Exception Safety and Exception Handling

GKxx

July 4, 2022

Contents

strcpy

You are asked to write a strcpy function...

```
void strcpy(char *dest, const char *source) {
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

strcpy

You are asked to write a strcpy function...

```
void strcpy(char *dest, const char *source) {
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

In reality, things may go wrong:

- Null pointers?
- Buffer overflow?

We may not be able to detect buffer overflow.

Which is Better?

1. Terminate the program on failure and report the error.

```
void strcpy(char *dest,
    const char *source) {
  if (!dest || !source) {
    std::cerr << "Invalid
        arguments for
        strcpy.\n";
    exit(1);
  }
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

2. Return false on failure:

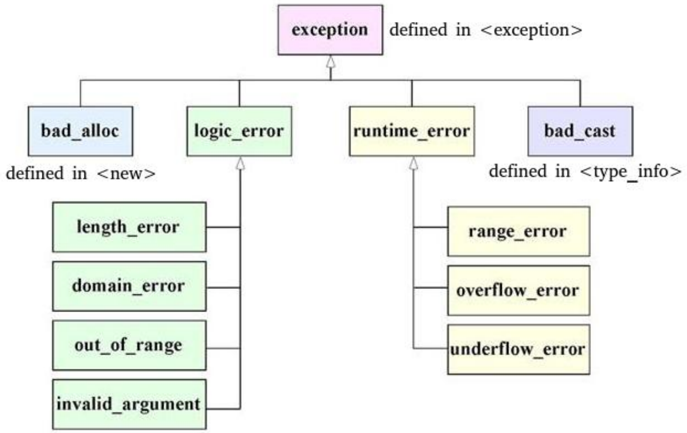
```
bool strcpy(char *dest,
    const char *source) {
  if (!dest || !source)
    return false;
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
  return true;
}
```

3. Be silent and just let the user ensure that the arguments are valid.

Throwing an Exception

```
void strcpy(char *dest, const char *source) {
  if (!dest || !source)
    throw std::invalid_argument("Null pointers passed
        to strcpy.");
  while (*source)
    *dest++ = *source++;
  *dest = '\0';
}
```

Standard Exceptions



- logic_error, runtime_error and their subclasses are defined in <stdexcept>.

Standard Exceptions

- The normal `new` and `new[]` operators throw `std::bad_alloc` when running out of memory.
- `dynamic_cast` for references throws `std::bad_cast` when the casting fails.
  - `dynamic_cast` for pointers does not throw. It returns `nullptr` on failure.

Standard Exceptions

- The normal `new` and `new[]` operators throw `std::bad_alloc` when running out of memory.
- `dynamic_cast` for references throws `std::bad_cast` when the casting fails.
  - `dynamic_cast` for pointers does not throw. It returns `nullptr` on failure.
- `std::system_error` is thrown in many cases, especially in functions that interface with OS facilities, e.g. the constructor of `std::thread`.
- `<chrono>` defines `std::nonexistent_local_time` and `std::ambiguous_local_time`.

Standard Exceptions

operator[] for STL containers does not check boundaries, but at() does.

```
std::vector<int> v;
v.at(0) = 42; // Throws std::out_of_range.
v[0] = 42; // Does not throw, but probably causes a
    segmentation fault.
```

We will see that exceptions thrown could be catched and handled.

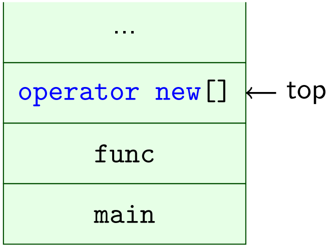
Standard Exceptions

Let our Array do the same thing?

```
template <typename T>
class Array {
 public:
  T &at(std::size_t n) {
    if (n >= m_size)
      throw std::out_of_range("Array subscript out of
          range.");
    return m_data[n];
  }
  const T &at(std::size_t n) const {
    // ...
  }
  // ...
};
```

Call Stack Unwinding

```cpp
void func(int n) {
  int x = 42;
  int *p = new int[n];
  // ...
}
int main() {
  int size = 100;
  func(size);
  // ...
}
```



Suppose `operator new[]` encounters shortage of memory...