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	

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.

Assignment operators:

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

……

Relational operators:

>	x > y
<	x < y
>=	x >= y
<=	x <= y
!=	x != y
==	x == y

Increment and decrement operators:

Pre and post increment

Pre and post decrement

variable ++

++ variable

variable –

--variable

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.

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

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:

- 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;
```

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

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’

- 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.

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’};
```

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.

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.

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;

Structure

A structure is a user defined data type in C.

A structure creates a data type that can be used to group items of possibly different types 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;
};
```

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);

Linked lists

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

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

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() …….

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;
};
```

Unions

User defined data type just like a structure.

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

```
#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;
}
```

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.

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.

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

Command line arguments

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

argc ??

argv[0] - ??

How to pass an argument like “PES University”?

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

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

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.

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

Template functions

Example

Abstract datatypes in C++

String, Vector

Range-Based for

for (declaration : expression)
statement

Example

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

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:

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

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++.

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

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

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

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.

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.

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.

- 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.

Operator functions must be either member functions (non-static) or friend functions.

Not all operators can be overloaded.

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.

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

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

- 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

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.*

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

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.

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

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@pes.edu