MANCHESTER 1824
The University of Manchester

COMP26020 Programming Languages and Paradigms

Lecture 40: Move semantics

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk

new considered harmful

Manual allocation → manual deallocation

Memory leaks

Use-after-free

RAII

Dynamic memory management in automatic objects

- Constructor + methods allocate memory
- Destructor deallocates memory
- Destructor is called automatically

Problem solved, right?

Traditional C++ still needs *new*

1. Creating and returning large objects
2. Dynamic lifetime patterns

Traditional C++ still needs *new*

1. Creating and returning large objects

2. Dynamic lifetime patterns

Returning large objects

```
{
  std::ifstream infile(“numbers.txt”); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
}
```

Returning large objects

```
std::vector<int> readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
  return v;
}

auto v = readfile(“numbers.txt”); // copy ret value into v
```

Returning large objects

```
std::vector<int> readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
  return v;
}

auto v = readfile("numbers.txt"); // copy ret value into v
```

```
template <typename T>
vector<T>::operator= (vector<T>& other) {
  for (auto& elem: other)
    push_back(elem);
}
```

Returning large objects - Workarounds

By Value

```
std::vector<int> readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
  return v;
}
```

By Ref

```
void readfile (std::string fname, std::vector<int>& v) {
  std::ifstream infile(fname); // Create and open file
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
}
```

By Pointer

```
std::vector<int>* readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  auto v = new std::vector<int>;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v->push_back(num);
  return v;
}
```

Unintuitive semantics

new → delete?

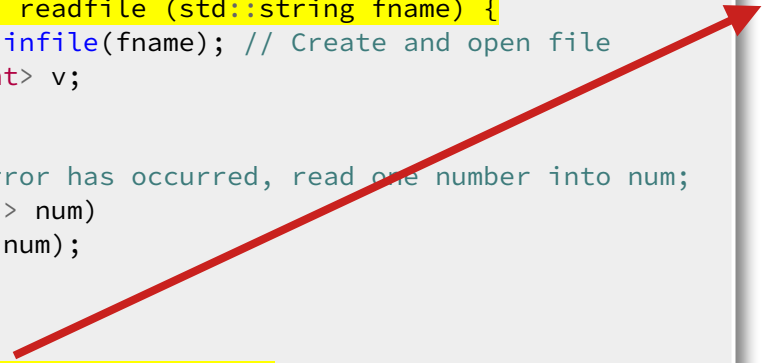
This is just typical C++ making our lives difficult

Returning large objects

```
std::vector<int> readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
  return v;
}

auto v = readfile("numbers.txt"); // copy ret value into v
```



Return value → Temporary

- Read it
- Copy it
- Then delete it

Returning large objects

```
std::vector<int> readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
  return v;
}

auto v = readfile("numbers.txt"); // copy ret value into v
```

Pointless! Why not:

- Use ret value directly
- No copy, no delete!
- Stack values, scopes, etc

Returning large objects

```
template <typename T>
class vector {
  T* data;
  size_t size;
  size_t capacity;
public:
  …
}
```

Even stack objects have parts on the heap!

Standard containers:

- Mostly on the heap
- Do we really need to copy this?

Move semantics

rvalue references → Obj&&

About to be deleted

Obj&& std::move(Obj&)

Move constructors → argument is an rvalue

	Copy	Move
Constructor	Obj::Obj(Obj& other);	Obj::Obj(Obj&& other);
Assignment	Obj::operator=(Obj& other);	Obj::operator=(Obj&& other);

Move constructors!

All standard containers have move constructors

Move ownership of allocated resources

```
template <typename T>
vector<T>::operator= (vector<T>& other) {
  resize(other.size);
  for (auto it = begin(), last = end(),
       other_it = other.begin();
       it != last; ++it, ++other_it) {
    *it = *other_it;
  }
}
```

O(N) complexity
Read, allocate, copy N Ts

```
template <typename T>
vector<T>::operator= (vector<T>&& other) {
  capacity = other.capacity;
  size = other.size;
  std::swap(data, other.data);
}
```

O(1) complexity
Copy three values

Returning large objects

```
std::vector<int> readfile (std::string fname) {
  std::ifstream infile(fname); // Create and open file
  std::vector<int> v;
  int num;

  // While no error has occurred, read one number into num;
  while(infile >> num)
    v.push_back(num);
  return v;
}

auto v = readfile(“numbers.txt”); // move ret value into v
```

Returning standard containers almost as fast as returning pointers Much clearer & safer!

Returning large objects

```
std::vector<int> readfile (s          ing fname)
  std::ifstream infile(fname,                and
  std::vector<int> v;
  int num;

  // While no e
  while(infile >>
    v.push_back(num);
  return v;
}

auto v = read
```

Unless they are already on the heap or another object owns them
Return objects by value

almost as retu g pointers
Mu learer & safer!

Move constructors

Usually you don’t need to define them

- Containers provide them
- Implicit move constructors if no dynamic resources (rule of five)

Recap

- Old days
  - Return → Copy value
  - Expensive for large objects
  - Pointers to heap
- Move semantics
  - Return/Temp values handled separately
  - Special constructors which transfer ownership, not copy
  - std containers do that!

Up Next

Smart pointers