C++

Patrick Marais   Simon Perkins

Computer Science Department
University of Cape Town

February 5, 2024

Outline

C++ History

- Developed by Bjarne Stroustrop around 1979.
- "C With Classes".
- Goal: Object Orientation to C without sacrificing performance, portability or **low-level functionality**.
- OO component refined over two decades, adding virtual functions, operator overloading, multiple inheritance, and exception handling.
- **Generics** (Templates). Powerful but difficult.
- `http://www.cplusplus.com/info/history/`

C++ Design Goals

- From Wikipedia
- C++ is designed to:
  - be a statically typed, general-purpose language
  - support multiple programming styles
    - (procedural programming, data abstraction, object-oriented programming, and generic programming)
  - give the programmer choice.
  - be compatible with C, and as efficient and portable.
  - function without a sophisticated programming environment
- Avoids platform specific, non-general features
- No overhead for unused features (the "zero-overhead principle")

1983 - Bjarne Stroustrup bolts everything he's ever heard of onto C to create C++. The resulting language is so complex that programs must be sent to the future to be compiled by the Skynet artificial intelligence. Build times suffer. Skynet's motives for performing the service remain unclear but spokespeople from the future say "there is nothing to be concerned about, baby," in an Austrian accented monotones. There is some speculation that Skynet is nothing more than a pretentious buffer overrun.

C++ vs Java

- Java ByteCode vs C++ Compiled Binaries (Portability vs Speed).
- Java enforces a strong OO paradigm, C++ does not.
  - OO Discipline vs many different coding styles.
  - Modern C++ code utilises a combination of **imperative**, **OO**, **generics** and **functional**.
- C++ has a pre-processor for specifying code inclusion and compiler directives.
- C++ is the older language and sports more complex features (operator overloading, multiple inheritance, templates)
  - C++ Templates far more advanced than Java Generics.
- C++ is still the industry standard for performant code.
  - Databases, Webservers and Games are more likely to be written in C++.

Useful online references

- Good references:
  - `http://www.cppreference.com`
  - `http://isocpp.org/tour`
  - `http://cplusplus.com`
- `http://www.learncpp.com`
  - good online course. covers material in similar order to this course.

C++ Hello World

```
#include <iostream> // Include IO stream library header

// Command line arguments
// argc: Number of parameters
// argv: Array of pointers to chars (strings).
int main(int argc, char * argv[])
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

- main function is entry point to program

C++ Program Structure

- C++ program usually composed of *header* (.h) and *source* (.cpp) files.
- Headers (.h) **declare** the existence of code elements.
  - Function signatures. Name, variables, return variables.
  - Class structure. Methods and members.
- Source files (.cpp) **define** the code.
  - Write code for functions and class methods
  - Compiled into **binary** object files (.o)
  - Source files include header files so that they can determine:
    - how to invoke functions and class methods in other object files.
    - how much memory a class occupies.
- Practically, included files inserted in place within .cpp file
  - and compiled.

C++ Program Structure

- Fibonnaci example
- We create
  - Driver source file containing **main** function. (fibdriver.cpp)
  - Header file containing function declaration (fib.h)
  - Source file containing function definition (fib.cpp)

C++ Structure Example

```
// Pre-processor directives that guard against
// redeclaration 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
```

C++ Structure Example

```
// 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);
}
```

C++ Structure Example

```
#include <iostream> // Include I/O Stream library headers.
#include "fib.h"    // 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;

  return 0; // Function needs a return value
}
```

Simple compile and link

- Compile source (.cpp) files to create binary code.
- **-c** compile to object file.

```
# 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
```

- Link object files to create an executable
- Function calls between binary objects are “linked”.

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

- **-o name** name of final executable

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 bytecode, although JIT
- **Declaration** (header file) needed by compiler to interpret library (**definition**).
  - Compiler needs to know HOW to call a given function. Arguments + return values.
  - e.g. #include <string> to use string class type.

C++ Libraries

- Directory structure

```
/usr/local
+-- mathlib
    +-- include
    |   +-- mathlib
    |       +-- bignumber.h
    +-- lib
        +-- libmathlib.so
```

- Header file tells compiler what bignumbers and multiply look like.

```
#include <mathlib/bignumbers.h>
...
bignumber huge_one, huge_two;
multiply(huge_one, huge_two);
```

- **-I directory** instructs compiler about header location.

```
# Compiler can now find bignumbers.h
g++ fib.cpp -c -I/usr/local/mathlib/include
```

C++ Libraries

- Directory structure

```
/usr/local
+-- mathlib
    +-- include
    |   +-- mathlib
    |       +-- bignumber.h
    +-- lib
        +-- libmathlib.so
```

- **libmathlib.so** contains code implementing bignumbers functionality.
- **-L directory** location. **-l libname** library

```
# Link against libmathlib.so
g++ fib.o fibdriver.o -o fib -L/usr/local/mathlib/lib -lmathlib
```

- Note (1) No lib prefix with flag. (2) libraries *after* object files.
- Tell unix about unusual shared library locations:

```
export LD_LIBRARY_PATH=/usr/local/mathlib/lib
```

More about linking

- **main** function in **fibdriver.cpp** files calls **fib** defined in **fib.cpp**.
- Compiled into .o binary code files.
- **main** binary code in **fibdriver.o** calls the **fib** binary code in **fib.o**.
- During linking phase, binary dependencies are **linked together**.
- **main** binary code knows how **fib** is called because it has seen the declaration in **fib.h**.
- Compiler may complain about "*undefined symbol*" or "*undefined reference*".
- May also produce a **mangled**, somewhat decipherable name.
  - You're forgotten to link an object (.o) file, or library (.so).
  - Compiler can't find the binary for the function.

The Make Utility

- Projects consist of multiple source files and headers.
- changes may require recompile of the project.
- **make** automates the process of dependency checks.
- Configured with a **Makefile**, a collection of dependency rules.
- Makefile rules are of form:
  target: dependencies
  action
- after editing, type **make** and all will be consistent
  - **make** (use default Makefile) or **make –f** *makefilename*

The Makefile

- Example:

```
myexec: file1.cpp file2.cpp hdr.h
	g++ -o myexec file1.cpp file2.cpp -lm
```

- Builds **myexec** dependant on file1.cpp, file2.cpp and hdr.h
- .cpp and .h changes trigger rebuild
- Rebuild rule is given on the second line
- many different rule types etc, refer to man pages
- things to remember:
  - no spaces after return
  - tab to start rule (not a space etc!!)
- if your rules are wrong, the project code may be out of date

Makefile Variables

- Makefiles use variables to ease configuration
    - e.g. `CC=g++ # sets C++ compiler to g++`
- rules rewritten to use these variables

```
mytest: mytest.cpp
  $(CC) $(CCFLAGS) mytest.cpp -o mytest
```

- CCFLAGS has a default value (see man pages)
- "macros" or built in rules can be used to simplify e.g.

```
.cpp.o:
  $(CC) -C $<
```

    - the macro $< expands to a matching .cpp file
    - builds .o files using this build action for all .cpp files.
    - have default behaviour which we can override
- **.cc** is standard suffix, can add others: *.SUFFIXES*:

Other Makefile Info

- no space after line end
- tab for rule indent
- @ suppresses echo of rule (does not print out as it executes)
- if no target given, 1st rule made
- can make *target* (no dependency reqd)
    - **# make clean** (on command line)

```
# removes .o files
clean:
  rm *.o
```

- only changed files should rebuilt
- can insert files in a makefile with **include** statement

Canonical Makefile Example

```
CC=g++                               # Compiler
CUSTOMDIR=/home/alice/local          # Custom package location
LIBDIRS=-L$(CUSTOMDIR)/libs          # Library Locations
INCLUDES=-I$(CUSTOMDIR)/includes     # Header File Locations
LIBS=-lalice                         # libalice.a
CXXFLAGS=$(INCLUDES) -Wall           # Headers + All Warnings
LDFLAGS=$(LIBDIRS) $(LIBS)           # Libs + their locations
TARGET=myprog                        # Name of executable
OBJECTS=f1.o f2.o f3.o               # Object files to build into exe
# Linking Rule
$(TARGET):  $(OBJECTS)
  $(CC) $(OBJECTS) -o $(TARGET) $(LDFLAGS):
  @cp $(TARGET) ./binaries
# Macro: automatically associate *.o with *.cpp
.cpp.o:
  $(CC) $(CXXFLAGS) -c $<

clean:
  @rm -f *.o
```

IDE's and Debuggers

- IDE's:
  - QtCreator
  - Eclipse CDT
  - Geany (Text Editor really)
- Compilers:
  - **g++** and **clang++** on linux.
  - **Minimalist GNU for Windows** (MingW) has **g++** . `http://nuwen.net/mingw.html`
  - **Visual Studio for Windows**.
- Debugging
  - g++/clang++. Compile with -g flag.
  - Then run GDB. QtCreator and Eclipse do this automagically.

The C++ Pre-processor

- No Java equivalent
- Modifies/processes source code prior to compilation
- pre-processor *directives* introduced by #
  - Preprocessor deals with all of these prior to compilation
  - Recursively processes include files, modifying source code passed to compiler
- Common uses:

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

- Used to optimize, target platforms, compile only certain parts of code.
- use *#include file* to collect class/function declarations in one place C++
  - *source files* should contain (mainly) function **implementations**

Include Files

- The **#include** directive "inserts" the indicated file
  - At the point of the **#include**
  - Textual insertion which **modifies** the current file *before* compilation
- Include *header files*, for function prototypes, class defs etc
- Two include conventions:

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

- Version 1 searches default dirs (/usr/include)
- Version 2 searches explicit include dirs (-I/usr/local/matlib/include)
- If you include a file twice can give "redefinition" errors
  - use **#define** (see later)

Pre-processor Macros

- Define a *macro* with **#define**
- Can be "function" or constant :

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

  - MYINT replaced by 22 in C++ source
  - MYSQR(3) replaced by ((3)*(3)) in C++ source
- Convention is Upper-case Macro names
- Can set Macro values using compiler
  - g++ -DMYINT=22 ...
- Remember:
  - code replacement ONLY
  - final expression *must* preserve C++ syntax
- No semi-colon at end; to continue over lines, use \

```
#define MYLONGSTR "The quick brown fox\
jumped over the fence"
```

Conditional Macro Expansion

- conditionals: **#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
```

- use **#ifdef** or **#ifndef** to test if a macro has been defined
- can also **#undef** a defined macro
- Useful for writing multi-platform 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
```

Header Files

- How to avoid multiple file inclusions:

```
//   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
```

- If (in source file) we write:

```
#include "dog.h"
#include "dog.h"
```

- the second inclusion will do nothing, since the macro _DOG_H has been defined by the first inclusion.

Macro String Operations

- Stringizing: turning identifier into character string
  - **#define PRINT(y) std::cerr** << **#y "="** << **y** << **std::endl**
  - In code
    - PRINT(x); **std::cerr** << **"x="** << **x** << **endl;**
- Note: two strings next to each other are concatenated
- Token pasting: making new label/token from supplied tokens (tokens are NOT strings!)
- These operations are very useful for debugging code

```
#ifdef _DEBUG
#define INVOKE_FN(f)   debug_##f
#else
#define INVOKE_FN(f)   f
#endif
```

- In code: INVOKE_FN(F(3)); ⇒ **debug_F(3);** or **F(3);**
- depending on **_DEBUG** setting

Brief Explanation of code snippets

```
// A vector (resizable array) of ints.
// std::vector is templated with int variable
std::vector<int> intvec;
// std::cout is an output stream variable
// << adds data to the stream
// std::endl is a newline
std::cout << "5" << std::endl;
```

Simple C++ Types

- three kinds: simple, aggregate and class
- simple types are:
  - **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
- integral simple types can be **signed**/**unsigned**
  - **unsigned char c**: u_char may also be defined
  - **std::size_t**: standard unsigned integral type.
- simple type sizes are system dependent - **sizeof(type)**

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

Integral Types

- 16-bit operating systems
  - int 16-bit, long 32-bit
- 32-bit OS
  - int & long 32-bit, long long 64-bit
- 64-bit OS
  - int 32-bit, long & long long 64 bit
- Modern alternative to **sizeof() – numeric_limits**.

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

- Info on signed, integral, float parameters.
- **cstdint** header provides int32_t, int64_t, uint32_t, uint64_t types.

Scope

- What variable/function is a label bound to at this point in the code?
- Scope: classes, functions, code block.

```
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
```

- Variable is visible once defined
- Rule of thumb: Pairs of { } define new scope
- variable defined at **global** scope is visible everywhere
  - No Global scope in Java... and it's bad practice!
  - Every function/code in a source file can access a global variable!
- Labels **not** visible outside of scope:

```
{ int i = 3; i *= 2; }
// cannot see i here
```

Scope

- 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; }
```

  - local copy of **i** referenced; global unaffected
- 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;}
```

- ignore local version and print global version always
- Globals can cause confusing errors - try not to use them

Scope

- Scope is important!
- In C++, *lifetime of* **automatic** *variables is bounded to scope.*

```
void somefunction(void)
{
  anobject a; // Automatic variable

  {
    anobject b; // Automatic variable
    int c;      // Automatic variable
  } // Leaving scope, b and c are DESTROYED

} // Leaving function scope. a is DESTROYED
```

- *Resource Acquisition is Initialisation* (RAII)
- More to come...

Namespaces

- Large projects may have name "clashes"
- Overcome by qualifying each "label" in some way
- Solution: use 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
- Syntax **: namespace id** { **// names** }

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

- In fib.h

```
namespace maths { int fib(int n); }
```

- In fib.cpp

```
namespace maths {
int fib(int n) {
  if(n <=2) return 1
  else return fib(n-1)+fib(n-2)
} // close function brace
} // close namespace brace
```

- In fibdriver.cpp

```
int main(void) {
  std::cout << maths::fib(10) << std::endl;
}
```

- Works the same with classes.

Namespaces

- Nested namespaces

```
namespace foo { namespace bar { int a; } }
foo::bar::a = 5;
```

- Unnamed namespaces. Local to current compilation unit (.cpp file).

```
namespace { int a = 1; }
std::cout << ::a << std::endl;
```

- Don't mixed global namespace and unnamed namespace

```
namespace { int a = 1; }
int a = 2;
std::cout << a << std::endl; // Outputs 2
```

Namespaces

- Explicit qualification:

```
std::cout << project::p1 << std::endl;
```

- **using** *declaration* (one label only):

```
for (int i=0; i < 10; ++i) {
  using project::p1;
  std::cout << p1 << std::endl;
}
```

- Aliasing

```
namespace ublas = boost::numeric::ublas;
ublas::blastype a;
```

- Can use **using** *directive* (all labels part of current scope):

```
using namespace std; // everything now at current scope
cout << "foobar" << endl; // no std::cout or std::endl
// DON'T USE IN HEADERS - exposes whole namespace in each included file.
```

- C++ standard namespace **std** contains system classes, global

Variable Qualifiers

- a variable can have following qualifiers:
  - **extern**: variable defined outside current scope
  - **static**: variable bound to class/function/file
  - **const**: the value cannot be changed after initialisation
  - **register**: suggests that compiler use CPU registers to store variable
  - **volatile**: variable protected from compiler optimisations

Extern Qualifier

- Refers to a variable declared in some other compilation unit.
- globals.h

```
#ifndef _GLOBALS_H
#define _GLOBALS_H
namespace corsairs { extern int long_john_silver; }
#endif
```

- globals.cpp

```
#include "globals.h"
namespace corsairs { int long_john_silver; }
```

- driver.cpp

```
#include "globals.h"
void main(void) { corsairs::long_john_silver = 5; }
```

- binary code in driver.o can access variable in globals.o

Static Qualifier

- **static** variables can be *local* or *global*
  - global bound to file; invisible outside file, global in file
  - local bound to class or function; persistent

```
// Only visible within source file. Can't extern it
static int k = 0;

int func(void)
{
  static int i = 0; // Set to 0 on initial execution
  return i++;
}
...
cout << func() << endl; // outputs 0
cout << func() << endl; // outputs 1
cout << func() << endl; // outputs 2
```

Const Qualifier in Function Variables

```
void func(const int arg)
{
  const int a = arg;
  const int b = 2;

  arg = 6 // Compiler complains
  a = 5;  // Compiler complains
  b = 3;  // Compiler complains

  // Alternative expression - Not convention
  int const c = 4;
}
```

Type Definitions

- Create new type name from old.
- Simpler code, less typing.
- Obeys scoping principles.
- Example 1:

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

- Example 2: Long type names

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

C++11 - type related constructs

- Example 1: C++11 – **auto** keyword
- Deduce type of variable from expression on lhs

```
std::vector<float> vec;
// Didn't even have to type
// the iterator type out (even) once
auto i = vec.begin();
```

- Example 2: C++11 – **decltype** keyword
- 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;
```

Aggregate Types: structures

- groups data into a "record"
- ancestor of the **class**
  - All data and methods are **public**
  - For backward compatibility with C
  - Use a class if you really want a class!
- introduced by keyword **struct**:

access member vars with .

can actually include methods, but just use class

```
struct DataEntry {
  int IdNumber;
  char name[40];
  char address[300];
}; // NB! semi-colon
```

- **DataEntry** is now a valid type:

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

Structures

- can create *singleton* structures (no **struct** name):

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

- unique instance
- Assignment operator does a shallow copy, byte-by-byte

```
DataEntry d1 = d2;
```

- pointers will not be accessed – shallow copy

Enumerations

- Not related to Java Enumeration
- Create a set of named integer constants:

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

often used to manage error states

- first integer will be zero, increment for each label
- can change integer mapping e.g.

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

- this is now a valid type:

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

- Enumeration type is not **int**
- Enumeration scope is global or class
- For a class enum use :: outside of class, e.g

often created inside classes, but must then conform to class names

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

Class Types

- Declared with class keyword (cf. Java)
- Declares new type in header file (person.h)

user s.d.H for assignments

```
#ifndef PERSON_H
#define PERSON_H
namespace spooks { // in namespace
class person {
private:            // private members
  std::string n;
public:             // public members
  person(std::string name); // constructor
  void set_name(std::string name);
}; // NB! semi-colon
}  // end namespace
```

- C++ separates method code from class declaration
- Note ';' after last bracket !!!
- Access applies to all following members, until changed

Class Types

- Implement methods in source file (person.cpp)

```
#include "person.h" // incl class declaration
// implementation of methods
namespace spooks {
person::person(std::string name) : n(name) {}   // constructor
void person::set_name(std::string name) { n = name; } // member fn
} // end namespace
```

initialise a variable of a class

- Scope operator :: and class name associates declaration + definition

```
// file: driver.cpp
#include "person.h"
int main(void) {
  spooks::person X("Fox Mulder"); // create instance of person
  X.set_name("Dana Scully");      // apply method to it. operator.
  return 0;
}
```

- Note the namespace qualification.

Variable Initialisers

- 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};
```

- initialised in order; brackets for 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*.

C++11 Initializer Lists

- Extends Initializer Lists to work with non-POD constructs.

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

a{0,1,2 .. } also works (preferred for some instances, but mad use)

- instead of

```
std::vector<int> a;
a.push_back(0);
a.push_back(1);
```

- 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 */ }
    }
}
```

Type Conversion

- C++ strongly typed language (unlike C...)
- automatic or explicit *type casts*.
- automatic casts: expressions/assignments, function params, class etc
- Explicit:

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

C

essentially constr an int

C++

→ new style doesn't work since they're separate words.

- old style for e.g. **unsigned char**, **long long**
- type conversion may be unsafe: compiler tries to limit this
- e.g. cast **int** to **short** and back to **int** looses 2 bytes of data

strong type checking

There's also static cast.

std::static_cast<int>(5.0) → preferred compiler will check for you

C++11 constexpr

- Always had constant expressions e.g. $3 + 4$
- Now can have "constant" functions

```
constexpr int get_five(void) {return 5;}
// Create an array of 12 integers.
int some_value[get_five() + 7];
```

- Some Limitations
  - Must return non-void
  - Can't declare variables or new types in body
- Evaluated at *compile time*

C++ I/O

C++ I/O

C++ Input and Output

to log file

- Console I/O, via global objects **cout**, **cerr**, **clog** and **cin**.

#inc.. <string>

```
#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
```

sucks up whitespace in the input buffer

- File I/O via instantiated **ofstream** and **ifstream** objects.

```
#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
```

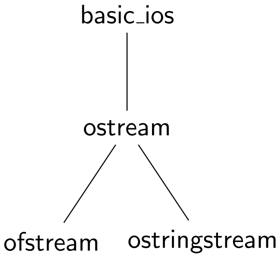
- Memory I/O via **ostringstream** and **istringstream**.

```
#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.
```

get the underlying memory

I/O Stream Hierarchy

- http://en.cppreference.com/w/cpp/io
- **ofstream** + **ostringstream** inherit from **ostream**.



- base class **ostream &** can bind to **ofstream/ostringstream** variables.
- Similarly with input classes.

I/O Stream Operators

- I/O 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
```

Console I/O

- via methods and overloaded operators (<<,>>)
- Example of simple I/O:

```
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 (normally)
- **cerr** is mapped to stderr (no redirection)
- the I/O operators (<<,>>) write/read text data

flushes output buffer, then add a newline.

```
cout << "\tApple\n";        // ASCII escape codes
cout << hex << 45 << endl; // hex and endl I/O manipulators
cout.setf(ios::left | ios::fixed); // Status Flags
```

- Special characters embedded in text string
- can cause newline, carriage return, "alarm", backspace etc.
- some examples:
  - **\n,\t**: newline and tab
  - **\"**: double quote
  - **\NNN,\xNNN**: print char with octal/hex code NNN
- examples:
  - **cout << "The terminal will beep now \a!";**
  - **cout << "Insert a \t tab or \t two";**

Manipulators

- Manipulators more powerful than escape codes
- `http://www.cplusplus.com/reference/iostream/manipulators/`
- affects behaviour of stream e.g. change printing precision
- 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;
```

- can write own manipulators
- can test stream for errors e.g.

```
if (!cin) { panic(); }
if (cin.eof()) { panic(); }
```

Reading Console Data

- $>>$ operator is *overloaded* to support different data types.
- examples:

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

- many other methods available - special formatting etc

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

Reading Lines of Console Data

- **getline** is an *extremely* useful function.

```
#include <string>      // Needed for string objects
#include <iostream>    // 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
}
```

- Watch for dodgy input

```
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
```

Reading from cin

→ end of line

- Are we there yet? **cin.eof()** tests for end of input.
- Ctrl D signals EOF for input data.
- Example:

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

- Note: **eof()** only true when input buffer empty
- The **ws** manipulator consumes white space

Memory-based I/O

- Format strings or read from strings in memory.
- Useful for converting non-string types to string
- Use **stringstream** preferably
  - **strstream** for old **char *** strings. deprecated + buggy.
  - **istringstream**/ **ostringstream** depending on input/output
- Example:

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

→ instantiate a istrstr obj

stream operator

File-based I/O

- two basic types: input and output file streams
- simple file I/O: **fstream**, **ifstream**, **ofstream**
- easier to use most appropriate one:

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

- Create and open at same time

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

can also wrap the code in a local scope, but not good practice

- Always close files (destructors tho)

Text File I/O

- overloaded operators (<<,>>) available; derived from **ios**
- example of reading from file (text):

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

- text must have ints; other input ignored → check if i actually has an int value stored in it
- type of input determined by variable
- white space separates data items (use **ws**!)

Binary File I/O

Binary data: more efficient storage

- Example of reading in binary data:

```
#include <iostream>
#include <fstream>
...
const int ARRAY_SZ=40;
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
```

reading 40 bytes at a time

byte is a char in C++

opens in binary mode

must always ensure that you don't go over the array bound

primitive arrays don't do bound checking for you.

Files: some comments

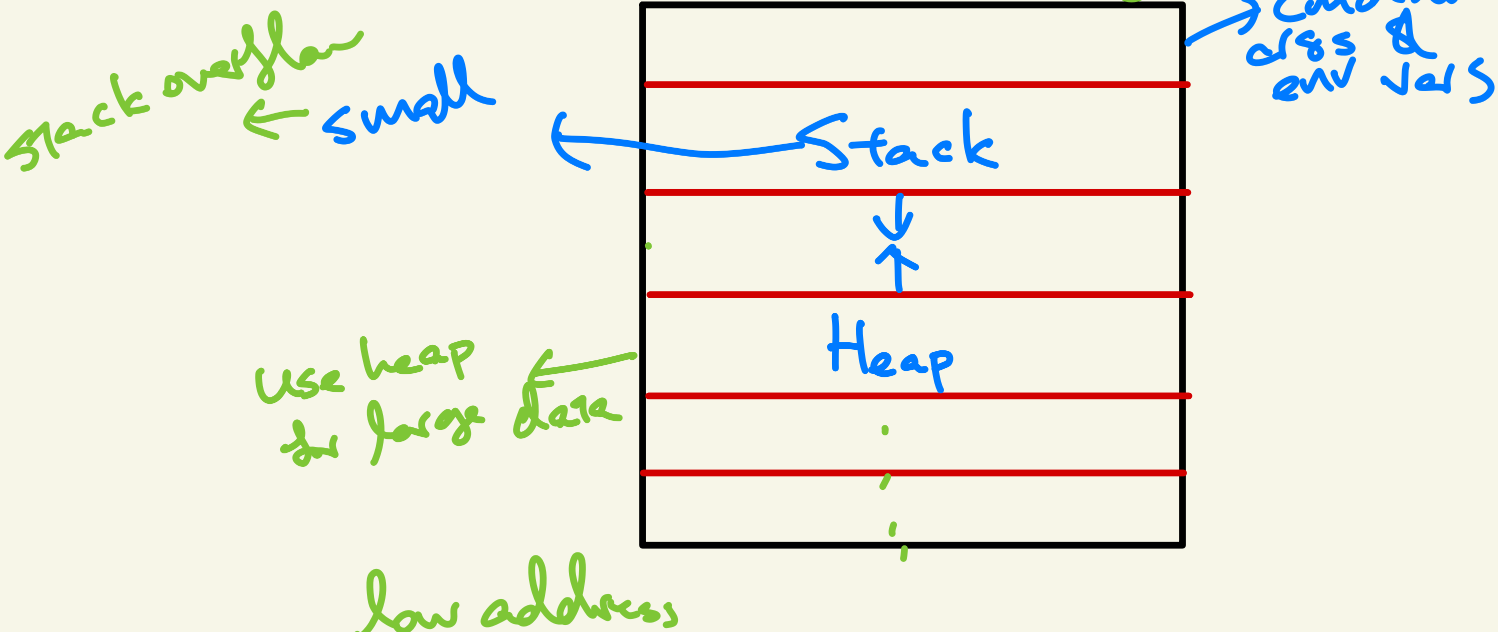
- there are many possible file error conditions
- a class enumeration, **ios::io_state** contains these:
  - **badbit, eofbit, failbit** etc → error states that you can check
- status functions check and set these "stream status bits"
- for file open modes, **ios::open_mode**:
  - **trunc, in, out, app, binary, ate**
- for specialised streams, default values used
- direct (random access) of files is possible:
  - tell()/seek())
- every file has a stream pointer, indicating current position

done implicitly, but can be controlled as well

Pointers

Pointers and Dynamic Memory Allocation

Memory layout



Why Pointers? .

- Imagine its the 1970's.
    - Your computer has 64KB RAM and a slow CPU
- Copying large objects is **slow**.
- Memory is scarce resource! Avoid extra copies.
    - Rather copy object's memory address around.
    - $2^{16} = 64KB$. 16-bit integer for memory address range.
    - This principle still holds today! References.
- Need to dynamically obtain memory for program.
    - OS returns memory address of allocated memory chunk
    - Memory is scarce resource! Deallocate ASAP when finished with it.
- Hence, **POINTERS**.

Pointers

- Imagine program's memory is a huge array.
- Index into that array is a pointer.
- Usually written in hex: 0XFFFFFFA0 (32-bit address)
- **Pointer** variable holds a **memory address**. General declaration:

```
type * ptrname;
```

also do compile time type checking when deref-ing

- **Value** at address obtained or *dereferenced* using * operator:

Must tell the compiler the type of data stored in the address for memory allocating

```
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.
```

generic pointer

• Either set the correct address, or set it to nullptr (NULL is deprecated)

- be careful when writing to memory – errors!

Pointers

local vars go on stack
can also allocate heap memory

- **&** operator ⇒ address of int **a**

address operator

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

- Initiliase default ptr value to **nullptr**.

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

- NULL SEGFAULTS faster (NullPointerExceptions)

Address	Data Value	Notes
0xFFFFFFA0	5	int a=5;
0xFFFFFFB0	0xFFFFFFA0	int * ptr=&a;

Pointer Arithmetic

- pointers hold memory addresses
- step size deduced from type
- 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; }

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

address of first element

must be careful with the bound

Potential performance gain:
use ++x instead of x++
but compiler will take care of ++, unless you overload the increment operator of a class

- can use array indexing:

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

Address	Data Value	Notes
0xC0000000	0xFFFFFFA0	char * ptr
0xFFFFFFA0	'D'	*(ptr+0);
0xFFFFFFA1	'O'	*(ptr+1);
0XFFFFFFA2	'G'	*(ptr+2);
0xFFFFFFA3	'E'	*(ptr+3);

String doesn't have this problem it's managed by the class

Pointer Indirection

- Pointer Indirection ⇒ pointers to pointers

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

→ we're not talking abt the address of nullptr but the address of the pointer pointing to the nullptr

- add a * for every level of indirection.
- Use brackets to think about the type. **(((char) *)*)***
- from left. **pointer** to a **pointer** to a **pointer** to a char.
- Uses:
  - Old style of pass by reference in C programs. → can change a parameter in a local scope globally
  - Old style multidimensional structures (2D arrays for e.g.)

Pointer Indirection

```
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;
```

Address	Data Value	Notes
0xFFFFFFA0	'a'	char c;
0xFFFFFFB0	0xFFFFFFA0	char * cptr;
0XFFFFFFB8	0xFFFFFFB0	char ** ccptr;
0XFFFFFFBB	0xFFFFFFB8	char *** cccptr;

Dereferencing members of Struct/Class Pointers

- A 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;
```

- Could dereference the class pointer

```
(*node).f = 1.0f;
```

- or use -> operator. Prettier code.

```
node->f = 1.0f;
node->left = node->right = NULL;
cout << node->node_func() << endl;
```

Pointers as function arguments

- Old style of reference passing.
- For interests sake (and C programmers)
- 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.
- Fairly legacy, error-prone arg passing mechanism.

Generic Pointers: void *

- declared: **void * ptr;**
  - can point to any type, but cannot be directly dereferenced!
  - must cast explicitly:

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

- functions can receive and return void pointers.
- If you're using them, you're probably doing it wrong.
- Note that **ALL** pointers have same size!
  - Architecture dependent (32-bit or 64-bit usually)
  - Max addressable memory for 32-bits is 4GB for e.g.

Function Pointers

- function **name** is pointer to code in memory
- function pointer syntax: **return_type (* PtrName)(args);**
- "dereference" with function call:

```
// 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);
}
```

- Superceded by function objects (more to come).

Pointers: comments

- Be sure to understand these things:

```
int a = 5;              // Declare int
int * ptr = nullptr; // Declare int pointer. Init to nullptr
ptr = &a;               // Address operator & returns a's address
int b = *ptr;           // Dereference operator *. Set b=a
*ptr = 6;               // Dereference operator *. Set a=6
```

- Don't confuse declaration and dereference.
- Pointer arithmetic styles:

```
int a[4];
int * ptr = a;
for(int i=0; i<4; ++i) { *(ptr+i) = i; }
for(int i=0; i<4; ++i) { ptr[i] = i; }
for(int * ptr=a, i = 0; ptr!=a+4; ++ptr, ++i) { *ptr = i; }
```

Memory Management

- Why manage memory?
- Why manage resources?
- Why release resources?
- Scarcity leads to good management...
- Eclipse (Java): 384MB, Codeblocks (C++): 44MB

Dynamic Memory Allocation

- C++ does not garbage collect
- Dynamic Memory allocated/deallocated by programmer
    - Other memory managed automatically e.g. variables
- Address of Dynamic Memory stored in a pointer
- Usage: **type * ptrname = new type;**

```
// 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.

should always come in pairs

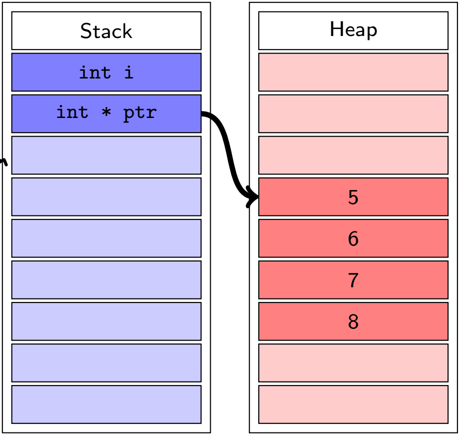
Stack

Vars are name entities ⇒ go on stack

- Dynamic Memory acquired from the **heap**.
- Local Variables and Arguments live on the **stack**.

```
void f(int i)
{
  int * ptr = new int[4];
  for(int j=0; j<4; ++j)
    { ptr[j] = i+j; }
  delete [] ptr;
}
```

pointer on stack, pointing to a location on heap.



Dynamic Memory Allocation

- can use **new** to create dynamic arrays:

```
// 2 rows, different column sizes for each row
int rows = 2; int cols[2] = { 3, 2 };
float ** array = nullptr;
// Allocate an array of float *'s
if((array = new float*[rows]) != nullptr) { // Allocate outer array
  for (int i=0; i<rows; ++i) {
    // Allocate float array, current row determines size
    if((array[i] = new float[cols[i]]) == nullptr)
      { cerr << "Allocation error!"; break; }
  }
}
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
```

row elements are ptrs to a float each

after C++11, the default 'behaviour' is to throw an exception

- Fast code, but invites progammer error and segfaults

Dynamic Memory Allocation: Simpler version

- can use **new** to create dynamic arrays:

```
// 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
```

- Fast code, but invites progammer error and segfaults

Dynamic Memory Allocation

- Pen & Paper. Work out memory allocation

Address	Data Value	Notes
0x00	?	float ** array
0x04		
0x08		
0x0C		
0x10		
0x14		
0x18		
0x1C		
0x20		
0x24		
0x28		

Dynamic Memory Allocation

- Pen & Paper. Work out memory allocation.

Address	Data Value	Notes
0x00	0x08	float ** array
0x04		
0x08	0x14	array[0]
0x0C	0x24	array[1]
0x10		
0x14	1.0f	array[0][0]
0x18	2.0f	array[0][1]
0x1C	3.0f	array[0][2]
0x20		
0x24	4.0f	array[1][0]
0x28	5.0f	array[1][1]

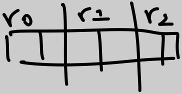
the references take up space aswell

Large multi-dimensional arrays

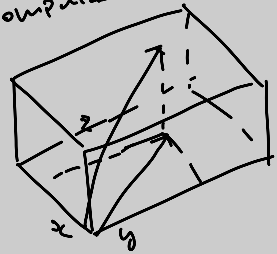
- For simplicity in allocation and types
- Use one big chunk of mem

```
void floatarray3d(int xdim, int ydim, int zdim)
{
  float * ptr = new float[xdim*ydim*zdim];     // Allocate
  for(int z=0, z<zdim; ++z) {
    for(int y=0, y<ydim; ++y) {
      for(int x=0; x<xdim; ++x) {
        // (x,y,z) related business here
        int offset = z*xdim*ydim + y*xdim + x;
        ptr[offset] = 1;
        cout << offset << ' ';
      }
      cout << endl;
    }
    cout << endl;
  }
  delete [] ptr; // Clear up memory
}
```

computational cost?



e.g.



Large multi-dimensional arrays

- Eliminate multiplies in inner loop
- strides + offsets

```
void floatarray3d(int xdim, int ydim, int zdim)
{
  int ystride=xdim; int zstride=xdim*ydim;     // Strides
  float * ptr = new float[xdim*ydim*zdim];     // Allocate
  for(int z=0, zoff=0; z<zdim; ++z, zoff+=zstride) {
    for(int y=0, yoff=zoff; y<ydim; ++y, yoff+=ystride) {
      for(int x=0; x<xdim; ++x) {
        ptr[yoff+x] = 1; // Do business related to (x,y,z) here;
        cout << yoff+x << ' ';
      }
      cout << endl;
    }
    cout << endl;
  }
  delete [] ptr; // Clear up memory
}
```

Large multi-dimensional arrays

- Eliminate extraneous add ops in for loops

```
void floatarray3d(int xdim, int ydim, int zdim)
{
  int ystride=xdim; int zstride=xdim*ydim; // Strides
  int fullstride=xdim*ydim*zdim;
  float * ptr = new float[fullstride];     // Allocate
  for(int zoff=0; zoff<fullstride; zoff+=zstride) {
    for(int yoff=zoff; yoff<zoff+zstride; yoff+=ystride) {
      for(int xoff=yoff; xoff<yoff+ystride; ++xoff) {
        ptr[xoff] = 1;
        cout << xoff << ' ';
      }
     cout << endl;
    }
    cout << endl;
  }
  delete [] ptr; // Clear up memory
}
```

Dynamic Allocation: comments

- Use Pointers + Dynamic Allocation to create complex objects
- Generally use in 2 ways
- Allocate chunk/array of objects
  - Traverse with pointer arithmetic
  - Generalises to container iterators later

```
int * ptr = new int[4];
for(int i=0; i<4; ++i) { *(ptr+i) = i; }
delete [] ptr;
```

- Allocate single object
  - Create linking structures.
  - linked lists, binary trees etc.

```
node * head = new node;  // Allocate linked list head node
head->next = new node;  // Allocate next ll node
delete head->next;      // Delete next ll node
delete head;            // Deallocate ll head node
```

Pointers: Conceptual Issues

- Pointers have two use cases:
  1. Point at/Reference object at a memory address.
  2. Hold the address of dynamic allocation.
- Need to call **delete** in the second case.
- Syntax does not distinguish between
  - Single dynamically allocated object.
  - Dynamically allocated array of objects.

```
node * head = new node; delete head;
int * ptr = new int[4]; delete [] ptr;
```

- *Confusion*. At each level of indirection, Array or Single Object ?

```
node *** head;
```

- You must know this, the compiler will not save you.

Resource Acquisition is Initialisation (RAII)

The C++ Memory Model Resource Acquisition is Initialisation

{
Acquire Resource
Use Resource
Release Resource
}

The Circle of Life in C++: (Creation and Destruction)

- C++ Memory Model pairs Object Construction and Destruction within same scope. **Resource Acquisition is Initialisation**
- Variables declared as follows:

```
for(int i=0; i<N; ++i)
{                     // scope begins
  record r;           // Construct a record r
}                     // scope ends. Destroy record r (and i)
```

- are destroyed *automatically* when scope ends.
- Hence the term **automatic variable**.
- Applies to both class, function and code block scope.
- Important for guaranteeing **exception safety**.

The Circle of Life in C++: (Creation and Destruction)

- Two examples of scope: Function and Class

```
void f(int N) {
  record base;
  for(int i=0; i<N; ++i) {
    // temp constructed.
    record temp;
    // scope ends. temp
    // destroyed automatically
  }
  // scope ends. base
} // destroyed automatically
```

```
class record {
private:
  int N;
  metadata meta;
  std::vector<std::string> lines;
  // when a record is destroyed,
  // N, meta and lines
  // destroyed automatically
};
```

- Wave of creation and destruction as scope expands and recedes.
- Automatic variables automatically destroyed. Have predefined behaviours to destroy themselves
- Anything special (like memory allocated to pointers or other resources) must be managed.

Resource Acquisition is Initialisation

- Useful discusison links:
  - http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization
  - http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Resource_Acquisition_Is_Initialization
  - http://www.hackcraft.net/raii/

Automated Pointer Management

Automated Pointer Management

or – *No Naked Pointers*

- std::unique_ptr<T>– Unique ownership
- std::shared_ptr<T>– Shared ownership

e.g. for trees. But for DAG you might face issues because of cycles.

Wrapping Pointers

- Dynamically Allocated Memory isn't managed by the RAII paradigm.
- If it was:
  - No more explicit deletes
  - Important for Exception Safety.

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

- Solution: encapsulate and guarantee pointer release (RAII).

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

→ takes care of garbage management when going out of scope.

unique_ptr

- **unique_ptr** wraps a pointer in **automatic** variable.
- Automatically deletes pointer when unique_ptr leaves scope.

```
#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
```

- Just as efficient as normal pointers. **Zero Overhead**.
- **USE** for holding dynamically allocated memory.
- Separate template parameters for single objects and arrays

```
auto single = make_unique<T>(arg);       // C++14 style
std::unique_ptr<T> single(new T(arg));  // C++11 style
std::unique_ptr<T []> array(new T[10]); // C+11 style arrays
```

- Safe for use in STL containers.

unique_ptr Usage patterns

- 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
```

Unique Ownership

→ move makes things movable! = is the part that is actually wrong

- They cannot be copied, only **moved**. copy operator= 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.

unique_ptr

- Class only composed automatic variables.
  - All resources managed by objects
  - Pointer programming starts getting simpler.
  - RAII
  - No destructor needed!
  - Can only move by default, need to implement copy.

```
class buffer {                    // Class only has
private:                          // automatic variables
  std::unique_ptr<int []> a;      // No destructor needed.
  int size;
public: // unique_ptr holds result of dynamic memory allocation
  buffer(int size=10) : size(size), a(new int[size]) {}
  resize(int new_size) // Resize the array
    { a.reset(new int[new_size]);
};
```

unique_ptr

- Encourages a "hot potato" style of coding with dynamically allocated memory.

```
// Return unique_ptr to dynamically allocated object
unique_ptr<pirate> factory_method(int doubloons)
  { return unique_ptr<pirate>(new pirate(doubloons)); }
// Takes unique_ptr as argument and returns one too
unique_ptr<pirate> hihosilveraway(unique_ptr<pirate> pirate_ptr) {
  pirate_ptr->plunder(100);
  return std::move(pirate_ptr);
}
unique_ptr<pirate> ptr = factory_method(10);
// Pointer ownership transferred to argument of hihosilveraway
unique_ptr<pirate> richer_ptr = hihosilveraway(std::move(ptr));
ptr->avast("!!!!"); // Fails. Lost the potato.
richer_ptr->avast("!!!!!!!!!!!!!"); // Succeeds.
```

- Only one unique_ptr can hold the pointer at any time.
  - Responsibility for pointer is explicitly mandated by this mechanic.

shared_ptr

- Next logical step is reference counted pointers.

more expensive

```
#include <boost/shared_ptr.hpp> // C++03
#include <memory>               // C++11

shared_ptr<pirate> pirate_ptr_1 = maked_shared<pirate>(100); // 1
shared_ptr<pirate> pirate_ptr_2 = pirate_ptr_1;  // Ref count 2
shared_ptr<pirate> pirate_ptr_3 = pirate_ptr_2;  // Ref count 3
pirate_ptr_1->yaaaaaaaaaar();    // Fine
pirate_ptr_2->yaaaaaaaaaar();    // Fine
pirate_ptr_3->yaaaaaaaaaar();    // Fine
```

work on same obj.

obj kept alive till ref counts == 0.

shared_ptr

- Fits nicely into RAII paradigm
  - When shared_ptr is copied/copy constructed, ref count incremented.
  - When shared_ptr destroyed, ref count decremented.
- If count reaches 0, managed pointer deleted.

can't do ptr offsetting

```
{
  shared_ptr<pirate> p1 = maked_shared<pirate>(100); // Count 1
  {
    shared_ptr<pirate> p2 = p1;                      // Count 2
    {
      shared_ptr<pirate> p3 = p2;                    // Count 3
    }  // Count 2
  }    // Count 1
}      // Pointer deleted
```

- Use sparingly, most of the time you don't need them.
- Extra overhead from pointer indirection + count maintenance.
- Use when lifetime of allocated object is uncertain.

shared_ptr cycles

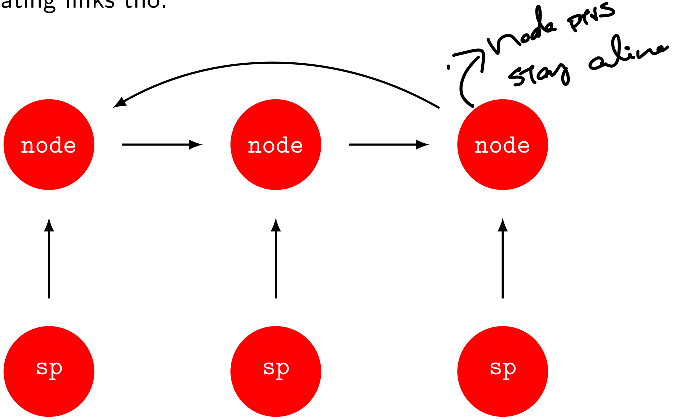
- **shared_ptr**'s can be subject to **cycles**.
- One wouldn't usually construct a linked list with shared_ptr's but:

```
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,
  // No named variable has access to the shared_ptr internals
  // memory leaks
```

right way to do this is with unique_ptr

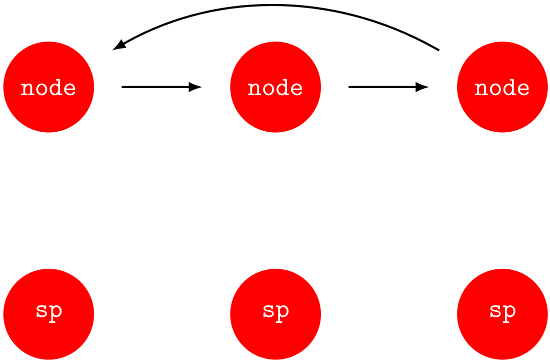
Automated Pointer Management

- Using `shared_ptr<node>` for next implies that node *has some responsibility* for allocated memory.
- We're just creating links tho.



Automated Pointer Management

- When A, B and C leave scope, internal nodes still *have responsibility* for allocated memory



weak_ptr

- Use **weak_ptr**'s
  - to break cycles. only goal
  - create links.
  - point to allocated memory without asking for responsiblity
- **weak_ptr**'s point to **shared_ptr**'s.
- **weak_ptr**'s don't increment/decrement the count.
- To use **weak_ptr**'s, *promote* to **shared_ptr**. to lock the thing in place, otherwise it might have vanished.

```
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
```

weak_ptr

- Make *links* between nodes weak_ptr's.

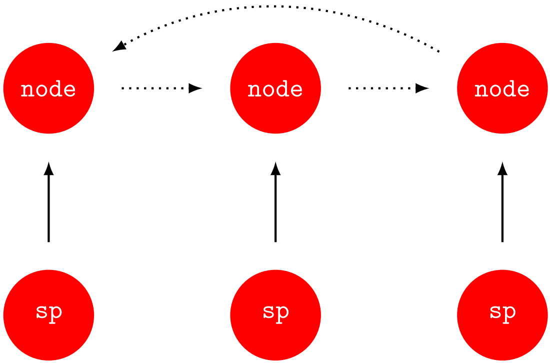
```
class node {
public:
  std::weak_ptr<node> next;
};
...
{
  shared_ptr<node> A = make_shared<node>(); // Count of 1
  shared_ptr<node> B = make_shared<node>(); // Count of 1
  shared_ptr<node> C = make_shared<node>(); // Count of 1
  A->next = B; B->next = C; C->next = A; // No cycles
  // Counts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

- Then can do:

```
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;
}
```

Automated Pointer Management

- Dotted lines are `weak_ptr`'s.



Automated Pointer Management

- Know when you're holding memory and when you're linking to objects in memory
  - **unique_ptr** for mandating sole responsibility of held pointer.
  - **shared_ptr** for mandating shared responsiblity for shared pointer.

when passing by value to a, say method, count ↑

when passing by reference / &, then count stays the same,

Check share_ptr. CPP example.

Value and Reference Semantics

Value 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.

l-values and r-values

- Distinguish between persistent and temporary values.

```
Matrix 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** persist. Can be referred to later.
  - e.g. **A**, **B** and **C**.
  - Objects with names.
- **r-value** do not persist. Think RAII.
  - e.g. `B + C + D`, `Matrix(1.0,2.0,3.0,4.0)`, `multiply(C,D)`
  - Temporary values.

l-value & vs r-value && references

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

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

→ not the same as address

- **r-value references (&&)** bind to unnamed, temporary, "about-to-die" variables.

```
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. which is
Matrix && A = multiply(B, C); // about to die due to RAII
```

→ reference will keep the temporary alive till out of scope

- We can "move/steal" these variable's **values** before their destruction.
- Can obtain r-value ref to an automatic var using move.

```
Matrix A;
Matrix && Arref = std::move(A);
```

Value vs Reference Semantics

- Automatic Variable

```
Object o;
```

- Copy construct variable.
- Copy the **value**.

```
Object o;            // Copy o
Object ocopy = o;    // into ocopy
```

- Modifying o doesn't modify ocopy.

```
o.id = 100;
ocopy.id = 200;
(o.id == ocopy.id) == false;
```

- Reason with **values**.

discussed essentially addresses

- Reference to automatic var.

```
Object & oref;
```

- Construct reference from var.
- Copy the **identity**

```
Object o;            // Reference o
Object & oref = o;   // using oref
```

- However, modifying o does modify oref.

Same piece of memory being referred to

```
o.id = 100;
oref.id = 200;
(o.id == oref.id) == true;
```

- Reason with **identities**.

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 tho!

```
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)

Values vs Reference Semantics

- C++ **operator=** implies **deep** copy. Java's implies **shallow**.
- Value Semantics, deep copy entire object (value), slow if large.

```
myobj first;      // C++. Create 2
myobj second;     // objects. Copy
first = second;   // 2nd into 1st.
```

```
myobj first = new myobj(); // Java. Create
myobj second = new myobj();// two objects.
first = second.clone();    // Copy 2nd to 1st
```

- Reference Semantics, just copy memory address (identity)

Move semantics have been made more tolerable to passing by values

```
myobj first; // C++. Bind ref to
myobj & second = first; // first
```

```
myobj first = new myobj(); // Java
myobj second = first;   // Copy reference
```

- Passing reference to arg vs value is highly efficient.

```
void f(myobj obj); // C++, pass
...                // by value
myobj m;      // Implied deep copy
f(m);         // of m into obj arg
```

```
void f(myobj & obj); // C++, pass
...                  // by reference
myobj m;
f(m); // Copies mem address into obj
```

- Arg assigned 64-bit address (**identity**) instead of copying **value**.
- However, compilers are very good at **eliding** copies.

References

- Can only be constructed once, and never re-assigned.

```
Object first;
Object second;
Object & oref1 = first; // oref1 references first.
// Doesn't reference second now
// Deep copies second's value into first
oref1 = second;
Object & oref2 = oref1; // oref1 also references first.
```

- Pointers are example of reference semantics. Store memory address (**identity**)

```
int i = 10;        // Create pointer iptr, and
int * iptr = &i;   // assign it the memory address of i
```

- **Confusion:** & on expression
  - **lhs** is a reference.
  - **rhs** is "address of operator". Returns variable's memory address.

Basil Exposition: an arbitrary Potterverse resource

- Define C-style library that manages wands.
- Number limited. Must release them. Can't release twice.

```
int acquire_wand(void);
void set_charges(int wand, int charges);
int get_charges(int wand);
void release_wand(int wand);
```

- Define *null*/non-existent wand to be -1.
- Apart from the memory required to represent an integer,
- Consider the wand **value** to be a resource.

Basil Exposition: an arbitrary resource

- Think RAII......

```
{
  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 :(
}
```

- Your OS hands out file handles, sockets, threads, mutexes etc. in this way...
- And wants them back.

Implementing RAII and Value Semantics

- Six Special Member Functions
- They are:
  1. Default Constructor
  2. Copy Constructor
  3. Move Constructor
  4. Copy Assignment Operator
  5. Move Assignment Operator
  6. Destructor
- Compiler creates them even if you don't.
- Collectively define the behaviour for object
  1. creation
  2. copying (deep)
  3. moving
  4. cleanup

Special Member Functions

- If your class has automatic variables that can be copied and moved.
  - Then the compiler will generate **defaults** for you.
  - You can explicitly ask for **defaults**.

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

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

- Or disallow them with the **delete** keyword.

Special Member Functions: Rule of Five

- Should manually implement if class manages special resource.
- If we manually define **one** of these:
    1. Copy or Move Constructor
    2. Copy or Move Assignment Operator
    3. Destructor
- Then we should probably define all five.
- i.e. if we have wands, we should:
    1. manually acquire (construct)
    2. manually copy resources (deep copy)
    3. manually move resources
    4. manually free resources (destroy)

Special Member Functions: The Default Constructor

- Sensibly construct an object with no arguments

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

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

- Invoked as follows. Important for arrays.

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

Special Member Functions: The Default Constructor

- Always try use initialiser list vs constructor body in constructors.

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

- C++ auto vars must be constructed. Java refs can be nulled.
- Member vars always constructed in initialiser list.
- Default constructed if you don't explicitly construct.

```
class student {                // Using C++ constructor body to init
  public student(void)         // is like the following java code
    { name = new String(); name = new String("Harry"); }
}
```

- Can 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
```

Special Member Functions: The Destructor

- Release resources managed by object...
- Invoked at end of scope. **Deterministic** cleanup.
- Java **finalize** 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 automatically calls destructors of name. Memory freed.
- **But**, the wand value (resource) must be manually released.

Special Member Functions: The Destructor

- Now compare

```
{
  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
```

- Achieve RAII functionality for wand resource by wrapping in a class.
- Class has **responsiblity** for the resource. **Zero-overhead.**

Special Member Functions: The Copy Constructor

- Constructs by copying another object.

```
student harry1;          // Default constructor called
student harry2 = harry1; // Copy constructor invoked
student harry3(harry2);  // alternative 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 wand
};
```

- Delegate to **name** copy constructor in initialiser list.
- How do we construct **wand** and set its charges?

Special Member Functions: The Move Constructor

- Constructs, by moving **resources** from another object, **rhs**.
- **rhs** usually a temporary 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)),
      wand(rhs.wand),  // std::move(int) just copies anyway!
  { rhs.wand=-1; }                // We've taken rhs' wand. Null so
                                  // destructor won't try release!

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

- Why don't we call set `set_charges`?

Special Member Functions: Move Constructor Subtleties

- Want to call move constructors for **name** member.

```
  student(student && rhs) : name(std::move(rhs.name)),
      wand(rhs.wand),  // std::move(int) just copies anyway!
  { rhs.wand=-1; }           // We've taken rhs' wand. Null so
...                          // destructor won't try release!
string(const string & rhs); // (1) string copy constructor
string(string && rhs);      // (2) string move constructor
```

- BUT, **rhs**, an r-value ref, **binds** to something unnamed (an r-value).
  - *Confusion*. **rhs.name** is an l-value.
  - Thus **name(rhs.name)** calls string copy constructor (1).
  - Hence **name(std::move(rhs.name))** so string move constructor is called (2).
- **wand** is different: how do you move an int really?

Special Member Functions: The Copy Assignment Operator

- Copies contents of one object to another
- **Releases existing resources**.
- Overloads the "=" operator!
- "=" does different things for each type.
- Differentiate from the 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
```

- Combination of destructor + copy constructor.

Special Member Functions: The Copy Assignment Operator

- Strategy:
  1. Acquire new resources
  2. Release old resources
  3. Assign new resource handles
- one argument, **constant l-value ref** to class type:

avoid copying sth into the same thing

things that require changing the behavior for

not trying to modify

```
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.
}
```

no need to use "this". Have automatic access to all instance vars

∴ also don't need getters/setters

Special Member Functions: The Move Assignment Operator

- Moves contents of one object to another
- **Releases existing resources**.
- 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
```

actual moving move() makes the variable passed in movable

- Combination of destructor + move constructor.

Special Member Functions: The Move Assignment Operator

- 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.
}
```

- NB, `std::move(name)` so that move operator= invokes.

Type Conversion

- C++ is strongly typed (like Java).
- Has both **explicit** and **automatic** type casts.
- Explicit. Please use **static_cast**. → compiler does more type-checking

```
float x = 4.0f;
int i = (int)(x) + 2;           // old C style
int j = int(x) + 2;             // old functional style
int k = static_cast<int>(x) + 2; // new C++ style
```

- Automatic Type coercion. Convert supplied arg into required arg type.

```
class buffer {
  buffer(int size) /* more constructor stuff */ {}
};
void f(const buffer & b) { /* more function stuff */ }
f(100); // Makes a temporary buffer(100).
```

- Disable this behaviour with the **explicit** keyword.

```
class buffer {
  explicit buffer(int size) /* more constructor stuff */ {}
};
```

Type Conversion

- Constructor which takes argument of *different* type
- Form: **ClassName(const Type& rhs);**
- will be called automatically or when explicit cast is performed:

```
class Matrix3i {}; // 3x3 matrix of ints
class Matrix3f     // 3x3 matrix of floats
{
public:
  Matrix3f(const Matrix3i & rhs)
          : /* init float values with ints */ {}
  Matrix3f & operator=(const Matrix3i & rhs)
          { /* copy assignment op */ }
};
...
Matrix3i f, g;
Matrix3f A = f; // implicit conversion occurs
Matrix3f B = Matrix3f(g); // explicit cast
```

not actually assigning!. Copying constructing

Const Correctness

- **const** keyword: Specifies whether variable may be modified.

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

- C++ assumes all methods modify the object.
- declare **const** methods: will work on **const** objects + refs.
- **const** methods can also be applied to 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; };
}
```

disabled on a const obj

not returning const.
simply saying the method can work on both const and non-const obj

- Then:

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

- May not modify class members! Like name.

Const Correctness

- **const** references are frequently used to specify read only access to objects.
- in function arguments for example.

```
std::ostream & operator<<(std::ostream & out, const student & s) {
  out << s.get();
  s.set("Hermione"); // Why are you trying to do this?
  return out;
}
```

- Contrast with

```
std::istream & operator<<(std::istream & in, student & s) {
  std::string name;
  in >> name;
  s.set(name);        // OK, s is not const
  return in;
}
```

- So its a good idea to properly implement **constness** in your classes.
- Defines a read/write interface on your class methods.

Function Arguments

- Pass by **constant l-value ref** if only reading.

```
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 */ }
```

passing by value, so we don't really need this. Can just it directly

- Actually, pass by **value** may become the standard way of doing things!
  - Move semantics + copy elision eliminate unnecessary copies.

Function Return Values

→ compiler likes this more

- Constructing a new object, return by **value**. Try for this generally.

```
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; }
```

↘ must return by ref

- and **const refs** to class members.

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

↘ things returned can't be reassigned etc

- **non-const ref** gives access to class internals. might be bad...

Containers and Iterators

Containers and Iterators

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

Containers

- Containers hold elements (objects/simple) **TEMPLATED**.
- Code stamped out for each type. inlined. efficient.

```
// vector
#include <vector> // resizable array, with random access
vector<int> V(3); // create with 3 default ints, else empty
// list
#include <list>   // linked list. O(n) access
list<Animal> A;   // empty list of Animals
//set
#include <set>    // holds unique values
set<int> S;       // red-black tree O(log n) access
// map
#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
// unordered set
#include <unordered_set>
unordered_set<int> S;          // set backed with hash table
//unordered map
#include <unordered_map>
unordered_map<string, int> m; // assoc map backed with hash table
```

Iterators

- Containers are **heavyweight**
  - Contain lots of data.
- Iterators are **lightweight**.
  - Small class.
  - **References** a data element within container.
  - Methods to move between container elements.
  - Flat view of container

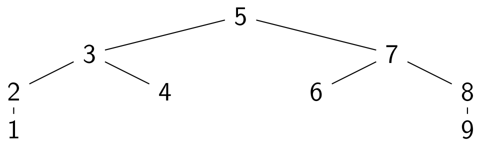
```
vector<int> data = { 6, 8, 2, 4, 0 };
for(vector<int>::const_iterator i = data.begin();
                                i != data.end(); ++i)
  { cout << *i << endl; }
for(auto const & ref : data)
  { cout << ref << endl; }
```

Iterators in a Binary Tree

- **set** is a red-black tree underneath.
- **ordered** by element.

```
set<int> data = { 5, 3, 6, 1, 9, 7, 2, 4, 8 };
// Prints out 1 2 3 4 5 6 7 8 9
for(auto & ref : data) { cout << ref << ' '; }
```

- Iterators visit tree **inorder**.



Nested Classes

- Define classes within the namespace of another class.
- Similar to Java **static** inner classes only.
- Not inner class: Doesn't have reference (**this**) to enclosing class.

```
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.

Iterators

- Implementation of location varies by container.
  - `vector::iterator` stores a pointer.
  - `list::iterator` pointer to node object which has next and prev pointers.
  - `set::iterator` pointer to node object, pointers to left and right children. pointer to node parent probably.

Iterator Access

- Iterators obtained via container `begin()` and `end()` methods

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

- `end()` is iterator pointing at **logical container end**.
- Doesn't reference data, identifies when we've iterated through all container data.
- Move forwards with `++i` and (possibly) backwards with `--i`.
- Move multiples with `std::advance(i, n)`.
- `*i` dereferences the iterator to gain access to container element.

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

- `operator==` and `operator!=` overloaded. Comparisons possible

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

Container Methods

- Various methods modifying containers.
  - `push_back()` for lists/vectors, `insert` for maps.
  - `erase(iterator position)`.
- Iterators identify container location at which op happens.

Vector Container Example

```
#include <vector>
#include <iostream>
using namespace std;
int main(void) {
  vector<int> v = {-1, 0, 2, 4, 6}; // Create a vector with some ints
  v.push_back(10);
  // Use const iterator to traverse vector elements for read-only
  for(vector<int>::const_iterator i=v.begin(); i!=v.end(); ++i)
    { cout << "Number: " << *i << endl; } // Pointer Deref syntax
  // Use iterator to traverse vector elements for read/write
  for(vector<int>::iterator i=v.begin(); i!=v.end(); ++i)
    { *i += 1; } // Pointer dereference syntax
  return 0;
}
```

Vector Container Continued

- vectors continued...

```
// Constructs another vector using a pointer/iterator range
int a[] = {-1, 0, 2, 4, 6}; const int N = sizeof(a)/sizeof(int);
vector<int> w(a, a+N);
// Overloaded operator[]! No range checking
cout << "Element 2 " << is w[2] << endl;
// May throw out_of_range
cout << "Element 2 " << is w.at(2) << endl;
cout << w.size() << endl;
```

- Constant random access, Constant inserts+deletes @ end of vector
- inserts+deletes in middle costly. Array shift **moves** elements.
- Automatically expands. **Moves** existing elements.
- If you know it in advance, set the size:

```
vector<largeobj> w; // Avoid excessive moving from resizing
w.reserve(N);       // Reserve space for N large objects
```

List Container

- Linked List implementation

```
#include <list>
list<int> w = = {-1, 0, 2, 4, 6};
// Exact same iterator semantics as vector
for(list<int>::const_iterator i=w.begin(); i!=w.end(); ++i)
  { cout << *i << endl; }
```

- Good for inserting, deleting, shifting elements.
- Efficiently holds and moves large objects too (via pointers).
- Linear random access tho.

Remember this? (Dynamic Memory Allocation)

```
// 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*cols[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
```

2D Array with variable column sizes

```
// 2 rows, different column sizes for each row
int rows = 2; int cols[2] = { 3, 2 };
vector<vector<float> > array(rows); // NB space between > >
for(int i=0; i<rows; ++i)
{ // Could preallocate column space for this row to avoid vector
  // array[i].reserve(cols[i]); // resizing during push_backs
  for(int j=0; j<cols[i]; ++j) // Init
    { array[i].push_back(float(i*cols[i]+j+1)); }
}
// operator[] returns float& and vector<float> &
cout << array[1][1] << endl;
cout << array.at(1).at(1) << endl; // bounds checking
```

- Some more space overhead compared to bare arrays.
- Access efficiency is *probably the same*.
- Fine if array dimensions are kept static after init...
- And no rearranging rows...

Operator Overloading

Operator Overloading

Operator Overloading

- Motivation

```
class vector // C++
  { /* implementation */};
vector n; vector p; double d;
vector projected_point =
  p - n*(n*p + d)/(n*n);
```

```
class vector // Java
  { /* implementation */};
vector n = new vector();
vector p = new vector(); double d;
vector projected_point =
  p.subtract(n.multiply(
   n.dot(p).add(d) / n.dot(n)));
```

- Overload operators for cleaner code.
- Criticism: Whats happening under the hood?
- Rebuttal: Decent conventions go a long way.
- Lots of this sort of thing in C++ libraries.
  - Linear Algebra, Computational Geometry etc.
  - Container iterators support pointer dereference semantics.
  - Kleene star in BOOST.Spirit parser.

Operator Overloading

- any class member function can be overloaded (except destructor)
- operators can be overloaded (given a new interpretation)
- List of operators, precedence and associativity:
  - http://en.wikipedia.org/wiki/Operators_in_C_and_C++
- can also overload: **(), [], new, delete**
- cannot overload: **. .* :: ?: #**

Operator Overloading: 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;
```

Operator Overloading: 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+=**'s **lhs** arg is non-const → modify + return ref to it.
- `operator+` frequently defined using `operator+=`.
- Avoids code duplication.
- `operator+` and `operator+=` need access to `Matrix` internals.
  - must **friend** them.

Operator Overloading: Class Member Functions

- Can overload via class member function:

```
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. → less general.

Operator Overloading: Contextuality and Unary Overloads

- Operand Types determine which overloaded operator is called (contextual).
- 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);
```

- Prefix scalar multiplication can **only** be implemented as standalone function.
  - Can't overload `operator*` in the "double" class.
- can have unary and binary versions of an operator (such as *)
- unary 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;
```

Operator Overloading: 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 object to hold result.
- **temporaries** to hold intermediate result of **operator+**.

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

- By contrast, **operator+=** just adds **rhs** to current object.
- And returns a reference.

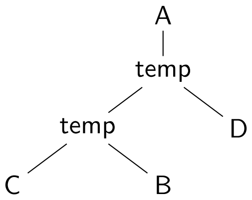
```
Matrix A, B, C, D;
A = D;  // 1 copy
A += C; A += B;
```

Operator Overloading: 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.
```

Operator Overloading: Parenthesis

- Overload parenthesis:

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

Matrix A(2,2); // constructor, don't get confused with
A(0,0) = 0.0;  // the parenthesis operator, invoked on
A(0,1) = A(1,0) = 1.0; // named object, A
```

- reference required so we can use array value as lvalue
- **operator()** can take many arguments
- Used to define **functors** or function objects. More to come.

Operator Overloading: Array subscript

- Array subscript **operator[]** takes one argument only.

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

- But we can chain return values in more complex objects:

```
// 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;
```

Why is ++operator faster than operator++?

- Trivial coordinate class

```
class xcoord { public: int x; };
```

- Prefix returns reference to object **after** increment.

```
xcoord & xcoord::operator++(void) // prefix
  { ++x; return *this; } // Return reference to self
```

- Postfix makes copy of **this** object. Increments **this**. Returns copy by value.

```
xcoord xcoord::operator++(int) const // postfix. NB! dummy int arg
  { xcoord temp = *this; operator++(); return temp; }
```

- Then:

```
xcoordinate c1(5), c2(5);
cout << ++c1; // Prefix version. Outputs 6.
cout << c2++; // Postfix version. Outputs 5.
```

Friend Functions and Classes

- OO requires strict encapsulation of data.
- Practicality requires ways around this...
- In Java, there is the concept of **package private**.
- C++ allows certain 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.
- keyword **friend** indicates this; permission granted by class.
- friend functions are **not** inherited.

Friend Classes

- friend class syntax:

```
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
- Use on tightly coupled classes e.g. containers and iterators
- **vector<int>** and **vector<int>::iterator**.

Friend Functions and Classes

- friend function syntax:

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

- function definition has no **friend** keyword
- **press** can now access **X**'s private members

Friend Functions and Classes: Stream Operators

- Naïve: implement `operator<<` on ostream class.

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

- Can't implement ostream for every class to be and need private access. Try the other way round?

```
ostream & Matrix::operator<<(ostream & rhs) // Silly example
  { os << /* internal members */; return rhs; }
A << cout;         // ostream must now be the rhs
(B << A) << cout; // This breaks bcos of left assoc.
```

- Need 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);
```

Friend Functions and Classes: Symmetric Operators

- Friend functions allow us to define symmetric overloaded operators:
- Can't do `Matrix double::operator*(const Matrix & A)`.

```
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;
C = fact * A; // operator*(fact, A). prefix
C = B * fact; // B.operator*(fact).  postfix
```

- `operator<<` and `operator>>` usually standalone friend functions too.

C++ Inheritance

C++ Inheritance

- can sub-class an existing "parent" or "base" class: add features
- terminology: "derived class" or "sub-class"
- inheritance encourages code re-use (don't re-invent wheel)
- C++ has no special keyword; Java uses **extends**
- C++ does not have the **super** keyword
- C++ supports multiple inheritance: multiple parents
- inheritance syntax:

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

C++ Inheritance

- Supports both **Static** and **Dynamic** Polymorphism.
- **Static** Polymorphism resolved at Compile-time.
    - **C++ uses Static Polymorphism by default.**
    - C++ can redefine functions in Derived classes.
    - C++'s zero-overhead principle in action.
- **Dynamic** Polymorphism resolved at Run-time.
    - Java uses Dynamic Polymorphism by **default**.
    - Explicitly introduce in C++ with the **virtual** keyword.
    - Also need **reference semantics** (pointers/references).
    - C++ Pointers + References to Base objects work similarly to Java refs.

Static vs Dynamic 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"
d->print(); // Output "Derived"
```

```
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
// the base class
Base * b = dynamic_cast<Base *>(
             new Base);
Base * d = dynamic_cast<Base *>(
             new Derived);
b->print(); // Output "Base"
d->print(); // Output "Derived"
```

- **virtual** keyword induces dynamic polymorphism.

Constructors for inherited classes

- A child class has to correctly initialise its parent
- We use the **initialiser list** to do this

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

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

Accessing base members and functions

- inheritance can hide (override) base class variables (functions).
- use **::** operator.

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

- derived class can serve as a parent: original parent is *indirect base*
- **not** inherited: special member functions, friends

Accessing base members and functions

- Access base class **print()**. Redefine **print()** in Derived.

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

- Function signature must match otherwise new function declared.

```
Base b; Derived d;
b.print(); // Outputs "sugar"
d.print(); // Outputs "sugar & spice"
```

- Compiler uses declared type of object pointer/ref to choose method.
- **Static Polymorphism**.

Static Polymorphism

- no run-time binding.

```
class Base {
public:
  void print(void) { cout << "sugar"; }
  ~Base(void) { cout << "Base Destructor" << endl; }
};
class Derived : public Base {
public:
  void print(void) { Base::print(); cout << " & spice" }
  ~Derived(void) { cout << "Derived Destructor" << endl; }
};
// This is all valid code and compiles
Base * b = dynamic_cast<Base *>(new Base);
Base * d = dynamic_cast<Base *>(new Derived);
b->print(); // ?
d->print(); // ?
delete b;
delete d; // ??????????????
```

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);
```

- if **dynamic_cast** fails:
  - and casted type is pointer, returns **nullptr**
  - and casted type is reference, throws **std::bad_cast**

Access Control

- **private** members are not inherited
- **protected** members are inherited, but not visible outside class
- C++ has 3 levels of access control: can modify inherited access
- syntax: **class B : access_specifier A ...;**
- the 3 levels are (**private** members are never inherited):
  - **public**: **public** remain **public**, **protected** remain **protected**
  - **protected**: **public** become **protected**, **protected** remain **protected**
  - **private**: **public** and **protected** become **private**
- Java provides public inheritance only

Access Declarations

- override inheritance access spec using **access declaration**
- restriction: new access cannot be more accessible
- example:

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

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

Composition vs Inheritance

- If you need
  - to *EXTEND/ENHANCE* class functionality
  - 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);
};
```

- Here we say that Child **IS-A** Base object
  - Wherever we used a Base object, we can **always** use a Child object.
  - Can redefine inherited methods.
  - We can add new function and member variables.

Composition vs Inheritance

- **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.
- thus, NewClass doesn't need to be extended.

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 same signature in every sub-class
- do not need virtual keyword in sub-classes

Virtual Function Table

- **virtual** functions supported by **virtual function table**.
- Each class with **virtual** functions backed by function pointer array.
- Point to most *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.

Virtual Functions and Dynamic Binding

- constructors cannot be **virtual**
- destructors should be **virtual** for a **dynamically polymorphic** class.

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

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

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

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; }
};
```

- NB! Works with refs.

```
Derived2 d; Base & b = d;
b.call(); // "Derived2"
```

```
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;
}
```

C++11 Features: Override

- Sometimes coders get function signature in derived class wrong.
- Mechanism 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 {}
};
```

- **override** tells compiler to complain if no override happens.
- Its expecting to override `f(float arg)`.
- Can't find it in the base class.

C++11 Features: Final

- 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
};
```

Abstract Classes

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

Abstract functions

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
// class declaration - should really be in header file}
class Account {
private:
  string name;
  long IDnumber;
  float balance;
public:
  virtual void Debit(float f) = 0;
  virtual void Credit(float f) = 0;
  virtual float GetBalance(void) { return balance; }
  virtual void SetBalance(float f) { balance = f; }
  Account(const string & nm, long ID, float b = 0.0) :
    name(nm), IDnumber(ID), balance(b) {}
};
```

Abstract functions

```
class Maccount : public Account {
public:
  MAccount(string & nm, long id, float b = 0.0) : Account(nm,id,b) {}
  virtual void Debit(float f) override
    { SetBalance(GetBalance() - f); }
  virtual void Credit(float f) override
    { SetBalance(GetBalance() + f); }
};
// etc for other derived classes, e.g. class "Caccount"

// This function works for any sub-class of account
float SumOfBalances(const vector<Account*> & accts) {
  float t = 0.0;
  for (int i = 0; i < accts.size(); i++)
  t += accts[i]->GetBalance();
  return t;
}
```

Abstract functions

```
int main(void) {
  vector<Account*> accts;
  accts.push_back(new Maccount("Bob", 12345, 30000.0));
  accts.push_back(new Caccount("Sally", 12352, 35000.0));
  accts.push_back(new Caccount("Barbie", 223344, 0.0));
  cout << "Sum of Balances is: " << SumOfBalances(accts)<< endl;
  accts[0]->Credit(1000.0);
  accts[2]->Debit(1000.0);
  cout << "Sum of Balances is: " << SumOfBalances(accts) << endl;
  for (int i=0; i < accts.size(); ++i)
    { delete accts[i]; }
  return 0;
}
```