```rust
//! The nagiosplugin crate provides some basic utilities to make it easier to write nagios checks.

use std::cmp::Ordering;
use std::process;

#[macro_use]
mod macros;

mod helper;
pub use crate::helper::{safe_run, safe_run_with_state};

/// A Resource basically represents a single service if you view it from the perspective of nagios.
/// If you init it without a state it will determine one from the given metrics.
///
/// You can also create a Resource filled with metrics via the *resource!* macro, which is much
/// like the *vec!* macro.
///
/// ```rust
/// # #[macro_use]
/// # extern crate nagiosplugin;
/// # use nagiosplugin::{SimpleMetric, State};
/// # fn main() {
/// let m1 = SimpleMetric::new("test", Some(State::Ok), 12, None, None, None, None);
/// let m2 = SimpleMetric::new("other", None, true, None, None, None, None);
/// let resource = resource![m1, m2];
/// assert_eq!(&resource.to_nagios_string(), "OK | test=12 other=true");
///
/// // Prints "OK | test=12 other=true" and exits with an exit code of 0 in this case
/// resource.print_and_exit();
/// # }
/// ```
pub struct Resource {
    state: Option<State>,
    metrics: Vec<Box<dyn ResourceMetric>>,
    description: Option<String>,
    name: Option<String>,
}

impl Resource {
    /// If state is set to Some(State) then it will always use this instead of determining it from
    /// the given metrics.
    ///
    /// If you want to create a Resource from some metrics with automatic determination of the
    /// state you can use the *resource!* macro.
    pub fn new(state: Option<State>, description: Option<&str>) -> Resource {
        Resource {
            state,
            metrics: Vec::new(),
            description: description.map(|d| d.to_owned()),
            name: None,
        }
    }

    /// Pushes a single ResourceMetric into the resource.
    pub fn push<M>(&mut self, metric: M)
    where
        M: 'static + ResourceMetric,
    {
        self.metrics.push(Box::new(metric))
    }

    /// Returns a slice of the pushed metrics.
    pub fn metrics(&self) -> &[Box<dyn ResourceMetric>] {
        &self.metrics
    }

    /// Manually set the state for this resource. This disabled the automatic state determination
    /// based on the included metrics of this resource.
    pub fn set_state(&mut self, state: State) {
        self.state = Some(state)
    }

    /// Set the name of this resource. Will be included in the final string output.
    pub fn set_name(&mut self, name: &str) {
        self.name = Some(name.to_owned())
    }

    /// Returns a string which nagios understands to determine the service state.
    ///
    /// This function will automatically determine which service state is appropriate based on the
    /// included metrics. If state has been set manually it will always use the manually set state.
    pub fn to_nagios_string(&self) -> String {
        let mut s = String::new();

        if let Some(ref name) = self.name {
            s.push_str(&format!("{} ", name))
```

```
        }

        s.push_str(&self.get_state().to_string());

        if let Some(ref description) = self.description {
            s.push_str(&format!(": {}", description));
        }

        if self.metrics.len() > 0 {
            s.push_str(" |");

            for metric in self.metrics.iter() {
                s.push_str(&format!(" {}", metric.perf_string()));
            }
        }

        s
    }

    /// Will determine a State by the given metrics.
    ///
    /// In case a state is manually set for this resource,
    /// it will return the manually set state instead.
    pub fn get_state(&self) -> State {
        let mut state = State::Unknown;
        if let Some(ref st) = self.state {
            state = st.clone()
        } else {
            for metric in self.metrics.iter() {
                if let Some(st) = metric.state() {
                    if state < st {
                        state = st;
                    }
                }
            }
        }
        state
    }

    /// Get the description of this resource.
    pub fn get_description(&self) -> Option<&String> {
        self.description.as_ref()
    }

    /// Set the description of this resource.
    pub fn set_description(&mut self, description: &str) {
        self.description = Some(description.to_owned());
    }

    /// Will return the exit code of the determined state via Self::state.
    pub fn exit_code(&self) -> i32 {
        self.get_state().exit_code()
    }

    /// Will print Self::to_nagios_string and exit with the exit code from Self::exit_code
    pub fn print_and_exit(&self) {
        println!("{}", self.to_nagios_string());
        process::exit(self.exit_code());
    }
}

impl Default for Resource {
    fn default() -> Self {
        Resource::new(None, None)
    }
}

/// Represents a single metric of a resource. You shouldn't need to implement this by yourself
/// since the crate provided types already implement this.
pub trait ResourceMetric {
    fn perf_string(&self) -> String;
    fn name(&self) -> &str;
    fn state(&self) -> Option<State>;
}

impl<T, O> ResourceMetric for T
where
    O: ToPerfString,
    T: Metric<Output = O> + ToPerfString,
{
    fn perf_string(&self) -> String {
        self.to_perf_string()
    }

    fn name(&self) -> &str {
        self.name()
```

```
    }

    fn state(&self) -> Option<State> {
        self.state()
    }
}

/// Represents a service state from nagios.
#[derive(Clone, Debug, PartialEq)]
pub enum State {
    Ok,
    Warning,
    Critical,
    Unknown,
}

impl State {
    /// Returns the corresponding nagios exit code to signal the service state of self.
    pub fn exit_code(&self) -> i32 {
        match self {
            &State::Ok => 0,
            &State::Warning => 1,
            &State::Critical => 2,
            &State::Unknown => 3,
        }
    }
}

impl ToString for State {
    fn to_string(&self) -> String {
        match self {
            State::Ok => "OK".to_owned(),
            State::Warning => "WARNING".to_owned(),
            State::Critical => "CRITICAL".to_owned(),
            State::Unknown => "UNKNOWN".to_owned(),
        }
    }
}

impl PartialOrd for State {
    fn partial_cmp(&self, other: &State) -> Option<Ordering> {
        let f = |state| match state {
            &State::Unknown => 0,
            &State::Ok => 1,
            &State::Warning => 2,
            &State::Critical => 3,
        };

        f(self).partial_cmp(&f(other))
    }
}

/// The purpose of ToPerfString is only so one can define custom representations of custom types
/// without using the ToString trait so we don't interfere with that.
///
/// Also used internally for generation of the final output.
///
/// It's already implemented for some basic types.
pub trait ToPerfString {
    fn to_perf_string(&self) -> String;
}

impl_to_perf_string_on_to_string!(bool, usize);
impl_to_perf_string_on_to_string!(u8, u16, u32, u64, u128);
impl_to_perf_string_on_to_string!(i8, i16, i32, i64, i128);
impl_to_perf_string_on_to_string!(f32, f64);
impl_to_perf_string_on_to_string!(String);

impl<'a> ToPerfString for &'a str {
    fn to_perf_string(&self) -> String {
        self.to_string()
    }
}

impl<T, O> ToPerfString for T
where
    O: ToPerfString,
    T: Metric<Output = O>,
{
    fn to_perf_string(&self) -> String {
        metric_string!(
            self.name(),
            self.value(),
            self.warning(),
            self.critical(),
            self.min(),
```

```
259                 self.max()
260             )
261         }
262     }
263
264     impl<T> ToPerfString for Option<T>
265     where
266         T: ToPerfString,
267     {
268         fn to_perf_string(&self) -> String {
269             match self {
270                 Some(ref s) => s.to_perf_string(),
271                 None => String::new(),
272             }
273         }
274     }
275
276     /// This trait can be implemented for any kind of metric and will be used to generate the final
277     /// string output for nagios. Calls to the functions should return immediately and not query the
278     /// service every time.
279     pub trait Metric {
280         type Output: ToPerfString;
281
282         fn name(&self) -> &str;
283         fn state(&self) -> Option<State>;
284         fn value(&self) -> Self::Output;
285         fn warning(&self) -> Option<Self::Output>;
286         fn critical(&self) -> Option<Self::Output>;
287         fn min(&self) -> Option<Self::Output>;
288         fn max(&self) -> Option<Self::Output>;
289     }
290
291     /// A PartialOrdMetric is a metric which will automatically calculate the State
292     /// based on the given value and warning and/or critical value.
293     ///
294     /// It doesn't matter if you provide warning or critical or both of none of these. Even though
295     /// you should choose SimpleMetric if you aren't providing any warning or critical value.
296     ///
297     /// The state function of the implemented Metric trait will always be one of: Ok, Warning, Critical
298     ///
299     /// ```rust
300     /// # extern crate nagiosplugin;
301     /// # use nagiosplugin::{Metric, State, PartialOrdMetric};
302     /// let metric = PartialOrdMetric::new("test", 15, Some(15), Some(30), None, None, false);
303     /// assert_eq!(metric.state(), Some(State::Warning));
304     /// assert_eq!(metric.value(), 15);
305     /// ```
306     pub struct PartialOrdMetric<T>
307     where
308         T: PartialOrd + ToPerfString + Clone,
309     {
310         name: String,
311         value: T,
312         warning: Option<T>,
313         critical: Option<T>,
314         min: Option<T>,
315         max: Option<T>,
316         lower_is_critical: bool,
317     }
318
319     impl<T> PartialOrdMetric<T>
320     where
321         T: PartialOrd + ToPerfString + Clone,
322     {
323         /// Creates a new PartialOrdMetric from the given values.
324         ///
325         /// *In debug builds this will panic if you pass incorrect values for warning and critical.*
326         pub fn new(
327             name: &str,
328             value: T,
329             warning: Option<T>,
330             critical: Option<T>,
331             min: Option<T>,
332             max: Option<T>,
333             lower_is_critical: bool,
334         ) -> Self {
335             #[cfg(debug_assertions)]
336             {
337                 if warning.is_some() && critical.is_some() {
338                     let warning = warning.clone().unwrap();
339                     let critical = critical.clone().unwrap();
340
341                     if lower_is_critical && warning < critical {
342                         panic!("lower_is_critical is set to true while warning is lower than critical, this is not
    correct");
343                     } else if !lower_is_critical && warning > critical {
```

```
                    panic!("lower_is_critical is set to false while warning is lower than critical, this is not correct");
                }
            }

            if min.is_some() && max.is_some() {
                let min = min.clone().unwrap();
                let max = max.clone().unwrap();

                assert!(min < max, "minimum value is not smaller than maximum value")
            }
        }

        PartialOrdMetric {
            name: name.to_owned(),
            value: value.clone(),
            warning: warning.map(|w| w.clone()),
            critical: critical.map(|c| c.clone()),
            min: min.map(|m| m.clone()),
            max: max.map(|m| m.clone()),
            lower_is_critical,
        }
    }
}

impl<T> Metric for PartialOrdMetric<T>
where
    T: PartialOrd + ToPerfString + Clone,
{
    type Output = T;

    fn name(&self) -> &str {
        &self.name
    }

    fn state(&self) -> Option<State> {
        if let Some(ref critical) = self.critical {
            if self.lower_is_critical {
                if &self.value <= critical {
                    return Some(State::Critical);
                }
            } else {
                if &self.value >= critical {
                    return Some(State::Critical);
                }
            }
        }

        if let Some(ref warning) = self.warning {
            if self.lower_is_critical {
                if &self.value <= warning {
                    return Some(State::Warning);
                }
            } else {
                if &self.value >= warning {
                    return Some(State::Warning);
                }
            }
        }

        Some(State::Ok)
    }

    fn value(&self) -> <Self as Metric>::Output {
        self.value.clone()
    }

    fn warning(&self) -> Option<<Self as Metric>::Output> {
        self.warning.clone()
    }

    fn critical(&self) -> Option<<Self as Metric>::Output> {
        self.critical.clone()
    }

    fn min(&self) -> Option<<Self as Metric>::Output> {
        self.min.clone()
    }

    fn max(&self) -> Option<<Self as Metric>::Output> {
        self.max.clone()
    }
}

/// Represents a simple metric where no logic is performed. You give some values in and the same
/// get out.
```

```
#[derive(Clone)]
pub struct SimpleMetric<T>
where
    T: ToPerfString + Clone,
{
    name: String,
    state: Option<State>,
    value: T,
    warning: Option<T>,
    critical: Option<T>,
    min: Option<T>,
    max: Option<T>,
}

impl<T> SimpleMetric<T>
where
    T: ToPerfString + Clone,
{
    pub fn new(
        name: &str,
        state: Option<State>,
        value: T,
        warning: Option<T>,
        critical: Option<T>,
        min: Option<T>,
        max: Option<T>,
    ) -> Self {
        SimpleMetric {
            name: name.to_owned(),
            state,
            value,
            warning,
            critical,
            min,
            max,
        }
    }
}

impl<T> Metric for SimpleMetric<T>
where
    T: ToPerfString + Clone,
{
    type Output = T;

    fn name(&self) -> &str {
        &self.name
    }

    fn state(&self) -> Option<State> {
        self.state.clone()
    }

    fn value(&self) -> <Self as Metric>::Output {
        self.value.clone()
    }

    fn warning(&self) -> Option<<Self as Metric>::Output> {
        self.warning.clone()
    }

    fn critical(&self) -> Option<<Self as Metric>::Output> {
        self.critical.clone()
    }

    fn min(&self) -> Option<<Self as Metric>::Output> {
        self.min.clone()
    }

    fn max(&self) -> Option<<Self as Metric>::Output> {
        self.max.clone()
    }
}

#[cfg(test)]
mod tests {
    use super::{Metric, PartialOrdMetric, Resource, SimpleMetric, State};

    #[test]
    fn test_partial_ord_metric() {
        let metric = PartialOrdMetric::new("test", 12, None, None, None, None, false);
        assert_eq!(metric.name(), "test");
        assert_eq!(metric.state(), Some(State::Ok));
        assert_eq!(metric.value(), 12);
        assert_eq!(metric.warning(), None);
        assert_eq!(metric.critical(), None);
```

```
515
516         // Cases with lower_is_critical = false
517
518         let metric = PartialOrdMetric::new("test", 12, Some(15), Some(30), None, None, false);
519         assert_eq!(metric.state(), Some(State::Ok));
520         assert_eq!(metric.value(), 12);
521         assert_eq!(metric.warning(), Some(15));
522         assert_eq!(metric.critical(), Some(30));
523
524         let metric = PartialOrdMetric::new("test", 15, Some(15), Some(30), None, None, false);
525         assert_eq!(metric.state(), Some(State::Warning));
526         assert_eq!(metric.value(), 15);
527
528         let metric = PartialOrdMetric::new("test", 18, Some(15), Some(30), None, None, false);
529         assert_eq!(metric.state(), Some(State::Warning));
530         assert_eq!(metric.value(), 18);
531
532         let metric = PartialOrdMetric::new("test", 30, Some(15), Some(30), None, None, false);
533         assert_eq!(metric.state(), Some(State::Critical));
534         assert_eq!(metric.value(), 30);
535
536         let metric = PartialOrdMetric::new("test", 35, Some(15), Some(30), None, None, false);
537         assert_eq!(metric.state(), Some(State::Critical));
538         assert_eq!(metric.value(), 35);
539
540         // Cases with lower_is_critical = true
541
542         let metric = PartialOrdMetric::new("test", 35, Some(30), Some(15), None, None, true);
543         assert_eq!(metric.state(), Some(State::Ok));
544         assert_eq!(metric.value(), 35);
545         assert_eq!(metric.warning(), Some(30));
546         assert_eq!(metric.critical(), Some(15));
547
548         let metric = PartialOrdMetric::new("test", 30, Some(30), Some(15), None, None, true);
549         assert_eq!(metric.state(), Some(State::Warning));
550         assert_eq!(metric.value(), 30);
551
552         let metric = PartialOrdMetric::new("test", 20, Some(30), Some(15), None, None, true);
553         assert_eq!(metric.state(), Some(State::Warning));
554         assert_eq!(metric.value(), 20);
555
556         let metric = PartialOrdMetric::new("test", 15, Some(30), Some(15), None, None, true);
557         assert_eq!(metric.state(), Some(State::Critical));
558         assert_eq!(metric.value(), 15);
559
560         let metric = PartialOrdMetric::new("test", 10, Some(30), Some(15), None, None, true);
561         assert_eq!(metric.state(), Some(State::Critical));
562         assert_eq!(metric.value(), 10);
563     }
564
565     #[test]
566     fn test_simple_metric() {
567         let metric = SimpleMetric::new("test", Some(State::Ok), 12, None, None, None, None);
568         assert_eq!(metric.state(), Some(State::Ok));
569         assert_eq!(metric.value(), 12);
570         assert_eq!(metric.warning(), None);
571         assert_eq!(metric.critical(), None);
572
573         let metric = SimpleMetric::new(
574             "test",
575             Some(State::Unknown),
576             22,
577             Some(15),
578             Some(30),
579             None,
580             None,
581         );
582         assert_eq!(metric.state(), Some(State::Unknown));
583         assert_eq!(metric.value(), 22);
584         assert_eq!(metric.warning(), Some(15));
585         assert_eq!(metric.critical(), Some(30));
586
587         let metric = SimpleMetric::new("test", Some(State::Ok), "test", None, None, None, None);
588         assert_eq!(metric.state(), Some(State::Ok));
589         assert_eq!(metric.value(), "test");
590         assert_eq!(metric.warning(), None);
591         assert_eq!(metric.critical(), None);
592     }
593
594     #[test]
595     fn test_resource() {
596         let m1 = SimpleMetric::new("test", Some(State::Ok), 12, None, None, None, None);
597         let m2 = SimpleMetric::new("other", None, true, None, None, None, None);
598
599         let resource = resource![m1, m2];
600
```

```
601          assert_eq!(&resource.to_nagios_string(), "OK | test=12 other=true");
602
603          let m1 = SimpleMetric::new("test", Some(State::Ok), 12, Some(14), None, Some(0), None);
604          let m2 = SimpleMetric::new("other", None, true, None, None, None, None);
605
606          let resource = resource![m1, m2];
607
608          assert_eq!(
609              &resource.to_nagios_string(),
610              "OK | test=12;14;;0 other=true"
611          );
612
613          let m1 = SimpleMetric::new("test", Some(State::Ok), 12, Some(14), None, Some(0), None);
614          let m2 = SimpleMetric::new("other", None, true, None, None, None, None);
615
616          let mut resource: Resource = resource![m1, m2];
617          resource.set_description("A test description");
618
619          assert_eq!(
620              &resource.to_nagios_string(),
621              "OK: A test description | test=12;14;;0 other=true"
622          );
623      }
624
625      #[test]
626      fn test_resource_with_name() {
627          let mut resource = Resource::new(Some(State::Ok), None);
628          resource.set_name("foo");
629          assert_eq!(&resource.to_nagios_string(), "foo OK")
630      }
631
632      #[test]
633      fn test_state() {
634          assert_eq!(State::Ok.exit_code(), 0);
635          assert_eq!(State::Warning.exit_code(), 1);
636          assert_eq!(State::Critical.exit_code(), 2);
637          assert_eq!(State::Unknown.exit_code(), 3);
638
639          assert_eq!(&State::Ok.to_string(), "OK");
640          assert_eq!(&State::Warning.to_string(), "WARNING");
641          assert_eq!(&State::Critical.to_string(), "CRITICAL");
642          assert_eq!(&State::Unknown.to_string(), "UNKNOWN");
643      }
644  }
```