Introduction to Scientific Programming in C++

Ramses van Zon   Scott Northrup

SciNet/Compute Canada

March 15, 2011

Outline of the course

1. Introduction
2. C review (+make)
3. Running example
4. C++ as a better C
5. Big C++ (object oriented programming)
6. Important libraries
7. Further reading. . .

Part I

Introduction

Introduction

Programming strategies

1. Procedural programming
2. Structured programming
3. Object oriented programming

Introduction

Programming strategies

1. Procedural programming
2. Structured programming
3. Object oriented programming

Definition

In procedural programming, one takes the view that a sequence of actions are performed on given data.

Introduction

Programming strategies

1. Procedural programming
2. Structured programming
3. Object oriented programming

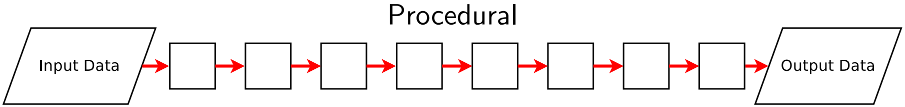
Definition

In procedural programming, one takes the view that a sequence of actions are performed on given data.

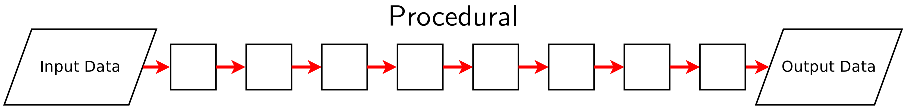
Definition

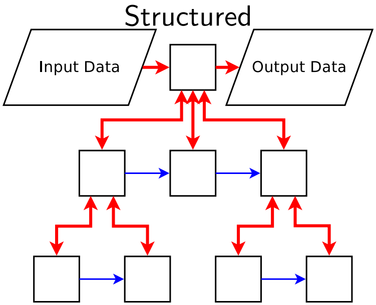
Structured programming uses a systematic break-down of this sequence of actions down to the simplest task level.

Introduction - Procedural and structured programming



Introduction - Procedural and structured programming





Introduction - Procedural and structured programming

Problems

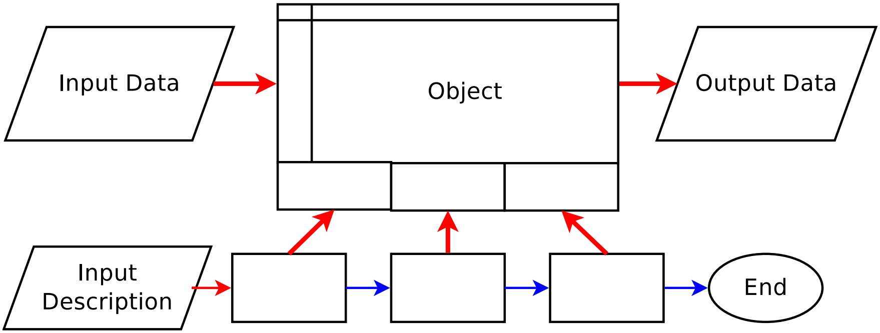
- Complex input data
- Multiple actions to be performed on data
- Separation data+code is bad for reusability
- Leads to reinventing the wheel

One would instead like to build "components" with known properties and known ways to plug them into your program.

Introduction - Object oriented programming

Definition

Object oriented programming treats data and the procedures that act on them as single "objects".



Introduction - Object oriented programming

Advantages

- Complexity can be hidden inside each component.
- Can separate interface from the implementation.
- Allows a clear separation of tasks in a program.
- Reuse of components.
- Same interface can be implemented by different kinds of objects.
- Helps maintanance.

Introduction - Object oriented programming

Advantages

- Complexity can be hidden inside each component.
- Can separate interface from the implementation.
- Allows a clear separation of tasks in a program.
- Reuse of components.
- Same interface can be implemented by different kinds of objects.
- Helps maintanance.

Gotcha: Mind The Cost!

Complexity may be hidden, but you should know:

- the computational cost of the operations
- what temporary objects may be created,
- and how much creating different types of object costs.

On a low level, OOP rules may need to be broken for best performance.

Introduction - Language choice

- You can apply these programming strategies to almost any programming languages, but:
- The amount of work involved in object-oriented or generic programming differs among languages.
- As a result, the extent to which the compiler helps you by forcing you not to make mistakes differs among languages.

C++

- C++ was designed for object oriented and generic programming,
- and C++ has better memory management, stricter type checking, and easier creation of new types than C,
- while you can still optimize at a low level when needed.

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition “The C++ Programming Language”
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members
- 1989 multiple inheritance, abstract classes, static member functions

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members
- 1989 multiple inheritance, abstract classes, static member functions
- 1995 ISO/ANSI C Standard

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members
- 1989 multiple inheritance, abstract classes, static member functions
- 1995 ISO/ANSI C Standard
- 1990 templates

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members
- 1989 multiple inheritance, abstract classes, static member functions
- 1995 ISO/ANSI C Standard
- 1990 templates
- 1993 namespaces, cast operators, bool, mutable, RTTI

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members
- 1989 multiple inheritance, abstract classes, static member functions
- 1995 ISO/ANSI C Standard
- 1990 templates
- 1993 namespaces, cast operators, bool, mutable, RTTI
- 1995 Sun releases Java

Introduction: History of C++

- 1967 *Simula 67:* First object-oriented language by Dahl & Nygaard.
- 1969 – 1973 *C* developed by Dennis Ritchie.
- 1979 *Bjarne Stroustup* writes preprocessor for classes in C
- 1980 Renamed *C with classes*.
- 1983 Now called *C++*.
- 1985 1st edition "The C++ Programming Language"
  C++ supports: *classes, derived classes, public/private, constructors/descructors, friends, inline, default arguments, virtual functions, overloading, references, const, new/delete*
- 1986 Object Pascal (Apple,Borland)
- 1987 pointers to members, protected members
- 1989 multiple inheritance, abstract classes, static member functions
- 1995 ISO/ANSI C Standard
- 1990 templates
- 1993 namespaces, cast operators, bool, mutable, RTTI
- 1995 Sun releases Java
- 1998 ISO C++ standard

source: www.devx.com/specialreports/article/38900

Part II

Review of C

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.

Most basic C program:

```
int main() {
   return 0;
}
```

- main is first called function: must return an `int` .
- C expresses a lot with punctuation.

C review: Language elements

Variables

Define a variable with

```
type name;
```

where *type* may be a

- built-in type:
  - floating point type:
    float, **double**, **long double**
  - integer type:
    short, [**unsigned**] **int**, [**unsigned**] **long int**
  - character or string of characters:
    char, **char***
- structure
- enumerated type
- union
- array
- pointer

C review: Language elements

Pointers

```
type *name;
```

Assignment:

```
int a,b;
int *ptr = &a;
a = 10;
b = *ptr;
```

C review: Language elements

Pointers

```
type *name;
```

Assignment:

```
int a,b;
int *ptr = &a;
a = 10;
b = *ptr;
```

Automatic arrays

```
type name[number];
```

C review: Language elements

Pointers

```
type *name;
```

Assignment:

```
int a,b;
int *ptr = &a;
a = 10;
b = *ptr;
```

Automatic arrays

```
type name[number];
```

Gotcha: limitations on automatic arrays

- There's an implementation-dependent limit on *number*.
- C standard only says at least 65535 bytes.

C review: Language elements

Dynamically allocated arrays

Defined as a pointer to memory:

```
type *name;
```

Allocated by a function call:

```
name = (type*)malloc(sizeof(type)*number);
```

Deallocated by a function call:

```
free(name);
```

- System function call can access all available memory.
- Can check if allocation failed ($name == 0$).
- Can control when memory is gived back.
- Can even resize memory.

C review: Language elements

Dynamically allocated arrays

Defined as a pointer to memory:

```
type *name;
```

Allocated by a function call:

```
name = (type*)malloc(sizeof(type)*number);
```

Deallocated by a function call:

```
free(name);
```

- System function call can access all available memory.
- Can check if allocation failed ($name == 0$).
- Can control when memory is gived back.
- Can even resize memory.

Even better in C++

C review: Language elements

Structures: collection of other variables.

```
struct name {
   type1 name1;
   type2 name2;
   ...
};
```

C review: Language elements

Structures: collection of other variables.

```
struct name {
   type1 name1;
   type2 name2;
   ...
};
```

Example

```
struct Info {
   char name[100];
   unsigned int age;
};
struct Info myinfo;
myinfo.age = 38;
strcpy(myinfo.name, "Ramses");
```

C review: Language elements

Enums

Used to define integer constants, typically increasing.

```
enum name {
   enumerator[=value], ...
};
```

By default, successive enumerators get successive integer values.

- In C, interconvertible with an int.
- Useful to reduce number of **#define**'s.

Unions

Put one variable on top of another; rarely used.

```
union name {
   type1 name1;
   type2 name2;
   ...
};
```

C review: Language elements

Typedefs

Used to give a name to an existing data type, or a compound data type.

```
typedef existingtype newtype;
```

Similar to *existingtype name*; but defines a type instead of a variable.

C review: Language elements

Typedefs

Used to give a name to an existing data type, or a compound data type.

```
typedef existingtype newtype;
```

Similar to *existingtype name*; but defines a type instead of a variable.

Example (a controversial way to get rid of the **struct** keyword)

```
typedef struct Info Info;
```

Then you can declare a **struct Info** simply by

```
Info myinfo;
```

This works become the name **Info** in "**struct Info**" does not live in the namespace of typenames.

C review: Language elements

Functions

Function declaration (prototype)

```
returntype name(argument-spec);
```

Function definition

```
returntype name(argument-spec) {
   statements
}
```

Function call

```
var = name(arguments);
f(name(arguments);
```

C review: Language elements

Functions

Function declaration (prototype)

```
returntype name(argument-spec);
```

Function definition

```
returntype name(argument-spec) {
   statements
}
```

Function call

```
var = name(arguments);
f(name(arguments));
```

Procedures

Procedures are just functions with return type `void` and are called without assignment.

C review: Language elements

Conditionals

```
if (condition) {
   statements
} else if (other condition) {
   statements
} else {
   statements
}
```

```
switch (integer-expression) {
   case integer:
      statements
      break;
   ...
   default:
      statements;
      break;
}
```

C review: Language elements

Loops

```
while (condition) {
   statements
}
```

```
for (initialization;condition;increment) {
   statements
}
```

You can use **break** to exit the loop.

C review: Operators

C has many operators

```
()   []   ->   .
!         ++   --   (type)   -    *    &
*    /    %
+    -
<<   >>   <    <=   >        >=
==   !=
&    ^    |    &&   ||       ?:
=    +=   -=   *=   /=       %=   |=   &=
,
```

C review: Operators

C has many operators

```
()   []   ->   .
!         ++   --   (type)   -    *    &
*    /    %
+    -
<<   >>   <    <=   >        >=
==   !=
&    ^    |    &&   ||       ?:
=    +=   -=   *=   /=       %=   |=   &=
,
```

Gotcha: Bad precendence

Relying on operator precedence is error-prone and makes code harder to read and thus maintain (except for `+`, `-`, `*`, `/` and maybe `%`).

C review: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <stdio.h>
#include "mpi.h"
```

- Include the libraries at link time.
  (not needed for standard libraries)

C review: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <stdio.h>
#include "mpi.h"
```

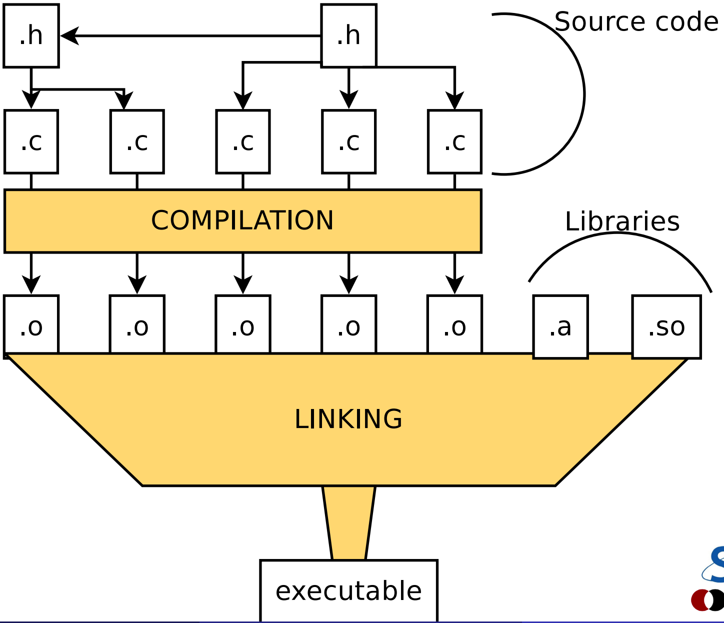
- Include the libraries at link time.
  (not needed for standard libraries)

Common standard libraries

- stdio.h: input/output, e.g.,`printf` and `fwrite`
- stdlib.h: memory, e.g. `malloc`
- string.h: strings, memory copies, e.g. `strcpy`
- math.h: special function, e.g. `sqrt`

Compilation:
Workflow

Compilation workflow



Compiling

Scientific computing = performance: Compile with optimization!

Compiling C from the command-line

If the source is in **main.c**, type

```
$ gcc main.c -O3 -o main
```

or

```
$ icc main.c -O3 -o main
```

Compiling

Scientific computing = performance: Compile with optimization!

Compiling C from the command-line

If the source is in **main.c**, type

```
$ gcc main.c -O3 -o main
```

or

```
$ icc main.c -O3 -o main
```

Compiling C++ from the command-line

If the source is in **main.cpp**, type

```
$ g++ main.cpp -O3 -o main
```

or

```
$ icpc main.cpp -O3 -o main
```

Compilation:

Using make

Compiling with make

Single source file

```
# This file is called makefile
CC = gcc
CFLAGS = -O3
main:  main.c
    $(CC) $(CFLAGS) main.c -o main
```

Compiling with make

Single source file

```
# This file is called makefile
CC = gcc
CFLAGS = -O3
main:  main.c
    $(CC) $(CFLAGS) main.c -o main
```

Multiple source file application

```
CC = gcc
CFLAGS = -O3
main:  main.o mylib.o
    $(CC) main.o mylib.o -o main
main.o:  main.c mylib.h
mylib.o:  mylib.h mylib.c
clean:
    \rm main.o mylib.o
```

Compiling with make

When typing **`make`** at command line:

- Checks if **`main.c`** or **`mylib.c`** or **`mylib.h`** were changed.
- If so, invokes corresponding rules for object files.
- Only compiles changed code files: faster recompilation.

Compiling with make

When typing **`make`** at command line:

- Checks if **`main.c`** or **`mylib.c`** or **`mylib.h`** were changed.
- If so, invokes corresponding rules for object files.
- Only compiles changed code files: faster recompilation.

Gotcha:

Make can only detect changes in the dependencies.
It does not detect changes in compiler, or in system.
But .o files are system/compiler dependent, so they should be recompiled.
So always specify a “clean” rule in the makefile, so that moving from one system or compiler to another, you can do a fresh rebuild:

```
$ make clean
$ make
```

Part III

Our running example, starting in C

mymatrix.h/main.c/mymatrix.c

```
#ifndef _MYMATRIXH_
#define _MYMATRIXH_
/* struct to hold a matrix */
struct matrix {
   int rows,cols;     /* matrix dimensions */
   double *elements; /* points to elements */
};
/* initialize matrix (needed before usage) */
void matrix_construct(struct matrix *m,int r,int c);
/* free memory held by matrix (do after last usage) */
void  matrix_destructor(struct matrix *m);
/* access matrix elements (through a pointer) */
double * matrix_element(struct matrix *m,int i,int j);
/* set all matrix elements to value */
void matrix_fill(struct matrix *m,double value);
#endif
```

```
#include <stdio.h>
#include "mymatrix.h"
void print(struct matrix *m) {
   int i,j;
   for (i=0; i < m->rows; i++) {
      for (j=0; j < m->cols; j++)
         printf("%8.2g ",*matrix_element(m,i,j));
      printf("\n");
   }
}
int main() {
   struct matrix A;
   matrix_construct(&A, 5, 5);
   matrix_fill(&A, 0.0);
   *matrix_element(&A, 0, 0) = 1.3;
   *matrix_element(&A, 4, 3) = -5.2;
   *matrix_element(&A, 1, 3) = -3.3e-4;
   print(&A);
   matrix_free(&A);
}
```

```
#include <stdlib.h>
#include "mymatrix.h"
void matrix_construct(struct matrix *m,int r,int c) {
   m->elements = (double *)malloc(sizeof(double )*r*c);
   if (m->elements==0) exit(1); /* exit program */
   m->rows = r;
   m->cols = c;
}
void matrix_free(struct matrix *m) {
   free(m->elements);
}
double * matrix_element(struct matrix *m,int i,int j) {
   return  m->elements+i*m->cols+j;
}
void matrix_fill(struct matrix *m, double value) {
   int i,j;
   for (i=0; i < m->rows; i++)
      for (j=0; j < m->cols; j++)
         *matrix_element(m,i,j)=value;
}
```

makefile

```
CC = gcc
CFLAGS = -O3 -march=native
main:  main.o mymatrix.o
    $(CC) main.o mymatrix.o -o main
main.o:  main.c mymatrix.h
mymatrix.o:  mymatrix.h mymatrix.c
```

```
$ make
gcc -O3 -march=native -c -o main.o main.c
gcc -O3 -march=native -c -o mymatrix.o mymatrix.c
gcc main.o mymatrix.o -o main
$ main 1.3 0 0 0 0
0 0 0 -0.00033 0
0 0 0 0 0
0 0 0 0 0
0 0 0 -5.2 0
```

Part IV

C++ as a better C

Nice C++ features

1. Comment style
2. Declare variables anywhere
3. Namespaces
4. Improved I/O approach
5. References
6. Improved memory allocation

Nice C++ features: Comment style

- C comments start with `/*` and end with `*/`
- C++ allows comments which start with `//` and last until the end-of-the-line.
- In addition, C-style comments are still allowed.
- C99 shares this nicety.

Nice C++ features: Comment style

- C comments start with `/*` and end with `*/`
- C++ allows comments which start with `//` and last until the end-of-the-line.
- In addition, C-style comments are still allowed.
- C99 shares this nicety.

Example

C:

```
/* This is a C comment*/
```

C++:

```
// This is a C++ comment
```

Nice C++ features: Declare variables anywhere

- C: variables are declared at start of function or file.
- C++: you can mix statements and variable declarations.
- C99 shares this nicety.

Nice C++ features: Declare variables anywhere

- C: variables are declared at start of function or file.
- C++: you can mix statements and variable declarations.
- C99 shares this nicety.

Example

C:

```
double f() {
   double a,b;
   int c;
   a=5.2;
   b=3.1;
   for (c=0; c < 10; c++)
      a+=b;
   return a;
}
```

C++:

```
double f() {
   double a=5.2, b=3.1;
   for (int c=0; c < 10; c++)
      a+=b;
   return a;
}
```

Nice C++ features: Namespaces

- In larger projects, name clashes can occur.

  I had a 3d vector struct called **`vector`**. *Then came along the Standard Template Library, which defined* **`vector`** *to be a general array. Before namespaces, I had to rename* **`vector`** *to* **`Vector`** *in all my code.*

Nice C++ features: Namespaces

- In larger projects, name clashes can occur.

  I had a 3d vector struct called **`vector`**. *Then came along the Standard Template Library, which defined* **`vector`** *to be a general array. Before namespaces, I had to rename* **`vector`** *to* **`Vector`** *in all my code.*

- No more: put all functions, structs, . . . in a namespace:

```
namespace nsname {
   ...
}
```

- Effectively prefixes all of `...` with `nsname::`

Nice C++ features: Namespaces

- In larger projects, name clashes can occur.

  I had a 3d vector struct called `vector`*. Then came along the Standard Template Library, which defined* `vector` *to be a general array. Before namespaces, I had to rename* `vector` *to* `Vector` *in all my code.*

- No more: put all functions, structs, . . . in a namespace:

```
namespace nsname {
   ...
}
```

- Effectively prefixes all of `...` with `nsname::`
- Many standard functions/classes are in namespace `std`.

Nice C++ features: Namespaces

- In larger projects, name clashes can occur.

  I had a 3d vector struct called **`vector`**. *Then came along the Standard Template Library, which defined* **`vector`** *to be a general array. Before namespaces, I had to rename* **`vector`** *to* **`Vector`** *in all my code.*

- No more: put all functions, structs, ... in a namespace:

```
namespace nsname {
   ...
}
```

- Effectively prefixes all of `...` with `nsname::`
- Many standard functions/classes are in namespace `std`.
- To omit the prefix, do "`using namespace nsname;`"
- Can selectively omit prefix, e.g., "`using std::vector`"

Nice C++ features: I/O streams

Standard input/error/output

- Streams objects handle input and output.
- All in namespace **`std`**.
- Global stream objects (header: **`<iostream>`**)
  - **`cout`** is for standard output (screen)
  - **`cout`** is the standard error output (screen)
  - **`cin`** is the standard input (keyboard)

Nice C++ features: I/O streams

Standard input/error/output

- Streams objects handle input and output.
- All in namespace **std**.
- Global stream objects (header: **<iostream>**)
  - **cout** is for standard output (screen)
  - **cout** is the standard error output (screen)
  - **cin** is the standard input (keyboard)
- Use insertion operator **<<** for output:

```
std::cout << "Output to screen!" << std::endl;
```

(**endl** ends the line and flushes buffer)

Nice C++ features: I/O streams

Standard input/error/output

- Streams objects handle input and output.
- All in namespace `std`.
- Global stream objects (header: `<iostream>`)
  - `cout` is for standard output (screen)
  - `cout` is the standard error output (screen)
  - `cin` is the standard input (keyboard)
- Use insertion operator `<<` for output:

```
std::cout << "Output to screen!" << std::endl;
```

  (`endl` ends the line and flushes buffer)
- Use extraction operator `>>` for input:

```
std::cin >> variable;
```

- These operators figure out type of data and format.

Nice C++ features: I/O streams

File stream objects (header: `<fstream>`)

- **`ofstream`** is for output to file.
  Declare with filename: good to go!

```
std::ofstream file("name.txt");
file << "Writing to file";
```

- **`ifstream`** is for input from a file.
  Declare with filename: good to go!

```
std::ifstream file("name.txt");
int i;
file >> i;
```

- Can also open and close by hand.

Nice C++ features: I/O streams

Example

Nice C++ features: I/O streams

Example

C:

```
double a,b,c;
FILE* f;
scanf(f, "%lf %lf %lf", &a, &b, &c);
f = fopen("name.txt","w");
fprintf(f, "%lf %lf %lf\n", a, b, c);
fclose(f);
```

Nice C++ features: I/O streams

Example

C:

```
double a,b,c;
FILE* f;
scanf(f, "%lf %lf %lf", &a, &b, &c);
f = fopen("name.txt","w");
fprintf(f, "%lf %lf %lf\n", a, b, c);
fclose(f);
```

C++:

```
using namespace std;
double a,b,c;
cin >> a >> b >> c;
ofstream f("name.txt");
f << a << b << c << endl;
```

Nice C++ features: I/O Streams

Formatting (header: **<iomanip>**)

- Set width of next output:

```
double d = 14.545;
cout << "[" << setw(10) << d << "]" << endl;
```

```
[    14.525]
```

- Set significant digits of output to follow:

```
cout << "[" << setprecision(3) << d << "]" << endl;
```

```
[14.5]
```

- Set precision of next output:

```
cout << setw(9) << setfill('#') << d << endl;
```

```
#####14.5
```

- Change to scientific notation

```
cout << scientific << d << endl;
```

```
1.454e+01
```

(revert with **fixed**)

Nice C++ features: I/O Streams

Gotcha: text (ASCII) versus binary I/O

While easy, writing ASCII is rarely the best choice in scientific code.
"What is wrong with ASCII," you ask, "isn't it nice that it is readable?"

- ASCII typically doesn't preserve the data's accuracy.
- ASCII typically takes more space than writing binary.
- Writing and reading ASCII is much slower than binary:
  Writing 128M doubles

Format	/scratch (GPFS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173s	174s	260s
Binary	6s	1s	20s

Nice C++ features: I/O Streams

Gotcha: text (ASCII) versus binary I/O

While easy, writing ASCII is rarely the best choice in scientific code.
"What is wrong with ASCII," you ask, "isn't it nice that it is readable?"

- ASCII typically doesn't preserve the data's accuracy.
- ASCII typically takes more space than writing binary.
- Writing and reading ASCII is much slower than binary:
  Writing 128M doubles

Format	/scratch (GPFS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173s	174s	260s
Binary	6s	1s	20s

Writing binary

`std::ofstream` has a **`write(char*,int)`** member function.
`std::ifstream` has a **`read(char*,int)`** member function.
Remember **`sizeof`***!*

Nice C++ features: References

- A reference gives another name to an existing object.
- References are similar to pointers.
- Do not use pointer dereferencing (`->`), but a period `.`
- Cannot be assigned null.

Nice C++ features: References

- A reference gives another name to an existing object.
- References are similar to pointers.
- Do not use pointer dereferencing (`->`), but a period `.`
- Cannot be assigned null.

Standalone definition (rare)

```
type & name = object;
```

- *object* has to be of type *type*.
- *name* is a reference to *object*.
- *name* points to *object*, i.e., changing *name* changes *object*.
- Members accessed as *name.membername* (as you would for *object*).

Nice C++ features: References

- A reference gives another name to an existing object.
- References are similar to pointers.
- Do not use pointer dereferencing (`->`), but a period `.`
- Cannot be assigned null.

Standalone definition (rare)

```
type & name = object;
```

- *object* has to be of type *type*.
- *name* is a reference to *object*.
- *name* points to *object*, i.e., changing *name* changes *object*.
- Members accessed as *name.membername* (as you would for *object*).

Definition as arguments of a function

Nice C++ features: References

- A reference gives another name to an existing object.
- References are similar to pointers.
- Do not use pointer dereferencing (`->`), but a period `.`
- Cannot be assigned null.

Standalone definition (rare)

```
type & name = object;
```

- *object* has to be of type *type*.
- *name* is a reference to *object*.
- *name* points to *object*, i.e., changing *name* changes *object*.
- Members accessed as *name.membername* (as you would for *object*).

Definition as arguments of a function

```
returntype functionname(type & name, ...);
```

Nice C++ features: References

Example

To change a function argument, need a pointer in C:

```
void makefive(int * a) {
   *a = 5;
} ...
int b = 4;
makefive(&b); /* b now holds 5 */
```

Nice C++ features: References

Example

To change a function argument, need a pointer in C:

```
void makefive(int * a) {
   *a = 5;
} ...
int b = 4;
makefive(&b); /* b now holds 5 */
```

C++: can pass by reference using `&`:

Nice C++ features: References

Example

To change a function argument, need a pointer in C:

```
void makefive(int * a) {
   *a = 5;
} ...
int b = 4;
makefive(&b); /* b now holds 5 */
```

C++: can pass by reference using `&`:

```
void makefive(int & a){
   a = 5;
} ...
int b = 4;
makefive(b); /* b now holds 5 */
```

Nice C++ features: References

Gotcha: Avoid copies of objects in function calls

Nice C++ features: References

Gotcha: Avoid copies of objects in function calls

Compare these two functions

```
struct Point3D {
   double x,y,z;
};
void print1(Point3D a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z << std::endl;
}
void print2(Point3D& a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z << std::endl;
}
```

Nice C++ features: References

Gotcha: Avoid copies of objects in function calls

Compare these two functions

```
struct Point3D {
   double x,y,z;
};
void print1(Point3D a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z << std::endl;
}
void print2(Point3D& a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z << std::endl;
}
```

- Calling **print1** copies the content of **a** to the stack (24 bytes).
- Calling **print2** only copies the address of **a** to the stack (8 bytes).
- Memory copies are not cheap!
- If we do this with classes, a so-called constructor is called everytime **print1** is called, whereas **print2** still only copies 8 bytes.

Nice C++ features: Improved memory allocation

Basic allocation

```
type* name = new type;
```

Allocation with initialization

```
type* name = new type(arguments);
```

Array allocation

```
type* name = new type[arraysize];
```

Nice C++ features: Improved memory allocation

Basic allocation

```
type* name = new type;
```

Allocation with initialization

```
type* name = new type(arguments);
```

Array allocation

```
type* name = new type[arraysize];
```

Basic de-allocation

```
delete name;
```

Array de-allocation

```
delete [] name;
```

Nice C++ features: Improved memory allocation

Example

```
struct credit {
   long number, balance;
};
```

Nice C++ features: Improved memory allocation

Example

```
struct credit {
   long number, balance;
};
```

No more of this mess:

```
#include "stdlib.h"
struct credit* a;
double * b;
a = (struct credit*)malloc(sizeof(struct credit));
b = (double *)malloc(sizeof(double )*10000);
...
free(a); free(b);
```

Nice C++ features: Improved memory allocation

Example

```
struct credit {
   long number, balance;
};
```

No more of this mess:

```
#include "stdlib.h"
struct credit* a;
double * b;
a = (struct credit*)malloc(sizeof(struct credit));
b = (double *)malloc(sizeof(double )*10000);
...
free(a); free(b);
```

Instead, simply:

```
credit* a = new credit;
double * b = new double [10000];
...
delete a; delete[] b;
```

HANDS-ON 1:

Use these nice c++ features to rewrite the matrix routines and the main function.

Hands-on 1 - instructions

- Make sure your laptop is connected to the net (see whiteboard).
- Login into scinet using ssh

```
$ ssh login.scinet.utoronto.ca
```

- From there, login into the course's dedicated development node:

```
$ ssh gpc-f109n010
```

- We will use the GNU gcc/g++ compilers in this course, so you'll need to load its module:

```
$ module load gcc
```

Hands-on 1 - instructions - continued

- Make a directory for this course in your home directory, e.g.

```
$ mkdir scinetc++
$ cd scinetc++
```

- Copy the whole directory **/home/rzon/Teaching/scinet/c++/example**

```
$ cp -r /home/rzon/Teaching/scinet/c++/example .
```

  This is the matrix example that we look at after the c review.

- Work from that new directory:

```
$ cd example
```

- Try to build the code

```
$ make
```

- If successful, try to execute the program

```
$ ./main
```

Every with me so far?

Hands-on 1 - instructions continued

- Copy the **example** directory to **example_nice**, and work there:

```
$ cd ..
$ cp -r example example_nice
$ cd example_nice
```

  This will be the first c++ version of the matrix example.

- Rename a the .c files to .cpp files:

```
$ mv main.c main.cpp
$ mv mymatrix.c mymatrix.cpp
```

- Copy the makefile for this set of files:

```
$ cp /home/rzon/Teaching/scinet/c++/example_nice/makefile .
```

- Try to build and run the code

```
$ make
$ ./main
```

Still with me?

Hands-on 1 - instructions continued

Modify the code to use (one at a time):

1. C++ comment style
2. Declarations of iteration variables in for loops
3. Improved memory allocation
4. Improved I/O
5. References

Test that the code builds and runs after implementing each feature.

Hands-on 1 - answers

If you did not quite get there, or if you have a few remaining bugs:

- Copy the c++ version I made, so we can continue later.

```
$ cp /home/rzon/Teaching/scinet/c++/example_nice/* .
```

- Test that the code builds and runs.
- Be sure to look at the source code and see if it make sense to you.

Part V

Big C++

Object oriented programming (OOP)

- **Non-OOP:** functions and data that are accessible from everywhere.
- **OOP:** Data and functions (methods) together in an object.

Object oriented programming (OOP)

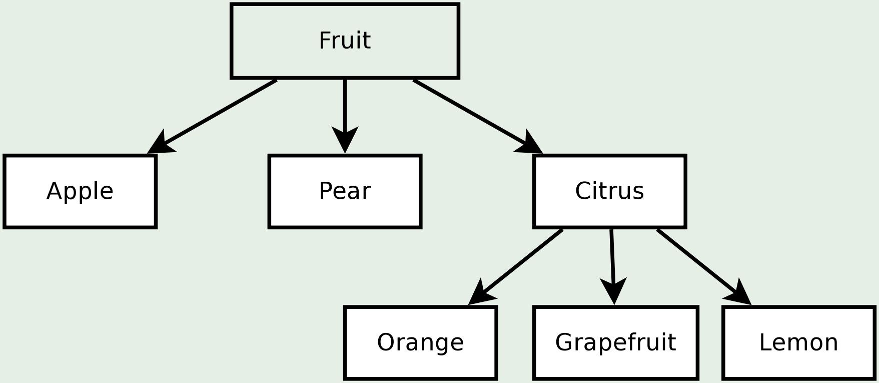
- Data is encapsulated and accessed using methods specific for that (kind of) data.
- The interface (collection of methods) should be designed around the meaning of the actions: abstraction.
- Programs typically contain multiple objects of the same type, called instances.

Object oriented programming (OOP)

- Programs typically contain different types of objects.
- Types of objects can be related, and their methods may act in the same ways, such that the same code can act on different types of object, without knowing the type: polymorphism.
- Types of object may build upon other types through inheritance.

OOP Example

Example (abstract object hierarchy)



OOP Languages

- C++ was one of the earlier languages which supported OOP. (it also supports other programming paradigms.)
- Not the earliest OOP language though: Simula, Smalltalk
- Java, C#, D all came later.
- And one can program in an object oriented fashion in almost any modern programming language (see matrix example in C).

Aspects of OOP in C++

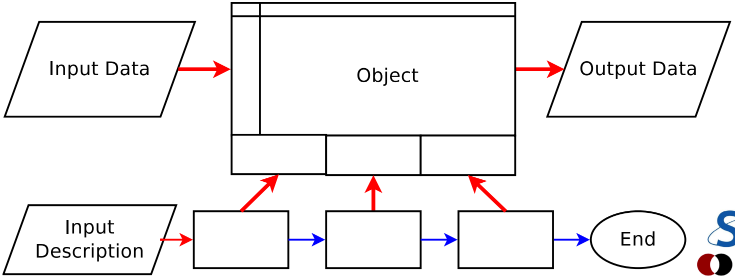
1. Classes and objects
2. Polymorphism
3. Derived types/Inheritance
4. Advanced: Generic programming/Templates

Big C++: Classes and objects

What are classes and objects?

- Objects in C++ are made using 'classes'.
- A class is a type of object.
- Using a class, one can create one or more instances of that class.
- These are the objects.

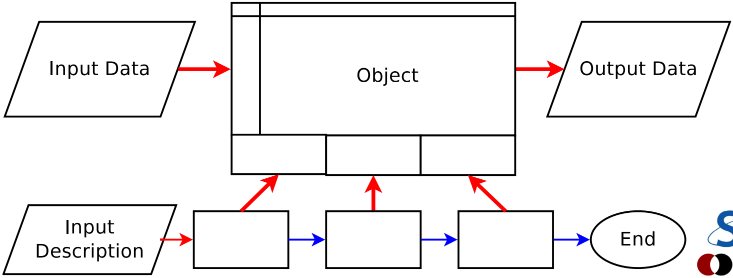
Syntactically, classes are structs with member functions.



Classes: Why structs with member functions?

An object should have properties.

1. A struct can already collect properties of different types.
2. It should be possible to declare several objects of the same type, just as in "`int x,y;`". A struct already constitutes a type definition.
3. Functions on structs often pass a pointer to a struct as a parameter. Embedding functions in structs gives a natural implied parameter.



Classes: How do we add these member functions?

```
class classname {
   public:
      type1 name1;
      type2 name2;
      type3 name3(arguments); // declare member function
      ...
};
```

- `public` allows use of members from outside the class (later more)

Classes: How do we add these member functions?

```
class classname {
   public:
      type1 name1;
      type2 name2;
      type3 name3(arguments); // declare member function
      ...
};
```

- `public` allows use of members from outside the class (later more)

Example

```
class Point2D {
   public:
      int j;
      double x,y;
      void set(int aj,double ax,double ay);
};
```

Classes: How do we define these member functions?

The scope operator ::

```
type3 classname::name3(arguments) {
   statements
}
```

Classes: How do we define these member functions?

The scope operator ::

```
type3 classname::name3(arguments) {
   statements
}
```

Example

```
void Point2D::set(int aj,double ax,double ay) {
   j = aj;
   x = ax;
   y = ay;
}
```

Classes: How do we use the class?

Definition

```
classname objectname;
classname* objectptrname = new classname;
```

Access operator . and ->

```
objectname.name                   // variable access
objectname.name(arguments);       // member function access
objectptrname->name               // variable access
objectptrname->name(arguments);   // member function access
```

Classes: How do we use the class?

Definition

```
classname objectname;
classname* objectptrname = new classname;
```

Access operator . and ->

```
objectname.name                    // variable access
objectname.name(arguments);        // member function access
objectptrname->name                // variable access
objectptrname->name(arguments);    // member function access
```

Example

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.j << std::endl;
```

Classes: How do we use the class?

The **`this`** pointer

- The member functions of a class know what object to work on because under the hood, they are passed the pointer to the object.
- For those cases where the pointer to the object is needed, its name is always **`this`**.
- In other words, in the **`set`** function, **`j`** and **`this->j`** are the same.
- **`this`** is implicitly the first argument to the member function (this will become important in operator overloading later).

Data hiding: The secret agenda of classes

- Good components hide implementation data and member functions.
- Each member function or data member can be
  1. **`private:`** only member functions of the same class can access these
  2. **`public:`** accessible from anywhere
  3. **`protected:`** only this class and its derived classes have access.
- These are specified as sections within the class.

Data hiding: The secret agenda of classes

- Good components hide implementation data and member functions.
- Each member function or data member can be
  1. **private:** only member functions of the same class can access these
  2. **public:** accessible from anywhere
  3. **protected:** only this class and its derived classes have access.
- These are specified as sections within the class.

Example (Declaration)

```
class Point2D {
  private:
    int j;
    double x,y;
  public:
    void set(int aj,double ax,double ay);
    int get_j();
    double get_x();
    double get_y();
};
```

Data hiding: The secret agenda of classes

Example (Definition)

```
int Point2D::get_j() {
   return j;
}
double Point2D::get_x() {
   return x;
}
double Point2D::get_y() {
   return y;
}
```

Example (Usage)

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.get_j() << std::endl;
```

Data hiding: The secret agenda of classes

Gotcha:

When hiding the data through these kinds on accessor functions, now, each time the data is needed, a function has to be called, and there's an overhead associate with that.

- The overhead of calling this function can sometimes be optimized away by the compiler, but often it cannot.
- Considering making data is that is needed often by an algorithm just `public,` or use a `friend` .

Constructors and deconstructors

- A class defines a type, and when an instance of that type is declared, memory is allocated for that struct.
- A class is more than just a chunk of memory.
  For example, arrays may have to be allocated (`new` ...) when the object is created.
- When the object ceases to exist, some clean-up may be required (`delete` ...).

Constructors and deconstructors

- A class defines a type, and when an instance of that type is declared, memory is allocated for that struct.
- A class is more than just a chunk of memory.
  For example, arrays may have to be allocated (`new ...`) when the object is created.
- When the object ceases to exist, some clean-up may be required (`delete ...`).

Constructor

...is called when an object is created.

Destructor

...is called when an object is destroyed.

Constructors

Declare constructors as member functions of the class with no return type:

```
class classname{
      ...
   public:
      classname(arguments);
      ...
}
```

Constructors

Declare constructors as member functions of the class with no return type:

```
class classname{
    ...
  public:
    classname(arguments);
    ...
}
```

Define them in the usual way,

```
classname::classname(arguments) {
  statements
}
```

Constructors

Declare constructors as member functions of the class with no return type:

```
class classname{
     ...
   public:
     classname(arguments);
     ...
}
```

Define them in the usual way,

```
classname::classname(arguments) {
   statements
}
```

Use them by defining an object or with new.

```
classname object(arguments);
classname* object = new classname(arguments);
```

- You usually want a constructor without arguments as well.

Constructors

Example

```
class Point2D {
   private:
      int j;
      double x,y;
   public:
      Point2D(int aj,double ax,double ay);
      int get_j();
      double get_x();
      double get_y();
};
Point2D::Point2D(int aj,double ax,double ay) {
   j = aj;
   x = ax;
   y = ay;
}
Point2D myobject(1,-0.5,3.14);
```

Destructors

Destructor

. . . is called when an object is destroyed.
Occurs when a non-static object goes out-of-scope, or when `delete` is used.
Good opportunity to release memory.

Destructors

Destructor

. . . is called when an object is destroyed.
Occurs when a non-static object goes out-of-scope, or when **delete** is used.
Good opportunity to release memory.

Example

```
classname* object = new classname(arguments);
...
delete object;// object deleted: calls destructor
```

```
{
   classname object;
}// object goes out of scope: calls destructor
```

Destructors

Declare destructor as a member functions of the class with no return type, with a name which is the class name plus a ~ attached to the left.

```
class classname{
      ...
   public:
      ~classname();
      ...
}
```

Define a destructor as follows:

```
classname::~classname() {
   statements
}
```

- A destructor cannot have arguments.

Gotcha: Mixing new/delete and malloc/free

- Trivial objects (plain structs without constructors) can in principle be a allocated with `new` or with `malloc`.
- But pointers allocated with `new` cannot be freed using `free`, and for pointers allocated with `malloc`, `delete` should not be used.
- Non-trivial objects cannot be allocated with `malloc`, since the constructor is not called.
- It is best to stick to `new` and `delete` .

More member functions

. . . to support the class as a new type:

1. Default constructor

   The default constructor is a constructor without arguments.
   If you have no constructors at all, C++ already knows what to do upon construction with no arguments (i.e., nothing), and you do not need to supply a default constructor (but it can still be a good idea).
   If you have any constructors with arguments, omitting a default constructor severely limits the use of the class.

2. Copy constructor

3. Assignment operator

   If the constructor allocates memory, the latter two should be supplied. If there is no memory allocation in the constructor, C++ can generate the copy constructor and assignment operator for you, performing a bit-wise or shallow copy.

Default constructor

Declaration

```
classname {
      ...
   public:
      classname();
      ...
};
```

Definition

```
classname::classname() {
   statements
}
```

This function is needed to be able to

- Declare an object without parameters: `classname name;`
- Declare an array of objects: `name = new classname[number];`

Should set elements to values so that destruction or assignment work.

Copy constructor

Declaration

```
classname {
      ...
   public:
      classname (classname & anobject);
      ...
};
```

Definition

```
classname::classname(classname & anobject) {
   statements
}
```

Used to

- Define an object using another object: `classname name(existing);`
- Pass an object by value to a function (often a bad idea).
- Return an object from a function.

Assignment operator

Declaration

```
classname {
    ...
  public:
    classname& operator=(classname & anobject);
    ...
};
```

Definition

```
classname& classname::operator=(classname & anobject) {
  statements
  return *this;
}
```

- Used to assign one object to another object: `name = existing;`
- But not in `classname name = existing;` calls the copy constructor.
- Returns a reference to this, to allow for the common C-construction

  `name = anothername = existing;`

HANDS-ON:

Convert the matrix structure to a proper c++ class, and rewrite main to use it.

Hands-on 2 - instructions

Copy the whole example_nice directory to example_big

```
$ cp -r example_nice example_big
```

Modify the code to use:

1. Classes instead of structs
2. Member functions
3. Constructors and deconstructors
4. Private member variables

Test that the code builds and runs.

Hands-on 2 - answers

If you did not quite get there, or if you have a few remaining bugs:

- Copy the c++ version I made, so we can continue later.

```
$ cp /home/rzon/Teaching/scinet/c++/example_big/* .
```

- Test that the code builds and runs.
- Be sure to look at the source code and see if it make sense to you.

Big C++: Polymorphism

Poly what now?

- Objects that adhere to a standard set of properties and behaviors can be used interchangeably.
- Implemented by Overloading and Overriding

Big C++: Polymorphism

Poly what now?

- Objects that adhere to a standard set of properties and behaviors can be used interchangeably.
- Implemented by Overloading and Overriding

Why bother?

- Avoid code duplication/reuse where not necessary
- Simplifies and structures code
- Common interface
- Consistency of design should be more understandable
- Debugging

Operator Overloading

- Use expected syntax for non-built in Types
- A = B + C, regardless of what A, B, or C is.

Syntax

Declaration

```
classname {
     ...
   public:
     classname& operator=(classname & anobject);
     ...
};
```

Definition

```
classname& classname::operator=(classname & anobject) {
   statements
   return *this;
}
```

Operator Overloading

Example (Matrix Class)

```
class matrix {
   private:
      int rows, cols;
      double *elements;
   public:
      matrix(int r, int c);
      ~matrix();
      matrix& operator= (matrix &m);
      int get_rows();
      int get_cols();
      void fill(double value);
      matrix operator+ (const matrix &C);
};
```

Operator Overloading

Example (Add two martices)

```
matrix A(5,5), B(5,5), C(5,5);
A.fill(1.0); B.fill(1.0); C.fill(1.0);
for (int i=0, i<row; i++)
   for (int j=0, j<cols; j++)
      A[i][j] = B[i][j] + C[i][j];
```

Operator Overloading

Example (Add two martices)

```
matrix A(5,5), B(5,5), C(5,5);
A.fill(1.0); B.fill(1.0); C.fill(1.0);
for (int i=0, i<row; i++)
   for (int j=0, j<cols; j++)
      A[i][j] = B[i][j] + C[i][j];
```

Example (Add two martices using "+" operator)

```
matrix A(5,5), B(5,5), C(5,5);
A.fill(1.0); B.fill(1.0); C.fill(1.0);
A = B + C;
```

Operator Overloading

"+" Operator

```
matrix matrix::operator+ (const matrix &C) {
   matrix Temp(*this);
   for (int i=0, i<rows*cols; i++)
      Temp.elements[i] += C.elements[i];
   return Temp;
};
```

Operator Overloading

"+" Operator

```
matrix matrix::operator+ (const matrix &C) {
   matrix Temp(*this);
   for (int i=0, i<rows*cols; i++)
      Temp.elements[i] += C.elements[i];
   return Temp;
};
```

"+=" Operator

```
matrix& matrix::operator+= (const matrix &C) {
   for (int i=0, i<rows*cols; i++)
      elements[i] += C.elements[i];
};
```

Operator Overloading

const

- set a constant variable at compile time
- keyword to protect your variables
- const references

Operator Overloading

const

- set a constant variable at compile time
- keyword to protect your variables
- const references

"+=" Operator with bounds checking

```
matrix& matrix::operator+= (const matrix &C) {
   if ( rows == C.rows && cols == C.cols) {
      for (int i=0, i<rows*cols; i++)
         elements[i] += C.elements[i];
   } else {
      cerr<<"Matrix Indicies don't match, can't add";
      exit(1);
   }
};
```

Operator Overloading

"( )" Operator

```
double & matrix::operator() (int &i, int &j) {
     return elements[i*cols + j];
};
```

```
A(1,4) = 6;
double y = A(1,4);
```

Operator Overloading

"( )" Operator

```
    double & matrix::operator() (int &i, int &j) {
          return elements[i*cols + j];
    };
```

```
    A(1,4) = 6;
    double y = A(1,4);
```

"[ ]" Index Operator

```
    double matrix::operator[] (int &i) {
          return elements[i];
    };
```

Operator Overloading

"<<" ">>" Stream Operators

```
std::ostream& operator << (std::ostream& o, matrix& m) {
   for (int i=0; i<m.get_rows() ; i++) {
      for (int j=0; j<m.get_cols(); j++) {
         o << m(i,j) << " ";
      }
      o << std::endl;
   }
};
```

Operator Overloading

"<<" ">>" Stream Operators

```
std::ostream& operator << (std::ostream& o, matrix& m) {
   for (int i=0; i<m.get_rows() ; i++) {
      for (int j=0; j<m.get_cols(); j++) {
         o << m(i,j) << " ";
      }
      o << std::endl;
   }
};
```

```
   std::cout<<"Matrix A = "<< A << std::endl;
```

Operator Overloading

Friends

- friend keyword allows non-member functions access to private data.

Operator Overloading

Friends

- friend keyword allows non-member functions access to private data.

"<<" ">>" Stream Operator using friend

```
class matrix {
 ...
 friend std::ostream& operator<<(std::ostream& o, matrix& m);
};
```

```
std::ostream& operator<<(std::ostream& o, matrix& m) {
   for (int i=0; i<rows ; i++) {
      for (int j=0; j<cols; j++) {
         o << elements[i*cols + j] << " ";
      }
      o << std::endl;
   }
};
```

Operator Overloading

C++ operators available to overload

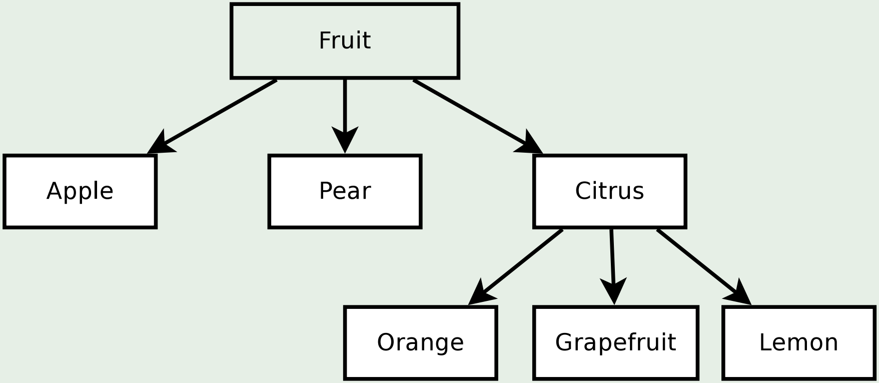
+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|
<<	>>	>> =	<< =	==	!=	<=
>=	&&	||	++	--	->*	,
->	[ ]	( )	new	new[ ]	delete	delete[ ]

HANDS-ON:

Overload **`+=`**, **`()`**, and stream operators for matrix c++ class, and rewrite main to use it.

OOP: Inheritance (Derived Classes)

Example (abstract object hierarchy)



Inheritance

Definition

- child classes are derived from other parent classes
- automatically include parent's members
- inherit all the accessible members of the base class

Inheritance

Definition

- child classes are derived from other parent classes
- automatically include parent's members
- inherit all the accessible members of the base class

Specifics

A derived class inherits every member of a base class except:

- its constructor and destructor
- its assignment operator
- its friends

Inheritance

Syntax

Base Class

```
class baseclass {
   protected:
   ...
   public:
      baseclass ()
      ...
};
```

Derived Class

```
class derivedclass : public baseclass {
   ...
   public:
      derivedclass : baseclass ()
      ...
};
```

Inheritance

Example (Matrix Base Class)

```
class matrix {
   protected:
      int rows, cols;
      double *elements;
   public:
      matrix(int r, int c);
      ~matrix();
      int get_rows();
      int get_cols();
      void fill(double value);
      matrix operator+ (const matrix &C)
};
```

Inheritance

Example (Square Matrix Derived Class)

```
class squarematrix :  public matrix {
   private:
   protected:
   public:
      squarematrix(int r, int c) :  matrix(r,c) {
         if(r!=c) std::cerr<<"not a square matrix"; exit(1);
      }
      double trace() {
         double sum(0.0);
         for(int i=0; i <rows ; i++)
         sum += elements[i*cols+i];
         return sum;
      }
};
```

Inheritance

Example

```
matrix P(5,5);
squarematrix Q(5,5);
P.fill(1.6);
Q.fill(1.6);
std::cout<<" Trace = "<<Q.trace();
```

HANDS-ON:

Come up with a derived class inheriting the matrix class as a base class.

Polymorphism in Inheritance

Idea

- Use base class as framework for derived classes usage.
- Define member functions with virtual keyword.
- Override base class functions with new implementations in derived classes.
- If virtual keyword not used, overloading won't occur.

Polymorphism comes from the fact that you could call the based method of an object belonging to any class that derived from it, without knowing which class the object belonged to.

Inheritance

Example (Matrix Base Class)

```
class matrix {
   protected:
      int rows, cols;
      double *elements;
   public:
      matrix(int r, int c);
      ~matrix();
      int get_rows();
      int get_cols();
      virtual void fill(double value);
};
```

Inheritance

Example (Square Matrix Derived Class)

```
class squarematrix :  public matrix {
   private:
   protected:
   public:
      squarematrix(int r, int c) :  matrix(r,c) {
         if(r!=c) std::cerr<<"not a square matrix"; exit(1);
      }
      double trace();
      void fill(double value) {
         for (int i=0; i < rows*cols; i++)
            elements[i] = value;
      }
};
```

Inheritance

Example (non-virtual)

```
    squarematrix Q(5,5);
    Q.fill(1.6);
    std::cout<<" Trace = "<<Q.trace();
```

Inheritance

Example (non-virtual)

```
squarematrix Q(5,5);
Q.fill(1.6);
std::cout<<" Trace = "<<Q.trace();
```

Example (virtual)

```
matrix *Q;
Q = new squarematrix(5,5);
Q->fill(1.6);
std::cout<<" Trace = "<<Q->trace();
```

Inheritance

Gotcha:

- Virtual functions are run-time determined
- Equivalent cost to a pointer dereference
- Not as efficient as compile time determined (ie non-virtual)
- Should be avoided for many use functions

Inheritance

Gotcha:

- Virtual functions are run-time determined
- Equivalent cost to a pointer dereference
- Not as efficient as compile time determined (ie non-virtual)
- Should be avoided for many use functions

Gotcha:

- Friend keyword allows non-member functions access to private data.
- Useful but does break OOP and will cause problems in inheritance.
- "friends are your enemies"

HANDS-ON:

Modify your derived and base class using the virtual keyword.

Generic Programming

Definition: Generic Programming

Programming style in which specific Types are not specified initially, but instantiated when needed.

Generic Programming

Definition: Generic Programming

Programming style in which specific Types are not specified initially, but instantiated when needed.

Templates

In C++ generic programming is implemented using Templates and instantiated at compile time.

Templates

Syntax

Functions

```
template < typename T > funcname (T &a)
```

Templates

Syntax

Functions

```
template < typename T > funcname (T &a)
```

Classes

```
template < class T >
class classname {
   private:
      T a, b;
   public:
      classname ();
      ...
      memberfunction(T &c, T &d);
};
```

Templates

Example (Double Matrix Class)

```
class matrix {
   private:
      int rows, cols;
      double *elements;
   public:
      matrix(matrix& m);
      ...
      void fill(double value);
      matrix& operator= (const matrix &m)
};
```

Templates

Example (Templated Matrix Class)

```
template <typename T>
class matrix {
   private:
      int rows, cols;
      T *elements;
   public:
      matrix(matrix<T>& m);
      ...
      void fill(T value);
      matrix<T>& operator= (const matrix<T>& m);
};
```

Templates

Example (Templated Matrix Class)

```
template <typename T>
class matrix {
   private:
      int rows, cols;
      T *elements;
   public:
      matrix(matrix<T>& m);
      ...
      void fill(T value);
      matrix<T>& operator= (const matrix<T>& m);
};
```

```
template <typename T>
matrix<T>::matrix() {
   rows = 0; cols = 0;
   elements = new T[0];
}
```

Templates

Example (Calling Templated Class)

```
matrix<double> A(5,5);
A.fill(0.0);
A(0,0) = 1.3;
A(4,3) = -5.2;

matrix<int> B(5,5);
B.fill(33);
B(0,0) = 1;
B(4,3) = 2;
```

Templates

Explicit Instantiation

- Can override generic template abstract-type instantiation for a specific concrete-type.
- Similar to derived class override of base class member function.

Templates

Explicit Instantiation

- Can override generic template abstract-type instantiation for a specific concrete-type.
- Similar to derived class override of base class member function.

```
template<> matrix<double>::fill(double value){
   for (int i=0; i < rows; i++) {
      for (int j=0; j < cols; j++){
         std::cout<<" I'm used for doubles ";
         (*this)(i,j)=value;
      }
   }
}
```

Templates

Gotcha:

- Compile times can be significantly longer.
- Large header files.
- Debugging can be a pain.
- Syntax can get complicated.

HANDS-ON:

Template your matrix class.

Part VI

Important libraries

Important libraries

Don't reinvent the wheel

- It may be interesting to code your own linear algebra solver (say), but is it worth your time?
- There are some good scientific libraries out there.
- The nice thing is, they needn't be c++ libraries, as you can use c libraries in c++.
- Even for basic functionality, there are libraries.

STL: Standard Template Library

Offers a lot of basic functionality

- Supplies a lot of data types and containers (templated).
- Often presented as part and parcel of the C++ language itself.
- Also contains a number of algorithms for e.g., sorting, finding
- Efficiency implementation dependent, and generally not great.

STL: Standard Template Library

Offers a lot of basic functionality

- Supplies a lot of data types and containers (templated).
- Often presented as part and parcel of the C++ language itself.
- Also contains a number of algorithms for e.g., sorting, finding
- Efficiency implementation dependent, and generally not great.

Some of the STL data types

`vector`	Relocating, expandable array
`list`	Doubly linked list
`deque`	Like vector, but easy to put something at beginning
`map`	Associates keys with elements
`set`	Only keys
`stack`	LIFO
`queue`	FIFO
`...`	

STL: Standard Template Library

Example

```
#include "iostream"
#include "vector"
class Grape {
   public:
      int nseeds;
};
int main() {
   using namespace std;
   Grape grapes[10];
   vector<Grape> bunch(grapes,grapes+9);
   bunch.push_back(grapes[9]);
   for (int i=0; i<bunch.size(); i++)
      cout << bunch[i].nseeds << endl;
   vector<Grape>::iterator i;
   for (i=bunch.begin(); i!=bunch.end(); i++)
      cout << (*i).nseeds << endl;
}
```

STL: Standard Template Library

Gotcha: Performance

- The purpose of the STL is not to provide a high performance library, i.e., runtime speed is not the objective.
- Rather it aims to have flexible containers with a uniform usage pattern.
- As a result, using e.g. an **`std::vector`** in an inner loop of you computation, instead of a simple array, can substantially slow down your code (even with the improvements in the implementation since the early days).
- The STL still does not have higher dimensional arrays, and the last thing you want is to have vectors of vectors.

Other useful (scientific) libraries

library	functionality	C++	parallel
MPI	distributed parallel program	✓	✓
OpenMP	shared memory parallelism	✓	✓
Blas/Lapack	linear algebra (in MKL, ESSL)	×	✓ ×
Petsc	matrices, vectors, linear solvers	×	✓
GSL	numerical library	×	×
Boost	continues where STL left off	✓	×
	(+math, statistics, random, blas)		Thread&MPI
IT++	templated matrix implementations	✓	×
Blitz++	(not exhaustive)	✓	×
Armadillo		✓	✓ ×
POOMA		✓	✓
Eigen		✓	×
...			

Again: Don't reinvent the wheel!

Part VII

Further reading

Not covered so we could get to the heart of the matter:

Basic stuff (you'll want to learn these)

- Const correctness
- Booleans
- Inline functions
- Preprocessor
- New names for c header files
- Default parameters

Advanced material

- Initializer lists
- Static class members and enums
- Advanced template parameters
- Abstract base classes
- Multiple inheritance
- Exceptions

Books and links

Books

- *C++ Interactive Course,* Lafore, Waite Group '96
- *C++ FAQs,* Cline, Lomow & Girou, Addison-Wesley '99
- *The C++ Programming Language,* Stroustup, Addison-Wesley '00
- *C+ Templates* Vandervoorde & Josuttis, Addison-Wesley '03
- *Effective C++,* Meyers, Addison-Wesley '03 Addison-Wesley,

Online

- *C++ FAQ,* www.parashift.com/c++-faq-lite
- *C++ Annotations,* www.icce.rug.nl/documents/cplusplus
- *C++ Reference,* www.cplusplus.com/reference

Google is your friend!