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

<u>Why Generic Programming?</u>

- Maximize code reuse
- type safety
- performance.

<u>Some popular languages which support genericity</u>
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	

An overview of C++

Basic aritmetic operations:

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++

Arrays and Pointers – Differences

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.

Generic Programming
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.

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);

Generic Programming

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++

File operations in C

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

fclose(FILE *)

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.

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

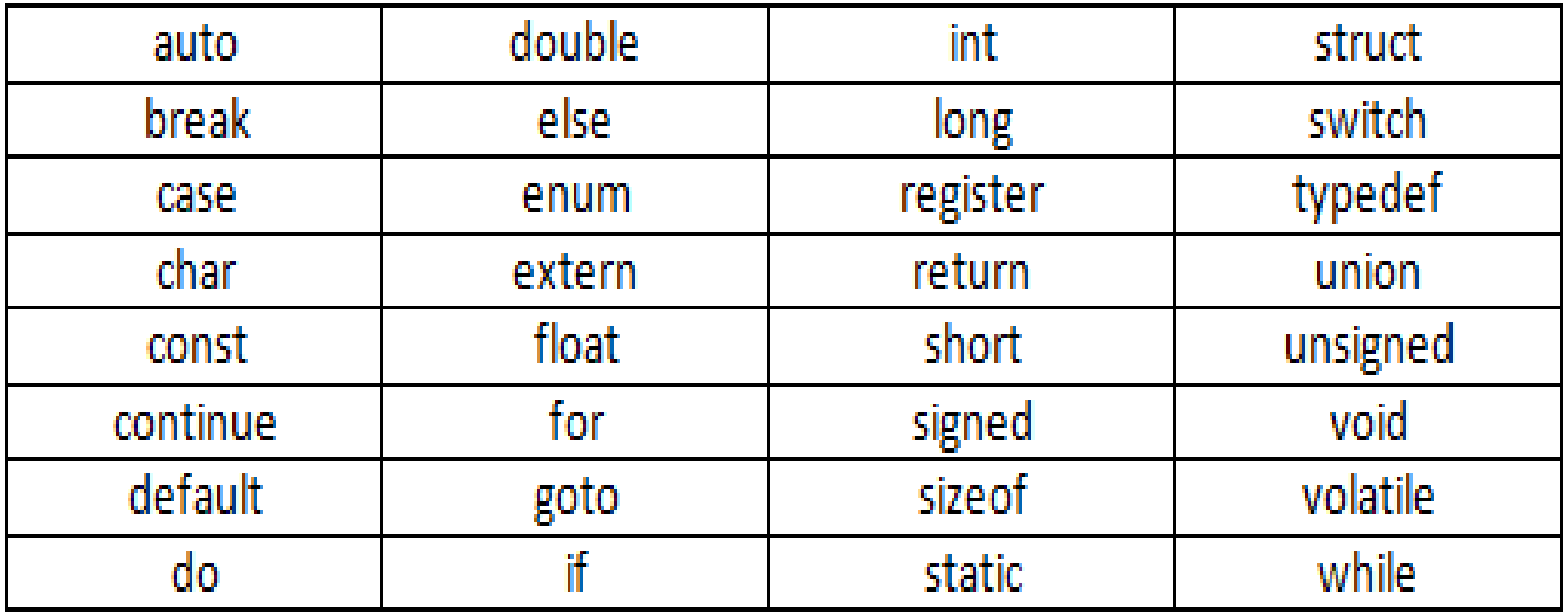
```
int func(int, ... )
{ . . .
}

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

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

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++.

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

Generic Programming
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. It cannot be invoked using the object as it is not in the scope of that class.
5. We can invoke it like any normal function of the class.
6. Friend functions have objects as arguments.
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.

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>

Generic Programming

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

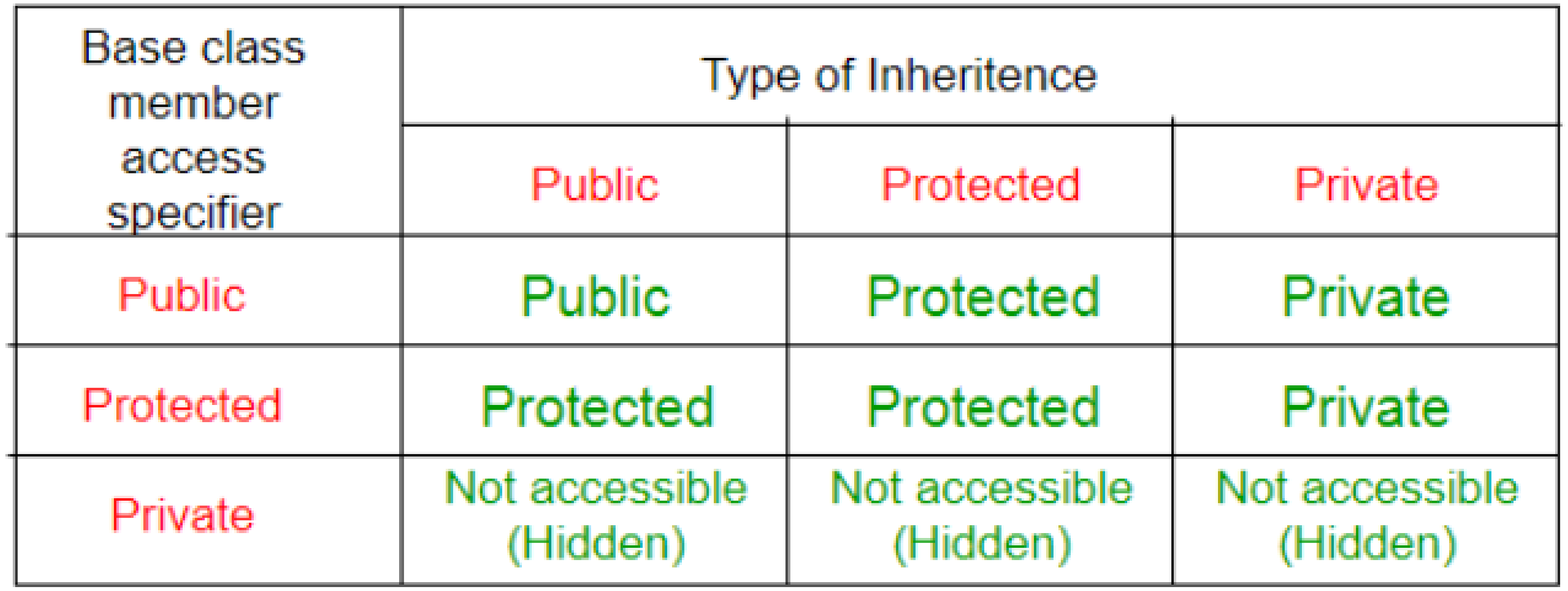
An overview of C++

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)

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.*

Generic Programming

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

Generic Programming

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 should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
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), in that case, the base class version of the function is used.
6. A class may have a virtual destructor 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.

Generic Programming
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.

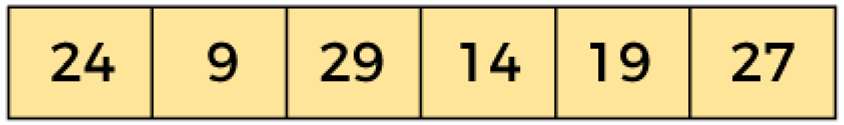
Combine: Combine the already sorted array.

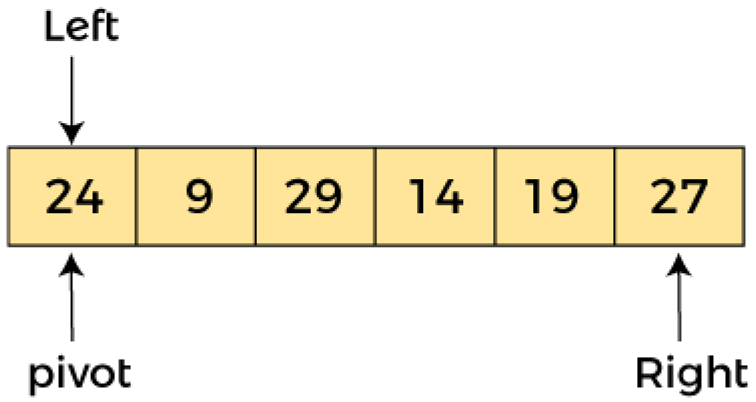
Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the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

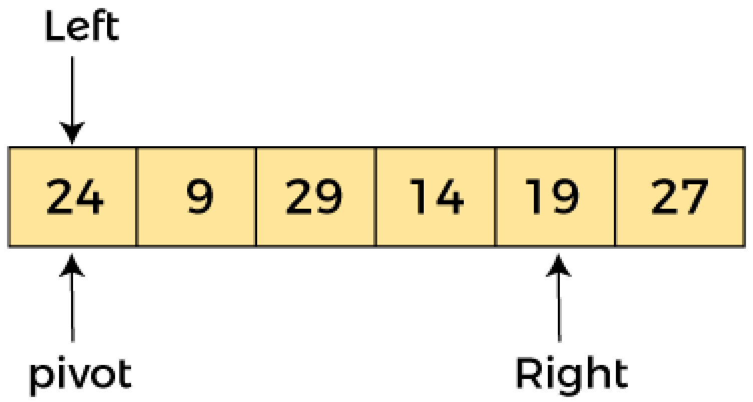
After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

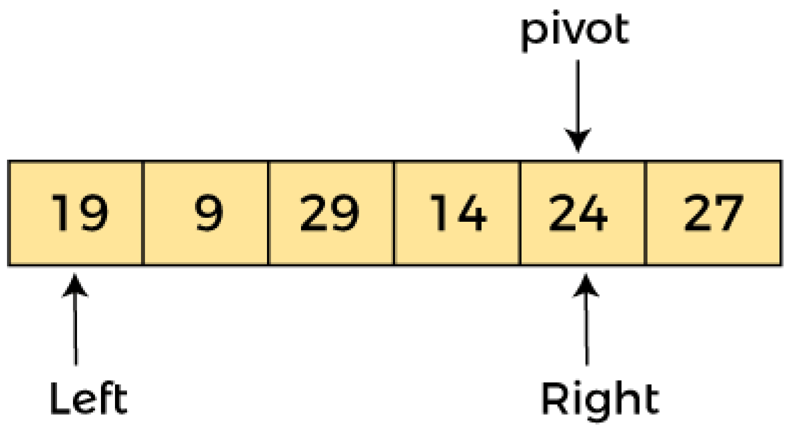
Generic Programming
Need for templates

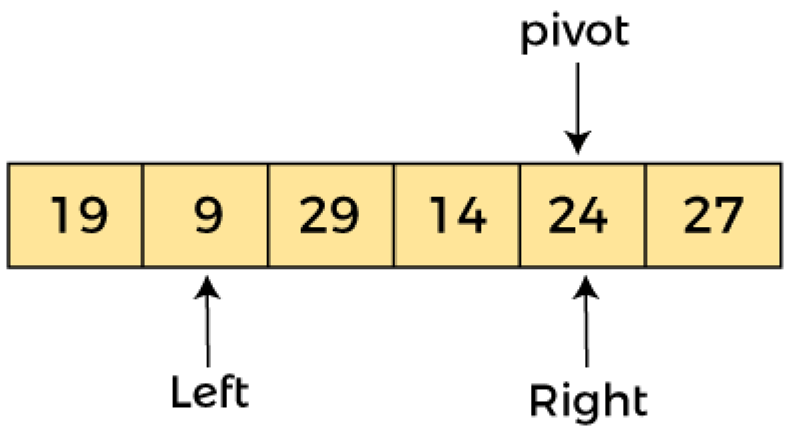
- This is the input array





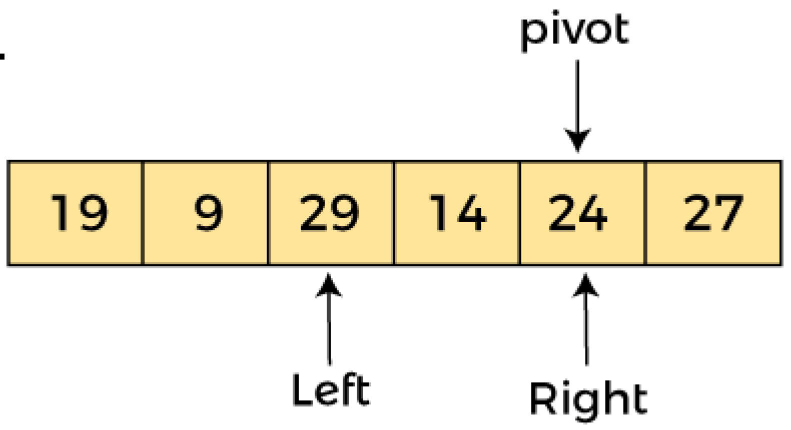


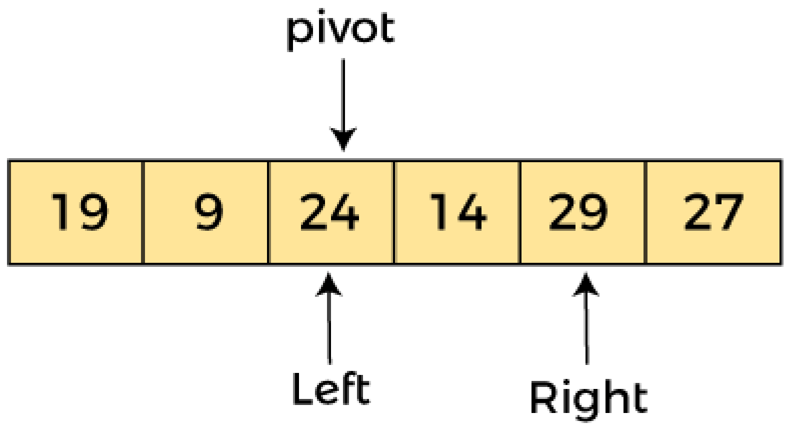


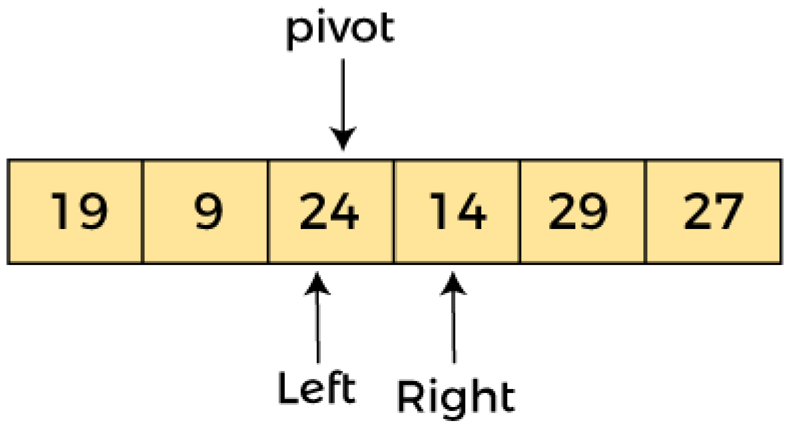


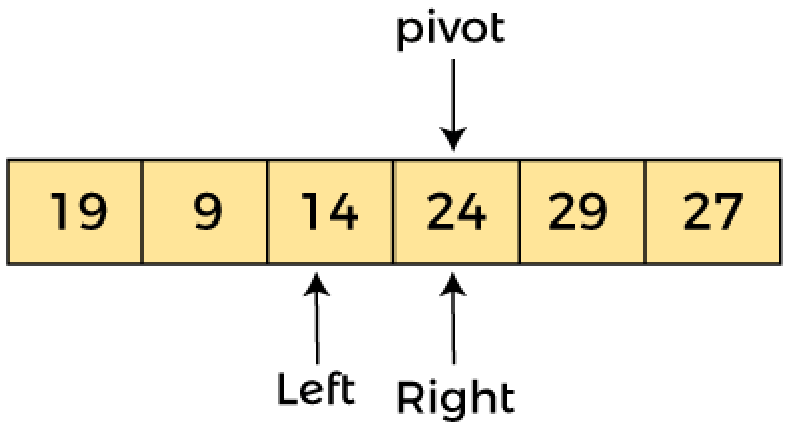
Generic Programming

Need for templates …





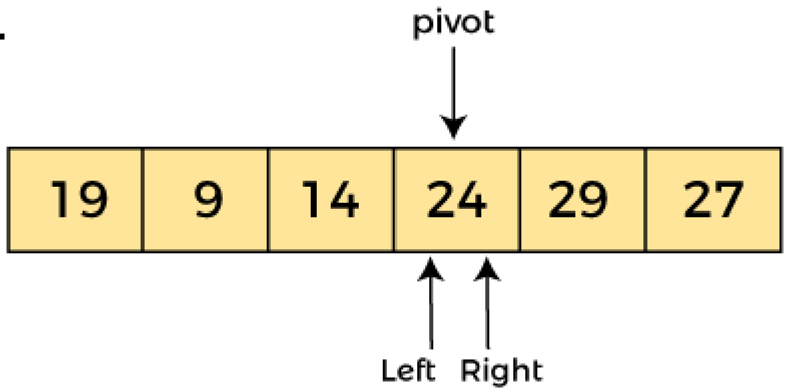


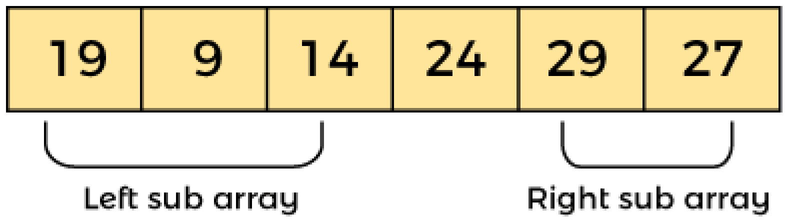


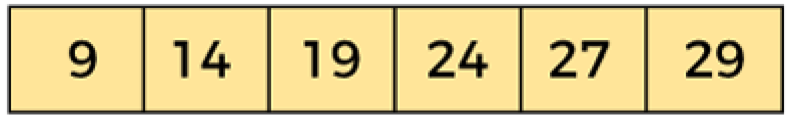
Generic Programming

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 if the specialized version of the vector class had a different interface (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 when all the template arguments are provided. 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. You can instead specify the definition the compiler uses for a given set of template arguments. 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

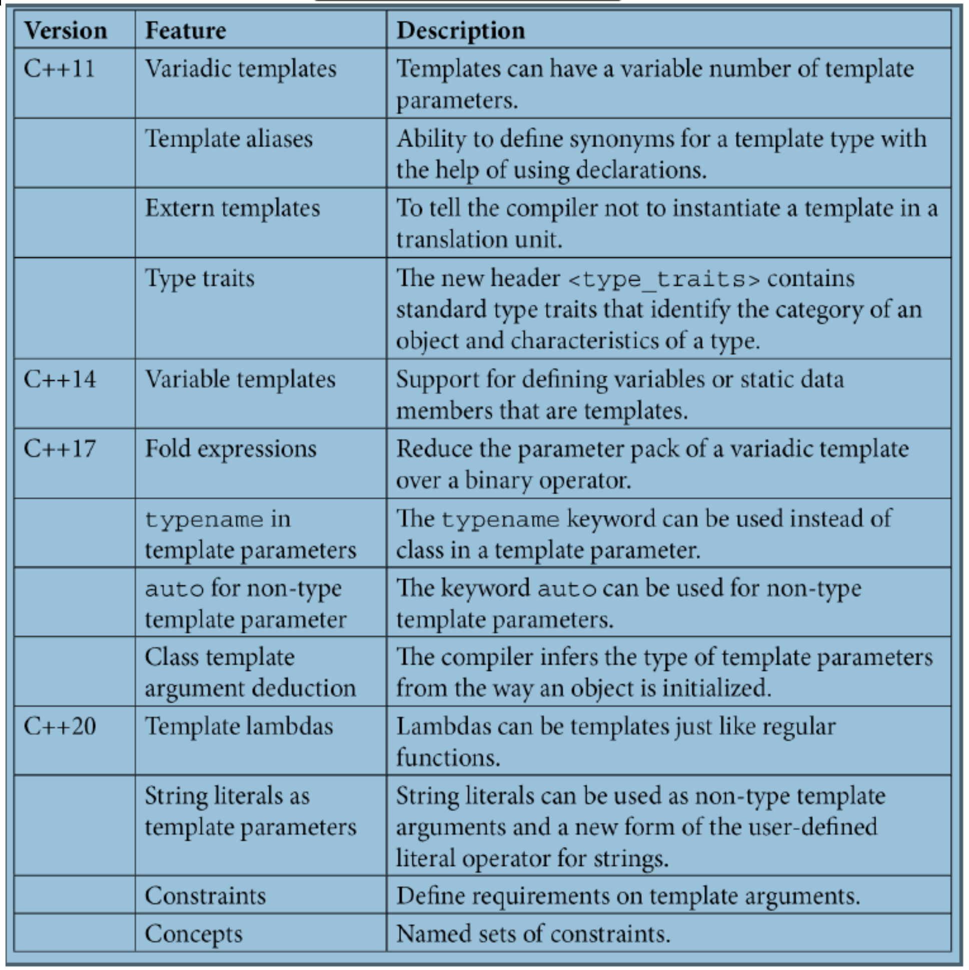
- Template metaprogramming is the C++ implementation of generic programming. This paradigm was first explored in the 1970s and the first major languages to support it were Ada and Eiffel in the first half of the 1980s. David Musser and Alexander Stepanov defined generic programming, in a paper called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

Generic Programming

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.

<u>If the type of the two parameters is ambiguous, the compiler will not be able to deduce them automatically.</u> 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

<u>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:</u>

```
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.

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@pes.edu