1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the `try` block
   - the `out_of_range` exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using gdb
Bugs, Exceptions, and Testing

1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the `try` block
   - the `out_of_range` exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using `gdb`
Classifying Errors

Three types of errors:

1. code does not compile: syntax or semantic error
   Compilers improve, catch more errors ...

2. program crashes: run-time error
   Wrong user input or forgot exception handler?

3. incorrect results: logic error
   Preconditions and postconditions may lead to formal proof of correctness.

Avoiding and fixing errors:

before: prepare for testing
after: debugging code
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter a number : ";
    cin >> n;
    cout << "your number " << n;
    if(n == 0)
        cout << " is zero" << endl;
    else
        cout << " is nonzero" << endl;

    return 0;
}
Many Wrongs...

Running `test4zero` ...

```
$ ./test4zero
Enter a number : 9
your number 9 is nonzero

$ ./test4zero
Enter a number : 0
your number 0 is nonzero

$ ./test4zero
Enter a number : a
your number 0 is nonzero
```


interpret the compiler messages, do `g++ -Wall`

test4zero.cpp:17:9: warning: using the result of an assignment as a condition without parentheses [-Wparentheses]
   if(n = 0)
     ~~^~~
test4zero.cpp:17:9: note: place parentheses around the assignment to silence this warning
   if(n = 0)
     ^
     (  )
test4zero.cpp:17:9: note: use ‘==’ to turn this assignment into an equality comparison
   if(n = 0)
     ^
     ==

1 warning generated.
Bugs, Exceptions, and Testing

1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the try block
   - the out_of_range exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using gdb
force type checking

Instead of

```
$ ./read_integer
Enter an integer number : abc
-> your number : 0
```

we want to force type checking on input:

```
$ ./force_type_check
Enter an integer number : abc
terminate called after throwing an instance of 'std::ios_base::failure'
  what(): basic_ios::clear
Abort trap
```

*crash after exception thrown*
#include <iostream>

using namespace std;

int main()
{
    int n;

    cin.exceptions(ios_base::badbit | ios_base::failbit);

    cout << "Enter an integer number : ";
    cin >> n;
    cout << "your number " << n << endl;

    cin.clear();

    return 0;
}
We prompt for an integer in a loop:

$ ./read_integer1
Enter an integer number : a
Enter an integer number : abc
Enter an integer number : 9
-> your number : 9
$

Continue as long as `cin >> n` fails.

In retry:
1. do `cin.clear();` and
2. skip the end of line symbol.
```cpp
#include <iostream>
#include <limits>
using namespace std;

int main()
{
    int n;

    do
    {
        cout << "Enter an integer number : ";
        if(cin >> n) break;
        cin.clear(); // clear failed state of cin
        cin.ignore(numeric_limits<int>::max(),'
');
    }
    while(true);
    cout << "-> your number : " << n << endl;
    return 0;
}
```
opening a file

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    string input_file_name;

    cout << "Give name of input file : ";
    cin >> input_file_name;

    ifstream ins(input_file_name.c_str());

    if(!ins)
    {
        cout << "Opening " << input_file_name
             << " failed!" << endl;
    }
}
```
Bugs, Exceptions, and Testing

1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the try block
   - the out_of_range exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using gdb
The `try` Block

To guard code against exceptions:

```cpp
try
{
  // code may throw exception
}
catch( exception-type parameter )
{
  // code to handle exception
}

An exception-type is e.g.: `ios_base::failure`.
```
reading integer (again)

```cpp
int n;

cin.exceptions(ios_base::badbit | ios_base::failbit);
do {
    cout << "Enter an integer number : ";
    try {
        cin >> n; break;
    }
    catch(ios_base::failure) {
        cin.clear(); // clear failed state of cin
        cin.ignore(numeric_limits<int>::max(),'
');
    }
}
while(true);
cout << "-> your integer : " << n << endl;
```
Bugs, Exceptions, and Testing

1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the `try` block
   - the `out_of_range` exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using gdb
Selecting Characters

To select a character of a string $s$, there are two ways:

1. with the operator $[$ $]$, as $s[k]$;
2. with the at method, as $s.at(k)$.

The difference?

If index is not in the range of $s$, then at method throws `std::out_of_range`.

The operator $[$ $]$ throws no exception.
using [ ] and at()

```cpp
int main()
{
    string s;

    cout << "Give a string : "; cin >> s;

    int k;
    cout << "Give an index : "; cin >> k;

    cout << "char at " << k
        << " : " << s[k] << endl;

    cout << "char at " << k
        << " : " << s.at(k) << endl;

    return 0;
}
```

much can go wrong...

Prompting for an index to a string,

1. the index may not be an integer; or
2. the index may be out of range; or
3. some other exception may occur.

```
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

int main()
{
    string s;

    cout << "Give a string : "; cin >> s;
    cin.exceptions(ios_base::badbit | ios_base::failbit);
```
hierarchy of exceptions

try
{
    int k;
    cout << "Give an index : "; cin >> k;
    cout << "char at " << k
         << " : " << s.at(k) << endl;
} catch(ios_base::failure &e)
{
    cerr << "index is not an integer" << endl;
    cerr << e.what() << endl;
} catch(out_of_range &e)
{
    cerr << "index out of range" << endl;
    cerr << e.what() << endl;
} catch(exception &e)
{
    cerr << "some fatal error occurred" << endl;
    cerr << e.what() << endl;
}
parameters of exceptions

running the program `except_hierarchy`:

$ ./except_hierarchy
Give a string : abc
Give an index : 9
index out of range
basic_string::at: __n (which is 9) >= this->size() (which is 3)
$

The last line is the result of

```cpp
    cout << e.what() << endl;
```

where `e` is the argument of

```cpp
    catch(out_of_range &e)
```
Bugs, Exceptions, and Testing

1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the `try` block
   - the `out_of_range` exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using gdb
converting `int` to `char`

To convert an `int` to `char`, we could use

```c
char to_char ( int n )
{
    return char(n);
}
```

However:

1. `char` is unsigned integer;
2. `char` is only 8 bits, `int` is 32 bits.
throwing an exception

If range check fails, throw bad_cast:

```cpp
char to_char ( int n )
{
    if((n < 0) || (n > 255))
    {
        cerr << "Throwing bad_cast exception ...";
        throw(bad_cast());
    }
    return char(n);
}
```
int2char converts n into c
returns true if okay, false otherwise

bool int2char ( int n, char& c )
{
    try
    {
        c = to_char(n);
        return true;
    }
    catch(bad_cast)
    {
        cerr << " caught bad_cast exception\n";
        return false;
    }
}
Bugs, Exceptions, and Testing

1 Bugs
   - classifying errors
   - forcing type checking on input

2 Catching and throwing exceptions
   - the try block
   - the out_of_range exception
   - throwing exceptions

3 Testing and Debugging
   - categories of software testing
   - using gdb
Categories of Software Testing

**white-box testing:** input based on internal structure;
**black-box testing:** input based on specification;

An extra dimension:

- **static:** read specification or source code;
- **dynamic:** execute software or test programs.

1. Static black-box testing: test the specification.
2. Static white-box testing: inspect the code.
3. Dynamic black-box testing: beta testing.

**Verification:** does software meet its specification?
**Validation:** does software meet user requirements?
Bugs, Exceptions, and Testing

1. Bugs
   - classifying errors
   - forcing type checking on input

2. catching and throwing exceptions
   - the try block
   - the out_of_range exception
   - throwing exceptions

3. Testing and Debugging
   - categories of software testing
   - using gdb
using gdb

gdb is the GNU debugger

To use gdb: compile code as g++ -g.

Capabilities of gdb:
1. set break points at line in source code
2. step by step execution
3. printing values
4. examine stack of function calls
**gdb on an example**

Using `int2char` in `in_char_range.cpp`:

1. compile with option `-g`
   
   ```
   $ g++ -g -o in_char_range in_char_range.cpp
   ```

2. launch debugger
   
   ```
   $ gdb in_char_range
   ```

3. set a break point
   
   The call `bool okay = int2char(n, c)` at line 31 is where the action happens.

   ```
   (gdb) b 30
   Breakpoint 1 at 0x1ae2: file \in_char_range.cpp, line 30.
   ```
running **gdb** continued

5 run the program

(gdb) r
Starting program: in_char_range
Give an integer : -1

Breakpoint 1, main () at in_char_range.cpp:31
31 bool okay = int2char(n,c);

6 stepwise execution

(gdb) step
int2char (n=-1, c=0xbfffff587) \ at in_char_range.cpp:58
58 c = to_char(n);
(gdb) step
to_char (n=-1) at in_char_range.cpp:46
46 if((n < 0) || (n > 255))
stack of function calls

1. do a backtrace

(gdb) bt
#0  to_char (n=-1) at in_char_range.cpp:46
#1  0x000019db in int2char (n=-1, c=@0xbfffff587) \
at in_char_range.cpp:58
#2  0x00001af4 in main () at in_char_range.cpp:31

2. continue to the end

(gdb) step
48        cerr << "Throwing bad_cast exception ...";
(gdb) step
Throwing bad_cast exception ... caught \nbad_cast exception.
Cannot convert -1 to a character.

Program exited normally.
examining values

Checking if `okay` value is right, with `print`. Line 40 is before `return 0;` we restart `gdb`:

```plaintext
(gdb) b 40
Breakpoint 1 at 0x1bcc: file in_char_range.cpp, line 40.
(gdb) r
Starting program: in_char_range
Give an integer : 98
The integer 98 corresponds to character 'b'.
```

Breakpoint 1, main () at in_char_range.cpp:41
41     return 0;
(gdb) print okay
$1 = true
Summary + Exercises

Starting Chapter 2: *Program Correctness and Efficiency*, we explored the use of exceptions in C++.

Exercises:

1. Write a function that prompts the user for an age. Throw an exception when the age is negative.

2. Include `cmath` or `math.h` and verify that `sqrt(x)` for `x` a negative double returns `nan` (not a number). Write a function `double MySqrt(double x)` that throws a `domain_error` exception for `x < 0`.

3. Prompt the user for two integers `p` and `q`. Compute `p/q`. Catch the exception `q = 0` and show an error message.