Assertion

C++ Object Oriented Programming

Pei-yih Ting

NTOU CS

Contents

- Errors
- Error handling in procedural programming language
- Error messages vs. error codes
- Modifying interface to help the client
- Assertions - make your code prove that it is correct
- Types of assertions
  - Preconditions
  - Postconditions
  - Class invariants
- Conditional compilation and assertions

What is an Error?

- **Compile-time error**: grammatical errors or typos such that the compiler cannot translate your program to machine instructions
- **Run-time error**: the running program does not provide its claimed functionalities
  - Input data is incorrect (either in format or semantics)
  - The data representations/algorithms are incorrect.
  - The computer resources do not satisfy the program requirements. (Not enough memory, disk space, process privilege, i/o capability…)
  - The employed tools (function libraries, external servers, …) do not provide required functionalities.

  Most of the above errors occur when the running program and environment do not meet the **program specification**.
- The interface between client codes and server codes is described in the specification. When either side of codes does not follow the spec, some errors occur.

Errors in Procedural Programming

- **Functions being called** (server codes, utility functions, supporting functions, lower level functions)

```
int server() {
    …
    error occurring position 1; // first type of error
    …
    error occurring position 2; // second type of error
    …
}
```

- **Calling functions** (client codes, controlling functions, upper level functions)

```
…
server(); // first call environment
…
server(); // second call environment
…
```

- Proper error handling depends on the knowledge of both
  - exactly what type of error occurs and
  - in which environment the server function is invoked

Server Handles Errors

```
const int kStackSize = 3;
const int kEmptyStack = -1;
class StackT {
public:
    StackT();
    void Push(int element);
    int Pop();
private:
    int fArray[kStackSize];
    int fTop;
};
```

```
StackT::StackT():fTop(kEmptyStack) {
}
```

```
void StackT::Push(int element) {
    if (fTop+1 == kStackSize)
        cout << "Error! Stack full.\n";
    else
        fArray[++fTop] = element;
}
```

```
int StackT::Pop() {
    if (fTop == kEmptyStack) {
        cout << "Error! Stack empty.\n";
        return kEmptyStack; // meaningless
    }
    else
        return fArray[fTop--];
}
```

Server Handles Errors (cont'd)

```
void main() {
    StackT stack;
    stack.Push(1); stack.Push(2); stack.Push(3); stack.Push(4);
    cout << stack.Pop() << '\n'
         << stack.Pop() << '\n'
         << stack.Pop() << '\n'
         << stack.Pop() << "\n";
}
```

Output:

```
Error! Stack full.
3
2
1
Error! Stack empty.
-1
```

Problems:

1. It does not know the calling environment.
2. It often handles errors uniformly and somewhat blindly.

Client Handles Errors

```
bool StackT::Push(int element) {
    if (fTop+1 == kStackSize)
        return true;
    else {
        fArray[++fTop] = element;
        return false;
    }
}
```

```
void main() {
    StackT stack;
    bool error;
    int value;

    error = stack.Push(1);
    if (error)
        cout << "1 is not pushed in\n";
```

```
int StackT::Pop(bool &error) {
    if (fTop == kEmptyStack) {
        error = true;  // type 1
        return kEmptyStack; // meaningless
    }
    else if (bLocked) {
        error = true;  // type 2
        return kEmptyStack; // meaningless
    }
    else {
        error = false;
        return fArray[fTop--];
    }
}
```

```
error = stack.Push(2);
if (error) cout << “2 is not pushed in\n";
```

Client Handles Errors (cont'd)

```
error = stack.Push(3);
if (error) cout << "3 is not pushed in\n";

error = stack.Push(4);
if (error) cout << "4 is not pushed in\n";

value = stack.Pop(error);
if (!error)
    cout << value << '\n';
else
    cout << "The first pop failed!\n";

value = stack.Pop(error);
if (!error)
    cout << value << '\n';
else
    cout << "The 2nd pop failed!\n";

value = stack.Pop(error);
if (!error)
    cout << value << '\n';
else
    cout << "The 3rd pop failed!\n";

value = stack.Pop(error);
if (!error)
    cout << value << '\n';
else
    cout << "The 4th pop failed!\n";
}
```

Output:

```
4 is not pushed in
3
2
1
The 4th pop failed!
```

Client Handles Errors (cont'd)

- **Problems**:
  1. It does not know where and why exactly the error occurs in the server codes.
  2. It often handles errors uniformly and somewhat blindly.
- Let the **server** handle the error usually can reduce the overall code size. However, it is only possible when the error handling methods for all usages are exactly the same. (perform the factoring operation)
- It's possible that the client code passes some environment identifying information in such that the server can handle errors properly.
- Let the **client** handle the error usually makes the client codes longer. Frequently, only client codes know what to do with a particular error.
- It's possible that the server code passes some exact error types (the **error code**) out such that the client code can handle different errors.

Interface Modification

- The StackT example shows that "**pushing errors**" and "**popping errors**" are **frequent/normal** behaviors by the specification.
  - It is preferred not to call them "**error**".
  - Also, it is preferred that each public method has only **single simple behavior**, for example, Push(item) puts for sure the specified item onto the stack, instead of various combined behaviors, i.e. nothing happens when stack is full, otherwise item is pushed onto the stack.
- Usually, we can improve the design by **modifying the interface** - provide client **extra** interface methods such that **the behaviors of Push(item) can be better controlled/predicted**
- In the following example, we add two more interface methods to the StackT class: **IsFull()**, **IsEmpty()** so that the behaviors of Push() and Pop() are simplified.

Helping the Client

- We can add two functions to the StackT class (the server)

```
bool StackT::IsEmpty() const {
    return fTop == kEmptyStack;
}
bool StackT::IsFull() const {
    return fTop+1 == kStackSize;
}
```

- In the server codes: **NOT** handling errors any more

```
void StackT::Push(int element) {
    if (!IsFull())
        fArray[++fTop] = element;
}
```

```
int StackT::Pop() {
    if (!IsEmpty())
        return fArray[fTop--];
    else
        return kEmptyStack; // meaningless
}
```

Helping the Client (cont'd)

- In the client code

```
void main() {
    StackT stack;

    if (!stack.IsFull())
        stack.Push(1);
    else
        cout << "Deal with push error\n";


    if (!stack.IsEmpty())
        cout << stack.Pop() << '\n';
    else
        cout << "Deal with pop error\n";
}
```

Exceptions vs. assert()

- assert():
  - Catches situations that SHOULD NOT happen (but did happen). For example, promise made by other classes. Basically these are cases **you don't want to handle** (at least NOT specified in the program specification).
  - Typically disabled before product delivery!
  - Should not be seen by the end customer!
  - Used to check / track down programmer's own bugs or negligence
- Exception: try-throw-catch
  - Should be seen by people using the code – the end customers. Not disabled in the final released version.
  - Indicates user errors (e.g. invalid argument errors)
  - Indicates some system errors (e.g. file not found)

assert() / the MS blue screen

- Your program **stops immediately**. Usually used in debugging.
- Why should your program continue if an error has occurred?

1. Non-fatal errors

```
void Stack::push(int element) {
    assert(!isFull());
    m_top++;
    m_array[m_top] = element;
}
```

The failure of the call to push may be non-fatal to the rest of the program.

2. Failing gracefully

```
p = new int[kBigArraySize];
assert(p!=0);
```

Although the memory is insufficient, the user may want to save the existing data before quitting.

3. Safety-critical programming

The patient will die if the software crashes. / System might be hacked.

Error Handling in C++

- Three levels:
  - assert() statements: those errors that the specification of the program excludes. You don't want it to be handled automatically by your program.
  - If statements: those expected situations that happened normally and quite often, e.g. user enter incorrect data, file not opened, …
  - Exceptions: those expected/unexpected situations that happened rarely (say 1 out of 100), e.g. disk access errors, … Or, you want to avoid long/ugly error handling codes…

Rule of thumb: If in doubt, use exceptions

Sometimes, there are still voices of using a single goto statement to handle all sorts of memory deallocation after program fails. In general, this mechanism can be replaced by exception handling.

Assertions

- An assertion is a **statement that must be true** for the function to be correct.
- Three types of assertions:
  - Preconditions
  - Postconditions
  - Class invariants

Preconditions

- An assertion that must be satisfied **before** execution of the function.

```
#include <assert.h>
void StackT::Push(int element) {
    assert(!IsFull());
    fArray[++fTop] = element;
}
int StackT::Pop() {
    assert(!IsEmpty());
    return fArray[fTop--];
}

void main() {
    StackT stack;
    stack.Push(1); stack.Push(2); stack.Push(3); stack.Push(4);
}
```

Assertion (!IsFull()) failed in stack.c on line …

do not follow the protocol

Postconditions

- An assertion that must be satisfied **after** execution of the function.

```
void StackT::Push(int element) {
    int originalTop = fTop;
    assert(!IsFull());
    fArray[++fTop] = element;
    assert(!IsEmpty() && (fTop == originalTop+1));
}
int StackT::Pop() {
    int originalTop = fTop;
    assert(!IsEmpty());
    int value = fArray[fTop--];
    assert(!IsFull() && (fTop == originalTop-1));
    return value;
}
```

Better Example of Postcondition

```
Class DataT {
    friend class StackT;
private:
    int fData;
    DataT(int data);
};
class StackT {
public:
    StackT();
    void Push(int element);
    ….
private:
    DataT *fArray[kStackSize];
    int fTop;
};
```

```
void StackT::Push(int element) {
    assert(!IsFull());
    DataT *temp = new DataT(element);
    fArray[++fTop] = temp;
    assert(temp!=NULL);
}
```

temp might actually be NULL if new operator fails to allocate required memory.

Class Invariants

- A class invariant is a condition that **holds true for the entire class**.
- A class invariant must satisfy two conditions:
  1. true at the end of every constructor
  2. true at entrance and exit from every public mutator function

Note: from the above

  a. A class invariant holds only for its client (might not hold at any particular instant, especially inside any member function)

  b. It is assumed that these objects work in a single-threaded environment.

- When is an invariant exempt from being true?

  inside a private member function

- bool StackT::ClassInvariant() {

  return (fTop>=kEmptyStack) && (fTop<kStackSize);

  }

Class Invariants (cont'd)

- First condition:

```
StackT::StackT() : fTop(kEmptyStack) {
assert(ClassInvariant());
}
```

- Second condition:

```
void StackT::Push(int element) {
    assert(ClassInvariant());
    assert(!IsFull());
    fArray[++fTop] = element;
    assert(!IsEmpty());
    assert(ClassInvariant());
}
```

```
void StackT::Pop() {
    int value;
    assert(ClassInvariant());
    assert(!IsEmpty());
    value = fArray[fTop--];
    assert(!IsFull());
    assert(ClassInvariant());
    return value;
}
```

Managing Assertions

- Problems of using assertions
  1. Many checkings require time, program might be sloppy
  2. The abort message should never be seen by the user. (e.g. the annoying MS window's blue error screen)
  3. These checkings should not be left effective in a released S/W.

- Use conditional compilation

```
#define _NDEBUG

StackT::StackT() : fTop(kEmptyStack) {
  #ifndef _NDEBUG
  assert(ClassInvariant());
  #endif
}
```

```
void StackT::Push(int element) {
  #ifndef _NDEBUG
  assert(ClassInvariant());
  assert(!IsFull());
  #endif
  fArray[++fTop] = element;
  #ifndef _NDEBUG
  assert(!IsEmpty());
  assert(ClassInvariant());
  #endif
}
```

errno in UNIX Environment

#include <string.h>

char * **strerror**(int errnum);

The strerror() function accepts an error number argument errnum and returns a pointer to the corresponding message string.

e.g. strerror(errno);

#include <stdio.h>

void **perror**(const char *string);

The perror() function finds the error message corresponding to the current value of the global variable errno (intro(2)) and writes it, followed by a newline, to the standard error file descriptor.

e.g. perror("module");

// module: error message corresponding to errno

GetLastError() in MS Windows

```
LPVOID lpMsgBuf;
FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
               FORMAT_MESSAGE_FROM_SYSTEM |
               FORMAT_MESSAGE_IGNORE_INSERTS,
               NULL,
               GetLastError(),
               MAKELANGID(LANG_NEUTRAL,
                          SUBLANG_DEFAULT), // Default language
              (LPTSTR) &lpMsgBuf,
              0,
              NULL ); // Process any inserts in lpMsgBuf.
// ...
MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK);
```

http://msdn2.microsoft.com/en-us/library/ms681385.aspx