Generic Programming: Templates

- **Generic Programming** enables the programmer to write a general (generic) algorithm (or an entire class) that will work with different data types.
- The idea is to pass the **data type as a parameter** so that we do not need to write the same code for different data types.
- "Generics" (function or class) are implemented in C++ using **templates**.
- Instead of specifying the actual data type used in a function or class, in templates, we provide a placeholder that gets replaced by the corresponding data type provided during compilation.
- The compiler generates **different executable codes from the same source code** based on the data type provided during compilation (instantiation of the code).
- The template feature in C++ provides a way to **reuse *source code***.
- In C++, we can write function templates and class templates.

Function Templates

Example:

Suppose you want to write a function that returns the absolute value of a number.

Ordinarily, this function would be written for a particular data **type** (for example, **int**):

```
int abs(int n) {                 // absolute value of integers
    return n < 0 ? -n : n;       // if n is negative, return -n
}
```

Here, the function is defined as taking an argument of **type int** and returning a value of this same type.

But now, suppose you want to find the absolute value of a **type long int**. You need to write a completely new function:

```
long int abs(long int n) {          // absolute value of long ints
    return n < 0 ? -n : n;
}
```

And again, for **type double**:

```
double abs(double n){               // absolute value of doubles
    return n < 0 ? -n : n;
}
```

Function Templates (contd)

- The function's body is the same in each case, but they must be separate functions because they handle variables of different types.

Can we solve this problem using function overloading?

- In C++, these functions can all be overloaded to have the same name, but we must nevertheless write **a separate** definition (body) **for each one**.
- In the C language, which does not support overloading, functions for different types cannot even have the same name. In the C function library, this leads to families of similarly named functions, such as `abs()`, `fabs()`, `labs()`, and so on.

Problems:

- Rewriting the same function body over and over for different types wastes time as well as space in the listing.
- Also, if you find an error in one such function, you must remember to correct it in each function body.

  Failing to do this correctly will introduce inconsistencies in your program.

> It would be nice if there were a way to write such a function **just once** and have it work for many different data types.
> This is exactly what function templates do for you.

Example: A function template to calculate the absolute value of a number

```
template <typename T>        // function template
T abs(T n)
{
    return n < 0 ? -n : n;
}
```

The `template` keyword signals the compiler that we will define a template.

When an instance of the template is created, every occurrence of T is replaced by an actual type.

The `typename` keyword identifies T as a generic data type.
The `class` keyword can also be used here.
There is really no distinction between types and classes.

The variable following the keyword `typename` or `class` (T in this example) is called the *template parameter*.
The type assigned to a type parameter T during instantiation is called a *template type argument*.

The name of a template type parameter T can be used anywhere in the template's function signature, return type, and body.
It is a placeholder for a type and can thus be used in any context where you would normally use the concrete type.

Example:

```
int int1{-599};
int2 = abs(int1);  // Template parameter T is replaced by int
```

Template is **instantiated** as `int abs(int n) {...}`

Example: A function template to calculate the absolute value of a number (contd)

- The key innovation in function templates is to represent the data type used by the function not as a specific type, such as `int`, but by a *generic name* that can stand for *any* type.

  In our example, this name (**template argument**) is `T`.

What does the compiler do when it sees the `template` keyword and the function definition that follows it?

- The function template itself does not cause the compiler to generate any code.

  It cannot generate code because it does not know yet what data type the function will be working with.
- It simply remembers the template for possible future use.
- Code is generated (compiled) according to the function call statement.
- This happens in expressions such as `abs(int1)` in the statement:

```
result = abs(int1);
```

- If the data type of input argument `int1` is `int`, the compiler generates a specific version of the function `T abs(T n)` for type `int`, substituting **int** wherever it sees the name `T` in the function template. **int** → **T**
- This is called ***instantiating*** the function template.

Example: A function template to calculate the absolute value of a number (contd)

We use the function in the normal way.
The compiler deduces the type to replace `T` from the argument in the `abs()` function call. This mechanism is referred to as *template argument deduction.*

```
int main()
{
   short int short_int1{ 500 };
   short int short_int2{ -600 };
   long long int  long_long_int1{ 123456789012345 };
   long long int  long_long_int2{ -9876543210987654 };
   double double1{ 9.95 };
   double double2{ -10.15 };
   // calls instantiate functions
   std::println("{}", abs(short_int1));          // abs(short int)
   std::println("{}", abs(short_int2));          // abs(short int)
   std::println("{}", abs(long_long_int1));      // abs(long long int)
   std::println("{}", abs(double1));             // abs(double)
    ...
}
```

See Example e09_1.cpp

With which data types can a template function work?

- The data type must support operations performed in the function.

  For example, two operators are used in the abs function, i.e., `n < 0` and `-n`.
- Each data type supporting these operators (`<` and `-`) can be used with the abs function.

Benefits:

- We have saved having to type three separate functions for different data types (`short int`, `long long int`, `double`) into the source file.
- This makes the listing shorter and easier to understand.
- Also, if we want to change the way the function works or if we need to improve it, we need to make the change in only one place in the listing instead of three.

Notice:

- The executable program uses the same amount of RAM whether we use the template approach or write three separate functions.
- The compiler generates each template instance for a particular data type (e.g., `int`) only once.
- If a subsequent function call requires the same instance, it calls the existing instance.

Objects as Template Arguments

Example:

- We define a template function `maxOf()` that returns the larger of two arguments.
- We want this function to operate on built-in types (e.g., `char`, `int`, `float`) and programmer-defined types (classes), e.g., complex numbers.

```
// The function returns the larger of two arguments
template <typename T>
const T & maxOf(const T & n1, const T & n2)
{
   return n1 > n2 ? n1 : n2;
}
```

- Since we will pass objects as arguments, the function's parameters, and return type are defined as **references**.
- Since the function uses the greater-than operator `>`, the programmer-defined types (classes) must support (overload) this operator if we want to apply this function on their objects.
- Note: The Standard Library has a `std::max()` template function.

Example (contd):

If we want to apply the `maxOf()` function on the programmer-defined complex number objects, the related class must overload the greater-than operator >.

```
class ComplexT{        // A class to define complex numbers
public:
  ComplexT(double in_r, double in_im) :m_re{ in_r }, m_im{ in_im }{}
  bool operator>(const ComplexT&) const;  // overloading operator >
  double getRe()const {return m_re;}
  double getIm()const {return m_im;}
private:
  double m_re{}, m_im{};
};

// The Body of the function for operator >
// The function compares the sizes of two complex numbers
bool ComplexT::operator>(const ComplexT& in) const {
   double size = m_re * m_re + m_im * m_im;
   double in_size = in.m_re * in.m_re + in.m_im * in.m_im;
   return size > in_size;
}
```


Example (contd):

```
int main()
{
  char c1{ 'D' }, c2{ 'N' };
  int i1{ 5 }, i2{ -3 };
  double d1{ 3.05 }, d2{ 12.47 };
  ComplexT z1(1.4,0.6), z2(4.6,-3.8);
  cout << maxOf(c1, c2) << endl;
  cout << maxOf(i1, i2) << endl;
  cout << maxOf(d1, d2) << endl;
  cout << maxOf(z1, z2) << endl;//operator << must be overloaded for ComplexT
  return 0;
}
```

```
// Overloading the operator << for Complex numbers
std::ostream& operator <<(std::ostream& out, const ComplexT& z)
{
  out << "( " << z.getRe() << " , " << z.getIm() << " )" << endl;
  return out;
};
```

See Example e09_2.cpp

Function Templates with Multiple Arguments, including built-in types

Example:

- We will write a function that searches an array for a specific value.
- The function returns the array index for that value if it finds it or -1 if it can't find it.
- This function template takes three arguments: two template arguments and one basic type.
- The arguments are a pointer to the array, the value to search for, and the size of the array.

```
// function returns the index number of an item, or -1
template <typename T>
int find(const T* array, T value, unsigned int size)
{
    for (unsigned int j = 0; j < size; j++)
        if (array[j] == value) return j;
    return -1;
}
```

See Example e09_3.cpp

Template Arguments Must Match:

- When a template function is invoked, all instances of the same template argument must be the same type.

  For example, in `find()`, if the array is of type `int`, the value to search for must also be of type `int`.
- The following statements generate a compiler error.

```
int intarray[ ] {1, 3, 5, 7};          // an array of ints
float f1{ 5.0 };                       // float value
int value = find(intarray, f1, 4);     // ERROR!
```

- The compiler expects all instances of `T` to be the same type.

  `int find(const T* array, T value, unsigned int size)`
- It can generate a function `find(int*, int, unsigned int);`
- However, it cannot generate `find(int*, float, unsigned int);`

Multiple Template Arguments

You can use more than one template argument in a function template.

Example:

- Suppose you like the idea of the `find()` function template but are unsure how large an array it might be applied to.
- If the array is too large, then type `unsigned long int` would be necessary for the array size instead of `unsigned int`.
- For a small array, the type `unsigned short int` would be sufficient.
- You want to select the type of the array size, as well as the type of data stored when you call the function.

```
// The type of size of the array is a template type
template <typename T1, typename T2>
T2 find(const T1* array, T1 value, T2 size)
{
    for (T2 j = 0; j < size; j++)
        if (array[j] == value) return j;
    return -1;              // or return static_cast<T2>(-1);
}
```

We can add an explicit cast to `T2` to silence any warnings about implicit conversions: `return static_cast<T2>(-1);`

Example (contd):

- Now, you can use the appropriate type, `short int`, `int`, or `long int` (or even a programmer-defined type) for the size.
- The compiler will generate different functions based on the type of the array and the value to be searched for, as well as on the type of the array size.

```
int main(){
  short int short_size{ 7 };
  char chrArr[] { 'a', 'c', 'f', 's', 'u', 'x', 'z'};      // array
  char ch{ 'f' };                                         // value to find
  cout << find(chrArr, ch, short_size);
```

See Example e09_4.cpp

- Note that multiple template arguments can lead to instantiating many functions from a single template.
- Two such arguments, if six basic types could reasonably be used for each one, would allow the creation of up to 36 functions.
- This can take up much memory if the functions are large. On the other hand, the compiler does not instantiate a version of the function unless you actually call it.

Explicit Template Arguments

- We can specify the argument for a template parameter explicitly when we call the function.
- The compiler no longer deduces the type to replace `T`; it accepts what we specify.

Example:

- We can force the compiler to generate the `double` version of the `maxOf` function in the example e09_2.cpp.

```
int i1{ 3 };
result = maxOf<double>(i1, 3.14);    // result = 3.14
```

- Similarly, we can force the compiler to generate the `int` version.

```
int i1{ 3 };
result = maxOf<int>(i1, 3.14);    // result = 3
```

In this case, the compiler creates a template instance with `T` as type `int`.

This necessitates an implicit conversion of the second argument `3.14` to `int`.

The result of this conversion is the value 3!

Most compilers will generate a warning message about such dangerous conversions.

Non-Type Template Parameters

- Function templates can also have *non-type parameters* that require *non-type arguments*.

Example:

We write a template function to perform range checking on a value with predetermined limits.

```
template <typename T, int lower, int upper>
bool is_in_range(const T& value)
{
   return (value <= upper) && (value >= lower);
}
```

Non-type parameters

Now we can use this template as follows:

```
if (is_in_range<double, 0, 100>(value)) ... //checks 0 to 100 for a double
    else ...
...
if (is_in_range<int, -10, 20>(value)) ... //checks -10 to 20 for an int
    else ...
```

Non-Type Template Parameters (contd)

- We can put the two non-type template parameters to the left of the template type parameter.

Example:

```
template <int lower, int upper, typename T>
bool is_in_range(const T& value)
{
  return (value <= upper) && (value >= lower);
}
```

In this case, we do not need to specify the argument for the template parameter `T` explicitly; the compiler can deduce the type argument.

Now, we can use this template as follows:

```
double value{25.7};
if (is_in_range<0, 100>(value)) ...    //checks 0 to 100 for a double
    else ...
```

We provide only the non-type parameters

Non-Type Template Parameters (contd)

- We can also use the name of the template type parameter `T` as the type of the other non-type template parameters.

```
template <typename T, T lower, T upper>
bool is_in_range(const T& value)
{
  return (value <= upper) && (value >= lower);
}

double value{25.7};
if (is_in_range<double, 0.5, 47.8>(value))
                                        //checks 0.5 to 47.8 for a double
    else ...
```

- In this case, we cannot put the non-type template parameters to the left of the template type parameter.
- You can only refer to the names of type parameters declared to the left of a non-type parameter.

Return Type Deduction in Templates using auto

- When type names are complicated (verbose, long), or you do not want to decide what the return type should be, you can use the auto keyword for the return type of a function.

Example:

```
auto function1(int, double){
    ....
    return ...;
}
```

- The compiler will deduce the function's return type by considering the return statements in the function definition.
- The keyword auto never deduces to a reference type, always to a value type.
- To have the compiler deduce a reference type, you should write **auto&** or **const auto&**.

Return Type Deduction in Templates using auto (contd)

Example:

We write a template function sumOf to add two numbers of different types.

```
template <typename T1, typename T2>
auto sumOf(const T1& n1, const T2& n2)   // What is the return type?
{
   return n1 + n2;    // Types must support the + operator
}

int main()
{
    int i1{ 5 }, i2{ -7 };
    double d1{ 3.05 }, d2{ -18.42 };

    std::println("Sum of i1 and i2 = {}", sumOf(i1, i2) );
    std::println("Sum of i1 and d1 = {}", sumOf(i1, d1) );
    ...
```

Return type: int

Return type: double

See Example e09_5.cpp

Abbreviated Function Templates

- Since C++ 20, we can use the `auto` keyword as a placeholder also for function parameter types.

Example:

Remember the function template that calculates the absolute value of a number:

```
template <typename T>       // function template
T abs(T n) { return n < 0 ? -n : n; }
```

We can write the same function template as follows:

```
auto abs(auto n) { return n < 0 ? -n : n; }
```

- Even though the **definition** does not use the `template` keyword, it is a *function template.*
- The only difference is that the new syntax is shorter. Therefore, it is called an **abbreviated function template**.
- Placeholder types such as `auto*`, `auto&`, and `const auto&` are also allowed.

Example:

```
auto abs(const auto & n) { return n < 0 ? -n : n; }
```


Abbreviated Function Templates (contd)

- Every occurrence of `auto` in the function parameter list introduces an implicit, unnamed template-type parameter.

The following two prototypes are, therefore, completely equivalent:

```
auto sumOf(const auto& n1, const auto& n2);

template <typename T1, typename T2>
auto sumOf(const T1& n1, const T2& n2);
```

See Example e09_6.cpp

Limitations:

- If the function template has multiple parameters of the same type, we must use the old syntax.

For example, we want to add two numbers only of the same type.

```
template <typename T>
auto sumOf(const T& n1, const T& n2);
```

- We use the old syntax if we want to refer to one of the parameter type names in the function body.

Remember the `find` template on slide 9.13.

Class Templates

- The template concept can also be applied to classes.
- Class templates are generally used for data storage (container) classes.
- For example, vectors, stacks, and linked lists.
- Non-template classes can store data of only a single basic type or related types, i.e., if there is an inheritance relation between them.

Example:

The `Stack` class in the program that is presented below can store data only of type `int`:

```
class Stack {
public:
   Stack();                    // constructor
   void push(int);             // takes int as argument
   int pop();                  // returns int value
   static inline const int MAX{100};
private:
   int m_data[MAX];            // array of ints
   int top{};                  // index number of top of the stack
};
```


Example (contd):

If we wanted to store data of type `double`, for example, in a stack, we would need to define a completely new class:

```
class DoubleStack {
public:
   DoubleStack();              // constructor
   void push(double);          // takes double as argument
   double pop();                  // returns double value
   static inline const unsigned int MAX{100};
private:
   double m_data[MAX];         // array of doubles
   unsigned int top{};         // index number of top of the stack
};
```

A class template to define stacks for different types:

```
template <typename T>
class Stack {
public:
  Stack() = default;
  void push(T);                        // put a number on the stack
  T pop();                             // take number off the stack
  static inline const int MAX{ 100 };
private:
  T m_data[MAX];                     // array of any type
  unsigned int m_top{};              // index number of top of the stack
};
```

To use this stack for objects, we should pass and return parameters using references.

```
void push(const T&);
const T& pop();
```


A class template to define stacks for different types (contd):

```
template<typename T>
void Stack<T>::push(T in)           // put a number into stack
{
   if(m_top == MAX)                 // if stack full,
     throw "Stack is full!";        // throw exception
   m_data[m_top++] = in;
}

template<typename T>
T Stack<T>::pop()                  // take number off the stack
{
   if(m_top == 0)                  // if stack empty,
        throw "Stack is empty!";   // throw exception
   else return m_data[--m_top];
}
```

A class template to define stacks for different types (contd):

```
int main()
{
  Stack<double> s_double;     // s_double is object of class Stack<double>
  try{
     s_double.push(1111.1);                     // push 3 doubles
     s_double.push(2222.2);
     std::println( "1: {}", s_double.pop());  // pop and print 2 doubles
     std::println( "2: {}", s_double.pop());
}
  catch(const char * msg)                     // exception handler
  {
     std::println("{}", msg);
  }

  Stack<long int> s_long;      // s_long is object of class Stack<long int>
  s_long.push(123123123L);                  // push long integers
    :
  std::println( "1: {}", s_long.pop());  // pop and print long integers
    :
```

See Example e09_7a.cpp

A class template to define stacks for different types (contd):

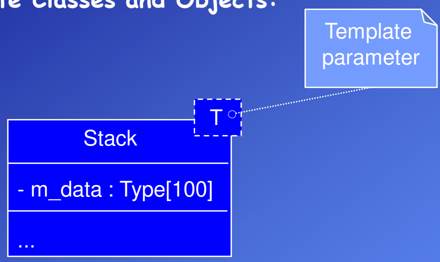
We can use this stack template also to store pointers to objects of Point and ColoredPoint classes.

```
int main()
{
  Stack<const Point *> s_pointPtr;     // stack for pointer to Points
  ColoredPoint  col_point1{ 10, 20, Color::Blue };
  s_pointPtr.push(&col_point1);       // Push a pointer to a colored point
  Point *ptrPoint1 = new Point {30, 40};    // Dynamic Point object
  s_pointPtr.push(ptrPoint1);        // Push a dynamic point into the stack
  s_pointPtr.pop()->print();         // pop and call the print()
  s_pointPtr.pop()->print();         // pop and call the print()
  delete ptrPoint1;
  return 0;
}
```

See Example e09_7b.zip

UML Notation for Template Classes and Objects:

A template class:



An object of template `Stack`.
In this example, the `intStack` object is an integer `Stack`.