

VE280 Recitation Class Notes

VE280 SU19 TA Group

UM-SJTU Joint Institute

June 20, 2019

Source code available at <https://github.com/ve280/VE280-Notes>
Slides modified from the version of YAO Yue's, with permission

Table Of Contents I

1 RC Week 6

- Mechanism behind function calling
- Recurse Recursively
- Engineering correctness: Testing
- Engineering robustness: Exceptions

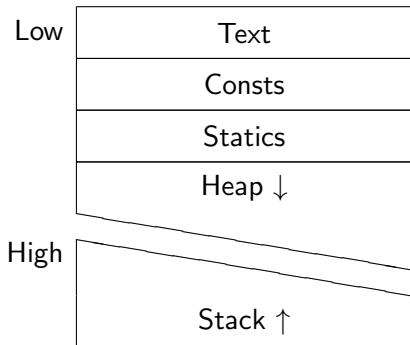
RC Week 6

Mechanism behind function calling

Memory Layout

Before function calling we would like to first give a brief introduction to the memory organization of the system

Explanation



* This graph is up-side-down.

- “Text”, just “code”
- “consts”, not like `const int const`, but string literals, “Hello world”.
- “Statics”, global variables, static variables in functions. “static” refers to lifetime.
- “Heap”, where you `new`.
- “Stack”, everything local. Arguments, return values, return addresses ...

Element in a stack

foo	<code>z = 4 @ foo(3)</code>
	<code>x = 3 @ foo(3)</code>
	<code>Ret = ?</code>
	<code>RA = &CALLS + 1</code>
	<code>q = ? @ main()</code>
main	<code>p = 3 @ main()</code>
	<code>...</code>

```

int foo(int x) {
    int z = x + 1;
    /* Here */
    return z + x;
}

int main() {
    int p = 3;
    p = foo(p); // <- CALLS
    int q = 10;
}

```

The stack maintains the following information

- Function arguments. They are evaluated and on to the stack.
- Local variables (arrays). They are reserved before initialized.
- Return value and return address. Return address tells which instruction to pick up when the call returned.

Remarks

Calling mechanism is about *Abstraction*

Calling mechanism is designed in such way to support procedural abstraction. In order for the abstraction to work, we require

Each function call is independent

This is especially important if you have recursion calls.

Calling mechanism is platform dependent

- Calling mechanism is neither specified by standard, nor unique!
- Whose responsibility to manage arguments? Caller / callee?
- Compiler might optimize unused variables out.
- Compiler might use register to store information.
- Compiler is allowed optimize the entire stack frame out.

Puzzles for FFFUUUNNNN!

Understanding calling stack is most useful in finding out what has gone wrong when you observe strange behavior of your program. This is very much like solving puzzles.

We now give you a few such puzzles to entertain you. Note all the code we gave you below contains **undefined behaviors** so don't be surprised if you cannot reproduce this problem.

This is actually worth noting. Many undefined behaviors would cause different behavior since given different situation on the stack.

- This code works on my computer but crashed on OJ.
- This code crashes / malfunctions if I change unrelated things.
- Student: "TA, my code can't work!".
TA : "Can you demo? I can't reproduce your problem"
Student: "Suddenly I can't Either! But it crashes on OJ!"
- This code randomly crashes.

Puzzle 0: Why not VLA?

Variable Length Arrays (VLA) are arrays whose size are determined at compile time. For example in the following code `arrX` is a VLA, and `arrY` is a usual array.

```
void foo() {int t = 20; int arrX[t * t]; int arrY[400];}
```

But both C++ and C choose **not** to support this language feature (above code won't compile). Explain what's the underlying reason.

Puzzle 0: Why not VLA?

Variable Length Arrays (VLA) are arrays whose size are determined at compile time. For example in the following code `arrX` is a VLA, and `arrY` is a usual array.

```
void foo() {int t = 20; int arrX[t * t]; int arrY[400];}
```

But both C++ and C choose **not** to support this language feature (above code won't compile). Explain what's the underlying reason.

Explanation

What would be the impact of this feature? Variable length array will consume variable amount of memory.

If the array is a local variable. The stack frame size of the function cannot be determined in compile time.

The need to know the size of a function's compile time stack frame size is centric in language design.

,

Puzzle 1: Orders matter

```
-- > code/rc5pz1/a.cpp
#include <iostream>
using namespace std;
struct S{int x = 4; char a[4];};
void foo() {
    S s; cin >> s.a;
    while(--s.x) cout << s.a << endl;
}
int main() {foo();}
```

User inputed Hello, symptoms are:

- Program crashes after printing 3 times of Hello strangely.
- Program runs fine if change input to Bad.
- Program doesn't crash if switch char a[4] and int x = 4.
But the program keeps printing Hellx (x decreasing char in ascii).

Solution 1

4B	s.x = 4 @ foo()
1B	s.a[0] @ foo()
1B	s.a[1] @ foo()
1B	s.a[2] @ foo()
1B	s.a[3] @ foo()
foo	RA = !!
main	...

Return address is corrupted.

1B	s.a[0] @ foo()
1B	s.a[1] @ foo()
1B	s.a[2] @ foo()
1B	s.a[3] @ foo()
4B	s.x = !! @ foo()
foo	RA = &main + 1
main	...

Variable s.x is corrupted.

C99 6.7.2.1 clause 13 states: Within a structure object, the non-bit-eld members and the units in which bit-elds reside have addresses that increase in the order in which they are declared.

Puzzle 2: Missing return value?

```
-- > code/rc5pz2/a.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int foo(int x) {
```

```
    if (x < 100) { x = x * x; foo(x);}
```

```
    else return x;
```

```
}
```

```
int main() { cout << foo(15);}
```

This code is supposed to keep squaring a number until it's greater than 100, but the real symptoms are

- Program output random number if c/with g++ a.cpp
- Program always return 225 if c/with g++ -O1 a.cpp
- Program spits random number if c/with g++ -O1 a.cpp, if we change "foo(x);" to "y = foo(x);".

Solution 2

	x = 225 @ foo(225)
	Ret = 225
foo	RA = &foo(15)
	x = 225 @ foo(15)
	Ret = ?
foo	RA = &main
main	...

Without optimization

	x = 225 @ foo(225)
	Ret = 225
foo	RA = &main
main	...

After optimization

Puzzle 2.5: Who moved my cheese?

```
-- > code/rc5pz25/a.cpp
#include <iostream>
using namespace std;
int setFirst(int x[][5], int size) {
    int cheese = 0;
    while (size >= 0) x[--size][0] = size;
    cout << cheese << endl;
}
int main() {
    int arr[10][5] = {0};
    setFirst(arr, 10);
}
```

We observe the output to be -1. However we haven't changed the variable `cheese`. Who mov-ed my cheese.

Solution 2.5

setFirst	cheese = !! @ setFirst()
	...
	RA = &main
	arr[0][0] @ main()
	arr[0][1] @ main()
	... @ main()
	arr[1][0] @ main()
	... @ main()
main	Ret = ?
	RA = ...
	...

- The loop out-of-bound.
- How 2D arrays are stored.
- Understand how indexing works.
- We omit the arguments and return values of foo on the graph

Puzzle 3: A hijack

For this puzzle to work you might need to turn off *Address Space Layout Randomization* and set `-fno-stack-protector` in g++.

```
-- > code/rc5pz3/a.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
void secretFunction() {
```

```
    cout << "You shouldn't be here..." << endl;
```

```
}
```

```
void echo() { char buffer[20]; cin >> buffer; }
```

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

The trick is a carefully constructed input:

```
voidsecretFunction2e34!2&&Q.%.x;" ]
```

We observe the output is `You shouldn't be here...`

The question is how does this happen, since `secretFunction` is not called at all?

Stack unwinding / unrolling

Hope you still remember *constructors* and *destructors*!

When the function returns, it not only simply discards the stack space, it actually *DESTROYs* the local objects, essentially calling their destructors.

This is a very useful feature! We can use this feature to automatically release resources. Consider the following File class.

-- > code/rc5unroll/file.h

```
class File {
    string name;
public:
    File(string file) : name(file) {
        cout << "Opened : " << name << endl; }
    ~File() {
        cout << "Closed " << name << endl; }
};
```

Stack unwinding / unrolling

The following code executes utilizes the above code: -- >
code/rc5unroll/a.cpp

```
#include <iostream>
using namespace std;
#include "file.h"
int foo() {
    File f1("file1.in"); File f2("file2.in");
    cout << "I Returned!" << endl;
}
int main() { foo(); }
```

We paste the output:

```
Opened : file1.in // Opended : file2.in
I Returned! // Closed file2.in // Closed file1.in
```

The files clean themselves up after returned.

Stack overflow

A final point is that in general your stack is rather small, compared to the heap.

We refer to an empirical experiment done by *Bruno Haible* in 2009.

- glibc i386, x86_64	7.4 MB
- Cygwin	1.8 MB
- Solaris 7..10	1 MB
- MacOS X 10.5	460 KB
- OpenBSD 4.0	64 KB

Usually heap size is hundreds of MB, if not GB on a modern computer. It could happen that you ran out of stack space. In such situation we say you encountered a *stack overflow*, because:

- Maybe you recurse too deep (Why?).
- Maybe you declare large arrays in the stack.

RC Week 6

Recurse Recursively

Recursion is the art of abstraction

A typical processes of designing a recursive function goes as follows:

- Be very clear first about the abstraction of the function you needed to input.
- Specify a base case, a set of input where the answer is immediately known (or can be calculated in a few steps).
- Assume that your abstraction actually works for input “simpler” than the current input (closer to the base case). Use this assumption to build your program.

You might find these steps surprisingly similar to a mathematical induction. That's true. They are similar and that's very useful.

Recursion is the art of abstraction

```
// REQUIRES: list is not empty  
// EFFECTS: returns largest element in the list  
int largest(list_t list) {  
    int first = list_first(list);  
    list_t rest = list_rest(list);  
    if (list_isEmpty(rest)) return first;  
    return max(first, largest(rest));  
}
```

- The abstraction is specified in the header.
- The base case is where list contains just one element.
- In the last line, assume abstraction works in simpler case (hope you see why rest is “simpler” than list). Thus largest(rest) returns the largest of the remaining list.
- The largest number can either be the largest of the remaining number, or the first number.

Example: Quick sort algorithm

A quick sort algorithm sorts a array in a quick way (I know that sound like bullshxx!). But list basic steps as following:

- Select an element in the array. We simply use the first element. We call this element *pivot*
- We need to *partition* the array. We need to move the elements less than the pivot to the left and elements larger than pivot to the right. Note we assume on the order of the elements left of the pivot (or elements on the right of the pivot).

6	5	3	8	-1	7	3	9	11	-4		5	3	-1	3	-4	6	8	7	9	11
<- pivot											<- pivot									

- Call QuickSort on the both sides of the pivot.

The majority of the work lies in the partition step.

A usual Quick Sort

```
-- > code/rc5qsort/nonrec.cpp
```

```
void quickSort(int *data, int left, int right) {  
    int len = right - left; if (len <= 1) return;  
    int pivotIndex = left; int pivot = data[pivotIndex];  
    int *pData = new int[len];  
    int top = 0; int bottom = len - 1;  
    for (int i = left; i < right; ++i) {  
        if (i == pivotIndex) continue;  
        if (data[i] <= pivot) pData[top++] = data[i];  
        if (data[i] > pivot) pData[bottom--] = data[i];  
    } pData[top] = pivot;  
    for (int j=0; j<len; ++j) data[left+j] = pData[j];  
    quickSortHelperExtra(data, left, left + top);  
    quickSortHelperExtra(data, left + top + 1, right);  
}
```


Our Quick Sort

Suppose we are using our list interface (in the project!).

- Suppose `QuickSort` is a function that takes in a list and returns a sorted list. Remember this is abstraction. Base case is when the list is empty.

```
list_t qSort(list_t lst); // Returns sorted lst
```

- Our computation goes as follows, we first acquire a the partition-left part and partition-right part. We call `qSort` on both parts and concatenate left, pivot and right part:

```
list_t sorted = cat(qSort(left), pivot, qSort(right))
```

- The left part are simply numbers less than pivot, the right part are simply numbers greater than pivot.

```
list_t left = filterLess(lst, pivot);
```

```
list_t right = filterGreater(lst, pivot);
```

- And we are done. Now we simply copy everything into one place.

Our Quick Sort

And here is the famous (almost) one-line quick sort

```
list_t qSort(list_t lst) {  
    if (isEmpty(lst)) return lst;  
    int pivot = list_first(lst);  
    return concatenate(  
        qSort(filterLess(list, pivot)),  
        pivot,  
        qSort(filterGreater(list, pivot))  
    );  
}
```

Now it just leaves us to implement the filter function and the concatenate function. But these two functions should be very easy. You have implemented the filter function in your project right?

Why not references and pointers?

Think about it, why this seems much easier (clearer, hopefully you do feel that way)?

In the traditional code, all functions works on the same array. We must manually control the process of copying, moving, etc. We are thinking in terms of *operations*, detailed step to be performed.

But the the new code, we can now begin think of data. We stop focusing on the concrete steps, but simply

What should I do with the data?

What is the expected input and the expected output?

It is the computation we needed to focus.

This is not easy. The immutability of the data and the fact that all functions are pure allows us to do such thing. Every function does calculation on its own and does not impact the outside world.

Bridging the old perspective

Consider the following problem:

Write a function `isMoreOdd` that takes a list $(a_0, a_1, a_2, \dots, a_n)$ and returns $\sum_{i=0}^n i^2 a_i$ (we call this an s -sum) Assuming the list is non empty.

An example as follows:

a_i:	6	5	3	8	-1	7	3
i :	0	1	2	3	4	5	6

We follow our usual steps. The abstraction is self-explanatory. The base case is also easy (a single element list). But the problem lies in the third step.

It seems that knowing s -Sum for the rest of the list doesn't help on the reducing the problem to a simpler point.

It would still be most desirable to have some sort of “accumulator”, something that registers a partial sum, a piece of

Accumulator passing style

This is still possible. Such construct is so common that gets its own name. We call this *Accumulator passing style* (APS).

```
int helper(list_t remain, int index, int acc) {  
    if (isEmpty(remain)) return acc;  
    acc += index * index * list_first(remain);  
    return helper(list_rest(remain), index + 1, acc);  
}  
  
list_t strangeSum(list_t list) {  
    return helper(list, 0, 0);  
}
```

How does this work?

The essential idea is to sum up the necessary information into two (could be more) accumulators. We essentially created a sort of running sum.

Accumulator passing style

We then further note the abstraction of the helper function.

The function `helper` takes the index of the first element in the remainder list and a partial sum of the elements before, and returns the s -sum of the entire list.

In this way we transform our original problem of finding out `helper(list, 0, 0)`. On each recurse call, we extract the first element, and use it to update our accumulators, and passed that on to the next call.

```
helper([6 5 3 8 -1 7 3], 0, 0)
```

```
helper([5 3 8 -1 7 3], 1, 0)
```

```
helper([3 8 -1 7 3], 2, 5)
```

```
helper([8 -1 7 3], 3, 17)
```

```
helper([-1 7 3], 4, 89)
```

```
helper([7 3], 5, 73)
```

```
helper([3], 6, 248) = helper([], 7, 356) := 356
```

Remarks

Understand APS in a broad sense

APS is extremely useful. In many sense this technique is very similar to loops, which you might feel more comfortable to deal with. On the other hand an accumulator can be more than just an sum of numbers. It could be any information you need to keep track of (for example, if the number before forms an arithmetic sequence, and if they do, what is the increment).

Recursion and correctness

It's extremely difficult to write correct code! It would be nice if we can formally prove that our code is correct. Since the usual procedural code involves state, this proof can be very complex. But if you express the idea using abstraction, proving correctness is very simple and forward. A good recursion construction is almost always correct, and you can prove it! Write once and be free of testing and bug. What a nice thing!

RC Week 6

Engineering correctness: Testing

The definitive correctness myth

We quote the following words from one of yours (@LukeXuan)

While testing did indeed help your code to behave correctly in most cases. It never gives full assurance. I think you should recommend the technology of formal methods, especially verification, to introduce the possibility of complete correctness of program to students.

Despite the obvious taunt in the words, these comment DOES speaks some, truth, that is:

It is fundamentally impossible, proved in theory, to guarantee the absolute *correctness* of the program by simply testing it.

After all testing is an attempt to engineer correctness. It is an engineer method that aims at decreasing chances of software malfunction in the field, i.e. reliability.

Two general strategies in testing

Black box testing

Treat your program under testing as a “blackbox”. The tester cares only about input and output. Essentially our OJ does black-box testing.

Glass box testing

Tester designs the test case according to the case. Test-cases are designed in such way that attempts to

- Achieve full coverage (Activate every branch once).
- Touch boundaries, base cases, or data-type boundaries.
- Stress the implementation, or exploit it for security reasons.

Testing is always an active activity, even for black box testing. Test cases are always designed with the technicals in heart.

Input Partitioning

The purpose of the testing, in most common cases, is to reveal possible defects, by trying to pick representative inputs.

The basic logic in designing test-cases is

If this program works on X , so it should work on Y .

The job of the tester is often to categorize all possible inputs into different *equivalent classes* (if you still remember that term from VE203). For each equivalent class we pick a few inputs and assume if the function works for those input, it should work for all inputs within that class.

Remember, again, **testing is an creative process** that relies on your understanding of both the problem and implementation at hand.

It is never an easy job to partition the input right.

Program under testing

The following program takes a string (of less than 100 characters) and decides whether it is palindromic recursively:

```
bool isPalindrome(const char* str, int size) {  
    if (size == 0) return true;  
    if (size == 1) return true;  
    if (str[0] != str[size - 1]) return false;  
    return isPalindrome(++str, size - 2);  
}  
  
int main() {  
    char str[100]; cin >> str;  
    int size = strlen(str);  
    cout << isPalindrome(str, size);  
}
```

Test Cases Designing: “Normal Input”

The most common kind of test cases are “Normal Inputs”. Normal inputs are considered normal in the following sense:

- They are normal in range.
- They are normally constructed, i.e. are not delicately constructed to sabotage / overload the program.
- They cover most normal outputs.

For our previous example some good test cases will be:

- "12321" Odd size palindrome
- "1221" Even size palindrome
- "1222" Even size non-palindrome
- "1234345" Odd size non-palindrome

Test Cases Designing: “Boundary Input”

Boundary cases are those input that pushes the program to the “boundary”:

- They are base case in recursion.
- They pushes program to the edge of used datatype range.
- Any point that is “tricky”. For example in a gcd program you should remember to test the input where one argument is a multiple of another. Any input that will trigger a special treatment.

For our previous example some good test cases will be:

- "" Empty string
- "1" Single character string
- 123...321 99 characters string, why 99?
- "11" Odd number (possibly) base case

Test Cases Designing: “Random / Malicious input”

Those are inputs that does not make sense. You normally don't expect your users to supply such arguments, but often technically they can.

For example you won't expect the user to input `I love lemonade` in a text box labeled *What is your social security number?*. But technically your program can receive such input.

It is often these situation that brings about the most trouble.

These input can potentially crash the program. Or worse, construct special input that corrupts / extracts confidential data.

The conclusion is: **Never trust your user, not a single byte!**

- Random input, random bytes...
- Malicious input.
- Assume that your user will not listen to your warnings. E.g. input more than 10 characters when prompted Input a string of less than 10 characters :.

Test Cases Designing: Design with abstraction

It is very important to keep the abstraction, or specification in general in mind when trying to design test cases.

- The specification specifies the what inputs are “normal”, i.e. what are the inputs that fits “REQUIRES” clause.
- The specification specifies the expected output.
- The specification often defines boundaries, the one value that divides valid input with invalid inputs.
- The specification often suggests program load in real world.
- The specification tells what inputs are considered “invalid”.

There is one more thing, the “MODIFIES” clause. Often the code under testing produce (or relies on) side-effects. The “MODIFIES” clause specify these things.

Again we emphasize the importance of creating a clear abstraction!

The following is adapted from the work of *Bill Sempf's* twitter. A

QA engineer walks into a bar

- 40/61

Unit test, Regression test and Integration Test

Rome is not built over night. So are softwares.

We now introduce you to some software engineering terms.

- Often software are first designed by an architect, partitioned into *modules*.
- Programmers code each module independently. They write Unit Tests for each module.
- When the modules are put together, architect team creates *Integration tests*
- The collection of test cases are called a *Test Suite*
- The team will keep updating the software. After each change, a test suite will be run to ensure the change didn't happen to break anything. This is called a *Regression Test*

Test automation

From you own experience:

- Testing is actually pretty common task.
- It takes time to create driver programs to run the tests.
- It takes time (and code) to create test cases. You often need to manually calculate the expected output.
- It takes time (and code) to analyze test results. To keep track what goes wrong, especially you.

This calls for *Test Automation* and *Testing Frameworks*, a testing framework is (often) a piece of library that does the following:

- Runs test cases automatically.
- Manages the test cases, selects what to run and what not.
- Automatically sets-up the test environment (test fixtures).
- Automatically keeps track of what's OK and what goes wrong.
- And much more...

Google Test framework

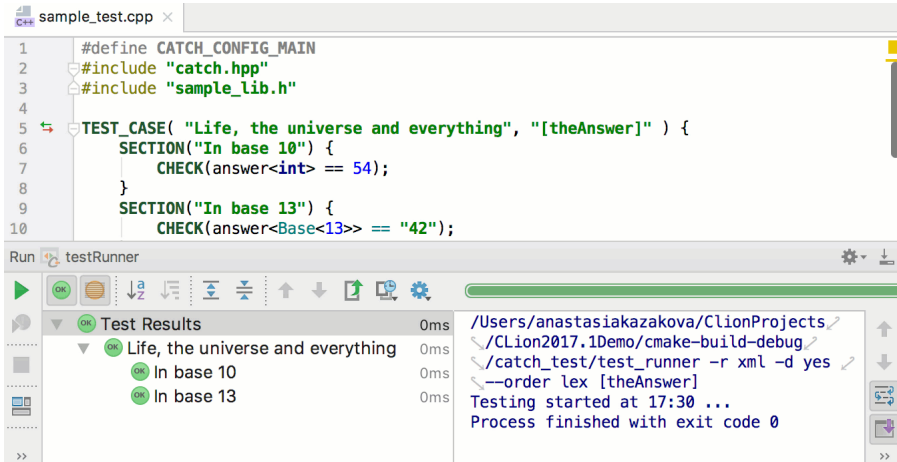
Project Homepage: <https://github.com/google/googletest>

Introduction is available if you click [here](#)

Here is a portion of code I used to test my VE281 project:

```
TEST_P(SelectionTest, DSelection) {  
    int *d= dataset->getCopied();  
    int size = dataset->getSize();  
    for (int i = 0; i < size; ++i) {  
        int val = deterministicSelection(d, size, i);  
        int answer = dataset->select(i);  
        ASSERT_EQ(val, answer);  
        delete[] d;  
        d = dataset->getCopied();  
    }  
    delete[] d;  
}
```

Google Test in CLion



The screenshot shows the CLion IDE with a C++ file named `sample_test.cpp` open. The code defines a test case for "Life, the universe and everything" with two sections: "In base 10" and "In base 13". The test runner has been executed, and the results are displayed in the bottom panel. The test results show that both sections passed successfully, with a total execution time of 0ms. The terminal output shows the command used to run the tests and the resulting output.

```
1  #define CATCH_CONFIG_MAIN
2  #include "catch.hpp"
3  #include "sample_lib.h"
4
5  TEST_CASE( "Life, the universe and everything", "[theAnswer]" ) {
6      SECTION("In base 10") {
7          CHECK(answer<int> == 54);
8      }
9      SECTION("In base 13") {
10         CHECK(answer<Base<13>> == "42");
11     }
```

Run testRunner

Test Results

- OK Life, the universe and everything 0ms
 - OK In base 10 0ms
 - OK In base 13 0ms

Terminal Output:

```
/Users/anastasiakazakova/CLionProjects
/CLion2017.1Demo/cmake-build-debug
/catch_test/test_runner -r xml -d yes
--order lex [theAnswer]
Testing started at 17:30 ...
Process finished with exit code 0
```

RC Week 6

Engineering robustness: Exceptions

Breaking the abstraction

The central question that goes around with exceptions are:

What if assumptions of an abstraction is broken?

Note that this should be understand in a broader sense. Any program runs under some assumption. For example,

- There is enough system memory for your program.
- There is enough space when you need to create a file.
- Input outside `REQUIRES` clause never happens.
- Your computer have an available Internet connections.
- ...

All these things constitutes the assumption you made about your abstractions. But it certainly could happen that one (or more) of them are broken. This could due to hardware limitations, or more likely due to an error in programming.

Fail-fast & “I give up.”

There is absolutely no point to save a flawed process. This is called the fail-fast fast.

- The program is already in a non-recoverable state
- It is probably due to a programming error
- Error might propagate, crash site far from source.
- May corrupt data. Programs can be fixed, data can't.
- Best strategy is to quit gracefully.

A typical situation:

```
void foo() {  
    int *p = malloc(sizeof(int) * 10);  
    // This should always success unless lacking memory  
    assert(!p);  
    // Do something with p  
    free(p);  
}
```


“It’s my problem.”

The attempt is to make the function essentially a “total” function. Essentially this strategy says

“Invalid inputs are part of my abstraction”

Which also implies testing for invalid inputs! An (not so good) example:

```
// Checks if all letters in a string are capital
bool isAllCapital(char* str, int size) {
    int len = strlen(str);
    if (size != len + 1) len = size; // Input validation
    for (int i = 0; i <= len; i++)
        if (*str < 'A' || *str > 'Z')
            return false;
    return true;
}
```

“It’s my problem.”

There are some serious problem with this approach:

- Function might not have well defined “default” value.
- It might hard to guess a “default” value.

Regardless above problem, much more serious problem comes with error propagation. There is probably a reason why this input is valid. It’s most likely because you code is incorrect (in some sense) or your running environment is problematic (stack overflowed, for example).

You need to understand whenever you took this approach, you are trying to fix an already “broken” program, going against the fail-fast principal.

- You could end up crashing somewhere far from the root cause.
- Your program could behave wired. Since your abstraction includes treatment of “special cases”

“It’s not my problem”

The problem of above approaches is significant because they are trying to deal with errors that does not come from themselves.

The root cause of these invalid inputs are somewhere else, probably only known by their caller.

The natural idea would be to find a method to pass the information up the chain of calling, a natural way of doing so is by return *Error Codes*.

There are in general two ways to do so:

```
// Returns negative value if error
```

```
int fact1(int n);
```

```
// Returns value signifies error, zero indicates no error
```

```
// Actual result is put into *rst
```

```
int fact2(int n, int* rst);
```

In some cases libraries use a global variable to store the error code of last function call.

Problem with error codes

Both methods have severe draw back:

```
double tan(double rad);
```

Function `tan` could return anything in `double`, now what should you use for error code?

```
int foo(double rad) {  
    double rst = 0.0; int err = tan(rad, &rst);  
}
```

The second makes calling “unnatural”. Also performance drawbacks.

- Error code can be ignored. The worse thing than crashing on errors is an unattended error
- Error code needs to be passed up the stream. Sometimes the direct caller also don't have a clue, it needs to pass it on.
- Breaks abstraction.

Structural Error handling: try, catch and throw

We now introduce you to *Structural Error Handling*. The following is a try block

```
// (1)
try {
    // (2) (may throw;)
} catch (const T& e) {
    // (3) ...
}
// (4)
```

Just like usually if, while blocks, a try block marks a special control flow, featuring (usually) 2 different code execution paths.

- Path 1. Exception raised in (2), we go (1) (2, partial) (3) (4).
- Path 2. Normally we go (1) (2) (4).

Path 1 assumes the exception is caught. Further discussion on next slide.

Structural Error handling: try, catch and throw

An concrete example.

```
try {  
    cout << "hello!" << endl; throw 123;  
    cout << "Goodbye" << endl;  
} catch (int x) {  
    cout << "Integer Exception";  
}
```

Prints "Hello//Interger Exception"

- A throw clause raise an exception (of arbitrary type).
- When an exception is raise, immediate stop the following execution and go for smallest enclosing try block.
- If there isn't one, terminates program.
- If there is one, begin matching catch clauses.
- If found, we say the error is *handled*, begin executing after try. If not, terminates program.

Structural Error handling: try, catch and throw

A try block can be matched no matter how deep in the function.
The following example is just for demonstration! **It is NOT a good practice!**

```
int foo(int n, int prod) {  
    if (n == 0) throw prod;  
    foo (n - 1, n * prod);  
}  
  
int realFact(int n) {  
    try {  
        foo(n , 1); cout << "LaLaLa";  
    } catch (const int& x) {  
        cout << "Aloha!" << endl;  
        return x;  
    }  
} // Prints Only "Aloha!".
```

Structural Error handling: try, catch and throw

An exception cannot be overlooked! Unhandled exception terminates the program.

```
int fact(int n) {  
    if (n < 0) throw n;  
    if (n == 0) return 1;  
    return n * fact(n - 1);  
}  
  
int main() { int x = fact(-50); }
```

Note we use the word “Terminate”. There is a difference in *crashing* and *terminating*.

The first term indicates the program is stopped by force (by the operating system), immediately.

The second one indicates a series of events will happen (for example, clearing the buffer of cout). The program terminates (somewhat) gracefully.

Structural Error handling: try, catch and throw

The smallest enclosing try block gets to handle the exception

```
void foo() {throw -1;}
void bar() {
    try{ foo(); }catch(double e){cout << "bar!";}
    cout << "Slotted Aloha!";
}
void rua() {
    try{ bar(); }catch(int e){cout << "rua!";}
    cout << "Alright!";
}
void baz() {
    try{ rua(); }catch(int e){cout << "baz!";}}
int main() {baz();}
```

Above programs prints rua!Alright!. It terminates normally.

Structural Error handling: try, catch and throw

An exception can be rethrown after being caught in a catch block. This is useful since you might need clean up, although you don't know how to deal with the error. It is also possible to throw a different exception.

```
void foo() {throw -1;}
void bar() {
    int *p = new int(10);
    try{ foo(); }
    catch(...){delete p; cout << "bar!"; throw; }}
void rua() {
    try{ bar(); }catch(int e){cout << "rua!"; throw 1.0;}}
int main() {rua();}
```

Above programs prints bar!rua!. It terminates due to an unhandled exception 1.0.

Rules for matching for catch

The rules for matching catch is given as follows:

- 1 First found first match.
- 2 Match only with **exact** same type. If the exception is an class instance, also matches with it's base catch.
- 3 `catch(...)` matches everything.

```
try{int x = 1; throw x;}  
catch(long int li) {  
    // No match  
}  
catch (int x) {  
    // Match here  
}  
catch (...) {  
    // Already matched  
}
```

```
try{int x = 1; throw x;}  
catch(long int li) {  
    // No match  
}  
catch (double x) {  
    // No match  
}  
catch (...) {  
    // Matched  
}
```

Exception Hierarchy

The fact that exception instances can be caught by their base class type is actually very useful in reality. You can catch a “class” of exceptions while leaving others un touched.

```
class Exception {};  
class IntegerException : public Exception {};  
class FileException    : public Exception {};  
class FileNotFoundException : public FileException {};  
class PermissionDenied : public FileException {};  
void doSomethingToFile(string filename);  
void foo(int n) {  
    string file = "str" + to_string(n);  
    try { doSomethingToFile(file) }  
    catch (const FileException& e) {  
        cout << "Something wrong with File";  
    }  
}
```

Exception Hierarchy in standard library

`std::exception`

Defined in header `<exception>`

```
class exception;
```

Provides consistent interface to handle errors through the `throw` expression.

All exceptions generated by the standard library inherit from `std::exception`

- `logic_error`
 - `invalid_argument`
 - `domain_error`
 - `length_error`
 - `out_of_range`
 - `future_error(C++11)`
 - `bad_optional_access(C++17)`
- `runtime_error`
 - `range_error`
 - `overflow_error`
 - `underflow_error`
 - `regex_error(C++11)`
 - `tx_exception(TM TS)`
 - `system_error(C++11)`
 - `ios_base::failure(C++11)`
 - `filesystem::filesystem_error(C++17)`
- `bad_typeid`

Tips

- 1 Throw by value and catch by (const) reference.
- 2 throw only on real exceptions

Throw by value

When an exception is raised, the programming is doing sort of recovery. The function that throws the exceptions most likely is going to terminate. Thus throwing reference (or address) to local objects won't make sense at the handling site.

Catch by (const) reference

The very reason that you need to catch exceptions by reference is simply because exceptions can contain virtual functions (polymorphism).