Software Design (C++)

1. Language Technicalities

Juha Vihavainen
University of Helsinki

Preview

- computation: algorithms plus data structures
  - some stuff on **std::vector**
  - using IO streams: states and flags
- handling errors and failures: *exceptions*
- pre-conditions and post-conditions
- a bit on debugging and testing
- *reference* and **const** types
- *namespaces* and headers
- scoped and unscoped *enumerations*
- *overloading* operators

Data for iteration – **std::vector**

- To do just about anything of interest, we need a collection of data to work on. We can store this data in a **vector**. For example:

```
// read some temperatures into a vector:
int main () {
    std::vector <double> temps;      // store temperatures
    double temp;                     // a variable for a value
    while (std::cin >> temp)         // cin reads a value into temp
        temps.push_back (temp);      // store temp in the vector
    // . . .  do something  . . .
}
// action cin >> temp  will indicate true until we reach the end of file ..
// .. or encounter something that isn't a double (meaning here "quit")
```

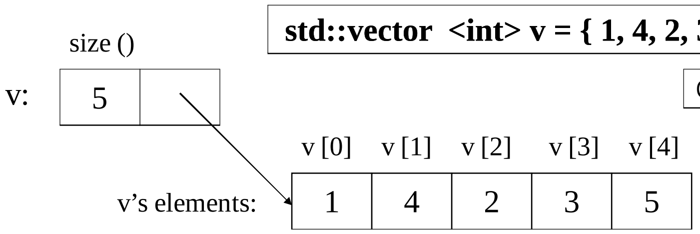

std::vector

- **std::vector** is the most useful standard library data type (Stroustrup)
  - a **std::vector <T>** holds an sequence of values of type **T**
  - we can *think* of a vector the following way (a bit simplified):

    a vector named **v** contains 5 elements: **{1, 4, 2, 3, 5}**

std::vector <int> v = { 1, 4, 2, 3, 5 };

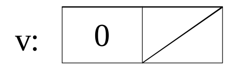
C++11



- indirection needed since a **std::vector** is *flexible*: grows/shrinks

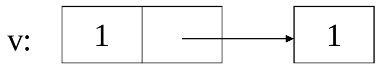
Handling vectors

std::vector <int> v; // *start off empty*

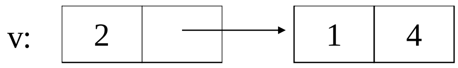


(a hypothetical implementation)

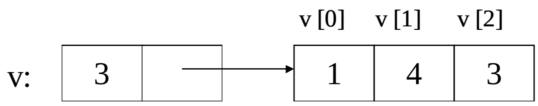
v.push_back (1); // *add an element* ***1***



v.push_back (4); // *add an element* ***4*** *at end* (“*the back*”)



v.push_back (3); // *add an element* ***3*** *at end* (“*the back*”)



Handling vectors (cont.)

```
// compute mean (average) and median:
int main () {
    std::vector <double> temps;          // say, temperature values
    double temp;
    while (std::cin >> temp)             // read and put into vector
          temps.push_back (temp);
    double sum = 0;
    for (int i = 0; i < temps.size (); ++i)  sum += temps [i];
                                         // sums temperatures
    std::cout << "Mean temperature: " << sum/temps.size () << '\n';
    std::sort (temps.begin (), temps.end ());    // standard algorithm
    std::cout << "Median temperature: " << temps [temps.size () / 2]
          << std::endl;                  // adds '\n' and flushes buffer
}
// what if the number stream is empty? - or if IO errors happen?
```

Example – Word list

```
/*  read a bunch of strings into a vector of strings, sort
    them into lexicographical (alphabetical) order ,
    and print the strings from the vector to see what we have
*/
std::vector <std::string> words;
std::string s;
while (std::cin >> s && s != "quit")          //  and not EOF . .
      words.push_back (s);
std::sort (words.begin (), words.end ());     //  standard algorithm
for (auto word : words)                       //  use range for, and
      std::cout << word << "\n";              //       type deduction
```

fail is not possible for a string

- but what about error handling? - discussed shortly

I/O error handling

[simplified from Stroustrup, 2014, Ch. 10.6, p. 354-358]

Read integers from **cin** into a **vector** until we reach **eof()** or **';'**

```
std::vector <int> v; int i = 0;            //  value buffer (initially empty)
while (std::cin >> i) v.push_back (i);     //  read and store until "failure"
if  (std::cin.eof ()) return v;            //  fine: we found the end of file
if  (! std::cin.bad ()) {                  //  not eof and not bad => fail
    // so state is fail  -  probably an integer format error due to ';'
    std::cin.clear ();                     //  clear state, so can read more
    char c = ' ';  std::cin >> c;          //  read a char, hopefully ';'
    if  (c == ';')  return v;              //  got the expected character - OK
    std::cin.unget ();                     //  clean mess: put that char back (?)
    std::cin.clear (std::ios_base::failbit);   //  and set state to fail()
}
error ("input stream is corrupted");  //  get out of here (defined later)
```

I/O error handling: summary

IO streams reduce all errors to one of four states (for stream **is**)

- **is.good ()** // *the last operation succeeded* (*no flags are set*)
- **is.eof ()** // *we hit the end of input* ("*end of file*")
- **is.fail ()** // *something* "*unexpected*" *happened* (*format*)
- **is.bad ()** // *something unexpected and very bad happened*

Sample integer read "failure"

- "1 2 3 4 5 *" => **fail ()** // ended by "terminator character": |*
- "1 2 3 4 5.6" => **fail ()** // ended by format error (for |".6" part)
- "1 2 3 4 5 " => **eof ()** // end of file,
  // Ctrl-Z (Windows), Ctrl-D (Unix)
- *disk error* => **bad ()** // something serious ("cannot recover")

If a stream is in an error state, all subsequent IO operations are ignored.
Reset state: "**cin.clear ();**", or sometimes: "**cin.clear (ios_base::failbit);**"
sets the bit!

Word list, again – eliminating duplicates

```
// eliminate the duplicate words (by copying only unique words):
std::vector <std::string> words;
std::string s;
while (std::cin >> s && s != "quit")  words.push_back  (s);
std::sort (words.begin (), words.end ());
std::vector <std::string> w2;
if  (words.size () >= 1) {                          // not empty
      w2.push_back (words [0]);                     // copy at least first
      for (std::size_t  i = 1;  i < words.size (); ++i)   // from 2nd item
           if (words [i-1] != words [i])            // not the same again
                w2.push_back (words [i]);
 }
std::cout<< "Found " << words.size ()-w2.size ()<< " duplicates\n";
for (auto word : w2)  std::cout << word << "\n";
```

Computation

- Our job is to express computations
  - correctly, simply, efficiently
- One tool is called *divide and conquer*
  - to break down big computations into sub-problems
- Another tool is *abstraction*
  - provide higher-level concepts that hide details
  - both named actions (functions) and user-defined types
- Organization of data is often the key to good code
  - input/output formats, protocols, data structures
- Note the emphasis on structure and organization
  - you don't get good code without analysis, design & some experimentation

Errors: overview

- Errors ("bugs") really are unavoidable in programming
  - sources of errors?
  - kinds of errors?
- To minimize errors
  - organize code and data
  - prepare for testing and debugging
- Do error checking and produce reasonable error messages
  - input data validation
  - function arguments
  - pre-/post-conditions
- Exceptions & a sample **error()** helper routine

```
int main () {
    try {
        // . . .
    } catch (std::out_of_range const&) {
            std::cerr << "vector index "
                                    "out of  range\n";
    } catch ( … ) {    // catches whatsoever
            std::cerr << "unknown error\n";
    }
}
```

On errors

- our most basic aim should be correctness
- we must deal with incomplete problem specifications, external errors and failures, and our own errors
  - prior experience, knowledge of the application domain, the programming language, tools, etc. matter, too
  - note that "incomplete specifications" may result from changing circumstances and requirements
- we'll mostly concentrate on one key area: how to deal with unexpected (invalid) function arguments
- also briefly discuss about techniques for finding errors in programs: debugging and testing

On errors (cont.)

- When we write programs, errors are natural and unavoidable; how to deal with them?
  - from the start, organize software to minimize errors
    - style, idioms, object-oriented design patterns
  - then try to eliminate most of the errors we make anyway
    - by systematically testing and debugging
  - actually cannot guarantee "absolute correctness"
    - eliminate at least the most serious errors
- Stroustrup:
  - ”avoiding, finding, and correcting errors is 95% or more of the effort for serious software development”
  - code complexity often grows exponentially

Detection of "*errors*"

Compile-time errors
- syntax and type errors: missing/extra token, type mismatch

Link-time errors
- missing or incompatible definition of data/functions

Run-time errors
- detected by computer, often by "crashing" the program
- detected by library; often will throw exceptions
- detected by application: lack of resources, connection failures

Logic errors : code compiles but produces incorrect output
- detected by testing (programmer/test driver) - we hope

Terminology: ***fault*** in code => ***error*** in data/state => ***failure*** in execution (state includes the *program counter* PC - i.e., control)

Checking arguments by the compiler

- The compiler helps by statically checking the number and types of arguments (depends on the language)

```
int area (int length, int width)  {  // illustrative, only
   return length  * width;
}
                                    // call arguments must match
int x1 = area (7);                  // error: too few arguments
int x2 = area ("seven", 2);         // error: 1st arg has a wrong type
int x3 = area (7, 10);              // ok
int x5 = area (7.5, 10);            // ok, but odd:  7.5 is truncated to 7;
                                    // many compilers will warn you
int x = area (10, -7);              // this is a difficult case in C/C++:
                                    // correct types but values make no sense
                                    //   -  it is a function domain error
```

Bad function arguments

- So, how about **int x = area (10, -7);** *// or* "**area (10, b);**", *b* < 0
- How to *catch* such an error? Alternatives:
  - all callers check: insecure, laborious, hard to do systematically
  - the function checks (so in one place only), and possibly
    - returns an "error value" – not general, problematic
    - sets an error status indicator – not general, problematic
    - throws an exception: forcing the program check, or terminate
  - a function has no control over how it is called, e.g., consider library routines; so it is wise to be suspicious ..

Note: sometimes we can't ourselves decide about error handling

- someone else wrote the code and we don't want to or even cannot change it

How to report an error: exceptions

Report an error by throwing an exception

```
#include <stdexcept>   // get the standard exceptions
  . . .                //   here, std:: domain_error
int area (int length, int width) {
  if (length < 0 || width < 0)
      throw std::domain_error ("Negative number");
  return length * width;
}
```

Catch and deal with the error (e.g., in **main()**)

```
try { . . .                     // prepare to detect and handle
  int z = area (x, y); . . .    // if area() doesn't throw an exception
}                               //   make the assignment and proceed
catch (std:: domain_error const&) {   // if area() throws
  std::cerr << "Bad area arguments – please, fix program\n";
}
```

Exceptions

- Exception handling is a general solution
  - especially with libraries of reusable components
  - you can't just forget about an exception: the program will terminate if someone doesn't handle it (with a **try** … **catch**)
  - most errors can be reported using exceptions
- You still need to figure out what to do about an exception
  - error handling is never really simple

Note. C++ does not (by default) use exceptions for IO operations

- can argue that they are not really "exceptional" errors since
- we must always check input data - it is part of the problem/task

Out of index range

Try this

```
std::vector <int> v (10);                  // a vector of 10 ints, each
                                           //     to the default value, 0,
                                           // referred to as  v [0] .. v [9]
for (int i = 0; i < v.size (); ++i) v [i] = i;    // set values
for (int i = 0; i <= 10; ++i)                     // print 10 values (oops!)
   std::cout << "v[" << i << "] == " << v [i] << std::endl;
```

- here, **operator []** (subscript) is called with a bad index (10)
- the C++ standard leaves the actual behavior *unspecified*
- the behavior can differ with different environmets
  - if you have some special library with check options and utilities, it *might* report by throwing a **std::out_of_range**

Example: handling an allocation failure

- In C++, **new**/**new**[] replace the *C* allocator macros (**malloc**(), etc.)
- If the runtime system cannot allocate memory for an object on the heap, then a **std::bad_alloc** exception is thrown

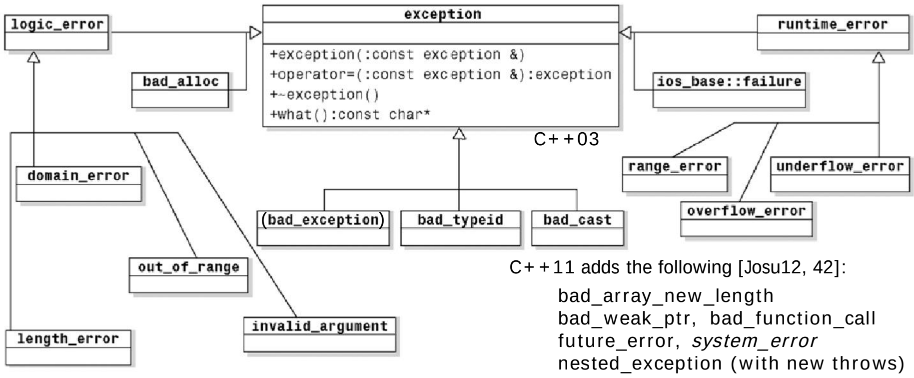
```
Student * michael,  * studentArr;                 // pointers
try {
   michael = new Student ("Mike");                // one student
   studentArr = new Student [1000000000000];  // huge array
      . . .
 } catch (std::bad_alloc const& e) {              // dyn. alloc. failed
      . . .
 }
```

- **Note 1. bad_alloc** can be thrown by any nontrivial program
- **Note 2**. We can also define a special *new-handler* function to deal with the failure of **new**, or we can use the *nothrow*-version of **new** (omitted here, see e.g. Stroustrup or online sources).

Standard exceptions

- Standard library defines a *hierarchy* of exceptions with **std::exception** as the root (in header <**stdexcept**>)
- Exceptions are divided into three main categories
  1. *logic errors*: precondition violations that in principle should be guaranteed before calling an operation; a failure will often mean an error (*bug*) in program logic (e.g., pop from an empty stack, invalid array index, etc.)
  2. *run-time errors*: dynamic errors that cannot really be tested or anticipated, e.g., numeric errors (overflow), communication line failure, or other such external failures..
  3. *language-support*: logic/run-time errors: **bad_cast**, **bad_alloc**,..
- Subclasses of logic errors use more-or-less self-explanatory names; e.g., **std::invalid_argument** exception

Standard exceptions (cont.)



- Logic and runtime exceptions are included from the header <**stdexcept**>. The header <**exception**> provides **std::exception**, and related functions. The header <**typeinfo**> provides **std::bad_typeid** and **std::bad_cast**. The header <**new**> provides **std:bad_alloc**.
- Not the perfect design (says Stroustrup) - but supports portability, uniform handling of exceptions, and is ready for use.

Standard exceptions (cont.)

Exceptions that are often used for library and run-time errors:

- *out_of_range*: invalid index for a STL container (*vector*::*at* ())
- *length_error*: a specified structure/range too long
- *range_error*: error in *numeric computation*

Special exceptions are used for C++ features ("language support")

- *bad_alloc*: operator **new** fails to allocate memory
- *bad_cast*: **dynamic_cast** operation fails (on reference **&**)
- *bad_typeid*: **typeid** operator fails on null ptr/ref (**nullptr**)
- *ios_base*::*failure*: IO failure (when a stream is *configured* to throw exceptions); derived from *system_error* (C++11)

By default, no exceptions are thrown from IO errors.

Standard exceptions (cont.)

- **std::exception** has a special operation **what ()** to report on the error (can be redefined in subclasses)

```
class exception {
public:
    virtual const char * what () const noexcept ();   . . .
};
```

doesn't throw any

```
. . .  std::cerr << e.what ();     //  caught exception e
```

- derived exception classes have constructors to specify the value returned by **what ()**:

```
logic_error::logic_error (std::string const& msg);
```

- there is a similar arrangement for other predefined exceptions
- the constructor takes a **std::string** value as a parameter but the query operation **what ()** returns a C-style character array

Idiom: always handle uncaught exceptions

Use exception handling to "terminate programs gracefully"

```
int main () try  {              //  using special function try block syntax
    // . . .
}
catch (std::out_of_range const&) {   //  (ignore const& ..here)
    std::cerr << "oops – some index out of range\n";
}
catch (std::exception const& e) {      //  some other standard exception
    std::cerr << "oops: " << e.what () << std::endl;
}
catch ( … ) {                            //  all other exceptions
    std::cerr << "oops – some unknown exception\n";
}
```

Pre-conditions

What does a function require of its arguments (data)?

- such a requirement is called a *pre-condition*
- often (depending on circumstances), it's good to check it

```
int area (int length, int width)  {            // calculate area
    if  (length < 0 || width <  0)
        throw std:: domain_error ("Negative number");
    return length  * width;
}
```

- problems are easier to recognize and handle at their beginning

Post-conditions

- What must be true when a function returns?
- Such a requirement is called a *post-condition*

```
int area (int length, int width) {                    // calculate
    if (length < 0 || width < 0)
        throw std::domain_error ("Negative number");
    // the post: return the result from integer multiplication
    return length * width;                             // always OK?
}
```

- Checking can be done by run-time systems, libraries, application code, extra sanity checks (**assert()**), or by test drivers
- Note that here **area()** may produce an unnoticed integer overflow
  - C++ doesn't check integer overflow here (neither does Java)
  - C# provides optional check blocks ("**checked (a*b)**")

Principle of "separate responsibilities"

- It is the *responsibility of the caller* to ensure that *pre-conditions* are not violated
  - the algorithm inside the routine body can then assume that the conditions are valid and just proceed with its calculation
  - failed pre-conditions can show data errors - or a forbidden state
- It is *the responsibility of the called routine* to ensure that the *post-condition* is true
  - the caller can assume that **if** the pre-condition is true **then** the post-condition is met on return from the routine
  - the failure in post-conditions (usually) means that there is a *bug*
- Such contract can simplify both user code and implementation code
- In reality, errors (bugs) do happen and so routines provide a pre-condition checks for general safety/robustness (e.g., *index checks)*

Pre- and post-conditions

- Always think about them as part of program design
  - if nothing else write them as comments
- Check them “where reasonable” (or - at least partially)
  - some failures manifest only after actual attempts
    - accessing scarce resources
    - also consider doing really complicated calculations..
- We will need to check a lot more when looking for a bug . .
- Analyzing and checking pre- and post-conditions can be tricky
  - how could the "post-condition" for **area ()** fail
    - after the pre-condition is established (as true)?
  - what are the *actual* pre- and post-conditions for **area ()**?

Functions and passing-by-reference

Pass-by-reference gives a reference (= an address) to the argument:

```
int f (int& a) { a = a+1;  return a; }

int main () {
    int xx = 0;
    std::cout << f (xx) << std::endl;     // writes 1 ;  changes xx
    std::cout << xx << std::endl;         // writes 1
    int yy = 7;
    std::cout << f (yy) << std::endl;     // writes 8 ;  changes yy
    std::cout << yy << std::endl;         // writes 8
}
```

a is an alias for xx

a:

1st call (refers to xx)

xx: 0

2nd call (refers to yy)

yy: 7

- Similar features: in Pascal, **var** parameter; C#: **ref/out** parameter

Functions and reference parameters

- Reference arguments may lead to obscure bugs when you forget which arguments can be changed

```
int incr1 (int a) { return a+1; }
void incr2 (int& a) { ++a; }
int x = 7;
x = incr1 (x);          // pretty obvious
incr2 (x);              // pretty obscure (in C#: incr2 (ref x); )
```

shows the side-effect

- Occasionally, reference arguments may be essential, *e.g.,*
  - for changing several values via one single call
  - representing so-called *lvalue* (the term is originally from C)
  - manipulating containers (*e.g.,* vector *subscripting* produces lvalues)
  - needed for many technical issues: IO, initialization/copying..
- Note that **const** reference arguments are very often useful

By-value / by-reference / by-const-reference

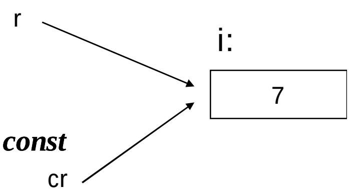
```
void g (int a, int& r, const int& cr) {
      ++a;                    // ok:  a acts like a local variable
      ++r;                    // ok:  r is changed
      int x = cr;             // ok:  cr is accessed for its value
      ++x;                    // ok:  local x is changed
}

int main ()  {
      int x = 0, y = 0, z = 0;
      g (x, y, z);            // after call:  x == 0; y == 1; z == 0
      g (1, 2, 3);            // error: reference r needs a variable to refer to
      g (1, y, 3);            // ok: since cr is const we can pass "a temporary"
}
   // 3rd argument in the last call can be an expression ("x + 1")
```


General references

- "reference" is a general concept, not just for pass-by-reference

```
int i = 7;
int& r = i;            // binds r to i
r = 9;                 // i becomes 9
const int& cr = i;     // binds cr to i
cr = 7;                // error:  cr refers to const
i = 8;                 // ok
std::cout << cr << std::endl;   // writes out the value of i (that's 8)
```



- you can think of a reference as an alternative name for an object
- actually, the *implementation* is just a pointer (memory address)
- but with some restrictions; e.g., we **can't**
  - bind a reference to another object after its initialization
    - traverse a linked data structure (like pointers)
  - modify an object through a **const** reference

Guidance for passing variables

For example

class Image { ***/**** *objects that are potentially huge* ****/*** **};**
void f (Image i); . . . f (myImage); *// copy: can be very slow*
void f (Image& i); . . . f (myImage); *// no copy, but bad style*
void f (Image const&); . . . f (myImage); *// can't mess* **myImage**

- use call-by-value for very small objects (fit registers and such)
- use call-by-**const**-reference for large objects (to avoid copying)
- use call-by-reference only when you have to (sometimes you do)
- generally, better to return a result rather than modify an object through a reference argument
  - more readable, less error-prone, problematic for large values
  - C++11 provides ways to return large objects efficiently

Motivation for namespaces

```
class Glob  { /* . . . */ };      // in Jack's header file jack.h
class Widget  { /* . . . */ };    // also in jack.h
```

```
class Blob  { /* . . . */ };      // in Jill's header file jill.h
class Widget  { /* . . . */ };    // also in jill.h
```

```
#include "jack.h"                 // this is in your code
#include "jill.h"                 // so is this

void myFunc (Widget . . .) {  //  error: multiple definitions of Widget
        // . . .
}
```

Namespaces

- the compiler will not compile multiple definitions for a name
- clashes may occur from including headers (historically: **String**)
- one way to prevent this problem is with namespaces:

```
namespace Jack {                    // in Jack's header file
    class Glob { /* . . .*/ };
    class Widget { /* . . .*/ };
}
```

```
#include "jack.h"                   // in your code
#include "jill.h"                   // so is this
    void myFunc (Jack::Widget . .) { // Jack's Widget class will not
    // . . .                         // clash with a different Widget
}
```


Namespaces

- A namespace is just a *named* scope
  - only a *compile-time* concept
  - no extra memory allocations (memory block)
  - no special initializations required
- The **::** syntax is used to specify (qualify)
  - which namespace you are using, and
  - which (of many possible) objects of the same **name** you are referring to

For example, **cout** is in namespace **std**, so we write:

```
std::cout << "Please enter stuff…" << std::endl;
```

using Declarations and Directives

In order to avoid using the qualifiers

```
std::cout << "Please enter stuff . . ." << std::endl;
```

you can write a *using declaration* (in a .cpp file)

```
using std::cout; . . .                 // cout now means std::cout
cout << "Please enter stuff . . .";    // ok: std::cout
cin >> x;                              // error: cin not in scope
```

or you can write a general *using directive* (but avoid this!)

```
using namespace std;                   // all std names available
cout << "Please enter stuff… ";        // ok: std::cout
cin >> x;                              // ok: std::cin
```

Note. Never place any **using** statements into header files

Enumerations

An **enum** (enumeration) is a very simple user-defined type, specifying its set of values (its "enumerators"); for example:

```
enum class Month {      // a user-defined type, with its constants
   Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

Month m = Month::Feb;                // ok
m = 7;                               // error: can't assign int to Month
int i = static_cast <int> (m);       // ok: convert to a numeric value
m = static_cast <Month> (7);         // ok: convert int to Month
i = 20000;                           // ok: assign to int (of course)
m = static_cast <Month> (i);         // Visual Studio doesn't complain!
```

Unscoped enumeration values (no class!)

- Can also define unscoped enum values (by default, **unsigned int**s)

```
// again, the first enumerator has the value 0, and
// the next enumerator has the value "one plus the value before it"
enum { Horse, Pig, Chicken };     // Horse = 0, Pig = 1, Chicken = 2
```

- Here, too we could explicitly control numbering

```
enum { Jan = 1, Feb, Mar . . . };                  // Feb = 2, Mar = 3
enum StreamState { Fail =2, Bad =4, Eof =8 };  // illustrative, only
```

- and practice some (often unsafe) bit manipulations

```
int ordValue = StreamState::Fail;                 // ok: assign 8
int flags = (int)StreamState::Fail + (int)mState::Eof;  // ok;  = 10
StreamState s = flags;  // error: can't assign an int to a StreamState
StreamState s2 = StreamState (flags);             // unsafe conversion!
```

C++ alternative cast notation

Using enumerations

(1) Simple list of named **unsigned int** constants (instead of macros):

```
enum { Red, Green };              // mere enum doesn't give a scope
int a = Red;                      // ok: Red is available here
enum { Red, Blue, Purple };       // error: Red is defined twice
```

The underlying type is here **unsigned** (but generally *impl. defined*).

(2) A new "scoped" type, with constant list

the underlying type is int unless otherwise specified

```
enum class Color { Red, Green, Blue . . . };
enum class Month { Jan = 1, Feb, Mar, . . . Nov, Dec };

Month m1 = Month::Jan;            // ok
Month m2 = Color::Red;            // error: Red isn't a Month
Month m3 = 7;                     // error: 7 isn't a Month
int i = m1;                       // error: m1 isn't an int
int i = static_cast <int> (m1);   // ok: is converted to an int (1)
```

Summary: operator overloading

- Can overload only *existing* operators (defined by C++ syntax)
  - *e.g.*, **+ - * / % [] = () ^ ! &** < <= > >=
- Can define operators only with their *conventional number* of operands
  - *e.g.*, no unary **<=** (*less than or equal*) and no binary **!** (*not*)
- An overloaded operator must have *at least one user-defined type* as operand
  - **int operator + (int, int);** // *error: can't overload built-in* **+**
  - **Vect2 operator + (Vect2 const&, Vect2 const&);** // *ok*
    . . . Vect2 v1, v2; . . . v1 = v1 + v2;
- Recommendations (for good programming style):
  - overload operators only with their "conventional meaning", e.g., **+** should be addition, ***** be multiplication, **[]** be access, **()** be call, etc.
  - generally, avoid overloading unless very good reasons to do it

Operator overloading

You can overload almost all C++ operators (for class or enum operands - but not **sizeof**, "? :", ..); here, using the scoped enum values:

```
enum class Month : char { Jan = 1, Feb, . . . };     // specify impl.

Month m = Month::Nov;
++m;                        // increment: m becomes Dec
Month Dec = m++;            // increment but use old value; m -> Jan

Month operator ++ (Month& m) {                    // prefix increment
    m = (m==Month::Dec) ? Month::Jan : Month ((char)m+1);
    // wraps around if necessary
    return m;
}
Month operator ++ (Month& m, int) {               // postfix increment
    Month old = m; ++m; return old;               // m wraps around
} . . .
```

dummy parameter (not used) to specify *post*++ (refers to **int**)

Assertions

- Assertions are Boolean expressions that define conditions that should never fail
- C++ provides (in header <**cassert**>) the predefined macro

  assert (*booleanExpression*); // *already in standard C*

- By default, is turned on in the test version
  - if NDEBUG is not defined and the argument of **assert** () evaluates to false, then source file and line are displayed, and the program is immediately aborted, by calling **abort** ()
- **assert** () is usually turned off in the production version
  - when macro NDEBUG is defined, **assert** () does nothing (it's empty) and thus "extra" checks are eliminated from the code

Exceptions vs. assertions

- Differences between exceptions and assertions
  - failed *assert* immediately terminates the program
  - you can catch exceptions and try to continue
  - you can turn off assertions (but usually not exceptions)
- Differences between *preconditions* and other assertions
  - preconditions (often) tests *external* failures which the component cannot handle itself; instead, it must throw failures back to their original source (the caller)
  - many run-time failures (e.g., math. overflow, memory alloc.) can be seen as a kind of "external" factors, too => use exceptions
- Invariants and post-conditions check the *internal* state that cannot possibly make sense to outsiders and indicate a bug in the component => use *asserts* to eliminate them

Summary: Why checks?

Why not leave all checks out of the production version

- we often don't really know the real reason of the failure: perhaps a programming error or some external resource/factor
- strongly-typed languages (such as Java and C#) use checks and exceptions to always prevent unsafe operations
- preconditions/external failures provide a pragmatic trade-off what to check, at the boundary of a component or module
  - note that the C++ standard library uses the same convention (provides pre-condition checks for selected operations)
- to generally use exceptions, must additionally design classes and operations to be *exception safe* (i.e., to tolerate unexpected errors and their handling thru exceptions); we will discuss this later

Debugging is hard

- Try to see what the program code really specifies, not what you hope or think it should say
- Pay special attention to "end cases" (beginnings and ends)
  - did you initialize every variable to a reasonable value?
  - did the function get the right arguments?
  - did the function return the right value?
  - did you handle the first/last element correctly?
  - did you handle the *empty case* correctly?
    - no elements, no input
  - did you open all files correctly?
  - did you actually read that input? write that output?
- Assertions help; IDE helps (breaks, stepping); logging helps
  - need to make the behavior of the program *apparent*/*visible*

How to test a program?

- Think of testing and correctness from the very start
- Practice "*test-driven development*" (TDD)
- Systematically analyze input data and design tests for them
- When possible, test parts of a program in isolation
  - e.g., for a critical or very complicated function
  - write code that calls it with different arguments to see how it behaves in isolation before putting it into the real program
  - *test drivers* help to organize tests and display reports
- Test *both* debug options set ON *and* options set OFF
  - need to try out both debug version and production version
- See more about this in, e.g, [Stroustrup, 2009, Ch. 26 *Testing*]

Program structure: some general rules

- Make the program easy to read so that we have a better chance of spotting the bugs
- Use meaningful descriptive names (most important)
- Comment and explain design ideas (why use this solution)
- Use a consistent layout and indentation
  - an IDE may help (but you are the one responsible)
- Break code into small functions
  - say, try to avoid functions longer than a page
- Avoid complicated/difficult code
  - but, of course, you sometimes cannot avoid such
- Use library facilities (abstractions that hide complexities)
- Use language-dependent idioms, and OO design patterns