# C++ Programming I

## Generic Programming
## Macros & Templates

*C++ Programming*
*May 3, 2018*

Dr. P. Arnold
Bern University of Applied Sciences

# Agenda

- **Macros**
  - Define Constants
  - Include Guards
  - Macro Functions
  - Pros & Cons of Macros



# Agenda

- **Macros**
  - Define Constants
  - Include Guards
  - Macro Functions
  - Pros & Cons of Macros
- **Templates**
  - Template Functions



# Agenda

- **Macros**
  - Define Constants
  - Include Guards
  - Macro Functions
  - Pros & Cons of Macros
- **Templates**
  - Template Functions
- **Template Classes**
  - Template with Multiple Paramters
  - Template Specialisation
  - Variadic Templates
  - Variadic Templates



# Macros



# Macros

## Intro

- The preprocessor runs before the compiler starts
- The preprocessor decides what is compiled depending how you set it!
- Preprocessor directives are characterized by the fact that they all start with a # sign. For example:

```
1  // instruct preprocessor to insert contents of iostream here
2  #include <iostream>
3
4  // define a macro constant
5  #define ARRAY_LENGTH 25
6  int numbers[ARRAY_LENGTH]; // array of 25 integers
7
8  // define a macro function
9  #define SQUARE(x) ((x) * (x))
10 int TwentyFive = SQUARE(5);
```

- The preprocessor makes a simple text replacement and does not check for type correctness!
- The square function for example works with int, double etc.

# Define Constants

## Intro

- The syntax of using `#define` to compose a constant is:

```
1 // General Syntax
2 #define identifier value
3
4 // For example
5 #define PI 3.1416 // double or float?
```

- The defined constants are replaced by the preprocessor where they appear and are applicable to every section of the code, *for loops, arrays* etc.

# Include Guards

## The most frequently used Macro

- Usually, code is split into `*.h` and `*.cpp` files. Hence multiple inclusion with `#include <header.h>` happens often
- For the preprocessor two header files that include each other is a problem of recursive nature
- Use macros in conjunction with preprocessor directives `#ifndef` and `#endif` to avoid multiple inclusion

```
1  #ifndef HEADER1_H_ //multiple inclusion guard:
2  #define HEADER1_H_ // read this and following lines once
3  #include <header2.h>
4  class Class1
5  {
6      // class members
7  };
8  #endif // end of header1.h
9
10 // and another class ////////////////////////////////
11
12 #ifndef HEADER2_H_//multiple inclusion guard
13 #define HEADER2_H_
14 #include <header1.h>
15 class Class2
16 {
17     // class members
18 };
19 #endif // end of header2.h
```

# Macro Functions

## Intro

- The capability of the preprocessor to simply replace text elements makes it capable to write simple functions

```
1  #define PI 3.1416
2  #define AREA_CIRCLE(r) (PI*(r)*(r)) // note brackets
3
4  int main()
5  {
6      int num = 2;
7      std::cout << AREA_CIRCLE(num) << std::endl; // ~12.56
8  }
```

# Macro Functions

## Intro

- The capability of the preprocessor to simply replace text elements makes it capable to write simple functions

```
1  #define PI 3.1416
2  #define AREA_CIRCLE(r) (PI*(r)*(r)) // note brackets
3
4  int main()
5  {
6      int num = 2;
7      std::cout << AREA_CIRCLE(num) << std::endl; // ~12.56
8  }
```

- Be careful with the brackets

```
1  #define PI 3.1416
2  #define AREA_CIRCLE(r) (PI*r*r) // less brackets
3
4  int main()
5  {
6      std::cout << AREA_CIRCLE(1+1) << std::endl;
7      // -> PI*1+1*1+1 = 5.1416
8  }
```



Bern University
of Applied Sciences



# Pros & Cons of Macros

## Intro

- \+ Macro Function can be reused with different data types

```
1  #define MAX(a, b) (((a) > (b)) ? (a) : (b))
2  #define MIN(a, b) (((a) < (b)) ? (a) : (b))
3  .
4  .
5
6  MIN(1, 12)
7  MIN(0.53, 2.8)
```

- \+ Inline expansion before compilation (superior performance)
- – No type safety, *i.e.* `AREA_CIRCLE("Hello World")`

**Templates**

Templates were among others introduced to supply a better and safer alternative to C++

# Templates



# Templates

## Intro

- Templates are one of the most powerful features of the C++ language
- Templates in C++ enable you to define a behavior that you can apply to objects of varying types
- This behaviour is similar to Macros, but Templates are **type safe**!
- You can declare templates for **functions**, thus template functions, as well as for **classes**, hence template classes

```
1 template <parameter list>
2 template function / class declaration..
```

# Templates

## Intro

- Templates are one of the most powerful features of the C++ language
- Templates in C++ enable you to define a behavior that you can apply to objects of varying types
- This behaviour is similar to Macros, but Templates are **type safe**!
- You can declare templates for **functions**, thus template functions, as well as for **classes**, hence template classes

```
1 template <parameter list>
2 template function / class declaration..
```

- The parameter list contains the keyword `typename` that defines the template parameter `objType`
- A typical template function declaration might look like this

```
1 template <typename T1, typename T2 = T1>
2 bool TemplateFunction(const T1& param1, const T2& param2);
```

- A detailed description follows on the next slides ...

# Template Functions

## Using Template Syntax

- A template function is capable of adapting itself to suit parameters of different types.
- Lets implement the `MAX` macro with templates:

```
1  template <typename T>
2  const T& getMax(const T& value1, const T& value2)
3  {
4      return (value1 > value2) ? value1 : value2; // Ternary op.
5  }
```

# Template Functions

## Using Template Syntax

- A template function is capable of adapting itself to suit parameters of different types.
- Lets implement the `MAX` macro with templates:

```
1  template <typename T>
2  const T& getMax(const T& value1, const T& value2)
3  {
4      return (value1 > value2) ? value1 : value2; // Ternary op.
5  }
```

- The function can then be called like this:

```
1  int num1 = 25;
2  int num2 = 40;
3  int maxVal = getMax<int>(num1, num2);
4  int maxVal = getMax(num1, num2);
5
6
7  double double1 = 1.1;
8  double double2 = 1.001;
9  double maxVal = getMax<double>(double1, double2);
10 double maxVal = getMax(double1, double2);
```

**Templates**

The `<int>` used in the call `getMax` defines the template parameter T, but is not necessary for template functions

# Template Functions

## Using Template Syntax

- The compiler generates a version for each parameter type, *i.e.* `int` & `double`
- Templates are type safe!
- A call like this would not compile

```
1 std::string name = "Joe";
2 double double2 = 1.001;
3 double maxVal = getMax(double1, name); // compile error!
```

- The type safety makes templates less prone to errors than macros

# Template Classes



# Template Classes

## Intro

- Template classes are the templatised versions of C++ classes!
- For example `std::vector<int>` is a class holding integers as type



# Template Classes

## Intro

- Template classes are the templatised versions of C++ classes!
- For example `std::vector<int>` is a class holding integers as type
- A simple template class that uses a single parameter T to hold a member variable can be written as the following:

```
1  template <typename T>
2  class TemplateClass
3  {
4  private:
5      T value;
6  public:
7      void setValue(const T& newValue) { value = newValue; }
8      T& getValue() { return value; }
9  };
```

- The type `T` of variable `value` is *instantiated* when the template is used

# Template Classes

## Simple Usage

- A simple usage of `TemplateClass` looks like this:

```
1 TemplateClass<int> tempClass; // template instantiation for int
2 tempClass.SetValue(5);
3 cout << "The value is: " << tempClass.getValue() << endl;
```



# Template Classes

## Simple Usage

- A simple usage of `TemplateClass` looks like this:

```
1 TemplateClass<int> tempClass; // template instantiation for int
2 tempClass.SetValue(5);
3 cout << "The value is: " << tempClass.getValue() << endl;
```

- Similarly, we can use the same class with char strings:

```
1 TemplateClass<char*> tempClass; // instantiation for char*
2 tempClass.setValue("Template");
3 cout << "The value is: " << tempClass.getValue() << endl;
```

# Template Classes

## Simple Usage

- A simple usage of `TemplateClass` looks like this:

```
1 TemplateClass<int> tempClass; // template instantiation for int
2 tempClass.SetValue(5);
3 cout << "The value is: " << tempClass.getValue() << endl;
```

- Similarly, we can use the same class with char strings:

```
1 TemplateClass<char*> tempClass; // instantiation for char*
2 tempClass.setValue("Template");
3 cout << "The value is: " << tempClass.getValue() << endl;
```

- Templates can also be instantiated using a class defined by you

```
1 #include "vector.h" // Your vector
2
3 Vector v(100,23);
4 TemplateClass<Vector> tempClass; // instantiation for char*
5 tempClass.setValue(v);
6 cout << "The value is: " << tempClass.getValue() << endl;
```

# Template Classes

## Multiple Parameters

- The template parameter list can be expanded to declare multiple parameters

```
1  template <typename T1, typename T2>
2  class HoldsPair
3  {
4  private:
5      T1 m_val1;
6      T2 m_val2;
7  public:
8      HoldsPair(const T1& val1, const T2& val2) : m_val1(val1),
           m_val2(val2)
9  };
```

- The usage will look like this:

```
1  // A template instantiation that pairs an int with a double
2  HoldsPair<int,double> pairIntDouble(6, 1.99);
3
4  // A template instantiation that pairs an int with an int
5  HoldsPair<int,int> pairIntDouble(6, 500);
```

# Template Classes

## Default Parameters

- You can also declare a default template parameter type

```
1  template <typename T1=int, typename T2=int>
2  class HoldsPair
3  {
4      /* Declarations */
5  };
```

- The usage will be more compact and look like this:

```
1  // Holds a pair of ints (default type)
2  HoldsPair<> pairInts(6, 500);
```

# Template Specialisation

## Special Cases

- When using a template the compiler is instructed to create a class for you using the template and instantite it for the types specified as template arguments
- Sometimes however, you require a (different) behavior of a template when instantiated with a specific type
- This can be achieved by **template specification**
- The specification for type `int` for example is declared as follows:

```
1 template<> class HoldsPair<int, int>
2 {
3     // implementation code here
4 };
```

# Variadic Templates C++ 14

## Adder

- We can easily implement a function to sum two values

```
1  template <typename T1, typename T2, typename T3>
2  void sum(T1& result, T2 num1, T3 num2)
3  {
4      result = num1 + num2;
5  }
```

# Variadic Templates C++ 14

## Adder

- We can easily implement a function to sum two values

```
1  template <typename T1, typename T2, typename T3>
2  void sum(T1& result, T2 num1, T3 num2)
3  {
4      result = num1 + num2;
5  }
```

- However, if you were required to write one single function that would be capable of adding any number of values!

```
1  long sum = adder(1, 2, 3, 8, 7);
2
3  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
4  std::string ssum = adder(s1, s2, s3, s4);
```



# Variadic Templates C++ 14

## Adder

```
1  // Base function
2  template<typename T>
3  T adder(T v)
4  {
5      return v;
6  }
7
8  // General function
9  template<typename T, typename ... Args>
10 T adder(T first, Args ... args)
11 {
12     return first + adder(args ...); // recursion
13 }
```

- Since C++ 14 we can use **variadic templates**
- `typename ... Args` is called a *template parameter pack*
- `Args ... args` is called a *function parameter pack*
- Write a base function and a general function which “recurses”
- With each call, the parameter pack gets shorter by one parameter
- (Demo `adder`)

# Thank You

## Questions

???