Stroustrup Introduction: Prefaces, 1-3

all three prefaces, skim #2, #3

Prefaces

Note emphasis on enjoyment of programming

3rd edition - standardized, and standard library allows programmer to start from a higher level

2nd edition - language supports safe and efficient libraries

safe means type-safe interface

efficient means comparable to hand-written C code

pre-standard library

1st edition -

first page of preface defines a lot of what the language is about

Ch. 1.

assumes you know ordinary programming concepts, but not necessarily OOP concepts

object-based - better abstraction, encapsulation

object-oriented - inheritance & polymorphism

section 1.2 on learning C++

section 1.3 on the design of C++

remark about little or no run-time overhead ... cf LISP

static typing - do checks at compile time - can get safer language without run-time or memory overheads - c.f. LISP again
enable larger programs to be structured ... so that ... a single person to cope with far larger amounts of code
a language also provides a tool for thinking about problems - not a good idea to be too restrictive to save programmer from errors ...

section 1.4 - historical note - a grass-roots, open effort

section 1.5 use of C++

low-level code will work well in C++ - OS levels

big applications where maintainability & extensibility are critical

section 1.6 C and C++

suggestions for C programmers

think of a program as a set of interacting concepts represented as classes and objects, instead of as a bunch of data structures with functions twiddling their bits

suggestions for C++ programmers

experienced with older versions? - probably don't know some of the newer techniques now possible

section 1.7 overview of programming in C++

virtues of using classes and objects

classes = concepts in the problem domain, objects = the objects in the domain

code has a structure and organization more closely resembling the real world

makes design more reliable for complex programs - can depend on the nature of the world for some of the organizational ideas

can give much more compartmentalized code

easier to work with

can give code that is much easier to extend

concept: add more features by ADDING code, not by changing code.

previous code stays in place, doesn't have to be tinkered with

Ch. 2 a tour of C++

Basic orientation

based on C for ubiquity

like C, emphasis on run time efficiency

like C, do as much at compile time as possible, as little extra at run time as possible

better than C, more static (compile-time) type checking and safety

Programming paradigms

Approaches to programming that attempt to make programs easier to design, write, debug, and maintain - making their structure more clear - helping to deal with complexity

As programs get large, need help to work with them - style of how you write the code becomes critical

early programming - write the code any way you want, trying to minimize some property, and try to get it to work, and hope you don't have to modify it

messy, arbitrary coding - what a mess!

languages made subroutines clumsy or inefficient to use, so only used for "libraries" like math sqrt, cos, etc.

constructs for expressing conditions or iterations very limited

e.g. FORTRAN II had only arithmetic IF, DO statement.

machines small, slow, limited memory, so premium on keeping programs small

so tendency to write spaghetti code, with lots of branches and gotos

example

procedural - decide what procedures you want, use the best algorithms you can find

use functions to organize the code for clarity and re-usability

notion of "structured" programming - code should be well structured - what's this mean?

use functions/subroutines so that code is highly organized, and can be read from the top down

no use of goto ... what is goto?

use of iteration constructs like for, while, do-while, where the scope of the loop is clearly marked syntactically.

other ideas - like the one-point-of-return policy - probably excessive if functions are kept short - can get code too convoluted

your main module for Project 1 should do these things - should not be one big ugly function

modular - decide what modules you want; partition the program so that data is hidden within the modules

data-hiding principle

organize the code into groups, modules, try to hide details and data within the module so that it can be used without accidents or confusion, and provide an interface to it

key idea: manage complexity by breaking a complex program down into simpler parts that can be designed, coded, tested, maintained, modified independently of each other.

OO approaches just do this better.

How to do it? Can either do it by "convention" - rules, or get language to help you

Programming by convention - common notion

"convention" - a treaty, an agreement

everybody working on the system, or the community of programmers, will write the code in a certain way, following certain practices.

low-level example: "style" things like all caps for #define macros

all functions that work on C-strings have names starting with "str"

higher level: C file routines work with a struct that keeps all the information about the stream state. has typedef'd name of FILE, and is always referred to by a pointer, FILE *. Programmer using the I/O library will NEVER, EVER, modify the contients of the struct - always call I/O library functions. Conventions have to be documented, passed along, and followed by everybody.

Compiler doesn't know about them, and so can't help enforce them - means people can break the system either through ignorance or by accident
malice is another matter.

Other approach: get the language and the compiler to help you.

Instead of conventions, express key ideas directly in the language - then compiler can detect and point out where not being followed.
e.g. make the FILE contents "private"

In C, can use separate compilation and internal linkage to help with modularization

c.f. Project 1's container module

note how structure of internal data is not visible outside, internally linked functions (static) so that not callable from elsewhere
main module doesn't need to know where and how the data is there

but module-defined types don't behave like "real"types in the language

use of indirection to hide the details means that everything done through a pointer, not like built-in types

namespaces in C++ can be used to "fence off" code and definitions so that their names are in a separate space, and so can't be confused or used accidently.
Exceptions support modularization because they make it easier to separate responsibilities. A flexible way to allow code in a module to report a problem without code outside the module having to know everything that is going on. Can do this with "return codes" but exceptions are better.

user-defined types - decide which types you want, provide a full set of operations on each type.

e.g. complex numbers

geometrical/trigonometric types

add a vector to a point, you get a translated point

a big goal of C++ is to allow user-defined types to be just as convenient and have almost the same status as built-in types

e.g. string object acts for all practical purposes just like it was a primitive type in the language.

being able to overload operators is a critical part of this;

reference type is needed to make it syntactically sensible

Stroustrup calls these concrete types

e.g. simple classes with clear and complete set of operations defined on them

the implementation is not accessible, but is present

very useful, don't underrate them.

object-oriented programming - decide which classes you want, provide a full set of operations, make commonality explicit by using inheritance

important point - not always useful, depends on the domain - e.g. graphics vs numerical work

most people would include use of virtual functions - polymorphism here

abstract types - help decouple interface and implementation

base class describes interface, derived classes have specific implementation

other ways to achieve decoupling - more later

decoupling is a key notion in extensibility

if the main-part of the code is decoupled from the details of the rest of the code, then the rest of the code can be modified, or extended, with little or no impact on the main part

add features by adding code, not by modifying code

generic programming - decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures

note - templates make sense for a strongly typed language only

note - templates are done at compile time, not run time

iterators used with containers and algorithms to abstract from details of container implementations

if you find yourself writing the same code repeatedly except for some small part of it, consider re-doing it with generic techniques

use an object to say that the different kinds of thing are - e.g. a function object

write a template that represents the common structure of the code

instantiate the template with different objects to achieve the results without writing duplicate code

Ch. 3 - a tour of the standard library

std namespace - usage? - handout will be read

The std namespace

Easiest, least enlightened, brute force way to use

#include <iostream>

using namespace std;

Ugliest way to use

always be explicit

std::cout << "Hello" << std::endl;

Best tight rules

#include <iostream>

using std::cout; using std::endl;

never put a using in a header file, always be explicit:

std:: string firstname;

Guidelines

no "using" statements in header files, ever!

this forces your using decision on whoever includes the header - not good

use explicit qualification for all std lib names in the header file

std::ostream operator<< (std::ostream& Mytype m);

std::string;

Only possible exception: if nested within an inline function body or a class declaration.

in implementation (.cpp) files,

#include ... all includes first

your choice:

using namespace std; - not the best, but OK - doesn't avoid name collisions

using std::cout; using std::endl; - my personal favorite

std::cout << x << std::endl; - you gotta be kidding!

main wart: while type-safe, are not necessary safe in other ways - undefined behavior is still present in many cases with iterators

for the usual efficiency reasons

iostream

string - concatenation with +, +=

containers

vector - note range checking policy - his Vec class overloads operator[] and is safe because throws an out-of-range exception

list

map, set - a self-balancing binary tree (red-black tree)

algorithms & iterators

a key idea - using iterators to designate a place in a sequence

operations defined on iterators make them behave with same concepts as pointers

"generalized pointers"

but behave appropriately depending on what kind of sequence they are pointing into

iterator++ -> next cell in vector, next node in list, next node in tree

can define algorithms in terms of iterator operations relatively independently of the details of the container data structure.

algorithms operate on sequences specified by two iterators; start with first, go up to but not including the last

end of a container specified as an iterator one past the end

use * dereference to get the item specified by the iterator

use ++ to change the iterator to point to the next item.

algorithms can also use an iterator for where to put their output

iterator adapters like back inserter can be used to append to the end of a container

can have iterators associated with input and output streams, which allow algorithms to read or write directly - see example p. 61

algorthms can take function pointers or function objects and either apply them or use them as predicates (p 62). special adapater neede for member functions due to the this pointer.

e.g. find, count

see table 3.8.6

math

complex, valarray

the C standard library is included

Note Standard Library header names for C++

C <math.h> is C++ <cmath>

C <string.h> is C++ <cstring>

in general, C Std. Lib. <x.h> is for C++ <cx>