ST;             // LIFO structure
priority_queue<int> P;     // Always pops the greatest element
```

Container classes can be traversed with **iterators**.

- **begin()** and **end()** define the full range of traversal.
- **begin()** is at 0.
- **end()** is at position *last_element+1*.

26.1.1. Iterators

All containers provide:

```
Container::iterator
Container::const_iterator
```

and may provide:

```
Container::reverse_iterator
Container::const_reverse_iterator
```

1. Iterator Requirements
    - Copy Constructor.
    - Copy Assignment Operator.
    - Dereference Operators (both * and -> usually).
    - Operator++ to move to the next data element.
    - Operator== and Operator!= for algorithm functions.
2. Iterator Traits

    These characterise an iterator.

    ```
    template <typename Iterator> struct iterator_traits
    {
      typedef typename Iterator::difference_type difference_type;
      typedef typename Iterator::value_type value_type;
      typedef typename Iterator::pointer pointer;
      typedef typename Iterator::reference reference;
      typedef typename Iterator::iterator_category iterator_category;
    };
    ```

    So when you write generic iterator code, you can use:

    ```
    std::iterator_traits<Iterator>::value_type X;
    ```

    Specialised for pointer types (**Pointers are iterators!**).
3. C++11 Range-Based Forloop

    C++11 new range-based forloops.

    <u>Arrays</u>

    ```
    int my_array[5] = {1, 2, 3, 4, 5};
    for (int & x : my_array) {
      x *= 2;
    }
    ```

    <u>Objects with begin() and end()</u>

    ```
    vector<string> my_array = { "Apple", "Pear", "Guava" };
    for (string & s: my_array) [
      cout << s;
    }
    ```

26.2. STL Algorithms

Standalone functions operating on iterator ranges.

- Templated by Iterators:

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator
result)
{
  for(; first != last; ++first, ++result)
    *result = *first;
  return result;
}
```

- Sample Usage:

```
vector<int> data = { 0, 1, 2, 3, 4 }; // C++11 initializer list
vector<int> result(data.size());      // Allocate space for result
copy(data.begin(), data.end(), result.begin());
```

- Also works with Arrays:

```
int A[5] = { 0, 1, 2, 3, 4 };
copy(A, A+5, result.begin());
```

26.2.1. Useful Iterator Adaptors

- **back_insert_iterator** adds things to the end of a container:

```
int A[5] = { 0, 1, 2, 3, 4 };
vector<int> result;  // Initialised empty
back_insert_iterator<vector<int>> bii(result);
copy(A, A+5, result.begin());  // Crashes, no space in result
copy(A, A+5, bii);             // espands result automatically
// Nice helper function automatically creates back_insert_iterator
copy(A, A+5, back_inserter(result));
// Also front_insert_iterator and front_inserter
list<int> lresult;
copy(A, A+5, front_inserter(lresult));
```

- **ostream_iterator** adds to an iostream:

```
// Outputs "0, 1, 2, 3, 4"
std::copy(A, A+5, ostream_iterator<int>(std::cout, ", "));
```

- **various**

```
vector<int> data(10);                     // Initialise with 10 ints.
fill(data.begin(), data.end(), 100);    // Set them all to 100.

data[5] = 5;                              // Remove all int
remove(data.begin(), data.end(), 5);    // set to 5.

data[5] = 20;   // Get iterator to element with value 20.
vector<int>::iterator i = find(data.begin, data.end(), 20);
```

- **sorting**

```
vector<int> data = { 2, 1, 4, 0, 3 };  // C++11 initializer list
sort(data.begin(), data.end());
```

- **std::numeric**

```
vector<int> data = { 0, 1, 2, 3, 4 };  // C++11 initializer list
// Returns 5 + (0 + 1 + 2 + 3 + 4)
int sum = accumulate(data.begin(), data.end(), 5);
```

26.2.2. Function Objects / Functors

Function Objects generalize C++ functions. This is heavily used in STL to facilitate generic programming. These are passed into STL algorithms instead of normal functions (both are allowable).

- A class object with *overloaded* parenthesis **operator()**

```
class F {
public:
  bool operator()(int a, int b) const
    { return a < b; }
};
F f;   // create an instance
if (f(i,j)) cout << "i < j";
```

1. Function Objects

   STL provides many predefined function objects.

   - **Arithmetic** (binary/unary)
     - plus, minus, divides, negate, modulus
   - **Relational** (binary, predicate)
     - equal_to, not_equal_to, greater, greater_equal, less, less_equal
   - **Logical** (binary/unary, preficate)
     - logical_not, logical_and, logical_or
   - **Predicate** example: return boolean values based on arguments.

   ```
   greater<int> l;
   if (l(3,2)) cout << "3 is greater than 2";
   ```

   Templated works with any appropriately defined data. **less<T>**: binary predicate, returns true if 1^st < 2^nd. Needs **operator<** to be defined for that class.

   ```
   less<string> l; string s1, s2;
   if (l(s1,s2)) cout << s1 << " is less than " << s2;
   ```

2. Functors

   Usually lightweight objects which store a pointer to value for short duration.

   Example with **transform** algorithm

   ```
   class custom_functor {
   public:
     int operator()(const int & x) const
       { return x*3; }
   };
   vector<int> data = { 2, 4, 6 }; vector<int> result;
   std::transform(data.begin(), data.end(), back_inserter(result),
   custom_functor());
   ```

   - Customize the **copy_if** algorithm. Copies elements in a range, if supplied

   functor returns true;

   ```
   template <typename InIterator, typename OutIterator, typename Predicate>
   OutIterator copy_if(InIterator first, InIterator last, OutIterator
   result, Predicate pred)
   ```

```
{
  for ( ; first != last; ++first)
    if (pred(*first))
      *result++ = *forst;
  return result;
}

class not_equal_to {
public:
  not_equal_to(int i) : cmp_value(i) {}
  bool operator()(int container_value) const
    { return cmp_value != container_value; }
  int cmp_value;
};
...
vector<int> data = { 0, 1, 2, 3, 4 };  // C++11 initializer list
vector<int> result;
not_equal_to net(3);  // CONSTRUCT Functor. Call not_equal_to(3)
net(3) == false;      // Call net.operator(3).  { return 3 != 3; }
net(2) == true;       // Call net.operator(2). { return 3 != 2; }
// Copies everything except 3
copy_if(data.begin(), data.end(), back_inserter(result),
not_equal_to(3));
```

Predicate is functor returning boolean value. Tests the value referenced by current iterator, copies into output range if true;

- Templated version of **not_equal_to.**

```
template <typename T> class not_equal_to {
public:
  not_equal_to(const T & cmp_value) : cmp_ptr(&cmp_value) {}
  bool operator() (const T & container_value) const
    { return *cmp_ptr != container_value; }

  const T * cmp_ptr  // Avoid deep copies, store a ptr1
};
...
vector<string> data = { "AA", "BB", "CC", "DD", "EE" }; // C++11
initializer
vector<string> result;
// Copies everything except "DD"
copy_if(data.begin(), data.end(), back_inserter(result),
not_equal_to<string>("DD"));
```

- **For_each** applies to functor in read-only manner.

```
template <typename T> class bit_examiner {
public:
  bit_examiner(std::size_t which_bit) : count(0), bit(which_bit) {}
  bit_examiner(const bit_examiner & rhs) : count(rhs.count), bit(rhs.bit)
{}

  void operator() (const T & value)
    { if(value & (0x1 << bit)) ++count; }

  std::size_t count;
  std::size_t bit;
};
```

- **Transform** applies functor to every value in a sequence

```
template <typename T>
class add_to{
  add_to(const T & value) : ptr(&value) {}
  T operator()(const T & value) const
    { return *ptr + value; }  // add value to every element

  const T * ptr;
};
...
vector<int> data = { 0, 1, 2, 3, 4 };  // C++11 initializer list
vector<int> result;
transform(data.begin(), data.end(),            // Adds 5 to every element.
Store
     back_inserter(result), add_to<int>(5));  // in result.
```

26.3. C++11 Lambdas

[capture](arguments)->return_type {body}

- **capture**: block for capturing variables from higher scope.
- **arguments**: just like function arguments.
- **return_type**: optional, otherwise compiler figures it out.
- **body**: code.

26.3.1. Simple Examples

```
[](int x, int y) { return x + y; }
[](int x, int y)->float { return x + y; }
```

Capture Block specifies capture of variables in lower scope.

```
[]           // Use of external variables generates an error.
[x, &y]      // x capture by value, y captured by reference.
[&]          // external variables implicitly captured by reference.
[=]          // external variables implicitly captured by value.
[&, x]       // x explicitly captured by value. Else by reference.
[=, &z]      // z explicitly captured by reference. Else by value.
```

26.3.2. Complex Example

```
std::vector<int> v = { 0, 1, 2, 3, 4 };  // C++11 initialiser
std::vector<int> result(v.size());
int total = 0;
int value = 5;

// total implicity by reference, value explicitly by value,
// this explicity capture by value
std::for_each(v.begin(), v.end(),
    [&, value, this](int x) { total += x * value * this-> func(); });
std::transform(v.begin(), v.end(), result.begin(),
    [](int value) { return value*3; } );
```

26.3.3. Sorting Example

Use *function* object to wrap char& string comparison behaviour.

```
class StrCmp {
public:
  bool operator() (char *s1, char *s2) {
    return strcmp(s1,s2) < 0; // char* compare
  }
};
template <typename IT, typename Comp>
void  sort(IT s, IT e, Comp cmp)
{ ... if (cmp(*s, *e)) ... } // function object

char *array[] = {"abc", "def", "ghi"};
list<char*> l;

copy(array, array+3, front_inserter(l));
sort(l.begin(), l.end(), StrCmp());
sort(array, array+3, StrCmp());
```

26.3.4. Function Object vs Function Example

Binary function takes two arguments.

```
bool f( int& x, int& y) { return x < y; }
class F {
private: int x;
public:
  F(int y = 0) : x(y) {}
  bool operator()(int& y, int& z){ return y < z; }
};

/* Algorithm Eval() - apply BinaryFunction to 2 arguments */
template <typename T, typename BinaryFunction>
  bool Eval( T& x, T& y, BinaryFunction B){ return B(x,y); }

int main() {
  int x = 3, y = 4;
  cout << Eval(x, y, f) << endl;  // apply a function
  cout << Eval(y, x, F()) << endl;  // apply function object
  return 0; }
}
```

26.3.5. Templated Sum with For_each Example

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
template <typename T>
class Sum {
private:
  int N;
  T sum;
public:
  Sum() : N(0) {}
  Sum(const Sum& S) : N(S.N), sum(S.sum) {}
  void operator()(const T & v) {
    if (N == 0)
      sum = v;
    else
```

```
      sum = sum + v;
  }
  T getsum(void) { return sum; }
  friend ostream & operator<< (ostream& os, const Sum<T>& obj)
    { os << obj.sum
};

int main() {
  int v[] = {5, 6, 7};
  Sum<int> S = for_each(v, v+3, Sum<int>());
  cout << "Sum is: " << S << endl;

  vector<string> w = {"The ", "rain ", "in ", "Spain..."};
  Sum<string> Q = for_each(w.begin
  cout << "Concatenation is: " << Q << endl;

  return 0;
}
```

26.4. Runtime Type Identification (RTTI)

typeid() allows type comparison. It can also print out the name.

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base { virtual void f(){} };
class Derived : public Base {};
int main () {
   Base* a = new Base;
   Base* b = new Derived;
   cout << "a is: " << typeid(a).name() << endl;
   cout << "b is: " << typeid(b).name() << endl;
   cout << "*a is: " << typeid(*a).name() << endl;
   cout << "*b is: " << typeid(*b).name() << endl;
   return 0;
}
```

26.5. C++11 Polymorphic Wrapper for Function Objects

Similar to functors, create callbacks:

- **syntax**: *function<return_type (args)>*

```
// Create wrapper to function like 'int f(int, int)'
std::function<int (int, int)> func;
// plus<int> has 'template<T> T operator()(T, T)' member
std::plus<int> add;
func = add;  // OK - Parameters and return types are the same.
int a = func(1, 2);  // NB: If 'func' does not refer to a function
                 // 'std::bad_function_call' is thrown.
```

- this can also be done on class variables

```
class X { public: int foo(int i) { return i*2; }
function<int (X*, int)> f = &X::foo;
X x;
cout << f(&x, 5) << endl;
```

and with lambdas!

26.5.1. Big Example

```
#include <iostream>    // std::cout
#include <functional>  // std::function, std::negate

// a function
int half(int x) {return x/2;}

// a function object class
struct third_t {
  int operator()(int x) {return x/3;}
};

// a class with data members:
struct MyValue {
  int value;
  int fifth() {return value/5;}
};

int main () {
  std::function<int(int)> fn1 = half;            // function
  std::function<int(int)> fn2 = &half            // function pointer
  std::function<int(int)> fn3 = third_t()        // function object
  std::function<int(int)> fn4 = [](int x){return x/4;};  // lambda
  std::function<int(int)> fn5 = std::negate<int>();  // std func. object

  std::cout << "fn1(60):  " << fn1(60) << endl;
  std::cout << "fn5(60):  " << fn5(60) << endl;

  // stuff with members
  std::function<int(MyValue&)> value = &MyValue::value; //ptr to data member
  std::function<int(MyValue&)> fifth = &MyValue::fifth; //ptr to member
function

  MyValue sixty {60};

  std::cout << "value(sixty): " << value(sixty) << endl;
  std::cout << "fifth(sixty): " << fifth(sixty) << endl;

  return 0;
}
```

26.6. C++11 Binding Function Arguments (Currying)

Bind default arguments to a function then call

```
int f(int, char, double);
auto ff = std::bind(f, std::placeholders::_1, 'c', 1.2);
int x = ff(7);

using namespace std::placeholders;
auto frev = std::bind(f,_3,_2,_1);  // Reverse
int x = frev(1.2, 'v', 7);
```

- Generally assign a **bind** to a **function**

```
std::function<int (int)> = std::bind(f, _1, 'c', 1.2);
```

- Then with class methods

```
class X {
```

```
public:
  void foo(int); }
X x;
using namespace std::placeholders;
std::function<void (int)> f = std::bind(&X::foo, &x, _1);
f(5);
```

26.6.1. Big Example

```
// bind example
#include <iostream>      // std::cout
#include <functional>    // std::bind

// a function (also works with function objects
double my_divide (double x, double y) {return x/y;}

struct MyPair{
  double a,b;
  double multiply() {return a*b;}
};

int main () {
  using namespace std::placeholders;

  // binding functions:
  auto fn_five = std::bind (my_divide, 10, 2);      // returns 10/2
  std::cout << fn_five() << endl;                   // 5

  auto fn_half = std::bind (my_divide, _1, 2);      // returns x/2
  std::cout << fn_half(10) << endl;                 // 0.2

  auto fn_rounding = std::bind<int> (my_divide, _1, _2); // returns int(x/y)
  std::cout << fn_rounding(10,3) << endl;            // 3

  Mypair ten_two {10, 2};

  // Binding Members:
  auto bound_member_fn = std::bind (&MyPair::multiply, _1); // ret:x.multiply
  std::cout << bound_member_fn(ten_two) << endl;            // 20

  auto bound_member_data = std::bind (&MyPair::a,ten_two);  // ret:ten_two
  std::cout << bound_member_data() << endl;

  return 0;
}
```