Creating and Using Templates

Templates provide generalised classes and functions that can be reused with different base types. Best sources of templates is the standard template library (STL).

Templates: Syntax and Overview

```
template<template_params> declaration
```

declaration is a function or class (structs and unions) work as well.

```
template<typename T>
class Vec2D {
    public:
    T x, y;
};
```

Each occurrence of `T` is a type supplied later. The value of `T` can be supplied when instantiating the object.

Function Templates

Easy to declare and use, to instantiate them they just need to be called.

```
template<typename type_args> return_type name (args) {
    ...
}
```

Dealing with Type Ambiguities

In some cases the argument types are ambiguous. A way around this is to use `static_cast` or to explicitly specify the declare the function's template parameter.

```
name<type>(args);
```

Function Templates with multiple parameters

```
template<typename T1, typename T2>
```

Class Templates

Class template builds a generalised class that can build on top of any data type. Template classes are more complex than instantiating a function template.

```
template<typename T> class class_name {
    // ...
}
```

Class Templates with Member Functions

Class templates become more useful with member function. Functions can be defined:

- inline
- as a prototype within the class and then defined outside

Syntax

```
template<typeargs>
return_type class_name<typeargs>::member_name () {
    // ...
}
```

A similar syntax applies to constructors, ex.

```
template<typename T>
Vec2D<T>::Vec2D() {
    // ...
}
```

Using Integer Template Parameters

Parameterised types are declared with the `typename` syntax, however, a template can also take integers as parameters. A common use is setting a size or limit of some kind.

Template Specialisation

In some cases it may be more beneficial to handle specific types differently - known as template specialisation

Syntax

```
template<>
return_type function_name(args) {
    // ...
}
```

Variadic Templates (C++11)

Variadic feature allows for argument lists of any size for template functions.

Summary of Rules

- Parameter packs can be declared for class templates as well as function templates

```
template<parms, typename... pack_name> class_declaration
template<parms, typename... pack_name> function_declaration
```

- Within a function declaration parameter pack can be used to declare a packed argument list, which comes at the end of the list of function arguments.
- the argument list can be qualified with `const` and/or reference operators
- parameter packs cannot be used to instantiate an object
- Number of elements can be determined using `sizeof...` operator