Debugging

- Just print
- Use gdb

gdb

- Pause execution

gdb

- Pause execution
- Step through the code

gdb

- Pause execution
- Step through the code
- Inspect variables

gdb

- Pause execution
- Step through the code
- Inspect variables
- Modify variables and call functions

gdb

- Pause execution
- Step through the code
- Inspect variables
- Modify variables and call functions

```
g++ -g -O0 myprog.cpp
gdb ./a.out
(gdb) break main
(gdb) run
(gdb) step
(gdb) print x
(gdb) set x = 5
(gdb) continue
```

- -g: Generate debugging information
- -O0: No optimization

Debug and CMake

Includes -g

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Debug and CMake

Includes -g

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Good idea to enable warnings

```
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
  if (MSVC)
    add_compile_options(/W4 /Od)
  else()
    add_compile_options(-O0 -Wall -Wextra -Wpedantic)
  endif()
endif()
```

Common uses of pointers

- Returning large objects

```
Vector* createBigClass() {
  auto vec = new Vector;
  // Fill the vector ptr
  return vec;
}
```

Common uses of pointers

- Returning large objects
- Passing interfaces as arguments (OOP)

```
void processInterface(Interface* iface) {
  iface->doSomething();
}
...
Implementation impl;
processInterface(&impl);
...
```

Common uses of pointers

- Returning large objects
- Passing interfaces as arguments (OOP)
- Passing elements of a container to an algorithm

```
std::vector = {5, 4, 3, 2, 1};
std::sort(vector.begin(), vector.end());
```

Common uses of pointers

- Returning large objects
- Passing interfaces as arguments (OOP)
- Passing elements of a container to an algorithm
- Implementing high-level types (`std::vector`)

```
class DynamicArray{
  int* data;
  size_t size;
  ...
};
```

A problem with pointers

```cpp
Vector* createBigClass() {
  auto vec = new Vector;
  // Fill the vector ptr
  return vec;
}
int main() {
  auto vec = createBigClass();
  // Use the vector
  delete vec;
}
```

- Who deletes the object?
- Always use RAII

Smart pointers

```
Shape* read_shape(std::istream &is) {
 //...Read a variety of shapes...
}
```

- User must remember to delete the object

Unique pointer

```cpp
std::unique_ptr<Shape> read_shape(std::istream &is) {
  std::string shape_type;
  is >> shape_type;
  if(shape_type == "circle") {
    // Like new
    return std::make_unique<Circle>(center, radius);
  }
  //... other shapes ...
}
```

- Object is deleted automatically

Unique pointer

```
void no_good(){
  auto p = std::make_unique<X>();
  auto q = p; // Error: fortunately
} // here p and q would delete the object
```

- Who deletes the object?

Unique pointer: Vector again

```
class Vector{
  int *data;
  size_t size;
public:
  Vector(size_t size) :
    size(size),data(new int[size]) {}
  // Rule of all:
  // Destructor, copy/move
   ↪  constructor/operators
};
```

Unique pointer: Vector again

```cpp
class Vector{
  std::unique_ptr<int[]> data;
  size_t size;
public:
  Vector(size_t size) :
    size(size),
    data(std::make_unique<int[]>(size)){}

  Vector(const Vector& other) :
    Vector(other.size) {
    std::copy(other.data.get(), other.data.get() + size,
              data.get());
  }
};
```

- `unique_ptr` provides move semantics and RAII.

Shared pointer

```
void very_good(){
auto p = std::make_shared<X>();
auto q = p; // OK: shared ownership
} // q (last to die) deletes the object
```

- shared_ptr is reference counted
- Object is deleted when the last owner is destroyed

Custom deleters

```
void close_file(FILE* file) {
  std::cout << "Closing file" << std::endl;
  fclose(file);
}
void foo(){
  std::shared_ptr<FILE> file(fopen("file.txt", "r"),
                              close_file);
  // Use file like a regular pointer
} // close_file is called automatically when file goes
↪  out of scope
```

- Control how the object is deleted

Resource management

Always try in this order:

1. Containers

Resource management

Always try in this order:

1. Containers
   - `std::vector`, `std::map`, etc.

Resource management

Always try in this order:

1. Containers
   - `std::vector`, `std::map`, etc.
2. Smart pointers

Resource management

Always try in this order:

1. Containers
   - `std::vector`, `std::map`, etc.
2. Smart pointers
   - `std::unique_ptr`, `std::shared_ptr`

Resource management

Always try in this order:

1. Containers
   - `std::vector`, `std::map`, etc.
2. Smart pointers
   - `std::unique_ptr`, `std::shared_ptr`
3. Raw pointers

Resource management

Always try in this order:

1. Containers
   - `std::vector`, `std::map`, etc.
2. Smart pointers
   - `std::unique_ptr`, `std::shared_ptr`
3. Raw pointers

- Strive for a "no naked new" policy.

Smart pointers

- std::unique_ptr<T>

Smart pointers

- `std::unique_ptr<T>`
    - Only one owner, transfers ownership

Smart pointers

- `std::unique_ptr<T>`
  - Only one owner, transfers ownership
  - Deletes the object when the owner is destroyed

Smart pointers

- `std::unique_ptr<T>`
  - Only one owner, transfers ownership
  - Deletes the object when the owner is destroyed
- `std::shared_ptr<T>`

Smart pointers

- `std::unique_ptr<T>`
  - Only one owner, transfers ownership
  - Deletes the object when the owner is destroyed
- `std::shared_ptr<T>`
  - Multiple owners, reference counting

Smart pointers

- `std::unique_ptr<T>`
  - Only one owner, transfers ownership
  - Deletes the object when the owner is destroyed
- `std::shared_ptr<T>`
  - Multiple owners, reference counting
  - Deletes the object when the last owner is destroyed

Smart pointers

- std::unique_ptr<T>
  - Only one owner, transfers ownership
  - Deletes the object when the owner is destroyed
- std::shared_ptr<T>
  - Multiple owners, reference counting
  - Deletes the object when the last owner is destroyed
- std::weak_ptr<T>

Smart pointers

- `std::unique_ptr<T>`
  - Only one owner, transfers ownership
  - Deletes the object when the owner is destroyed
- `std::shared_ptr<T>`
  - Multiple owners, reference counting
  - Deletes the object when the last owner is destroyed
- `std::weak_ptr<T>`
  - non-owning ("weak") reference to an object managed by `std::shared_ptr`