# Generic Programming

Compiled by
**M S Anand**

Department of Computer Science

## Generic Programming
## Introduction

What is “Generic Programming” ?

“**Generic programming** is a style of computer programming in which algorithms are written in terms of data types *to-be-specified-later* that are then *instantiated* when needed for specific types provided as parameters. This approach, pioneered by the ML programming language in 1973, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplicate code” - Wikipedia

## Generic Programming
### Introduction

Why Generic Programming?

- ➢ Maximize code reuse
- ➢ type safety
- ➢ performance.

Some popular languages which support genericity

C++ - using Templates (the most popular and commonly used approach in practice too). Pretty complex to understand for a beginner.

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as **lists**, **stacks**, **arrays**, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.

## Generic Programming
### Introduction

Java – Java’s parameterized classes, Collection framework

C# - The . NET class library contains several generic collection classes in the System

## Generic Programming

### Introduction

The purpose of the course is to enable the student to implement efficient generic algorithms and data structures in C++.

In generic programming an algorithm can be described in terms of abstract types that are only instantiated when the algorithm is used with concrete types.

Templates in C++ is a compile-time mechanism that makes it possible to implement generic algorithms on a high abstraction level, which are statically type safe and have minimal runtime overhead.

## Generic Programming

### Introduction

As part of the syllabus, most of the emphasis will be on C++.

A decent exposure to GP in Java will be part of the course.

It is a programming course. Hence, mostly hands-on

Will get to work on interesting mini projects as part of the course.

Any prerequisites?
Basic exposure to C++ and Java would be a big advantage.

If you have a flair for programming and want to be good at writing reusable, type safe, efficient code, this course is for you.

## Generic Programming
### C++ features

Primitive data types

char

short

integer

long

float

double

long long

# Generic Programming

## An overview of C++ – Data types and range of values

.

| Variable type | Keyword | Bytes required | Range |
|---|---|---|---|
| Character | char | 1 | -128 to 127 |
| Unsigned character | unsigned char | 1 | 0 to 255 |
| Integer | int | 2 (?) | -32768 to 32767 |
| Short integer | short int | 2 | -32768 to 32767 |
| Long integer | long int | 4 | -2,147,483,648 to 2,147,483,647 |
| Unsigned integer | unsigned int | 2(?) | 0 to 65535 |
| Unsigned short integer | unsigned short | 2 | 0 to 65535 |
| Unsigned long integer | unsigned long | 4 | 0 to 4,294,967,295 |
| Float | float | 4 | -3.4E+38 to +3.4E+38 |
| Double | double | 8 | -1.7E+308 to +1.7E+308 |
| Long long | long long | 8 | |
| Long double | long double | 10 | |

## Generic Programming
## An overview of C++

**<u>Basic aritmetic operations</u>**:

Addition – ‘+’

Subtraction – ‘-’

Multiplication – ‘*’

Division – ‘/’

Modulo division – ‘%’

Unary minus operator
It produces a positive result when applied to a negative operand and a negative result when the operand is positive.

## Generic Programming

## An overview of C++

Assignment operators:

| | |
|---|---|
| = | Example: sum = 10 |
| += | sum += 10; (Same as sum = sum + 10) |
| -= | sum -= 10; |
| *= | sum *= 10; |
| /= | sum /= 10; |
| %= | sum %= 10; |

……………………..

## Generic Programming
## An overview of C++

Relational operators:

| | |
|---|---|
| > | x > y |
| < | x < y |
| >= | x >= y |
| <= | x <= y |
| != | x != y |
| == | x == y |

## Generic Programming

## An overview of C++

Increment and decrement operators:

Pre and post increment

Pre and post decrement

variable ++

++ variable

variable –

--variable

## Generic Programming

## An overview of C++

Logical operators:

| | |
|---|---|
| && - Logical AND | True only if all operands are true |
| \|\| - Logical OR | True if any of the operands is true. |
| ! – Logical NOT | True if the operand is zero. |

## Generic Programming
## An overview of C++

Bitwise operators:

| | | |
|---|---|---|
| & | - | Bitwise AND |
| \| | - | Bitwise OR |
| ^ | – | Bitwise exclusive OR |
| ~ | - | Bitwise complement |
| << | - | Shift left |
| >> | - | Shift right |

Other operators
**Comma** operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

**sizeof** operator

## Generic Programming

## An overview of C++

Decision making

if, else, else if

switch case

The ternary operator

condition ? expression1 : expression2

```
x = y > 7 ? 25 : 50;
```

The goto statement

```
goto label;
```

label:

## Generic Programming

## An overview of C++

·Type conversions

**Explicit** – You do it yourself in the program

**Implicit** – The compiler does it for you

**Enumerations**

enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

enum Weekday today = Wednesday;

## Generic Programming
## An overview of C++

Operator precedence and associativity
The C operator precedence table is here.

The loops:

The “**for**” loop

The “**while**” loop

The “**do … while**” loop

The **break** statement

The **continue** statement

## Generic Programming

## An overview of C++

**Arrays**

Index starts from 0

Single, Multi-dimensional arrays

Row major or Column major ?

A string in C is a character array terminated with '\0'

## Generic Programming

## An overview of C++

·The nine most commonly used functions in the string library are:
**strcat** - concatenate two strings
**strchr** - string scanning operation
**strcmp** - compare two strings
**strcpy** - copy a string
**strlen** - get string length
**strncat** - concatenate one string with part of another
**strncmp** - compare parts of two strings
**strncpy** - copy part of a string
**strrchr -** a pointer to the **last** occurrence of c within s instead of the first.

## Generic Programming
## An overview of C++

**Using the sizeof operator with arrays**

The sizeof operator executed on an array gives the size of the array in bytes:

```
    char arr [20];
    sizeof (arr) – results in 20 bytes
```

What about the following:

```
    int iarr [20];
    sizeof (iarr)   -  ??

    long larr [40];
    sizeof (larr)   -   ??
```

```
const char hex_array[] = {'A', 'B'};
```

## Generic Programming

## An overview of C++

**Pointers**

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer.

## Generic Programming

## An overview of C++

**<u>Arrays and Pointers – Differences</u>**

1. the sizeof operator
   a. sizeof(array) returns the amount of memory used by all elements in array
   b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself.
2. the & operator
   a. &array is an alias for &array[0] and returns the address of the first element in array
   b. &pointer returns the address of pointer
3. a string literal initialization of a character array
   a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
   b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic on pointer variable is allowed.

## An overview of C++

**Pointers**
**Note**

char* is a **mutable** pointer to a **mutable** character/string.

const char* is a **mutable** pointer to an **immutable** character/string. You cannot change the contents of the location(s) this pointer points to. Also, compilers are required to give error messages when you try to do so. For the same reason, conversion from const char * to char* is deprecated.

char* const is an **immutable** pointer (it cannot point to any other location) **but** the contents of location at which it points are **mutable**.

const char* const is an **immutable** pointer to an **immutable** character/string.

**void *ptr;**

# Generic Programming

## An overview of C++

**Structure**

A structure is a <u>user defined data type </u>in C.

A structure creates a data type that can be used to group items of <u>possibly different types </u>into a single type.

***How to create a structure?***

'struct' keyword is used to create a structure.

```
struct address
{
   char name[50];
   char street[100];
   char city[50];
   char state[20];
   int pin;
};
```

## Generic Programming

### An overview of C++

Accessing structure members through the “.” Operator**.**

If you have a pointer to a structure, then use the -> operator.

sizeof a structure.

## An overview of C++

**Dynamic memory allocation**

void *malloc(int size);

void *calloc(int nmemb, int size); – Memory set to zero.

void *realloc(void **ptr***,** size_t *size*);

free(void *ptr);

## An overview of C++

**Linked lists**

```
struct Node
{
  int data;
  struct Node *next;
};

struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};
```

# Generic Programming

## An overview of C++

<u>**File operations in C**</u>

**FILE *fopen(const char *name, const char *mode);**

<u>**fclose(FILE *)**</u>

**size_t fread(void ***_ptr_**, size_t** _size_**, size_t** _nmemb_**, FILE ***_stream_**);**

**size_t fwrite(const void ***_ptr_**, size_t** _size_**, size_t** _nmemb_**, FILE ***_stream_**);**

**int fseek(FILE ***_stream_**, long** _offset_**, int** _whence_**);**

**long ftell(FILE ***_stream_**);**

Lot more functions like fgetc(), fgets(), fputc(), fputs() …….

## Generic Programming

## An overview of C++

**Bit fields**

```
// A space optimized representation of date
struct date
{
   // d has value between 1 and 31, so 5 bits are sufficient
   unsigned int d: 5;

   // m has value between 1 and 12, so 4 bits are sufficient
   unsigned int m: 4;

   unsigned int y;
};
```

## Generic Programming
## An overview of C++

**Unions**

User defined data type just like a structure.

```
union sample
{
    char name [4];
    long  length;
    short s1;
};
```

# Generic Programming
## An overview of C++

```
#include <stdio.h>

int main (void)
{
    union long_bytes
              {
                  char ind [4];
                  long ll;
              } u1;

              long l1 = 0x10203040;
              int i;
              u1.ll = l1;

              for (i = 0; i < sizeof (long); i ++)
                            printf ("Byte %d is %x\n", i, u1.ind [i]);
              return 0;
}
```

## Generic Programming

## An overview of C++

**Little Endian and Big Endian**

C language uses 4 storage classes,
**auto**, **extern**, **static** and **register**.

**auto**
This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language.

**extern**:
Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.

**static**
Their scope is local to the function to which they were defined.

## Generic Programming

## An overview of C++

**register**
This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available.

**volatile**
C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time--without any action being taken by the code the compiler finds nearby.

## Generic Programming

## An overview of C++

**Functions in C**

Function declaration

Function call

Function definition

Pointers to functions

function_return_type(*Pointer_name)(function argument list)

For example:

double  (*p2f)(double, char)

Functions with default arguments:  DaF

## Generic Programming
## An overview of C++

### Command line arguments

```
int main (int argc, char **argv)
{
}
```

argc ??

argv[0] - ??

How to pass an argument like “PES University”?

## Generic Programming

## An overview of C++

**Environment variables**

```
int main (int argc, char **argv, char **envp)

extern char **environ;
```

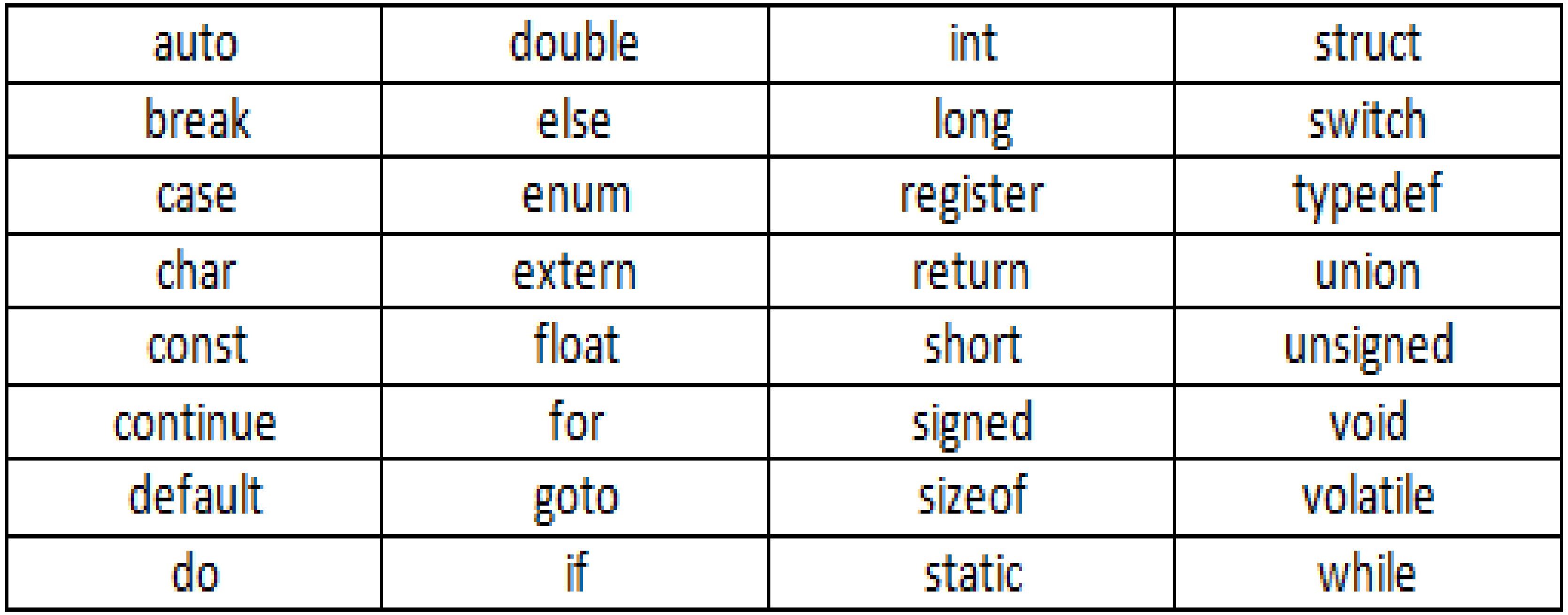
Functions with variable number of arguments

```
int func(int, ... )
{ . . .
}

int main()
{
		func(1, 2, 3);
		func(1, 2, 3, 4);
}
```

## Generic Programming

### Keywords in C++



| | | | |
|---|---|---|---|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |

## Generic Programming

## An overview of C++

**Namespace**

A namespace is **a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it**. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. You can define as many namespaces as you want.

**std** namespace contains most of the standard library functions:

```
#include <iostream>
using namespace std;

cin
cout ..   Etc

Or
std::cin
std::cout
```

Sample program: Namespace

## Generic Programming

## An overview of C++

Defining a class

```
class ClassName
{
            Access specifier: // private, public or protected

            Data members;  // Variables to be used

            Member functions() { } //Methods to access data members

};                    // Class name ends with a semicolon
```

Creating objects

```
            X x1;       // default constructor
            X x2(1);    // parameterized constructor
            X x3=3      // Parameterized constructor with single
      argument
```

Sample program: Class

# Generic Programming

## An overview of C++

**Types of variables**

Global, Local, Instance, Static, Automatic and external variables.

User defined functions

Function overloading: Example1

Overloaded functions can have:

1. Different number of arguments
2. Different types of arguments
3. Different order of arguments

Default arguments

Reference parameters: Example

Variable number of arguments.

## Generic Programming

## An overview of C++

**Inline functions**

Avoid function call overhead. A function specified as **inline** (usually) is expanded "in line" at each call.

Each time an inline function is called, the compiler copies the code of the inline function to that call location (avoiding the function call overhead).

Example

**Lambda functions**

C++ Lambda expression allows us to define anonymous function objects (functors) which can either be used inline or passed as an argument.

Lambda expression was introduced in C++11 for creating anonymous functors in a more convenient and concise way.

A basic lambda expression can look something like this:

```
auto greet = []() {
  // lambda function body
};
```

Example Example2

## Generic Programming
## An overview of C++

**Template functions**
Example

Abstract datatypes in C++

String, Vector

**Range-Based for**

```
for (declaration : expression)
            statement
```

Example

## Generic Programming

## An overview of C++

Use of "auto" keyword in C++

**Type Inference in C++ (auto and decltype)**

**auto keyword:** The auto keyword specifies that the type of the variable that is being declared will be automatically inferred from its initializer.

In the case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.

**Note:** The variable declared with auto keyword should be initialized at the time of its declaration or else there will be a compile-time error

Example

# Generic Programming

## An overview of C++

·The **decltype** type specifier

The **decltype** type specifier **yields the type of a specified expression**.

(The decltype type specifier, together with the auto keyword, is useful primarily to developers who write template libraries. Use auto and decltype to declare a function template whose return type depends on the types of its template arguments)

Example

## Generic Programming
## An overview of C++

A **vector** is a collection of objects, all of which have the same type. Every object in the collection has an associated index, which gives access to that object.

A vector is often referred to as a **container** because it “contains” other objects.

**Note**
vector is a template, not a type. Types generated from vector must include the element type, for example, vector<int>.

Example

## An overview of C++

**Arrays**

Mostly like arrays in C with a few additions.

Two functions to help find the beginning and end of an array.

begin (array name)
end (array name)

begin returns a pointer to the first, and end returns a pointer one past the last element in the given array:

## Generic Programming
## An overview of C++

**Templatized array**

In between arrays and vectors – This is an array wrapped around with an object which gives it more functionalities than the C-type arrays.

Statically allocated

Remembers the size (array) – we can use the size() member function

**This is passed by value to a function**

Example

## Generic Programming
## An overview of C++

Dynamic memory management

Recommended to use new and delete rather than

malloc () and free()

**new** operator – to allocate memory

    new data_type

**delete** operator – to free (de-allocate)memory

    delete pvalue; // Release memory pointed to by pvalue.

The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

## Generic Programming

## An overview of C++

structure in C++

Can have member functions also as part of structure definition.

All structure members are public; no access specifiers: Example

**Class**

Access private members through public get and set methods:

SetGet.cpp

Constructors and Destructor

Types of constructors:

Default: Example

Parameterized: Example

Copy - takes a single argument which is an object of the same class. Example Example2 Example3

Dynamic Example

## Generic Programming
### An overview of C++

Delegating constructor: Example

Destructor has the same name as the class with prefix tilde(~) operator and it cannot be overloaded.

**Initializer list**
Initializer List is used in initializing the data members of a class.
The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon.
Example

A copy constructor in C++ is further categorized into two types:
1. Default Copy Constructor
2. User-defined Copy Constructor Example

## An overview of C++

Operator overloading:
Example1:
Example2

Friend function
A friend function in C++ is a function that is declared outside (or inside) a class and is capable of accessing the private and protected members of the class.

Example

## Generic Programming

## An overview of C++

**Characteristics of Friend Function in C++**

1. The function is not in the ‘scope’ of the class to which it has been declared a friend.
2. Friend functionality is not restricted to only one class
3. Friend functions can be a member of a class or a function that is declared outside the scope of class.
4. <u>It cannot be invoked using the object as it is not in the scope of that class</u>.
5. We can invoke it like any normal function of the class.
6. Friend functions <u>have objects as arguments</u>.
7. It cannot access the member names directly and has to use dot membership operator and use an object name with the member name.
8. We can declare it either in the ‘public’ or the ‘private’ part.

## Generic Programming

## An overview of C++

Friend Class is a class that can access both private and protected variables of the class in which it is declared as a friend, just like a friend function. Classes declared as friends to any other class will have all the member functions as friend functions to the friend class. Friend functions are used to link both these classes.

## Generic Programming

## An overview of C++

**Operator functions**

An operator function can be either a nonstatic member function, or a nonmember function with at least one parameter that has class, reference to class, enumeration, or reference to enumeration type. You cannot change the precedence, grouping, or the number of operands of an operator.

What is an operator function?

A function which defines additional tasks to an operator or which gives special meaning to an operator is called an operator function.

## Generic Programming

## An overview of C++

·The general form of operator function is:

```
return-type class-name :: operator op (argument list)
{
        // Function body
}
```

return-type is the value returned by the specified operation and op is the operator being overloaded. **operator** is a keyword.

<u>Operator functions must be either member functions (non-static) or friend functions.</u>

<u>Not all operators can be overloaded.</u>

## An overview of C++

***Operator overloading is a compile-time polymorphism***. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

The main idea behind "Operator overloading" is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

## Generic Programming

## An overview of C++

Conversion functions

Conversion functions convert a value from one datatype to another.

User-defined data types are designed by the user to suit their requirements, the compiler does not support automatic type conversions for such data types; therefore, the users need to design the conversion routines by themselves if required.

Conversion of primitive data type to user-defined type

Conversion of class object to primitive data type

Conversion of one class type to another class type

Example1 Example2 Example3

## An overview of C++

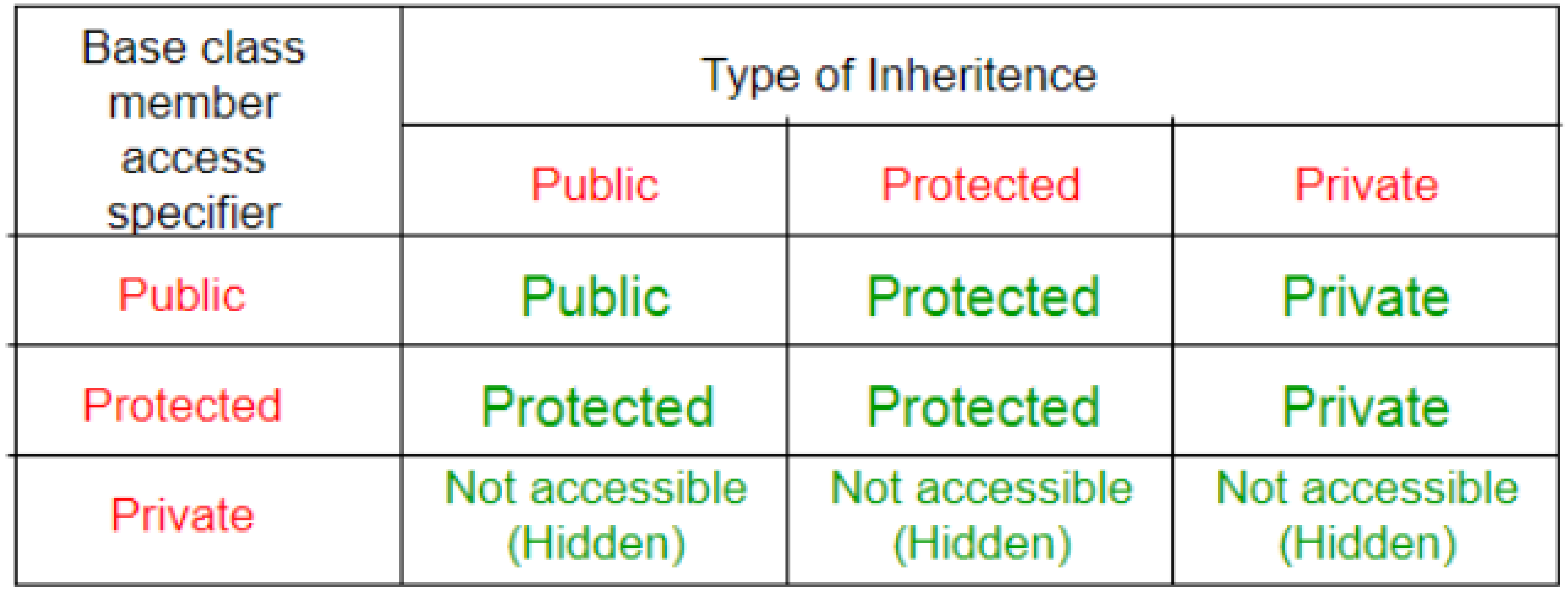
**Inheritance**

**Syntax**:
class <derived_class_name> : <access-specifier> <base_class_name> { //body }Where
class — keyword to create a new class
derived_class_name — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class

# Generic Programming

## An overview of C++

·The table below summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:



| Base class member access specifier | Type of Inheritence | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not accessible (Hidden) | Not accessible (Hidden) | Not accessible (Hidden) |

## Generic Programming

## An overview of C++

**Constructors and Destructors in Inheritance**

Parent class  constructors and destructors are accessible to the child class; hence when we create an object for the child class, constructors and destructors of both parent and child class get executed.

In the case of the default constructor, it is implicitly accessible from parent to the child class; but parameterized constructors are not accessible to the derived class automatically; for this reason, an explicit call has to be made in the child class constructor to access the parameterized constructor of the parent class to the child class using the following syntax

<class_name>:: constructor(arguments) or delegate

Example1 Example2 Example3

## Generic Programming

## An overview of C++

**Function overriding**

**Function overriding in C++** is termed as the redefinition of base class function in its derived class with the same signature i.e. return type and parameters. It falls under the category of Runtime Polymorphism.

Example Example2 Example3

**Virtual functions**

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**.

***Note**: In C++ what calling a virtual functions means is that; if we call a member function then it could cause a different function to be executed instead depending on what type of object invoked it. Because overriding from derived classes hasn't happened yet, the virtual call mechanism is disallowed in constructors.*

## An overview of C++

**What is the use?**

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing the kind of derived class object.

Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class.

Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer

Example1 Example2

## An overview of C++

**Rules for Virtual Functions**

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions <u>should be accessed using a pointer or reference of base class type to achieve runtime polymorphism</u>.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), <u>in that case, the base class version of the function is used</u>.
6. A class may have a <u>virtual destructor</u> but it cannot have a virtual constructor.

## Generic Programming

### An overview of C++

A pure virtual function is a function that must be overridden in a derived class and need not be defined in the base class.

**Abstract class**

A class is abstract if it has at least one pure virtual function**.**

Example1 Example2 Example3

**Virtual Destructor**

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Example1 Example2

## Generic Programming

## An overview of C++

**Multiple Inheritance**

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. The constructors of inherited classes are called in the same order in which they are inherited. The destructors are called in reverse order of constructors.

Example1 Example2

**Diamond problem and Virtual base classes**

**Example1** **Example2**

**Typecasting**

Type casting refers to the conversion of one data type to another in a program. Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user. Type Casting is also known as Type Conversion.

**Upcasting and Downcasting**

## Generic Programming

## An overview of C++

### Exception handling

C++ provides the following specialized keywords for this purpose:
***try***: Represents a block of code that can throw an exception.
***catch***: Represents a block of code that is executed when a particular exception is thrown.
***throw***: Used to throw an exception. Also used to list the exceptions that a function throws but doesn’t handle itself.

## An overview of C++

**Smart pointer**

The idea is to take a class with a pointer, destructor, and overloaded operators like ***** and **->**. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or the reference count can be decremented).

Example

Does this work only for int pointers? Do we need to write one class per type of pointer?

No, use templates.

Example2

# Generic Programming

## An overview of C++

**Explicit Keyword in C++** is used to mark constructors to not implicitly convert types in C++. It is optional for constructors that take exactly one argument and work on constructors(with a single argument) since those are the only constructors that can be used in typecasting.

Example1

Example2

Example3

## Generic Programming
## An overview of C++

**A few other topics**
C++ handles object copying and assignment through two functions called copy constructors and assignment operators.

While C++ will automatically provide these functions if you don't explicitly define them, in many cases you'll need to manually control how your objects are duplicated.

Always remember that the assignment operator is called only when assigning an existing object a new value. Otherwise, you're using the copy constructor.

Sample program: CopyAssignment.cpp

The above program shows the difference between a simple assignment v/s Copy Assignment.

## Generic Programming

## An overview of C++

rvalue references are a new category of reference variables that can bind to *rvalues*. Rvalues are slippery entities, such as **temporaries** and literal values;

Technically, an rvalue is an unnamed value that exists only during the evaluation of an expression. For example, the following expression produces an rvalue:
x+(y*z); // A C++ expression that produces a temporary

C++ creates a temporary (an rvalue) that stores the result of y*z, and then adds it to x. Conceptually, this rvalue evaporates by the time you reach the semicolon at the end of the full expression.

A declaration of an rvalue reference looks like this:
std::**string&& rrstr**; //C++11 rvalue reference variable
The traditional reference variables of C++ e.g.,
std::string& ref; are now called *lvalue references*

## Generic Programming
## An overview of C++

In C++03, there were costly and unnecessary deep copies which happened implicitly when objects were passed by value.

In C++11, the resources of the objects can be moved from one object to another rather than copying the whole data of the object to another. This can be done by using “move semantics” in C++11.

**Move semantics** points the other object to the already existing object in the memory. It avoids the instantiation of unnecessary temporary copies of the objects by giving the resources of the already existing object to the new one and safely taking from the existing one. Taking resources from the old existing object is necessary to prevent more than one object from having the same resources.

Move semantics uses move constructor and r-value references.

## Generic Programming
## An overview of C++

- The following example shows how the move operation works in contrast to the copy operation (with move constructor and copy constructor)

Source program: MoveExample1.cpp

The working of the “move assignment” operator is shown in the following example

Move_Assignment.cpp

## Generic Programming

## Need for templates

Let us look at one implementation of qsort for different types of variables:

Quicksort algorithm is here.

The quicksort algorithm needs to compare and swap any two elements. However, since we don't know their type, the implementation cannot do this directly. The solution is to rely on **callbacks**, which are functions passed as arguments that will be invoked when necessary.

One possible implementation is here

In order to invoke the quicksort function, we need to provide implementations for these comparisons and swapping functions for each type of array that we pass to the function.

## Generic Programming
### Need for templates

**Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively, sort two subarrays with Quicksort.

**Combine:** Combine the already sorted array.

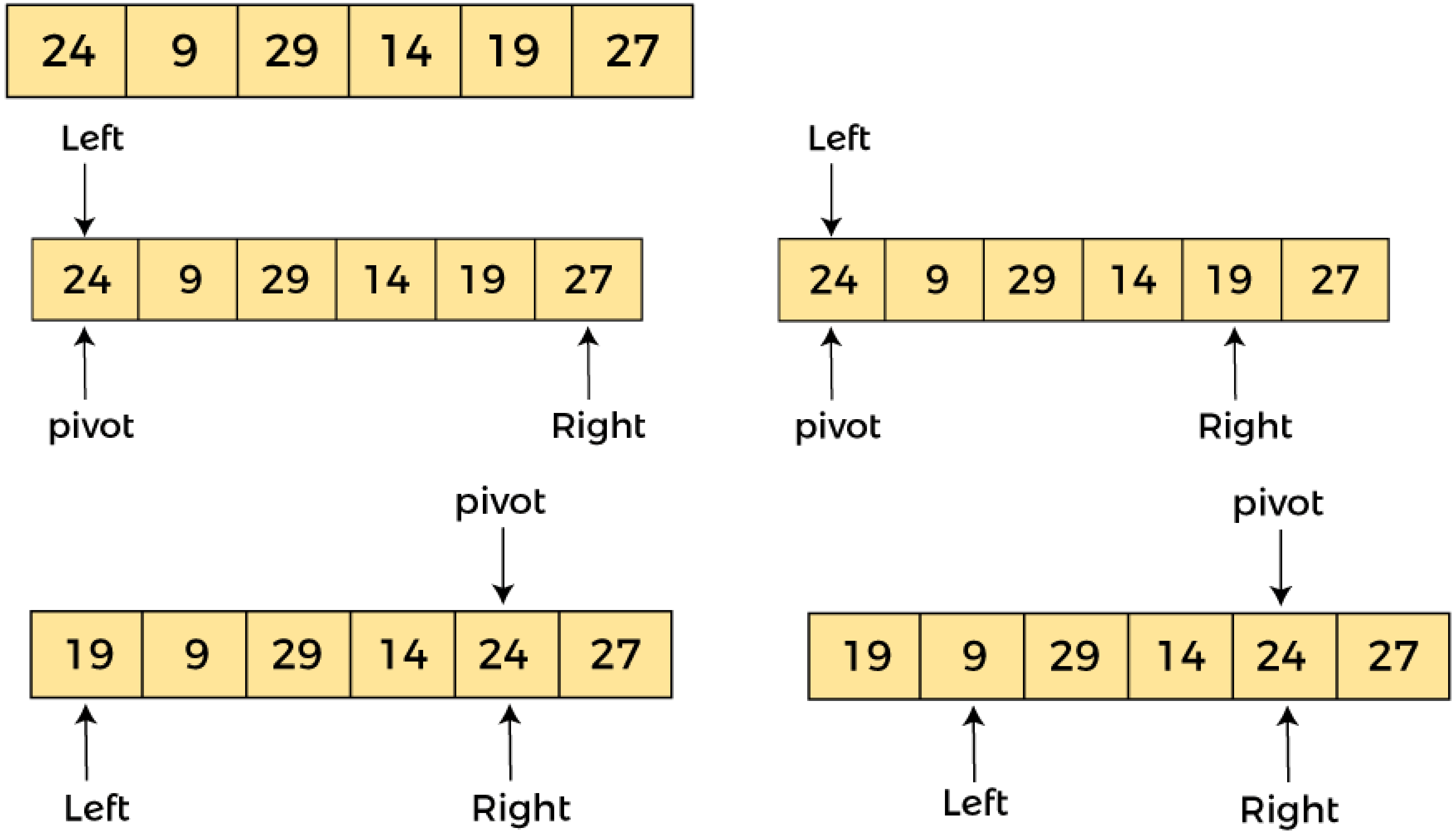
Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

## Generic Programming

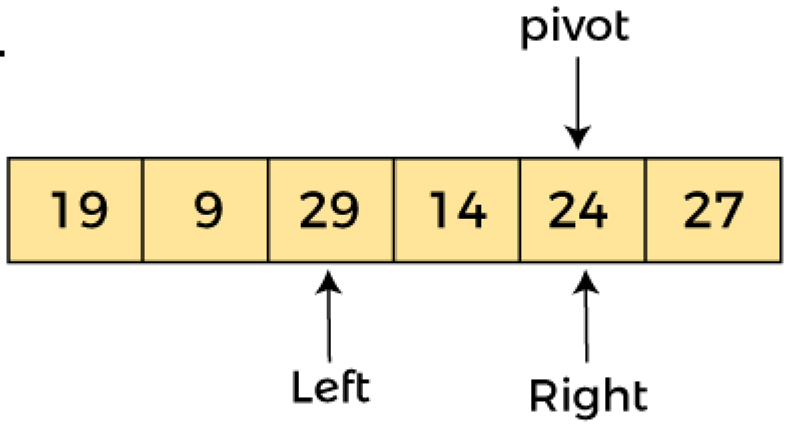
### Need for templates

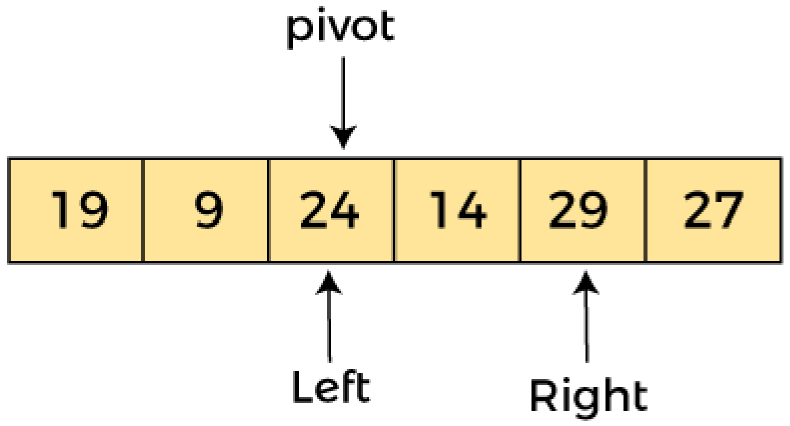
. This is the input array

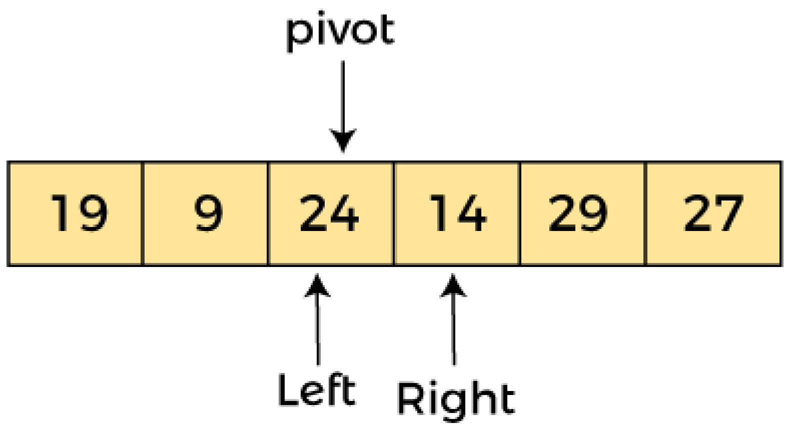


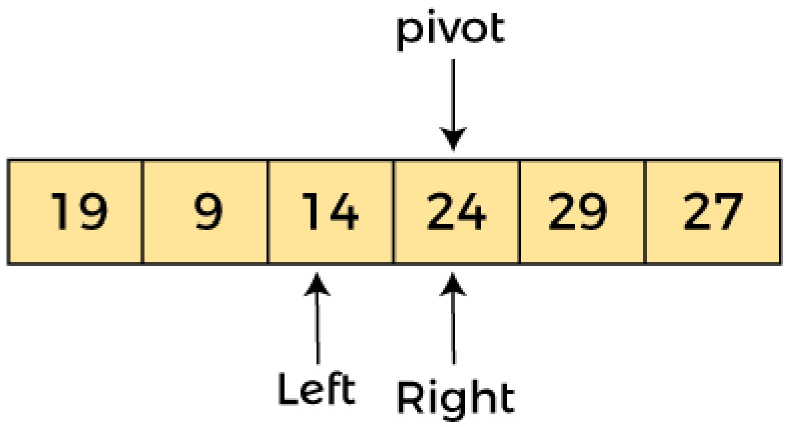
## Generic Programming

### Need for templates …





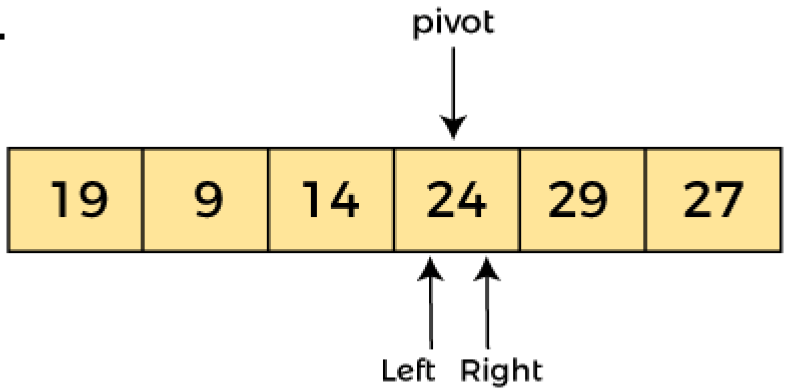


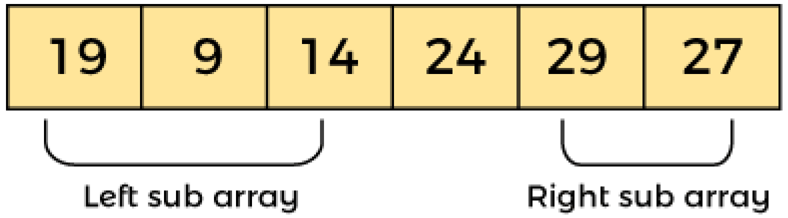


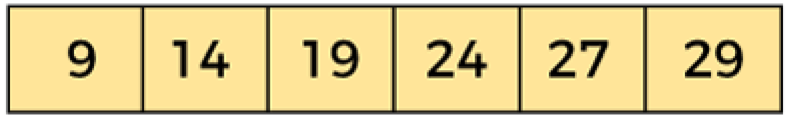
## Generic Programming

### Need for templates …

.







## Generic Programming

## Need for templates …

·The qsort program implemented through templates is here.

Another example is here
An implementation of the vector class could be this:
vector_ex.cpp

Let's consider the case of a variable that holds the new line character. You may need the same constant for a different encoding, such as wide string literals, UTF-8, and so on? You can have multiple variables, having different names, such as in the following example:

```
constexpr wchar_t NewLineW = L'\n';
constexpr char8_t NewLineU8 = u8'\n';
constexpr char16_t NewLineU16 = u'\n';
constexpr char32_t NewLineU32 = U'\n';
```

The implementation using templates is here.

## Generic Programming
## Template terminology

·**Template terminology**
**Function template** is the term used for a templated function. An example is the max template seen previously.

**Class template** is the term used for a templated class (which can be defined either with the class, struct, or union keyword). An example is the vector class we wrote in the previous section.

**Variable template** is the term used for templated variables, such as the NewLine template from the previous section.

**Alias template** is the term used for templated type aliases (to be covered later)

## Generic Programming
### Template terminology

·Templates are parameterized with one or more parameters (in the examples we have seen so far, there was a single parameter). These are called **template parameters** and can be of three categories:
**Type template parameters**, such as in template<typename T>, where the parameter represents a type specified when the template is used.
**Non-type template parameters**, such as in template<size_t N> or template<auto n>, where each parameter must have a structural type, which includes integral types, floating-point types (as for C++20), pointer types, enumeration types, lvalue reference types, and others (i.e., basic or derived data types)
**Template template parameters**, such as in template<typename K, typename V, template<typename> typename C>, where the type of a parameter is another template.

## Generic Programming
## Template terminology

·Template specialization

In many cases when working with templates, you'll write one generic version for all possible data types and leave it at that--every vector may be implemented in exactly the same way. The idea of template specialization is to override the default template implementation to handle a particular type in a different way.

For instance, while most vectors might be implemented as arrays of the given type, you might decide to save some memory and implement vectors of bools as a vector of integers with each bit corresponding to one entry in the vector. So you might have two separate vector classes.

The first class would look like:  FirstClass.cpp

## Generic Programming
### Template terminology

But when it comes to bools, you might not really want to do this because most systems are going to use 16 or 32 bits for each boolean type even though all that's required is a single bit. So we might make our boolean vector look a little bit different by representing the data as an array of integers whose bits we manually manipulate.

To do this, we still need to specify that we're working with something akin to a template, but this time the list of template parameters will be empty:

template <>
and the class name is followed by the specialized type: class className<type>. In this case, the template would look like this:

SecondClass.cpp

## Generic Programming

## Template terminology

It would be perfectly reasonable <u>if the specialized version of the vector class had a different interface</u> (set of public methods) than the generic vector class--although they're both vector templates, they don't share any interface or any code.

It's worth pointing out that the salient reason for the specialization in this case was to allow for a more space-efficient implementation, but you could think of other reasons why this might come in handy--for instance, if you wanted to add extra methods to one templated class based on its type, but not to other templates.

There are two forms of specialization:
**Explicit (full) specialization**: This is a specialization of a template <u>when all the template arguments are provided</u>. When you instantiate a template with a given set of template arguments, the compiler generates a new definition based on those template arguments. You can override this behavior of definition generation. <u>You can instead specify the definition the compiler uses for a given set of template arguments</u>. This is called *explicit specialization* (to be discussed, in detail, later).

## Generic Programming
## Template terminology

**Partial specialization**: This is an alternative implementation provided for only some of the template parameters. When you instantiate a class template, the compiler creates a definition based on the template arguments you have passed. Alternatively, if all those template arguments match those of an explicit specialization, the compiler uses the definition defined by the explicit specialization.

A *partial specialization* is a generalization of explicit specialization. An explicit specialization only has a template argument list. <u>A partial specialization has both a template argument list and a template parameter list</u>. The compiler uses the partial specialization if its template argument list matches a subset of the template arguments of a template instantiation. The compiler will then generate a new definition from the partial specialization with the rest of the unmatched template arguments of the template instantiation.

A parameter is a variable in a function definition. It is a placeholder and hence does not have a concrete value.

An argument is a value passed during function invocation.

## Generic Programming
## Template terminology

What is template instantiation?
The process of generating code from a template by the compiler is called **template instantiation**. This happens by substituting the template arguments for the template parameters used in the definition of the template. For instance, in the example where we used vector<int>, the compiler substituted the int type in every place where T appeared.

Alternatively

The act of creating a new definition of a **function**, **class**, or **member of a class** from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation to handle a specific set of template arguments is called a *specialization*.

## Generic Programming
## Template terminology

- Template instantiation can have two forms:

**Implicit instantiation**:
This occurs when the compiler instantiates a template due to its use in code. This happens only for those combinations or arguments that are in use. For instance, if the compiler encounters the use of vector<int> and vector<double>, it will instantiate the vector class template for the types **int** and **double** and nothing more.

**Explicit instantiation**:
This is a way to explicitly tell the compiler what instantiations of a template to create, even if those instantiations are not explicitly used in your code. This is useful, for instance, when creating library files, because uninstantiated templates are not put into object files. They also help reduce compile times and object sizes, in ways that we will see at a later time.
Explicit instantiation includes two forms: *explicit instantiation declaration* and *explicit instantiation definition*.

## Generic Programming

## Templates – A brief history

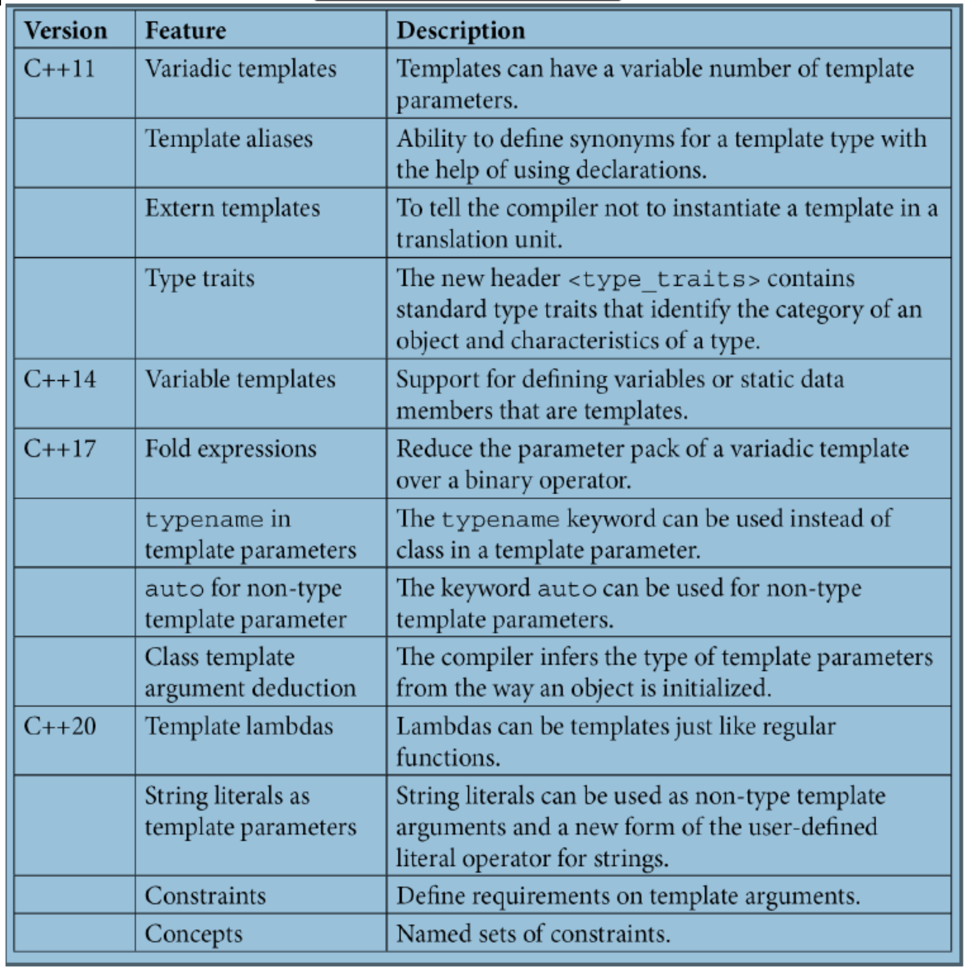
·Template metaprogramming is the C++ implementation of generic programming. This paradigm was first explored in the 1970s and the first major languages to support it were Ada and Eiffel in the first half of the 1980s. David Musser and Alexander Stepanov defined generic programming, in a paper called *Generic Programming*, in 1989, as follows:

*Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.*

This defines a paradigm of programming where algorithms are defined in terms of types that are specified later and instantiated based on their use.

# Generic Programming
## Templates – a brief history

C++ 98 – the first version when templates were formally introduced into C++.



| Version | Feature | Description |
|---|---|---|
| C++11 | Variadic templates | Templates can have a variable number of template parameters. |
| | Template aliases | Ability to define synonyms for a template type with the help of using declarations. |
| | Extern templates | To tell the compiler not to instantiate a template in a translation unit. |
| | Type traits | The new header `<type_traits>` contains standard type traits that identify the category of an object and characteristics of a type. |
| C++14 | Variable templates | Support for defining variables or static data members that are templates. |
| C++17 | Fold expressions | Reduce the parameter pack of a variadic template over a binary operator. |
| | `typename` in template parameters | The `typename` keyword can be used instead of class in a template parameter. |
| | `auto` for non-type template parameter | The keyword `auto` can be used for non-type template parameters. |
| | Class template argument deduction | The compiler infers the type of template parameters from the way an object is initialized. |
| C++20 | Template lambdas | Lambdas can be templates just like regular functions. |
| | String literals as template parameters | String literals can be used as non-type template arguments and a new form of the user-defined literal operator for strings. |
| | Constraints | Define requirements on template arguments. |
| | Concepts | Named sets of constraints. |

The keyword **constexpr** was introduced in C++11 and improved in C++14. It means constant expression. Like const , it can be applied to variables: A compiler error is raised when any code attempts to modify the value. Unlike const , constexpr can also be applied to functions and class constructors.

## Templates – pros and cons

Advantages

- Templates help us avoid writing repetitive code.
- Templates foster the creation of generic libraries providing algorithms and types, such as the standard C++ library (sometimes incorrectly referred to as the STL), which can be used in many applications, regardless of their type.
- The use of templates can result in less and better code. For instance, using algorithms from the standard library can help write less code that is likely easier to understand and maintain and also probably more robust because of the effort put into the development and testing of these algorithms.

## Generic Programming

## Templates – pros and cons

Disadvantages

- The syntax is considered complex and cumbersome, although with a little practice this should not really pose a real hurdle in the development and use of templates.
- Compiler errors related to template code can often be long and cryptic, making it very hard to identify their cause. Newer versions of the C++ compilers have made progress in simplifying these kinds of errors, although they generally remain an important issue. The inclusion of concepts in the C++20 standard has been seen as an attempt, among others, to help provide better diagnostics for compiling errors.
- They increase the compilation times because they are implemented entirely in headers. Whenever a change to a template is made, all the translation units in which that header is included must be recompiled.
- Template libraries are provided as a collection of one or more headers that must be compiled together with the code that uses them.

## Templates – pros and cons

- Another disadvantage that results from the implementation of templates in headers is that there is no information hiding. The entire template code is available in headers for anyone to read. Library developers often resort to the use of namespaces with names such as detail or details to contain code that is supposed to be internal for a library and should not be called directly by those using the library.
- They could be harder to validate since code that is not used is not instantiated by the compiler. It is, therefore, important that when writing unit tests, good code coverage must be ensured. This is especially the case for libraries.

**Note**

Templates need to be instantiated by the compiler before actually compiling them into object code. This instantiation can only be achieved if the template arguments are known. Now imagine a scenario where a template function is declared in a.h, defined in a.cpp and used in b.cpp. When a.cpp is compiled, it is not necessarily known that the upcoming compilation b.cpp will require an instance of the template, let alone which specific instance would that be. For more header and source files, the situation can quickly get more complicated.

## Generic Programming
### Defining function templates

Function templates are defined in a similar way to regular functions, except that the function declaration is preceded by the keyword **template** followed by a list of template parameters between angle brackets. The following is a simple example of a function template:

```
template <typename T>
T add(T const a, T const b)
{
		return a + b;
}
```

This function has two parameters, called a and b, both of the same T type. This type is listed in the template parameters list, introduced with the keyword **typename** or **class**. This function does nothing more than add the two arguments and returns the result of this operation, which should have the same T type.

## Generic Programming

## Defining function templates

Function templates are only blueprints for creating actual functions and only exist in source code. Unless explicitly called in your source code, the function templates will not be present in the compiled executable. However, when the compiler encounters a call to a function template and is able to match the supplied arguments and their types to a function template's parameters, it generates an actual function from the template and the arguments used to invoke it. To understand this, let's look at some examples:

```
auto a = add(42, 21);
```

In this snippet, we call the add function with two int parameters, 42 and 21. The compiler is able to deduce the template parameter T from the type of the supplied arguments, making it unnecessary to explicitly provide it. However, the following two invocations are also possible, and, in fact, identical to the earlier one:

```
auto a = add<int>(42, 21);
auto a = add<>(42, 21);
```

## Generic Programming
### Defining function templates

From this invocation, the compiler will generate the following function (keep in mind that the actual code may differ for various compilers):

```
int add(const int a, const int b)
{
		return a + b;
}
```

However, if we change the call to the following form, we explicitly provide the argument for the template parameter T, as the short type:

```
auto b = add<short>(42, 21);
```

In this case, the compiler will generate another instantiation of this function, with short instead of int. This new instantiation would look as follows:

```
short add(const short a, const short b)
{
		return static_cast<short>(a + b);
}
```

## Generic Programming
## Defining function templates

**static_cast** - It is a compile-time cast. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

```
static_cast <dest_type> (source);
```

The return value of static_cast will be of dest_type.

If the type of the two parameters is ambiguous, the compiler will not be able to deduce them automatically. This is the case with the following invocation:

```
auto d = add(41.0, 21);
```

In this example, 41.0 is a double but 21 is an int. The add function template has two parameters of the same type, so the compiler is not able to match it with the supplied arguments and will issue an error. To avoid this, and suppose you expected it to be instantiated for double, you have to specify the type explicitly, as shown in the following snippet:

```
auto d = add<double>(41.0, 21);
```

## Generic Programming
## Defining function templates

As long as the two arguments have the same type and the + operator is available for the type of the arguments, you can call the function template add in the ways shown previously. However, if the + operator is not available, then the compiler will not be able to generate an instantiation, even if the template parameters are correctly resolved. This is shown in the following snippet:

```
class foo
{
        int value;
        public:
                explicit foo(int const i):value(i)
                { }
                explicit operator int() const { return value; }
};
auto f = add(foo(42), foo(41));
```

## Generic Programming

### Defining function templates

In this case, the compiler will issue an error that a binary + operator is not found for arguments of type foo. Of course, the actual message differs for different compilers, which is the case for all errors. To make it possible to call add for arguments of type foo, you'd have to overload the + operator for this type.

A possible implementation is the following:

```
foo operator+(foo const a, foo const b)
{
        return foo((int)a + (int)b);
}
```

## Generic Programming
## Defining function templates

All the examples that we have seen so far represented templates with a single template parameter. However, a template can have any number of parameters and even a variable number of parameters. The next function is a function template that has two type template parameters:

```
template <typename Input, typename Predicate>
int count_if(Input start, Input end, Predicate p)
{
        int total = 0;
        for (Input i = start; i != end; i++)
        {
                if (p(*i))
                        total++;
        }
        return total;
}
```

## Generic Programming

## Defining function templates

- This function takes two input iterators to the start and end of a range and a predicate and returns the number of elements in the range that match the predicate. This function, at least conceptually, is very similar to the std::count_if general-purpose function from the <algorithm> header in the standard library and you should always prefer to use standard algorithms over hand-crafted implementations.

However, for the purpose of this topic, this function is a good example to help you understand how templates work.

**Note**:
The Predicate parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing the corresponding iterator returns a value testable as true.

## Generic Programming

## Defining function templates

We can use the count_if function as follows:

```
int main()
{
        int arr[]{ 1,1,2,3,5,8,11 };
        int odds = count_if(std::begin(arr), std::end(arr),
                        [](int const n) { return n % 2 == 1; });
        std::cout << odds << '\n';
}
```

Again, there is no need to explicitly specify the arguments for the type template parameters (the type of the input iterator and the type of the unary predicate) because the compiler is able to infer them from the call.

The source program is here.

## Generic Programming
### Class Templates

A class template starts with the keyword **template** followed by template parameter(s) inside <> which is followed by the class declaration.

```
template <class T>
class className {
  private:
    T var;
    ... .. ...
  public:
    T functionName(T arg);
    ... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used, and class is a keyword.

Inside the class body, a member variable var and a member function functionName() are both of type T;

## Generic Programming
## Class Templates

Creating a Class Template Object

Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax

```
className<dataType> classObject;
```

For example,

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Sample program is here.

# Generic Programming
## Class Templates

Class templates can be declared without being defined and used in contexts where incomplete types are allowed, such as the declaration of a function, as shown here:

```
template <typename T>
class wrapper;
void use_foo(wrapper<int>* ptr);
```

However, a class template must be defined at the point where the template instantiation occurs; otherwise, the compiler will generate an error. This is exemplified with the following snippet:

```
template <typename T>
class wrapper; // OK
void use_wrapper(wrapper<int>* ptr); // OK
int main()
{
		wrapper<int> a(42); // error, incomplete type
		use_wrapper(&a);
}
```

## Generic Programming
### Class Templates

```
template <typename T>
class wrapper
{
        // template definition
};
void use_wrapper(wrapper<int>* ptr)
{
        std::cout << ptr->get() << '\n';
}
```

When declaring the use_wrapper function, the class template wrapper is only declared, but not defined. However, incomplete types are allowed in this context, which makes it all right to use wrapper<T> at this point. However, in the main function we are instantiating an object of the wrapper class template. This will generate a compiler error because at this point the definition of the class template must be available. To fix this particular example, we'd have to move the definition of the main function to the end, after the definition of wrapper and use_wrapper.

## Generic Programming
## Class Templates

In this example, the class template was defined using the class keyword. However, in C++ there is little difference between declaring classes with the class or struct keyword:
• With struct, the default member access is public, whereas using class is private.
• With struct, the default access specifier for base-class inheritance is public, whereas using class is private.
You can define class templates using the struct keyword the same way we did here using the class keyword. The differences between classes defined with the struct or the class keyword are also observed for class templates defined with the struct or class keyword.
Classes, whether they are templates or not, may contain member function templates too. The way these are defined is discussed next.

## Generic Programming

## Member function templates

**Defining member function templates**

It is possible to define member function templates too, both in non-template classes and class templates. To understand the differences, let's start with the following example:

```
template <typename T>
class composition
{
public:
        T add(T const a, T const b)
        {
                return a + b;
        }
};
```

The composition class is a class template. It has a single member function called add that uses the type parameter T. This class can be used as follows:

```
composition<int> c;
c.add(41, 21);
```

## Generic Programming

## Member function templates

We first need to instantiate an object of the composition class. Notice that we must explicitly specify the argument for the type parameter T because the compiler is not able to figure it out by itself (there is no context from which to infer it).

When we invoke the add function, we just provide the arguments. Their type, represented by the T type template parameter that was previously resolved to int, is already known.

A call such as c.add<int>(42, 21) would trigger a compiler error. The **add** function is not a function template, but a regular function that is a member of the composition class template.

## Generic Programming
## Member function templates

Let us change the composition class slightly, but significantly. Let's see the definition first:

```
class composition
{
        public:
        template <typename T>
        T add(T const a, T const b)
        {
                return a + b;
        }
};
```

This time, composition is a non-template class. However, the add function is a function template.

## Generic Programming

### Member function templates

- Therefore, to call this function, we must do the following:

```
composition c;
c.add<int>(41, 21);
```

The explicit specification of the int type for the T type template parameter is redundant since the compiler can deduce it by itself from the arguments of the call.

However, it was shown here to better comprehend the differences between these two implementations.

## Generic Programming

## Member function templates

We can also have member function templates of class templates. In this case, however, the template parameters of the member function template must differ from the template parameters of the class template; otherwise, the compiler will generate an error.

Let's return to the wrapper class template example and modify it as follows:

```
template <typename T>
class wrapper
{
        public:
                wrapper(T const v) :value(v)
                {}
        T const& get() const { return value; }
        template <typename U>
        U as() const
        {
                return static_cast<U>(value);
        }
        private:
                T value;
};
```

## Generic Programming

## Member function templates

As you can see here, this implementation features one more member, a function called as. This is a function template and has a type template parameter called U. This function is used to cast the wrapped value from a type T to a type U, and return it to the caller.

We can use this implementation as follows:

```
wrapper<double> a(42.0);
auto d = a.get(); // double
auto n = a.as<int>(); // int
```

Arguments for the template parameters were specified when instantiating the wrapper class (double) – although in C++17 this is redundant, and when invoking the as function (int) to perform the cast.

## Generic Programming
## Member function templates

Defining a Class Member Outside the Class Template

We can do this with the following code:

```
template <class T>
class ClassName {
    ... .. ...
    // Function prototype
    returnType functionName();
};

// Function definition
template <class T>
returnType ClassName<T>::functionName() {
    // code
}
```

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.     An example is here.

# Generic Programming
## Template parameters

So far, we have seen multiple examples of templates with one or more parameters. In all these examples, the parameters represented types supplied at instantiation, either explicitly by the user, or implicitly by the compiler when it could deduce them. These kinds of parameters are called **type template parameters**. However, templates can also have **non-type template parameters** and **template template parameters**.

**Type template parameters**
As already mentioned, these are parameters representing types supplied as arguments during the template instantiation. They are introduced either with the **typename** or the **class** keyword. There is no difference between using these two keywords. A type template parameter can have a default value, which is a type. This is specified the same way you would specify a default value for a function parameter.

Examples for these are shown in the following snippet:

```
template <typename T>
class wrapper { /* ... */ };
template <typename T = int>
class wrapper { /* ... */ };
```

## Generic Programming
### Template parameters

·The name of the type template parameter can be omitted, which can be useful in forwarding declarations:

```
template <typename>
class wrapper;
template <typename = int>
class wrapper;
```

C++11 has introduced variadic templates, which are templates with a variable number of arguments. A template parameter that accepts zero or more arguments is called a **parameter pack**.

A **type template parameter pack** has the following form:

```
template <typename... T>
class wrapper { /* ... */ };
```

## Generic Programming

## Template parameters

**Variadic templates and Concepts and Constraints will be discussed later.**

# Generic Programming
## Template parameters

**Non-type template parameters**
Template arguments don't always have to represent types. They can also be compile-time expressions, such as constants, addresses of functions or objects with external linkage, or addresses of static class members. Parameters supplied with compile-time expressions are called **non-type template parameters**. This category of parameters can only have a **structural type**. The following are the structural types:

- Integral types
- Floating-point types, as of C++20
- Enumerations
- Pointer types (either to objects or functions)
- Pointer to member types (either to member objects or member functions)
- lvalue reference types (either to objects or functions)
- A literal class type that meets the following requirements:

. All base classes are public and non-mutable.
. All non-static data members are public and non-mutable.
. The types of all base classes and non-static data members are also structural types or arrays thereof.

cv-qualified forms of these types can also be used for non-type template parameters. Non-type template parameters can be specified in different ways.

An example program is here.

## Generic Programming
## Template parameters

·The possible forms are shown in the following snippet:

```
template <int V>
class foo { /*...*/ };

template <int V = 42>
class foo { /*...*/ };

template <int... V>
class foo { /*...*/ };
```

In all these examples, the type of the non-type template parameters is int. The first and second examples are similar, except that in the second example a default value is used. The third example is significantly different because the parameter is actually a parameter pack.

## Generic Programming
### Template parameters

.To understand non-type template parameters better, let's look at the following example, where we sketch a fixed-size array class, called buffer:

```
template <typename T, size_t S>
class buffer
{
        T data_[S];
        public:
                constexpr T const * data() const { return data_; }
                constexpr T& operator[](size_t const index)
                {
                        return data_[index];
                }

        constexpr T const & operator[](size_t const index) const
        {
                return data_[index];
        }
};
```

This buffer class holds an internal array of S elements of type T. Therefore, S needs to be a compile-type value.

## Generic Programming
### Template parameters

How do you instantiate this class?

```
buffer<int, 10> b1;
buffer<int, 2*5> b2;
```

These two definitions are equivalent, and both b1 and b2 are two buffers holding 10 integers. Moreover, they are of the same type, since 2*5 and 10 are two expressions evaluated to the same compile-time value.

You can easily check this with the following statement:

```
static_assert(std::is_same_v<decltype(b1), decltype(b2)>);
```

This is not the case anymore, for the type of the b3 object which is declared as follows:

```
buffer<int, 3*5> b3;   //Template Fundamentals
```

In this example, b3 is a buffer holding 15 integers, which is different from the buffer type from the previous example that held 10 integers.

## Generic Programming
### Template parameters

Conceptually, the compiler generates the following code:

```
template <typename T, size_t S>
class buffer
{
        T data_[S];
        public:
                constexpr T* data() const { return data_; }
                constexpr T& operator[](size_t const index)
                {
                        return data_[index];
                }

        constexpr T const & operator[](size_t const index) const
        {
                return data_[index];
        }
};
```

## Generic Programming

### Template parameters

- This is the code for the primary template but there are also a couple of specializations shown next:

```
template<> class buffer<int, 10>
{
        int data_[10];
        public:
                constexpr int * data() const;
                constexpr int & operator[](const size_t index);
                constexpr const int & operator[](
                const size_t index) const;
};

template<> class buffer<int, 15>
```

## Generic Programming
### Template parameters

C++ Class Templates With Multiple Parameters

In C++, we can use multiple template parameters and even use default arguments for those parameters. For example,

```
template <class T, class U, class V = int>
class ClassName {
  private:
    T member1;
    U member2;
    V member3;
    ... .. ...
  public:
    ... .. ...
};
```

## Generic Programming
## Template parameters

**Template template parameters**
We are referring to a category of template parameters that are themselves templates. These can be specified similarly to type template parameters, with or without a name, with or without a default value, and as a parameter pack with or without a name.

Sample code is here.

## Generic Programming
## Template parameters

**Default template arguments**

Default template arguments are specified similarly to default function arguments, in the parameter list after the equal sign. The following rules apply to default template arguments:

• They can be used with any kind of template parameters with the exception of parameter packs.

• If a default value is specified for a template parameter of a class template, variable template, or type alias, then all subsequent template parameters must also have a default value. The exception is the last parameter if it is a template parameter pack.

• If a default value is specified for a template parameter in a function template, then subsequent template parameters are not restricted to also have a default value.

• In a function template, a parameter pack may be followed by more type parameters only if they have default arguments or their value can be deduced by the compiler from the function arguments.

• They are not allowed in declarations of friend class templates.

• They are allowed in the declaration of a friend function template only if the declaration is also a definition and there is no other declaration of the function in the same translation unit.

• They are not allowed in the declaration or definition of an explicit specialization of a function template or member function template.

## Generic Programming
### Template parameters

**Examples:**

```
template <typename T = int>
class foo { /*...*/ };
template <typename T = int, typename U = double>
class bar { /*...*/ };
```

As mentioned previously, a template parameter with a default argument cannot be followed by parameters without a default argument <u>when declaring a class template </u>but this restriction does not apply to function templates. This is shown in the next snippet:

```
template <typename T = int, typename U>
class bar { }; // error
template <typename T = int, typename U>
void func() {} // OK
```

## Generic Programming
## Template parameters

A template may have multiple declarations (but only one definition). The default template arguments from all the declarations and the definition are merged (the same way they are merged for default function arguments).

**Example**

```
template <typename T, typename U = double>
struct foo;
template <typename T = int, typename U>
struct foo;
template <typename T, typename U>
struct foo
{
        T a;
        U b;
};
```

This is semantically equivalent to the following definition:

```
template <typename T = int, typename U = double>
struct foo
{
        T a;
        U b;
};
```

## Generic Programming

### Template parameters

However, these multiple declarations with different default template arguments cannot be provided in any order. The rules mentioned earlier still apply. Therefore, a declaration of a class template where the first parameter has a default argument and the ensuing parameters do not have one is illegal:

```
template <typename T = int, typename U>
struct foo; // error, U does not have a default argument
template <typename T, typename U = double>
struct foo;
```

Another restriction on default template arguments is that the same template parameter cannot be given multiple defaults in the same scope. Therefore, the next example will produce an error:

```
template <typename T = int>
struct foo;
template <typename T = int> // error redefinition of default parameter
struct foo {};
```

## Generic Programming

## Template parameters

When a default template argument uses names from a class, the member access restrictions are checked at the declaration, not at the instantiation of the template:

```
template <typename T>
struct foo
{
        protected:
                using value_type = T;
};
template <typename T, typename U = typename T::value_type>
struct bar
{
        using value_type = U;
};
bar<foo<int>> x;
```

When the x variable is defined, the bar class template is instantiated, but the foo::value_type typedef is protected and therefore cannot be used outside of foo. The result is a compiler error at the declaration of the bar class template. **This is the program.**

## Generic Programming
## Template instantiation

**Understanding template instantiation**
As mentioned before, templates are only blueprints from which the compiler creates actual code when it encounters their use.

The act of creating a definition for a function, a class, or a variable from the template declaration is called **template instantiation**.

This can be either **explicit**, when you tell the compiler when it should generate a definition, or **implicit**, when the compiler generates a new definition as needed.

## Generic Programming
### Template instantiation

**Implicit instantiation**

Implicit instantiation occurs when the compiler generates definitions based on the use of templates and when no explicit instantiation is present. Implicitly instantiated templates are defined in the same namespace as the template. However, the way compilers create definitions from templates may differ. This is something we will see in the following example. Let's consider this code:

```
template <typename T>
struct foo
{
        void f() {}
};
int main()
{
        foo<int> x;
}
```

## Generic Programming

## Template instantiation

Here, we have a class template called foo with a member function f. In main, we define a variable of the type foo<int> but do not use any of its members. Because it encounters this use of foo, the compiler implicitly defines a specialization of foo for the int type.

The compiler might generate a code as follows:

```
template<>

struct foo<int>
{
        inline void f();
};
```

Because the function f is not invoked in our code, it is only declared but not defined.

## Generic Programming

## Template instantiation

Should we add a call f in main, the specialization would change as follows:

```
template<>
struct foo<int>
{
        inline void f() { }
};
```

However, if we add one more function, g, with the following implementation that contains an error, <u>we will get different behaviors with different compilers</u>:

```
template <typename T>
struct foo
{
        void f() {}
        void g() {int a = "42";}
};
int main()
{
        foo<int> x;
        x.f();
}
```

## Generic Programming
## Template instantiation

·The body of g contains an error (you could also use a static_assert(false) statement as an alternative). This code compiles without any problem with some compilers (VC++), but fails with GCC. This is because VC++ ignores the parts of the template that are not used, provided that the code is syntactically correct, but the others perform semantic validation before proceeding with template instantiation.

For function templates, implicit instantiation occurs when the user code refers to a function in a context that requires its definition to exist. For class templates, implicit instantiation occurs when the user code refers to a template in a context when a complete type is required or when the completeness of the type affects the code. The typical example of such a context is when an object of such a type is constructed. However, this is not the case when declaring pointers to a class template.

## Generic Programming

## Template instantiation

Consider the following example:

```
template <typename T>
struct foo
{
        void f() {}
        void g() {}
};
int main()
{
        foo<int>* p;
        foo<int> x;
        foo<double>* q;
}
```

In this snippet, we use the same foo class template from the previous examples, and we declare several variables: p which is a pointer to foo<int>, x which is a foo<int>, and q which is a pointer to foo<double>. The compiler is required to instantiate only foo<int> at this point because of the declaration of x.

## Generic Programming
## Template instantiation

Now, let's consider some invocations of the member functions f and g as follows:

```
int main()
{
        foo<int>* p;
        foo<int> x;
        foo<double>* q;
        x.f();
        q->g();
}
```

With these changes, the compiler is required to instantiate the following:

- foo<int> when the x variable is declared
- foo<int>::f() when the x.f() call occurs
- foo<double> and foo<double>::g() when the q->g() call occurs.

On the other hand, the compiler is not required to instantiate foo<int> when the p pointer is declared nor foo<double> when the q pointer is declared. However, the compiler does need to implicitly instantiate a class template specialization when it is involved in pointer conversion.

## Generic Programming

## Template instantiation

- This is shown in the following example:

```
template <typename T>
struct control
{};
template <typename T>
struct button : public control<T>
{};
void show(button<int>* ptr)
{
        control<int>* c = ptr;
}
```

In the function show, a conversion between button<int>* and control<int>* takes place. Therefore, at this point, the compiler must instantiate button<int>.

## Generic Programming
### Template instantiation

Class template containing static members
When a class template contains static members, those members are not implicitly instantiated when the compiler implicitly instantiates the class template but only when the compiler needs their definition.

On the other hand, every specialization of a class template has its own copy of static members as exemplified in the following snippet:

```
template <typename T>
struct foo
{
        static T data;
};
```

## Generic Programming
## Template instantiation

```
template <typename T>
struct foo
{
        static T data;
};

template <typename T> T foo<T>::data = 0;
int main()
{
        foo<int> a;
        foo<double> b;
        foo<double> c;
        std::cout << a.data << '\n'; // 0
        std::cout << b.data << '\n'; // 0
        std::cout << c.data << '\n'; // 0

        b.data = 42;
        std::cout << a.data << '\n'; // 0
        std::cout << b.data << '\n'; // 42
        std::cout << c.data << '\n'; // 42
}
```

The program is here.

## Generic Programming
## Template instantiation

·The class template foo has a static member variable called **data** that is initialized after the definition of foo.

In the main function, we declare the variable a as an object of foo<int> and b and c as objects of foo<double>. Initially, all of them have the member field data initialized with 0.

However, the variables b and c share the same copy of data. Therefore, after the assignment b.data = 42, a.data is still 0, but both b.data and c.data are 42.

## Generic Programming
## Template instantiation

**Explicit instantiation**
As a user, you can explicitly tell the compiler to instantiate a class template or a function template. This is called explicit instantiation and it has two forms: **explicit instantiation definition** and **explicit instantiation declaration**.

**Explicit instantiation definition**
An explicit instantiation definition may appear anywhere in a program but after the definition of the template it refers to. The syntax for explicit template instantiation definitions takes the following forms:

• The syntax for class templates is as follows:
template class-key template-name <argument-list>
• The syntax for function templates is as follows:
template return-type name<argument-list>(parameter-list);
template return-type name(parameter-list);

## Generic Programming
## Template instantiation

As you can see, in all cases, the explicit instantiation definition is introduced with the template keyword but not followed by any parameter list. For class templates, the class-key can be any of the class, struct, or union keywords.

For both class and function templates, an explicit instantiation definition with a given argument list can only appear once in the entire program.

## Generic Programming
## Template instantiation

Example:

```
namespace ns
{
        template <typename T>
        struct wrapper
        {
                T value;
        };
        template struct wrapper<int>; // [1]
}
template struct ns::wrapper<double>; // [2]
int main() {}
```

In this snippet, wrapper<T> is a class template defined in the ns namespace. The statements marked with [1] and [2] in the code are both representing **an explicit instantiation definition**, for wrapper<int> and wrapper<double> respectively. An explicit instantiation definition can only appear in the same namespace as the template it refers to (as in [1]) to or it must be fully qualified (as in [2]).

## Generic Programming
## Template instantiation

We can write similar explicit template definitions for a function template:

```
namespace ns
{
		template <typename T>
		T add(T const a, T const b)
		{
				return a + b;
		}
		template int add(int, int); // [1]
}
template double ns::add(double, double); // [2]
int main() { }
```

This example has a striking resemblance to the first. Both [1] and [2] represent explicit template definitions for add<int>() and add<double>().

## Generic Programming
## Template instantiation

If the explicit instantiation definition is not in the same namespace as the template, the name must be fully qualified. The use of a **using** statement does not make the name visible in the current namespace. This is shown in the following example:

```
namespace ns
{
        template <typename T>
        struct wrapper { T value; };
}
using namespace ns;
template struct wrapper<double>; // error
```

The last line in this example generates a compile error because wrapper is an unknown name and must be qualified with the namespace name, as in ns::wrapper.

## Generic Programming
## Template instantiation

When class members are used for return types or parameter types, member access specification is ignored in explicit instantiation definitions.
Example:

```
template <typename T>
class foo
{
        struct bar {};

        T f(bar const arg)
        {
                return {};
        }
};
template int foo<int>::f(foo<int>::bar);
```

Both the class X<T>::bar and the function foo<T>::f() are private to the foo<T> class, but they can be used in the explicit instantiation definition shown on the last line.

## Generic Programming

## Template instantiation

Why would you tell the compiler to generate instantiation from a template? When is it useful?

The answer is that it helps distribute libraries, reduce build times, and executable sizes. If you are building a library that you want to distribute as a .lib file and that library uses templates, the template definitions that are not instantiated are not put into the library. But that leads to increased build times of your user code every time you use the library.

By forcing instantiations of templates in the library, those definitions are put into the object files and the .lib file you are distributing. As a result, your user code only needs to be linked to those available functions in the library file.

## Generic Programming
### Template instantiation

·This is what the Microsoft MSVC CRT (C Run Time) libraries do for all the stream, locale, and string classes.

The libstdc++ library does the same for string classes and others.

A problem that can arise with template instantiations is that you can end up with multiple definitions, one per translation unit (??)

If the same header that contains a template is included in multiple translation units (.cpp files) and the same template instantiation is used (let's say wrapper<int> from our previous examples), then identical copies of these instantiations are put in each translation unit. This leads to increased object sizes.

The problem can be solved with the help of explicit instantiation declarations, which we will look at next.

## Generic Programming
## Template instantiation

**Explicit instantiation declaration**

An explicit instantiation declaration (available with C++11) is the way you can tell the compiler that the definition of a template instantiation is found in a different translation unit and that a new definition should not be generated.

The syntax is the same as for explicit instantiation definitions except that the keyword **extern** is used in front of the declaration:

• The syntax for class templates is as follows:
extern template class-key template-name <argument-list>

• The syntax for function templates is as follows:
extern template return-type name<argument-list>(parameter-list);
extern template return-type name(parameter-list);

## Generic Programming
## Template instantiation

If you provide an explicit instantiation declaration but no instantiation definition exists in any translation unit of the program, then the result is a compiler warning and a linker error.

The technique is to declare an explicit template instantiation in one source file and explicit template declarations in the remaining ones. This will reduce both compilation times and object file sizes.

Refer to the following files:

wrapper.h
main.cpp
source1.cpp
source2.cpp

## Generic Programming
## Template instantiation

In the above source programs, we can see the following:

- ➢ The wrapper.h header contains a class template called wrapper<T>. On the line marked with [1] there is an explicit instantiation declaration for wrapper<int> that tells the compiler not to generate definitions for this instantiation when a source file (translation unit) including this header is compiled.

- ➢ The source1.cpp file includes wrapper.h and on the line marked with [2] contains an explicit instantiation definition for wrapper<int>. This is the only definition for this instantiation within the entire program.

- ➢ The source files source2.cpp and main.cpp are both using wrapper<int> but without any explicit instantiation definition or declaration. That is because the explicit declaration from wrapper.h is visible when the header is included in each of these files.

## Generic Programming

### Template instantiation

Alternatively, the explicit instantiation declaration could be taken away from the header file but then it must be added to each source file that includes the header and that is likely to be forgotten.

When you do explicit template declarations, keep in mind that a class member function that is **defined** within the body of the class is always considered inline and therefore it will always be instantiated. Therefore, you can only use the extern keyword for member functions that are defined outside of the class body.

## Generic Programming

### Template specialization

A program fspecialize.cpp

We’ve now provided an overloaded print function for fully specialized StaticArray<char, 14>

Although this solves the issue of making sure print() can be called with a StaticArray<char, 14>, it brings up another problem: using full template specialization means we have to explicitly define the length of the array this function will accept!

Consider the following example:
fspecialize2.cpp

## Generic Programming
## Template specialization

Calling print() with char12 will call the version of print() that takes a StaticArray<T, size>, because char12 is of type StaticArray<char, 12>, and our overloaded print() will only be called when passed a StaticArray<char, 14>.

Although we could make a copy of print() that handles StaticArray<char, 12>, what happens when we want to call print() with an array size of 5, or 22? We'd have to copy the function for each different array size. That's redundant.

Obviously full template specialization is too restrictive a solution here. The solution we are looking for is partial template specialization.

## Generic Programming
## Template specialization

**Partial template specialization**

Partial template specialization allows us to specialize classes (but not individual functions!) where some, but not all, of the template parameters have been explicitly defined.

For our challenge above, the ideal solution would be to have our overloaded print function work with StaticArray of type char, but leave the length expression parameter templated so it can vary as needed. Partial template specialization allows us to do just that!

Look at this sample program pspecialize.cpp

As can be seen here, we've explicitly declared that this function will only work for StaticArray of type char, but size is still a templated expression parameter, so it will work for char arrays of any size.

## Generic Programming
## Template specialization

Partial template specialization can only be used with classes, not template functions (functions must be fully specialized). Our void print(StaticArray<char, size> &array) example works because the print function is not partially specialized (it’s just an overloaded function using a class parameter that’s partially specialized).

See the following 2 programs:

Program1.cpp program2.cpp

What we want to do here is use a partial specialization based on whether the type is a pointer or a non-pointer.

## Generic Programming

## Template specialization

There are a couple of syntax points to notice here.

1. Our template parameter list still names T as the parameter, but the declaration now has a T * after the name of the class; this tells the compiler to match a pointer of any type with this template instead of the more general template.
2. T is now the type pointed to; it is **not** itself a pointer. For instance, when you declare a sortedVector<int *>, T will refer to the int type! This makes some sense if you think of it as a form of pattern matching where T matches the type if that type is followed by an asterisk. This does mean that you have to be a tad bit more careful in your implementation: note that vec_data is a T** because we need a dynamically sized array made up of pointers.

You might wonder if you really want your sortedVector type to work like this--after all, if you're putting them in an array of pointers, you'd expect them to be sorted by pointer type. But there's a practical reason for doing this: when you allocate memory for an array of objects, the default constructor must be called to construct each object. If no default constructor exists (for instance, if every object needs some data to be created), you're stuck needing a list of pointers to objects, but you probably want them to be sorted the same way the actual objects themselves would be!

## Generic Programming
## Template specialization

You can also partially specialize on template arguments--for instance, if you had a fixedVector type that allowed the user of the class to specify both a type to store and the length of the vector (possibly to avoid the cost of dynamic memory allocations), it might look something like this:

```
template <typename T, unsigned length>
class fixedVector { ... };
```

Then you could partially specialize for booleans with the following syntax

```
template <unsigned length>
class fixedVector<bool, length> {...}
```

Note that since T is no longer a template parameter, it's left out of the template parameter list, leaving only length. Also note that length now shows up as part of fixedVector's name (unlike when you have a generic template declaration, where you specify nothing after the name). (By the way, don't be surprised to see a template parameter that's a non-type: it's perfectly valid, and sometimes useful, to have template arguments that are integer types such as unsigned.)

## Generic Programming
### Template specialization

A final implementation detail comes up with partial specializations: how does the compiler pick which specialization to use if there are a combination of completely generic types, some partial specializations, and maybe even some full specializations?

The general rule of thumb is that the compiler will pick the most specific template specialization--the most specific template specialization is the one whose template arguments would be accepted by the other template declarations, but which would not accept all possible arguments that other templates with the same name would accept.

For instance, if you decided that you wanted a sortedVector<int *> that sorted by memory location, you could create a full specialization of sortedVector and if you declared a sortedVector<int *>, then the compiler would pick that implementation over the less-specific partial specialization for pointers. It's the most specialized since only an int * matches the full specialization, not any other pointer type such as a double *, whereas int * certainly could be a parameter to either of the other templates.

## Generic Programming
## Variable Templates

**Defining variable templates**

Variable templates were introduced in C++14 and allow us to define variables that are templates either at namespace scope, in which case they represent a family of global variables, or at class scope, in which case they represent static data members.

A variable template is declared at a namespace scope as shown in the following code snippet:

```
template<class T>
constexpr T PI = T(3.1415926535897932385L);
```

The syntax is similar to declaring a variable (or data member) but combined with the syntax for declaring templates.

## Generic Programming
## Variable Templates

How are variable templates actually helpful?
Let's consider we want to write a function template that, given the radius of a sphere, returns its volume. The volume of a sphere is 4πr^3 / 3. Therefore, a possible implementation is as follows:

```
constexpr double PI = 3.1415926535897932385L;
template <typename T>
T sphere_volume(T const r)
{
        return 4 * PI * r * r * r / 3;
}
```

In this example, PI is defined as a compile-time constant of the double type. This will generate a compiler warning if we use float, for instance, for the type template parameter T:

```
float v1 = sphere_volume(42.0f); // warning
double v2 = sphere_volume(42.0); // OK
```

## Generic Programming
### Variable Templates

A potential solution to this problem is to make PI a static data member of a template class with its type determined by the type template parameter. This implementation can look as follows:

```
template <typename T>
struct PI
{
		static const T value;
};

template <typename T>
const T PI<T>::value = T(3.1415926535897932385L);
template <typename T>
T sphere_volume(T const r)
{
		return 4 * PI<T>::value * r * r * r / 3;
}
```

## Generic Programming
### Variable Templates

This works, although the use of PI<T>::value is not ideal. It would be nicer if we could simply write PI<T>.

This is exactly what the variable template PI shown earlier allows us to do. Here it is again, with the complete solution:

```
template<class T>
constexpr T PI = T(3.1415926535897932385L);
template <typename T>
T sphere_volume(T const r)
{
        return 4 * PI<T> * r * r * r / 3;
}
```

## Generic Programming

## Variable Templates

Variable templates can be members of classes. In this case, they represent static data members and need to be declared using the static keyword.

Example

```
struct math_constants
{
        template<class T>
        static constexpr T PI = T(3.14159265358979323851L);
};

template <typename T>
T sphere_volume(T const r)
{
        return 4 * math_constants::PI<T> *r * r * r / 3;
}
```

# Generic Programming

## Variable Templates

You can declare a variable template in a class and then provide its definition outside the class. Notice that in this case, the variable template must be declared with static const and not static constexpr, since the latter one requires in-class initialization:

```
struct math_constants
{
        template<class T>
        static const T PI;
};
template<class T>
const T math_constants::PI = T(3.14159265358979323851L);
```

## Generic Programming
### Variable Templates

Variable templates are instantiated similarly to function templates and class templates. This happens either with an explicit instantiation or explicit specialization, or implicitly by the compiler. The compiler generates a definition when the variable template is used in a context where a variable definition must exist, or the variable is needed for constant evaluation of an expression.

## Generic Programming
## Alias Templates

**Defining alias templates**
In C++, an **alias** is a name used to refer to a type that has been previously defined, whether a built-in type or a user-defined type.

The primary purpose of aliases is to give shorter names to types that have a long name or provide semantically meaningful names for some types.
This can be done either with a typedef declaration or with a using declaration (the latter was introduced in C++11).

Here are several examples using typedef:

```
typedef int index_t;
typedef std::vector< std::pair<int, std::string>> NameValueList;
typedef int (*fn_ptr)(int, char);
```

## Generic Programming

### Alias Templates

```
template <typename T>
struct foo
{
        typedef T value_type;
};
```

In this example, index_t is an alias for int,
NameValueList is an alias for std::vector<std::pair<int, std::string>>,
while fn_ptr is an alias for the type of a pointer to a function that returns an int and has two parameters of type int and char.
Lastly, foo::value_type is an alias for the type template T.

## Generic Programming
### Alias Templates

Since C++11, these type aliases can be created with the help of **using declarations**, which look as follows:

```
using index_t = int;
using NameValueList =
std::vector<std::pair<int, std::string>>;
using fn_ptr = int(*)(int, char);

template <typename T>
struct foo
{
        using value_type = T;
};
```

## Generic Programming
### Alias Templates

Using declarations are now preferred over typedef declarations because they are simpler to use and are also more natural to read (from left to right). However, they have an important advantage over typedefs as they allow us to create aliases for templates.

An **alias template** is a name that refers not to a type but a family of types. Remember, a template is not a class, function, or variable but a blueprint that allows the creation of a family of types, functions, or variables.

To understand how alias templates work, let's consider the following example:

```
template <typename T>
using customer_addresses_t = std::map<int, std::vector<T>>; // [1]
struct delivery_address_t {};
struct invoice_address_t {};
```

## Generic Programming

### Alias Templates

```
using customer_delivery_addresses_t =
customer_addresses_t<delivery_address_t>; // [2]
using customer_invoice_addresses_t =
customer_addresses_t<invoice_address_t>; // [3]
```

The declaration on line [1] introduces the alias template customer_addresses_t. It's an alias for a map type where the key type is int and the value type is std::vector<T>. Since std::vector<T> is not a type, but a family of types, customer_addresses_ t<T> defines a family of types.

The using declarations at [2] and [3] introduce two type aliases, customer_delivery_addresses_t and customer_invoice_ addresses_t, from the aforementioned family of types.

## Generic Programming
### Alias Templates

Alias templates can appear at namespace or class scope just like any template declaration. On the other hand, they can neither be fully nor partially specialized.
How do you overcome this limitation?
A solution is to create a class template with a type alias member and specialize the class. Then you can create an alias template that refers to the type alias member.

Example.
Although the following is not valid C++ code, it represents the end goal we want to achieve, had the specialization of alias templates been possible:

```
template <typename T, size_t S>
using list_t = std::vector<T>;
template <typename T>
using list_t<T, 1> = T;
```

## Generic Programming
### Alias Templates

In this example, list_t is an alias template for std::vector<T> provided the size of the collection is greater than 1. However, if there is a single element, then list_t should be an alias for the type template parameter T. The way this can be actually achieved is shown in the following snippet:

```
template <typename T, size_t S>
struct list
{
		using type = std::vector<T>;
};
template <typename T>
struct list<T, 1>
{
		using type = T;
};
template <typename T, size_t S>
using list_t = typename list<T, S>::type;
```

# Generic Programming
## Alias Templates

In this example, list<T,S> is a class template that has a member type alias called T. In the primary template, this is an alias for std::vector<T>. In the partial specialization list<T,1> it's an alias for T. Then, list_t is defined as an alias template for list<T, S>::type.

The following asserts prove this mechanism works:
static_assert(std::is_same_v<list_t<int, 1>, int>);
static_assert(std::is_same_v<list_t<int, 2>, std::vector<int>>);

## Generic Programming

## Generic Lambdas and Lambda Templates

Lambd1.cpp Lambd2.cpp

Lambdas, which are formally called **lambda expressions**, are a simplified way to define function objects in the place where they are needed. This typically includes predicates or comparison functions passed to algorithms.

Lambda expressions, which were introduced in C++11, have received several updates in later versions of the standard. There are notably two:

• **Generic lambdas**, introduced in C++14, allow us to use the auto specifier instead of explicitly specifying types. This transforms the generated function object into one with a template function-call operator.

• **Template lambdas**, introduced in C++20, allow us to use the template syntax to explicitly specify the shape of the templatized function-call operator.

## Generic Programming

## Generic Lambdas and Lambda Templates

```
auto l1 = [](int a) {return a + a; }; // C++11, regular lambda
auto l2 = [](auto a) {return a + a; }; // C++14, generic lambda
auto l3 = []<typename T>(T a)
{ return a + a; }; // C++20, template lambda

auto v1 = l1(42); // OK
auto v2 = l1(42.0); // warning
auto v3 = l1(std::string{ "42" }); // error
auto v5 = l2(42); // OK
auto v6 = l2(42.0); // OK
auto v7 = l2(std::string{"42"}); // OK
auto v8 = l3(42); // OK
auto v9 = l3(42.0); // OK
auto v10 = l3(std::string{ "42" }); // OK
```

An example for Generic Lambda is shown here.

## Generic Programming

### Generic Lambdas and Lambda Templates

Here, we have three different lambdas: l1 is a regular lambda, l2 is a generic lambda, as at least one of the parameters is defined with the auto specifier, and l3 is a template lambda, defined with the template syntax but without the use of the template keyword.

We can invoke l1 with an integer; we can also invoke it with a double, but this time the compiler will produce a warning about the possible loss of data.

However, trying to invoke it with a string argument will produce a compile error, because std::string cannot be converted to int. On the other hand, l2 is a generic lambda. The compiler proceeds to instantiate specializations of it for all the types of the arguments it's invoked with, in this example int, double, and std::string.

# Generic Programming

## Generic Lambdas and Lambda Templates

The following snippet shows how the generated function object may look, at least conceptually:

```
struct __lambda_3
{
                template<typename T1>
                inline auto operator()(T1 a) const  {
                return a + a;
                }
template<>
inline int operator()(int a) const
{
                return a + a;
}
template<>
inline double operator()(double a) const
{
                return a + a;
}
template<>
inline std::string operator()(std::string a) const
{
                return std::operator+(a, a);
}
};
```

## Generic Programming

## Generic Lambdas and Lambda Templates

You can see here the primary template for the function-call operator, as well as the three specializations that we mentioned. Not surprisingly, the compiler will generate the same code for the third lambda expression, l3, which is a template lambda, only available in C++20.

How are generic lambdas and lambda templates different?

To answer this question, let's modify the previous example a bit:

```
auto l1 = [](int a, int b) {return a + b; };
auto l2 = [](auto a, auto b) {return a + b; };
auto l3 = []<typename T, typename U>(T a, U b)
{ return a + b; };
auto v1 = l1(42, 1); // OK
auto v2 = l1(42.0, 1.0); // warning
auto v3 = l1(std::string{ "42" }, '1'); // error
```

## Generic Programming

### Generic Lambdas and Lambda Templates

```
auto v4 = l2(42, 1); // OK
auto v5 = l2(42.0, 1); // OK
auto v6 = l2(std::string{ "42" }, '1'); // OK
auto v7 = l2(std::string{ "42" }, std::string{ "1" }); // OK
auto v8 = l3(42, 1); // OK
auto v9 = l3(42.0, 1); // OK
auto v10 = l3(std::string{ "42" }, '1'); // OK
auto v11 = l3(std::string{ "42" }, std::string{ "42" }); // OK
```

The new lambda expressions take two parameters. Again, we can call l1 with two integers or an int and a double (although this again generates a warning) but we can't call it with a string and char. However, we can do all these with the generic lambda l2 and the lambda template l3.

## Generic Programming

## Generic Lambdas and Lambda Templates

The code the compiler generates is identical for l2 and l3 and looks, semantically, as follows:

```
struct __lambda_4
{
        template<typename T1, typename T2>
        inline auto operator()(T1 a, T2 b) const
        {
                return a + b;
        }
        template<>
        inline int operator()(int a, int b) const
        {
                return a + b;
        }
        template<>
        inline double operator()(double a, int b) const
        {
                return a + static_cast<double>(b);
        }
```

## Generic Programming

## Generic Lambdas and Lambda Templates

```
.            template<>
             inline std::string operator()(std::string a,
             char b) const
             {
                       return std::operator+(a, b);
             }
             template<>
             inline std::string operator()(std::string a,
             std::string b) const
             {
                       return std::operator+(a, b);
             }
};
```

## Generic Programming

## Generic Lambdas and Lambda Templates

We see, in this snippet, the primary template for the function-call operator, and several full explicit specializations: for two int values, for a double and an int, for a string and a char, and for two string objects. But what if we want to restrict the use of the generic lambda l2 to arguments of the same type? This is not possible.

The compiler cannot deduce our intention and, therefore, it would generate a different type template parameter for each occurrence of the auto specifier in the parameter list. However, the lambda templates from C++20 do allow us to specify the form of the function-call operator.

## Generic Programming

## Generic Lambdas and Lambda Templates

Take a look at the following example:

```
auto l5 = []<typename T>(T a, T b) { return a + b; };
auto v1 = l5(42, 1); // OK
auto v2 = l5(42, 1.0); // error
auto v4 = l5(42.0, 1.0); // OK
auto v5 = l5(42, false); // error
auto v6 = l5(std::string{ "42" }, std::string{ "1" }); // OK
auto v6 = l5(std::string{ "42" }, '1'); // error
```

Invoking the lambda template with any two arguments of different types, even if they are implicitly convertible such as from int to double, is not possible. The compiler will generate an error. It's not possible to explicitly provide the template arguments when invoking the template lambda, such as in l5<double>(42, 1.0). This also generates a compiler error.

The decltype type specifier allows us to tell the compiler to deduce the type from an expression.

## Generic Programming

## Generic Lambdas and Lambda Templates

In C++14, we can use this in a generic lambda to declare the second parameter in the previous generic lambda expression to have the same type as the first parameter. More precisely, this would look as follows:

```
auto l4 = [](auto a, decltype(a) b) {return a + b; };
```

However, this implies that the type of the second parameter, b, must be convertible to the type of the first parameter, a. This allows us to write the following calls:

```
auto v1 = l4(42.0, 1); // OK
auto v2 = l4(42, 1.0); // warning
auto v3 = l4(std::string{ "42" }, '1'); // error
```

The first call is compiled without any problems because int is implicitly convertible to double. The second call compiles with a warning, because converting from double to int may incur a loss of data. The third call, however, generates an error, because char cannot be implicitly convertible to std::string. Although the l4 lambda is an improvement over the generic lambda l2 seen previously, it still does not help restrict calls completely if the arguments are of different types. This is only possible with lambda templates as shown earlier.

## Generic Programming

## Generic Lambdas and Lambda Templates

Another example of a lambda template is shown in the next snippet. This lambda has a single argument, a std::array. However, the type of the elements of the array and the size of the array are specified as template parameters of the lambda template:

```
auto l = []<typename T, size_t N>(
std::array<T, N> const& arr)
{
        return std::accumulate(arr.begin(), arr.end(),
        static_cast<T>(0));
};
auto v1 = l(1); // error
auto v2 = l(std::array<int, 3>{1, 2, 3}); // OK
```

# Generic Programming

## Generic Lambdas and Lambda Templates

Attempting to call this lambda with anything other than an std::array object produces a compiler error. The compiler-generated function object may look as follows:

```
struct __lambda_5
{
            template<typename T, size_t N>
            inline auto operator()(
            const std::array<T, N> & arr) const
            {
                        return std::accumulate(arr.begin(), arr.end(),
                        static_cast<T>(0));
            }

            template<>
            inline int operator()(
            const std::array<int, 3> & arr) const
            {
                        return std::accumulate(arr.begin(), arr.end(),
                        static_cast<int>(0));
            }
};
```

## Generic Programming

### Generic Lambdas and Lambda Templates

Another example of a lambda template is shown in the next snippet. This lambda has a single argument, a std::array. However, the type of the elements of the array and the size of the array are specified as template parameters of the lambda template:

```
auto l = []<typename T, size_t N>(
std::array<T, N> const& arr)
{
        return std::accumulate(arr.begin(), arr.end(),
        static_cast<T>(0));
};
auto v1 = l(1); // error
auto v2 = l(std::array<int, 3>{1, 2, 3}); // OK
```

A sample program is here.

## THANK YOU

**M S Anand**
Department of Computer Science Engineering

**anandms@pes.edu**