PROGRAMMING IN C++

Jülich Supercomputing Centre

8 – 12 May 2023 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

JÜLICH Forschungszentrum

Day 2

What is static typing?

- At creation, every variable must have a type that is known to the compiler, and that type can not change for its entire lifetime.
- Programs can only produce outcomes which can be deduced entirely from source code irrespective of runtime inputs.
- Both of the above.
- The uncanny ability of many C++ programmers to type their programs without moving their fingers.

1
```
while (true) { do_something(); }
```

2
```
for (;;) { do_something(); }
```

3
```
for (auto i=0.0F ; i < 1000'000'000; ++i) { do_something(); }!
```

Which ones above are infinite loops ?

A. 1 alone

B. 2 alone

C. 1 and 2

D. All of them

Stack execution model

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

f() int i=10

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

JÜLICH Forschungszentrum

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()	g() int i = 10

JÜLICH Forschungszentrum

```cpp
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```cpp
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```cpp
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```cpp
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

```cpp
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```cpp
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```cpp
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```cpp
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

h11() int i = 10

```
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

h1() int i = 10

```cpp
auto main() -> int
{
    auto N = 10;
    if (f(N) < g(N)) {
        h1(N);
    } else {
        h2(N);
    }
}
```

```cpp
auto f(int i)() -> int
{
    return (i * i) %12;
}
auto g(int i)() -> int
{
    return i % 12;
}
auto h1(int i)() -> int
{
    return h11(i);
}
 auto h2(int i)() -> int
{
    return h21(i);
}
```

```cpp
auto h11(int i)() -> int
{
    return i * i;
}
auto h21(int i)() -> int
{
    return i + h211(i);
}
```

```cpp
auto h211(int i)
    -> int
{
    return -i;
}
```

main()

FUNCTIONS AT RUN TIME

Sin(double x)
x:0.125663..
RP:<in main()>

main()
x:3.14159265...
i:4
RP:OS

```cpp
auto sin(double x) -> int {
  // Somehow calculate sin of x
  return answer;
}
auto main() -> int {
    double x{3.141592653589793};
    for (int i = 0; i < 100; ++i) {
        std::cout << i * x / 100
            << sin(i * x / 100) <<"\n";
    }
}
```

When a function is called, e.g., when we write `f(value1,value2,value3)` for a function `f` declared as `ret_type f(type1 x, type2 y, type3 z)` :

- A "workbook" in memory called a stack frame is created for the call
- The local variables `x` , `y` , `z` are created, as if using instructions `type1 x{value1}` , `type2 y{value2}` , `type3 z{value3}` .
- A return address is stored.
- The actual body of the function is executed
- When the function concludes, execution continues at the stored return address, and the stack frame is destroyed

FUNCTION ARGUMENTS

```cpp
int x{ 1 };
int y{ x };

y = y + 1;
// What is x now?
```

FUNCTION ARGUMENTS

```cpp
int x{ 1 };
int& y{ x };

y = y + 1;
// What is x now?
```

FUNCTION ARGUMENTS

```
auto f(int x) -> int
{
    x = x + 1;
    return x;
}
void elsewhere()
{
    int z{ 0 };
    f(z);
    // what is z now?
}
```

FUNCTION ARGUMENTS

```cpp
auto f(int& x) -> int
{
    x = x + 1;
    return x;
}
void elsewhere()
{
    auto z = 0;
    f(z);
    // what is z now?
}
```

```cpp
void get_lims(int i, int j)
{
    i = 10;
    j = 20;
}
auto main() -> int
{
    auto i = 2, j = 3;
    get_lims(i,j);
    std::cout << i << ", " << j << "\n";
}
```

What does the `std::cout` line print ?

A. `2, 3`

B. `10, 20`

C. `0, 0`

D. `3, 2`

```cpp
void get_lims(int& i, int& j)
{
    i = 10;
    j = 20;
}
auto main() -> int
{
    auto i = 2, j = 3;
    get_lims(i,j);
    std::cout << i << ", " << j << "\n";
}
```

What does the `std::cout` line print ?

A. `2, 3`

B. `10, 20`

C. `0, 0`

D. `3, 2`

THE REFERENCE TYPE IN FUNCTION PARAMETERS

```
// Argument passed by value
auto find_arsenic_tolerance(Rat R)
    -> double
{
  double qnty = 0, dqnty = 1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty += dqnty;
  }
  return qnty;
}
...
auto lab() -> int
{
  Rat r;
  double t = find_arsenic_tolerance(r);
  // r is still alive! But we know
  // how much arsenic it can take.
}
```

Pass a normal type by value

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat`.
- So you send a copy or clone of `r`
- The clone gets injections and is eventually destroyed.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

```
// Argument passing by reference
auto find_arsenic_tolerance(Rat& R)
    -> double
{
  double qnty = 0, dqnty = 1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty += dqnty;
  }
  return qnty;
}
...
auto lab() -> int
{
  Rat r;
  double t = find_arsenic_tolerance(r);
  // r is no more!
}
```

Pass a reference argument

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat &`, i.e., a reference to which Rat.
- So you send a copy of the Id tag on `r` to the function.
- The function acts on the `Rat` object which was referenced.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

```cpp
// Argument passing by reference
auto find_arsenic_tolerance(Rat& R)
    -> double
{
  double qnty = 0, dqnty = 1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty += dqnty;
  }
  return qnty;
}
...
auto lab() -> int
{
  Rat r;
  double t = find_arsenic_tolerance(r);
  // r is no more!
}
```

Pass a reference argument

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat &`, i.e., a reference to which Rat.
- So you send a copy of the Id tag on `r` to the function.
- The function acts on the `Rat` object which was referenced.

Information about the original rat, but the rat was modified.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

We want to change an object

- When we want our object to be modified in some way by a function, it is no good to pass only a copy.
- In this example, a clone of the wounded leg will be bandaged

```
void bandage_leg(Leg l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
auto main() -> int
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //No benefits to h.
}
```

THE REFERENCE TYPE IN FUNCTION PARAMETERS

We want to change an object

- Modifying a copy of our object is useless
- But a copy of a reference is good enough.
- In this example, the function works on the leg that was referred to.

```cpp
void bandage_leg(Leg & l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
auto main() -> int
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //Intended benefits to h
}
```

THE REFERENCE TYPE IN FUNCTION PARAMETERS

We want to change an object

- Modifying a copy of our object is useless
- But a copy of a reference is good enough.
- In this example, the function works on the leg that was referred to.

```
void bandage_leg(Leg & l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
auto main() -> int
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //Intended benefits to h
}
```

We can use a function working with a reference when we want it to change our original object.

THE REFERENCE TYPE IN FUNCTION PARAMETERS

Cloning is expensive

- Sometimes, the data structures are very large, and copying them is expensive
- Functions taking that kind of classes will implicitly perform big cloning operations, slowing the program down.

```
auto count_bad_tires(Truck t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
auto main() -> int
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // Unnecessary cloning of mytruck
}
```

THE REFERENCE TYPE IN FUNCTION PARAMETERS

Cloning is expensive

- If the function signature asks for a reference, we only create a reference to the truck when invoking the function
- The same effect can be achieved by a pointer, but the syntax with references is cleaner

```cpp
auto count_bad_tires(Truck& t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
auto main() -> int
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // another reference to truck, not
  // clone of truck
}
```

THE CONSTANT REFERENCE TYPE

Cloning is expensive

- We want to use a reference as the argument because it is efficient
- How do we ensure that the original object would not be allowed to change ?

```cpp
auto count_bad_tires(Truck& t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      check_pressure(t.wheel(i));
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
auto main() -> int
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // Was there any change to mytruck ?
}
```

THE CONSTANT REFERENCE TYPE

Cloning is expensive

- We want to use a reference as the argument only because it is efficient
- How do we ensure that the original object would not be allowed to change ?
- Using a `const reference`

```
auto count_bad_tires(const Truck& t) -> int
{
  int n = 0;
  for (int i = 0; i < t.n_wheels(); ++i) {
      check_pressure(t.wheel(i));
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
int main()
{
  Truck mytruck;
  ...
  nbad = count_bad_tires(mytruck);
  // Was there any change to mytruck ?
  // Not if this compiled!
}
```

Runtime error handling

RUN-TIME ERROR HANDLING

Exceptions: When there is nothing reasonable to return

```
auto mysqrt(double x) -> double
{
    const auto eps2 = 1.0e-24;
    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

Exceptions

- A function may be called with arguments which don't make sense
- An illegal mathematical operation
- Unexpected values, e.g., an arbitrary string when expecting a number
- Too much memory might have been requested

THROWING AND CATCHING EXCEPTIONS

```cpp
using error_code = int;
auto mysqrt(double x) -> double
{
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) throw error_code{-1};
    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    try {
        if (x < y) std::swap(x, y);
        return mysqrt(x + y) + mysqrt(x - y);
    } catch (error_code& error) {
        std::cout << "Caught error_code: "
                  << error << "\n";
        // somehow fix the situation and
        // return something sensible. If that
        // doesn't work...
        throw;
    }
}
```

THROWING AND CATCHING EXCEPTIONS

```cpp
using error_code = std::string;
auto mysqrt(double x) -> double
{
    using std::format;
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) throw
        format("Bad input {} for mysqrt", x);

    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    try {
        if (x < y) std::swap(x, y);
        return mysqrt(x + y) + mysqrt(x - y);
    } catch (error_code& error) {
        std::cout << "Caught error_code: "
                  << error << "\n";
        // somehow fix the situation and
        // return something sensible. If that
        // doesn't work...
        throw;
    }
}
```


JÜLICH Forschungszentrum

THROWING AND CATCHING EXCEPTIONS

```cpp
auto mysqrt(double x) -> double
{
    using std::format;
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) throw
      std::runtime_error{
        format("Bad input {} for mysqrt", x)
      };

    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    try {
        if (x < y) std::swap(x, y);
        return mysqrt(x + y) + mysqrt(x - y);
    } catch (std::runtime_error& error) {
        std::cout << "Caught runtime error: "
                  << error.what() << "\n";
        // somehow fix the situation and
        // return something sensible. If that
        // doesn't work...
        throw;
    }
}
```

TRY AND CATCH BLOCKS

```
void f() {
  try {
  // lines
    try {
      // a line throwing an exception
    } catch (exception_type_0& err) {
      // handle errors of type 0
    }
  // more lines
  } catch (exception_type_1& err) {
  // handle errors of type 1
  } catch (exception_type_2& err) {
  // ...
  }
}
void g(int i) { if (i > -3) f(); }
auto main(int argc, char* argv[]) -> int {
    try {
      g(argc)
    } catch (exception_type_3& err) {
    // handle error type 3
    }
}
```

- Exceptions are monitored and handled in `try..catch` blocks
- When an exception is thrown in the `try` part of a `try..catch` block, the attached `catch` blocks are checked checked for a handler matching the **type** of the thrown exception.
- If no matching handler is found, we look for an next bigger `try..catch` block
- If an exception is thrown in an area inside a function, not inside a `try` section, the enclosing `try` section is searched based on the *call site* for the call
- This search can unwind till it reaches `main()`. If still no matching handler is found, the program exits with error.

TRY AND CATCH BLOCKS

```
void f() {
  try {
  // lines
    try {
      // a line throwing an exception
    } catch (exception_type_0& err) {
      // handle errors of type 0
    }
  // more lines
  } catch (exception_type_1& err) {
  // handle errors of type 1
  } catch (exception_type_2& err) {
  // ...
  }
}
void g(int i) { if (i > -3) f(); }
auto main(int argc, char* argv[]) -> int {
    try {
      g(argc)
    } catch (exception_type_3& err) {
    // handle error type 3
    }
}
```

- Once an exception is thrown, the program control flow enters a special mode
- Imagine all other lines, except `try..catch` blocks and the `throw` expression being "grayed out"
- In this view, the code looks like a smallish tree of `try..catch` blocks. Find the the smallest enclosing `catch` block with the matching type!
- The type matching and jump destinations can all be determined by the compiler
- This jump in program control still follows all the rules regarding variable scopes: when we leave a block of code by flying away on the back of an exception, **it still counts as leaving the block**. Automatic variables declared in that scope are therefore destroyed.

IS IT NEEDLESSLY EXPENSIVE TO USE EXCEPTIONS?

```cpp
auto f(double x, bool& succeeded) -> double
{
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0) {
        succeeded = false;
    } else {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        succeeded = true;
    }
    return r1;
}
```

```cpp
auto appl(double x, double y) -> double
{
    if (x < y) std::swap(x, y);
    bool ep{false}, em{false};
    auto rp = f(x + y, ep);
    auto rm = f(x - y, em);
    if (ep and em) {
        return rp + rm; // normal case
    } else {
        // handle errors
    }
}
```

- Cumbersome because of extra flag variables
- A value is returned even in the case of failure. A programmer can accidentally or out of carelessness, ignore checking the success/error flags. The subsequent calculations will be incorrect.

IS IT NEEDLESSLY EXPENSIVE TO USE EXCEPTIONS?

```cpp
auto f(double x) -> double
{
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x < 0)
        throw std::runtime_error{
          format("Bad input {} for square root!", x)
        };
    auto r0 = 0.5 * (1. + x);
    auto r1 = x / r0;
    while ((r0 - r1) * (r0 - r1) > eps2) {
        r0 = 0.5 * (r0 + r1);
        r1 = x / r0;
    }
    return r1;
}
```

An error handling method with functionality comparable to exceptions will have a similar cost!

```cpp
auto appl(double x, double y) -> double
{
    if (x < y) std::swap(x, y);
    try {
        return f(x + y) + f(x - y);
    } catch (std::runtime_error& err) {
      // handle errors
    }
}
```

- Normal, successful flow is separated from error handling code
- In case there is an error, it is impossible to ignore! The function does not return with a value. The only choices are to handle the error or to terminate the program.

NOEXCEPT

```
auto sum(unsigned i, unsigned j)
    -> unsigned {
    return i + j;
}
void contained(int i) {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions

NOEXCEPT

```
auto sum(unsigned i, unsigned j) noexcept
    -> unsigned {
    return i + j;
}
void contained(int i) noexcept {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions
- Such functions can be decorated with the `noexcept` specifier to tell the compiler that it does not need to make arrangements about propagating exceptions out of those functions

NOEXCEPT

```
auto sum(unsigned i, unsigned j) noexcept
    -> unsigned {
    return i + j;
}
void contained(int i) noexcept {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions
- Such functions can be decorated with the `noexcept` specifier to tell the compiler that it does not need to make arrangements about propagating exceptions out of those functions
- By discarding some exception handling code, the compiler may in some cases generate better optimised code

NOEXCEPT

```cpp
auto sum(unsigned i, unsigned j) noexcept
    -> unsigned {
    return i + j;
}
void contained(int i) noexcept {
    try {
    // some code
    } catch (ET_1& err) {
    } catch (ET_2& err) {
    } catch (...) {
    // handle every exception
    }
}
```

- Sometimes, we know that an exception will never escape certain functions
- Such functions can be decorated with the `noexcept` specifier to tell the compiler that it does not need to make arrangements about propagating exceptions out of those functions
- By discarding some exception handling code, the compiler may in some cases generate better optimised code
- If you lie, and call a function `noexcept`, but an exception reaches the outer most block of the function, the program is `std::terminate`d.

OPTIONAL VALUES

```cpp
#include <optional>
auto f(double x) -> std::optional<double> {
    std::optional<double> ans;
    const auto eps2 = 1.0e-24;
    if (x >= 0) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        ans = r1;
    }
    return ans;
}
// Elsewhere...
std::cout << "Enter number : ";
std::cin >> x;
if (auto r = f(x); r.has_value()) {
    std::cout << "The result is "
              << r.value() << '\n';
}
```

- `std::optional<T>` is analogous to a box containing exactly one object of type `T` or nothing at all
- If created without any initialisers, the box is empty
- You store something in the box by assigning to the `optional`
- Evaluating the optional as a boolean gives a `true` outcome if there is an object inside, irrespective of the value of that object
- Empty box evaluates to `false`

OPTIONAL VALUES

```cpp
#include <optional>
auto f(double x) -> std::optional<double> {
    std::optional<double> ans;
    const auto eps2 = 1.0e-24;
    if (x >= 0) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        ans = r1;
    }
    return ans;
}
// Elsewhere...
std::cout << "Enter number : ";
std::cin >> x;
if (auto r = f(x); r) {
    std::cout << "The result is "
              << *r << '\n';
}
```

- `std::optional<T>` is analogous to a box containing exactly one object of type `T` or nothing at all
- If created without any initialisers, the box is empty
- You store something in the box by assigning to the `optional`
- Evaluating the optional as a boolean gives a `true` outcome if there is an object inside, irrespective of the value of that object
- Empty box evaluates to `false`

C++23 STD::EXPECTED

```cpp
#include <expected>
auto mysqrt(double x) -> std::expected<double, std::string> {
    const auto eps = 1.0e-12;
    const auto eps2 = eps * eps;
    if (x >= 0.) {
        auto r0 = 0.5 * (1. + x);
        auto r1 = x / r0;
        while ((r0 - r1) * (r0 - r1) > eps2) {
            r0 = 0.5 * (r0 + r1);
            r1 = x / r0;
        }
        return { r1 };
    } else {
        return std::unexpected { "Unexpected input!" };
    }
}
// Elsewhere...
if (auto rm = mysqrt(x); rm) std::cout << "Square root = " << rm.value() << "\n";
else std::cout << "Error: " << rm.error() << "\n";
```

- Similar to `std::optional`, but has more capacity to describe the error
- The *unexpected* value can be of a type of our choosing, making it very flexible

ASSERTIONS

```cpp
#include <cassert>
bool check_things()
{
    // false if something is wrong
    // true otherwise
}
double somewhere()
{
    // if I did everything right,
    // val should be non-negative
    assert(val >= 0);
    assert(check_things());
}
```

- `assert(condition)` aborts if `condition` is false
- Used for non-trivial checks in code during development. The errors we are trying to catch are logic errors in implementation.
- If the macro `NDEBUG` is defined before including `<cassert>` `assert(condition)` reduces to nothing

ASSERTIONS

```cpp
#include <cassert>
bool check_things()
{
    // false if something is wrong
    // true otherwise
}
double somewhere()
{
    // if I did everything right,
    // val should be non-negative
    assert(val >= 0);
    assert(check_things());
}
```

- After we are satisfied that the program is correctly implemented, we can pass `-DNDEBUG` to the compiler, and skip all assertions.

- `assert(condition)` aborts if `condition` is false
- Used for non-trivial checks in code during development. The errors we are trying to catch are logic errors in implementation.
- If the macro `NDEBUG` is defined before including `<cassert>` `assert(condition)` reduces to nothing

Exercise 2.1:

The program `examples/exception.cc` demonstrates the use of exceptions. Rewrite the loop so that the user is asked for a new value until a reasonable value for the function input parameter is given.

Exercise 2.2:

Handle invalid inputs in your `gcd.cc` program so that if we call it as `gcd apple orange` it quits with an understandable error message. Valid inputs should produce the result as before.

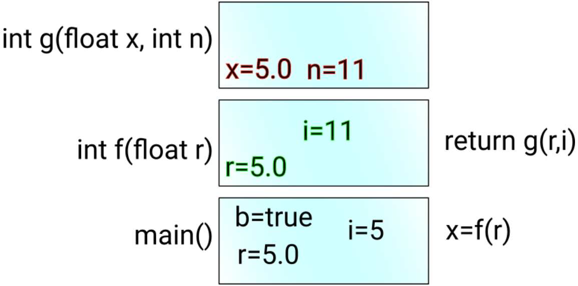
Exercise 2.3:

In the folder `examples/sqrt_error_handling`, you will find the solution to the square root exercise from the first day, with different error handling methods discussed here: exceptions, `std::optional` and `std::expected`. Study the code, experiment, ask for clarifications!

Dynamic memory management

HEAP VS STACK

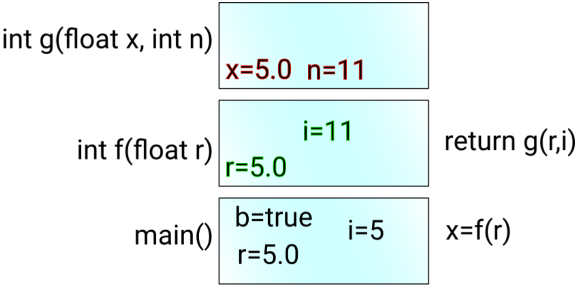
```
auto f(double x) -> double
{
  int i = static_cast<int>( x );
  double M[1000][1000][1000]; // Oops!
  M[123][344][24] = x;
  return x - M[i][555][1];
}
auto main() -> int
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```



- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits

HEAP VS STACK

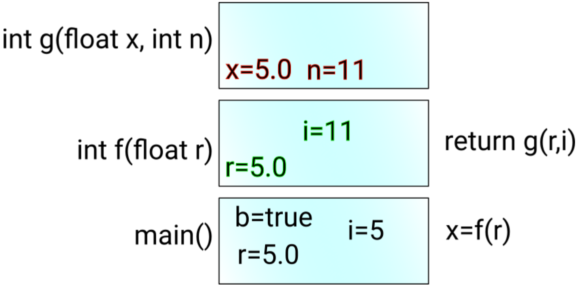
```cpp
auto f(double x) -> double
{
  int i = static_cast<int>( x );
  double M[1000][1000][1000]; // Oops!
  M[123][344][24] = x;
  return x - M[i][555][1];
}
auto main() -> int
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```



- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits
- We do not know how much space we should reserve for a variable (e.g. a `string`)

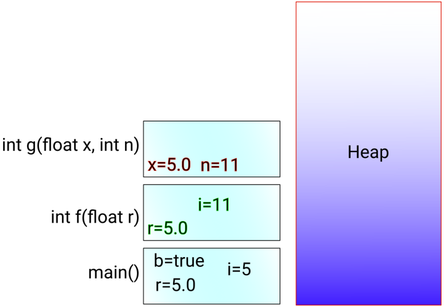
HEAP VS STACK

```
auto f(double x) -> double
{
  int i = static_cast<int>( x );
  double M[1000][1000][1000]; // Oops!
  M[123][344][24] = x;
  return x - M[i][555][1];
}
auto main() -> int
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```



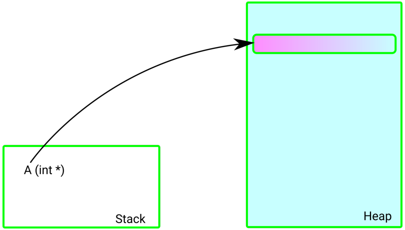
- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits
- We do not know how much space we should reserve for a variable (e.g. a `string`)
- We need a way to allocate from the "free store"

HEAP MEMORY



int g(float x, int n)
x=5.0 n=11
int f(float r)
i=11
r=5.0
main()
b=true
i=5
r=5.0
Heap

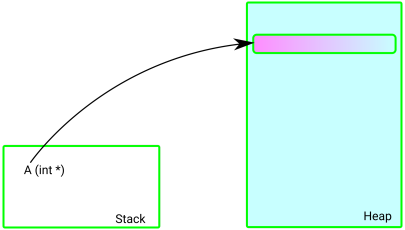
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).

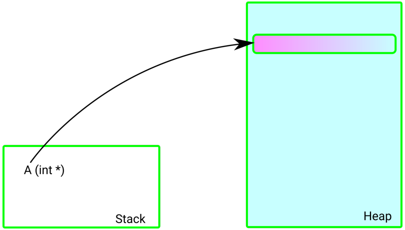
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space

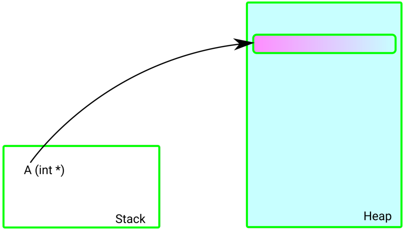
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space
- Memory allocated from the heap stays with your program until you free it, using `delete`

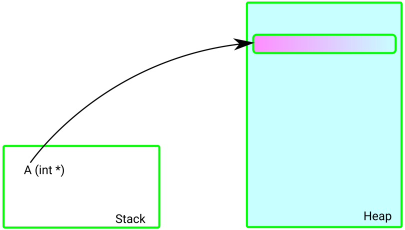
HEAP VS STACK




```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space
- Memory allocated from the heap stays with your program until you free it, using `delete`
- The pointer we used to store its address is subject to scoping rules, and might expire at a certain `}`

HEAP VS STACK

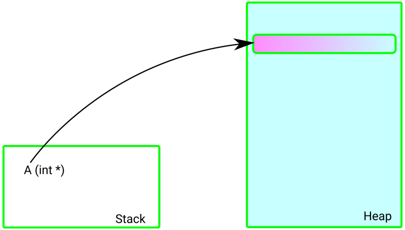



```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  delete [] A;
}
```

Note: Heap allocation and deallocation are slower than those on the stack!

- `operator new`: Request that a specific amount of memory be reserved for you on the free store. The return value of the `new` operation is an address, which you store in a pointer (`A` here).
- The pointer `A` is a normal variable on the stack. But it's value is the address of the allocated space
- Memory allocated from the heap stays with your program until you free it, using `delete`
- The pointer we used to store its address is subject to scoping rules, and might expire at a certain `}`
- Unless you ensure that `delete` is called before the pointer expires or that the address is stored elsewhere before that happens, you have a memory leak

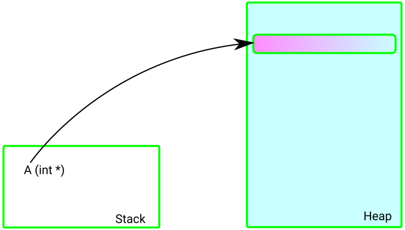
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations

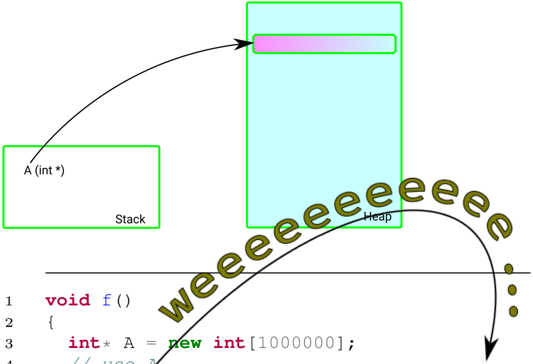
HEAP VS STACK




```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?

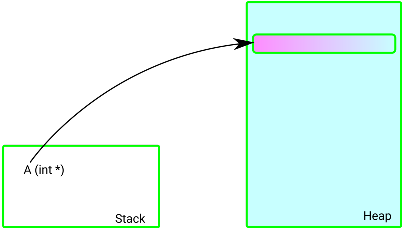
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete` ?
- We may never reach the meticulously placed `delete` statement!

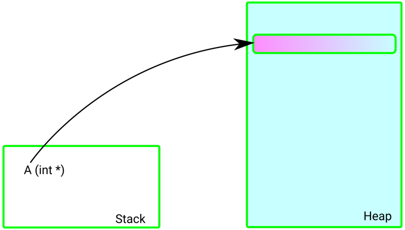
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete` ?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near “exit sites” is not a sound strategy

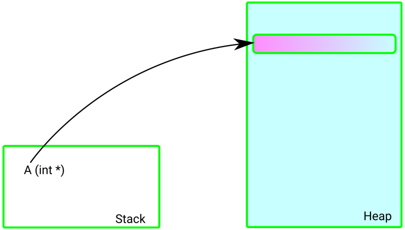
HEAP VS STACK




```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near "exit sites" is not a sound strategy
- This also applies to other "resources" which must be acquired for use, e.g., threads, mutexes...

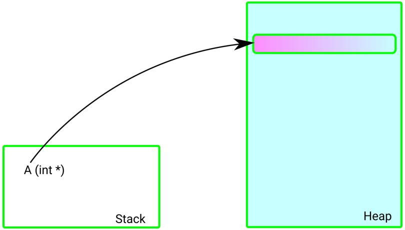
HEAP VS STACK




```
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near "exit sites" is not a sound strategy
- This also applies to other "resources" which must be acquired for use, e.g., threads, mutexes...
- The right way to handle that is by using RAII

HEAP VS STACK




```cpp
void f()
{
  int* A = new int[1000000];
  // use A
  g(A);
  //
  delete [] A;
}
```

- Allocations with `new` should be matched by corresponding `delete` operations
- But, what if we throw an exception before we reach `delete`?
- We may never reach the meticulously placed `delete` statement!
- Generally, because of exceptions, in C++ programs, placing clean-up operations near "exit sites" is not a sound strategy
- This also applies to other "resources" which must be acquired for use, e.g., threads, mutexes...
- The right way to handle that is by using RAII
- Standard library facilities use the same technique

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to ”forget“ to delete the memory owned by `unique_ptr`

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`
- Several instances of `shared_ptr` may refer to the same block of memory. When the last of them expires, it cleans up.

OBJECT LIFETIME MANAGEMENT WITH SMART POINTERS

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`
- Several instances of `shared_ptr` may refer to the same block of memory. When the last of them expires, it cleans up.
- Helper functions `make_unique` and `make_shared` can be used to allocate on heap and retrieve a smart pointer to the allocated memory

DYNAMIC MEMORY WITH SMART POINTERS

```
using big = std::array<int, 1000000>;
int f()
{
  auto u1 = std::make_unique<big>();
  // use u1
} // u1 expires, and frees the allocated memory
```

- Current recommendation: avoid free `new` / `delete` calls in normal user code
- Use them to implement memory management components
- Use `unique_ptr` and `shared_ptr` to manage resources
- You can then assume that an ordinary pointer in your code is a ”non-owning“ pointer, and let it expire without leaking memory

MEMORY ALLOCATION/DEALLOCATION

- You don't need it often:
  - `std::string` takes care of itself
  - Using standard library containers like `vector`, `list`, `map`, `deque` even rather complicated structures can be created without explicit memory allocation and de-allocation.
- When you nevertheless must (first choice):

```cpp
auto c = make_unique<complex_number>(1.2,4.2); // on the heap
int asize=100; // on the stack
auto darray = make_unique<double[]>(asize);
// The stack frame contains the unique_ptr variables c and darray.
// The memory locations they point to on the other hand, are not
// on the stack, but on the heap. But, you don't need to worry about
// releasing that memory explicitly. If you don't have any way of
// accessing the resource (the pointers expire), the memory will be
// freed for you.
//
```

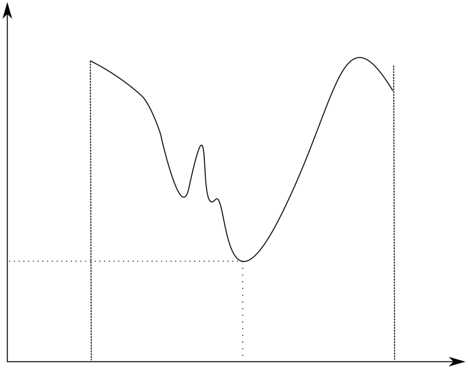
MEMORY ALLOCATION/DEALLOCATION

- You don't need it often:
  - `std::string` takes care of itself
  - Using standard library containers like `vector`, `list`, `map`, `deque` even rather complicated structures can be created without explicit memory allocation and de-allocation.
- When you nevertheless must (second choice):
  - Wrap the managed resource in a class
  - Allocate in constructors, using `new`, if you must
  - Clean up in the destructor, using `delete`, if you used `new`
  - Object lifetime rules will ensure the clean up, even in case of exceptions.

C++ classes

C++ classes

AD HOC STRUCTS



- Some times calculations involve bundles of entities which belong together, e.g., the location of a minimum of a function and the corresponding minimum value

AD HOC STRUCTS

```
struct minimize_return_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minimize_return_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto sol = minimize(0., 10., cost_func);
    cout << "Minimum found at " << sol.min_loc
         << "with a value " << sol.min_val
         << "\n";
}
```

- `struct` : Staple together objects of arbitrary types
- Can be done in global as well as block scope

AD HOC STRUCTS

```cpp
struct minim_ret_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minim_ret_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto [loc, val] = minimize(0., 10.,
                               cost_func);
    cout << "Minimum found at " << loc
         << "with a value " << val << "\n";
}
```

- `struct` : Staple together objects of arbitrary types
- Can be done in global as well as block scope
- We can now use the name of the `struct` to create variables , such that each of them has a `min_loc` *member* and a `min_val` *member*
- Can be function argument (and hence can participate in overload resolution), or return value (and hence gives us a way to return multiple values)
- Names of the components can be chosen to reflect any meanings associated with the content, making it a good practical way of returning multiple objects from a function

AD HOC STRUCTS

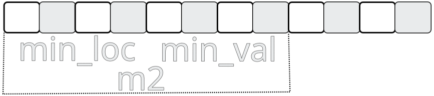
```
struct minim_ret_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minim_ret_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto [loc, val] = minimize(0., 10.,
                               cost_func);
    cout << "Minimum found at " << loc
         << "with a value " << val << "\n";
}
```

- `struct` : Staple together objects of arbitrary types
- Can be done in global as well as block scope
- We can now use the name of the `struct` to create variables , such that each of them has a `min_loc` *member* and a `min_val` *member*
- Can be function argument (and hence can participate in overload resolution), or return value (and hence gives us a way to return multiple values)
- Names of the components can be chosen to reflect any meanings associated with the content, making it a good practical way of returning multiple objects from a function
- Structured bindings can be used to create aliases for the components. The binding names are independent of the names in the `struct`

AD HOC STRUCTS

```
struct minimize_return_type {
    double min_loc, min_val;
};
auto minimize(double r1, double r2,
              FunctionType f)
{
    minimize_return_type m;
    // Find minimum somehow
    m.min_loc = the_location;
    m.min_val = the_value;
    return m;
}
void elsewhere()
{
    auto m1 = minimize(-10., 0., constfunc1);
    minim_ret_type m2 = minimize(-10., 0.,
                                 constfunc1);
    auto * mptr = &m2;
    if ( m1.min_val > mptr->min_val )
        haha();
}
```

- A `struct` is a user defined data type
- Each *instance* has a bundle, with a `min_loc` and `min_val` member
- Members are accessed from the object using the `.` notation, and from a pointer to an object using the `->` notation. `(*mptr).min_val` is the same as `mptr->min_val`



DESIGNATED INITIALISERS

```
// examples/desig2.cc
struct v3 { double x, y, z; };
struct pars { int offset; v3 velocity; };
auto operator<<(std::ostream& os, const v3& v) -> std::ostream&
{
    return os << v.x << ", " << v.y << ", " << v.z << " ";
}
auto example_func(pars p)
{
    std::cout << p.offset << " with velocity " << p.velocity << "\n";
}
auto main() -> int
{
    example_func( {.offset = 5, .velocity = {.x=1., .y = 2., .z=3.}} );
}
```

- Simple struct type objects can be initialized by designated initialisers for each field.
- Can be used to implement a kind of "keyword arguments" for functions. But remember, at least in C++20, the field order can not be shuffled.

C++ CLASSES

```
1  // examples/trivialclassoverload.cc
2  class A {};
3  class B {};
4  void func(int i, A a)
5  {
6      cout << "Called f input types (int, A)\n";
7  }
8  void func(int i, B b)
9  {
10     cout << "Called f input types (int, B)\n";
11 }
12 auto main() -> int
13 {
14     A xa;
15     B xb;
16     func(0, xa) ;
17     func(0, xb) ;
18 }
```

- User defined data types. Independently created classes are different, even if they have the same content.
- Function overloading: The two versions of the function `func` shown here are different entities from the compiler's viewpoint. No ambiguity about which function is called in lines 16 and 17 in `main()` .

C++ CLASSES

Overloading operators

```cpp
// examples/op_overload.cc
class A {};
class B {};
auto operator+(A x, A y) -> A
{
    std::cout << "operator+(A, A)\n";
    return x;
}
auto operator+(B x, B y) -> B
{
    std::cout << "operator+(B, B)\n";
    return x;
}
auto operator+(A x, B y) -> A {...} // similar
auto main() -> int {
    A a1, a2;
    B b1, b2;
    a1 + a2;
    a1 + b1;
    b1 + b2; // b1 + a2; doesn't work. Think why!
}
```

- For C++ class types, operators like `+`, `-`, `*`, `/`, `||`, `&&` ... are functions
- As long as at least one of the arguments to an operator is of a class type (not a built-in type like `int`, `double` ...), it is possible to provide a recipe to interpret expressions like `a1 + a2`
- `a1 + a2` is interpreted as a function call `operator+(a1, a2)`
- Using suitably chosen operators to overload, we can make expressions involving objects of a class type more intuitive

OVERLOADING OPERATORS

<table>
<tr><td>+</td><td>−</td><td>*</td><td>/</td><td>%</td><td>&</td><td>ˆ</td><td>|</td><td></td></tr>
<tr><td>+=</td><td>−=</td><td>*=</td><td>/=</td><td>%=</td><td>&=</td><td>ˆ=</td><td>|=</td><td>=</td></tr>
<tr><td>++</td><td>−−</td><td>&&</td><td>||</td><td>!</td><td>!=</td><td>==</td><td></td><td></td></tr>
<tr><td><</td><td>></td><td>!=</td><td>==</td><td><=</td><td>>=</td><td><=></td><td>=</td><td>=</td></tr>
<tr><td>()</td><td>[]</td><td>,</td><td>-></td><td>->*</td><td><<</td><td><<=</td><td>>>=</td><td>>></td></tr>
<tr><td>new</td><td>delete</td><td>new[]</td><td>delete[]</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Table: List of operators you can overload. (But remember, *can* and *should* are not the same thing!)

- Think carefully about the impact an overloaded operator will have on the readability of your code. Whether or not it the impact is beneficial depends on the use case
- Many important commonly used C++ features depend on suitably overloaded operators. E.g., `std::cout << "Hello\n";`

C++ CLASSES

```cpp
struct Vector3 {
  double x, y, z;
};
```

- Usually, encapsulates some data to represent an idea

C++ CLASSES

```cpp
struct Vector3 {
    double x, y, z;
    auto mag2() -> double
    {
        return x * x + y * y + z * z;
    }
};
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data

C++ CLASSES

```cpp
struct Vector3 {
    double x, y, z;
    auto mag2() -> double
    {
        return x * x + y * y + z * z;
    }
};

void somefunc()
{
    int a, b, c;
    Vector3 d, e, f;
    // ...
    if (d.mag2() < e.mag2()) doX();
}
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data
- Once defined, one can create and use variables of the new type

C++ CLASSES

```cpp
struct Vector3 {
    double x, y, z;
    auto mag2() -> double
    {
        return x * x + y * y + z * z;
    }
};

void somefunc()
{
    int a, b, c; // On the stack
    Vector3 d, e, f; // On the stack
    // ...
    if (d.mag2() < e.mag2()) doX();
}
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data
- Once defined, one can create and use variables of the new type

In C++, objects of user defined types live on the stack by default, unless explicitly created on the heap.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```cpp
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a. modulus();// 1 * 1 + 2 * 2
auto d = b. modulus();// 3 * 3 + 4 * 4
auto e = cptr-> modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```cpp
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.
- Inside a member function, member variables correspond to the invoking instance.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
struct complex {
    double real, imaginary;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imaginary * imaginary);
    }
};
...
complex a{1, 2}, b{3, 4};
complex* cptr = &a;
auto c = a.modulus();// 1 * 1 + 2 * 2
auto d = b.modulus();// 3 * 3 + 4 * 4
auto e = cptr->modulus();// 1 * 1 + 2 * 2
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.
- Inside a member function, member variables correspond to the invoking instance.
- Think of a call like `a.modulus()` as `complex::modulus(a)` The address of the object on the left of the "." is the implicit first argument to the member function.

C++ source #1

```cpp
struct Example {
    double x, y;
    auto mod() -> double;
};

auto Example::mod() -> double
{
    return x * x + y * y;
}

auto unrelated(Example* ptr) -> double
{
    return ptr->x * ptr-> x + ptr->y * ptr -> y;
}
```

x86-64 clang 12.0.0 (Editor #1, Compiler #1) C++ — -O3

```asm
Example::mod():                         # @Example::mod()
        movupd  xmm1, xmmword ptr [rdi]
        mulpd   xmm1, xmm1
        movapd  xmm0, xmm1
        unpckhpd        xmm0, xmm1                      # x
        addsd   xmm0, xmm1
        ret
unrelated(Example*):                    # @unrelated(Example*
        movupd  xmm1, xmmword ptr [rdi]
        mulpd   xmm1, xmm1
        movapd  xmm0, xmm1
        unpckhpd        xmm0, xmm1                      # x
        addsd   xmm0, xmm1
        ret
```

Output (0/0) x86-64 clang 12.0.0 - cached (17817B) ~333 lines filtered

OPERATORS AS MEMBER FUNCTIONS

```cpp
struct complex {
    double real, imag;
    auto modulus() -> double
    {
        return sqrt(real * real +
                    imag * imag);
    }
    auto operator+(complex other) -> complex
    {
        return {real + other.real,
            imag + other.imag};
    }
};
```

- Since operators working with class types are normal functions, one can have operators as member functions
- The implicit argument (invoking instance) is on the left hand side for binary operators. That's why the binary operator `+` is defined here as a member function taking only one argument

MEMBER FUNCTIONS AND CONST

```
struct complex {
    double m_real, m_imag;
    auto modulus() -> double;
    auto operator-( const complex& b) -> complex;
};

void somewhere_else()
{
    complex z1, z2;
    auto z3 = z1 - z2;
    // We know z2 didn't change.
    // But did z1 ?
}
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function

MEMBER FUNCTIONS AND CONST

```cpp
struct complex {
    double m_real, m_imag;
    auto modulus() -> double;
    auto operator-( const complex& b) -> complex;
};

void somewhere_else()
{
    complex z1, z2;
    auto z3 = z1 - z2;
    // We know z2 didn't change.
    // But did z1 ?
}
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function
- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance.
- But as that is implicit, where do we put a `const` qualifier, if we want to express that the calling instance must not change ?

MEMBER FUNCTIONS AND CONST

```
struct complex {
    double m_real, m_imag;
    auto modulus() const -> double;
    auto operator-( const complex& b) const
        -> complex;
};

void somewhere_else()
{
    complex z1, z2;
    auto z3 = z1 - z2;
    // We know z2 didn't change.
    // We know z1 didn't change,
    // as we called a const member
}
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function
- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance.
- But as that is implicit, where do we put a `const` qualifier, if we want to express that the calling instance must not change ?
- Answer: After the closing parentheses of the function signature.

SOME EXAMPLE CLASSES

```
class Angle {
    double rd = 0;
public:
    enum unit {
        radian,
        degree
    };
    Angle operator-(Angle a) const ;
    Angle operator+(Angle a) const ;
    Angle operator-=(Angle a) ;
```

```
class Vector3
{
public:
    enum crdtype {cartesian=0,polar=1};
    inline auto x() const -> double {return dx;}
    inline void x(double gx) {dx=gx;}
    auto dot(const Vector3 &p) const -> double;
    Vector3 cross(const Vector3 &p) const;
```

```
class IsingLattice {
public:
    using update_type =
        std::pair<size_t, size_t>;
    IsingLattice();
    IsingLattice(size_t Nx, double JJ);
    void setLatticeSize(size_t ns);
```

```
class KMer {
public:
    Nucleotide at(size_t i);
    auto operator==(const KMer &) const -> bool;
```

```
class SimulationManager {
public:
    void loadSettings(std::string file);
    auto checkConfig() const -> bool;
    void start();
```

OBJECT INITIALISATION: CONSTRUCTORS

- In C++, initialisation functions for a struct have the same name as the struct. They are called *constructors*.

```cpp
struct complex {
    complex(double re, double im)
    {
        real = re;
        imaginary = im;
    }
};
```

- Alternative syntax to initialise variables in constructors

```cpp
struct complex
{
    complex(double re, double im) : real{re}, imaginary{im} {}
};
```

- A class can have as many constructors as it needs.

CONSTRUCTORS

```
struct complex
{
    complex(double re, double im)
    {
      real = re;
      imaginary = im;
    }
    complex()
    {
      real = imaginary = 0;
    }
    double real, imaginary;
};
...
complex a(3.2, 9.3);
// C++11 and older
complex b{4.3, 1.9};// since C++11
```

- Constructors may be (and normally are) overloaded.
- When a variable is declared, a constructor with the appropriate number of arguments is implicitly called
- The default constructor is the one without any arguments. That is the one invoked when no arguments are given while creating the object.

CONSTRUCTORS

```cpp
struct complex
{
    complex(double re, double im)
    {
        real = re;
        imaginary = im;
    }
    complex() {}
    double real{0.};
    double imaginary{0.};
};
...
complex a(4.3, 23.09), b;
```

- Member variables can be initialised to ”default values“ at the point of declaration

CONSTRUCTORS

```
struct complex
{
    complex(double re, double im)
    {
        real = re;
        imaginary = im;
    }
    complex() {}
    double real{0.};
    double imaginary{0.};
};
...
complex a(4.3, 23.09), b;
```

- Member variables can be initialised to ”default values“ at the point of declaration
- Member variables not touched by the constructor stay at their default values

CONSTRUCTORS

```
struct complex
{
    complex(double re, double im)
      : real{re}, imaginary{im}
    {
    }
    complex() {}
    double real{0.};
    double imaginary{0.};
};
...
complex a(4.3, 23.09), b;
```

- Member variables can be initialised to ”default values“ at the point of declaration
- Member variables not touched by the constructor stay at their default values
- Preferred syntax for initialisation of members in a constructor is shown here . This form of initialisation outside the constructor function body is only possible for constructors

FREEING MEMORY FOR USER DEFINED TYPES

```cpp
struct darray
{
    double *data=nullptr;
    size_t sz=0;
    darray(size_t N) : sz{N} {
        data = new double[sz];
    }
};

auto tempfunc(double phasediff) -> double
{
    // find number of elements
    darray A{large_number};
    // do some great calculations
    return answer;
}
```

What happens to the memory ? The struct `darray` has a pointer member, which points to dynamically allocated memory

- When the life of the variable `A` ends, the member variables (e.g. the pointer `data`) go out of scope.
- How does one free the dynamically allocated memory attached to the member `data` ?

FREEING MEMORY FOR USER DEFINED TYPES

```
struct darray
{
    double *data{nullptr};
    size_t sz{0};
    darray(size_t N) : sz{N} {
        data = new double[sz];
    }
    ~darray() {
        if (data) delete [] data;
    }
};

auto tempfunc(double phasediff) -> double
{
    // find number of elements
    darray A{large_number};
    // do some great calculations
    return answer;
}
```

For any class which explicitly allocates dynamic memory

- We need a function that cleans up all explicitly allocated memory in use, so that we call it for every object whose lifetime is about to end.
- In C++, such functions are called destructors, and have the name ~ followed by the class name.
- Destructors take no arguments, and there is exactly one for each class
- The destructor is automatically called when a variable expires. You don't call it explicitly. It is always called whenever the scope of an object ends! It is impossible to forget.

DESTRUCTORS

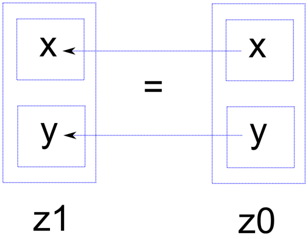
```cpp
class A {
    A()  {}
    ~A()  {}
};
auto demo(A)
{
    A v1 ;
    try {
        A v2 ;
        // calc

    } //  ~A()  for v2
    catch {
        // ...
    }
} //  ~A()  for v1
```

- No matter how you exit a scope, if the scope of a variable ends, its destructor is invoked automatically
- What if we acquire resources in constructors and clean up in the destructor? It would be impossible to forget to free resources when we are done!

COPYING AND ASSIGNMENTS

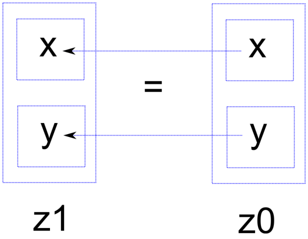
```
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members

COPYING AND ASSIGNMENTS

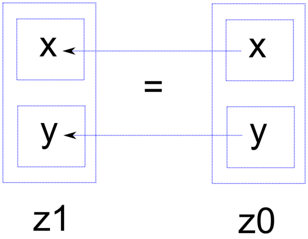
```cpp
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations

COPYING AND ASSIGNMENTS

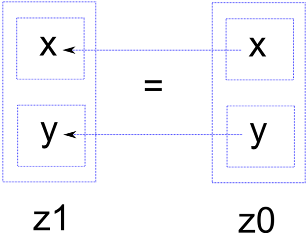
```
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class

COPYING AND ASSIGNMENTS

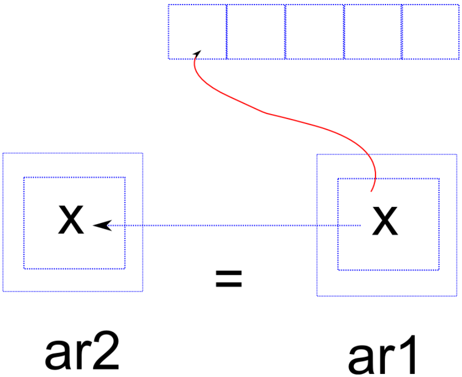
```cpp
struct complex
{
    double x, y;
};
//...
complex z0{2.0, 3.0}, z1;
z1 = z0; // assignment operator
complex z2{z0}; //copy constructor
```



- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class
- Why would you want to ?

COPYING AND ASSIGNMENTS

```
class darray {
    double *x;
};
darray::darray(unsigned n)
{
    x=new double[n];
}
void foo()
{
    darray ar1(5);
    darray ar2{ar1}; //copy constructor
    ar2[3] = 2.1;
    //oops! ar1[3] is also 2.1 now!
} //trouble
```

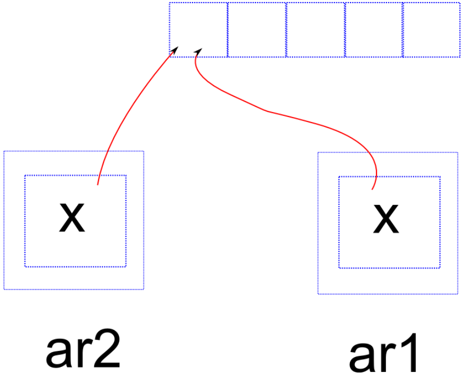


Copying pointers with dynamically allocated memory

- May not be what we want

COPYING AND ASSIGNMENTS

```cpp
class darray {
    double *x;
};
darray::darray(unsigned n)
{
    x=new double[n];
}
void foo()
{
    darray ar1(5);
    darray ar2{ar1}; //copy constructor
    ar2[3] = 2.1;
    //oops! ar1[3] is also 2.1 now!
} //trouble
```



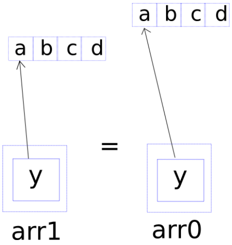
Copying pointers with dynamically allocated memory

- May not be what we want
- Leads to "double free" errors when the objects are destroyed

COPYING AND ASSIGNMENTS

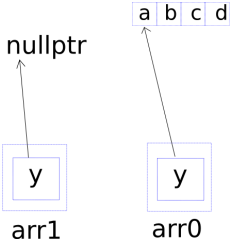
```cpp
class darray {
    double *x{nullptr};
    unsigned int len{};
public:
    // Copy constructor
    darray(const darray &);
    //assignment operator
    auto operator=(const darray&) -> darray&;
};
darray::darray(const darray& other)
{
    if (other.len!=0) {
        len = other.len;
        x = new double[len];
        for (unsigned i = 0; i < len; ++i) {
            x[i] = other.x[i];
        }
    }
}
auto darray::operator=(const darray& other) -> darray&
{
    if (this != &other) {
        if (len != other.len) {
```

```cpp
            len = other.len;
            if (x) delete [] x;
            x = new double[len];
        }
        for (unsigned i = 0; i < len; ++i) {
            x[i] = other.x[i];
        }
    }
    return *this;
}
```



MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

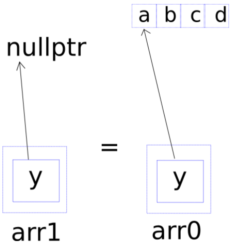
```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

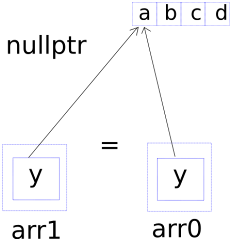
```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

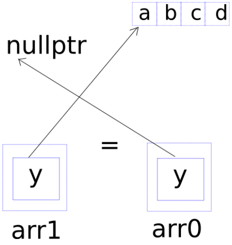
```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

```cpp
class darray {
    darray(darray &&); //Move constructor
    auto operator=(darray &&) -> darray&;
    //Move assignment operator
};
darray::darray(darray&& other)
{
    len = other.len;
    x = other.x;
    other.x = nullptr;
}
auto darray::operator=(darray&& other)
    -> darray& {
    len = other.len;
    x = other.x;
    other.x = nullptr;
    return *this;
}
darray d1(3);
init_array(d1); //d1 = {1.0,2.0,3.0}
darray d2{d1}; //Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS
- Put disposable content in RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly. Results of the evaluation of expressions might create a nameless object containing the resultant value (*prvalue*: pure r-value). A function may be returning a named entity which is about to expire (*xvalue*: expiring value) References to such objects are called R-value references. A move constructor or assignment operator is automatically invoked if constructor argument is an R-value reference

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly. Results of the evaluation of expressions might create a nameless object containing the resultant value (*prvalue*: pure r-value). A function may be returning a named entity which is about to expire (*xvalue*: expiring value) References to such objects are called R-value references. A move constructor or assignment operator is automatically invoked if constructor argument is an R-value reference
- You can invoke the move constructor by casting the function argument to an R-value reference, e.g. `darray d3{std::move(d1)}`

BIG FIVE (OR ZERO)

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator

- How many of these do you have to write for each and every class you make ?

BIG FIVE (OR ZERO)

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator

- How many of these do you have to write for each and every class you make ?
- Answer: **None!** If you don't have bare pointers in your class, and don't want anything fancy happening, the compiler will auto-generate reasonable defaults. "Rule of zero"

BIG FIVE

```
class darray {
public:
    darray(double x, double y) : re{x}, im{y} {}
    darray() = default;
    darray(const darray &) = default;
    darray(darray &&) = default;
    auto operator=(const darray&) -> darray& = default;
    auto operator=(darray&&) -> darray& = default;
};
```

- If you have to write any constructor yourself, auto-generation of the default constructor is disabled

BIG FIVE

```
class darray {
public:
    darray(double x, double y) : re{x}, im{y} {}
    darray() = default;
    darray(const darray &) = default;
    darray(darray &&) = default;
    auto operator=(const darray&) -> darray& = default;
    auto operator=(darray&&) -> darray& = default;
};
```

- If you have to write any constructor yourself, auto-generation of the default constructor is disabled
- But you can request default versions of the any of these functions as shown

BIG FIVE

```cpp
class darray {
    darray() = delete;
    darray(const darray &) = delete;
    darray(darray &&) = default;
    auto operator=(const darray &) -> darray& = delete;
    auto operator=(darray &&) -> darray& = default;
};
```

- You can also explicitly request that one or more of these are not auto-generated
- In the example shown here, it will not be possible to copy objects of the class, but they can be moved

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management

```
auto operator=(const darray& oth) -> darray& {
  if (this!=&oth) {
    if (arr && sz!=oth.sz) {
      sz=oth.sz;
      delete [] arr;
      arr=new T[sz];
    }
    for (size_t i=0;i<sz;++i)
      arr[i]=oth.arr[i];
  }
  return *this;
}
auto operator=(darray&& oth) -> darray& {
  swap(oth);
  return *this;
}
```

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference

```
auto operator=(darray d) -> darray& {
  swap(d);
  return *this;
}
// No further move assignment operator!
```

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference
- Use the class member function `swap` to swap the data with the newly created copy

```
auto operator=(darray d) -> darray& {
  swap(d);
  return *this;
}
// No further move assignment operator!
```

- Neat trick that works in most cases
- Reduces the big five to big four

JÜLICH Forschungszentrum

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the modulus and argument, instead of the real and imaginary parts.

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```cpp
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the modulus and argument, instead of the real and imaginary parts.
- We have to change a lot of code.

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
auto foo(complex a, int p, truck c) -> int
{
    complex z1, z2, z3 = a;
...
    z1 = z1.argument() * z2.modulus() * z3.conjugate();
    c.start(z1.imaginary * p);
}
```

Imagine that ...

- External code calling only member functions to access member data can survive
- Direct use of member variables while using a class is often messy, the implementer of the class then loses the freedom to change internal organisation of the class for efficiency or other reasons

C++ CLASSES

```cpp
class complex
{
public:
    complex(double re, double im)
    : m_real(re), m_imag(im) {}
    complex() = default;
    auto real() const -> double { return m_real; }
    auto imag() const -> double { return m_imag; }
...
private:
    double m_real = 0., m_imag = 0.;
};
```

struct $\Longrightarrow$ members public by default

class $\Longrightarrow$ members private by default

- Members declared under the keyword `private` can not be accessed from outside
- Public members (data or function) can be accessed
- Provide a consistent and useful interface through public functions
- Keep data members hidden
- Make accessor functions `const` when possible

Exercise 2.4:

The program `examples/complex_number_class.cc` contains a version of the complex number class, with all syntax elements we discussed in the class. It is heavily commented with explanations for every subsection. Please read it to revise all the syntax relating to classes. Write a `main` program to use and test the class.

CONSTRUCTOR/DESTRUCTOR CALLS

Exercise 2.5:

The file `examples/verbose_ctordtor.cc` demonstrates the automatic calls to constructors and destructors. The simple class `Vbose` has one `string` member. All its constructors and destructors print messages to the screen when they are called. The `main()` function creates and uses some objects of this class. Follow the messages printed on the screen and link them to the statements in the program. Does it make sense (i) When the copy constructor is called ? (ii) When is the move constructor invoked ? (iii) When the objects are destroyed ?

Suggested reading: `http://www.informit.com/articles/printerfriendly/2216986`

Exercise 2.6:

The program `examples/onexcept.cc` shows the behaviour of constructor/destructor calls when an exception is called. Observe that exiting a function via an exception is also leaving the scope, and therefore invokes the destructor.

MAKING STD::COUT RECOGNIZE CLASS

Teaching cout how to print your type: overload operator `<<`

```cpp
auto operator<<(std::ostream& os, const complex& a) -> std::ostream&
{
    os << a.real();
    if (a.imag() < 0) os << a.imag() << " i ";
    // If imag() is negative, it already has a - sign
    else os << " +" << a.imag() << " i ";
    return os;
}
complex a;
...
std::cout << "The roots are " << a << " and " << a.conjugate() << '\n';
```

AND SIMILARLY FOR STD::CIN

```
auto operator>>(std::istream& is, complex& a) -> std::istream&
{
    double x, y;
    is >> x >> y;
    a.set_real(x);
    a.set_imag(y);
    return is;
}
```

- It is up to you to decide IO operations for your classes
- The stream parameters can not be `const`, because by reading from or writing to the stream, we change its state

PRACTISE: WRITE A DATA ROW CLASS

Exercise 2.7:

You now have all the ingredients to write a data row class. A tabular data file has 5 columns. The first two are integers, the rest are doubles. Let's call the columns id, cat, x, y, and z, respectively. Make sure that there are IO stream overloads for the reading and writing objects of that type. Demonstrate by reading a suitable datafile "multicolumn.dat", and storing the rows in a vector of your `DataRow` type. You should then be able to sort the vector according to any of the data columns.

JÜLICH Forschungszentrum

DATATYPES

Type	Bits	Value
Float	0100 0000 0100 1001 0000 1111 1101 1011	3.1415927
Int	0100 0000 0100 1001 0000 1111 1101 1011	1078530011

- Same bits, different rules $\Longrightarrow$ different type

From arbitrary collection of members to a new "data type"

```
class Date {
    int m_day, m_month, m_year;
public:
    static auto today() -> Date;
    auto operator+(int n) const -> Date;
    auto operator-(int n) const -> Date;
    auto operator-(const Date &) const -> int;
};
```

- Make sure every way to create an object results in a valid state
- Provide only those operations on the data which keep the essential properties intact

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
    - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
  - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
  - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
  - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
  - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1
- Using `private` data members and well designed `public` interfaces, we can ensure that assumptions behind an idea are always true.

CLASS INVARIANTS

```
class darray {
private:
    double * dataptr = nullptr;
    size_t sz = 0;
public:
// initialize with N elements
    darray(size_t N);
    ~darray();
// resize to N elements
    void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants

CLASS INVARIANTS

```
class darray {
private:
    double * dataptr = nullptr;
    size_t sz = 0;
public:
// initialize with N elements
    darray(size_t N);
    ~darray();
// resize to N elements
    void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants
- Maintain Invariants in every member

CLASS INVARIANTS

```
class darray {
private:
    double * dataptr = nullptr;
    size_t sz = 0;
public:
// initialize with N elements
    darray(size_t N);
    ~darray();
// resize to N elements
    void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants
- Maintain Invariants in every member
- $\rightarrow$ a structure which always has sensible values

STATIC MEMBERS

```cpp
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.

STATIC MEMBERS

```
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application

STATIC MEMBERS

```
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

STATIC MEMBERS

```cpp
class Triangle {
public:
    static unsigned counter;
    Triangle() : ...
    {
        ++counter;
    }
    ~Triangle() { --counter; }
    static auto instanceCount() -> unsigned
    {
        return counter;
    }
};
... Triangle.cc ...
unsigned Triangle::counter = 0;
```

- Static member functions do not have an implicit `this` pointer argument. They can be invoked as `ClassName::function()`.

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

SOME FUN: OVERLOADING THE () OPERATOR

```cpp
class swave
{
private:
  double a = 1.0, omega = 1.0;
public:
  swave() = default;
  swave(double x, double w) :
     a{x}, omega{w} {}
  auto operator()(double t) const -> double
  {
    return a * sin(omega * t);
  }
};
```

```cpp
const double pi = acos(-1);

int N = 100;
swave f{2.0, 0.4};
swave g{2.3, 1.2};

for (int i = 0; i < N; ++i) {
  double ar = 2 * i * pi / N;
  std::cout << i << " " << f(ar)
            << " " << g(ar)
            << '\n';
}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator

SOME FUN: OVERLOADING THE () OPERATOR

```cpp
class swave
{
private:
  double a = 1.0, omega = 1.0;
public:
  swave() = default;
  swave(double x, double w) :
     a{x}, omega{w} {}
  auto operator()(double t) const -> double
  {
    return a * sin(omega * t);
  }
};
```

```cpp
const double pi = acos(-1);

int N = 100;
swave f{2.0, 0.4};
swave g{2.3, 1.2};

for (int i = 0; i < N; ++i) {
  double ar = 2 * i * pi / N;
  std::cout << i << " " << f(ar)
            << " " << g(ar)
            << '\n';
}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator
- If they return a `bool` value, they are called predicates

FUNCTIONALS

Using function like objects

- They are like other variables. But they can be used as if they were functions!
- You can make vectors or lists of functionals, pass them as arguments ...
- Although you can run any recipe you want by overloading an operator, most operators are limited to one or two arguments. `()` can take as many as you need. This also contributes to functionals looking like functions when in use.

WRITE YOUR OWN FUNCTIONAL!

Exercise 2.8:

Write a functional class where the return value of f(x) is given by a user specified piece-wise continuous linear function. You should write a class PieceWise. It should have a function to read a vector of x_i,y_i values from a file. Sort them according to x values. Then implement an `operator()` function, so that when you write

```cpp
PieceWise f;
f.read_file("somefile.dat");
auto y = f(x);
```

you get the correct piecewise linear function evaluated. Use the standard library function `std::lerp` to perform the linear interpolation.

OVERLOADING OTHER OPERATORS FOR EXPRESSIVE CODE

```cpp
// examples/collect.cc
class collect {
    std::vector<int> dat;
public:
    auto operator|(int i) -> collect&
    {
        dat.push_back(i);
        return *this;
    }
    auto operator~() const noexcept -> decltype(dat)
    {
        return dat;
    }
};
auto main() -> int
{
    auto C = collect{};
    C | 1 | 2 | 3 | 4 ;
    for (auto el : (~C)) {
        std::cout << el << "\n";
    }
}
```

- Operator overloading is not limited to arithmetic and shift operators.
- Sometimes, choosing the right operator to overload can increase the expressiveness of the code

```cpp
args | sv::drop(1) | sv::transform(str)
```

USER DEFINED LITERALS

Redefining the "" operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value

```
auto main() -> int
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    auto U = Temperature{373.0};
    auto T2 = 350_C;
    auto T3 = 900_K;
    complex c = 1+2_i;
    ...
}
```

USER DEFINED LITERALS

Redefining the "" operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value

```cpp
int main()
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    auto U = Temperature(373.0);
    auto T2 = 350_C;
    auto T3 = 900_K;
    complex c = 1+2_i;
    ...
}
```


JÜLICH Forschungszentrum

USER DEFINED LITERALS

Redefining the "" operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value
- It's far cooler to initialise with your own literals!
- Redefine how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

```
int main()
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    auto T2 = 350_C;
    auto T3 = 900_K;
    complex c = 1+2_i;
    ...
}
```

USER DEFINED LITERALS

```
auto operator "" _K(long double d) -> Temperature
{
    return { static_cast<double>(d), Temperature::Unit::K };
}
auto operator "" _C(long double d) -> Temperature
{
    return { static_cast<double>(d), Temperature::Unit::C };
}
```

- Defining your own rules for how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

USER DEFINED LITERALS

Exercise

- The demo program `examples/literals.cc` shows how this is done using a simple "temperature" class
- Make something similar for a `Distance` class!

```
auto main() -> int
{
    double N = 6.023e23;
    auto T2 = 350_C;
    auto T3 = 900_K;
}
```


JÜLICH Forschungszentrum