Modern C++ For Software Developers

Serious C++ Development

Karan Singh Garewal

Modern C++ For Software Developers

Karan Singh Garewal

Cover Image Design by Kirill Tonkikh, unsplash.com

Changelog

Date	Version	Notes
October 10, 2022	1.3	new chapter on coroutines additions to the GDB Debugger chapter
August 30, 2022	1.2	std::format library function Revise threads chapter with additional working examples. Numerous edits to improve readability and improve comprehension.
July 30, 2022	1.1	*setjmp* and *longjmp* section refactor smart pointers chapter C++ Sanitizers chapter fix errors in some code examples [[maybe_unused]] – C++17 features C++ SonarLint chapter various edits to explain concepts more simply
February 28, 2022	1.0	Initial release

Preface

This is a book on serious software development with Modern C++. It is my go-to desktop reference when I am developing C++ applications. I hope that you will also consider this book in such an exalted light.

This book is divided into eight main sections. The first section is a C language refresher. Knowledge of the C language is presumed. The next section exposes core C++. This is C++ before C++11. This section is followed by four sections that discuss important features of C++11, C++14, C++17 and the latest standard, C++20. This is followed by a section that discusses important functions in the C++ Standard Library. The next section is devoted to the GNU C++ compiler and the GNU debugger. The last section looks at some miscellaneous topics.

Every important C++ feature that is discussed in this book is accompanied by a short C++ program to demonstrate how the feature works. You can cut and paste these examples into **wandbox** (https://wandbox.org) and then compile and execute them.

This book can be read sequentially or you can go directly to topics of interest.

The reminder of this preface is merely my opinionated meandering on C++ web applications. Is C++ web application development an anachronism? You can safely omit this part of the preface.

A Journey Through C++ Land

When I was going to university, I had a friend who was fond of saying "*I think in C*". He obviously liked C a lot. Over forty years later, after it's inception, C still remains very popular. A lot of people really like C. It appears that C imposes very little cognitive load on the brain; C fits easily into the brain.

But C has it's limitations. A few years back I was developing a deep neural network in C. After writing about 30,000 lines I started having difficulty reasoning about the application in it's entirety. This is not a critique about the language, it is an observation that C does not provide language constructs to architect large applications.

This is where C++ comes in. C++ provides constructs to structure and architect extremely large applications. Applications in the multi-million line range are eminently feasible. This is entirely due to constructs such as classes, encapsulation, function overloading, run-time polymorphism and templates. These constructs are remarkably versatile and a very large domain of problems can be modelled with these constructs.

But this not to say that C++ is perfect. It is possible that absolutely massive software applications such as the F-35 fighter jet program (up to 30 million C++ Source lines of code) are exposing the deficiencies in C++. In other words, C++ source code is not reliable enough at this scale. The C++ Steering Committee is aware of this issue. Each new release of C++ brings new constructs to improve the correctness of

C++ programs. Consider for example, *const*, *static_assert*, *constexpr*, *consteval*, *constinit*, *typed enum*s, *constant templates*, *auto*, *type deduction*, *braced initialization*, *noexcept*, *override*, [[nodiscard]] and a plethora of other language constructs.. The list of C++ correctness constructs is long and expanding.

C++ Is Not A Web Programming Language

This is a frequently repeated canard on the Internet. It may have been true in the past but it is not true today. Web frameworks based on dynamically typed languages are invariably held forth as being the best environments to develop web applications. I am alluding to the Python/Django and Ruby/Rails frameworks in particular. I came from a heavy C++ background and migrated to web development. I spent nearly a decade developing Rails applications. Never liked the language or the framework. But it would have been an unforgivable heresy to oppose conventional wisdom.

Frameworks based on dynamically typed languages are held out as being suitable for fast web development. I disagree. An experienced C++ programmer can develop web applications just as fast as someone developing in the Ruby and Rails environment for example. If not faster. How can web development in C++ be faster? It all boils down to the compile-link cycle in C++. If your C++ web application compiles and links properly it is almost certain that it will also run on the web. Your main concern will be the “business” logic of your program. In contrast, if the application is developed in a dynamically typed language, you can write the application quickly but you will spend a lot of time debugging the program at run-time. This is due to the fact that in a dynamically typed language variable types frequently change at run-time. Additionally the debugging tools for dynamically typed languages are mediocre.

In the discussion that follows I will use Ruby and Rails as a *bete noire*. Simply because this is the framework with which I have the most experience and this framework has a huge hype machine behind it. However the discussion applies equally to other frameworks built with dynamically typed languages.

Performance

There is a dirty little secret concerning web applications built with dynamically typed languages. These applications are very slow and a lot of engineering effort goes into making them run faster. For example, a C++ web application developed in the Drogon framework:

(https://github.com/drogonframework/drogon) is over fifty times faster than a Ruby/Rails application.

See: https://www.techempower.com/benchmarks/#section=data-r19&hw=ph&test=composite

The main culprit is the interpreter in these languages which must read each source code statement (or it’s bytecode) before executing the statement.

Also see a comparison of the Drogon webserver with the gold standard in webservers: Nginx.

https://drogon.docsforge.com/master/benchmarks/#test-plan-and-results

It is worth-noting that a medium-sized C++ web application will hardly register any CPU usage. An equivalent Ruby/Rails application will stress the CPU and you might end up having to load balance incoming HTTP requests to multiple machines.

Libraries vs. Frameworks

Using a web framework leads the developer into vendor lock-in. If you develop a web application in Python and Django or Ruby and Rails, your application will for all practical purposes be locked into the vendor’s framework. Migration to another framework will be a major undertaking.

Libraries do not produce such a lock-in effect. After all, you can easily switch libraries or develop your own library functions. For example. Drogon is a very thin framework layer wrapped around the drogon webserver.

Deployment

Deploying Ruby and Rails applications to production servers is complicated. All of the source files and static resources have to be deployed on the production server. In contrast the deployment of a C++ web application is simple. One or more executable files and the static resources are deployed on the production server. *rsync* can easily manage this deployment.

Trying The Theory Out

I wanted to confirm my suspicions with a production quality C++ web application. Since I am an Attorney at Law, I developed https://truelawdocs.com as a Drogon C++ application. This is a multi-threaded application for document production. It took about four months of part-time work. The application uses the Cassandra NoSQL database for document storage. This is clearly massive overkill for an application like TrueLawDocs. Cassandra slowed the development cycle since I had zero prior Cassandra experience. I wanted to learn Cassandra on the supposition that I might need this column-oriented NoSQL database in the future. I really liked Cassandra, it's very well designed.

TrueLawDocs has a front-end developed in Bootstrap, Vue, Javascript and CSS. The C++ back-end and the front-end communicate through websockets. As you can surmise TrueLawDocs has a strongly decoupled architecture.

Aside from the Drogon framework and the C++ standard library, the only libraries used were *nlohmann/json* a C++ JSON library, AES.hpp a header only C++ encryption library, the DataStax C driver library for Cassandra, ASAN (address sanitizer) and UBSAN (undefined behaviour sanitizer). That is it for third party libraries. Everything else was hand-coded.

For payment integration with Stripe, the *LibCurl* library was used to interact with the Stripe payment server. Stripe does not support C++ and provides no sample code to integrate C++ applications. The integration of the TrueLawDocs C++ code with the Stripe server exemplifies the raw power of C++.

The overall development experience was very pleasant. The Drogon source code is easy to digest; it’s written by a master of Modern C++.

TrueLawDocs including Cassandra is running on a seven dollar a month VPS rented from a company In Sharjah.

Lessons Learned

During my tenure contracting on Rails applications I rarely met developers who had broad experience with other programming languages. They typically only knew their framework. In a decade of contracting, I never met a Rails programmer who was reasonably proficient in C++.

One of the reasons that dynamically typed web languages are popular is that they are easy to learn. You can master Rails in less than three months. In contrast one cannot gain reasonable proficiency in C++ unless one has programmed in the language for about two years. The Modern C++ learning curve is very steep. Unfortunately. It’s unavoidable.

Another factor is economics. Web applications built in dynamically typed languages rely on cheap labour. Fortune 500 companies use large inputs of foreign contractors and software quality testers to develop and maintain their applications. To my mind, this is a statement on the inadequacy of dynamically typed languages for web development. If these cheap resources were not available, the development model would shift to conserving expensive, high quality manpower.

In terms of the debugging effort that was expended developing *TrueLawDocs*, I estimate that 75% of the effort involved debugging the Vue and Javascript front-end. C++ provides superb debugging tools so debugging the back-end was a snap. Insofar as the back-end debugging infrastructure was concerned, I relied on the extensive use of Debug macros, ASAN (to detect memory errors), UBSAN and the GDB debugger. Sonarlint was used check syntax.

The development environment was Linux Mint (classic GNOME interface with 24 open virtual windows). The IDE was VisualCode. The compiler suite was GCC. Compiling and debugging was done in a terminal. A python script executed the compile-link-run-debug cycle. A terminal was always open to *tail* the debug output. The application generates a huge quantity of debug messages. Special attention was devoted to generating high quality debug output. This made tracing errors exceptionally easy. User acceptance testing was effected with Selenium.

As you may have surmised I walked away from Rails. It was a good decision. I have been playing around in the back of my mind with developing a C++ YouTube clone. Like Rails, I finally got sick of google's censorship. Anyways, that's the power of C++ for you.

Thank you for your kind consideration.

Karan Singh

Table of Contents

C Language Review

Declarations And Definitions

In C and C++ an **identifier** is a name that is assigned by a user to a program element such as a variable, type, template, class, function or namespace.

A variable holds a value. In C and C++, variables must be declared before they can be used. A **declaration** specifies the type of the variable and the name of the variable. A declaration implicitly specifies the amount of memory that the variable needs to hold a value. For example:

```
int x;
```

declares that *x* is a variable of type *int* and this type of variable needs 32 bits of memory.

A declaration of a function consists of a function return type, the function name and the types of the function parameters. For example:

```
int foo(int x, int y);
```

A variable **definition** creates a variable and may set (*assign*) a value to the variable:

```
int x;           // declaration of a variable x
x = 5;           // definition of variable x
```

This definition places the value 5 inside the memory that has been allocated to the variable *x*.

Declarations and definitions can be combined into one statement. For example:

```
int x = 5;

or

int x;        // x holds some unknown value
```

For primitive C and C++ types such as *int*, *long*, *float* and *double*, a declaration is also a definition. This however is not the case for more complex types that we will encounter later.

A definition of a function tells the compiler how the function works. It shows the instructions that are executed by a function:

```
// declaration of function foo (also called a fuction prototype)
int foo();

// definition of function foo;
int foo () {
  int x = 5;
  return x;
}
```

Statement Termination

In C and C++ statements are not terminated by a new line (‘/n’). They are terminated by a semi-colon. Thus C and C++ statements can freely extend over more than one line.

Literals

Literals are constants. C has four types of literals: integer literals such as 45, floating point literals such as 3.12, string literals such as “*hello world*”; and character literals such as ‘*c*’.

A literal is a program element that directly represents a value. For example, 509 is an integer literal or constant.

Hexadecimal And Octal Literals

Hexadecimal and octal literals are literals of integer type.

A hexadecimal literal, begins with the prefix `0x` or `0X` and is followed by a sequence of digits in the range{ `0` through `9, a-f, A-F}`. For example:

```
0x23A, 0Xb4C, 0XFEA, 0xDEAD, 0x5
```

Hexadecimal literals have base 16.

An octal literal, begins with `0` and is followed by a sequence of digits in the range 0-7. For example:

```
045, 076, 06210
```

Octal literals have base 8.

Floating Point Literals

Floating point literals represent real numbers. A real number has a real part, a fractional part and an optional exponential part:

```
9.87654
987.654
9.87654e+006
9.87654e-005
```

e is the irrational number 2.7182818...

Valid floating point literals have the following properties:

- In decimal form, the number must include either the decimal point or an exponent part or both.
- In exponential form, the number must include an exponential part.

Floating point numbers have type *float*, *double* or *long doubl*e.

Char Literals

A char literal is a single character enclosed by single quotes. An error will be thrown if there is more than one character enclosed by single quotes.

A char literal has type char:

```
char chr = 'G';

wchar_t chr =  L'A';            // wchar_t (wide)
```

String Literals

A string literal is a sequence of characters enclosed in double quotation marks.

```
 char* s0 =   "hello world";                    // char*
 char* s1 = u8"hello world";                    // char*, encoded as UTF-8
wchar_t* s2 =  L"hello world";                 // char*, encoded as wchar_t
```

A string literal is always terminated by the null character '\0'. The null character is automatically appended by the compiler to the string literal.

Literal Prefixes And Suffixes

In C, a literal can optionally have a prefix and a suffix. A prefix to an integral literal indicates the base of the number. The suffix indicates the type of the integer. For example, in 0xDEAD, 0x implies that DEAD is a hexadecimal integer (base 16).

An integral literal without a prefix indicates a decimal number (base 10).

Type	**Representation**
unsigned int	U or u
unsigned long int	UL or ul
long long	LL or ll
unsigned long long	ULL or ull

Variable Scope In C And C++

Variable scope refers to the region of a program where a variable exists and can be accessed.

In C there are three places where a variable can be declared:

- inside a brace delimited block { ... }   (local variables)
- in the parameter list of a function
- outside all functions   (global variables)

A pair of braces {} defines a block and variables declared inside a block have their scope restricted to the block. In C, the scope of a variable begins at the point of it's declaration.

C and C++ implement **lexical scope**. This means that the scope of any variable can be determined at compile-time by examining the source code.

Lexical scope can be explained as follows. Suppose that the compiler encounters a variable name in a block of code and wants to know whether the name exists and it's value. The compiler first examines the current block for the declaration of the name and if not found; it examines any outer enclosing block(s) and finally the global scope for the variable name.

The following C++ program shows C/C++ lexical scoping:

```
#include <iostream>
int value = 999;

void f()
{
  {
    int value  = 1;
    std::cout << "value in f - " << value << std::endl;
  }
  {
     std::cout << "value in f - " << value << std::endl;
  }
}


int main()
{
  int value = 2;
  f();
  std::cout << "value in main - " << value << std::endl;
  return 0;
}

// output
value in f - 1
value in f - 999
value in main - 2
```

C Language: Function Prototypes

Forward Declarations

The following program will not compile:

```
#include <stdio.h>

int main() {
    printMe(5);
}

// printme declared and defined here
void printMe(int x) {
    printf(“%d”, x);
}
```

The problem is that when the compiler encounters the *printMe* statement in *main*, it does not know what *printMe* is, since *printMe* has not been declared. This problem can be solved by including the definition of *printMe* before *main* or by including the declaration of *printMe* before it it is used.

```
#include <stdio.h>

// forward declaration
void printMe(int);

int main() {
    printMe(5);
}

// definition of printMe
void printMe(int x) {
    printf(“%d”, x);
 }
```

This forward declaration of *printMe* is also called a function prototype.

The void Type

The data type *void* represents no value. Functions should always be declared as *void* if they do not take any parameters.

Consider the following two prototypes:

```
void foo(void);
void foo();
```

The C compiler interprets the first prototype as not taking any parameters and not returning any value. The second declaration is interpreted by the C compiler as not knowing whether *foo* takes any parameters and returning no value (*the C++ compiler behaves differently*).

C Language: C Style Casts

A variable value is stored as a sequence of bits. The data type of a variable tells the compiler how to translate these bits into meaningful values. Different data types may represent the “same” number differently. For example, the integer value 3 and the float value 3.0 are stored as completely different bit patterns.

In C we can convert a variable or a literal from one type to another by preceding the variable or literal with the target type in round brackets:

float y = **(float)** 6 /4;

This is an example of type conversion. The compiler will convert the integer number 6 to a float and then divide it by 4 to get 1.25. Note that if the float type conversion is not present on the right hand side, then the right hand side will evaluate to 1 and 1.0 will be assigned to y. Since y is a float type the compiler performs an implicit type conversion of the integer 1 to a float type.

Function-Style Casts

C also implements function-style casts:

```
T(expression)
```

where T is some type. For example

```
int x = int(3.14)          // 3.14 is cast to an int type.
```

Implicit Type Conversion

The type conversions that we have shown in the examples above are explicit type conversions. However the compiler will automatically attempt an implicit type conversion when the target type is a type different from the subject type. An implicit type conversion or **automatic type conversion** will occur in three cases:

<u>During Initialization:</u>

```
float y = 5;       // int type 5 converted to float
```

<u>When Setting Function Parameters:</u>

```
int myFunc(float p);

// myFunc is called with an int argument. For example, myFunc(3)
```

<u>Function Return Values:</u>

A function’s return value is changed to a different type.

```
float y = myFunc(5);        // myFunc returns an integer type
```

Automatic type conversion frequently occurs whenever a value from one fundamental data type is converted into a value of a larger fundamental data type from the same family (this is called widening). For example, an *int* can be widened into a *long*, and a *float* promoted into a *double*.

Shortening will occur when a type conversion converts a value to a type which has a shorter bit length. For example, this occurs when a variable of an *int* type is converted to a *char* type or a *long* type variable (64 bit length) is converted to a variable of *int* type (32 bits). These types of conversions are called <u>shortening conversions</u>. They are dangerous because they can result in a loss of value.

C Language: Comma Operator

The comma operator causes a sequence of expressions to be evaluated from left to right. The value of a comma-separated sequence of expressions is the value of the right-most expression. For example:

```
int j = 10
int val = j+10, j + 5, j + 1;
```

In this example, first the expression *j+10* is evaluated. Then the expression *j+5*. Finally the expression *j+1* is evaluated. The result of the last expression (26) is assigned to val.

Comma separated expressions are frequently found in loops:

```
for (int i = 0, int y = 10; i  < 5; i++, y--) {
     ...
}
```

Note: the comma operator has a lower precedence than the assignment operator.

C Language: Pointers

A pointer is a variable that stores the address of some other variable. The syntax for a pointer variable declaration is:

```
int* ptr;
```

This states that *ptr* is a pointer variable and it’s value is the address of a variable of *int* type. Note that this is a declaration only and *ptr* has not been initialized (so *ptr* will contain some garbage value). Suppose that we have:

```
int x = 5;
```

Then the address of *x* (the memory address where the value of x is stored) is:

```
&x
```

The **address operator &** gets the address of the variable x.

We can initialize *ptr* (point *ptr* to x) as follows:

```
ptr = &x;
```

Now *ptr* contains the address of *x*. We can obtain the value at the address pointed to by *ptr* as follows:

```
int value = *ptr;
```

This is referred to as dereferencing the pointer. * is the **dereferencing operator** and gets the value at the location *ptr* points to.

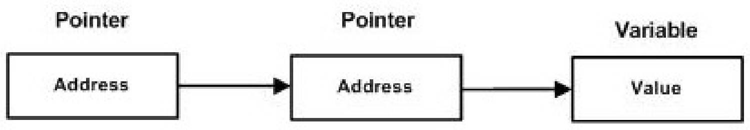
Pointer Arithmetic

Pointer variables can be incremented and decremented. An increment sets the value of the pointer variable to a location that is size of (the data type pointed to) bytes ahead of the current value of the pointer.

In our example, pointer is pointing to an int type which is 4 bytes long, so ++ptr points to the location four bytes ahead of the current location pointed to. Similarly with the decrement operator. Ordinary addition and subtraction work similarly.

Multiple Indirection With Pointers: Pointer To Pointer (**ptr)

A pointer to a pointer is a form of multiple indirection. A pointer contains the address of a variable. When we define a variable that is a pointer to a pointer, the variable points to an address that contains the address of a pointer variable. The second pointer points to the location that contains the actual value. This can be shown as:



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of the name of the double pointer:

```
    int x **ptr;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing the value requires that the dereferencing operator * be applied twice to the pointer variable:

```
#include <iostream>

int main() {
  int x = 5;
  int *ptr1;
  int **ptr2;

  ptr1 = &x;
  ptr2 = &ptr1;

  int z = **ptr2;
  std::cout << z << std::endl;
}

# result
5
```

C Language: Arrays

An array is a collection of variables of the same type that are referenced through a common name. A specific element in an array is accessed through an index. In C and C++, arrays consist of contiguous memory locations. The lowest address corresponds to the first element of the array and the highest address to the last element. Arrays may have from one to several dimensions. The most common array is the null-terminated string, which is simply a sequence of characters enclosed in quotation marks terminated by a null character (‘\0’).

Single Dimensional Arrays

These types of arrays are specified with the following syntax:

```
type name[size];
```

Here, type declares the *type* of the array, which is the type of each element of the array, and *size* defines the number of elements that the array will hold. Arrays must be explicitly declared so that the compiler can allocate space for them in memory. Here is an example:

```
int flowers[5];
```

Array elements are accessed with the bracket operator ([]) and an index value which starts at zero and ranges up to but not including the size of the array. The index 0 refers to the first element of the array:

```
int p = flowers[0];
int p = flowers[2];                // third array element
```

C and C++ do not perform bounds-checking on arrays.

Pointers To Arrays

In C as well as C++, the name of an array is defined to be a pointer to the first element of the array:

```
int *p;
int sample[10];
p = sample;
```

We can also define a pointer to the first element with:

```
p = &sample[0];
```

Arrays And Double Indirection (** ptr)

Consider an array where each element of the array is a null terminated string:

```
char* names[10];
names[0] = “hello”;
...
names[9] = “world”;
char* ptr = names;
```

ptr is a pointer to the names array. *ptr = names* and *ptr = &names[0]* are equivalent. However *names[0]* is itself a pointer to a null terminated string. Thus **ptr* is a pointer to a null terminated string. Thus the first string in the array is ***ptr*.

Due to this, *char **ptr* is a pointer to an array of null terminated strings. **(ptr+1) points to the second null terminated string.

Array Initializers

In C++ we can initialize an array as follows:

```
int array[5] = {1, 2, 3, 4, 5};
```

We can have fewer initializers than the length of the array, in which case, the remaining array elements will be initialized to the ‘zero’ of the type of the array.

Passing Single-Dimension Arrays to Functions

In C and C++, we cannot pass an entire array as an argument to a function. We can, however, pass a pointer to the array by specifying the array's name without an index.

```
int main(void)
{
     int sample[10];
     func(sample);
}
```

An array name (without an index) is a pointer to the first element of the array.

The formal parameter of a function that receives an array can be declared as a pointer to the array’s data type or as a sized or unsized array. For example:

```
func (int *p)
func (int m[10]);
func(int m[])
```

All of the above are equivalent. The array size can be any value; for example, *func(int m[100000])* is also valid. An array parameter in a function parameter list is syntactic sugar since only a pointer to an array is received by the function. The programmer is responsible to ensure that a buffer overrun does not occur.

Multi-Dimensional Arrays

A multi-dimensional array can be specified with the following syntax:

```
int m[6][8];                  // two dimensional array of integers
char s[7][3][5];              // three dimensional array of chars
```

An array can have an arbitrary number of dimensions.

C Language: Strings

C an C++ implement null-terminated strings. A string is a null-terminated character array (a null is ASCII 0 and is represented as ‘\0’). Thus a C-style null-terminated string is an array of characters that has a null character as it’s last array element. Here is an example:

```
“hello world”
```

The C compiler will automatically append a null character to the end of this string literal. A sequence of characters enclosed in double quotation marks is interpreted as a constant character string by the C compiler. A constant string cannot be changed.

```
char* str = “hello world”;
*str = ‘z’;                   // error – attempt to change ‘h’ to ‘z’
```

When declaring a character array that will hold a null-terminated string, we must declare it to be one character longer than the largest string that it can hold. The extra character is for the null character (‘\0’).

The following string declarations are all equivalent:

```
char *str = “hello world”;
char str[12] = “hello world”;
char str[] = “hello world”;
```

If we use an array declaration (instead of a string literal) then the string is mutable:

```
char* str[] = “hello world”;
str[0] = ‘z’;                     // OK
```

str without the array notation points to the first character of the string.

We can get the length of a string with:

```
strlen(str);
```

We can copy a string with:

```
char* source = “hello world”;
char dest[12];
strcpy(dest, source);
```

n characters can be copied with:

```
strncpy(dest, source, n);
```

C Language: Structs

In C, a struct is a container data type that can hold data of different types. A struct is declared as follows:

```
struct employee {
  char name[];
  int employee_no;
  float salary;
};
```

This declaration creates a new type called *struct employee* (not just *employee*). We can instantiate instances of this data type as follows:

```
struct employee alice_smith;
```

Members of alice_smith can be accessed with the dot operator:

```
char* name = alice_smith.name;
float slr = alice_smith.salary;
```

Struct Initialization

A struct can be initialized at the time that it is declared as follows:

```
struct employee emp = {“alice_smith”, 1200, 10000.25};
```

All of the fields of the struct do not have to be initialized. Missing initializers are set to the zero of the field’s type.

There is a problem with this initialization. If we subsequently change the order of the fields of the struct then the initialization will become invalid. We can avoid this problem by making the initialization independent of the field order:

```
struct employee emp = {.name=”alice_smith”,.employee_no=1200,.salary=10000};
```

Passing Structs to Functions

A struct can be passed to a function by value or by reference (pointer).

Arrow Operator

Suppose that we have a pointer to a struct:

```
      struct employee emp = {“alice_smith”, 123, 10000};
      struct employee* ptrEmployee;

      ptrEmployee = &emp;
```

We can address members of *emp* as follows:

```
      ptrEmployee->name;
```

This is the same as:

```
      (*ptrEmployee).name;
```

C Language: Files

When we do any kind of I/O in C, we do so through a pointer to a structure of type FILE. FILE holds all the information needed to communicate with the I/O subsystem: the name of the file that we have opened, where we are in the file, and so on.

In the C programming language, a file is a type of stream. A stream is simply a flow of data from some source.

When a C language program is executed, three streams (pointers to FILE objects) are opened by default, these are:

stdin	Standard Input, generally the keyboard by default.
stdout	Standard Output, generally the screen by default.
stderr	Standard Error, also generally the screen by default.

Reading Text Files

Streams are either text streams or binary streams. A text file is a sequence of characters where each sub-sequence is separated by a newline (*n*).

The following C program is a canonical way to read a text file containing the string: “Hello World!”:

```
#include <stdio.h>

int main(void)
{
 // Variable to represent an open file
 FILE *fp;

 // Open a file for reading
 fp = fopen("hello.txt", "r");

// Read a single character
 char c = fgetc(fp);

 // Print the character to stdout
 printf("%c\n", c);

 // Close the file when done
 fclose(fp);
}
```

EOF: End Of File

The special symbol EOF is returned if *fgetc* attempts to read a character past the end of a file. EOF is a macro. The previous program can be written as:

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  char c;
  fp = fopen("hello.txt", "r");
  while ((c = fgetc(fp)) != EOF)
     printf("%c", c);

  fclose(fp);
}
```

Reading Lines From A Text File

The function *fgets()* can be used to read lines from a text file. *fgets* returns NULL if there is an error or an attempt is made to read past the end of a file. It’s signature is:

```
fgets(char* buffer, size_t sizeof buffer, FILE* fptr)
```

The first parameter is a buffer large enough to hold a line. The second parameter is the size of the buffer. The third parameter is a pointer to a FILE structure.

Formatted Input For A File

The function *fscanf* reads data from a text file and puts the data into variables. The data elements in the file must be separated by white space. fscanf returns EOF if there is an error or an attempt is made to read past the end of the file. Here is an example:

```
#include <stdio.h>

int main(void)
{
FILE *fp;
char name[1024];
float length;
int mass;

fp = fopen("animals.txt", "r");
while (fscanf(fp, "%s %f %d", name, &length, &mass) != EOF)
    printf("%s animal, %d lbs, %.1f meters\n", name, mass, length);

fclose(fp);
}
```

Writing To Files

We open a file for writing with *fopen("filename", "w")*. If the file already exists, it is truncated to length 0.

The file read functions *fgetc*, *fgets* and *fscanf* have file write analogues: *fputc*, *fputs* and *fprintf*. We use these functions as follows:

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  int x = 32;

  fp = fopen("output.txt", "w");
  fputc('B', fp);
  fputc('\n', fp);
  fprintf(fp, "x = %d\n", x)
  fputs("Hello, world!\n", fp);
```

```
        fclose(fp);
    }
```

Notice that since *stdout* is a pointer to a FILE, we can do *f* = *stdout*.

Binary Files

Binary files work very similarly to text files, except that the I/O subsystem doesn't perform any translations on the data like it might with a text file. With binary files, we get a raw stream of bytes that is not translated by the I/O subsystem.

To open a file in binary mode, we add a '*b*' to the file mode (*'rb' or 'wb'*). The most common functions are *fread()* and *fwrite()*. These functions read or write a specified number of bytes from or to a filestream.

C Language: Memory Allocation

C uses the *malloc*() function to allocate memory. *malloc()* takes the number of bytes to allocate as it's only parameter, and returns a void pointer to a contiguous block of newly allocated memory. The pointer to a void type can be cast to any other type. *malloc()* returns NULL if memory cannot be allocated. *malloc* does not initialize the newly allocated memory.

After we have finished using the memory, we de-allocate the memory with *free()*. Here is an example:

```
#include <stdlib.h>

//implicit type conversion from void* to int*

int *p = malloc(sizeof(int));
*p = 12;

printf("%d\n", *p);
free(p);
```

calloc() is similar to *malloc()*. It takes the size of one element and the number of elements to allocate. *calloc()* initializes the allocated memory to ascii null bytes ('\0').

```
#include <stdlib.h>

int *p = calloc(sizeof(int), 10);
...

free(p);
```

The *realloc*() function can be used to shrink or grow already allocated memory. *realloc* takes a pointer to already allocated memory and the size of the new memory allocation. It then grows or shrinks that memory, and returns a void pointer to it.

```
float *p = malloc(40);    //allocate 40 bytes
...
float *new_p = realloc(p, 80);
```

C Language: Storage Class Specifiers

Storage Class specifiers tell the compiler how to store a variable. C has four storage class specifiers:

- extern
- static
- register
- auto

Variables with storage class specifiers are declared as follows:

```
storage_class_specifier type variable_name;
```

extern

C and C++ allow a large program to be broken up into files and for these files to be compiled separately and then linked together. Because of this, there must be some way of telling a file about variables that are declared in some other file.

A variable declaration declares the name and type of a variable. A variable definition causes storage to be allocated for the variable. In most cases, variable declarations and definitions are combined together (*e.g. int x = 5;*). However when we precede a variable declaration with the *extern* specifier, we inform the compiler that the variable is declared in some other file.

One prominent use of *extern* is to declare all of the global variables in one file and use *extern* declarations in other files to access these variables.

Example:

```
File 1                                File 2
```

```
int x, y;                        extern int x, y;
```

The extern specifier tells the compiler that the variable names that follow it have been declared in some other file. In other words, *extern* lets the compiler know what the types and names of these global variables without actually creating storage for them again. When the linker links the two object files, all references to the external variables are resolved.

The *extern* keyword has this general form:

```
extern type variable_name;
```

Static Variables

Unlike global variables, static variables are not known outside the function or file where they are declared. Furthermore static variables that are declared inside a function maintain their values between function calls.

Static Variables Inside Functions

A static local variable inside a function retains its value between function calls.

Here is an example:

```
int series(void)
{
    static int series_num;
    series_num = series_num+23;
    return series_num;
}
```

A static local variable can be initialized as follows:

```
int series(void) {
    static int series_num = 100;
    series_num = series_num+23;
    return series_num;
}
```

Static Global Variables

Applying the specifier static to a global variable instructs the compiler to create a global variable that is known only in the file in which it has been declared. This means that even though the variable is global, functions in other files do not have any knowledge of this variable and cannot access it’s value directly.

Register Variables

The *register* storage class provides a hint to the compiler that a local variable will be heavily used and should therefore be kept in a CPU register instead of RAM memory to provide quicker access. The compiler is free to ignore this suggestion.

In C, we cannot take the address of a register variable.

The *register* keyword is deprecated in C++ since optimizing compilers can automatically put heavily used variables in CPU registers.

C Language: const

const is a storage specifier that signifies that a variable cannot be modified. Here is an example:

```
const int x = 9;
```

x is an integer type variable those value cannot be changed. It is a read-only variable. Consider the following statement:

```
const int*  p;
```

This means that *p* is a pointer to an integer and the integer value cannot be changed. Thus if **p = 5*, then incrementing the pointer, *++p*, is valid but the statement: **p = 11;* will not compile.

Consider the following statement:

```
int x*  const p;
```

This states that that the pointer p is a constant those value (the location it points to) cannot be changed. Thus the statement *++p* will not compile.

Consider:

```
const int* const p;
```

Here the pointer p is a *const* and thus it cannot be incremented or decremented. Furthermore the value pointed to cannot be changed.

The general rule for pointers and const is:

If the *const* attribute is to the left of the * then the value pointed to is a constant. If the *const* is to the right of the * then the pointer is a constant.

The benefits of using const are:

- prevents accidentally writing to a variable that is meant to be a constant.
- enables the compiler to perform more optimizations
- permits a variable to be placed in ROM.

const In Functions

The use of const in functions is particularly important. Consider

```
#include <stdio.h>

int main() {

  int x = 5;
  f(x);

}

void f(const int y) {
    y++;
    std::cout << y << std::endl;
}
```

f receives a const copy of the integer variable *x*. This program will not compile since f is trying to change y which is a const parameter.

C Language: Function Pointers

Even though a function is not a variable, it has a physical location in memory. A function has a block of memory that contains the function's executable code. The address at the beginning of this chunk of memory is the entry point of the function. It is the address that is used when the function is called.

The address at the beginning of a function's block of memory can be assigned to a pointer variable. This variable is then called a function pointer. A function can be called through a function pointer. Function pointers also allow arguments to be passed to a function.

We obtain the address of a function (the address at the beginning of the memory used by the function) by using the function's name without any parentheses or arguments.

Suppose that we want to specify a function pointer for a function that takes a pointer to a char and an integer as arguments and returns an integer. The declaration for this type of function pointer is:

```
int  (* func)(char* ch, int b)
```

func is a function pointer for this type of function. The expression *(* func)* is merely a symbolism that the compiler understands as denoting a function pointer. The compiler identifies *func* as a pointer to a function. *func* can be assigned to any function that has a char pointer and an *int* as parameters and returns an *int*.

Suppose that we have a function: *int pick(char*, int),* then we can assign our *pick* function to the function pointer *func* as follows:

```
func = pick;
```

Here is a concrete example:

```
#include <stdio.h>
#include <string.h>

// forward declaration for the check function
// the third parameter is a function pointer
void check(char *a, char *b, int (*cmp)(const char *, const char *));

int main(void)
{
     char s1[80] = "hello world";
     char s2[80] = "bye";

     // declare a function pointer p
     int (*p)(const char *, const char *);

     // assign the strcmp library function to p
     p = strcmp;
    check(s1, s2, p);
    return 0;
}


// implementation of the check function

void check(char *a, char *b, int (*cmp)(const char *, const char *))
 {
     printf("Testing for equality.\n");

     if (!cmp(a, b)) printf("Equal");
     else printf("Not Equal");
 }
```

When the *check()* function is called, two character pointers and one function pointer are passed as arguments. Inside *check()*, the expression *(*cmp)(a, b)* calls *strcmp()*, which is pointed to by *cmp*, with the arguments a and b.

C Language: setjmp And longjmp

The functions *setjmp()* and *longjmp()* can be used to save the execution context of a running program and restore it at some later point in time. By execution context, we mean the state of the CPU registers, the stack and allocated memory. *setjmp()* saves the execution context of the program and *longjmp()* restores it.

setjmp() and *longjmp*() can be used to jump to different parts of a program. Practical uses include jumping out of deeply nested loops, invoking error handling routines and jumping to named labels in the code. All of this can be explained with a simple example:

```
#include <setjmp.h>
#include <stdio.h>

void jumper(jmp_buf env) {
    printf("inside jumper function\n");
    longjmp(env, 1);
    printf("after longjmp\n");
}


int main() {

    jmp_buf env;

    int ret = setjmp(env);

    if (ret == 0) {
        printf("setjmp executed\n");
    }
    else {
        printf("returned from longjmp\n");
        return 1;
    }

    jumper(env);
    return 0;
}
```

output

```
setjmp executed
inside jumper function
```

```
returned from longjmp
// main() returns 1
```

jmp_buf is an opaque struct that will hold the execution context of the program. *setjmp(env)* saves the execution context of the program in *env*. *setjmp* always returns 0. The execution context that is saved is the state of the program after *setjmp()* is executed but before the return value 0 is assigned to *ret*.

The program now executes the *setjmp*, evaluates *ret == 0* and continues running. It then calls the *jumper* function with the *env* argument. Inside *jumper*, *longjmp(env, retvalue)* restores the program's execution context. The second argument of *longjmp* is the integer value that *longjmp* will return. If this argument is 0 then *longjmp* will return 1. Now it evaluates *ret == 0* again and continues running.

Note: In C++ *setjmp* and *longjmp* have been largely replaced by exceptions.

The goto Statement

The C and C++ the *goto* keyword is implemented with *setjmp* and *longjmp*. *goto* allows us to jump to a program label:

```
#include <setjmp.h>
#include <stdio.h>

int main() {

   printf("hello world\n");
   goto foo_label;
   printf("1+1=2\n");

   foo_label:
   printf("goodbye\n");
   return 0;
}

output
hello world
goodbye
```

C Language: typedef

We can define new names for existing data types by using the keyword **typedef**. With *typedef* we are not actually creating a new data type, but rather defining a new name for an existing type.

typedef can help make machine-dependent programs more portable. If we define our own type name for each machine-dependent data type used by our program, then only the typedef statements have to be changed when compiling the program on machines with different architectures.

typedef has syntax:

```
typedef old_type_name new_type_name;
```

For example:

```
typedef float balance;
```

balance is just another name for the type *float*;

Now that *balance* is defined as a type, we can do:

```
balance over_due;
```

over_due is a variable of type *balance*.

C Language: Enumerations

An enumeration is a user defined type those values are confined to a set of named integer constants. A variable of such a type can only have a value that is equal to one of the named integer constants that are specified by this type.

The syntax for enumerations is:

```
enum enum_name { enumeration list } variable_list;
```

Both the enum name and the variable list are optional. Here is an example:

```
enum FILETYPE { FILE_NONE, FILE_C, FILE_ASM, FILE_OBJ } FileType;
```

The key point to understand about an enumeration is that each of the symbols in the enumeration list is an integer value. As such, they may be used anywhere that an integer can be used. Each symbol is given a value that is one greater than the symbol that precedes it. The value of the first enumeration symbol is 0.

Here is an example:

```
enum coin { penny, nickel, dime, quarter,  half_dollar, dollar};
enum coin money
```

enum coin is an enumeration type and *money* is a variable of type *enum coin*. Note that the type is *enum coin*, not *coin*.

In C++, an enumeration name is a complete type, so we can declare the variable *money* as:

```
coin money
```

We can also declare enumerations with specific enum values:

```
enum coin {penny, nickel, dime, quarter=25,  half_dollar, dollar=100};
```

Now:

penny	0
nickel	1
dime	2
quarter	25
half-dollar	26
dollar	100

Anonymous Enums

An anonymous enum is defined as follows:

```
enum { COLOR = 1 };
```

This is the same as *int COLOR = 1*; except that the enum *COLOR* is a constant. An anonymous enum behaves like a *const int* except that we cannot get the address of the enum. This is very useful when we want to define *const* integer variables that cannot be addressed through a pointer.

An anonymous enum is effectively a compile-time integer constant.

C Language: size_t And sizeof

size_t is an unsigned integer data type. It is defined in <*stdio.h*>, <*stdlib.h*> and various other headers.

size_t is used to represent the size of objects in bytes. It is the return type of the *sizeof* operator. *size_t* is guaranteed to be big enough in size to hold the largest integer that the host system can handle. The maximum permissible size is dependent on the compiler and hardware architecture. If the compiler is a 32 bit compiler then *size_t* is simply a typedef for *unsigned int* and if the compiler is 64 bit then *size_t* is a typedef for *unsigned long long*. The *size_t* data type is never negative.

The Compile Time Operator sizeof

sizeof is a unary compile-time operator that returns the length in bytes of the variable or parenthesized type that it precedes;

```
double f;

printf("%d", sizeof(f);
printf("%d", sizeof(int));
printf("%d", sizeof(double));
printf("%d", sizeof struct Employee;
```

If the operand of the sizeof operator is a native C or C++ data type then it must be enclosed in parentheses.

sizeof primarily helps to generate portable code that depends upon the size of the built-in data types.

Structures and unions can be used to create variables of different sizes. The actual size of these variables may change from machine to machine. The *sizeof* operator computes the size of any variable or type and can help eliminate machine-dependent code from our programs. This operator is especially useful where structures or unions are concerned.

C Language: Preprocessor

We can include various instructions to the compiler in the source code of a C or C++ program. These instructions are called preprocessor directives. At the outset of the compilation process, the C/C++ preprocessor processes directives in the source files. These directives guide the compiler during the compilation process.

The C++ preprocessor is virtually identical to the one defined by C. The main difference between C and C++ in this regard is the degree to which each relies upon the preprocessor. In C, each preprocessor directive is necessary. In C++, some directives have been rendered redundant by newer and better C++ language constructs. In fact, one of the long-term design goals of C++ is the elimination of the preprocessor altogether.

The preprocessor recognizes the following directives:

#if	#elif	#else	#endif
#error	#define	#ifdef	#ifndef
#include	#line	#pragma	#undef

Preprocessor Syntax

All preprocessor directives begin with a # sign. In addition, each preprocessor directive must be on its own line. If a directive is to continue onto the next line then the directive must terminate each line but the last with a backslash:

```
#define square(x) \
      ((x)        \
        *         \
      (x))
```

The #define Directive

The *#define* directive specifies an identifier and a character sequence (i.e., a set of characters). The character sequence will be substituted for the identifier each time the identifier is encountered in a source file. A *#define* statement is called a macro.

The identifier is called a **macro name** and the replacement characters are called the **macro replacement**. The general form of this directive is:

```
#define macro-name char-sequence
```

Notice that there is no semicolon at the end of this statement. There may be any number of spaces between the macro name and the character sequence, but once the character sequence begins, it is terminated only by a newline.

If we wish to use the value 0 for the word LEFT and the value 1 for the word RIGHT, we can declare these two #define directives:

```
#define LEFT    0
#define RIGHT 1
```

This causes the preprocessor to substitute a 0 or a 1 each time LEFT or RIGHT macro-name is encountered in the source file.

Once a macro name has been defined, it may be used in the definition of other macro names. For example:

```
#define ONE        1
#define TWO        ONE + ONE
```

```
#define THREE         TWO + ONE
```

Macro substitution is simply the replacement of a macro name by the character sequence associated with it.

Note: Macro name substitutions do not occur if the macro name is inside quotation marks.

If the character sequence is longer than one line, we can continue it on the next line by placing a backslash at the end of the line, as shown here:

```
#define LONG_STRING "this is a very long \
string that is used as an example"
```

By convention, upper case letters are used to denote macro names. It is also a convention to place all the program macros in a separate file.

One of the common uses of macros to identify magic numbers.

The problem with using *#define* macros is that the compiler never sees a macro since the preprocessor performs the macro replacement before handing the file over to the compiler. For example, if we specify *#define GREEN 1*, the compiler will never see the identifier *GREEN*. Furthermore, if we fail to include a file containing this macro definition, then the compiler will emit an unknown declaration error each time it sees *GREEN*.

Defining Function-like Macros

The ***#define*** directive has a powerful feature: a macro name can have arguments. Each time a macro name is encountered, the arguments used in its definition are replaced by the actual arguments found in the program. This form of a macro is called a function-like macro:

```
#include <stdio.h>

#define ABS(a)   (a)<0 ? -(a) : (a)

int main(void)
{
   printf("abs of -1 and 1: %d  %d", ABS(-1), ABS(1));

   return 0;
}
```

When this program is compiled, *a* in the macro definition will be substituted with the values –1 and 1. The parentheses that enclose *a* ensures proper substitution in all cases.

The use of function macros is strongly discouraged since it can produce strange results. Look at the following program fragment:

```
// call f with the maximum of a and b
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5, b = 0;
CALL_WITH_MAX(++a, b);              // a is incremented twice
CALL_WITH_MAX(++a, b+10);           // a is incremented once
```

The number of times that a is incremented before calling f depends on what it is being compared with.

The use of a function-like macro in place of real functions has one major benefit: It increases the execution speed of the code because there is no function call overhead. However, if the size of the function-like macro is very large, then this increased speed may be paid for with an increase in the size of the program because of duplicated code.

The NULL Macro

In the C language, NULL is defined as follows:

```
#define NULL 0
```

#error

The *#error* directive emits an error message and forces the compiler to stop compilation. It is used primarily for debugging. The general form of the #error directive is:

#error error-message

An example of usage is:

```
#if __STDC__ != 1
#error "Not a standard compliant compiler"
#endif
```

The #include Directive

The *#include* directive instructs the compiler to read another source file and place it in the source file that contains the *#include* directive. The file will be included at the line where the *#include* directive occurs. The name of the additional source file must be enclosed between double quotation marks or angle brackets.

#include "stdio.h"

or

#include <stdio.h>

Included files can have *#include* directives in them. This is referred to as **nested includes**. The number of levels of nesting allowed varies between compilers. However, Standard C stipulates that at least eight nested inclusions must be available. Standard C++ recommends that at least 256 levels of nesting be supported.

Whether the filename is enclosed in quotes or in angle brackets determines how the search for the specified file is conducted. If the filename is enclosed in angle brackets, then the file is searched for in a manner defined by the creator of the compiler. Often, this means searching some special directories that are set aside for include files. This is the typical way library files are included.

If the filename is enclosed in quotation marks, the file is searched for in an implementation or program creator defined manner. This is the way user files are included. For many compilers, this means searching the current working directory. If the file is not found, the search is repeated as if the filename had been enclosed in angle brackets.

Angle brackets are used to include standard library header files.

Conditional Compilation Directives

There are several directives that allow us to selectively compile portions of our program's source code.

#if, #elif, #else and #endif

The most commonly used conditional compilation directives are the *#if*, *#elif*, *#else*, and *#endif*. These directives allow us to conditionally include portions of code based upon the evaluation of a constant expression.

The *#if p*reprocessor directive has the form:

```
#if constant-expression
    statements
#endif
```

If the constant expression following *#if* is true, the code that is between *#if* and *#endif* is compiled. Otherwise, the intervening code is skipped. The *#endif* directive marks the end of an *#if* block.

```
/* Simple #if example */

#include <stdio.h>

#define MAX 100

int main(void)
{
     #if MAX > 99
         printf("Compiled for array greater than 99.\n");
    #endif

    return 0;
}
```

The expression that follows the *#if* is evaluated at compile-time. Therefore, it must contain only previously defined identifiers and constants—no variables can be used.

The ***#else*** directive establishes an alternative to *#if*, if *#if* fails.

```
/* Simple #if/#else example */
#include <stdio.h>

#define MAX 10

int main(void)
{
 #if MAX > 99
    printf("Compiled for array greater than 99.\n");
 #else
    printf("Compiled for a small array.\n");
 #endif

  return 0;
}
```

The **#elif** directive means *"else if"*. It establishes an if-else-if chain for multiple compilation options. *#elif* is followed by a constant expression. If the expression is true, the block of code attached to the *#elif* is compiled and no other *#elif* expressions are tested. Otherwise, the next block in the *#if* chain is tested.

The general form for *#elif* is:

```
#if expression
statement sequence
#elif expression 1
statement sequence
#elif expression 2
statement sequence
#elif expression 3
statement sequence
#elif expression 4
.
.
.
#else
statement sequence
#endif
```

Example:

```
#define US 0
#define ENGLAND 1
#define FRANCE 2

#define ACTIVE_COUNTRY US

#if ACTIVE_COUNTRY == US
char currency[] = "dollar";
#elif ACTIVE_COUNTRY == ENGLAND
char currency[] = "pound";
#else
char currency[] = "franc";
#endif
```

#ifdef and #ifndef

There is another preprocessor for conditional compilation that uses the *#ifdef* and *#ifndef* directives. These directives mean *"if defined"* and *"if not defined"* respectively. The general form of *#ifdef* is:

```
#ifdef macro-name
statement sequence
#endif
```

If the macro-name has been previously defined in a *#define* statement, the block of code will be compiled.

The general form of **#ifndef** is:

```
#ifndef macro-name
statement sequence
#endif
```

If macro-name is currently not defined by a *#define* statement, the block of code is compiled.

Both *#ifdef* and *#ifndef* directives may use the *#elif* or *#else* directives. For example:

```
#include <stdio.h>

#define __FREE 10

int main(void)
{
    #ifdef __MONTHLY_SUBSCRIPTION
    printf("Monthly\n");
    #elif defined __YEARLY_SUBSCRIPTION
    printf("yearly\n");
    #elif defined __FREE
    printf("Free\n");
    #else
    printf("error\n");
    #endif

    #ifndef ALICE
    printf("ALICE not defined\n");
    #endif

    return 0;
}
```

Using defined

In addition to *#ifdef*, there is a second way to determine if a macro name is defined. We can use the *#if* directive in conjunction with the *defined* compile-time operator. The *defined* operator has the form:

```
defined macro name
```

This expression is true if the macro name is defined and false otherwise. Thus we can also do:

```
#if defined SUM
 ...
#endif
```

We may also precede *defined* with the ! operator to reverse the condition:

```
#if !defined SUM
 ...
#endif
```

#undef

The *#undef* directive removes a previously defined definition of the macro name that follows it. That is, it "undefines" a macro name. The general form for *#undef* is:

```
#undef macro-name
```

*#undef i*s primarily used to allow macro names to be localized to only those sections of code that need them.

#line

The __*LINE*__ identifier contains the line number of the currently compiled line of code. The __*FILE*__ identifier is a string that contains the name of the source file being compiled. #line changes the line number and optionally the file name. The general form for *#line* is:

```
      #line number "filename"
```

where number is any positive integer. This becomes the new value of __*LINE*__ and the optional filename is any valid file identifier, which becomes the new value of __*FILE*__. *#line* is primarily used for debugging and special applications.

#pragma

#pragma is a compiler defined directive that allows various instructions to be given to a compiler. We must check the compiler's documentation for it’s pragma details and options.

C Language: assert

The *assert* macro tests whether an expression is true or false at run-time. The program is aborted if the expression is false. The syntax is:

```
#include <assert.h>

assert(expression);
```

In C++ the idiomatic way to include *assert.h* is: *#include <cassert>*

In C, a zero value of an expression is interpreted as false. Non-zero expressions are interpreted as true.

Here is a C++ example:

```
#include <iostream>
#include <cassert>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    assert(1 == 0);
}

// Output
Hello, World!

prog.exe: prog.cc:9: int main(): Assertion `1 == 0' failed.
Aborted
Finish
```

The *assert* macro can be disabled by including the macro:

```
#define NDEBUG
```

before the *assert* macro.

C Language: Calling C Functions From C++

Introduction

C++ began as an expanded version of C. The C++ extensions were first invented by Bjarne Stroustrup at Bell Laboratories in 1979.

The invention of C++ has been made necessary by one major programming factor: increasing complexity. Computer programs have become larger and more complex. C has its limits once programs become very large. In C, once a program exceeds 25,000 to 100,000 lines of code, it becomes so complex that it is difficult to grasp as a totality.

Header Files

When we use a library function in a program, we must include its header file. This is done by using an *#include* statement. For example, in C we include the header file for the I/O functions like this:

```
#include <stdio.h>
```

In C++ we can do:

```
#include <cstdio>
or
#include <stdio.h>
```

In idiomatic C++, a standard C Library file is included by adding the c prefix to the library name and dropping the *h* suffix.

The filename can be enclosed in quotation marks or in angle brackets. This determines how the search for the specified file is conducted. If the filename is enclosed in angle brackets, the file is searched for in a manner defined by the creator of the compiler. Typically, this means searching some special directories set aside for include files. If the filename is enclosed in quotation marks, the file is looked for in some implementation-defined manner. For many compilers, this means searching the current working directory.

Including A Non-System C Header File In C++ Code

If we are including a C header file that isn't in the C Standard Library, we must wrap the *#include* line in an *extern "C"* construct. This tells the C++ compiler that the functions declared in the header file are C functions.

The *extern* specifier is required because C and C++ have different conventions for placing function arguments and the return value on the stack.

```
// In a C++ file

extern "C" {
#include "my_c_header_file.h"
}
```

If we are including a C header file that isn't provided by the system, and if we are able to change the C header, we should consider adding the extern "C" {...} logic inside the C header file in order to make it easier for C++ users to #include it into their C++ code. Since a C compiler won't understand the extern "C" construct, you must wrap the "C" header file code in an *#ifdef* block so that it won't be seen by the C compiler. The following steps explain how this is done:

Step 1: Put the following lines at the very top of the C header file:

```
#ifdef __cplusplus
extern "C" {
#endif
#include statements
```

```
    ...
```

Step 2: Put the following lines at the very bottom of the C header file:

```
#ifdef __cplusplus
}
#endif
```

Now we can `#include` the header without any `extern "C"` wrapper in our C++ code.

Note: the symbol *__cplusplus* is *#defined* if and only if the compiler is a C++ compiler.

Call A Non-System C function From C++ Code

If we have an individual C function that we want to call from C++ code, and if for some reason we don't have or don't want to `#include` a C header file in which that function is declared, we can declare the individual C function prototype in our C++ code using the `extern "C"` syntax:

```
extern "C" void f(int i, char c, float x);
```

A block of C function prototypes can be declared as follows:

```
extern "C" {
void f(int i, char c, float x);
int g(char* s, const char* s2);
double h(double a, double b);
}
```

The General Form Of A C++ Program

Typically C++ programs will have this general form:

```
#includes
base-class declarations
derived class declarations
nonmember function prototypes

int main()
{
//...
}
nonmember function definitions
```

In large projects, all class declarations will be placed inside header files.

C++ Classes

Encapsulation binds data to the program code that manipulates it. Encapsulation controls access to both the data and the code that manipulates it. This prevents the misuse of the data as well as the code. In an object-oriented language such as C++, code and data are combined to create a self-contained "black box". This black box is a C++ class.

In C++ encapsulation is implemented through classes.

In C++, classes behave like types and are in fact user defined abstract data types. And just like the built-in C++ types such as *int* and *float*, objects can be created from abstract data types.

Classes are created using the keyword *class*. A class declaration declares a new type that encapsulates code and data. A class is a logical abstraction. A physical representation of a class is called an object or an instance of a class. The act of creating an object from a class is called instantiating the class.

A class has the following syntax:

```
class class-name
{

// private data and functions
private:
    ...

// public data and functions
public:
   ...

// protected data and functions
protected:
    ...

} object-list;
```

The object list is optional. If this list is not present, we have a class declaration only.

Access to class members is governed by these three C++ keywords:

```
public
private
protected
```

By default, functions and data declared within a class are *private.* This means that they can only be accessed by other members of the class. The *public* access specifier allows functions and data to be accessible from other parts of the program; that is from outside the class. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

We can change access specifications in a class declaration as often as we like.

Here is an example of a class:

```
class Employee
{
   char name[80];
   double wage;

 public:
    void putname(char *n);
    void getname();
    void putwage(double w);
    double getwage();
};
```

Note that this class contains the function prototypes (or function declarations) only. The actual implementation of a member function will follow outside the class declaration. Nevertheless a class declaration can contain function implementations.

Functions that are declared within a class are called **member functions** or **methods**. Member functions may access any other member of the class of which they are a part.

This includes all private elements.

Member data variables in a class declaration have some limitations:

- No member variable can be declared as an extern or register variable.
- A non-static member variable cannot be initialized through an assignment (at the point of it’s declaration in the class).

As a general rule, we should enforce encapsulation of data. That is, all of the data members in a class should be declared as private.

Accessing Public Members

The syntax for accessing a public member is as follows: Specify the object's name, then the dot operator, and then the member variable or member function.

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

class Foo
{
public:
   // these data variables are accessible to the entire program
   int i, j, k;
};


int main()
{
  Foo a, b;
  // access to i, j, and k is OK
  a.i = 100;
  a.j = 4;
  a.k = a.i * a.j;

  b.k = 12;                // different class instance

  cout << a.k << " " << b.k;
  return 0;
}

// output
400 12
```

The Scope Resolution Operator

The **::** scope resolution operator associates a member with a class name. This tells the compiler which class the member belongs to. The scope resolution operator also has another related use: it can allow access to a name within some scope. This allows access to a name in some scope that is identical to a local declaration with the same name.

The following example shows how this works:

```
int i;    // global variable i

void f()
{
   int i;          // local variable i
   i = 10;         // uses local i
}


void g()
{
   int i;           // local variable i
   ::i = 10;        //  refers to the global variable i
}
```

Class Function Implementations

Consider the following class specification:

```
#include <iostream>
#include <cstring>

using namespace std;

class Foo
{
public:
   void buildstr(char *s);
   void showstr();

private:
   char str[255];
};

void Foo::buildstr(char *s)
{
  // initialize the member string variable
  strcpy(str, s);
}

void Foo::showstr()
{
  cout << str << "\n";
}
```

The class in this example only contains the class member function prototypes. The actual class function implementations follow the class and use the scope resolution operator to identify the class to which each function belongs. This is the normal way in which a class is declared. We can however provide function implementations within a class.

Static Class Members

Member functions and member variables of a class can be declared as static. Static members are shared by all of the instances of a class. The following restrictions apply:

- A static member function may only refer to other static member functions and static data members of the class
- A static member function does not have a *this* pointer.
- We cannot have a static and non-static version of a member function.
- A static function cannot be declared with a *const* or *volatile* attribute.

The following is an example:

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

class Foo
{
   static int resource;
public:
   static int get_resource();
   void free_resource() { resource = 0; }
};


int Foo::resource = 0;           // initialize resource

int Foo::get_resource()
{
     if(resource)
       return 0;
     else
     {
       resource = 1;
       return 1;
     }
};

int main()
{
   int ret = Foo::get_resource();
   cout << "resource = " << ret << endl;
   return 0;
}

// output
resource = 1
```

Note that in this example, *Foo* is not instantiated.

Object Assignment

Assuming that two objects are of the same type, we can assign one object to another using the usual assignment operator =. This causes the data of the object on the right side to be bitwise copied into the object on the left.

Local Classes

A local class is a class defined within a function. The following example defines a class within a function:

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

// forward declaration of function f
void f();

int main()
{
  f();
  return 0;
}


void f()
{
  // define a local class inside f
  class Foo
  {
   int i;
   public:
    void put_i(int n) { i=n; }
    int get_i() { return i; }
   };

   Foo foo;

   foo.put_i(10);
   cout << "foo i data member = " << foo.get_i() << endl;
}

// output
foo i data member = 10
```

There are stringent restrictions on the use of local classes:

- all member functions must be defined within the class declaration.
- The local class may not access any local variables of the function in which it is declared; except for static variables declared inside the function.
- Static variables cannot be declared inside a local class.

A local class may access typedefs and enumerators defined in the enclosing function.

C++ Namespaces

The purpose of namespaces is to localize the names of identifiers in order to avoid name collisions. Prior to the invention of namespaces, top-level names competed for slots in the global namespace and thus conflicts could occur. For example, if our program defined a function called *abs*(), it could (depending upon its parameter list) override the standard library function *abs*() because both names would be stored in the global namespace.

The creation of the namespace keyword solves this problem. A namespace allows the same name to be used in different contexts without conflicts arising. This is done by localizing the visibility of names to the namespace in which they are declared.

The **namespace** keyword allows us to partition the global namespace by creating a declarative region. A namespace defines a scope for all of the identifiers that are declared in it. The syntax for a namespace is:

```
namespace name {
     // declarations
}
```

The visibility of any name defined within a namespace block is confined to the scope of the namespace.

```
namespace counter_namespace {

     int upperbound;
     int lowerbound;

     class Counter
     {
       int count;
      public:
        counter(int n) {
           if(n <= upperbound)
             count = n;
```

```
              else
                count = upperbound;
           }

        void reset(int n)
        {
          if(n <= upperbound)
           count = n;
        }

        int run()
        {
          if(count > lowerbound)
            return count--;
          else
            return lowerbound;
        }
     };

    } // namespace closed
```

Here the visibility of *upperbound*, *lowerbound*, and the class *Counter* is confined to the scope defined by the c*ounter_namespace* namespace.

Identifiers declared within a namespace block can be referred to in the namespace without any namespace qualification. For example, within the namespace c*ounter_namespace*, the *run*() function can refer directly to *lowerbound* in it's body. However, since a namespace defines a scope, we need to use the scope resolution operator in order to refer to objects declared within a namespace from outside that namespace.

For example, to assign the value 10 to *upperbound* from code outside c*ounter_namespace*, we must use the scope resolution operator:

```
counter_namespace::upperbound = 10;
```

and to declare an object of type *Counter* from outside CounterNameSpace, we use:

```
counter_namespace::Counter foo;
```

The Global Namespace

If an identifier is not declared in any namespace, it is part of the global namespace. In general, we should avoid making declarations in the global namespace, except for the entry point function *main* which is required to be in the global namespace.

To use an identifier such as *someFunction()* that is in the global namespace, we use the scope resolution operator without any prefix, as in:

```
::someFunction(x);
```

Best Practise

Whenever you want to refer to a global variable (it is in the global namespace), preface the variable with the scope resolution operator. This will let us know that the variable is defined in the global namespace. Example:

```
foo(int x) {
  return x + ::y;
}
```

Bringing An Identifier Into Scope

We can bring an identifier into the scope of another namespace by qualifying the identifier with the scope resolution operator. The syntax is as follows:

```
 void foo()
{
   std::cout << std::string("Hello World");
}
```

This brings *cout* and the string class which are in *std* namespace into scope.

Importing Namespaces: The using Directive

If our program includes frequent references to identifiers in a namespace, having to specify the namespace and the scope resolution operator each time we refer to such identifiers quickly becomes tedious. The *using* statement alleviates this problem. The *using* statement has these two general forms:

```
using namespace foo;

using foo::bar;
```

In the first form, *foo* specifies the name of the namespace we want to access. All of the members defined within the *foo* namespace are brought into scope (they become part of the current namespace) and may be used without a scope resolution operator (for example, *vector* instead of *std::vector*). In the second form, only a specific member *bar* of the *foo* namespace is brought into scope.

Given c*ounter_namespace* as shown above, the following *using* statement and assignment is valid:

```
 // only lowerbound is visible
using counter_namespace::lowerbound;

// OK because lowerbound is visible in the current scope
lowerbound = 10;
```

The Dangers of using namespace

It is considered a bad practise to declare *using namespace <some identifier>* in a header file since this introduces the namespace throughout the entire program.

The declaration of using namespace in implementation files is also discouraged since it can give rise to very difficult to debug ambiguities. Consider this example:

```
#include <iostream>

namespace my_library {
  double calculate(double x) {  return x + x; }
}

namespace foreign_lib {
 int calculate(int x) {  return x + x; }
}

using namespace my_library;
using namespace foreign_lib;


int main() {

     calculate(2);
     calculate(2.3):
}
```

The problem with this code is that the version of *calculate* that gets called depends on the type of the *calculate* parameter. This can introduce a very difficult to trace error into the program.

The best practise is to always prefix an identifier with it’s namespace.

Unnamed Namespaces

There is a special type of namespace, called an unnamed namespace, that allows us to create identifiers that are unique within the namespace and invisible outside the namespace.

Unnamed namespaces are also called anonymous namespaces. They have this syntax:

```
namespace {
     // declarations
}
```

Unnamed namespaces allow us to establish unique identifiers that are known only within the scope of a namespace. Consider a file those contents are wrapped by an unnamed namespace. Within the file that contains the unnamed namespace, the members of the namespace may be used directly without qualification. But outside the file, the identifiers are unknown and cannot be addressed.

Joining Namespaces

There can be more than one namespace declaration with the same (namespace) name. This allows a namespace to be split over several files or even separated within the same file.

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

namespace NS {
     int i;
}

//
// intervening code

namespace NS {
     int j;
}

int main()
{
     NS::i = NS::j = 10;

     // refer to NS specifically
     cout << NS::i * NS::j << endl;

    // use NS namespace
     using namespace NS;
     cout << i * j << endl;

}

// output
100
```

```
100
```

In this example, the namespace NS is split into two pieces. However, the contents of each piece are still within the same namespace NS.

A namespace can be nested inside another namespace. However a namespace cannot be declared inside the body of a function. However we can use *using namespace name* within a function.

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

namespace NS1 {
   int i;
     namespace NS2 { // a nested namespace
          int j;
        } // end of NS2
} // end of NS1


int main()
{
  NS1::i = 19;
  // NS2::j = 10;            // Error, NS2 is nested inside NS1
  NS1::NS2::j = 10;          // this is right

  cout << NS1::i << endl;

  // use NS1
  using namespace NS1;

  // Now that NS1 is in view, NS2 can be
  // used by itself to refer to j.
  cout << i * NS2::j;

  return 0;
}

// output
19
190
```

The std Namespace

The Standard C++ library defines its entire library in its own namespace called *std*.

```
using namespace std;
```

This *using* statement causes the identifiers in the *std* namespace to be brought into the current namespace, which gives us direct access to the names of the functions and classes defined in the library without having to qualify each one with *std*.

Namespace Alias

A namespace identifier can be replaced by an alias. For example:

```
namespace newfarm = farm;
```

Now the *newfarm* alias can now be used instead of the *farm* namespace:

```
newfarm::pets = 5;     // equivalent to farm::pets;
```

```
#include <iostream>

namespace T {
  int ctr = 0;
}

int main()
{
  namespace Z = T;
  std::cout << Z::ctr << std::endl;
}

//output
0
```

C++ Structs And Unions

In C++, structs are simply classes in which all the data members and member functions are public by default. Unlike C, a C++ struct can contain member functions.

Here is an example of a struct declaration in C++:

```
// A C++ struct

#include <iostream>
#include <cstring>

using namespace std;

struct Foo
{
  // public
  void buildString(const char *s);
  void showString();

 private:
  char str[255];
};

void Foo::buildString(const char *s)
{
  // initialize string
  strcpy(str, s);
}

void Foo::showString()
{
   cout << str << "\n";
}

int main()
{
  Foo foo;
  foo.buildString("hello world");
  foo.showString();
}

// output
hello world
```

Notice that we can use a struct wherever we can use a class. Usually it is best to use a class when we want a class, and a struct when we want a C-like structure.

Unions And Classes

In C++, unions can contain both member functions and data members. They can also include constructor and destructor functions. A union in C++ retains all of its C features. In a union, all of the data elements share the same location in memory. Like a struct, union members are public by default.

Here is an example:

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

union swapByte
{
  void swap();
  void setByte(unsigned short i);
  void showWord();

 private:
  unsigned short u;
  unsigned char c[2];
};

void swapByte::swap()
{
    unsigned char t;

    t = c[0];
    c[0] = c[1];
    c[1] = t;
}

void swapByte::showWord()
{
    cout << u;
}

void swapByte::setByte(unsigned short i)
{
    u = i;
}

int main()
```

```
{
  swapByte foo;
  foo.setByte(49034);
  foo.swap();
  foo.showWord();
}

// output
35519
```

Unions in C++ have several restrictions:

- a union cannot inherit from any class
- a union cannot be used as a base class for some other class
- a union cannot contain virtual functions
- a union cannot contain static members
- a union cannot overload the = operator
- no object can be a member of a union if the object has an explicit constructor or destructor function.

Anonymous Unions

There is a special type of union in C++ called an anonymous union. An anonymous union does not have a name, and no objects of an anonymous union can be created. Instead, an anonymous union tells the compiler that it's data members share the same memory location. Anonymous unions must only contain public data members.

The data members in the anonymous union are referred to directly, without the normal dot operator syntax. For example, consider this program:

```
#include <iostream>
#include <cstring>

using namespace std;
```

```
int main()
{
  // define an anonymous union
  union
  {
    long l;
    double d;
    char s[4];
  };

  // now, reference the union elements directly
   l = 100000;
   cout << l << " ";

   d = 123.2342;
   cout << d << " ";

   strcpy(s, "hi");
   cout << s;

}

// output
100000 123.234 hi
```

The elements of the union are referenced as if they had been declared as normal local variables.

All of the restrictions involving unions apply to anonymous unions. Global anonymous unions must be specified as static.

C++ Friend Functions And Friend Classes

We can grant a non-member function access to the private members of a class by using the *friend* keyword. A friend function has access to all of the private and protected members of the class of which it is a friend. To declare a friend function, include its declaration within the class, preceding the declaration with the keyword *friend*.

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

class Foo
{
int a, b;
public:
   friend int sum(Foo);
   void setAb(int i, int j);
};

void Foo::setAb(int i, int j)
{
   a = i;
   b = j;
}

// Note: sum() is not a member function of any class.
int sum(Foo foo)
{
  /* Because sum() is a friend of Foo, it can
     directly access the private members a and b.
  */
     return foo.a + foo.b;
}

int main()
{
  Foo klass;
  klass.setAb(3, 4);
  cout << sum(klass) << endl;
}

// output
7
```

In this example, the *sum()* function is not a member of *Foo*. However, it still has full access to the private members of *foo* since it is a friend of *Foo*.

Forward Declarations Or Forward References

A forward declaration is required when an object is used before it has been declared. The following example shows how a forward declaration of a class is made:

```
// forward declaration of class C2
class C2;

class C1
{
  int status;
public:
  void setStatus(int);
  int getStatus(C2);
};
```

Friend Classes

It is also possible to make a class a friend of another class. In this case, all of the member functions of the friend class will have access to the private and protected members of the host class. For example:

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

// forward declaration of class Min
class Min;

class TwoValues
{
  int a;
  int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;

};

class Min
{
 public:
   int minimum(TwoValues x);
};

int Min::minimum(TwoValues x)
{
   return x.a < x.b ? x.a : x.b;
}

int main()
{
  TwoValues obj(10, 20);
  Min m;

  cout << m.minimum(obj);
  return 0;
}

// output
10
```

C++ Inline Functions

In C++, we can create short functions that are not actually called; rather, their code is expanded inline at the point of declaration. This results in faster code since the stack is not used to load parameters. We can cause a function to be expanded inline rather than being called, by preceding its declaration with the *inline* keyword.

```
#include <iostream>

inline int max(int a, int b)
{
   return a > b ? a : b;
}

int main()
{
  std::cout << max(10, 20) << std::endl;
  std::cout << max(99, 88) << std::endl;
  return 0;
}

// output
20
99
```

The reason that inline functions are an important addition to C++ is that they allow us to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. Each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then these registers are restored when the function returns. The problem is that these instructions are time-consuming. However, when a function is expanded inline, none of those operations occur.

Although expanding function calls inline can produce faster run-times, it will also result in larger code size because of code duplication. For this reason, it is best to use the inline attribute only for very small functions. Furthermore, it is also a good idea to inline only those functions that will have a significant impact on the performance of our program (they are called frequently).

inline is actually just a request and not a command to the compiler. The compiler can choose to ignore it.

Defining Inline Functions Within a Class

Class member function can be declared inline implicitly or explicitly as follows:

```
#include <iostream>

class Foo
{
  int a, b;
public:
  void init(int i, int j);
  // implicitly inline
  void show() { std::cout << a << " " << b << "\n"; };
};

// Explicitly create an inline member function.
inline void Foo::init(int i, int j)
{
  a = i;
  b = j;
}


int main()
{
   Foo foo;
   foo.init(1, 3);
   foo.show();
}

// output
1 3
```

As this example shows, we can define inline functions completely within a class or explicitly outside the class declaration. When a function is defined inside a class, it is automatically made into an inline function (if possible). It is not necessary to precede the function declaration with the *inline* keyword in this case.

C++ Constructors And Destructors

A constructor is a member function that has the same name as the class. A constructor is automatically called when an instance of a class is created.

We can pass arguments to constructor functions. These arguments initialize the object when it is created (instantiated):

```
#include <iostream>

class Foo
{
   int a, b;

public:
   // constructor
   Foo(int i, int j) { a=i; b=j;}

   void show() { std::cout << a << " " << b << std::endl; }
};

int main()
{
   Foo obj(3, 5);
   obj.show();
}

// output
   3 5
```

Notice that **Foo obj(3, 5)** in *main* causes the arguments of *obj* to be passed to the constructor.

Constructors With One Formal Parameter

When a class has only one parameter in it’s constructor function, we can initialize an object of this class as follows:

```
#include <iostream>
```

```
class Foo
{
  int a;

public:
 // constructor
 Foo(int j) { a = j; }

 int get() { return a; }
};

int main()
{
   // passes 99 to the constructor
   Foo obj = 99;

   std::cout << obj.get() << std::endl;
   return 0;
}

// output
99
```

In the function *main()*, *Foo obj = 99* is equivalent to *Foo obj(99).*

The reason that this assignment works is because whenever we instantiate a class object that takes one argument in it's constructor, the compiler performs an implicit conversion from the type of the variable on the right-hand side to the type of the class.

Initializing An Object

Consider the following class:

```
class Foo {
int num;
string str1;
string str2;

public:
  Foo(int n, string a, string b) {
    num = n;
    str1 = a;
    str2 = b;
  }
};
```

```
int main() {
    Foo foo(1, “hello”, “world”);
}
```

C++ stipulates that except for data members which are built-in C++ types, the data members of an object will be initialized before the body of a constructor is entered. This means that the data members *str1* and str2 have already been constructed (by calling their default constructors) before the body of the *Foo* constructor is entered. The data member *num* is left in its uninitialized state. Inside the constructor body, values are then assigned to all of the data members. This is inefficient since the string type data members are in effect being initialized twice. The more efficient solution is to initialize the *foo* object as follows:

```
Foo(1, “hello”, “world”) : num(1), str1(“hello”), str2(“world”) { };
```

This will initialize the data members only once. The expression to the right of the colon is a member initializer list. The compiler will automatically call default constructors for the data members with types that are not native C++ types.

Data members with built-in types that are not initialized in a member initialization list are left uninitialized, unless they are initialized in the constructor’s body.

const and reference data members require values to be assigned to them. The place to do this is in the member initializer list.

Data members are always initialized in the order that they are declared in the class and not in the order of the initialization list.

Calling Constructors And Destructors

A destructor function has the same name as the class name with a ~ prefix.

An object's constructor is called when the object comes into existence, and an object's destructor is automatically called when the object is destroyed.

Global objects have their constructor functions execute before *main* begins execution. Global constructors are executed in the order of their declaration, within the same file. The order of execution of global constructors is unknown when they are spread among several files. Global destructors execute in reverse order after *main* has terminated.

A local object's constructor function is executed when the object's definition is encountered. The destructor functions for local objects are executed in the reverse order of the constructor functions.

Private Destructors

Consider the following example:

```
#include <iostream>

class Foo
{
private:
  ~Foo();
};

int main() {
  Foo foo;
}

// output
 error: 'Foo::~Foo()' is private within this context
```

This program will not compile. The reason is that *foo* is created on the stack and once *foo* leaves it’s scope, the destructor cannot be called since it is private. Making a class destructor private ensures that only the class object can destroy itself. For example, a reference counting object can destroy itself when it’s count reaches zero.

Rule: if a class has a private destructor then it should be created on the heap (not on the stack).

Passing Objects To Functions

Objects can be passed to functions by using the default *call by value* mechanism. This means that a copy of the object is made when it is passed to a function. The function receives a bit copy of the object.

However, the fact that a copy is created means, in essence, that another object is created. This raises the question of whether the object's constructor function is executed when the copy is made and whether the destructor function will be executed when the copy is destroyed.

```
// Passing an object to a function
#include <iostream>

class Foo
{
  int i;
public:
  Foo(int n);
  ~Foo();

  void set(int n) { i=n; }
  int get() { return i; }
};

// constructor for Foo
Foo::Foo(int n)
{
  i = n;
  std::cout << "Foo Constructor: " << i << "\n";
}

// destructor for Foo
Foo::~Foo()
{
  std::cout << "Destroying foo " << "\n";
}

// forward declaration for function func
void func(Foo obj);
```

```
int main()
{
  Foo foo(1);
  func(foo);          // foo object is passed by value

  std::cout << "This is i in foo = ";
  std::cout << foo.get() << "\n";
}

void func(Foo obj)
{
  obj.set(2);
  std::cout << "This is i in func: " << obj.get() << "\n";
}

// output
Foo Constructor: 1
This is i in func: 2
Destroying foo
This is i in foo = 1
Destroying foo
```

Notice that two calls to the destructor function are executed, but only one call is made to the constructor function. As this output shows, the constructor function is not called when a copy of *obj* is passed to *func*. The reason that the constructor function is not called when the copy of the object is made is easy to understand. When we pass an object to a function, we want to pass the current state of the object. If the constructor is called when the copy is created, initialization will occur, possibly changing the object. Thus, the constructor function is not executed when a copy of an object is generated for a function call.

Although the constructor function is not called when an object is passed to a function, it is necessary to call the destructor when the copy is destroyed. The object copy could be performing operations such as memory allocation that will require the destructor function to be called when the copy is destroyed.

By default, when a copy of an object is made, a bitwise copy is made. This means that the new object is an exact duplicate of the original. The fact that an exact copy is made can be a source of trouble. For example, suppose that an object has allocated memory and then this object is passed by value to a function. When the function exits, the copy's destructor will be automatically called and the memory allocated by the original object will be freed. This will leave the original object damaged state.

Returning Objects

A function may return an object to the caller:

```
// Returning objects from a function.
#include <iostream>

class Foo
{
  int i;
public:
  void set(int n) { i=n; }
  int get() { return i; }
};

// forward declaration; func returns an object of type Foo
Foo func();

int main()
{
  Foo obj = func();
  std::cout << obj.get() << "\n";
}

// define func()
Foo func()
{
  Foo foo;
  foo.set(99);

  return foo;
}

// output
99
```

When an object is returned by a function, a temporary object is automatically created on the stack. It is this object that is actually returned by the function. In our example, the returned object is assigned to the local object *obj (*by a bitwise copy*)* and then the returned object *foo* on the stack is destroyed.

The destruction of this temporary object on the stack may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, then this memory will be freed even though the object that is receiving the return value is going to use it.

C++ Memory Allocation

C++ supports the C memory allocation functions, *malloc, calloc, realloc* and *free*. In addition, C++ provides two more memory operators: *new* and *delete*. These operators are used to allocate and free memory at run-time.

The *new* operator allocates memory on the heap and returns a pointer to the start of this memory. The *delete* operator frees memory previously allocated by *new*. The general forms of *new* and *delete* are:

```
#include <new>

type_t* ptrMem = new type_t;

delete ptrMem;
```

Here, *ptrMem* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type_t*.

Since the heap is finite, it can be exhausted. If there is insufficient memory available to fill an allocation request, *new* will fail with a *bad_alloc* error. This error is actually an exception that we will discuss in a later chapter. *bad_alloc* is defined in the header file *<new>*. The program should handle this error and take appropriate action if a failure occurs. The program will be aborted if this error is not handled by our program

The following program allocates memory from the heap to hold an integer:

```
#include <iostream>
#include <new>

using namespace std;

int main()
{
  int *p;

  try {
    // allocate memory for an int
    p = new int;
  }

  catch (bad_alloc& xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  *p = 100;

  cout << "At memory location " << p << endl;
  cout << "the value is " << *p << "\n";

  delete p;

  return 0;
}
```

This example uses a *try...catch* exception block that we will discuss subsequently.

The *delete* operator must only be used with a valid pointer previously returned by *new*. Using any other type of pointer with *delete* is undefined and will almost certainly cause serious problems such as a system crash.

Although *new* and *delete* perform functions similar to *malloc()* and *free()*, they have several advantages. Firstly, *new* automatically allocates enough memory to hold an object of the specified type. We do not need to use the *sizeof* operator to get the size of the object. Since the size is computed automatically, it eliminates any possibility of an error in this regard. Second, *new* automatically returns a pointer to the specified type. We don't need to use an explicit type cast as we do when allocating memory with *malloc*. Finally, both *new* and *delete* can be overloaded, allowing us to create customized allocation systems.

Initializing Allocated Memory

We can initialize allocated memory for a scalar type by putting the initializer after the type name in the *new* statement:

```
scalarType* pMem = new scalarType (45);
```

The parenthesis must not be empty.

Allocating Memory For Arrays

We can allocate memory for arrays using *new* as follows:

```
array_type* ptrMem = new array_type[size];
```

size specifies the number of elements in the array.

To free an array, use this form of *delete*:

```
delete [] ptrMem;                     // or delete []ptrMem
```

The [] informs *delete* that an array is being released.

Here is an example:

```
#include <iostream>
#include <new>

using namespace std;

int main()
{
    int *p, i;
    try {
       // allocate an integer array with 10 elements
        p = new int[10];
    }
    catch (bad_alloc& xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    // initialize the array
    for(i=0; i < 10; i++)
          p[i] = i;

    for (i = 0; i < 10; i++)
          cout << p[i] << " ";

    delete [] p; // release the array

}

// output
0 1 2 3 4 5 6 7 8 9
```

Allocating Memory For Objects

We can allocate memory for objects with *new*. When we do this, an object is created and a pointer to the object is returned. This dynamically created object acts just like any other object. When it is created, its constructor function (if it has one) is called. When the object is freed, it’s destructor function is executed.

```
#include <iostream>
#include <new>
#include <cstring>

using namespace std;

class Balance
{
  double cur_bal;
  char name[80];
public:
  void set(double bal, const char *s)
  {
    cur_bal = bal;
    strcpy(name, s);
  }

  double get_bal() {
    return cur_bal;
  }

};
```

```
int main()
{
   Balance *p;
   double bal;

   try {
        p = new Balance;
   }
   catch (bad_alloc& xa) {
        cout << "Allocation Failure\n";
        return 1;
   }

   p->set(12387.87, "alice");
   bal = p->get_bal();

   cout << "balance is: " << bal << endl;

   delete p;
}


//output
balance is: 12387.9
```

The Balance object is allocated with the following syntax:

```
p = new Balance;
```

The syntax for deleting the object is:

```
delete p;
```

C++ Arrays

In C++, it is possible to have an array of objects. The syntax for declaring and using an object array is the same as it is for any other type of array. In the following example, we create an array of three objects of type *Foo*:

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

class Foo
{
int i;
public:
  void set(int j) { i=j; }
  int get() { return i; }
};

int main()
{
  Foo foo[3];
  int i;

  for (i = 0; i < 3; i++) foo[i].set(i+1);
  for (i = 0; i < 3; i++)
      cout << foo[i].get() << " ";

}

// output
1 2 3
```

Array Initialization With Single Parameter Constructors

If a class *Foo* contains a single parameter constructor, then an array of *Foo* objects can be initialized as follows:

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

class Foo
{
int i;
public:
  Foo(int j) { i=j; }             // single parameter constructor
  int get() { return i; }
};

int main()
{
  Foo obj[3] = {1, 2, 3};         // initializer list
  int i;

  for(i = 0; i < 3; i++)
     cout << obj[i].get() << " ";

  return 0;
}

// output
1 2 3
```

If a class constructor has two or more arguments, we will have to use the following initialization syntax:

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

class Foo
{
  int h;
  int i;
  public:
  // constructor with 2 parameters
  Foo(int j, int k) { h=j; i=k; }
  int get_i() {return i;}
  int get_h() {return h;}
};

int main()
{
  Foo obj[3]  {
              Foo(1, 2),      // initializer list of constructors
              Foo(3, 4),
              Foo(5, 6)
             };

  int i;

  for (i = 0; i < 3; i++)
  {
    cout << obj[i].get_h();
    cout << ", ";
    cout << obj[i].get_i() << "\n";
  }

}


// output
1, 2
3, 4
5, 6
```

Creating Initialized And Uninitialized Arrays

A special case occurs if we want to create both initialized and uninitialized arrays of objects. Consider the following class.

```
class Foo
{
int i;
public:
  Foo(int j) { i=j }
  int get() { return i; }
};
```

The following array declaration for *Foo*: *Foo ar[9];* will fail because each object in the array must be instantiated by invoking the constructor and *Foo* does not have an appropriate constructor. Compilation will succeed only if *Foo* also has a constructor with no parameters.

We can specify an array that handles initialized and uninitialized objects by overloading the Foo constructor:

```
class Foo
{
int i;
public:
  // constructor for non-initialized arrays
  Foo() { i = 0; }

  // constructor for initialized arrays
  Foo(int j) { i = j; }
  int get_i() { return i; }
};
```

Given this class declaration, the following are valid object instantiations:

```
Foo a[3] = {3, 5, 6};      // initialized array
Foo b[34];                 // uninitialized array
```

Constructing Arrays Dynamically

Suppose that we want to create an array of fifty double values dynamically. The following is the syntax for allocating this array on the heap:

```
double* ptrMem = new double[50];
```

We free this memory allocation with:

```
delete []ptrMem;
```

C++ Pointers

Like pointers to variables, we can have pointers to objects. Given a pointer to an object, we access members of the object with the arrow (–>) operator instead of the dot operator. The following program illustrates how to access an object given a pointer to it:

```
#include <iostream>

class Foo
{
int i;
public:
  Foo(int j) { i=j; }
  int get() { return i; }
};

int main()
{
  // p is a pointer to a Foo object
   Foo* p;
   Foo obj(77);

  // get the address of obj
  p = &obj;

  // use -> to call get()
  std::cout << p->get() << std::endl;
  return 0;
}

// output
77
```

Incrementing Pointers

When a pointer is incremented, it points to the next element of the type that it points to. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the type of the object or native type pointed to.

Since a type or a class has a size in bytes, incrementing a pointer increments the pointer address by a number of bytes equal to the size of the type or object pointed to.

Pointers To Member Functions

Suppose that we have a class *Foo* that has a public member function *add* with the following declaration: *int add(int, int)*.

Then a pointer to a member function such as *add* is declared as follows:

```
return_type (Class_Name::* pointer_name) (Argument_List);
```

This states that *pointer_name* is a pointer to a member function in the named class and this member function has a list of type parameters and returns a certain type. An example of a list of type parameters is: *(string, double, int)*

Consider the following class:

```
class Foo
{
public:
  int adder(int a, int b) {
     return a + b;
  }
};
```

We declare a pointer to a member function of type *adder* as follows:

```
int (Foo::*pfn)(int, int);
```

A pointer to the member *adder* function is defined as follows:

```
// declaration of the member function pointer
int (Foo::*pfn)(int, int);

// initialize the member function pointer
pfn = &Foo::adder;
```

Note that this is a pointer to a class member function and thus it does not hold an exact address like a regular function pointer. Instead it holds an offset address to the member function relative to the class address.

We can call the member function *adder* for a particular instantiated object *foo* through the pointer *pfn* as follows:

```
int sum = (foo.*pfn)(11, 5);
```

**pfn* dereferences the offset address to the class member function *adder. (foo.*pfn)* becomes *(foo.adder)*. And the right hand side reduces to: *(foo.adder)(11, 5)*.

Note: the parentheses are important.

Here is a complete example that uses pointers to member functions:

```
#include <iostream>

class Foo
{
public:
   int adder(int a, int b) {
        return a + b;
   }
};

int main()
{
  int sum = 0;
  Foo foo;
  // declare a pointer to a class member function in class Foo
  int (Foo::*pfn)(int, int);

  // initialize the member function pointer
  pfn = &Foo::adder;
  // invoke the member function adder in the foo object
  sum = (foo.*pfn)(11, 5);
  std::cout << sum << std::endl;
}

// output
16
```

Type Checking C++ Pointers

We can assign one pointer to another only if the two pointer types are compatible. Consider:

```
int   *pi;
float *pf;
```

Then the following is an illegal assignment:

```
pi = pf;                      // error type mismatch
```

We can override this by explicitly casting *pf* to an integer type pointer:

```
pi =  (int *) pf;
```

Pointers To Data Members In A Class

Consider this code fragment:

```
int Foo::*d;       // pointer to an int data member of Foo
int *p;            // pointer to an integer

Foo obj;

p = &obj.val   // this is the address of a specific val in obj
d = &Foo::val  // this is an offset to the data member val in the class
```

Here, *p* is a pointer to an integer inside a specific object. However, *d* is simply an offset that indicates where *val* will be found in any object of type *Foo*. We also have:

```
p == foo.*d
```

The this Pointer

Every C++ object has a *this pointer* which is a pointer to the object itself. The value of the *this* pointer is the address of the object.

When a non-static member function is called, it is automatically passed the *this* pointer as an implicit argument.

Consider the following program:

```
#include <iostream>

class Foo
{
int i;
public:
  Foo(int x) { i = x; }
  int square()                // this pointer passed implicitly
  {
     return this->i*this->i;
  }
};

int main()
{
  Foo foo(5);
  std::cout << foo.square() << std::endl;

}

// output
25
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside *square()*, the statement:

```
return this->i * this->i;
```

is equivalent to:

```
      return i*i;
```

Note that friend functions are not members of a class and, therefore, are not passed a *this* pointer. Secondly, a static member function does not have a *this* pointer since it is not uniquely associated with any instantiated object.

C++ References

C++ contains an important feature that is related to pointers called references. A reference is an implicit pointer variable that can be manipulated as if it is the actual object or variable that it points to.

The syntax for creating a reference variable is as follows:

```
#include <iostream>

int main() {

    int z = 5;
    int& x = z;                      // or int &x = z;

    std::cout << x << std::endl;

}

# output
5
```

Here *x* is a reference to a variable of type *int. A* reference variable must be initialized when it is created. The reference variable must point to some initialized variable when it is declared.

A declaration such *int& x = 5;* will not compile since 5 is an ephemeral value that does not have an address and so cannot be pointed to (*more on this later in the section on right hand values*). Furthermore the literal value 5 will disappear once the statement is executed.

There are three ways to use a reference variable: as a function parameter, as a function return value, and as a stand-alone variable. In the previous example, x is a stand-alone reference variable.

Function Reference Parameters

The most important use of a reference variable is to create functions that use call by reference parameter passing. Arguments can be passed to a function in one of two ways: by using ***call by value*** or by using ***call by reference***. When call by value is used, a copy of the argument is passed to the function. Call by reference passes the address of the argument to the function.

By default, C++ uses call by value for passing parameters to a function, but it also provides two ways to pass an argument by reference. Firstly, we can explicitly pass a pointer variable to a function. Secondly, we can specify that the function receives a reference parameter. In most circumstances the best practise is to declare that the function receives a reference parameter instead of passing a pointer to the function.

The following example demonstrates how to implement call by reference using pointers.

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

// forward declaration of the negative function
void negative(int* i);

int main()
{
  int x = 10;
  cout << "x is: " << x << endl;
  negative(&x);
}

void negative(int* i)
{
  *i = -*i;
  cout << "x in negative function is: " << *i << endl;
}
```

```
// output

x is: 10
x in negative function is: -10
```

In this example, we have passed a pointer to the function *negative* and printed the value of x by dereferencing the pointer.

To specify that a function parameter is a reference variable, we precede the parameter's name with an **&**. For example, here is how to specify *negative* with i declared as a reference parameter:

```
void negative(int& i);                // or void negative(int &i)
```

i is an implicit pointer variable but it can be manipulated as if it is the variable that it points to. Once *i* has been made into a reference, it is no longer necessary (or even legal) to apply the * pointer dereference operator to it. *i* is an alias for the variable that it points to. That is, *i* can be manipulated as the variable itself. Furthermore, when calling *negative*, it is no longer necessary (or legal) to precede the argument's name with the & address operator. We pass the actual variable *x*. The compiler will silently pass an implicit pointer to the function.

Here is the reference variable version of the preceding program:

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

// forward declaration of the negative function
void negative(int& i);

int main()
{
  int x = 10;
  cout << "x is: " << x << endl;
  negative(x);
  cout << "x after the call to negative is: " << x << endl;
}

void negative(int& i)
{
  i = -i;
  cout << "i in negative function is: " << i << endl;
}

// output
x is: 10
i in negated function is: -10
x after the call to negated is: -10
```

Notice that changing the value of the reference parameter *i* inside *negative*, changes the value of *x*. This is because *negative* has implicitly received a pointer to *x*.

Inside the function *negative*, it is not possible to change what the reference parameter *i* is pointing to. A statement like:

```
i++:
```

inside *negative* increments the value of the variable *i*. It does not cause *i* to point to a new location.

Passing References To Objects

By default, when an object is passed to a function, a bitwise copy of that object is passed. When the function terminates, the copy's destructor is called. If we do not want the destructor function to be called when the function terminates, we can pass the function a pointer to the object or we can pass the object by reference. When we pass an object by reference, a copy of the object is not made, instead an implicit pointer to the object is passed to the function. This implies that the object passed by reference is not destroyed when the function terminates.

In the following example, an object *bar* is passed by value to the *negative* function in the object *foo*:

```
#include <iostream>
```

```
class Foo
{
public:
  int id;
  const char* name;
  Foo(int i, const char* nm)
  {
     id = i;
     name = nm;
     std::cout << "Foo constructor name = " <<  name << std::endl;
  }

  ~Foo() { std::cout << "destroying " << name  << std::endl; }
  void negative(Foo object) { object.id = -object.id; }
};


int main()
{
   Foo foo(1, "foo");
   Foo bar(100, "bar");
   foo.negative(bar);

   return 0;
}

// output
Foo constructor name = foo
Foo constructor name = bar
destroying bar
destroying bar
destroying foo
```

Notice that the copy of *bar* that was passed to *foo* is destroyed after the call to *negative* returns and then the object bar created on the stack of *main* is destroyed.

If we pass *bar* to *Foo::negative* by reference then:

```
#include <iostream>

class Foo
{
public:
  int id;
  const char* name;
  Foo(int i, const char* nm)
  {
     id = i;
     name = nm;
```

```
       std::cout << "Foo constructor name = " <<  name << std::endl;
    }

    ~Foo() { std::cout << "destroying " << name  << std::endl; }
    void negative(Foo& object) { object.id = -object.id; }
  };

  int main()
  {
     Foo foo(1, "foo");
     Foo bar(100, "bar");
     foo.negative(bar);

     return 0;
  }

  // output
  Foo constructor name = foo
  Foo constructor name = bar
  destroying bar
  destroying foo
```

Since *Foo::negative* receives a reference to the *bar* object, the bar object is not destroyed when *negative* returns.

Passing all but the smallest objects by reference is faster than passing them by value. Arguments are passed on the stack. Thus, large objects take a considerable number of CPU cycles to push onto and pop from the stack.

Returning References

A function may return a reference. This has the rather startling effect of allowing a function to be used on the left side of an assignment statement.

```
#include <iostream>

char s[80] = "Hello World";

// return a reference to a char
char &replace(char t,  int i)
{
   s[i] = t;
   std::cout << "string s in replace(): " << s << std::endl;
   return s[i];
}

int main()
{
  std::cout << "string s before call: " << s << std::endl;
  replace('X', 5) = 'Z';
  std::cout << "string s after call: " << s << std::endl;
}

# Output of program
string s before call: Hello World
string s in replace(): HelloXWorld
string s after call: HelloZWorld
```

When the *replace* function executes it replaces the fifth character of *s* with an ‘X’. The function then returns a reference (implicit pointer) to the fifth character in the string *s*. Next the reference value that is pointed to is changed to ‘Z’. The assignment **replace('X', 5) = 'Z'** assigns Z to the fifth character of *s*.

Independent References

We can declare a variable that is simply a reference variable. This type of variable is called an *independent reference*. For example:

```
int y = 9;
int& x = y;
```

The independent reference *x* is another name (*alias*) for the variable *y*. An independent reference must be initialized when it is declared because it must point to something when it is declared. A bare declaration such as *int& x;* will not compile. The following declaration will also not compile:

```
int& x = 5;
```

The reason is that 5 is an ephemeral value that does not have an addressable location (we will discuss this in length later when we look at rvalues). The right hand side memory location with a 5 value will not exist after the statement has finished executing.

The following shows examples of manipulating independent reference variables:

```
#include <iostream>

int main()
{
    int a = 10;
    // independent reference to a
    int &r = a;

    // a and r have the same values
    std::cout << "a: " << a << " ref: " << r << "\n";

    // changing the value of r changes a
    r = 100;
    std::cout << "a: " << a << " ref: " << r << "\n";

    // put variable b’s value in r
    int b = 19;
    r = b;
    std::cout << "b: " << b << " a: " << a << " r: " << r << "\n";

    // decrementing r also decrements b
    r--;
    std::cout << "b: " << b << " r: " << r << "\n";
}

// output
a: 10 r: 10
a: 100 r: 100
b: 19 a: 19 r: 19
a: 18 r: 18
```

Restrictions On References

A number of restrictions apply to references:

- null references are prohibited (*the implicit pointer cannot point to null*)
- we cannot reference a reference. Put differently, we cannot obtain the address of a reference.
- we cannot create a pointer to a reference

An independent reference variable must be initialized when it is declared.

Dangling References

A dangling reference refers to a reference variable that points to a value that does not exist or does not have an addressable memory location. In the example below, the function returns a reference that points to a local value on the stack:

```
#include <iostream>

int& foo() {
  int x = 5;
  return x;
}

int main()
{
  int& ret = foo();
  std::cout << ret << std::endl;
}

# error: reference to local variable x returned.
```

This program will not compile. The return value exists on the stack and will cease to exist once the function returns.

Returning A Const Reference From A Class Member Function

Consider the following program:

```
#include <string>
#include <iostream>

class Dog
{
  std::string name;
public:
  Dog(std::string nm) { name = nm; }
  const std::string& get_name() { return name; }
};
```

```
int main()
{
  class Dog d("rover");

  const std::string& dog_name = d.get_name();

  std::cout << dog_name << std::endl;
}

//output
rover
```

The *const* reference in the *Dog::get_name* declaration signifies that the value of the reference variable *dog_name* cannot be changed. Because of this restriction, a statement such as:

```
d.get_name() = “fido”
```

will not compile.

C++ Function Overloading

Function overloading is an example of compile-time polymorphism. Function overloading is the process of using the same function name for two or more functions. In function overloading, each redefinition of a function uses different types of function parameters or a different number of parameters. These differences enables the compiler to know which function to call in any given situation.

Here is an example:

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

// forward declarations
// these functions differ in the types of the parameters
int myfunc(int i);
double myfunc(double i);

int main()
{
  cout << myfunc(10) << " ";       // calls myfunc(int i)
  cout << myfunc(5.4);             // calls myfunc(double i)
  return 0;
}

double myfunc(double i)
{
  return i;
}

int myfunc(int i)
{
  return i;
}

// output
10 5.4
```

Two functions that differ only in their return types cannot be overloaded.

Sometimes, two function declarations will appear to be different, when in fact they are not. For example:

```
void f(int *p);
void f(int p[]);           // error, *p is same as p[]
```

A program containing these declarations will not compile. To the compiler *p is the same as p[].

Overloading Constructor Functions

There are three main reasons why we will want to overload a constructor: to gain flexibility, to allow both initialized and uninitialized objects to be created, and to define copy constructors.

Finding the Address of an Overloaded Function

A function name by itself is also the address of a function. One reason that we need function addresses is to assign an address to a function pointer and then call this function through the pointer.

If a function is not overloaded, obtaining a function address is straightforward. However, for overloaded functions, the process is more subtle. To understand why, first consider this statement, which assigns the address of a function called *myfunc()* to the pointer *p*:

```
p = myfunc;
```

If *myfunc()* is not overloaded, there is one and only one function called *myfunc*, and the compiler has no difficulty assigning its address to p. However, if *myfunc()* is overloaded, how does the compiler know which function address to assign to p? The answer is that it depends upon how p is declared.

Consider this program:

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

// forward declarations
int myfunc(int a);
int myfunc(int a, int b);

int main()
{
  // pointer to a function with declaration int f(int)
  int (*fp)(int);
  fp = myfunc;           // points to myfunc(int)

  cout << fp(5)<< endl;

// pointer to a function with declaration int f(int, int)
  int (*f)(int, int);
  f = myfunc;

  cout << f(5, 6)<< endl;

}

int myfunc(int a) {
  return a;
}

int myfunc(int a, int b) {
  return a*b;
}

// output
5
30
```

Here, there are two versions of *myfunc()*. Both return an *int*, but one takes a single integer argument and the other requires two integer arguments. In this program, *fp* is declared as a pointer to a function that takes one integer argument and returns an integer. When *fp* is assigned the address of *myfunc()*, C++ uses this information to select the *myfunc(int a)* version of *myfunc*. Had *fp* been declared like this:

```
int (*fp)(int, int);
```

Then *fp* would have been assigned the address of the *myfunc(int a, int b)* version of *myfunc()*.

Default Function Arguments

C++ allows a function parameter to be assigned a default value when an argument corresponding to that parameter is not specified in a call to the function. The following example declares *foo()* as taking one double argument with a default value of 0.0:

```
void foo(double d = 0.0)
{

}
```

Now, *foo()* can be called in any of two ways, as the following examples show:

```
foo(198.234);       // pass an explicit value
foo();              // use the default parameter value of foo
```

All parameters that take default values must appear to the right of those that do not. Once we begin the definition of parameters that take default values, we cannot specify a non-defaulting parameter. That is, a declaration like this will not compile:

```
int myfunc(float f, char *str, int i=10, int j);
```

It is permissible to use default parameters in an object's constructor.

Function Overloading And Implicit Casts

C++ automatically tries to convert the arguments used to call a function into the parameter types expected by the function.

```
int myfunc(double d);

// not an error, implicit conversion applied
cout << myfunc('c');
```

This is not an error because C++ automatically converts the character c into its double equivalent. Type conversions that are not narrowing are always implicitly applied by the compiler when required.

C++ Const And Mutable Member Functions

const Member Functions

Class member functions can be declared as *const* functions. A *const* function does not modify the data members of the class that it belongs to. A member function that is declared as a *const* function cannot call a non-const member function.

The syntax for a *const* member function is:

```
class Foo
{
  int some_var;
public:
  int func() const;            // const member function
};
```

The attribute keyword *const* is part of the function declaration.

The following example shows a program with *const* member functions that will not compile:

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

class Foo
{
  int i;
public:
  int geti() const {
          return i;      // OK
  }

   void seti(int x) const {
       i = x;                      // compiler error
   }
};
```

```
int main()
{
  Foo ob;
  ob.seti(1900);                  // compiler error
  cout << ob.geti() << endl;

  return 0;
}
```

This program will not compile because *seti()* is declared as *const*. This means that it is not allowed to modify the data member *i*. Since *seti()* attempts to change *i*, the program fails with a compiler error.

A *const* member function may not call a non-const member function, since the latter can mutate a class data member.

Overloading Functions With const Reference Parameters

A class can contain two member functions that have identical parameter types and only differ in that one of these functions has a const parameter:

```
void set_name(const std::string name);

void set_name(std::string name);
```

The compiler can distinguish between these two member functions on the basis of whether a const or non-const argument is received.

The following program shows this:

```
#include <string>
#include <iostream>

class Dog {
public:
   void set_name(std::string& nm)  {
     std::cout << "function called with " + nm << std::endl;
   }
```

```
    void set_name(const std::string& nm) {
      std::cout << "function called with const parameter: ";
      std::cout << nm << std::endl;
  }
};


int main() {
  const std::string name1("rover");
  std::string name2("fido");
  class Dog d;

  d.set_name(name1);
  d.set_name(name2);

  return 0;
}

// output
function called with const parameter: rover
function called with fido
```

Const Classes

Consider a class which is defined as a *const* class. A *const* class can only call *const* member functions. The syntax for a const class definition is:

```
const class Foo foo;
```

As the following example shows, *foo* must only call const functions in a const class. The called functions must be declared as a const functions. The compiler will throw an error if an attempt is made to call a non-const function:

```
#include <string>
#include <iostream>

class Dog
{
public:
   void print_name(std::string name) const { std::cout << "const function called: ";
```

```
     std::cout<<  name << std::endl;}

   void print_name(std::string name) { std::cout << "non-const function called: ";
     std::cout << name << std::endl; }
};

int main()
{
  class Foo foo;
  foo.print_name("fido");

  const class Foo g;
  g.print_name("rover");

  return 0;
}
```

```
Output
non-const function called: fido
const function called: rover
```

Mutable Data Members

Sometimes there are one or more data members of a class that we want a const function to be able to modify, even though we don't want the function to be able to modify any other data members. We can implement this by declaring these data members to be **mutable**. A mutable data member can be modified by a const member function.

```
#include <iostream>

class Foo
{
  mutable int i;
public:
  Foo() {i = 0;}

  int geti() const
  {
     return i; // OK
  }

  void seti(int x) const
  {
     i = x;               //  OK since i is mutable
  }
};
```

```
int main()
{
  Foo obj;
  obj.seti(5);
  std::cout << obj.geti() << std::endl;
}

// output
5
```

Why Using const Is Important

const correctness is a powerful technique for writing better and safer C++ code. The use of *const* can avoid a lot of trouble and debugging time because violations of const cause compile-time errors. In addition, using *const* enables the compiler to perform more optimizations.

Principle: Pay attention to const-correctness and use *const* whenever possible.

Volatile Variables

The `volatile` keyword informs the compiler that a variable may change without the program knowing about the change. Variables that are declared as `volatile` will not be cached by the compiler.

Volatile Member Functions

Class member functions can be declared as volatile. This causes the *this* pointer to be treated as a volatile pointer (the pointer can change without the program knowing about the change). The syntax for a *volatile* member function is:

```
class X {
public:
void f2(int a) volatile;        // declare volatile member function
};
```

C++ Copy Constructors

By default, C++ performs a bitwise copy during an object assignment such as *Foo = Bar*. That is, an identical copy of the object is made. Although this is perfectly adequate for many cases and exactly what we want to happen, there are situations in which a bitwise copy must not be used. One of the most common cases is when an object allocates memory when it is created. For example, assume we have a class called *MyClass* that allocates memory whenever an instance is created. Let A be an instance of this class and suppose that we have the following assignment:

```
MyClass B = A;
```

If a bitwise copy is performed, then B will be an exact copy of A. This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own memory. Clearly, this is dangerous. For example, if *MyClass* includes a destructor that frees allocated memory, then the same piece of memory will be freed twice when A and B are destroyed.

C++ solves this problem with copy constructors. A copy constructor initializes an object using another object.

The syntax for a copy constructor is:

```
ClassName (const ClassName& obj) {
    // body of constructor
}
```

obj is a reference to an object of type *ClassName.* It is permissible for a copy constructor to have additional parameters. However, in all cases, the first parameter must be a reference to the object that will be used to initialize the new object.

Note: If the ampersand & is omitted in the copy constructor, the constructor will do a bitwise copy.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization. Initialization can occur in any of the following three ways:

- when one object explicitly initializes another in a declaration.
- when a copy of an object is made during a function call.
- when an object receives a function return value

The copy constructor applies only to initializations. It does not apply to assignments. For example, assume that we have a class called *Foo*, and that *foo* is an object of this class, then each of the following statements involves initialization:

- Foo foo(x)   // constructor invoked with argument x
- Foo foo = 5;   // constructor has one scalar parameter
- func(foo)   // foo passed as a function argument
- foo = func();   // foo receives a temporary return object

The following example shows two ways to create an object that wraps an array:

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

class Foo
{
  int *p;
  int size;
public:
  // constructor
  Foo(int sz) {
  try {
      p = new int[sz];
  }
  catch (bad_alloc& err) {
     cout << "Allocation Failure\n";
     exit(1);
   }

    size = sz;
  }

  ~Foo() { delete [] p; }

  // copy constructor declaration
  Foo(Foo& a);

  void put(int i, int j) {
     if(i >= 0 && i < size) p[i] = j;
  }

  int get(int i) { return p[i]; }
  int get_size() { return size; }
};

// Copy Constructor
Foo::Foo(Foo& a)
{
   int i;
   int sz = a.get_size();
   try {
        p = new int[sz];
      }
    catch (bad_alloc& err) {
        cout << "Allocation Failure\n";
        exit(1);
    }

    for(i = 0; i < a.size; i++) p[i] = a.get(i);
 }

int main()
{
  Foo numarray(10);
  int i;
  for(i = 0; i < 10; i++) numarray.put(i, i);
  for(i = 9; i >= 0; i--) cout << numarray.get(i) << ", ";
  cout << endl;

  // copy constructor invoked
  Foo foo(numarray);
  for(i = 0; i < 10; i++) cout << foo.get(i) << ", ";
}
```

```
// output
9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

When the *foo* object is created by invoking the copy constructor, memory for the new array is allocated and the array in *numarray* is copied to *foo's* array. In this way, *foo* and *numarray* have arrays that contain the same values, but each array is separate and distinct. (That is, *numarray::p* and *foo::p* do not point to the same location in memory.)

If the copy constructor had not been used, the default bitwise initialization would have resulted in *foo* and *numarray* sharing the same memory for their arrays. (That is, *numarray::p* and *foo::p* would have pointed to the same location.)

The copy constructor is only called to initialize an object. For example, the following code does not call the copy constructor defined in the preceding program:

```
Foo foo;

Foo bar;

// does not invoke the copy constructor
// this is a copy assignment
foo = bar;
```

In this case, *foo = bar* performs an object assignment. Assignments do not invoke the copy constructor (unless the assignment operator is overloaded, see the next chapter).

C++ Operators

Operators are symbols that denote operations on variables called operands. Familiar operators include: +, /, -, *, %. These are the arithmetic operators. We also have relational operators such as <, >, =, <=, >= and !=. All of these operators are binary operators since they require two operands. We also have unary operators such as ++, !, -- and so on. Unary operators need only one operand.

Operators work with native types such as *int* and *float*, for example. In C++ the semantics of operators are extended to apply to user-defined data types (*classes*). This is accomplished by operator overloading.

Operator overloading is closely related to function overloading. In C++, we can overload most operators so that they perform special operations relative to the class in which they are defined. For example, a class that maintains a stack might overload the '+' operator to perform a push operation and '–' to perform a pop. Ideally, when an operator is overloaded, none of its original meaning is lost. Instead, the type of objects to which it applies is expanded.

The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, we can use objects in expressions in the same way that we use these operators with the C++ built-in (*native*) data types.

We overload operators by creating class member operator functions. A member operator function defines the operation that the overloaded operator will perform relative to the class that it is member off.

An operator function is created by using the keyword **operator**. Operator functions can be either member functions of a class or free functions. Typically, free operator functions are friend functions of a class.

Creating An Operator Member Function

A class member operator function has the following syntax:

```
return-type class-name::operator#(argument list)
{
      // program statements
}
```

The # is a placeholder for an operator such as +, -, *, /, ==, >= and so forth. When we create an operator function, we substitute such an operator for the #. For example, if we are overloading the division operator ‘/’, the operator function is *operator/* (or *operator /*). If we are overloading the ‘+’ operator, the operator function is *operator+* (or *operator* +).

Operator functions typically return an object of the class that they are a member off, but the return type can be anything.

When we are overloading a unary operator, the member operator function’s argument list will be empty. When we are overloading a binary operator, the argument list will contain one parameter (the other parameter is the class to which the operator belongs).

The following program declares a class called *Location*, which stores longitude and latitude values. This class overloads the addition operator +:

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

class Location
{
  int longitude, latitude;
public:
  // constructors
  Location() {}
  Location(int lg, int lt)
  {
    longitude = lg;
    latitude   = lt;
   }

   void show() {
     cout << longitude << ":" << latitude << endl;
   }

   // operator+ declaration
   Location operator+(Location);

};

// definition of the overloaded operator +
Location Location::operator+(Location loc2)
{
  Location temp;
  temp.longitude  = loc2.longitude + longitude;
  temp.latitude   = loc2.latitude + latitude;
  return temp;
}


int main()
{
  Location obj1(10, 20), obj2(5, 30);
  obj1.show();                    // displays 10:20
  obj2.show();                    // displays 5:30
  obj1 = obj1 + obj2;
  obj1.show();                    // displays 15:50

  return 0;
}

// output
10:20
5:30
15:50
```

Note that:

```
a + b = a.operator+(b)
```

It is common for an overloaded operator function to return an object of the class that it belongs to, for example:

```
obj3 = obj1 + obj2;
```

Having the *operator+()* return an object of type L*ocation* makes it possible to write the following statement:

```
(obj1+obj2).show();
```

It is important to understand that an operator function can return an object of any type and that the type returned depends solely upon our implementation.

The following example overloads the unary increment operator ++ for the *Location* class. This operator does not have any arguments:

```
#include <iostream>

class Location
{
 public:
  int longitude, latitude;
  // constructor
  Location(int lng, int lat)
  {
     longitude = lng;
     latitude = lat;
  }
  // operator++ declaration
  Location operator++();
};

// Overloaded ++ operator for Location
Location Location::operator++()
{
  longitude++;
  latitude++;
  return *this;
}

int main()
{
  Location foo(100, 49);

  //foo.operator++()
  ++foo;
  std::cout << foo.longitude << ":" << foo.latitude << std::endl;

}
```

```
// output
101:50
```

Overloading The Shorthand Operators

We can overload any of the C++ "shorthand" operators, such as +=, – =, and the like.

Operator Overloading Restrictions

The following restrictions apply to operator overloading:

- the precedence of an operator cannot be changed.
- we cannot change the number of operands that an operator takes.
- except for the function call operator (), operator functions cannot have default arguments.

Operator Inheritance

Except for the assignment operator =, operator functions can be inherited by derived classes. A derived class can overload an inherited operator.

Operator Overloading Using a Friend Function

We can overload an operator for a class by using a non-member friend function of the class. Since a friend function is not a member of the class, it does not have a *this* pointer. Due to this, an overloaded friend operator function has to be passed it’s operands explicitly. This means that a friend function that overloads a binary operator must have two parameters, and a friend function that overloads a unary operator must have one parameter.

Note that when we overload a binary operator using a friend function:

```
a + b is the same as operator+(a, b)
```

The following program defines a friend operator function for the addition + operator:

```
// Use a friend operator function
#include <iostream>

class Location
{
  int longitude, latitude;
public:
  // constructors
  Location() {}
  Location(int lg, int lt)
  {
    longitude = lg;
    latitude = lt;
  }

  void show() {
      std::cout << longitude << " " << latitude << "\n";
  }

    // friend operator+ declaration
    friend Location operator+(Location op1, Location op2);
 };

// operator+ is overloaded using a friend function.
Location operator+(Location op1, Location op2)
{
  Location temp;
  temp.longitude = op1.longitude + op2.longitude;
  temp.latitude = op1.latitude + op2.latitude;
  return temp;
}


int main()
{
  Location obj1(10, 20), obj2( 5, 30);
  obj1 = obj1 + obj2;
  obj1.show();
  return 0;
}

// output
15 50
```

Some restrictions apply to friend operator functions:

- the =, (), [], or –> operators cannot be overloaded by using a friend function.

- when overloading the increment or decrement operators with a friend function, we must use reference parameters.

C++ Class Inheritance

Inheritance enables the creation of hierarchical classifications. For example, we can create a general class that defines some basic traits. This class can then be inherited by other, more specific classes, each adding only those attributes that are unique to the inheriting class.

A class that is inherited is called a **base class**. A class that inherits from a base class is called a **derived class**.

Base Class Access Control

When a class inherits from another class, some of the members of the base class become members of the derived class. Class inheritance uses this syntax:

```
class derived-class-name : access_mode base-class-name {
       // body of derived class
};
```

The access mode keyword determines which attributes of the base class are inherited. The access mode must be either *public*, *private*, or *protected*. If no access mode is specified, it is private by default if the derived class is a class. If the derived class is a struct, then the default access mode is public.

When the access mode for a base class is public, all of the public members of the base class become public members of the derived class, and all the protected members of the base class become protected members of the derived class.

When a base class is inherited through private access mode, all of the public and protected members of the base class become private members of the derived class.

In both of these access cases, the private members of the base class remain private to the base class and are not inherited by the derived class.

Note: When a base class member function is inherited through public access, the function is not re-implemented in the derived class. The base class member function is accessed as if it is a member of the derived class. This is made possible because the member function implicitly receives the this pointer of the object it is a member of. Behind the scenes, the base class member function is invoked. This implementation promotes code reuse since there is only one actual implementation of a member function.

A derived class that inherits a public data member has it's own copy of this data member. Here is an example of public inheritance:

```
#include <iostream>

class Base
{
  int i, j;
public:
  void set(int a, int b) { i = a; j = b; }
  void show() { std::cout << i << " " << j << "\n"; }
};

class Derived : public Base
{
  int k;
public:
  derived(int x) { k=x; }
  void showk() { std::cout << k << "\n"; }
};

int main()
{
  Derived obj(3);
  obj.set(1, 2);           // uses member function of the base class
  obj.show();
  obj.showk();             // uses member function of derived class
  return 0;
}

// output
1 2
3
```

Understanding Class Inheritance

In C++, inheritance is implemented as follows:

- each object gets its own copy of inherited data members.
- all objects in a class hierarchy access the same function definitions.

The issue is that if only one copy of a member function exists and it is used by multiple objects, then how are the proper data members in an object accessed by such a member function? The compiler supplies an implicit *this pointer* that is passed to all non-static member functions. The *this pointer* points to the object. The *this* pointer is used to access and modify the data members of the object pointed to.

Inheritance And Protected Members

By using the protected attribute, we can create class members that are private but can still be inherited by a derived class.

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

class Base
{
  protected:
  // private to Base, but can be inherited by a derived class
  int i, j;
public:
  void set(int a, int b) { i = a; j = b; }
  void show() { cout << i << " " << j << "\n"; }
};

class Derived : public Base
```

```
{
  int k;
public:
    // derived class inherits i and j from the base class
    void setk() { k=i*j; }
    void showk() { cout << k << "\n"; }
 };


int main()
{
  Derived obj;
  obj.set(2, 3);             // OK, set is known to derived
  obj.show();                // OK, show is known to derived
  obj.setk();
  obj.showk();

  return 0;
}

// output
2 3
6
```

In this example, because *Base* is inherited by *Derived* as public and because *i* and *j* are declared as protected, the derived class inherits these variables and they are protected in the derived class. If *i* and *j* had been declared as private by *Base*, then *Derived* would not be able to inherit them, and the program would not compile.

When a derived class is used as a base class for another derived class, any protected members of the initial base class that are inherited (in public access mode) by the first derived class will also be inherited as protected again by a second derived class (providing that the second derived class inherits in public access mode).

When a base class is inherited as private by a derived class, the base class public and protected elements are inherited as private members. These members cannot be inherited by any class derived further from the first derived class.

Protected Base-Class Inheritance

We can inherit a base class in protected access mode. When this is done, all of the public and protected members of the base class become protected members of the derived class.

```
#include <iostream>

class Base {
protected:
  // private to base, but inheritable by Derived
  int i, j;
public:
  void setij(int a, int b) { i = a; j = b; }
  void showij() { std::cout << i << " " << j << "\n"; }
};

// Inherit Base as protected.
class Derived : protected Base
{
  int k;
public:
   void setk() { setij(10, 12); k = i*j; }
   void showall() { std::cout << k << " "; showij(); }
};

int main()
{
  Derived obj;

  obj.setk();          // OK, setij is a protected member of derived
  obj.showall();       // OK, public member of derived

  // obj.showij();

  return 0;
}

// output
120 10 12
```

Pointers To Derived Classes

In general, a pointer to one type of object cannot point to an object of a different type. However, there is an important exception to this rule that only applies to derived classes. To begin, assume that we have two classes called *Foo* and *Bar*. Furthermore, assume that *Bar* is derived from the base class *Foo*. In this situation, a pointer of type *Foo* can also point to an object of type *Bar*. In general, a base class pointer can also be used to point to an object of any class derived from the base class.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type *Bar* cannot be used to point to an object of type *Foo* (unless a *cast* is performed).

Although we can use a base class pointer to point to a derived object, we can only access the members of the derived object that are present in the base class. We cannot access any members that are added by the derived class. However, we can cast a base class pointer to a derived class pointer and gain full access to the entire derived class.

Here is an example:

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

class Base
{
  int i;
public:
  void set_i(int num) { i = num; }
  int get_i() { return i; }
};

class Derived: public Base
{
  int j;
public:
  void set_j(int num) { j = num; }
  int get_j() { return j; }
 };


int main()
{
  Base *bp;
  Derived d;

  // base class pointer points to the derived object
  bp = &d;
  // access the derived object using the base class pointer
  bp->set_i(10);
  cout << bp->get_i() << endl;

 /* The following won't work. We can't access a member of
    the derived class that does not exist in the base class */

 bp->set_j(88);                    // error
cout << bp->get_j() << endl;       // error

return 0;
}

// output
 error: 'class Base' has no member named 'set_j'
```

Although we must be careful, it is possible to cast a base class pointer to a pointer to a derived class so that we can access members of the derived class through the base class pointer. For example, this is valid C++ code:

```
/ access now allowed because of cast
((derived *)bp)->set_j(88);

cout << ((derived *)bp)->get_j() << endl;
```

It is important to remember that pointer arithmetic for a base class pointer is relative to the base class type of the pointer. For this reason, when a base class pointer is pointing to a derived object, incrementing the pointer does not cause it to point to the next derived class object. Instead, it will point to the next base class type object.

References To Derived Types

Similar to base class pointer semantics, a base class reference can be used to refer to an object of a derived class. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from the base class.

Inheriting From Multiple Base Classes

It is possible for a derived class to inherit from two or more base classes:

```
class derived: public base1, public base2
{
  int x,y;
public:
  void set(int i, int j) { x = i; y = j; }
};
```

Be sure to use an access-specifier for each base class that is inherited.

Constructors, Destructors And Inheritance

The following example shows when the base class and derived class constructor and destructor functions are called when the base class is inherited.

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

class Base
{
public:
  Base()  { cout << "Calling base class constructor\n"; }
  ~Base() { cout << "Calling base class destructor\n"; }
};

class Derived: public Base
{
public:
  Derived()  { cout << "Calling derived class constructor\n"; }
  ~Derived() { cout << "Calling derived class destructor\n"; }
};

int main()
{
  Derived foo;
  // do nothing but construct and destroy foo
   return 0;
}
```

This program displays:

```
Calling base object constructor
Calling derived class constructor
Calling derived object destructor
Calling base object destructor
```

When an object of a derived class is created, if the base class contains a constructor, it will be called first, followed by the constructor for the derived class.

When a derived class object is destroyed, it's destructor is called first, followed by the destructor for the base class, if it exists. Put differently, constructor functions are executed in their order of derivation. Destructor functions are executed in reverse order of derivation.

In the case of multiple inheritance, constructors are called in order of derivation, left to right, as specified in the derived class inheritance list. Destructors are called in reverse order, right to left.

Passing Parameters to Base-Class Constructors

The following example shows how parameters are passed to base class constructors:

```
Derived(arg-list) : Base1(arg-list), Base2(arg-list), ... BaseN(arg-list)
{
// body of the derived constructor
}
```

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class and also the names of the base class constructors. A colon separates the declaration of the constructor of the derived class from the base-class constructors. If the derived class inherits from multiple base classes, then the base-class constructors are separated from each other by commas.

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

class Base
{
protected:
  int i;
public:
  Base(int x) { i=x; cout << "base constructor called\n"; }
  ~Base() { cout << "base destructor called\n"; }
};

class Derived: public Base
{
  int j, p;

public:
  // argument y is passed to the base class constructor
  Derived(int x, int y): Base(y) {
      j=x;
      cout << "derived constructor called\n";
      p=y;
 }


  ~derived() { cout << "derived destructor called\n"; }
  void show() { cout << i << " " << j << " " << p << "\n"; }
};

int main()
{
  Derived obj(3, 4);
  obj.show();
}

// output
base constructor called
derived constructor called
4 3 4
derived destructor called
base destructor called
```

Here, the constructor of the derived class is declared as taking two parameters, *x* and *y*. *derived()* uses the *x* and *y* arguments; furthermore the *y* argument is passed to the base class constructor. The constructor of a derived class must declare all the parameters that it requires as well as any parameters that are required by it's base classes.

Even if a derived class constructor does not use any arguments, it will still need to declare parameters if the base class constructor has parameters in it's definition.

Virtual Base Classes

Ambiguity can be introduced into a C++ program when a base class is inherited more than once. The following program shows the **diamond inheritance problem**. This program will not compile:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;

class Base {
public:
int i;
};

// Derived1 inherits from base
class Derived1 : public Base {
public:
  int j;
};

// Derived2 inherits from base
class Derived2 : public base {
public:
  int k;
};

// Derived3 inherits from both Derived1 and Derived2

class Derived3 : public Derived1, public Derived2 {
public:
  int sum;
};

int main()
{
  Derived3 obj;
  // i is ambiguous since there are two inherited i data
  // members in derived3: the i inherited from derived1
  // and the i inherited from derived2
  obj.i = 10;
  obj.j = 20;
  obj.k = 30;

  // i is ambiguous here, too
  obj.sum = obj.i + obj.j + ob.k;

  // also ambiguous, which i?
  cout << obj.i << " ";
  cout << obj.j << " " << obj.k << " ";
  cout << obj.sum << endl;
  return 0;
}

// output
error: request for member 'i' is ambiguous
```

As this program indicates, both *Derived1* and *Derived2* inherit the data member *i* from the base class. However, *Derived3* inherits from both *Derived1* and *Derived2*. This means that the base class member *i* is present twice in *Derived3*. Thus, in an expression such as:

```
obj.i = 10;
```

we do not know which i is being referred to.

There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to i and manually select one i:

```
int main()
{
  Derived3 obj;
  obj.derived1::i = 10;      // scope resolved, use Derived1's i
```

A better solution is to prevent *Derived3* from including two copies of the base class data member *i*.

When two or more classes are derived from a common base class, we can prevent multiple copies of base class data members from being present in a class derived from these classes by declaring the base class as virtual when it is inherited. We accomplish this by preceding the base class name with the keyword *virtual* when it is inherited:

```
 // This program uses a virtual base class to solve
 // the diamond inheritance problem
 #include <iostream>
 using namespace std;

class Base {
 public:
   int i;
  };

 // Derived1 inherits from a virtual base class
 class Derived1 : virtual public Base {
 public:
   int j;
 };

 // Derived2 also inherits base as a virtual base class
 class Derived2 : virtual public Base {
 public:
   int k;
 };

 /* Derived3 inherits from both Derived1 and Derived2.
 This time, there is only one copy of member i of the base class. */

 class Derived3 : public Derived1, public Derived2 {
 public:
   int sum;
 };

 int main()
 {
   derived3 obj;
   obj.i = 10; // now unambiguous
   obj.j = 20;
   obj.k = 30;

   // unambiguous
   obj.sum = obj.i + obj.j + obj.k;
   // unambiguous
   cout << obj.i << " ";
   cout << obj.j << " " << obj.k << " ";
   cout << obj.sum << endl;
   return 0;
 }

 // output
 10 20 30 60
```

Now that both *Derived1* and *Derived2* inherit from a virtual base class, inheritance of *Derived3* from *Derived1* and *Derived2* will cause only one copy of the base class data members to be inherited by *Derived3*.

The only difference between a normal base class and a virtual one is what happens when an object inherits the base class more than once. If virtual base classes are used, then base class data members will be present only once in any derived object.

C++ Virtual Functions And Polymorphism

C++ implements compile-time and run-time polymorphism. Compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions.

Virtual Functions

A virtual function is a member function that is declared in a base class and (possibly) redefined in a derived class. By redefining the function, we mean changing the body of the function but not the function signature. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class can redefine the virtual function to fit its own needs.

In essence, virtual functions implement the "*one interface, multiple methods*" philosophy that underlies polymorphism. The virtual function in the base class is part of the interface to the class. Each redefinition of the virtual function in a derived class re-implements the interface so that it specifically relates to the derived class.

When accessed "normally," virtual functions behave just like any other class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when they are accessed through a pointer. A base class pointer can be used to point to an object of a class that is derived from the base class. When a base class pointer points to a derived object that inherits a virtual function, C++ determines which version of that function to call based upon the type of object pointed to by the pointer. This determination is made at run-time. Thus, when different objects are pointed to, different versions of the virtual function are executed.

The following example, demonstrates this:

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

class Base
{
public:
  virtual void vfunc() {
     cout << "This is the base class vfunc() \n";
  }
};

class Derived1 : public Base
{
public:
  void vfunc() {
     cout << "This is the derived1 class vfunc()\n";
  }
};

class Derived2 : public Base
{
public:
  void vfunc() {
      cout << "This is the derived2 class vfunc()\n";
  }
};

void print(base* bPtr)
{
   bPtr->vfunc();

}


int main()
{
  Base *p, b;
  Derived1 d1;
  Derived2 d2;

  // point to the Base class instance
  p = &b;
  print(p);               // access the base instance vfunc()

   // point to Derived1
   p = &d1;
   print(p);              // access the derived1 instance vfunc()
  // point to Derived2
   p = &d2;
   print(p);              // access the derived2 instance vfunc()

   return 0;
}

// output
This is the base class vfunc()
This is the derived1 class vfunc()
This is the derived2 class vfunc()
```

Notice that the keyword *virtual* in the base class precedes the rest of the function declaration. The keyword *virtual* is not needed when *vfunc()* is redefined in *Derived1* and *Derived2*.

The kind of object to which the base class pointer *p* points, determines which version of *vfunc* that is executed. This is run-time polymorphism.

A virtual function must be a non-static member of a class. Additionally, virtual functions cannot be friend functions. Finally, constructors cannot be virtual, but destructors can.

Calling a Virtual Function Through a Base Class Reference

In the preceding example, a virtual function was called through a base class pointer. The polymorphic properties of a virtual function are also available when the function is called through a base class reference. Since a reference is an implicit pointer, a base class reference can be used to refer to an object of the base class or any object derived from the base class. When a virtual function is called through a base class reference, the version of the function that is executed is determined by the object that is referred to at the time of the function call.

```
// A base class reference is used to access a virtual function
#include <iostream>
using namespace std;

class Base
{
```

```
public:
  virtual void vfunc() {
     cout << "This is the base vfunc() \n";
  }
};

class Derived1 : public Base
{
public:
  void vfunc() {
      cout << "This is the derived1 vfunc() \n";
  }
};

class Derived2 : public Base
 {
 public:
   void vfunc() {
       cout << "This is the derived2 vfunc() \n";
   }
};

// invoke a virtual function through a base
// class reference parameter
void f(Base& r) {
     r.vfunc();
}

int main()
{
  Base b;
  Derived1 d1;
  Derived2 d2;
  f(b);                // pass a base class reference to f()
  f(d1);               // pass a derived1 class reference to f()
  f(d2);               // pass a derived2 class reference to f()
  return 0;
}

// output
This is the base vfunc()
This is the derived1 vfunc()
This is the derived2 vfunc()
```

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual attribute is also inherited. This means that when a derived class that has inherited a virtual function, is itself used as a base class for another derived class, then the virtual function is also inherited and can be overridden. Put differently, no matter how many times a virtual function is inherited, it remains virtual.

Virtual Functions Are Hierarchical

When a function is declared as virtual in a base class, it may be overridden in a derived class. However, the function does not have to be overridden. When a derived class does not override a virtual function, then when an instance of the derived class accesses the virtual function, the function defined in the base class will be used.

In general, when a derived class does not override a virtual function, the first redefinition found in the reverse order of derivation is used.

Pure Virtual Functions

When a virtual function is not redefined in a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function in a base class. For example, a base class may not be able to define an object sufficiently to allow a base class virtual function to be used. Furthermore, in some situations we will want to ensure that all derived classes override a virtual function in the base class. To handle these two cases, C++ supports pure virtual functions.

A pure virtual function is a virtual function that is declared but not defined in the base class. We use the following syntax to declare a pure virtual function:

```
virtual return-type func-name(parameter list) = 0;
```

When a virtual function is pure, a derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

Abstract Classes

A class that contains at least one pure virtual function is said to be an abstract class. Because an abstract class contains one or more functions for which there is no definition (that is, pure virtual functions), no objects of an abstract class can be instantiated.

Although we cannot create objects of an abstract class, we can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base class pointers and references to select the proper virtual function.

Using Virtual Functions

One of the fundamental principles of object-oriented programming is the principle of "**one interface, multiple methods**." This means that we can define an abstract class with pure virtual functions. This is the interface. Each derived class defines its own specific operations by implementing the interface.

One of the most powerful and flexible ways to implement the "*one interface, multiple methods*" technique is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, we can create a class hierarchy that moves from general to specific (base to derived). We define all the common features and the interface in the base class. In cases where certain actions can be only be implemented by a derived class, we use pure virtual functions in the base class. We declare everything that relates to the general case in a base class. The derived classes fills in the specific details.

Implementing Class Interfaces In C++

An interface is a collection of virtual member functions in a base class. A derived class must provide implementations of all the pure virtual member functions in it’s base class.

Interfaces enable loose coupling between classes.

The following base class demonstrates an implementation of an interface:

```
class Switch
{
public:
   virtual void on() = 0;
   virtual void off() = 0;
}
```

Early And Late Binding

Early binding occurs when all of the information needed to call a function is known at compile-time. (Put differently, early binding means that a function call is bound during compilation). Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and calls to overloaded operators. The main advantage of early binding is efficiency. These types of function calls are very fast because all of the information necessary to call a function is known at compile time.

The opposite of early binding is **late binding**. Late binding refers to function calls that are not resolved until run-time Virtual functions are used to achieve run-time binding. As we know, when an object is accessed through a base class pointer or reference, the virtual function actually called is determined by the type of object pointed to by the base class pointer at run-time. Because in most cases this cannot be determined at compile-time, the object and the function are not linked until run-time. The main advantage of late binding is flexibility.

Calling Virtual Functions In Constructors And Destructors

A constructor should be as simple as possible. Calling a virtual function from a constructor will produce unknown behavior. Take a look at the following example:

```
class Transaction {
public:
  Transaction() {
     logTransaction();
  }
  virtual void logTransaction() {  };
};

class BuyTransaction : public Transaction {
public:
 void logTransaction() {  };

};

BuyTransaction b;
```

The problem with this code is that when b is created, the base class *Transaction* will be created first and it will call the virtual *logTransaction* function in the base class since the derived class *BuyTransaction* does not yet exist. The *logTransaction* function in the derived class will never be called even though we are creating a *BuyTransaction* object.

Similarly, we should never call virtual functions in destructors. Since class objects are destroyed in the reverse order of derivation, virtual functions in destructors may call functions in derived objects that have already been destroyed.

Virtual Destructors

Base classes must declare their destructors as virtual. This enables derived objects to be deleted through a pointer to the base class. Look at the following program:

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

class Foo
{
public:
  ~Foo() { std::cout << "Foo destructor called" << std::endl;}
};

class Bar : public Foo
```

```
{
public:
  ~Bar() { std::cout << "Bar destructor called" << std::endl;}
};

class FooFactory
{
public:
  Bar* makeBar() { return new Bar; }
};

int main()
{
  FooFactory factory;
  Foo* pBar = factory.makeBar();
  delete(pBar);
}

// output
Foo destructor called
```

This example shows that only the base class portion of the *Bar* object has been destroyed. In order to destroy the *Bar* object, we must declare the destructor in the Base class *Foo* as virtual.

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

class Foo
{
public:
  virtual ~Foo() { std::cout << "Foo destructor called" << std::endl;}
};

class Bar : public Foo
{
public:
  ~Bar() { std::cout << "Bar destructor called" << std::endl; }
};

class FooFactory
{
public:
  Foo* makeBar() { return new(Bar); }
};

int main()
{
  FooFactory factory;
  Foo* pBar = factory.makeBar();
  delete(pBar);
}

// output
Bar destructor called
Foo destructor called
```

C++: Static Class Members

Static Class Members

A static class member is a member function or data member that is shared by all of the instances of a class. A static class member exists even if there are no instances of the class.

Here is an example of initializing static class members:

```
#include <iostream>

class Circle
{
public:
  double r;                        // not static
  // static class members
  static double pi;
  static double multiply(double);
};


// static data member initialization
double Circle::pi = 3.14159;

double Circle::multiply(double x) { return x*x; }


int main()
{
   std::cout << Circle::pi << std::endl;
   std::cout << Circle::multiply(Circle::pi) << std::endl;
}

output:
3.14159
9.86959
```

Since a static class member is not a member of any class object, it is accessed through the scope resolution operator. For example:

```
float pif = Circle::pi
and:
Circle::multiply(Circle::pi)
```

Note: A static member function cannot call a non-static member function.

Static Data Members

The declaration of a static data member in a class declaration is not a definition. The static data member must be initialized (defined) outside of the class declaration and at namespace scope.

```
class Square
{
public:
  static float length;
};

Square::length = 5;
```

However, if the static data member is const, it can initialized in the class declaration:

```
class Circle
{
public:
  static const float pi = 3.14;
};
```

If a static data member is declared as private then it will be inaccessible outside the class. There are two ways to initialize this variable. Firstly, we can declare the data member as const and initialize it inside the class declaration. Secondly, we can specify a public static member function that initializes the private data member. A static member function has access to all of the public and private data members of the class.

C++ Templates: Generic Functions

Templates are an advanced feature of C++. Templates permit us to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a generic parameter. Thus, we can use one function or class with several different types of data without having to explicitly re-code specific versions of the function or class for each data type.

Generic Functions

A generic function defines a set of operations or an algorithm that will be applied to data of various types. A generic function permits a single general algorithm to be applied to many data types.

Many algorithms are logically the same, no matter what type of data is being operated upon. For example, the bubble sort algorithm is the same whether it is applied to an array of integers or an array of doubles. Only the type of the data being sorted is different. By creating a generic function, we can define the algorithm, independent of the type of data. Once this is done, the compiler can automatically generate the correct function code for the type of data that will be actually used.

In essence, when we create a generic function we are creating a function that can automatically overload itself at compile-time.

A generic function is created using the *template* keyword. The general form of a function template declaration is:

```
template <class Ttype> return-type function_name(parameter list)
{
        // body of function
    }
```

Ttype is a symbolic name or placeholder for the data type used in the function (*read Ttype as template type*). *Ttype* is a generic type. This symbolic name may be used within the function's parameter list and body as well as the return type. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function.

Notice that a template function declaration creates a family of functions parameterized on *Ttype.*

Although the use of the keyword *class* to specify a generic type in a template declaration is traditional, we can also use the keyword **typename**.

Here is an example of a function template:

```
// function template example: swap two values
#include <iostream>
using namespace std;

// This is a function template declaration
template <class Ttype> void swapargs(Ttype &a, Ttype &b)
{
  Ttype temp;
  temp = a;
  a = b;
  b = temp;
}

int main()
{
   int i = 10, j = 20;
   double x = 10.1, y = 23.3;
   char a = 'x', b = 'z';

   cout << "Original i, j: " << i << ' ' << j   << '\n';
   cout << "Original x, y: " << x << ' ' << y << '\n';
   cout << "Original a, b: " << a << ' ' << b << '\n';

   swapargs(i, j);                    // swap integers
   swapargs(x, y);                    // swap floats
   swapargs(a, b);                    // swap chars

   cout << "Swapped i, j: "   << i << ' ' << j   << '\n';
   cout << "Swapped x, y: " << x << ' ' << y  << '\n';
   cout << "Swapped a, b: " << a << ' ' << b  << '\n';

    return 0;
}

// output
Original i, j:   10 20
Original x, y: 10.1 23.3
Original a, b: x z
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

Consider the generic function declaration again:

```
template <class Ttype> void swapargs(Ttype &a, Ttype &b)
```

Ttype is a generic type that is used as a placeholder. After the template expression, the function *swapargs* is declared, using *Ttype* as the data type of the values that will be swapped.

In *main*, the *swapargs* function is called using three different data types: *int*, *double*, and *char*. Because *swapargs* is a generic function, the compiler automatically creates three versions of s*wapargs*: one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters. These functions are created at compile-time.

When the compiler creates a specific version of a function template, it is said to have created a **specialization**. Put differently, a specialization is a specific instance of a function template.

When we create a function template, we are, in essence, allowing the compiler to generate different versions of the function for handling the different types that are used by the function.

Since C++ does not recognize the end-of-line as a statement terminator, the template clause of a generic function definition does not have to be on the same line as the function declaration. Thus, for readability, we can specify the *swapargs* function template as follows:

```
template <class Ttype>
void swapargs(Ttype &a, Ttype &b)
{
     Ttype temp;
     temp = a;
     a = b;
     b = temp;
}
```

A Function With Two Or More Generic Types

We can specify more than one generic data type in a function template declaration by using a comma-separated list of template types.

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

template <class Ttype1, class Ttype2>
void foo(Ttype1 x, Ttype2 y)
{
  cout << x << ' ' << y << '\n';
}

int main()
{

   // call foo with different argument types
      foo(10, "I like C++");
      foo(98.6, 19L);

   return 0;
}

// output
10 I like C++
98.6 19
```

In this example, the symbolic types *Ttype1* and *Ttype2* are replaced by the compiler with the data types *int* and *char **, and *double* and *long* respectively, when the compiler generates specific instances of *foo* at compile-time.

Explicitly Overloading a Generic Function

Even though a generic function automatically overloads itself as needed, we can also explicitly overload a generic function. This is formally called **explicit specialization**. If we explicitly overload a generic function, the explicitly overloaded function overrides the generic function and it will be used instead of the generic version.

```
// Explicitly Overriding a function template
#include <iostream>
using namespace std;

template <class Ttype>
void swapargs(Ttype &a, Ttype &b)
{
     Ttype temp;
     temp = a;
     a = b;
     b = temp;
     cout << "Inside template swapargs.\n";
}

// This overrides the generic version of swapargs for int types
void swapargs(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
   cout << "Inside swapargs for int specialization.\n";
}

int main()
{
  int i = 10, j = 20;
  double x = 10.1, y = 23.3;
  char a = 'x', b = 'z';

  cout << "Original i, j: " << i << ' ' << j << '\n';
  cout << "Original x, y: " << x << ' ' << y << '\n';
  cout << "Original a, b: " << a << ' ' << b << '\n';

  // calls the explicitly overloaded swapargs()
  swapargs(i, j);
  // calls the generic swapargs()
  swapargs(x, y);
  // calls the generic swapargs()
  swapargs(a, b);

  cout << "Swapped i, j: " << i << ' ' <<  j  << '\n';
  cout << "Swapped x, y: " << x << ' ' <<  y  << '\n';
  cout << "Swapped a, b: " << a << ' ' <<  b  << '\n';

  return 0;
}

// output
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs for int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j:   20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

In the previous example, because of the explicit function definition *void swapargs(int &a, int &b)*, the compiler knows that it must override the generic function with the explicitly specialized version when *swapargs* has two *int* arguments.

We can signal explicit specialization by declaring an explicit *swapargs* function in any of the three following ways:

```
template<> void swapargs<int, int>(int &a, int &b) { }
or

void swapargs<int, int>(int &a, int &b) { }
or
void swapargs(int&, int&) {...}            // ordinary function
```

The *template<>* declaration indicates to the compiler that an explicit declaration is being made.

Overloading Function Templates

We can overload a function template with respect to the number of function parameters:

```
// Overload a function template declaration
#include <iostream>
using namespace std;

// The first overloaded version of the func() template.
template <class Ttype>
void func(Ttype a)
{
    cout << "Inside func(Ttype a)\n";
}

// The second overloaded version of the func() template.
template <class Ttype1, class Ttype2>
void func(Ttype1 a, Ttype2 b)
{
    cout << "Inside func(Ttype1 a, Ttype2 b)\n";
}

int main()
{
  func(10);            // calls func(X)
  func(10, 20);        // calls func(X, Y)
  return 0;
}

// output
Inside func(Ttype a)
Inside func(Ttype1 a, Ttype2 b)
```

Using Ordinary Parameters With Function Templates

We can mix ordinary parameters with generic parameters in a function template:

```
// Using ordinary parameters in a function template
#include <iostream>
using namespace std;

template<class Ttype>
void print(Ttype data, int val)
```

```
{
    if (val > 10) {
      cout << data << "\n";
     }
     else {
      cout << "invalid value" << "\n";
     }
}

int main()
{
   print("This is a test", 11);

   return 0;
}


// output
This is a test
```

C++ Templates: Generic Classes

A class template is a family of classes where the actual type of the data that is manipulated in the class is specified as a generic parameter.

Class templates are useful when classes use logic that can be generalized. For example, the algorithm that maintains a queue of integers will also work for a queue of characters.

When we create a class template, it can perform the operation we define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically create the correct type of object at compile-time, based upon the type we specify when the object is instantiated.

The general form of a class template declaration is:

```
template <class Ttype>
class class-name {
.
.
.
};
```

This is a declaration and not a definition. Since the parameter *Ttype* is a generic type, this declaration declares a family of classes parameterized on *Ttype*. A definition instantiates an object using a particular template type.

A specific class is instantiated when a particular *Ttype* (*template type*) is specified. If necessary, we can specify more than one generic data type in a template declaration by using a comma-separated list of template types.

Once we have defined a generic class (template class), we can create a specific instance of this class using the following syntax:

```
class-name<type> obj;
```

Here, *type* is the type of the data that the class will be operating upon. The **class-name**<**type**> declaration determines the type of object that will be instantiated.

Member functions of a generic class are themselves automatically generic. We do not need to use function templates to explicitly specify them as such in a class declaration.

In the following program, a stack template class is declared:

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

const int SIZE = 10;

// Declare a stack template class
template <class Ttype>
class Stack
{
  Ttype my_stack[SIZE];          // the stack
  int num_el;                    // number of elements in the stack
public:
  Stack() { num_el = 0; }            // constructor
  void push(Ttype obj);              // push object on stack
  Ttype pop();                       // pop object off stack
};

// Push function
template <class Ttype>
void Stack<Ttype>::push(Ttype obj)
{
  if (num_el == SIZE) {
     cout << "Stack is full.\n";
     return;
  }
  my_stack[num_el] = obj;
  num_el++;
}

// Pop function
template <class Ttype>
Ttype Stack<Ttype>::pop()
```

```
{
  if (num_el == 0) {
     cout << "Stack is empty.\n";
     return '1';
  }
  num_el--;
  return my_stack[num_el];
}

int main()
{
  // Demonstrate with two character stacks.
  Stack<char> s1, s2;          // create two character stacks

  int i;
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  // demonstrate two double type stacks
  Stack<double> ds1, ds2;            // create two double stacks

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for (i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";

  for (i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

  return 0;
}

// output
Pop s1: c
Pop s1: b
Pop s1: a
Pop s2: z
Pop s2: y
Pop s2: x
Pop ds1: 5.5
Pop ds1: 3.3
Pop ds1: 1.1
Pop ds2: 6.6
Pop ds2: 4.4
Pop ds2: 2.2
```

Pay special attention to the stack declarations:

```
Stack<char> s1, s2;              // create two character stacks
Stack<double> ds1, ds2;          // create two double stacks
```

Also notice the class function declarations:

```
template <class Ttype>
Ttype Stack<Ttype>::push(Ttype ob)
{

}
```

The template function definition must start with *template<class Ttype>*, this is followed by the return type of the function, and then the scoped function *push*.

An Example With Multiple Generic Data Types

The following template declaration specifies a template class with two generic data types:

```
template <class Ttype1, class Ttype2>
class Foo
{
  Ttype1 i;
  Ttype2 j;
 public:
   Foo(Ttype1 a, Ttype2 b) { i = a; j = b; }
   void show() { cout << i << ' ' << j << '\n'; }
};
```

Using Non-Generic Arguments with class templates

A class template can have non-template parameters. Here is an example:

```
// Non-template arguments in a class template
#include <iostream>
using namespace std;

// Here, size is an int, it is a not a template type parameter
// since it’s type is specified in the template

template <class Ttype, int size>
class Arry
{
  Ttype a[20];
public:
  Arry(int sz)
  {
    int i;
    for (i = 0; i < sz; i++) a[i] = i + size;
  }

  Ttype get(int i) { return a[i]; }

};

int main()
{
  Arry<int, 10> ar1(2);              // integer array declaration
  Arry<double, 15> ar2(3);           // double array declaration

  for (int i = 0; i < 2; i++)
    std::cout << ar1.get(i) << " ";
  std::cout << std::endl;


   for (int i = 0; i < 3; i++)
     std::cout << ar2.get(i) << " ";
   std::cout << std::endl;
}

// output
10 11
15 16 17
```

Look carefully at the template declaration for *Arry*. Note that size is declared as an *int* and not as a generic type.

Allowed Non-generic Template Parameters

Non-generic parameters in templates are restricted to integers, pointers, or references. Other types, such as double and float, are not allowed. This restriction has been loosened in Modern C++.

Using Default Arguments With Class Templates

A generic type in a class template can specify a default type. For example:

```
template <class Ttype=int>
Foo;

Foo<>  foo;    // the default type will be used
```

Default values are specified using the = syntax.

It is also permissible for non-generic arguments to have default values in a template class declaration. The default value of the non-generic parameter will be used when an explicit value is not specified for the non-generic parameter when a particular template class is instantiated.

```
template <class tType=int, int size=10>
class Foo.

Foo<>  foo;
```

Here, *tType* defaults to type int, and size defaults to 10. Notice how the class *Foo* is declared with <>, this means that *Foo* uses the default template parameters.

Explicit Class Specializations

When a particular class object is instantiated from a class template (using specific type parameters and non-parameter values) we say that we have a specialization of the template class.

As with function templates, we can create explicit specializations for class templates. An explicit specialization will take precedence over a specialization produced by a template. To create an explicit specialization, we use the *template*<> syntax, which works in the same manner as it does for explicit function specializations.

```
// Demonstrate an explicit class template specialization
#include <iostream>
using namespace std;

template <class Ttype=int>
class Bar
{
  Ttype x;
public:
  Bar(Ttype a)
  {
     cout << "Inside Bar class\n";
     x = a;
  }

  Ttype getx() { return x; }
};

// Explicit specialization for the int type template
// This will be used rather than the Bar<int> produced
// by the template

template<> class Bar<int>
{
  int x;
public:
  Bar(int a)
  {
     cout << "Inside the Bar<int> specialization\n";
     x = a * a;
  }

  int getx() { return x; }
};

int main()
{
   Bar<> bar(109);
   cout << bar.getx() << endl;
}
```

```
// output
Inside the Bar<int> specialization
11881
```

The explicit specialization is made by using the declaration *Bar<> bar(109)* instead of *Bar<int> bar(109)*

The typename Keyword

The *typename* keyword can be used instead of the keyword *class* in a template definition. For example, the *swapargs* function template could be specified like this:

```
template <typename X>
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
};
```

C++ Exception Handling

An exception is a catastrophic error which will abort program execution unless it is handled. Exception handling allows us to manage serious run-time errors in a deterministic manner. With exception handling, our program can automatically invoke an error-handling routine when an exception occurs. The main advantage of exception handling is that it automates much of the error-handling code that previously had to be coded "by hand" in any large C++ program.

C++ exception handling is built upon three keywords: *try*, *catch*, and *throw*. Program statements that you want to monitor for exceptions are contained in a try block. If an exception occurs within a try block, the system throws the error. The exception can then be caught in a *catch* block and processed.

Code that you want to monitor for exceptions must execute from within a try block. Functions called within a try block may also throw exceptions. Exceptions thrown by monitored code can be caught in an optional catch block, which immediately follows the try block in which the exception was thrown. The syntax for try and catch blocks is:

```
try {
  // try block
  statements
}

// catch handlers
catch (type1& arg) {
    // catch statements
}
catch (type2& arg) {
    // catch block
}
catch (type3& arg) {
    // catch block
}
.
.
.
```

```
catch (typeN& arg) {
    // catch block
}
```

The try block can be as short as one statement or an all- encompassing block such as enclosing the *main* function code within a try block (this effectively causes the entire program to be monitored for exceptions).

When an exception is thrown, it can be caught by an applicable catch handler, which processes the block of statements associated with this catch handler. There can be more than one catch handler associated with a try block. The catch handler that is invoked depends on the type of the exception that is thrown. That is, if the data type specified by a catch handler matches the type of the exception, then that catch block is executed (and all other catch blocks are ignored). When an exception is caught, the catch argument will receive the exception value. The catch type can be any type including classes that we create. If an exception is not thrown (that is, no error occurs within the try block), then no catch block will be executed.

We can deliberately throw an exception in a try block with a *throw* statement:

```
// examples of throwing an exception

throw Foo;
throw “exception thrown”;
throw 20;
```

These *throw* statements generate an exception of the type specified by the argument to *throw,* for example, *type Foo, const char** or *int*.

Here is an example of throwing an exception:

```
#include <iostream>

int main()
{
    throw "test";
}

// output
terminate called after throwing an instance of type 'const char*'
```

If we throw an exception for which there is no applicable catch statement, an abnormal program termination will occur.

An unhandled exception causes the standard library function *terminate* to be invoked. By default, *terminate* calls the *abort* function to halt the program, however we can specify our own termination handler

```
// A simple exception handling example
#include <iostream>
using namespace std;

int main()
{
   cout << "Start\n";
   // specify a try block
   try {
         cout << "Inside try block\n";
         // throw an exception of type int
         throw 100;
         cout << "This will not execute";
       }

    // catch an error of type int
   catch (int& i) {
          cout << "Caught an exception, the value is: ";
          cout << i << "\n";
          cout << endl;
          }

   return 0;
}

// output
Start
Inside try block
Caught an exception, the value is: 100
```

the *catch(int& i)* handler processes exceptions of type *int*. Once an exception has been thrown, the try block is terminated and control passes to the matching catch block. *catch* is not called as in a function call (and is not callable), rather program execution is transferred (jumps) to it.

Usually, the code within a catch handler attempts to remedy the error by taking some appropriate action. If the error can be fixed, execution will continue with the statements following the catch block. However, sometimes an error cannot be fixed. In such a case, the catch handler will terminate the program with a call to a standard library function such as *exit(), terminate()* or *abort()*.

In order for an exception to be caught, the type of the exception must match the type specified in a catch handler. The code associated with a catch handler will be executed only if the handler catches the exception. Otherwise, execution simply bypasses the catch handler block altogether.

Levels Of Exception Safety

A function can implement and guarantee four levels of exception safety:

No Exception Safety	The function may throw an exception that is not handled. No function should ever offer this level of exception safety.
Basic Exception Safety	If an exception is thrown, it will be handled. Memory is guaranteed not to leak. There will be no corruption of data or memory afterwards, and all objects will be in a healthy and consistent state. However, it is not guaranteed that the data content will be the same as before the function or method was called.
Strong Exception Safety	Strong exception-safety guarantees everything that is guaranteed by the basic exception safety level. In addition, the content of the data will be recovered to exactly the same state as before the exception

	occurred. In other words, with this exception safety level we get rollback semantics.
No Throw Guarantee	The function guarantees that it will never throw an exception. This level of exception safety is also called **failure transparency**.

The C++ Standard Library implements at least basic exception safety. Strong exception safety is expensive to implement and should not be implemented unless absolutely necessary.

Destructors and move constructors must have a no throw guarantee. Constructors should also preferably have a no throw guarantee.

If we are confronted with an exception from which we cannot recover, the best strategy is to log the exception (if possible), generate a crash dump file for later analysis, and terminate the program immediately.

Catching Exception Class Types

An exception have any type, including class types that we create. In real-world programs, most exceptions will be class types rather than the C++ built-in types. The most common reason that we want to define a class type for an exception is to create an object that describes the error that has occurred. This information can then be used by the exception handler to help it process the error.

Here is an example of implementing an exception class type:

```
// Catching exception class types
#include <iostream>
#include <cstring>

using namespace std;

class FooException
{
```

```
public:
  std::string error_description;
  int errorcode;

  FooException(const char *s, int e)
  {
    error_description = s;
    errorcode = e;
  }
};

int main()
{
  int i = -9;
  try {
        if(i < 0)
           throw FooException("Number Not Positive", i);
      }

    // catch an error of type FooException
    catch (FooException& e)
     {
        cout << e.error_description << ": ";
        cout << e.errorcode << "\n";
     }

     return 0;
}

// output
Number Not Positive: -9
```

Standard Library Exceptions

When a function in the C++ standard library throws an exception, it will be an object derived from the base class *Exception*. To use exception objects specified in the standard library, we must include the <*exception*> header.

Using Multiple Catch Statements

A try block can throw exceptions of various types. We can thus have more than one catch handler associated with a try block. Each catch handler must catch a different type of exception.

Handling Derived-Class Exceptions

We have to be careful in ordering catch statements when catching exception types that involve polymorphic base and derived classes because a catch clause for a base class will also match any class derived from the base class. Thus, if we want to catch exceptions for a base class type and a derived class type, we must put the derived class catch block before the base class catch block. If we don't do this, the base class catch block will also catch all derived class exceptions.

```
// Catching derived class exceptions
#include <iostream>
using namespace std;

class Foo
{
public:
  virtual void print() {std::cout << "in Foo \n"; };
};

class Bar: public Foo
{
public:
  virtual void print() {std::cout << "in Bar \n"; };
};

int main()
{
  Bar derived;
  try {
        throw derived;
      }

  catch(Foo&  b) {
      cout << "Caught a base class exception\n";
  }
  catch(Bar& c) {
     cout << "This derived class catch block won't execute\n";
  }

  return 0;
}

// output
warning: exception of type 'Bar' will be caught by earlier handler
```

Here, because the *derived* object has *Foo* as it's polymorphic base class, the exception will be caught by the first catch block and the second catch block will never execute.

Catch All Exception Handler

In some circumstances you will want an exception handler that can catch all exceptions instead of just exceptions of a certain type. This is easy to accomplish. Simply use the following *catch all* syntax:

```
catch(...)
{
     // process all exceptions
}
```

the ellipsis matches all exception types.

One very good use for *catch(...)* is as the last catch handler in a group of catch handlers. In this capacity, it provides a useful default or "catch all" statement.

Rethrowing An Exception

We can rethrow an exception from within the body of a catch handler. We do this by calling *throw* with an optional argument. This causes an exception to be thrown to an outer *try/catch* block, if any.

If we rethrow an exception without specifying an argument, then the current exception will be rethrown. If an argument is supplied, then an exception with the same type as the argument will be rethrown.

The most common reason for re-throwing an exception is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler handles an another aspect.

When we rethrow an exception, it will not be caught in the same catch block. The exception will propagate outward to the next try/catch block that wraps around the current try/catch block.

```
// Example of "rethrowing" an exception
#include <iostream>
using namespace std;

void foo()
{
  try {
       throw "hello";           // throw a const char* exception
     }

  // catch a const char * exception

  catch(const char *) {
       cout << "Caught char * exception inside foo \n";
      // rethrow the exception out of
      // the current try/catch block
      throw;
  }
}

int main()
{
  try{
       foo();
     }

   catch(const char *) {
       cout << "Caught char * exception inside main\n";
   }

   return 0;
}

// output
Caught char * exception inside foo
Caught char * exception inside main
```

This example shows that if we rethrow an exception without specifying an argument, then the same exception will be rethrown.

Exceptions And Destructors

Exceptions should not be thrown in destructors. Unhandled exceptions in destructors will cause undefined behavior. If a destructor can throw an exception, it should be handled in the destructor itself.

terminate() and unexpected()

terminate and *unexpected* are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their declarations are:

```
void terminate();
void unexpected();
```

Both functions require the **<exception>** header.

The *terminate* function is called whenever the exception handling subsystem fails to find a matching catch handler for an exception. The *terminate* function is also called under various other, more obscure circumstances. For example, such a circumstance could occur when, in the process of unwinding the stack because of an exception, a destructor for an object being destroyed throws an unhandled exception.

In general, *terminate* is the handler of last resort when no other handlers for an exception are available. By default, *terminate* calls *abort* which aborts a program.

Setting Terminate and Unexpected Handlers

The *terminate* and *unexpected* functions simply call other functions to actually handle an error. By default, *terminate* calls *abort*, and *unexpected* calls *terminate*. Thus, by default, both of these functions halt program execution when an unhandled exception occurs. However, we can change the functions that are called by *terminate* and *unexpected*.

To change the terminate handler, use *set_terminate*:

```
terminate_handler* set_terminate(terminate_handler* newhandler) throw();
```

newhandler is a function pointer to a new terminate handler function. The *set_terminate* function returns a function pointer to the old terminate handler. The *newhandler* function pointer must be of type *terminate_handler*, which is defined as:

```
void (*terminate_handler) (void);
```

The only thing that the new terminate handler can do is stop program execution. It must not return to the program or resume it in any way.

```
# Example of setting a terminate handler

#include <iostream>
#include <cstdlib>
#include <exception>
using namespace std;

void my_TerminateHandler(void) {
cout << "Inside new terminate handler, aborting\n";
abort();
}

int main()
{
  // set a new terminate handler
  set_terminate(my_TerminateHandler);

  try {
     cout << "Inside try block\n";
     throw 100; // throw an error
  }

  catch (double i)
  { // won't catch an int exception
    cout << "will not execute\n";
  }

  return 0;
}

// output
Inside try block
Inside new terminate handler, aborting
```

The uncaught_exception Function

The C++ exception handling subsystem implements an additional useful function: *uncaught_exception*. It’s declaration is:

```
bool uncaught_exception(void);
```

This function returns true if an exception has been thrown but not yet caught. Once caught, the function returns false.

C++ Run-time Type Identification

Run-time type identification (**RTTI** for short) provides enhanced support for run-time polymorphism. The *typeid* operator allows us to identify the type of an object during the execution of a program. We can also use *typeid* to determine whether two objects belong to the same class.

C++ implements run-time polymorphism through the use of class inheritance, virtual functions, and base-class pointers. Since base-class pointers can be used to point to an object of a base class or any object derived from the base class, it is not always possible to know in advance what type of object will be pointed to by a base class pointer at any given moment in time. This determination can be made at run time, using run-time type identification.

The **typeid** operator returns a const reference to an object of type **type_info** that describes the type of an object. We must include the header file <*typeinfo*> in order to use *typeid:*

```
#include <typeinfo>
const type_info&  t = typeid(obj)
```

The primary use of the *typeid* operator is to determine whether two classes have the same type.

In the example above, *obj* is the object whose type we want to determine. It may be any type, including a native C++ type such as *int* or *double,* it can also be a class type that we have created.

The *type_info* class has the following public members:

```
        bool operator ==(const type_info &obj);
        bool operator !=(const type_info &obj);
        const char *name();
        bool before(const type_info &obj);
```

The overloaded operators == and != enable the type comparison of two type_info objects.

The member function *name* returns the name of an object's type. For example:

```
        #include <typeinfo>

        string obj_type_name = typeinfo(obj).name();
```

```
// A simple example that uses typeid

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

class MyClass1 {

};
class MyClass2 {

};

int main()
{
     int i, j;
     float f;
     char *p;
     MyClass1 obj1, obj3;
     MyClass2 obj2;

     cout << "The type of i is: " << typeid(i).name() << endl;
     cout << "The type of f is: " << typeid(f).name() << endl;
     cout << "The type of p is: " << typeid(p).name() << endl;

     cout << "The type of obj1 is: " << typeid(obj1).name() << endl;
     cout << "The type of obj2 is: " << typeid(obj2).name() << "\n\n";

     if(typeid(i) == typeid(j))
        cout << "The types of i and j are the same\n";
     if (typeid(i) != typeid(f))
        cout << "The types of i and f are not the same\n";
     if(typeid(obj1) != typeid(obj2))
        cout << "obj1 and obj2 are of differing types\n";
     if(typeid(obj1) == typeid(obj3))
        cout << "obj1 and obj3 are the same type\n";

    return 0;
}

       // output
       The type of i is: i
       The type of f is: f
       The type of p is: char*
       The type of ob1 is: myclass1
       The type of ob2 is: myclass2

       The types of i and j are the same
       The types of i and f are not the same
       ob1 and ob2 are of differing types
       ob1 and ob3 are the same type
```

The most important use case for *typeid* occurs when it's argument is a pointer to a polymorphic base class. In this case, it will automatically return the type of the actual object pointed to, whether it is a base-class object or an object of a derived class.

Therefore by using *typeid* we can determine at run-time the actual type of the polymorphic object that is being pointed to by a base-class pointer.

```
// An example that uses typeid on a polymorphic class hierarchy
#include <iostream>
#include <typeinfo>
using namespace std;

// Mammal is a polymorphic class
class Mammal
{
public:
  virtual bool lays_eggs() { return false; }
};

class Cat: public Mammal {
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
};

int main()
{
   Mammal *p, AnyMammal;
   Cat cat;
   Platypus platypus;
   p = &AnyMammal;

   cout << "p is pointing to an object of type ";
   cout << typeid(*p).name() << endl;

   p = &cat;
   cout << "p is pointing to an object of type ";
   cout << typeid(*p).name() << endl;

   p = &platypus;
   cout << "p is pointing to an object of type ";
   cout << typeid(*p).name() << endl;

   return 0;
}

# output
p is pointing to an object of type Mammal
p is pointing to an object of type Cat
p is pointing to an object of type Platypus
```

Since the base class is polymorphic, *p* points to the actual object even though it is a pointer to a base class type.

When *typeid* is applied to a base-class pointer of a polymorphic class, the type of object pointed to is the type of the actual object. When *typeid* is applied to a pointer of a non-polymorphic class hierarchy, then the type of the object to which the pointer was pointing at the time of its declaration will be returned. For example, comment out the virtual keyword before the function *lays_eggs* in Mammal and then compile and run the program.

When the pointer being dereferenced is null, *typeid* will throw a *bad_typeid* exception.

typeid And References

References to an object of a polymorphic class hierarchy work in the same manner as pointers. When *typeid* is applied to a reference to an object of a polymorphic class, it will return the type of the object actually being referred to, which may be an object of a derived class.

```
// Demonstrate typeid with a reference parameter
#include <iostream>
#include <typeinfo>
using namespace std;

// Mammal is a polymorphic class
class Mammal
{
public:
  virtual bool lays_eggs() { return false; }
};

class Cat: public Mammal {
};

class Platypus: public Mammal
{
public:
  bool lays_eggs() { return true; }
};


void MammalType(Mammal& obj)
{
   cout << "obj is referencing an object of type ";
   cout << typeid(obj).name() << endl;
}

int main()
{
   Mammal AnyMammal;
   Cat cat;
   Platypus platypus;

   MammalType(AnyMammal);
   MammalType(cat);
   MammalType(platypus);
}

// output
obj is referencing an object of type Mammal
obj is referencing an object of type Cat
obj is referencing an object of type Platypus
```

Applying Typeid To Class Templates

The *typeid* function can be applied to instances of class templates. The type of an object that is an instance of a class template is determined by the type parameters that are actually used to instantiate the specialization. Two instances of the same class template that are instantiated using different template parameters are different types.

```
// Using typeid with class templates
#include <iostream>
#include <typeinfo>
using namespace std;

// Foo template class declaration
template <class T>
class Foo
{
  T a;
public:
  Foo(T i) { a = i; }
};

int main()
{
  Foo<int> object1(10), object2(9);
  Foo<double> object3(7.2);

  cout << "Type of object1 is ";
  cout << typeid(object1).name() << endl;
  cout << "Type of object2 is ";
  cout << typeid(object2).name() << endl;
  cout << "Type of object3 is ";
  cout << typeid(object3).name() << endl;
  cout << endl;

 if (typeid(object1) == typeid(object2))
     cout << "object1 and object2 have same type\n";

 if(typeid(object1) == typeid(object3)) cout << "Error\n";
 else
    cout << "object1 and object3 are different types\n";

 return 0;
}

// output
Type of object1 is Foo
Type of object2 is Foo
Type of object3 is Foo
```

```
o1 and o2 have same type
o1 and o3 are different types
```

In this example, even though *object1* and *object3* are generated from the same class template, their actual types do not match and consequently these objects are different types.

C++ Type Cast Operators

A type cast changes the type of a variable. C++ defines five type casting operators. The first is the traditional cast inherited from C. The remaining casts are *dynamic_cast*, *const_cast*, *static_cast* and *reinterpret_cast*.

dynamic_cast

dynamic_cast safely converts pointers and references to polymorphic classes up and down an inheritance hierarchy. It's syntax is:

```
dynamic_cast< new-type > ( expression )
```

The parenthesis around the expression that is being type cast is required. *new_type* is a class in a polymorphic class hierarchy.

If the cast is successful, *dynamic_cast* returns a value of type *new-type*. If the cast fails and *new-type* is a pointer type, it returns a null pointer. If the cast fails and *new-type* is a reference type, it throws an exception of type *std::bad_cast*.

Look at the following program:

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

class Base
{
protected:
   int m_value;

public:
   Base(int value) :  m_value(value) { }
   virtual ~Base() {}
};

class Derived : public Base
{
protected:
   string m_name;

public:
   Derived(int value, string name) : Base(value), m_name(name) {}
   const string& getName() const { return m_name; }
};

Base* getObject(bool bDerivedObject)
{
   if (bDerivedObject)
      return new Derived(1, "Apple");
   else
      return new Base(2);
}

int main()
{
  Base* bptr = getObject(true);
  delete bptr;

  return 0;
}
```

Notice that in this program, the function *getObject()* always returns a pointer to a Base class object, but the pointer may actually be pointing to either a Base class object or a Derived class object. The problem is, how can we call *Derived::getName()* when we do not know whether the pointer returned by the function *getObject* is a pointer to a Base class object or a Derived class object.

One way to solve this problem would be to add a virtual function to *Base* called *getName()*. Now the base class pointer can dynamically resolve to *Derived::getName())* if *bptr* actually points to a derived class object. The problem with this solution is that *getName* may make no sense in the base class, in which case we are simply polluting this class.

We know that C++ lets us implicitly convert a Derived class pointer to a Base class pointer. In fact, *getObject()* does just this. This process is called **upcasting**. However, what if there was a way to convert a Base class pointer back into a Derived class pointer? Then we could call *Derived::getName()* directly using this pointer, and we would not have to worry about virtual function call resolution at all.

Downcasting converts a base class pointer to a pointer to a derived class. Downcasting is done with the *dynamic_cast* operator. We can rewrite our program as follows:

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

class Base
{
protected:
         int m_value;

public:
   Base(int value) :  m_value(value) { }
   virtual ~Base() {}
};

class Derived : public Base
{
protected:
   string m_name;

public:
   Derived(int value, string name) : Base(value), m_name(name) {}
   const string& getName() const { return m_name; }
};

Base* getObject(bool bDerivedObject)
{
   if (bDerivedObject)
      return new Derived(1, "Apple");
   else
      return new Base(2);
}

int main()
{
   Base* bptr = getObject(true);

   // use dynamic cast to downcast the Base class pointer to a
   // Derived class pointer
   Derived* dptr = dynamic_cast<Derived*>(bptr);

  // make sure d is non-null, d will be null if the
  // downcast fails
  if (dptr) {
    std::cout << "The name of Derived class: ";
    std::cout <<  dptr->getName() << '\n';
    delete dptr;
  }
  else delete bptr;

  return 0;
}

// output
The name of Derived class: Apple
```

If *bptr* is actually a pointer to a derived class object, *dynamic_cast* can be used to downcast it from a base class pointer to a derived class pointer. But if *bptr* actually points to a *Base* class object (or some other type of object), then the *dynamic_cast* will fail and return a null pointer.

Note*:* *dynamic_cast* only works with polymorphic classes. Thus the base class must contain at least one virtual function. An exception will be thrown if the base class is not polymorphic.

The next point to notice about a *dynamic_cast* is that it returns a pointer to a derived object if it succeeds and a null pointer otherwise.

Since *dynamic_cast* involves expensive run-time processing it is best to use *static_cast* if we are sure that the base class pointer can be safely downcast to a derived class pointer.

Replacing dynamic_cast With typeid

Consider:

```
Base     *bp;
Derived  *dp;
...
if (typeid(*bp) == typeid(Derived))
    dp = (Derived *) bp;            // traditional C type cast
```

In this case, a traditional C-style cast is used to actually perform the cast. This is safe because the *if* statement checks the validity of the type cast using *typeid* before the cast is actually performed. However, a better way to accomplish this is to replace the *typeid* operator and the *if* statement with a *dynamic_cast:*

```
dp = dynamic_cast<Derived *> (bp);
```

The parentheses around *bp* are required.

dynamic_cast And Different Derived Types

Consider the following example:

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

class Base
{
  virtual const char* print()
  {
    cout << "base class" << endl;
    return "in base class";
  }
};

class Derived1 : public Base
{
public:
  const char* print()
  {
     cout << "derived1 class" << endl;
     return "derived1 class";
  }
};

class Derived2 : public Base
{
public:
   const char* print() {
     cout << "derived2 class" << endl;
     return "derived2 class";
   }
};
```

```
int main(void)
{
  Base*  bptr =  new Derived1;

  Derived2* d2 = dynamic_cast<Derived2*> (bptr);
  std::cout << d2->print() << std::endl;
}

// output
segmentation fault
```

In this example, the dynamic cast fails because we are trying to typecast the base class pointer to a type which is not a downcast from a pointer to Derived1.

dynamic_cast And References

dynamic_cast also works with references:

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

class Base
{
  virtual void print()
{
   cout << "base class" << endl; }
};

class Derived1 : public Base
{
  void print()
  {
    cout << "derived1 class" << endl;
  }
};

class Derived2 : public Base
{
  void print()
  {
    cout << "derived2 class" << endl;
  }
};

int main(void)
{
  Base* baseptr;

  baseptr = (Base*) new Derived1;            //upcast

  // succeeds
  Derived1& ref = dynamic_cast<Derived1&> (*baseptr);
  cout << "dynamic cast from base class to derived1 succeeded: ";
  cout << &ref << endl;

  try {
        // fails
        Derived2& ref = dynamic_cast<Derived2&> (*baseptr);
        std::cout << &ref << std::endl;
      }
  catch(...) {
        std::cout << "dynamic cast to Derived2 failed" << std::endl;
  }
}

// output
dynamic cast from base class to derived1 succeeded: 0xe56b20
dynamic cast to Derived2 failed
```

An exception will be thrown if the dynamic cast of a reference fails.

static_cast

The `static_cast` operator converts an expression from one type to another type at compile-time. *static_cast* is intended to be a safer version of the traditional C-style cast. Its syntax is:

```
static_cast<target-type> (expression)
the parenthesis is required
```

target-type specifies the target type of the cast, and *expression* is the expression being cast into the new type. The parenthesis around *expression* is required.

In C, the compiler performs casts implicitly when required. For example:

```
int i = 5;
float f;
```

```
f = i;
```

The compiler will implicitly convert the *int* to a *float* when it performs the assignment. Implicit type conversions can give rise to subtle errors which are difficult to detect:

```
float f = 5.1;
int i;
i = f;
```

The compiler will do an implicit conversion from the *float* to an *int*. This will produce a loss of precision (i = 5). In the example below, a dangerous implicit type conversion will be performed by the compiler:

```
int i = 7;
double*  f;

f = &i;
*f = 10.5;
```

A *static_cast* makes a cast explicit and assists debugging by letting the compiler check a cast at compile-time. For example, the compiler will emit an error for the previous pointer cast.

Here is another example:

```
// static_cast example

#include <iostream>
using namespace std;

int main()
{
  int i;
  for(i = 0; i < 5; i++)
     cout << static_cast<double> (i) / 3 << " ";
  cout << endl;

  return 0;
}

// output
```

```
0 0.333333 0.666667 1 1.33333
```

static_cast can be used in the following cases:

- narrowing conversions between numbers (int to short, double to int, …)
- **conversions between integrals and enums.**
- **conversion from void* to any other pointer type.**
- downcasts of pointers or references in class hierarchies when we know the dynamic type of the object.

The **static_cast** operator cannot cast away the **const**, **volatile**, or **__unaligned** attributes.

const_cast

The *const_cast* operator is used to explicitly remove the *const* and *volatile* modifiers. The target type must be the same as the source type except for the possible presence of a *const* or *volatile* attribute. The general form of *const_cast* is:

```
const_cast<type> (expr)
the parenthesis around expr is required
```

Here, *type* specifies the target type of the cast, and *expr* is the expression being modified.

Here is an example:

```
// Demonstrate const_cast

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

```
void squareValue(const int *value)
{
  int* p;
  // cast away the const attribute on value
  p = const_cast<int *> (value);

   // now, modify the object through p
  *p = (*value) * (*value);
}

int main()
{
int tmp = 10;
const int x = tmp;
  cout << "x before call: " << x << endl;
  squareValue(&x);
  cout << "x after call: " << x << endl;

  return 0;
}

// output
x before call: 10
x after call: 100
```

reinterpret_cast

The *reinterpret_cast* operator is used in two situations:

- to cast between two potentially incompatible pointer types,
- cast a pointer type to an integer type or an integer type to a pointer type

reinterpret_cast simply does a bit copy from the source to the destination type. An exception will occur if the target type has less allocated space than the source type. *reinterpret_cast* is a dangerous conversion and it can produce non-portable code. The only thing that *reinterpret_cast* promises is that we can cast back to the original type without hindrance.

The general form of *reinterpret_cast* is:

```
new_type x = reinterpret_cast<new_type> (expr)
the parenthesis is required
```

Here, *new_type* specifies the target type of the cast, and *expr* is the expression being cast into the new type.

```
// An example that uses reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
  size_t x;
  const char *p = "This is a string";

  // cast pointer to long
  x = reinterpret_cast<long> (p);
  cout << "integer is: " << x << endl;
  return 0;
}

// output
integer is: 4197988
```

Here, *reinterpret_cast* typecasts the pointer p into a long integer type. This conversion represents a fundamental change of types and is a typical use of *reinterpret_cast.* Another example is changing a pointer to one ty*pe* of object (*Apricot**) to a pointer to another type of object (*Orange**).

```
Orange* ptrOrange = reinterpret_cast<Orange*> ptrApricot;
```

Note: *reinterpret_cast* can cast unrelated pointer types, but can't remove *const* or volatile qualifiers. We need *const_cast* for this.

C++ And Assembler

There are a few highly specialized situations that C++ cannot handle. For example, there is no C++ statement to disable CPU interrupts. C++ provides access to very low level functionality by allowing us to drop into assembly language code at any time from C++ code. This functionality is provided through the *asm* statement.

With a*sm* we can embed assembly language code directly into a C++ program. This assembly language code is compiled without any modification, and it becomes part of a program's code at the point at which the *asm* statement occurs.

The general form of the *asm* keyword is:

```
asm ("op-code");
```

op-code is an assembly language instruction that will be embedded into our C++ program. Several compilers also allow the following forms of *asm*:

```
asm op-code;

asm {
      op-code
      op-code
      ...
    }
```

Here is an example:

```
#include <iostream>

int main()
{
     asm int 5;                 // generate interrupt 5
     return 0;
}
```

C++ Linkage Specification

In C++ we can specify how a function is linked into our program. By default, functions are linked as C++ functions. However, by using a linkage specification, you can cause the object code for a function written in a different language to be linked into a C++ program. The general form of a linkage specifier is:

```
extern "language" function declaration;
```

Language denotes the language of the function that is to be linked. All C++ compilers support both C and C++ linkage. Some will also allow linkage to Fortran and Pascal language functions.

Every language has it’s own conventions about how function parameters and the function return value have to be loaded on the stack, and how CPU registers are to be saved when program code calls a function. The *extern* keyword informs the compiler about this convention.

The following program causes the function *myCfunc*( ) to be linked as a C function:

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

extern "C" void myCfunc(int);

int main()
{
     myCfunc(5);
     return 0;
}

// Because of the extern statement,
// the following function will be linked as a C function.
void myCfunc(int val)
{
    cout << "This links as a C function with value: ");
    cout << val << endl;
}
```

A linkage specification must be declared at the global level; it cannot be placed inside a function.

We can use the following linkage specification to link more than one function at a time:

```
extern "language" {
                     function declaration;
                     ...
                  }
```

Summarizing the Differences Between C and C++

For the most part, Standard C++ is a superset of Standard C, and virtually all C programs are also C++ programs. However, a few differences do exist.

In C++, local variables can be declared anywhere within a block. In C, they must be declared at the start of a block, before any "action" statements occur.

In C, a function declared as follows:

```
int f();
```

says nothing about the parameters of the function. In C, when there is nothing specified between the parentheses following the function's name, this means that nothing is being stated, one way or the other, about any parameters of the function. It might have parameters, or it might not. However, in C++, a function declaration like this means that the function does not have any parameters. In C++, the following two declarations are equivalent:

```
int f();
int f(void);
```

In C++, all functions must have declarations which are typically in header files.

A small but potentially important difference between C and C++ is that in C, a character constant such as *‘c’* is automatically elevated to an integer. In C++ it is not.

In C, although it is unusual, you can call *main* from within your program. This is not allowed in C++.

In C, it is not an error to declare a global variable several times, even though this is bad programming practice. In C++, it is an error.

In C, you cannot take the address of a register variable. In C++, this is allowed, however the *register* keyword is deprecated.

In C, if no type specifier is present in a declaration statement, the type *int* is assumed. This "default-to-int" rule is not permissible in C++.

C++ Standard Template Library

The C++ Standard Template Library provides solutions to a variety of common programming problems. The library is composed of three foundational components: *containers*, *iterators* and *algorithms*.

Containers

Containers are objects that hold other objects. There are several different types of containers. For example, the *vector* class (container) defines a dynamic array, the deque container implements a double-ended queue, and the list container provides a linear list. These containers are called **sequence containers**. The STL also implements **associative containers**, which map keys to values and thus enable the retrieval of values from keys.

Each container class defines a set of member functions that may be applied to objects in the container.

Algorithms

Algorithms act on containers. They provide the means by which we can manipulate the contents of containers. Algorithms can initialize, sort, search and transform the contents of containers.

Iterators

Iterators are objects that behave like pointers. They give us the ability to traverse through the items in a container in much the same way that we can use a pointer to traverse through an array. There are five types of iterators:

Random Access Iterators	Store and retrieves values. Elements are accessed randomly.

Bidirectional Iterators	Store and retrieve values. These iterators move forward and backwards sequentially.
Forward Iterators	Store and retrieve values. This type of iterator is forward moving only.
Input Iterators	Store but does not retrieve values. Forward moving only.
Output Iterators	Retrieves but does not store values. Forward moving only.

Iterators are handled just like pointers. We can increment and decrement them. We can apply the * dereferencing operator to them to get the value pointed to.

The STL also supports **reverse iterators**. Reverse iterators move through a sequence in the reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

Other STL Elements

In addition to containers, algorithms, and iterators, the STL also implements several other components such as allocators, predicates, comparison functions, and function objects (*functors*).

Each container has at least one allocator defined for it. Allocators manage memory allocation for a container. The default allocator is an object of class *allocator.* We can define our own allocators if needed.

Some of the algorithms and containers use a special type of function called a **predicate function.** A predicate function returns true or false.

The templates in the header file **<functional>** help us construct **function objects (*functors*).** Function objects can be used in place of function pointers in many places.

There are a number of predefined function objects declared in <*functional*>. These are:

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

Using function objects rather than function pointers allows the STL to generate more efficient code.

The STL also uses **binders** and **negators**. A binder binds an argument to a function object. A negator returns the complement of a predicate.

The STL also implements **adaptors**. An adaptor transforms one thing into another. For example, the queue container is an adaptor for the deque container.

The Container Classes

Containers are STL objects that store other objects. The STL defines the following containers:

Container	**Description**	**Required Header**
bitset	A set of bits.	<bitset>
deque	A double-ended queue.	<deque>
list	A linear list.	**<list>**
map	Stores key/value pairs. Each key is associated with only one value.	**<map>**
multimap	Stores key/value pairs in which one	<map>

	key may be associated with two or more values.	
multiset	A set in which each element is not necessarily unique.	<set>
priority_queue	A priority queue.	<queue>
queue	A queue.	<queue>
set	A set in which each element is unique.	<set>
stack	A stack.	<stack>
vector	A dynamic array.	**<vector>**

General Theory Of Operation

Firstly, we must decide on the type of container that we want to use. Each container type offers certain benefits and trade-offs. For example, a *vector* is a very good choice when a random-access, array-like object is required and not too many insertions or deletions will occur. A list has low-cost insertions and deletions but has slower access speed than a vector.

Once we have chosen a container, we can use it's member functions to add elements to the container, access and modify container elements, and delete elements.

Except for the *bitset* container, a container will automatically grow as needed when elements are added to it and shrink when elements are removed.

A common way to access elements within a container is through an iterator. The sequence and associative containers implement the member functions *begin* and *end*, which return iterators to the start and end of a container. These iterators are very useful for accessing the contents of a container. For example, in order to cycle through the elements of a container, we obtain an iterator to its beginning using the *begin()* function and then increment the iterator until its value is equal to *end()*.

Important Note: In general, STL containers will not accept pointer variables or reference parameters in template declarations.

Vector Containers

A vector is a dynamic array. This is an array that can automatically grow or shrink as needed. In C++, the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be varied at run-time to accommodate changing program conditions. A *vector* container solves this problem by automatically allocating memory for the array, as needed.

The template declaration for a vector is:

```
template <class T, class Allocator = allocator<T>>
class vector {};
```

T is the type of data to be stored in the vector. *Allocator* specifies the memory allocator, this defaults to the standard allocator.

A vector container has the following constructors:

```
explicit vector(const Allocator &a = Allocator() );
```

This constructor creates an empty vector.

The **explicit keyword** prevents any implicit type conversions from occurring. The = sign indicates a default template parameter value.

The second constructor makes a vector that has *num* elements each with the value *val*.

```
explicit vector(size_t num, const T&  val,
     const Allocator &a = Allocator());
```

The third constructor makes a vector that contains the same elements as a second vector.

```
vector(const vector<T, Allocator>& second_vector);
```

The fourth constructor makes a vector that contains elements in the range specified by the iterators *start* and *end:*

```
vector(Iterator_t start, Iterator_t end, const Allocator &a =
Allocator());
```

Any object that will be stored in a vector must have a constructor. It must also define the < and == operators. Some compilers may require that other comparison operators be defined. All of the built-in C++ types automatically satisfy these requirements.

Although vector template syntax looks complex, declaring a vector is easy. Here are some examples:

```
// create a zero length integer vector
vector<int> iv;

 // create a five element char vector
vector<char> cv(5);

// create a five character vector initialized with ‘x’
vector<char> cv(5, 'x');

  // create a vector from the dv integer type vector
vector<int> dv2(dv);
```

STL defines the following comparison operators for vectors:

==, <, <=, !=, >, >=

The index operator [ ] implements random access to the elements of a vector.

The Vector container has numerous class member functions. The *size* function returns the current size of the vector. The *begin* function returns an iterator to the start of the vector. The *end* function returns an iterator to the end of the vector.

The *push_back* function pushes a value onto the end of a vector. If necessary, the vector is increased in length to accommodate the new element. *pop_back* removes the last element of the vector. i*nsert* inserts an element at a specified position in the vector. We can remove all of the elements in a vector with *clear*. *empty()* is a predicate that tests if a vector is empty.

The following example demonstrates a vector class:

```
// Demonstrate a vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  // create an integer vector of length 10
  vector<int> v(10);
  size_t i;

  // display the size of v
  cout << "Size = " << v.size() << endl;

  // assign values to the elements of the vector
    for (i = 0; i < 10; i++) v[i] = i;

  // display contents of the vector
  cout << "Current Contents:\n";
  for (i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // push more values onto the end of the vector,
  // the vector will grow as needed
  for (i = 0; i < 5; i++) v.push_back(i + 10);

  // display the current size of v
  cout << "Vector size now = " << v.size() << endl;

  // display contents of the vector
  cout << "Current contents:\n";
  for (i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // change the contents of the vector
  for (i = 0; i < v.size(); i++) v[i] = v[i] + 100;

  cout << "Modified vector contents:\n";
  for (i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}

// output
size = 10
Current Contents:
0 1 2 3 4 5 6 7 8 9
Vector size now = 15
Current contents:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Modified vector contents:
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114
```

Accessing a Vector Through an Iterator

Arrays and pointers are closely related in C and C++. An array element can be accessed either through it's index or through a pointer. In the STL we can also access the data members of a vector with an index or through the use of an iterator:

```
// Access the elements of a vector through an iterator
#include <iostream>
#include <vector>
using namespace std;

int main()
{
      vector<int> v(10);      // create an int vector of length 10

      // create an iterator to a (generic) int vector
      vector<int>::iterator p;
      size_t i;

    // get the iterator to the first element of v
     p = v.begin();
     i = 0;

    // assign values to elements in vector v
    while (p != v.end()) {
         *p = i;
         p++;
         i++;
    }

// display contents of vector v
cout << "Original vector contents:\n";

p = v.begin();
while(p != v.end()) {
      cout << *p << " ";
      p++;
}
cout << "\n\n";

// change the contents of vector v
p = v.begin();
while(p != v.end()) {
      *p = *p + 100;
      p++;
}

// iterate through the contents of the changed vector
cout << "Modified vector contents:\n";
p = v.begin();
while(p != v.end()) {
      cout << *p << " ";
      p++;
}
cout << endl;

return 0;
                }

// output
Original vector contents:
0 1 2 3 4 5 6 7 8 9

Modified vector contents:
100 101 102 103 104 105 106 107 108 109
```

Understanding begin() And end()

begin() returns an iterator to the first element of the vector. This iterator can then be used to access the vector one element at a time by incrementing the iterator. This process is similar to the way a pointer is used to access the elements of an array. The *end()* member function determines when the end of the vector has been reached.

end() does not return an iterator to the last element in a container. Instead, it returns an iterator that is one past the last element. Thus, the last element in a container is pointed to by *p.end( ) - 1*.

Inserting and Deleting Elements in a Vector

In addition to putting new values at the end of a vector, we can insert elements at a specific index position using the *insert* function. We can also remove elements using *erase*.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
     vector<char> v(10);
     vector<char> v2;
     char str[] = "<Vector>";

     size_t i;
    // initialize v
   for (i=0; i < 10 ; i++) v[i] = 'a';

    // copy characters in str into v2
    for (i=0; str[i] != '\0';  i++) v2.push_back(str[i]);

   // display the contents of the vector
     cout << "Original contents of v:\n";
     for (i = 0; i < v.size(); i++) cout << v[i] << " ";
     cout << "\n\n";

      // get an iterator to vector v
     vector<char>::iterator p = v.begin();
     // point to the 3rd element of v
     p += 2;

    // insert 10 X's into v
    v.insert(p, 10, 'X');

   // display contents after insertion
   cout << "Size after inserting X's = " << v.size() << endl;
   cout << "Vector contents after insertion:\n";
   for(i = 0; i < v.size(); i++) cout << v[i] << " ";
   cout << "\n\n";

// remove the inserted elements
p = v.begin();
                       // point to 3rd element
p += 2;
// remove next 10 elements
v.erase(p, p+10);

// display contents after deletion
cout << "Vector size after erase = " << v.size() << endl;
cout << "Vector contents after erase:\n";
for (i = 0; i < v.size(); i++) cout << v[i] << " ";
cout << "\n\n";

// Insert v2 into v
v.insert(p, v2.begin(), v2.end());

cout << "Size after v2's insertion into v = ";
cout << v.size() << endl;
cout << "Contents of vector after insertion:\n";

for (i = 0; i < v.size(); i++) cout << v[i] << " ";
     cout << endl;
return 0;
   }
```

Output:

```
Original contents of v:
a a a a a a a a a a

Size after inserting X's = 20
Vector contents after insertion:
a a X X X X X X X X X X a a a a a a a a

Vector size after erase = 10
Vector contents after erase:
a a a a a a a a a a
Size after v2's insertion into v = 18
Contents of vector after insertion:
a a < V e c t o r > a a a a a a a a
```

Storing Class Objects in a Vector

Any type of object can be stored in a Vector including objects of classes that we create, providing that they have a constructor and implement the comparison operators > and ==.

```
// Store class objects in a vector.
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

class DailyTemperature {
     int temperature;
     public:
     // constructors
     DailyTemperature() { temperature = 0; }
     DailyTemperature(int x) { temperature = x; }
     DailyTemperature& operator=(int x) {
                  temperature = x; return *this;
        }

       double get_temperature() { return temperature; }
};

bool operator<(DailyTemperature a, DailyTemperature b)  {
     return a.get_temperature() < b.get_temperature();
}

bool operator==(DailyTemperature a, DailyTemperature b) {
     return a.get_temperature() == b.get_temperature();
}

int main()
{
     vector<DailyTemperature> v;
     size_t i;

     for(i = 0; i < 7; i++)
          v.push_back(DailyTemperature(60 + rand()%30));

     cout << "Fahrenheit temperatures:\n";
     for ( i= 0; i < v.size(); i++)
           cout << v[i].get_temperature() << " ";
     cout << endl;

// convert from Fahrenheit to Centigrade
for (i = 0; i < v.size(); i++)
     v[i] = (v[i].get_temperature() - 32) * 5/9 ;
cout << "Centigrade temperatures:\n";

for (i = 0; i < v.size(); i++)
     cout << v[i].get_temperature() << " ";

return 0;
}

//output
Fahrenheit temperatures:
73 76 87 85 83 85 76
Centigrade temperatures:
22 24 30 29 28 29 24
```

List Containers

A list container implements a bidirectional, linear list. Unlike vectors which have random access, lists can only be accessed sequentially. Since lists are bidirectional, they can be accessed by traversing front to back or back to front. Lists have very fast insertion and deletion times but they have slower access times than vectors.

The declaration for a list template is:

```
template <class T, class Allocator = allocator<T>> class list;
```

Here, *T* is the type of data stored in the list. The allocator is specified by Allocator, which defaults to the standard allocator.

We instantiate a list as follows:

```
include <list>

list<int> alist;              // create an empty list of integers
list<Foo> foo;                // create an empty list of Foo objects
```

The following list constructor constructs an empty list.

```
explicit list(const Allocator &a = Allocator( ) );
```

The second constructor creates a list with *num* elements of type T and value ‘g’.

```
explicit list(size_type num, const T &val = ‘g’, const Allocator &a
= Allocator( ));
```

The third form constructs a list that contains the same elements as a second list:

```
list(const list<T, Allocator> &second_list);
```

The fourth constructor form constructs a list that contains the elements in the range specified by an iterator’s *start* and *end:*

```
list(InIter start, InIter end, const Allocator &a = Allocator());
```

The STL implements the following comparison operators for lists:

==, <, <=, !=, >, >=

Some commonly used list member functions are:

void clear( )	Removes all the elements of a list.
bool empty( ) const;	Returns true if the list is empty, false otherwise.
iterator begin();	Returns an iterator to the first element of a list
iterator end( );	Returns an iterator to the end of a list

iterator erase(iterator i);	Erases the element pointed to by the iterator.
iterator insert(iterator i, const T &val);	Inserts val immediately before the element pointed at by i. An iterator to the element is returned.
T& front( );	Returns a reference to the first element in a list.
T& back( );	Returns a reference to the last element in a list.
void pop_back( );	Removes the last element of a list
void pop_front( );	Removes the first element of a list.
void push_back(const T &val);	Adds the element val to the back of the list.
void push_front(const T &val);	Add the element val to the front of the list.
void remove(const T &val);	Remove elements with value val from the list.

An object that will be held in a list container must have a default constructor.

Here is an example of a list:

```
// basic list management
#include <iostream>
#include <list>
using namespace std;

int main()
{
        list<int> lst;           // create an empty list of integers
        int i;

        for (i = 0; i < 10; i++) lst.push_back(i);

        cout << "Size = " << lst.size() << endl;

        cout << "Contents: ";
        // get an iterator to the list
        list<int>::iterator p;                // declaration
        p = lst.begin();                      // definition

        while(p != lst.end()) {
              cout << *p << " ";
              p++;
        }
        cout << "\n\n";

        // change the contents of the list
        p = lst.begin();
        while(p != lst.end()) {
              *p = *p + 100;
                p++;
       }

       cout << "Contents modified: ";
       p = lst.begin();

       while(p != lst.end()) {
               cout << *p << " ";
               p++;
       }
        return 0;
}

Output
Size = 10
Contents: 0 1 2 3 4 5 6 7 8 9
Contents modified: 100 101 102 103 104 105 106 107 108 109
```

push_front vs push_back

We add elements to the end of a list with *push_back*. To add elements to the front of a list, use *push_front*.

Storing Class Objects in a List

Here is an example that uses a list to store objects of type *Foo*. Notice that the <, >, !=, and == are overloaded for objects of type *Foo*. A list needs a way to compare elements when searching, sorting, or merging list elements.

```
// Store class objects in a list.
#include <iostream>
#include <list>
#include <cstring>

using namespace std;

class Foo {
     int a, b;
     int sum;
     public:
     // constructors
     Foo() { a = b = 0; sum = 0;}
     Foo(int i, int j) {
                  a = i;
                  b = j;
                  sum = a + b;
              }

      int getsum() { return sum; }

      friend bool operator<(const Foo &o1, const Foo &o2);
      friend bool operator>(const Foo &o1, const Foo &o2);
      friend bool operator==(const Foo &o1, const Foo &o2);
      friend bool operator!=(const Foo &o1, const Foo &o2);
};

bool operator<(const Foo &o1, const Foo &o2)
{
return o1.sum < o2.sum;
}

bool operator>(const Foo &o1, const Foo &o2)
{
return o1.sum > o2.sum;
}

bool operator==(const Foo &o1, const Foo &o2)
{
return o1.sum == o2.sum;
}
```

```
bool operator!=(const Foo &o1, const Foo &o2)
{
return o1.sum != o2.sum;
}


int main()
{
     int i;
     // create the first list of type Foo elements
     list<Foo> lst1;

    for (i = 0; i < 10; i++) lst1.push_back(Foo(i, i));
    cout << "First list: ";

    list<Foo>::iterator p = lst1.begin();
    while (p != lst1.end()) {
          cout << p->getsum() << " ";
          p++;
   }
   cout << endl;

   // create a second list
   list<Foo> lst2;

for (i = 0; i < 10; i++) lst2.push_back(Foo(i*2, i*3));
cout << "Second list: ";

p = lst2.begin();
while(p != lst2.end()) {
     cout << p->getsum() << " ";
     p++;
}
cout << endl;

// merge list lst2 into lst1
lst1.merge(lst2);
// display merged list
cout << "Merged list: ";
p = lst1.begin();
while (p != lst1.end()) {
     cout << p->getsum() << " ";
     p++;
}
return 0;

}
```

```
output:
First list: 0 2 4 6 8 10 12 14 16 18
Second list: 0 5 10 15 20 25 30 35 40 45
Merged list: 0 0 2 4 5 6 8 10 10 12 14 15 16 18 20 25 30 35 40 45
```

Maps (std::map)

The map class implements associative containers. An associative container maps keys to values. A map can be visualized as a set of (*key, value*) pairs. Two different keys cannot map to the same value. We retrieve a value from a map by using its associated key. We can look up a value in a map if we have it’s key.

A map can only hold unique keys. Duplicate keys are not allowed.

Maps maintain their keys in sorted order.

The standard library also implements the **multimap** template, this allows us to construct maps where more than one key maps to the same value.

The syntax for a map template declaration is:

```
template <class Tkey, class Tvalue, class Comp = less<Key>, class
Allocator = allocator<T>>
class map;
```

Tkey is the data type of the keys in the map. Tv*alue* is the data type of the values in the map. *Comp* is a function that compares two keys. *Comp* defaults to the *less* function object. *Allocator* is the memory allocator (this defaults to allocator) .

The following map constructor constructs an empty map. The function *cmpfn*, if present, determines the ordering of the map.

```
explicit map(class Tkey, class Tvalue, const Comp &cmpfn = Comp(),const
Allocator &a = Allocator());
```

The second constructor instantiates a map that contains the same elements as the map *obj*.

```
map(const map<Key, T, Comp, Allocator> &obj);
```

The third form constructs a map that contains the elements in the range specified by an iterator.

```
template <class InIter> map(InIter start, InIter end),
   const Comp &cmpfn = Comp(), const Allocator &a = Allocator());
```

If a class type is used as a key then the class must define a default constructor and overload any necessary comparison operators.

Key/value pairs are stored in a map as objects of type **pair** (*see subsequent chapter stdlib::pair*).

The STL defines the following comparison operators for a map:

==, <, <=, !=, >, >=

Some member functions defined for map containers are*:*

`iterator begin();`	Returns an iterator to the first element in a map.
`iterator end();`	Returns an iterator to the end of a map.
`size_type size() const;`	Returns the number of elements in a map.
`size_type count(const key_type &k) const;`	Returns the number of times a key k occurs in a map (must be one or zero).

`void clear();`	Removes all the elements in a map.
`bool empty() const;`	Returns true if the map is empty, false otherwise.
`void erase(iterator i);`	Removes the element pointed to by i.

Inserting Elements Into A Map

The function declaration for insert is:

```
      iterator map_name.insert({key, element})
```

Note the parentheses which are required since we are inserting a pair object. This member function returns an iterator to the inserted element.

Here is an example:

```
#include <map>
#include <iostream>
#include <string>

using namespace std;

int main() {

  // initialize an int type map container
  map<int, int> mp;

  // insert some elements
  mp.insert({ 2, 30 });
  mp.insert({1, 10});

  // declare an iterator that will iterate over a map<int,int>
  map<int, int>::iterator p;


  // initialize the iterator
  p = mp.begin();

  // iterate over the map getting keys and values

  for (; p != mp.end(); ++p) {

   cout << "key: " << p->first << "\n";
   cout << "value: " << p->second << "\n";
  }

}
```

Getting A Map Key And A Key Value

Given a map iterator *p*, p points to a pair object {key, value}. *p->first* points to the key and *p->second* points to the key value.

The following program creates a telephone book map, mapping names to telephone numbers.

A Map Container Example

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main()
{
        map<string, string> tel;

        // insert elements into the map
         tel.insert({"alice smith", "555-555-5555"});
         tel.insert({"john brown", "656-890-9080"});

         map<string, string>::iterator iter;

         // given a name, find the tel number
         iter = tel.find("alice smith");

         if(iter != tel.end()) {
            cout << "name is " << iter->first << endl;
            cout << "telephone number is: " << iter->second << endl;
        }
```

```
            else
               cout << "Name is not in map.\n";

         return 0;
      }

      Output:
      name is alice smith
      telephone number is: 555-555-5555
```

Note: for a map iterator *iter*, **iter* points to the pair object *{ iter->first, iter->second }*.

Algorithms

Algorithms act upon containers. Although each container provides support for its own basic operations, the standard algorithms implement complex actions. They also allow us to work with two different types of containers simultaneously. To use STL algorithms, we must include the header file **algorithm** in the program.

All of the algorithms are function templates. They can be applied to any type of container.

There are a large number of STL algorithms, a few interesting ones are:

binary_search	Performs a binary search on an ordered sequence.
count_if	Returns the number of elements that satisfy a predicate.
find	Searches a range for a value and returns an iterator to the first occurrence of the element.
find_if	Searches a range for an element for which a user-defined unary predicate returns true.
make_heap	Constructs a heap from a sequence.

random_shuffle	Randomizes a sequence.
sort	Sorts the elements of a container.
unique, unique_copy	Eliminates duplicate elements from a container.

The *count* algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* that match *val*. The *count_if* algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* for which a unary predicate *pfn* returns true.

The following program demonstrates the use of the *count* algorithm:

```
// Demonstrate count() algorithm
#include <iostream>
#include <cstdlib>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<bool> v;
    int i;

    // populate the vector with random bools
    for (i = 0; i < 10; i++) {
         if (rand() % 2) v.push_back(true);
         else v.push_back(false);
}

cout << "Sequence:\n";
for (i = 0; i < v.size(); i++)
       cout  << v[i] << " ";
cout << endl;

i = count(v.begin(), v.end(), true);
cout << i << " elements are true.\n";
return 0;
}


Output:
Sequence:
```

```
1 0 1 1 1 1 0 0 1 1
7 elements are true.
```

C++: Functors

A function object or functor is simply an object or class that implements the *parentheses operator ()*. Because of the overloaded () operator, functors look and behave like functions and are callable.

For example, consider the *Foo* class declaration below which makes *Foo* into a functor:

```
#include <iostream>

class Foo
{

public:
   int operator()(int a, int b) { return a + b;}
};

int main()
{
  int sum = 0;
  Foo foo;

  // We can also do: sum = foo(11, 88);
  sum = foo.operator()(11, 88);
  std::cout << sum << std::endl;

  sum = foo(5, 7);
  std::cout << sum << std::endl;
}
// output
99
12
```

Note that *foo.operator()(11, 88)* can be written in the shorthand form: *foo(11, 88)*.

Since Functors are just objects, they can be stored, copy constructed, moved and assigned.

C++ Strings

C++ does not have a built-in native string type. It does, however, have two ways to handle strings. First, we can use the C-style null-terminated character array. This is referred to as a C string. The second way is as a class object of type *string*.

The string class is a specialization of a more general class template called *basic_string*. *basic_string* has are two specializations: *string*, which supports 8-bit character strings, and *wstring*, which supports 16-bit wide-character strings.

String Constructors

The most commonly used string constructors are:

```
string();                                 // empty string
string(const char *ptrStr);               // string from a C string
string(const string& str);                // copy constructor
string(int num, const char c)
```

The first constructor creates an empty string object. The second creates a string object from a null-terminated const string pointed to by *ptrStr*. The third constructor is the copy constructor. It creates a new string from a string object reference. The fourth constructor creates a new string where the character '*c*' is repeated *num* times.

String Operators

std::string implements the following operators:

=	assignment
+	concatenation
==	equality
!=	inequality

<	Less than
<=	Less than or equal to
>, >=	Greater than and greater than or equal to
[]	Index operator
<<	Output
>>	Input

These operators eliminate the need for calls to functions such as *strcpy* or *strcat*.

String Concatenation

The + operator can be used to concatenate a string object with another string object or with a C-style string.

```
// A string demonstration

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

int main()
{
  string str1("Alpha");
  string str2("Beta");
  string str3("Omega");
  string str4;

  // assign a string
  str4 = str1;
  cout << str1 << "\n" << str4 << "\n";

  // concatenate two strings
  str4 = str1 + str2;
  cout << str4 << "\n";

  // concatenate a string with a C-string
  str4 = str1 + " to " + str3;
  cout << str4 << "\n";

  // compare strings
  if(str3 > str1) cout << "str3 > str1\n";

  if(str3 == str1 + str2)
    cout << "str3 == str1+str2\n";
  // Assign a C-style string to a string object
  str1 = "This is a null-terminated string: ";

  // copy constructed string object
  string str5(str1);
  cout << str5 << endl;

  // input and output a string
  cout << "Enter a string: ";
  cin >> str5;
  cout << str5 << endl;

}
```

A string object is automatically sized to hold the string that it encapsulates. Thus, when assigning or concatenating strings, the target string will grow or shrink as needed to accommodate the size of the new string. It is not possible to have a buffer overflow for a string object.

Convert A String Object To A C-Style String

The member function *c_str()* converts a string object to a null terminated const C-style string:

```
#include <iostream>
#include <cstring>
#include <string>

int main ()
{
  std::string str("hello world");

  const char* cstr = str.c_str();

  // cstr now contains a c-style string copy of str; including
  // the terminating null char

}
```

STL Algorithms For Strings

The string class meets all of the basic requirements required for a container. Thus, it supports most common container functions, such as *begin*, *end*, *size* etc. It also supports iterators. A string object can also be manipulated by the STL algorithms.

Modern C++: Type_Traits (C++11)

Type traits are a template meta-programming technique that lets us determine the type properties of variables at compile-time. Since the type information is obtained at compile-time there is zero run-time cost.

The C++ standard library implements a large number of predefined type traits that are defined in the <*type_traits*> header file.

A type trait is a struct that contains a single constant static data member. This member is named *value* if it answers a question about a type. The member is named *type* if it transforms a type.

Consider the following type trait declaration:

```
#include <type_traits>

template<typename T>
struct is_floating_point;
```

This trait asks the question whether type T is a floating point type or not. The struct answers this question. The struct *is_floating_point* contains a single static constant variable called *value*. *value* is either true or false.

Suppose that we want to determine if a given type is a floating point type. The following code sample shows how this is done:

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

class Foo {};

int main()
{
    cout << is_floating_point<Foo>::value << '\n';
    cout << is_floating_point<float>::value << '\n';
    cout << is_floating_point<int>::value << '\n';
}

#output
0
1
0
```

In order to evaluate the previous expressions, the compiler generates the following explicit structs and static constant values at compile-time:

```
struct is_floating_point_Foo {
    static const bool value = false;
};

struct is_floating_point_float {
    static const bool value = true;
};

struct is_floating_point_int {
    static const bool value = false;
};
```

Note: const static data members of a class or struct can be initialized by assignment as indicated above.

Since the struct data member *value* is a static constant, we can access it through the scope operator as follows:

```
std::is_floating_point<Foo>::value
```

Modern C++: Automatic Type Inference (C++11)

Prior to C++11, the type of each program variable had to be explicitly specified in the variable’s declaration. Since C++11 we can ask the compiler to deduce the type of a variable for us. C++11 has two new keywords for automatic type deduction: *auto* and *decltype*.

In C++11, we can declare a variable without specifying its type by setting the type of the variable to **auto**.

auto i = 42; // i has type *int*

The *auto* type declaration, instructs the compiler to deduce the type of *i* at compile time. Because of type inference, we can replace a statement such as *int i = 42,* with the statement *auto i = 42;*

Here is a more complex example where *auto* is used to simplify a type declaration:

```
std::vector<int> vec;
std::vector<int>::iterator itr;
itr = vec.first();
```

Instead of this we can use the simpler declaration:

```
std::vector<int> vec;
auto itr = vec.first();
```

The compiler will automatically deduce *itr* to have type: *std::vector<int>::iterator.*

Take a look at the following function template:

```
template <typename BuiltType, typename Builder>
      builtType
      createObject (const Builder& builder) {
```

```
   return builder.makeObject();
 }
```

This function template has two template parameters: one for the type of the Builder object, and a second for the type of the object that is built. Since the type of the built object cannot be deduced from the template parameter *builder*, we have to do an explicit specialization for each type of builder and the body of each such function will specify the *BuiltType* that is returned by the function *makeObject*. This makes the template definition virtually useless.

But with *auto*, we can automatically deduce the return type of *builder.makeObject()*. This permits us to simplify the function template definition:

```
template <typename Builder>
      auto
      createObject (const Builder& builder)
      {
         return builder.makeObject();

      }
```

Now we only need a single template parameter, and the *BuiltType* is inferred by the compiler when the function *builder.makeObject()* is called.

Modern C++: Type Deduction With decltype (C++11)

The *decltype* function returns a type. *decltype* has a variable or an expression as it's only parameter:

```
int x = 3;
decltype(x) y;        // same as int y;
```

In this example, *decltype(x)* returns type *int.*

We can use *decltype* with almost any expression including a function declaration. In particular, we can use *decltype* to declare the return type of a function or method.

For example, here is a use of *decltype* in a function template:

```
template <typename Builder>
auto
createObject(const Builder& builder) -> decltype(builder.makeObject())
{
    auto val = builder.makeObject();

    return val;
}
```

We will discuss the arrow notation shortly.

Type Attributes With auto And decltype

Look at the following example;

```
int &foo();                 // function returns a reference

auto bar = foo();
```

The function *foo* returns a reference to an *int*. Is *bar* an *int* type or a reference to an *int type?* The answer is an *int* type, since *auto* does not preserve attributes such as *const*, *volatile* or *reference*. (If we want to return a reference with *auto* use *auto&*)

The `decltype(...)` type-specifier deduces a type just like `auto` does. However, *decltype* deduces return types while preserving attributes such as *reference*, *const* and *volatile*.

Modern C++: Trailing Return Types (C++11)

C++11 introduces a new syntax for the return type of a function. This syntax is called a trailing return type.

The syntax for trailing return types is:

```
auto getSum() -> float;
```

The declaration *auto getSum() -> **float*** is equivalent to:

```
float getSum();
```

When we use a trailing return type, the function declaration must start with *auto*. The type of the return value is deduced from the trailing return type and not auto.

One use of trailing return types occurs in function templates where the return type of the function depends on the types of the arguments:

```
#include <iostream>

template<class T, class U>
    auto multiply(T lhs, U rhs) -> decltype(lhs * rhs)
    {
       return lhs * rhs;
    }

int main()
{
   auto value =  multiply(2.3, 6);
   std::cout << value << std::endl;
}

// output
13.8
```

The following declaration will give a compile error:

```
#include <iostream>

template<class T, class U>
    decltype(lhs * rhs) multiply(T lhs, U rhs)
    {
       return lhs * rhs;
    }

int main()
{
   auto value =  multiply(2.3, 6);
   std::cout << value << std::endl;
}

output:
error: 'lhs' was not declared in this scope
```

The reason is that the compiler parses statements from the left to the right. Thus when it encounters the expression *(lhs * rhs)* it does not know what *lhs* and *rhs* are. The trailing return type solves this problem.

Simplification Of Code

Using trailing return types can simplify code. Consider the following function pointer declaration:

```
void (*func(int i))(int) newFunc;
```

This declares *newFunc* as a function pointer taking an *int* parameter and returning a pointer to a function taking an *int* parameter and returning *void*. This can be declared much more clearly as:

```
auto newFunc(int x) -> void (*)(int);
```

Modern C++: Variable Initialization (C++11)

Variable initialization involves declaring a variable and giving it a value. C++11 introduces a standard syntax for variable initialization.

Traditionally in C++ initialization can happen with parentheses, braces, or assignment operators. This has caused confusion about how to initialize a variable or an object. C++11 introduces uniform initialization, which means that we can use a standard syntax for initialization.

Copy Assignment

Copy assignment is the traditional way to put a value into a variable. After a variable has been declared, we can give it a value by using the assignment = *operator*. This type of initialization is called copy assignment (or just assignment) for short.

```
int width;          // declare an integer variable named width
width = 5;          // copy assignment of value 5 into the variable
```

Copy assignment is named as such because it copies the value on the right-hand side of the = *operator* to the variable on the left-hand side of the operator. The = *operator* is called the assignment operator.

The problem with copy assignment is that it uses two statements (a declaration and an assignment) and thus consumes more CPU cycles.

We can combine the declaration and initialization of a variable into one step in order to conserve a few CPU cycles:

```
int width = 5;
```

Direct Initialization

C++11 introduces direct initialization:

```
// direct initialization of the variable width with the value 5
int width(5);       // equivalent to int width (5);
```

For native data types (like integers), copy assignment and direct initialization are essentially the same. But for some complex types, direct initialization can perform better than copy initialization.

Next consider:

```
Foo foo(obj)
```

For this direct initialization of the object *foo* to work, *Foo* must have a copy constructor or a constructor that accepts *obj* as a call by value. This statement will not compile otherwise.

Brace Initialization

Brace initialization comes in two forms:

// **direct brace initialization** of the variable width with 5 (preferred technique)
int width { 5 };
// **copy brace initialization** of the variable with 5
int width = { 5 };

Initializing a variable or an object with empty braces indicates zero initialization. **Zero initialization** initializes a variable to a symbolic zero (a zero that is appropriate for the given type). For objects, zero initialization requires a constructor that takes no arguments.

```
// zero initialization of the string to the empty string
std::string msg {};
```

Brace initialization has the added benefit of disallowing type conversions that are narrowing. This means that if we try to use brace initialization to initialize a variable with a value that it cannot safely hold, the compiler will throw a warning or an error.

Floating-point to integer conversions are always narrowing — even 5.0 to 5.

```
// error: narrowing initialization: trying to convert a float type
// type to an int type

int width  { 4.0 };
```

The best practise is to always initialize variables. But sometimes, in order to save CPU cycles, we can leave variables uninitialized.

The use of brace initialization forces value initialization. A variable that is initialized with {} is set to the zero value for the type of the variable (for example, *nullptr* if it is a pointer):

```
string s;                  // s has an undefined value
string s {}                // s is initialized to the empty string

int j;                     // j has an undefined value
int j {};                  // j is initialized by 0

int* p;                    // p has undefined value
int* p {};                 // p is initialized to nullptr
```

The Meaning Of return {}

Consider the function declaration:

```
Foo myFunction() { return {}; };
```

myFunction will return a default object of type *Foo*. For this statement to compile properly, *Foo* must have a default constructor that takes no arguments.

std::initializer_list

Also See: Standard Library: *initializer_list*

Consider an array of integers in C++:

```
int arry[5];
```

We can initialize this array with an initializer list:

```
int arry[5]  {0, 1, 2, 3, 4};
```

The expression in braces is called an initializer list.

The following initialization of a dynamically allocated array will also compile:

```
int *pArray = new int[5] { 5, 4, 3, 2, 1 };
```

Now consider the following array class :

```
#include <iostream>
#include <initializer_list>

class IntArray
{
    private:
    int m_length;
    int *m_data;

    public:
    IntArray(): m_length(0), m_data(nullptr)  { }

    IntArray(int length): m_length(length)
    {
        m_data = new int[length];
    }

    ~IntArray()
    {
       delete[] m_data;
    }

    int& operator[](int index)
    {
      return m_data[index];
    }

    int getLength() { return m_length; }
};

int main()
{
   IntArray array { 5, 4, 3, 2, 1 };
}

// output
error: no matching function for call to 'IntArray::IntArray(<brace-
enclosed initializer list>)'
```

In this example the brace initialization *IntArray array { 5, 4, 3, 2, 1 }* will not compile because the *IntArray* class does not have a constructor that can handle initializer lists.

In C++11, when a compiler sees a braced initializer list, it implicitly converts it into an object of type *std::initializer_list<T>*. All STL containers have a constructor that takes an initializer list as a parameter. This enables vectors and other containers to support brace initialization.

Prior to C++11, the only way to initialize a vector would have been as follows:

```
std::vector<int> vec1;
for(int i = 0; i < 5; i++)
      vec1.push_back(i);
```

However since C++11, *std::vector* contains a constructor for initializer lists, so we can initialize a vector as follows:

```
 // Compile Error before C++ 11
std::vector<int> vec = {1,2,3,4,5};
or:
std::vector<int> vec {1,2,3,4,5};
```

Whenever the compiler sees a braced initializer list such as {1, 5, 6, 9, 4}, it implicitly creates an object of type *std::initializer_list<T>* from it. T is the data type of the list.

We can create a light-weight object of type *std::initializer_list<T>* as follows:

```
#include <initializer_list>
std::initializer_list<int> data = {1,2,3,4,5};
```

std::initializer_list<T> is an array-like container. It *has* the following member functions:

size():	returns the size of the list
begin():	returns an iterator to the beginning of the list
end():	returns an iterator to the end of the list

Modern C++: explicit Keyword (C++11)

The C++ compiler will implicitly perform type conversions when required. For example:

```
int x = 3;
float y = x + 5.5.;
```

x will be automatically cast to a float data type before the addition is performed. Implicit type conversions typically involve type widening.

We can prevent the compiler from implicitly converting a type by using the *explicit* keyword. *explicit* is frequently used to qualify constructors. Consider the following example:

```
class Array {
  public:
    explicit Array(size_t x);

};
```

This will prevent the implicit type conversion of the parameter x in the constructor.

Modern C++: Range-Based For Loops (C++11)

C++11 introduces a new type of for loop that iterates over all of the elements of a range. A range is any collection that has an iterator. The syntax for range-based loops is:

```
for (type item : collection) {
        statements
}
```

item is an element that is received from the collection. *item* must be declared with a type. For each element in the collection, the statements in the body of the loop are executed:

```
for (int item : { 2, 3, 5, 7, 9, 13, 17, 19 }) {
            std::cout << item << std::endl;
}
```

Notice that in this example, we are iterating over {2, 3, 5,..}, which is a *std::initializer_list* object.

Suppose that we want to multiply each element of a double vector *vec* by 3, we can program this as follows:

```
std::vector<double> vec { 1.4, 5.34, 7.9 };
...
for (auto& element : vec) {
     element *= 3;
}
```

Here, declaring *element* as a reference is important because otherwise the statements in the body of the for loop will act upon a local copy of the elements in *vec*.

The following example, uses references and type inference:

```
#include <iostream>

int main()
{
  int x[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  for (const auto &y : x)
  {
    std::cout << y << " ";
  }
  std::cout << std::endl;
}

// output
1 2 3 4 5 6 7 8 9 10
```

We can break out of a range based loop with a *break*, *return* or a *goto* statement. *continue* will terminate the current iteration of the loop and start a new iteration.

Modern C++: Lambda Functions (C++11)

Lambdas were introduced in C++11. A lambda is an unnamed or anonymous function. An anonymous function does not have a function name.

Lambdas can be assigned to variables, stored in variables or used as function parameters. If we want to reuse a lambda, we can save it in a variable.

Lambdas are very useful for creating short inline functions.

A minimal lambda function has no parameters and simply does something. Here is an example of such a lambda:

```
[] {
         std::cout << "hello world" << std::endl;
   };
```

We can call a lambda directly, as follows:

```
[] {
         std::cout << "hello world" << std::endl;
   } ();
```

We can also assign a lambda to a variable and then call the lambda through this variable:

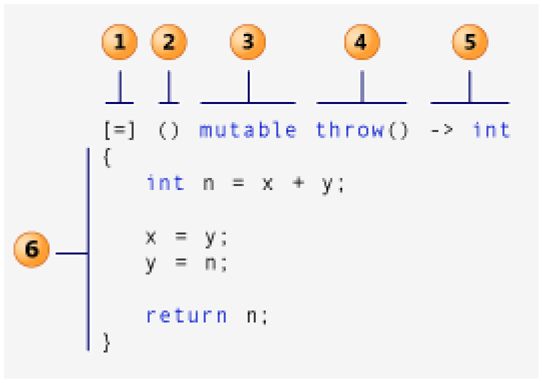
```
#include <iostream>

int main()
{
  auto my_var = [] {
                     std::cout << "hello world" << std::endl;
                   };
  my_var();
}

// output
```

```
hello world
```

A lambda always has a **lambda introducer.** An introducer is represented by the square brackets [] in a lambda expression. Introducers permit us to access variables in the outer scope of the lambda. If there is no need to access variables in the outer scope, the brackets are just empty.

The lambda body is the block of code enclosed by parentheses.

Lambda Specification



```
[=] () mutable throw() -> int
{
    int n = x + y;

    x = y;
    y = n;

    return n;
}
```

Where:

1. lambda introducer: mandatory (also known as a capture clause)
2. parameter list: optional (also known as a lambda declarator)
3. mutable specification: optional attribute.
4. exception specification: optional.
5. trailing return type: optional.
6. the lambda body.

Lambda Parameters

Between the *lambda introducer* and the lambda body, we can specify the parameters that the lambda will receive enclosed in round brackets. Next, we can optionally specify lambda attributes, such as an exception specification and finally an optional trailing return type.

Lambda parameters are enclosed in round brackets. Parameters are optional, but if they occur, then the round brackets surrounding the parameters are mandatory.

Thus, the syntax of a lambda is:

```
[lambda introducer] (optional parameters) optional attributes optional
trailing return type {    };
```

In the following example, a lambda receives a reference to a const string:

```
      auto my_var = [] (const std::string &s) {
                  std::cout << s << std::endl;
                  };

      my_var("hello world");                  // prints ‘‘hello world’’
```

A lambda can also return a value. The return type will be deduced if the lambda does not explicitly declare a trailing return type.

```
      auto x = [] {
              return 42;
          }
```

In this example, the return type is deduced to be type *int*.

We can explicitly specify a return type for a lambda by using the trailing return type syntax. The syntax for this is as follows:

```
[] () -> double { return 42; };
```

Parentheses for lambda parameters are mandatory when the return type is explicitly specified with a trailing return type.

Access To Variables In The Outer Scope

Inside the lambda *introducer* (the brackets at the beginning of a lambda), we can specify a capture to access variables in the outer scope.

- **[=]** means that the variables in the outer scope are passed to the lambda by value. Thus, we can read the variables in the outer scope that existed when the lambda was defined. We cannot modify the variables in the outer scope.
- **[&]** means that the variables in the outer scope are passed to the lambda by reference. Thus, we can modify the variables in the outer scope that existed when the lambda was defined.

Variables in the outer scope, are accessed through the lambda introducer. If we pass these variables by value (*[=]*) then we can read these variables in the lambda body but we cannot modify their values in the outer scope (because its a call by value). Variables passed by value in the introducer are treated as read-only.

If we pass these variables by reference (*[&]*) then we can modify the variables that are in the outer scope of the lambda.

Here is an example:

```
#include <iostream>

int main()
```

```
{
  int x = 0;
  int y = 42;

  std::cout << "x value before lambda: " << x << std::endl;
  std::cout << "y value before lambda: " << y << std::endl;

  // x is passed by value. y is passed by reference
  auto lex = [x, &y] {
           y++;
           std::cout << "x value in lambda: " << x << std::endl;
           std::cout << "y value in lambda: " << y << std::endl;
         };

  lex();

  std::cout << "x value after lambda: " << x << std::endl;
  std::cout << "y value after lambda: " << y << std::endl;

}

// output
x value before lambda: 0
y value before lambda: 42
x value in lambda: 0
y value in lambda: 43
x value after lambda: 0
y value after lambda: 43
```

In this example, x is passed by value, therefore we cannot modify the outer scope x inside the lambda. Changing x inside the lambda will give a compilation error. However, y is passed by reference, so it can be modified inside the lambda.

```
#include <iostream>

int main()
{
    int x = 0;
    auto test = [=] {
     x++;
     };

     test();

    std::cout << x << std::endl;
}
```

```
// result
```

```
error: increment of read-only variable 'x'
```

Instead of *[x, &y]*, we can also specify *[=, &y]*, this will pass y by reference and all other variables in the outer scope by value.

Mutable Lambdas

We can declare a lambda as mutable. In this case, variables that are passed by value can be changed inside the lambda. Changing the value of the variable inside the lambda will change it's value in the outer scope.

```
#include <iostream>

int main()
{
  int id = 0;

  auto f = [id] () mutable {
               id++;
               std::cout << "id: " << id << std::endl;
             };

  id = 42;
  f();
  f();
  f();
  std::cout << "id: " << id << std::endl;
}

// output
id: 1
id: 2
id: 3
id: 42
```

Notice that the value received by a lambda is the value of the variable at the time that the lambda is declared.

The Type Of A Lambda

The type of a lambda is an anonymous function object (or functor) that is unique for each lambda expression.

Generic Lambda Expressions

Consider the lambda expression:

```
[](int x, int y) -> int { return(x + y); };
```

This lambda function adds two integers. If we want to add two floating point numbers, then we need to specify a new lambda:

```
[](float x, float y) -> float { return(x  + y); }
```

And so forth for each additional type. In C++14, we can simply specify the lambda parameters as auto and the compiler will do compile-time introspection to create a lambda with the correct types. The compiler will also deduce the return type of a lambda expression:

```
[](auto x, auto y) -> auto { return(x+y); }
```

Here is an example:

```
#include <iostream>

int main()
{
  auto sum = [](auto a, auto b) {  return (a + b); };

  std::cout << sum(1, 6) << std::endl;
  std::cout << sum(1.2, 5.6) << std::endl;
}

// output
7
```

```
6.8
```

Generalized Lambda Introducer

C++14 also introduces a generalized lambda introducer or capture clause. In C++14, we can declare and initialize new variables in the introducer clause, without the need for these variables existing in the lambda function’s outer scope. An expression can be used to initialize a variable in the introducer. The type of the new variable is deduced from the type produced by the expression.

Here is an example:

```
#include <iostream>

int main()
{
// the introducer declares and initializes a new variable val

auto adder = [val = 916]
                 {
                   std::cout << val << std::endl;
                 };

adder();
}

// output
916
```

Modern C++: Constexpr (C++11)

C++11 introduces the keyword *constexpr*. This is an acronym for *constant expression*. The *constexpr* keyword directs the compiler to evaluate a variable or object at compile-time. In the following example, a variable is declared as a *constexpr*:

```
constexpr int val = 10;
```

val is evaluated at compile-time and has a value of 10. This is different from the statement: *int val2 = 10*; which is only evaluated at run-time even though *val* may never change it’s value. Furthermore *val* is a read-only variable. The following program will not compile since *val* cannot be incremented:

```
#include <iostream>

int main(void) {

    constexpr int i = 10;
    ++i;
    std::cout << i;
}

output:
error: increment of read-only variable 'i'
```

The basic idea of *constexpr* is to allow computations to take place while the code compiles, rather than when the program itself is run. A compile-time computation replaces a run-time computation. If something can be done at compile time, it will be done once, rather than while the program is running.

A *constexpr* variable is valid only if it can be evaluated at compile-time.

All *constexpr* expressions are *const* but there are two main differences between *const* and *constexpr* variables. Firstly, a *const* variable can be initialized at run-time, for example: *const int x = constantSum()*. A *constexpr* variable is always initialized at compile-time. Secondly, unlike *const*, *constexpr* can be used as the return value of a function.

In order for an expression to be a *constexpr*, we must be able to evaluate it at compile-time. Look at this example:

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

int main()
{

    int y = -10;
    constexpr int x = abs(y);
    cout << x << endl;
}
```

This expression will not compile since even though y is constant, y will be evaluated at run-time. However the program will compile without error if we specify: *constexpr int y = -10* or *const int y = -10.*

In order to induce the compiler to perform compile time processing, we must use the *constexpr* keyword with the expression or function that we want to compute at compile-time.

constexpr Functions

The C++ compiler can evaluate a *constexpr* function at compile-time or run-time depending on it’s arguments:

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

constexpr int square(int n) {
  return n * n;
}

int main() {
```

```
// executes at compile-time; value n is set at compile-time
const int n = 100;
constexpr int prod1 = square(n);

cout << prod1 << endl;
static_assert(prod1 == 10000);  //succeeds

// executes at run-time; y is initialized at run-time
int y = 10;
square(y);

//static_assert(y == 100);    // fails
cout << y << endl;          // OK

}
```

When the arguments of a *constexpr* function are *constexpr* values, the *constexpr* function produces a compile-time constant. When called with *non-constexpr* arguments the function produces a value at run time like a regular function. This dual behaviour saves us from having to write *constexpr* and *non-constexpr* versions of the same function.

Restrictions On Constexpr Functions

constexpr functions that have *constexpr* arguments and only call other *constexpr* functions will be evaluated at compile-time. *constexpr* functions cannot be virtual or allocate and de-allocate memory.

Arrays And constexpr Functions

Prior to C++11, the declaration of an array could not contain the return value of a function. With C++11, this is now possible, providing that the function is a *constexpr* function that can be evaluated at compile-time:

```
#include <iostream>

constexpr int getDefaultArraySize (int base_size)
{
    return 2 * base_size;
}
```

```
int main()
{
  constexpr int x = 3;
  int foo_array[getDefaultArraySize( x )];

  int asize = getDefaultArraySize(3);

  for (int ctr = 0; ctr < asize; ctr++)
  {
    foo_array[ctr] = ctr;
  }

  for (int ctr = 0; ctr < asize; ctr++)
  {
    std::cout << foo_array[ctr] << " ";
  }

  //static_assert(foo_array[0] == 0); fails
  std::cout << std::endl;
}

// output
0 1 2 3 4 5
```

In this example, the array is instantiated at compile-time but initialized at run-time.

Evaluating constexpr Objects at Compile-time

Suppose that we have a circle class:

```
class Circle
{
   public:
   Circle (int x, int y, int radius) : _x(x), _y(y), _radius(radius) {}
   double getArea () const
   {
      return _radius * _radius * 3.1415926;
   }
   private:
      int _x;
      int _y;
      int _radius;
};
```

Next suppose that we want to construct a circle object at compile time and get its area.

We can do this with a few small modifications to the Circle class. First, we must declare the constructor as a *constexpr* function, and second, we must declare the *getArea* function as a *constexpr* function.

Declaring the constructor as a *constexpr* function allows the Circle object to be instantiated at compile time if the constructor arguments are *constexpr*. Declaring the method *getArea* as a *constexpr* function allows it to be called at compile time:

```
#include <iostream>

class Circle
{
public:
  constexpr Circle (int x, int y, int radius) : _x( x ), _y( y ),
     _radius( radius ) {}

  constexpr double getArea ()
  {
     return _radius * _radius * 3.1415926;
  }

private:
  int _x;
  int _y;
  int _radius;
};

int main()
{
  Circle c(0, 0, 10);          // constructor arguments are constexpr
  double area = c.getArea();

  std::cout << "area of circle: " << area << std::endl;

}

// output
area of circle: 314.159
```

Modern C++: lvalue And rvalue References (C++11)

Understanding lvalues and rvalues

In C++, a *lvalue* (*left value*) is a variable or constant that has a specific addressable memory location. In contrast, a *rvalue (right value)* doesn't have an addressable memory location. In general, rvalues are ephemeral and may exist only for the duration of one statement. In contrast lvalues live a longer life since they exist as variables or constants with addressable memory. Consider this example:

```
int x = 966;
```

x is a lvalue since it has an addressable location in memory (*&x*). However, 966 is a rvalue since it does not have an addressable memory location. For example, 966 may be in some CPU register or unknown memory location for an indeterminate period of time. For the previous statement, the rvalue exists only for the duration of the statement. As soon as the statement is executed, the rvalue ceases to exist. C++ requires the left-hand side of an assignment statement to be a lvalue.

Consider:

```
int square(const int x) {
   return x**2;
}

int main() {
    auto y = square(10);
}
```

In the assignment statement, the value returned by the function on the right hand side is a rvalue. This value exists on the stack and will disappear as soon as the statement is executed. In order to use the rvalue, it must be stored in the variable *y*, which is a lvalue.

Functions Returning lvalues And rvalues

Take a look at the following code:

```
#include <iostream>

int global = 100;

int& setGlobal()
{
    return global;
}

int main()
{
  setGlobal() = 400;                 // OK
  std::cout << global << std::endl;
}

// output
400
```

The *setGlobal* returns a reference and so it is implicitly a pointer and thus must point to an addressable location in memory. The return value is a lvalue because it has an addressable memory location (*&global*). With the assignment of 400, we are changing the value of the reference variable.

In the following example:

```
int x = 1;
int y = 3;
int z = x + y;
```

The intermediate result *x + y* is a rvalue.

lvalue References

Consider the assignment:

```
int& yref = 10;  // will this compile?
```

Since *yref* is a reference, it is implicitly a pointer and thus must point to an addressable location in memory. The assignment fails because 10 is a rvalue, it does not have an addressable memory address (we cannot obtain a pointer to 10). We cannot bind a lvalue reference to a rvalue, because the latter does not have an addressable memory location.

However the C++ specification lets us bind a const lvalue reference to a rvalue, so the following is allowed:

```
#include <iostream>

int main()
{
  const int& ref = 10;                // OK

  std::cout << ref << std::endl;
}

// output
10
```

The idea behind this is quite straightforward. The literal constant `10` is a volatile rvalue that will expire in no time, so a reference to it is just meaningless. But if we make *ref* a reference to a *const int* type, then the value *ref* points to can't be modified. That is, the rvalue 10 cannot be modified. The compiler places 10 in the Data Segment and binds *ref* to this location.

So we can do this:

```
#include <iostream>
```

```
int square(const int &x) {
   return x*2;
}

int main()
{
  // since the parameter in the square function is a reference
  // to a const integer type, it can bind to the constant
  // argument 10

  int ret = square(10);

  std::cout << ret << std::endl;
}

// output
20
```

But the following will fail because 10 is a rvalue and x cannot be a reference to it:

```
#include <iostream>

int square(int& x) {
   return x**2;
}

int main() {

   int ret = square(10);

   std::cout << ret << std::endl;
}

// output
error: cannot bind non-const lvalue reference of type 'int&' to an
rvalue of type 'int'
```

Modern C++: Move Semantics (C++11)

By default, copying an object in C++ involves doing a bit by bit copy of the object. This is called a deep copy. We can avoid making deep copies of objects by using references or copy constructors. In addition to this, another technique is available. Move semantics allows us to efficiently transfer resources from one object to another, without deep copying or using references or copy constructors.

lvalues And rvalues

Every expression is either a lvalue (*left value*) or a rvalue (*right value*). A lvalue is a persistent value with an addressable location in memory. We can store values at this address on an ongoing basis. A rvalue is only stored transiently. A rvalue does not have an addressable memory location.

Historically, lvalues and rvalues are so called because a lvalue typically appears on the left-side of an assignment, whereas a rvalue appears only on the right side. If an expression is not a lvalue, it is a rvalue.

Consider the following example:

```
int result = std::abs(a*b)
```

The intermediate result a*b is a rvalue. Its is a transient value that is not addressable. The return value of the function *std::abs* is also a rvalue. It is a transient value that exists only until the assignment is completed. rvalues exist in memory for a transient period of time and cannot be addressed.

Note that the following assignment will generate a compiler error since a rvalue does not have an addressable location:

```
long* ptr  = &std::abs(a*b);
```

Similarly, we cannot take the address of a literal constant:

```
int *ptr = &5;                // will not compile
```

If we can obtain the address of an expression, then it is a lvalue.

References

As we have seen previously, a reference is an alias for a variable:

```
int square(int&);                // forward reference

int x = 5;
int y = square(x);
```

The function *square()* receives a reference to a variable. This is implicitly a pointer. However the reference variable can be manipulated just like a non-pointer variable. This type of reference is a **lvalue reference**.

A **rvalue reference** is an alias for a variable, just like a lvalue reference, but it differs from a lvalue reference in that it references a rvalue, even though this value is ephemeral. By binding a rvalue reference to a transient rvalue we extend the lifetime of this transient value. It's memory will not be discarded as long as the rvalue reference remains in scope.

We define a rvalue reference type by using two ampersands followed by the variable name. Here is an example:

```
int  &&rref =  square(5*2);     // or int&& rtemp = square(5*2);
```

In this example, the result of *square(5*2)* is a rvalue. *rref* is defined as a reference to a rvalue and it’s value is the rvalue on the right hand side.

Here is another example:

```
#include <iostream>

int main()
{
  std::string s1 = "hello ";
  std::string s2 = "world";

  std::string&& rref = s1 + s2;

  rref += ", Foo";
  std::cout << rref << std::endl;
}

// output
hello world, Foo
```

rref is a rvalue reference to a rvalue of type *std::string*. The value on the right side of the *rref* assignment is a rvalue. Because we are binding a rvalue reference to it, this value is not ephemeral. In the fourth line (*rref += ", Foo";*) we are changing the value of the rvalue reference.

Moving Objects

Move semantics is a new way of moving resources by avoiding unnecessary copies of temporary objects. This can improve performance substantially. Consider this example:

```
class Container
{
public:

  Container(int size)         // Constructor
  {
    m_data = new int[size];
    m_size = size;
  }
```

```
    ~Container()                // Destructor
    {
      delete[] m_data;
    }

  private:
    int*   m_data;
    size_t m_size;
  };
```

This is a very simple class. It's a container class that holds an array.

A copy constructor creates a copy of an existing object. The following example shows when the copy constructor of the *Container* class is invoked:

```
Container h1(10000);     // regular constructor
Container h2(h1);        // copy constructor of h2 invoked
```

And here is the copy constructor for the *Container* class:

```
Container(const Holder& other)
{
  m_data = new int[other.m_size];
  std::copy(other.m_data, other.m_data + other.m_size, m_data);
  m_size = other.m_size;
}
```

The assignment of h1 to h3, in the following example will invoke the copy assignment operator of h1:

```
Container h1(10000);         // regular constructor
Container h3(60000);         // regular constructor

h3 = h1;           // copy assignment operator of h3 is invoked
```

And the copy assignment operator for the *Container* class is:

```
Container& operator=(const Container& other)
{
   if(this == &other) return *this;          // (1)

   delete[] m_data;                          // (2)

   m_data = new int[other.m_size];
   std::copy(other.m_data, other.m_data + other.m_size, m_data);
   m_size = other.m_size;

   return *this;                            // (3)
}
```

First, there is protection against self-assignment (1). Then, the data in the object is deleted since we are replacing the content of this object with the content of another object.

There are problems with our *Container* object design. Consider this function:

```
Container createContainer(int size)
{
   return Container(size);
}

void main()
{
  Container container4(createContainer(20000000));
}
```

CreateContainer returns a *Container* object by value. That is, it creates and returns a completely new temporary Container object on the stack. This is a rvalue. This rvalue is then used to create a new Container object using the Container copy constructor.

So we have two very expensive memory allocations of the Container class. The first one is when the *createContainer* function creates and returns a Container object as a rvalue on the stack. The second occurs when the copy constructor allocates memory for the array and initializes *container4*.

There is a similar problem with the Container assignment operator:

```
int main()
{
  Container h;
  h = createContainer(20000000);        // copy assignment

}
```

The *createContainer* function creates a rvalue on the stack for a Container object. Then the copy assignment operator in the Container class de-allocates the data in the *h* object and allocates memory for the array and initializes it again using the assignment operator.

There are too many expensive copies. We already have a fully-fledged Container object. This is the temporary and short-lived rvalue object on the stack which is returned by `createContainer().` This object is built by the compiler, it's a transient rvalue that will become extinct before the next instruction is executed. Why not *move* the allocated data from this temporary object to the lvalue? This move will remove the need to allocate memory again. This is what move semantics enables.

In order to implement move semantics for the Container class, we have to create a move constructor and a move assignment operator for this class.

This is our move constructor:

```
Container(Container&& other)        // other is a rvalue reference
{
  m_data = other.m_data;      // (1)
  m_size = other.m_size;

  other.m_data = nullptr;     // (2)
  other.m_size = 0;
}
```

This constructor has a reference to a Container rvalue as it’s only parameter. In (1), we initialize the *m_data* pointer to the rvalue *m_data* pointer. Similarly we initialize *m_size* to the rvalue *other.m_size.* In (2), we null the *other* object’s pointer to the rvalue data so that it can’t be deleted or accidentally reused. This is important. Lastly, we set *other.m_size* to 0.

We have thus replaced an expensive memory allocation and copy operation with a pointer copy. The time complexity for creating the Holder object is constant regardless of the size of the underlying array.

The move assignment operator is similar:

```
Container& operator=(Container&& other)
{
  if (this == &other) return *this;

  delete[] m_data;         // (1)

  m_data = other.m_data;   // (2)
  m_size = other.m_size;

  other.m_data = nullptr;  // (3)
  other.m_size = 0;

  return *this;
```

Moving A lvalue To A rvalue

We can move a *lvalue* to a *rvalue* by using **std::move()**. *move()* performs a static cast of a *lvalue* to a *rvalue*. Look at the following program:

```
int main()
{
   Container h1(100000);     // h1 is an lvalue

   // copy constructor h2(Container&) invoked
   Holder h2(h1)
}
```

In this example, the *h2* copy constructor and the expensive memory allocation in it is invoked because *h1* is a *lvalue*. But we can avoid this expensive copy by moving h1 to a *rvalue*.

```
int main()
{
  Container h1(100000);            // h1 is an lvalue

  // move constructor is invoked because
  // the h2 constructor receives a rvalue

  Holder h2(std::move(h1));

}
```

In this example, the move constructor is invoked. *std::move* converts a *lvalue* to a *rvalue* and in our example, invokes the move constructor.

Throwing Exceptions

Ideally move constructors and move assignment operators should never throw an exception. The rationale is that we should not allocate memory or call functions in move constructors and move assignment operators. We should only move data and set other objects to null, these are non-throwing operations.

Modern C++: Smart Pointers (C++11)

A C++ program has five distinct physical memory areas. These are: the *Text* segment which stores the program code, the *Data* segment which stores static initialized variables, The *Block Started By Segment* (BSS) which stores static uninitialized variables, the *Stack Segment*, and the *Heap Segment (*you can also refer to the *Memory Layout Of C Programs* portion in this book).

The Stack

By default, variables and objects in C++ are stored on the stack:

```
include <iostream>

void foo(int a)
{
  if (a > 0) {
       std::string s = "positive number received: ";
       std::cout << s << a << "\n";
  }
}

int main()
{
  foo(5);
}

// output
positive number received: 5
```

In this example, the variables *a* and *s* are stored on the stack. The fundamental rule for stack variables is: objects allocated on the stack are automatically destroyed when they go out of scope.

In C++, a scope is defined by a pair of brackets { and } provided that these brackets are not used for variable initialization. Thus:

```
std::vector<int> v = {1, 2, 3};             // this is not a scope
```

```
if (v.size() > 0)
{                        // this is the beginning of a scope

}                       // this is the end of a scope
```

In C++ there are only three ways a variable can go out of scope:

- encountering a closing bracket *}*,
- encountering a return statement,
- having an exception thrown inside the current scope that is not caught inside the scope

The Heap

Dynamically allocated objects are stored on the heap. These are objects that are allocated memory by a call to *new*. *new* returns a pointer to the allocated memory:

```
float* pi = new float(3.127);
```

Objects that are allocated with *new* can only be destroyed by manually calling *delete*:

```
delete pi;
```

Contrary to objects on the stack, objects allocated on the heap are not destroyed automatically. This offers the advantage of being able to keep these objects around beyond the end of a scope.

There are benefits in having pointers to allocated memory. Objects and functions can receive copies of pointers; these copies are very cheap to make. Pointers also allow us to manipulate objects through polymorphism: a pointer to a base class polymorphic object can point to an object of a derived class.

However deleting objects on the heap is not trivial. *delete* has to be called once and only once to free the memory allocated to a heap-based object. If *delete* is not called, the memory allocated to the object is not freed, and this memory space is not reusable This is called a memory leak. If on the other hand, the program will have undefined behaviour if *delete* is called more than once on the same address.

Types Of Memory Errors

When we are dealing with memory allocation and pointers there are five types of possible errors:

Memory Leak: failure to free memory:

```
#include <memory>

void main() {
    int *ptr = new int;
}
```

Double Free: Freeing the same piece of memory twice:

```
#include <memory>

void main() {
    int *ptr = new int;
    free ptr;
    free ptr;
}
```

Use After Free: Using a pointer after the memory to which it points has been freed:

```
#include <memory>

void main() {
    int *ptrFoo = new int;
    free ptrFoo;
    *ptrFoo = 5;
```

The problem here is that we are using *ptrFoo* after the memory allocation to which it points has been freed. *ptrFoo* is called a **dangling pointer**. The program statement: **ptrFoo = 5* is a *Use After Free* (UAF).

Buffer Overrun: Using the pointer to write to a memory location that has not been allocated:

```
#include <memory>

void main() {
   int* ptrFoo = new int;
   *ptrFoo = -1;
   ++ptrFoo;
   *ptrFoo = 5
```

In this example, we allocate memory for an *int* type. *ptrFoo* points to this memory location which is initialized to -1. Next we increment the pointer and it is now pointing to a memory location which has not been allocated. Finally we write 5 into this memory location. This is an example of a buffer or memory overrun.

Uninitialized Memory Access: Using the contents of an allocated memory location that has not been initialized:

```
 #include <memory>

void main() {
    int *ptr = new int;

    int x = *ptr;
}
```

All of these errors will result in undefined behavior. With smart pointers we can ensure that allocated memory will be freed and pointers cannot be used after they are freed. Address sanitizers detect double free, use after free and buffer overrun errors. We will look at address sanitizers later.

Memory Semantics

The problem with freeing memory is that objects can have ambiguous memory management semantics. Consider:

```
Sandwich* ptrSandwich = makeASandwich();
```

The function *makeASandwich* returns a pointer to a Sandwich object allocated on the heap. Suppose now that *ptrSandwich* is passed to some other function, tnen who should be responsible for deleting *ptrSandwich*, the caller or the called function? If the caller neglects to delete the pointer and no other caller deletes the pointer then there is a memory leak. If the caller deletes *ptrSandwich* and some other function also deletes this pointer, then the program has undefined behaviour. If *ptrSandwich* is deleted and subsequently some function uses this pointer, then the program will also have undefined behavior.

RAII

Smart pointers are a solution to dangling pointers and UAF errors.

RAII means *Resource Allocation Is Initialization*. The principle of RAII is simple: we wrap an object around a resource allocated on the heap and free the resource in the object’s destructor. The object exists on the stack and is thus deleted automatically when the object goes out of scope. C++ guarantees that an object’s destructor will be called when an object on the stack is destroyed. This is exactly how smart pointers work.

Lets consider specifying a smart pointer class that wraps a raw pointer of type T. A smart pointer class for T can be defined as follows:

```
#include <iostream>
```

```
template <typename T>
class SmartPointer
{
public:
  explicit SmartPointer(T* p) : ptr(p)
  {
     std::cout << "smart pointer constructor called" << std::endl;
  }
  ~SmartPointer()
  {
    delete ptr;
    std::cout << "smart pointer destructor called" << std::endl;
  }

  const T operator*() {  return *ptr; }

private:
  T* ptr;
};

int main()
{
  // allocate some raw memory
  int  *p = new int(5);

  // wrap a smart pointer class around the raw memory

  SmartPointer<int>  sp(p);
  std::cout << "sp value = " << *sp << std::endl;

}

// output
smart pointer constructor called
sp value = 5
smart pointer destructor called
```

The main benefit of a smart pointer object is that we can manipulate smart pointers as objects allocated on the stack. The compiler automatically takes care of calling the destructor for a smart pointer because objects allocated on the stack are automatically destroyed when they go out of scope.

If we overload the * and -> operators for a smart pointer class, then a smart pointer class can syntactically behave like a raw pointer. The smart pointer can be dereferenced with the * *operator* or the -> *operator*, that is to say you can call **sp* or *sp->* on it. If we enhance our smart pointer class so that it can be typecast to a *bool* type. We can then use it in an *if* statement like a raw pointer:

```
if (sp) {
   statements
}
```

sp tests the nullity of the underlying raw pointer.

For our example above, consider the following program fragment for two smart pointer classes:

```
smart_pointer<int>     sp1(p)
smart_pointer<int>     sp2(q)

sp1 = sp2
```

This will lead to undefined behaviour, since the assignment will invoke the default copy assignment operator on sp1 and we will get two objects that both use the same memory location. We are also liable to delete the same allocated memory twice. In addition, this assignment will cause the initial memory allocated by sp1 to be lost. This shows that our smart pointer class needs to be refined to handle copy initialization and copy assignment as well as other raw pointer features.

The Smart Pointer Types

Going back to the previous example, a decision has to be made about how a smart pointer should be copied. Otherwise, the default copy constructor will be used, which will produce undefined behaviour.

The Standard Library has a solution to this problem which leads to the specification of different smart pointer classes. Each pointer class reflects a design decision.

std::unique_ptr

The semantics of the *std::unique_ptr* class is that it is the sole owner of the memory that it wraps. A *std::unique_ptr* object holds a raw pointer and deletes it in it's destructor.

Consider a *makeASandwich()* function:

```
#include <memory>

Sandwich* ptr = makeASandwich()
std::unique_ptr<Sandwich>  sp(makeASandwich());
```

makeASandwich allocates memory for a *Sandwich* object and returns a pointer to the allocated memory. sp is a smart pointer object, it wraps a *std::unique_ptr* object called *sp* around the raw *Sandwich* pointer. We own the smart pointer object and no one no one can delete or use this object except us. We have ownership of the smart pointer and can safely modify the *Sandwich* object that is pointed to without worrying that someone may have deleted the pointer or altered the object.

std_unique pointer is polymorphic and can point to derived objects.

There are two types of *unique_ptr* objects. One for scalars and the other for arrays:

- `unique_ptr<Foo>` wraps a pointer to an object of type Foo.
- `unique_ptr<Foo[]>` wraps a pointer to an array of Foo objects.

The following examples show how to create *std::unique pointer* objects. In these examples *sp1, sp2* and *sp3* are smart pointer objects:

- **`std::unique_ptr<double>   sp1(new double(3.14));`**
- **`std::unique_ptr<double[]> sp2(new double[50]);`**
- **`std::unique_ptr<Foo>      sp3(new Foo)`**

The second declaration creates a std*::unique_ptr* object that wraps a raw pointer to an array of fifty doubles.

Here is an example of usage:

```
#include <iostream>
#include <memory>

int main()
{
  std::unique_ptr<double[]> sp(new double[5] {1,2,3,4,5});
  int i = 1;

  // get the raw pointer
  auto ptr = sp.get();

  for (int ctr = 0; ctr < 4; ctr++)
  {
    *ptr = 2 * i;
    i++;
    ptr++;
  }

  ptr = sp.get();
  for (int ctr = 0; ctr < 4; ctr++)
  {
    std::cout << *ptr << " ";
    ptr++;
  }

}

// output
2 4 6 8 10
```

Notice that *sp.get()* returns the raw pointer that is wrapped by the smart pointer object.

An std`::unique_ptr` object is the **owner** of the allocated memory that it wraps. If the smart pointer object goes out of scope without passing its ownership to another variable, then the object is deleted and the memory that it points to is freed in the object's destructor.

An *std::unique_ptr* has only one owner at any one time. We cannot simply copy a std`::unique_ptr:`

```
std::unique_ptr<int>   p(new int(5));
std::unique_ptr<int>   q;
q = p;                              // compiler error
```

Here is another example:

```
#include <iostream>
#include <memory>

void borrower1(std::unique_ptr<double[]> *sp)
{
  std::cout << "in borrower1. memory address of pointer to sp = ";
  std::cout << sp;
  std::cout << std::endl;
}

void borrower2(std::unique_ptr<double[]> &sp)
{
  std::cout << "in borrower2. memory address of reference to sp = ";
  std::cout << &sp;
  std::cout << std::endl;
}

void memory_address()
{
  // sp is created and owns a section of memory containing
  // 42 doubles.
  std::unique_ptr<double[]> sp(new double[42]);

  std::cout << "address of created sp = " << &sp << std::endl;

  borrower1(&sp);
  borrower2(sp);

}
```

```
int main()
{
  memory_address();
}

// output
address of created sp = 0x7fff565e8488
in borrower1. memory address of pointer to sp = 0x7fff565e8488
in borrower2. memory address of reference to sp = 0x7fff565e8488
```

Moving A unique_ptr

If an owner wants to yield it's ownership of a *std::unique_ptr* object to another variable, it must use *std::move*:

```
#include <iostream>
#include <memory>
#include <utility>                 // required header for std::move

void calc(std::unique_ptr<double[]>& x)
{
    std::cout << "in calc function " <<  &x << std::endl;
}

void taker(std::unique_ptr<double[]>& x)
{
    std::unique_ptr<double[]> d = std::move(x);
    std::cout << "ownership taken: " << &d << std::endl;
}

int main()
{
  // p is created and owns a contiguous block of memory
  // containing 42 double numbers
  std::unique_ptr<double[]> p(new double[42]);

  // move p to q
  std::unique_ptr<double[]> q = std::move(p);

  calc(q);

  if (q != nullptr)
  {
    std::cout << "q was not destroyed in calc" << std::endl;
  }

  // move q into the argument of the taker function
  taker(q);

  // memory allocated to q is freed when the scope of
  // taker is exited.
  if (q == nullptr)
  {
    std::cout << "q destroyed in taker()";
  }
}
```

```
// output
in calc function 0x7fffe994e530
q was not destroyed in calc
ownership taken: 0x7fffe994e508
q destroyed in taker()
```

Deleting A std::unique_ptr Object

Normally an `std::unique_ptr` object will automatically delete the pointer that it wraps when the smart pointer object leaves scope. However we can manually delete a smart pointer object with ***sp.reset()***.

Releasing A unique_ptr To A Raw Pointer

A *unique_ptr* object can also release itself to the raw pointer that it wraps:

```
double*   raw_pointer = new double[50];
std::unique_ptr<double[]> sp(raw_pointer);
raw_ptr = sp.release();
```

After the release, the smart pointer object will be destroyed.

std::shared_ptr

A single memory resource can be shared by several *std::shared_ptr* objects that wrap the resource. The application maintains an internal count of how many times the resource is shared by *std::shared_ptr* objects. When the shared pointer count reaches zero, the memory allocated to the shared resource is automatically freed.

Therefore *std::shared_ptr* allows copies of the shared smart pointer object to be made, but uses reference-counting to make sure that every shared smart pointer object is deleted only once. The memory allocated to the resource will be automatically destroyed when there are no shared smart pointer objects using the resource.

Each *std::shared_ptr* object maintains pointers to two shared memory areas: the memory area containing the shared data and a memory area containing the reference count.

When a new *std::shared_ptr* object is created, its constructor increases the reference count associated with the shared data by one.

When a *shared_ptr* object goes out of scope, its destructor decrements the reference count to the shared resource by one. If the reference count becomes 0, this means that no *shared_ptr* object is associated with the shared memory. In this case, the destructor of the last shared pointer object deletes the shared memory.

Shared Memory Pointers

The following is an example of shared memory pointer usage:

```
#include <iostream>
#include <string>
#include <memory>


void shared_user(std::string s, std::shared_ptr<double[]>& x)
{
  std::cout << "share taken by: " << s;
  std::cout << " reference count: " << x.use_count() << std::endl;
}

void share_memory()
{
  // shared pointer is created, it points to a
  // contiguous block of memory containing 42 double numbers
  // memory is automatically allocated for the reference count

  std::shared_ptr<double[]> sp( new double[42] );

  // share sp
  shared_user("user 1", sp);

  std::shared_ptr<double[]> q = sp;
  shared_user("user 2", q);

  // share sp again
  std::shared_ptr<double[]> r = sp;
  shared_user("user 3", r);

}

int main()
{
  share_memory();

}

// output
share taken by: user 1 reference count: 1
share taken by: user 2 reference count: 2
share taken by: user 3 reference count: 3
```

Creating A Shared Memory Pointer

The following example creates a shared memory pointer through a constructor:

```
#include <memory>
#include <utility>

std::shared_ptr<double>  p1(new double(1.5));
```

The following example invokes the copy constructor to create a new shared memory pointer object:

```
#include <memory>
#include <utility>

std::shared_ptr<double>  p1(new double(1.5));
std::shared_ptr<double>  p2(p1);
```

We can also use the copy assignment operator:

```
std::shared_ptr<float> f
std::shared_ptr<float> ft(new float(3.12))
f = ft
```

Get The Reference Count For A Shared Memory Object

```
sp.use_count();
```

Get The Raw Pointer

```
std::shared_ptr<float> sp(new float(3.12))

float* ptr = sp.get();
```

Detaching The Raw Pointer

The raw pointer wrapped by a shared pointer object can be detached with:

```
auto ptr = p1.reset();
```

This decreases the reference count of the *shared_ptr* object by 1 and if this reference count becomes 0, then the shared resource is deleted in the destructor of the last shared pointer object.

Check That Shared Memory Has Not Been Deleted

```
sptr.use_count();     // reference count

or:

sptr.expired();       // much faster than the reference count
```

Shared Pointers And nullptr

Setting a shared pointer object to *nullptr* will decrease the reference count of the shared resource by one.

Consider:

```
#include <iostream>
#include <memory>

int main()
{
  std::shared_ptr<int> p1(new int(6));       // reference count is 1
  std::cout << p1.use_count() << std::endl;

  std::shared_ptr<int>  p2(p1);              // reference count is 2
  std::cout << p1.use_count() << std::endl;

  p2 = nullptr;                             // reference count is 1
  std::cout << p1.use_count() << std::endl;
}

// output
1
2
1
```

Weak Pointers

Sometimes we want to access a shared object without causing the *shared_ptr* reference count to be incremented.

An *std::weak_ptr* object holds a pointer to a shared memory resource along with other *std::shared_ptr* objects; but a weak pointer object does not cause the reference count to be incremented. This means that if the reference count of s*td::shared_ptr* objects becomes zero, then the memory allocation for the shared resource will be deleted even if some weak pointers still point to the memory allocated for the resource.

Creating A Weak Pointer

The syntax for creating a weak pointer is:

```
#include <memory>
std::weak_ptr<int> wptr;
```

This creates an uninitialized weak pointer of type *int*.

Next. we create a shared pointer object and share it with a weak pointer as follows:

```
#include <memory>

std::shared_ptr<int> sp(new int(5));      // reference count 1
std::weak_ptr<int> wptr;                  // uninitialized

wptr = sp;                                // reference count 1
```

Use A Weak Pointer To Check That Shared Memory Has Not Been Deleted

```
wptr.use_count();     // reference count

or:

wptr.expired();       // much faster than the reference count
```

Writing To Shared Memory With A Weak Pointer

If we want to use a weak_pointer to write to shared memory then we have to first convert the *std::weak_ptr* to a *std::shared_ptr* (the reference count will increase).

To get a shared_ptr from a weak_ptr, we call lock():

```
#include <iostream>
#include <memory>


int main()
{
  std::shared_ptr<int> sp(new int(5));
  std::weak_ptr<int>  wp(sp);

  std::shared_ptr<int> spW = wp.lock();

  if (spW)
  {
    // the resource is still available and can be used
    std::cout << "resource available" << std::endl;
  }

  else
  {
   // the shared resource is no longer available
   std::cout << "resource not available" << std::endl;
  }

}

// output
resource available
```

Delete A Weak Pointer

```
wptr.reset();
```

Modern C++: Variadic Templates (C++11)

Before C++11, templates were required to have a fixed number of parameters. C++11 permits templates to have a variable number of parameters. These types of templates are called variadic templates.

In order to implement a variable number of parameters in a template, we define a *parameter pack* in the template definition, and expand the parameter pack when a template is specialized.

A parameter pack is defined with an ellipsis (...) after the class or typename keyword in a template declaration. There can be spaces before and after the ellipsis:

```
template <typename... Args>
void foo(Args... args) {

};
```

This declares a variadic function template *foo* that has an arbitrary number of function parameters. The parameters need not have the same type.

There are three kinds of template parameters, and we can define a parameter pack for each of them using the ellipsis notation:

```
// Ts is a list of parameters (not necessarily off the same type)
template <class... Ts>

// Ns is a list of unsigned integer parameters
template <unsigned... Ns>

// Us is a list of template parameters
template <class T>... class Us>
```

Note that in a function template such as *template<class ... T>,* the parameters need not have the same type.

We can mix single parameters and parameter packs, with the restriction, that only a single parameter pack can be defined in a template and it must be the last element in the template parameter list:

```
template <class X, int I, class... T2>
void foo(X x, int y,  T2... args) {
  ...
};
```

Note: A parameter pack can contain zero elements.

Variadic Function Templates

A function that has a parameter pack is called a **variadic function**. Here is a declaration of a variadic function template:

```
template <int i, class... Ts>           // Ts is the template parameter pack
void foo(int x, Ts...  ts) {            // ts is the function's parameter pack
   ...
}
```

This template definition specifies a parameter pack *Ts* which represents a variable number of template parameters with possibly different types. In the variadic function *foo,* we specify that the second argument of *foo* is a parameter pack *ts*. *ts* will expand to the arguments actually received by *foo*.

We can visualize a particular specialization of the variadic function template *foo* as:

```
template <int x, class T1, class T2, class T3>
```

```
void foo(int x, T1 t1, T2 t2, T3 t3) {

}
```

This specialization shows that a variadic function template can have different types of parameters.

Parameter Pack Expansion

The only thing that we can do with a parameter pack is to *expand* it. Parameter pack expansion yields a comma separated list of parameters:

```
template <int i, class... Ts>
void foo(int i, Ts... args) {

    ...
}
```

Upon specialization, the function template *foo* will expand into a comma separated list of parameters.

The sizeof… Operator

The `sizeof...` operator returns the number of parameter pack arguments in a specialization:

```
#include <iostream>

template <class... Ts>
void printCount(Ts... args)
{
  std::cout << "sizeof...(args): ";
  std::cout << sizeof...(args) << '\n';
}

int main()
{

  printCount(5, 11.2, 3);
  printCount();
  printCount("hello", 5);
}

// output
sizeof...(args): 3
sizeof...(args): 0
sizeof...(args): 2
```

sizeof... is a compile-time constant.

Using Recursion In Variadic Functions

Suppose that we want to add some numbers in a variadic function template. The following conventional implementation will generate a compiler error:

```
template <typename... Ts>
double sum(Ts... ts) {
    double result = 0.0;
    for (auto el : ts)
      result += el;
    return result;
}
```

The compiler cannot expand the variadic parameters into an iterable list. The addition of the variadic function arguments in *sum* must be implemented through recursion.

As we will see subsequently, C++17 allows variadic functions to be computed easily using fold expressions.

Modern C++: nullpointer (C++11)

C and C++ define the macro NULL as the integer 0 or 0l. NULL is an integer type. A pointer that has the value NULL is interpreted as pointing to no value. This gives rise to ambiguous behaviour:

```
void foo(int);       // forward declaration

char*  ptr;
ptr = NULL.

foo(ptr);            // equivalent to calling foo(0)
```

For *foo(0)* has *foo* received a pointer or an int?

nullptr is a new keyword in C++11. *nullptr* is meant to be a replacement for *NULL*. It is a literal constant of type *std::nullptr_t*. *nullptr* provides a typesafe pointer value representing an empty (null) pointer.

nullptr it is not convertible to any integral type. Thus we can perform boolean comparisons:

```
#include <iostream>

void foo(int* ptr)
{
   if (ptr == nullptr)
      std::cout << "nullptr received" << std::endl;
   else
      std::cout << *ptr << std::endl;
}

int main()
{
  int*  ptr;
  ptr = nullptr;

  foo(ptr);
}
```

```
// output
nullptr received
```

As the previous example shows, we can do boolean comparisons with *nullptr*:

```
if (ptr == nullptr) {  }
```

Modern C++: Raw Strings (C++11)

Raw Strings

C++11 lets us define raw strings. A raw string is a sequence of characters that is interpreted literally; it does not have any characters that are interpreted specially. For example, the character pair \n in a raw string will not be interpreted as a newline.

The general syntax for a raw string is: ***R"(...)"***

A raw string starts with ***R"(*** and ends with ***)”***. The raw string is inside the parentheses. ***(*** and ***)*** are the delimiters for the raw string. It can contain line breaks as well as other special characters which will be interpreted literally.

We can specify special delimiters for a raw string to handle quotation marks inside the parentheses.

A delimiter is a user-defined sequence of up to 16 characters that immediately precedes the opening parenthesis of a raw string, and immediately follows after the closing parenthesis. The delimiter must not include quotation marks. A delimiter marks the start and end of a raw string.

For example, in the expression:

```
R"abc(Hello"\()abc"
```

the delimiter is *abc* and the string content *Hello"\(* is inside the parentheses.

Here is a raw string example:

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

```
int main()
{
  // Normal string
  string str1 = "Sample 1.\nmultiline.\ncontent.\n";

  // Raw string
  string str2 = R"(Sample 2.\nmultiline.\ncontent.\n)";

  //Raw string with special delimiter 123
  string str3 = R"123(Sample 3.\nraw string.\ncontent.\t\r\n)123";


  // Output
  cout << str1 << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}

Sample 1.
multiline.
content.

Sample 2.\nmultiline.\ncontent.\n
Sample 3.\nraw string.\ncontent.\t\r\n/ output
```

Encoding String Literals

By using an encoding prefix, we can specify a special character encoding for a string literal. For example:

```
L"hello"            // defines “hello” as a wchar_t string literal
u8”hello”           // defines a string literal with UTF-8 encoding
```

The initial R that starts a raw string can be preceded by an encoding prefix. For example:

```
u8Rwww”(\nhello world\n)www”;
```

Modern C++: noexcept Attribute (C++11)

C++11 provides the function attribute: *noexcept*. *noexcept* indicates to the compiler that the function will not throw an exception. This enables the compiler to generate much more efficient and faster code since it does not have to generate code to handle run-time exceptions or unwind the stack. The syntax for *noexcept* is as follows:

```
void foo () noexcept
{

}
```

Here is an example:

```
#include <iostream>

int bar(int y)
{
  if (y == 5)
     throw("exception");
  else
     return y + 11;
}

int foo(int x) noexcept
{
  return bar(x);
}

int main()
{
   auto ret = foo(7);
   std::cout << ret << std::endl;
}

// output
18
```

Notice that a function with the *noexcept* attribute is not precluded from calling a function that can throw an exception.

We can also specify a condition under which a function will not throw an exception. For example:

```
void swap (Type& x, Type& y) noexcept(x == y)
{
   ...
}
```

The *noexcept* predicate expression *(x == y)* for this function, informs the compiler that the function will not throw an exception if the predicate expression is true. Specifying *noexcept* without a condition is shorthand for specifying *noexcept(true)*.

Modern C++: Override And Final (C++11)

The override Attribute

Every time we define a method in a derived class that overrides a `virtual` method in it's base class, we should give the function the `override` attribute. This is a safety feature:

```
class Base
{
  public:
    virtual void f()  {
      std::cout << "Base class f \n";
   }
};

class Foo : public Base
{
public:
  void f() override {
     std::cout << "Derived class f \n";
  }
};
```

If we tag a virtual member function with the *override* keyword, the compiler will make sure that the virtual member function exists in the base class, and it will prevent the program from compiling if this is not the case.

Using override Can Prevent A Dangerous Bug

When a function is marked with the override attribute, it must exist in the base class as a virtual function. Thus if the base class declares a function as *virtual void func(int)* but the derived class mistakenly declares a function: *void func(float) override*, then the compiler will catch the error since the *func* in the derived class should have an *int* type parameter. Of course, we can declare *func* in the derived class as *virtual func(float)* (without the override keyword), and then we will have a new virtual function that has a different parameter signature.

Using override can be extremely useful when, for example, the function *f* in the derived class is mistakenly tagged as **const**. Since the declaration of *f* in the derived class differs from the declaration in the base class, the compiler will emit an error and refuse to compile. If the *override* tag is not used in the derived class, the program will compile but the function in the derived class will not override the virtual function in the base class. That is, the function in the derived class is entirely new. This is a very subtle bug. Using the *override* keyword in the function declaration prevents these types of subtle function attribute errors.

Override And Trailing Return Types

When a virtual function declaration has a trailing return type, the override qualifier must come after the trailing return type.

```
virtual auto foo() const noexcept -> int override;
```

This is because override is not part of a function's type.

Final Member Functions

The attribute *final* can be used with virtual functions and classes. Consider this class:

```
class Base
{
 public:
   virtual void f()  {
      std::cout << "base class f \n";
   }
}

class Foo : public Base
{
public:
  void f() override final  {
      std::cout << "Derived class f \n";
  }
};
```

In this example, *f()* is declared as a virtual function in the base class. The *final* attribute of *f()* in the *Foo* class prevents any class derived from *Foo* from overriding the virtual member function `f`.

Final Member Classes

The *final* attribute can also be applied to class declarations:

```
class Foo final : public Base
{

};
```

This will prevent the class *Foo* from being inherited.

Modern C++: Constructor Delegation (C++11)

Sometimes it is useful for a constructor to be able to call another constructor of the same class. This feature is called **Constructor Delegation.** Prior to C++11, this was not possible.

Consider the program below:

```
#include <iostream>

class Foo {
    int x, y, z;

public:
    Foo() {
        x = 0;
        y = 0;
        z = 0;
    }
    Foo(int z) {
        x = 0;
        y = 0;
        this->z = z;
    }

    void show() {
        std::cout << x << " " << y << " " << z << std::endl;
    }
};

int main()
{
    Foo foo(3);
    foo.show();
}

// output
0 0 3
```

The problem with this implementation is that both *Foo* constructors share redundant code. We could get rid of this redundancy by implementing an *init* function:

```
#include <iostream>
```

```
class Foo
{
  int x, y, z;

  // init function to initialize x and y
  void init()
  {
    x = 0;
    y = 0;
  }

public:
   Foo()
   {
     init();
     z = 0;
    }

    Foo(int z)
    {
      init();
      this->z = z;
    }
    void show()
    {
        std::cout << x << " " << y << " " << z << std::endl;
    }
};

int main()
{
  Foo foo(3);
  foo.show();
}
```

There are two problems with this implementation: (i) *init* makes the implementation more complex by introducing another function, and (ii) what if *init* is called by mistake after the object has been initialized. *init* would need guard code to protect against this possibility.

Delegated constructors provide a better solution. Constructor delegation allows us to call another constructor of the same class by placing it in the initializer list of the invoking constructor:

```
#include <iostream>
```

```
class Foo
{
  int x, y, z;

public:
    Foo()
    {
        x = 0;
        y = 0;
        z = 0;
    }

    Foo(int z) : Foo()          // constructor delegation
    {
        this->z = z;
    }

    void show()
    {
        std::cout << x << " " << y << " " << z << std::endl;
    }
};

int main()
{
    Foo foo(3);
    foo.show();
}

// output
0 0 3
```

The delegated constructor on the right hand side of the colon is evaluated first; then the body of the invoking constructor is executed.

Note: we can have only one delegated constructor on the right hand side of the colon.

It is important to note that constructor delegation is different from calling the delegated constructor from inside the body of the constructor. The latter is bad practise.

Modern C++: Static Assertions (C++11)

The *static_assert* directive tests a constant expression at compile time. This constant expression must be an integral expression that can be cast to a *bool* type. If the expression is false, the compiler issues an error message and stops the compilation process; otherwise the *static_assert* has no effect. The condition that is being tested must be a constant integral expression that can be evaluated at compile-time.

The syntax for static assertions is:

```
static_assert( expression, string_literal );
```

expression is a predicate that is checked for truth or falsity at compile-time. If the condition is false, the compiler prints the string literal and stops.

Here is an example:

```
int main()
{
    constexpr double x = 123;
    static_assert(x == 456, "x value is not 456");
}


// output
static assertion failed: x value is not 456
```

The following example demonstrates the use of *static_assert* in templates:

```
#include <iostream>

template <class T, int Size>
class Foo
{
    public:
    Foo() { };
    // Compile time assertion to check if  the size of
    // Foo is greater than 3. If a Foo is declared with
    // size is less than 4, the static_assert will fail

    static_assert(Size > 3, "Foo size is too small");

    private:
    T  m_values[Size];
};

int main()
{
    Foo<int, 4>   foo1;
    Foo<double, 2> foo2;
}


// output
 error: static assertion failed: Foo size is too small
```

In this example, we have created a class template named *Foo* where we do not permit *Foo* objects of size less than four to be created. Therefore, inside the class template body, we placed a *static_assert* statement to test if the *Foo* object has the proper size.

static_assert can be used at namespace, class and block scopes.

In the following code snippet, we test the version of a library at compile-time:

```
static_assert(mylibray::version > 2, “wrong library version”)
```

Prior to *static_assert* we would have to test the library version at run-time using a function call or explicitly link the proper library version and use a macro such as *#define MYLIB 3* to identify the library.

static_assert is also useful for hardware and portability tests:

```
static_assert(sizeof(int) == 32, “unsupported int type”)
```

Modern C++: Deleted And Default Functions (C++11)

Compiler Generated Special Functions

C++ classes have six types of special member functions:

- Constructors
- Copy constructors
- Copy assignment operators
- Move constructors
- Move assignment operators
- Destructors

These special member functions are used to create, destroy, assign, copy and move class objects.

The compiler will generate these special functions if they are needed and have not been already declared. These functions are declared as public and inline.

default Keyword

C++ will automatically create default versions of these special functions if they are needed and have not been already defined.

In C++11, the *default* and *delete* keywords give us explicit control over whether these special member functions are to be automatically created by the compiler or deleted.

We can explicitly direct the compiler to create a default implementation for any one of these special functions. This can be more efficient than manually coding the function implementation by ourselves.

The syntax for explicitly creating a default implementation of a special function is as follows:

```
class farm
{
  farm() = default;
...
}
```

This will create the default farm class constructor. The *default* keyword states that this special member function is to be created using it's default implementation.

Note: empty destructors can cause optimization problems in modern C++ compilers and should be avoided. Instead use the following type of declaration: *~Foo() = default* or preferably do not declare a destructor at all.

Deleted Functions And The Delete Keyword

The *delete* keyword instructs the compiler to not create a special member function. Deleting a special member function prevents the compiler from generating the default implementation of the special member function.

Note: An attempt to provide an implementation for a defaulted or deleted special member function will raise a compile-time error.

The syntax for deleted special functions is:

```
Foo(Foo&&) = delete;
```

Note: A function must be defaulted or deleted at the point of it's declaration.

We can also delete ordinary member functions and as well as non-member functions to prevent them from being created.

Prior to C++11, if we wanted to prevent the compiler from automatically generating a default special function, we would declare it as a private member of the class, but not implement it. For example:

```
class Foo
{
   Foo() {};

private:
  Foo(const Foo&);
  Foo& operator=(const Foo&);
};
```

C++11 provides a simpler solution:

```
class Foo
{
  Foo() = default;
private:
  Foo(const Foo&) = delete;
  Foo& operator=(const Foo&) = delete;
};
```

Modern C++: Typed Enums (C++ 11)

C and C++ Enums

In C and old C++ an enum has the following syntax:

```
enum Color {RED, GREEN, BLUE};
```

RED is an integer with value 0, GREEN has value 1 and BLUE has value 2. *enum Color* is a type and a variable of this type can only have one of the three enumerated values. *enum Color* is called an enum type and it’s permissible values are called enumerators.

An instance of this type is declared as follows:

```
enum Colour flower_colour;
```

In C++11, we can write the previous declaration as:

```
Colour flower_colour;
```

Enum declarations can give rise to problems such as the following:

```
enum Colour {RED, GREEN, BLUE};
enum Feelings {EXCITED, BLUE, MOODY};

enum Color val = BLUE;
```

This will not compile. The compiler does not know whether *val* is referring to BLUE in *enum Color* or in *enum Feelings*.

There is also a semantic problem. The BLUE in the first enum means something different than the BLUE in the second enum.

This problem arises because both *Colour* and *Feelings* are unscoped enums. Enumerated values are not being accessed through a scope.

Typed Enums

C++11 solves the previous problem by introducing strongly typed enums:

```
enum class Colour {RED, GREEN, BLUE};
enum class Feelings {EXCITED, BLUE, MOODY};
```

These are scoped enums. To access BLUE, we have to use the scope resolution operator:

```
Colour::BLUE
or
Feelings::BLUE
```

We can use scoped enums as follows:

```
#include <iostream>

int main()
{
    enum class Color {RED, GREEN, BLUE};          // declaration
    Color my_color = Color::GREEN;

   if (my_color == Color::GREEN)
       std::cout << "green color" << std::endl;
  else
       std::cout << "not green" << std::endl;
}

// output
green color
```

Color is a type and we can initialize a variable *my_color of* of type *Color* using a scoped enum:

```
Color my_color = Color::GREEN;
```

Enumerator Values

By default, enumerator values start at 0 and increment by one for each successive enumerator. We can however, explicitly set the integer values for some or all of the enumerators. When a value for an enumerator is set, succeeding enumerators will have their values incremented by one until we encounter an enumerator those value is explicitly set:

```
enum class Money { cent=1, two_cent, three cent, quarter=25, dollar=100 };
```

Data Types For An Enum

By default, enumerators have *int* type. However it is up to the compiler to decide what specific *int* type to use: *unsigned int*, *signed int*, *long*, *char* and so forth. With scoped enumerations we can explicitly set the integer type of the enumerators:

```
// we only have three colours, so there is no need to use an int type
// we can use an 8 bit char type

enum class Colours::char { RED = 1, GREEN, BLUE = 10 };
```

Enum Type Safety

In C++11, enums are strongly typed. This means that an integer type will not be implicitly converted to an enum type by the compiler and an enum type will not be implicitly converted to an integer type. Thus the following program with a scoped enum will not compile:

```
#include <iostream>
#include <string>

int main()
{
  enum class Color { RED, GREEN, BLUE };
  Color my_color;

  my_color = 1;                              // error
  if (my_color == Color::GREEN)
     std::cout << "green color" << std::endl;

}

// output
error: cannot convert 'int' to 'main()::Color' in assignment
```

This strong type safety also applies to unscoped enums in C++11.

Forward References For Enums

We can specify forward declarations for scoped and unscoped enums:

```
# forward declaration of Mood enum
enum class Mood;
```

Well-defined Enum Sizes: <cstdint>

C++ does not ordinarily provide fixed, well-defined bit sizes for it’s native types. For example, sometimes we want a 32 bit integer, not just an *int* type. This is because an *int* type might have different sizes on different CPU architectures. C++11 solves this problem with the C99 header file **stdint.h**. We include this header file as ***#include***

<*cstdint*> to specify integer types with well-defined sizes.

The *cstdint* header includes types such *std::int8_t, std::int16_t, std::int32_t,* and *std::int64_t* (as well as unsigned versions that begin with a *u prefix).* For example*:* *std::uint8_t* refers *to an unsigned int* type that is 8 bits long. The integer types in the *cstdint* header specify the bit sizes of the various types.

We can use these *cstdint* types as follows:

```
#include <cstdint>
enum class Colour : std::int8_t { RED = 1, GREEN = 2, BLUE = 3 };
```

Modern C++: Type Alias (C++11)

Typedef

A C or C++ **typedef** declaration introduces a name that, within its scope, becomes a synonym for an existing type. The syntax is:

```
typedef [existing-type] [our-alias];
```

And here is an example:

```
// Declare UL as a synonym for the unsigned long type.

typedef unsigned long UL;
UL x = 5;
```

Type Alias Declarations With The *using* Keyword (using =)

We have previously seen that we can bring a namespace into scope with the *using* keyword:

```
using namespace std;
```

In C++11, we can also utilize the *using* keyword to declare that a name is an alias for a previously declared type. The syntax is:

```
      using new_type_name = some_declared_type;
```

The new type name is a synonym for the existing type within the new type's scope.

Here are some examples:

```
using iter_t = std::vector<int, string>::iterator
using counter = long;
using Map = std::map<std::string, std::string>;
```

using Type aliases also work with function pointers, but they are more readable than the equivalent *typedef*:

```
double adder(int x, double y) {
   double z = x + y;
   return z;
}

// declaration of type funcptr
using funcptr = double(*)(int, double);

// definition
funcptr f = adder;

this is equivalent to:
void (*)(int, double) f = adder;
```

typedef and *using* can be used to impart semantic content to types:

```
using velocity = long;
```

using And Templates

An advantage of *using* type alias declarations over *typedefs* is that the latter cannot be used with templates.

The following is the syntax for template aliases:

```
template<class T> using Vec = std::vector<T, MyAllocator<T> >;

Vec<int> vec;
```

Vec is an alias for a family of std::vector types.

We read this as *Vec<T>* is a typedef for *std::vector<T, MyAllocator<T>>*.

Modern C++: User Defined Literals (C++11)

In C++ there are four types of literals: integer, floating point, char and string literals. Integer literals can have a suffix that indicates the data type of the literal. For example, *890750ul* indicates an unsigned long integer literal.

With user defined literals we can define our own units of measurement. This is accomplished by specifying a literal value and then attaching a custom suffix to this value. The suffix converts the value into suffix units. For example:

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

float operator ""__sections(long double x) { return x / 640; }

int main()
{
  float result = 1055.44__sections;
  cout << result << endl;
}

// 1.64912
```

This example converts acres into sections. A section is 640 acres. An integer literal with a custom suffix such as sections is called a user defined literal. The declaration syntax for user defined literals uses operator functions as follows:

```
long double operator "" __cm(long double x) { return x * 10; }

long double operator""  __ms(long double x) { return x * 1000; }

long double operator "" __mi(long double x) { return x * 1000000; }
```

operator "" signifies that we are declaring a user defined literal. *__cm, __ms and __mi* are custom suffixes. These operators have a long double parameter. The parameter is multiplied by 10, 1000 or 1000000 and this value is returned. The return value can be any integer or floating point type. Permissible parameter types for user defined literals are described below.

The above definitions allow us to convert meters to centimetres, millimetres and micrometers.

The program below shows how to use user-defined literals:

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

float operator ""__fi(long double x) { return 3.5 * x; }

int main()
{

  float res3 = 4.88__fi;

  cout << res3 << endl;

}

// output
17.08
```

The previous program performs the floating point literal conversion at run-time. We can shift the calculation to compile-time by using *constexpr*. For example:

```
constexpr float operator ""__fi(long double x) { return 3.5 * x; }
```

Restrictions On User Defined Types

User defined literals can only be used with the following integer type parameters:

- const char*
- char
- **std::size_t**

- unsigned long long int
- long double
- float
- const wchar_t*

The return value can be any type.

Note: the *int* type is not allowed as a parameter type.

Naming Rules For User Defined Literals

The first rule is that user defined suffixes must start with an underscore.

The second rule is that user defined suffixes can start with a capital letter after one underscore, but in this case there must be no space between the operator "" and the starting underscore of the user defined suffix.

```
float operator "" _Fi(unsigned long long int value)      // error

float operator ""_Fi(unsigned long long int value)      // ok
```

Modern C++: Template Constants (C++14)

As we have seen, C++ allows template declarations for functions and classes. C++14 also allows template declarations for constants.

We can declare a template constant as follows:

```
template<class T>
T(constant)
```

The expression T(...) has the same syntax and semantics as a C function-style cast (*see function-style casts*).

Here is an example:

```
template<class T>
T pi = T(3.1415926535897932385);
```

pi is a template constant.

A particular template value of pi is defined as: *pi<int>*, *pi<float>*, *pi<double>* and so forth.

If *T* is an integer then the value of pi is 3. If *T* is a float then the value of pi is 3.141592 and if *T* is a double, then it's value is: 3.14159265358979. The type *T* is used to perform a type cast on a constant.

```
#include <iostream>

template<class T>
T pi = T(3.1415926535897932385);

int main() {
      std::cout <<  pi<int> << std::endl;
      std::cout <<  pi<float> << std::endl;
      std::cout <<  pi<double> << std::endl;
```

```
    return 0;
}

// output
3
3.14159
3.14159265358979
```

We can declare and initialize a template constant x as follows:

```
pi<char> x;             // x == char(3.1415926535897932385)
```

Modern C++: Return Type Deduction (C++14)

Return type deduction for functions

C++14 implements automatic type deduction for function return values:

```
auto square(int n)
{
   return n * n;
}
```

The type of the return value of *square* is deduced to be an *int* by the compiler.

Modern C++: Generic Lambdas (C++14)

In C++14 we can specify the type of lambda parameters as *auto*. The compiler will deduce the type of the lambda parameters from the values that the lambda receives. This feature effectively makes a lambda into an anonymous template function:

```
auto my_lambda = [](auto a, auto b) { return a * b; };
```

Here is an example:

```
#include <iostream>

int main() {

    auto f = [](auto a, auto b) {
      std::cout << "type of parameter a: " << typeid(a).name() << std::endl;
      std::cout << "type of parameter b: " << typeid(b).name() << std::endl;
      return a + b;
    };
    auto ret = f(2, 986.546);

    std::cout << "type of lambda return value: ";
    std::cout << typeid(ret).name() << std::endl;

}

// output
type of parameter a: i
type of parameter b: d
type of lambda return value: d
```

Extended Capture In Lambdas

In C++14, variables that are captured in a lambda introducer can be initialized:

```
#include <iostream>

int main() {
   int foo = 30;
   std::cout << "foo before lambda = " << foo << std::endl;
```

```
    auto decrementor = [foo = 2] {
      std::cout << "foo value in lambda: " << foo << std::endl;
      return foo - 12;
    };

    auto ret = decrementor();
    std::cout << "foo value after lambda: " << foo << std::endl;
    std::cout << "value returned by lambda =  " << ret << std::endl;

}

// output
foo before lambda = 30
foo value in lambda: 2
foo value after lambda: 30
value returned by lambda =  -10
```

In this example the variable *foo* is passed by copy to the lambda. Since *foo* is a read-only variable, it's value in the outer scope cannot be changed.

In the example that follows, a new variable is defined in the lambda introducer.

```
#include <iostream>

int main()
{
  auto decrementor =  [foo = 2] { return foo - 12; };
  std::cout << decrementor() << std::endl;
}

// output
-10
```

Since *foo* doesn't exist in the outer scope of the lambda, this is in effect a technique for adding data members to the lambda. A lambda variable in the introducer must be initialized if it does not exist in the outer scope of the lambda. A new lambda variable that is declared in an introducer is read only.

Note: The types of variables in a lambda introducer are not specified since they are inferred by the compiler either because the variable exists in the outer scope or because it is initialized in the lambda introducer.

Modern C++: Constexpr Template Constants (C++14)

C++14 permits template constants to be *constexpr*. These constants are evaluated at compile-time.

```
template<typename T>
constexpr T pi = T(3.1415926535897932385);
```

Note that T(x) is performing a C function-style type conversion of it’s parameter to type T (*see function-style casts*).

```
#include <iostream>

template<typename T>
constexpr T pi = T(3.1415926535897932385);

template<typename T>
constexpr T area_of_circle(T radius)
{
  auto area = pi<T> * radius * radius;
  return area;
}

int main()
{
   auto area = area_of_circle(5.78);
   std::cout << area << std::endl;
}

// output
104.956
```

Modern C++: Constexpr Functions (C++14)

C++14 allows more expressions inside the body of `constexpr` functions:

- local variable declarations are allowed inside constexpr functions provided that they are not *static* or *thread_local* declarations. Local variables must be initialized.

- *if*, *switch*, *for*, *while*, *do-while* expressions are allowed inside *constexpr* functions.

Modern C++: Binary Literals (C++14)

C++14 permits the declaration of binary literals. A binary literal begins with a `0B` or `0b` prefix that is followed by a sequence of 1's and 0's:

```
auto y = 0b00000101;         // integer 5 as a binary (base 2) literal
```

A binary literal has integral type.

Modern C++: Digit Separator (C++14)

C++14 permits integer literals to be separated by apostrophes for better readability:

```
12'345'678'901'245LL
cout << 12'345'678'901'245LL

std::cout << 0b1000'111'0 << std::endl

std::cout<< 0X12A'2b4 << std::endl
```

Modern C++: [[deprecated]] Attribute (C++14)

The *[[deprecated]]* keyword warns a user that an expression, function, class or feature can be used but it’s usage is discouraged. The syntax is:

```
[[deprecated]]
expression, function, class or enum that is deprecated
```

The compiler will emit a warning that the item is deprecated. An optional message can be attached by using the following syntax:

```
[[deprecated(“message”)]]
```

Examples:

```
// identifying the use of a deprecated function
[[deprecated(“postgres sqlConnect function is obsolete”)]]
int sqlConnect() { }


// identifying the use of a deprecated class
#include <iostream>
using namespace std;

int main() {

class  [[ deprecated("Foo is obsolete") ]] Foo {

  public:
    Foo() { cout << "in constructor" << endl; };
    int x;
};

Foo foo;

}

// returns
warning: 'Foo' is deprecated: Foo is obsolete
```

Modern C++: Template Type Deduction (C++17)

Template Argument Type Deduction For Functions

Template argument type deduction refers to the compiler's ability to deduce the types of the arguments of a function template specialization.

In C++ argument type deduction for template functions is implemented as follows. Consider the following function template declaration:

```
template<typename T>
void foo(const T& data) { statements };
```

We can create specializations from this template function as follows:

```
// The first specialization defines a foo function that
// takes an integer argument and then invokes the function

foo<int> foo(5) { statements }

foo<std::string> foo("a string") { ... }
```

The value in the angle brackets instantiates a particular type of *foo* function.

However sometimes the compiler can deduce the types of the arguments in a particular function template specialization from the arguments received by the function. In this case we can simply declare the two *foo* function specializations above as follows:

```
Foo(5) { ... }
foo("a string") { ... }
```

This works as long as the compiler can deduce the types used by a specialization.

In function template argument type deduction, the compiler can instantiate functions with the correct parameter types because it can deduce the types of the arguments from the argument values that the function receives.

Template Type Deduction For Classes

Until C++17, we could not deduce the types of parameters in template classes. So if we wanted to define a vector of type *int*, we would have to do:

```
std::vector<int>  x = {1,2,3,4}
```

That is, the compiler could not read the initialization values and deduce the template type as *int*.

With C++17, the compiler can apply type deduction to class templates, and thus we can specify the previous declaration as:

```
std::vector x = {1,2,3,4}
```

This will instantiate a vector of type *std::vector<int>*. The compiler examines the constructor arguments to deduce the template types used in a class specialization.

As another example, consider the declaration of a tuple prior to C++17. We could define a tuple in either of the following two ways:

```
std::pair< int, const char* >  foo { 1, “str” };

or:

auto foo = std::make_pair(1, “str”);
```

In the second declaration, the *make_pair* function returns a *std::pair tuple<int, const*

char>*.

With C++17, these *make* functions have become obsolete since we can simply write:

```
std::pair my_pair = { 1, “str” }
```

The compiler can deduce that the template class parameters are *int* and *const char** respectively by examining the constructor for *my_pair* and then the compiler can create a *std::pair* specialization with these types.

In general, compiler type deduction works automatically for a class template when the class has constructors with parameters that mention all of the template parameters.

Template Argument Deduction Guides

Look at the following example:

```
template <typename A, typename B>
struct MyPair
{
  MyPair() { }
  MyPair(const A&, const B&) { }
};

int main()
{
  MyPair p1 {11, 22};
  MyPair p2;
}
```

The compiler will throw an error when it tries to instantiate p2, since the constructor for p2 does not provide any information about the types A and B which are required by the constructor of p2 to create the object.

Modern C++: Non-Template Type Parameters (C++17)

In C++11 and C++14 we can use the type *auto* whenever we can automatically deduce the types of template parameters or function return types. *auto* can also be used in generic lambdas.

Until C++17 the types of non-template parameters had to be explicitly declared in a class template or function template since the compiler would not deduce the types of these parameters.

Summary Of Non Template-Type Parameters

The following is an example of using a parameter which is not a template type in a template:

```
#include <iostream>

// size is a non-template type parameter
template<class T, int size>
class Foo
{
  T array[size];

 public:
   Foo() {}
   ~Foo() {};

   int getSize() const
   {
     return size;
   };

};

int main()
{
  Foo<double, 20>  foo;
  auto size = foo.getSize();
  std::cout << size << std::endl;
}

// output
20
```

In this example, the non-template type parameter *size* is part of the template declaration. The value 20 is substituted into a Foo class specialization at compile-time.

A non-template type parameter is a special type of parameter that is replaced by a concrete value upon specialization of a class template or function template.

C++ requires template parameters that are not template type parameters to be one of the following types:

- **A value that has an integer type or is an enumeration**
- A pointer or reference to a class object
- A pointer or reference to a function
- A pointer or reference to a class member function
- A nullptr

Note that we cannot specify double or float types as non-template parameters.

Auto Non-Template Parameters

In C++17, we can declare a non-template parameter in a template declaration as type *auto*, providing that we can deduce the type of this non-template parameter in a specialization.

```
template <class T auto J>
void func(T t, x) { }

func(“string”, 10) { };      // deduces J as an int
```

In this example, *auto* is deduced to be an *int* type

Modern C++: Nested Namespaces (C++17)

Consider the nested namespace:

```
namespace A {

  namespace B {

    namespace C {

        int x = 100;
    }
  }
}
```

In C++17, we can specify the nested namespace C as:

```
namespace A::B::C {
    int x = 100;
}
```

Here is an example:

```
#include <iostream>

namespace A {
   int a = 1;
}
namespace A::B {
  int b = 2;
}
namespace A::B::C {
  int c = 3;
}

int main()
{
  std::cout << A::a << " " << A::B::b << " " << A::B::C::c;
  std::cout << std::endl;
}

// output
1 2 3
```

Modern C++: Constexpr If Statements (C++17)

Suppose that we have an *if* statement that can be evaluated at compile-time, then C++17 lets us discard branches of this *if* statement at compile-time based on the evaluation of the constant *if* expression. This is done using the following syntax:

if constexpr(condition)

```
if constexpr(condition)
     statement1;       // Discarded if condition is false
else
     statement2;       // Discarded if condition is true
```

Here is an example (this example uses *type traits*):

```
// the std::is_integral<T> predicate function evaluates to true
// if T is an integer type and is false otherwise

#include <iostream>
#include <type_traits>

template <typename T>
constexpr bool IntegralTest()
{
  if constexpr (std::is_integral<T>::value) {
    return true;
  }
  else {
    return false;
  }
}

int main()
{
  static_assert(IntegralTest<int>() == true);
  static_assert(IntegralTest<char>() == true);
  static_assert(IntegralTest<char*>() == true);
}

// output
error: static assertion failed
```

Modern C++: Initialization Of Auto Types (C++17)

See first: std::initializer_list

Initialization of variables in C++ can have several forms:

default initialization	std::string s;
default braced initialization	std::string s {};
direct initialization	std::string s(“hello”);
copy assignment initialization	std::string s = “hello”;
braced list initialization	std::string s {‘h’, ‘e’, ‘l’, ‘l’, ‘o’};
aggregate initialization	std::string s[5] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’};

reference initialization	std::string s[5] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’};
	char& c = s[1];

C++11 introduced a general syntax for initialization with initializer lists, often referred to as *braced list initialization*. There are two types of braced list initialization:

- direct list initialization: T object {arg1, arg2, ...};
- copy list initialization: T object = {arg1, arg2, ...};

Under C++17, type deduction for brace-initialized auto variables follow the following rules for type deduction:

- for direct list initialization, the compiler will deduce a type T for the auto type variable if the initialization list has a single element of type T. It will be be ill-formed if there is more than one element in the initialization list.
- for copy list initialization, the compiler will deduce the type of the auto variable

from *std::initializer_list<T>* if all of the elements in the list have the same type, otherwise the auto type variable will be ill-formed.

Because of these rules, the examples above become:

- auto b {42};   // ok, int is deduced
- **auto d {1, 2};   // error, too many initializers**
- auto a = {42};   // ok, std::initializer_list<int>
- auto c = {1, 2};   // ok, std::initializer_list<int>
- auto d = {1, 2.13};   // error, initializer list has different types

```
#include <iostream>

int main()
{
  auto tmp {1, 2};
  std::cout << tmp << std::endl;
}

// output
error: direct-list-initialization of 'auto' requires exactly one
element
```

Modern C++: Structured Bindings (C++17)

Structured bindings assign values to scalar variables from the public data members of an object such as a struct, class, array or a tuple object.

The syntax for a structured binding is:

```
auto [myvar1, myvar2, myvar3] = some_object;
```

In the example below, we bind the elements of an array to three **scalar** variables:

```
#include <iostream>

int main()
{
  int arr[] = { 1, 2, 3 };
  auto [foo1, foo2, foo3] = arr;

  std::cout << foo1 << " " << foo2 << " " << foo3 << std::endl;

  return 0;
}

// output
1 2 3
```

In the following example, we bind three scalar variables to the members of a struct:

```
#include <iostream>
#include <string>

struct
{
  int x = 9;
  std::string s = "hello world";
  double y = 5.37;
}  foo;

int main()
{
  auto [bar1, bar2, bar3] = foo;

  std::cout << bar1 << " " << bar2 << " " << bar3 << std::endl;
}

// output
9 hello world 5.37
```

Here, *bar1*, *bar2* and *bar3* have their types *int*, *string* and *float* auto deduced and initialized from the struct *foo*.

We can use qualifiers such as const and reference in a structured binding. For example, if we declare the following structured binding:

```
auto& [bar1, bar2, bar3] = foo;
```

then any change to *foo* will effect the variables *bar1*, *bar2* and *bar3* and vice versa.

Using Structured Bindings

Structured bindings can be used with the public data members of classes, structs, raw C-style arrays, and tuple objects:

- In structs and classes each non-static public data member must be bound to exactly one unique variable.
- For arrays, we must bind each array element to a unique variable.

In all cases, the number of elements or non-static public data members must equal the number of scalar variables in the declaration of the structured binding. It is not permissible to skip a data member or use a scalar variable twice.

```
int s[3] {1,2,3}
auto [a, b, c] = s;          // OK

auto [x, y] = s;             // Error: not enough scalar variables
auto [x, y, x] = s;          // Error: variable x is used twice
```

Nested decomposition is not supported.

Structured Binding For Arrays

```
// braced initialization of an array
int arr[] = { 47, 11 };

 // structured binding of x and y to the array
auto [x, y] = arr;

// error: all of the elements of arr are not bound
auto [z] = arr;
```

Inheritance And Structured Bindings

All of the non-static public data members of a class used in a structured binding must be members of the same class. Structured bindings do not support inheritance.

```
#include <iostream>
struct Foo
{
  int a;
  int b;
  Foo(x, y) {a = x; b = y;)
};

struct Bar : Foo
{
  int c;
  Bar(int val) : Foo(1, 2) { c = val; };
};

int main()
{
   Foo foo {1, 2};
   auto [i, j] = foo;              // OK
   Bar bar(3);


   auto [i, j, k] = bar{};        // Compile-Time ERROR
}

// output
error: cannot decompose class type 'Bar': both it and its base class
'Foo' have non-static data members
```

Modern C++: Initialization In If And Switch Statements (C++17)

As a general principle, we should always restrict the scope of variables as much as possible.

In C++17, the *if* and *switch* syntax permit us to specify an initialization clause aside from the usual condition or switch clause. This will narrow the scope of the initialized variable to the control structure. For example, we can write:

```
if (int status = check(); status != 0) {
      return status;
}

this has a narrower scope than:

int status = check();
if (status != 0) {
     return status;
}
```

The initialization expression in this *if* statement is:

```
int status = check();
```

The *status* variable has scope from the beginning of the definition to the end of the *if* block, including an *else* block, if present.

Here is a complete example:

```
#include <iostream>

int test(int x)
{
  if (x >= 0) return 1;
  return -1;
}

int main()
{
  int y = 5;

  if (auto x = test(y); x != -1)
    std::cout << "x = " << " " << x << std::endl;
  else
    std::cout << "x = " << " " << x << std::endl;

}

// output
x =  1
```

Note: Compound initialization is prohibited:

```
if (int y = -2; auto x = test(y); x + y  != -1) {....}   // error
```

Switch Statement With Initialization

Using an initialization clause in a switch statement allows us to initialize a variable for the scope of the switch:

```
#include <iostream>

int test(int x)
{
  if (x == 0) return 0;
  else if (x > 1) return 1;
  else if (x <= 1) return -1;
  return 0;
}

int main()
{
  switch(int x = -5; x = test(x))
  {
     case -1:
       std::cout << "switch = " << -1 << std::endl;
     break;
     case 1:
       std::cout << "switch = " << 1 << std::endl;
       break;
     default:
       std::cout << "switch = " << 0 << std::endl;
  }

}
```

```
// output
switch = -1
```

Modern C++: Inline Variables (C++17)

One of the best features of C++ is its support for header-only libraries. In a header-only library, the header definitions as well as the implementation are all contained in a single header file. However, up to C++17, a header only library could not be created if the library had global variables or objects. This restriction was a consequence of the *one definition rule.*

According to the **one definition rule (ODR)**, a variable or entity must be defined only once. That is, the variable must be defined in only one translation unit. Thus if a header only library defines a global variable, then this global variable will be included in all the implementation files (*.cpp*) that include the single header library. In other words, we will have multiple definitions of the global variable. This is an error. For the same reason, a static member variable in a class will potentially have multiple definitions unless it is also declared as a *const*. The *inline* variable attribute in C++17 solves this problem by ensuring that a global variable in a header file is defined only once.

Beginning with C++17, we can define a global variable in a header file to be *inline*. Then if this header file is used in multiple translation units, the compiler will ensure that there is only one unique definition of the inline variable. The syntax for an inline variable is:

```
inline int x = 0;

class Foo
 {
 public:
 static std::string greet = "hello world";

 };

inline class Foo foo;
```

Using Inline Variables

In C++17, the *inline* keyword permits us to define a global variable in a header file, even though this header file might be included in multiple implementation (*.cpp)* files. The inclusion of the header file a multiple number of times does not create multiple definitions of the global variable.

Any initialization of an inline variable will be performed by the first translation unit that includes the header file.

```
// some header file

class Foo
{
public:

  // OK since C++17
  inline static std::string greet = "hello world";

};

// OK even if included in multiple implementation (.cpp) files
inline Foo foo;
```

Modern C++: Lambda Extensions (C++17)

C++17 expands the use of lambdas as follows:

- lambdas can be constexpr
- a lambda can access the *this* pointer of an object

constexpr Lambdas

C++17 permits a lambda to be used in a compile-time context, provided that it can be evaluated at compile-time.

To specify that a lambda can be evaluated at compile-time, we declare it with the *constexpr* attribute:

```
// OK since C++17

auto squared = [](auto val) constexpr {
     return val*val;
};
```

constexpr lambdas cannot have any static variables, try/catch blocks or allocate memory.

A compiler error will occur if this *constexpr* lambda cannot be evaluated at compile-time. The following example demonstrates this:

```
#include <iostream>
#include <string>

int main()
{
    int value = 9;
    auto summand = [](int val) constexpr
    {
        static int multiplier = 2;
        return val*multiplier;
    };

    int sum = summand(value);

    std:: cout << sum << std::endl;

  return 0;
}

// output
error: 'multiplier' declared 'static' in 'constexpr' function
```

Accessing The This Pointer In A Lambda

Prior to C++17, a lambda declared in a member function did not have access to the *this* pointer of the object. C++17 rectifies this deficiency:

```
#include <iostream>
#include <string>

class Foo
{
   std::string name;
   public:
     Foo(const std::string&& nm)
     { name = nm; }

     void bar() {
       auto print = [this] { std::cout << this->name << std::endl; };
       print();
     }
};


int main() {
  Foo foo(std::move("daffodil"));
  foo.bar();
}

// output
daffodil
```

In the example above, the this pointer to the object is captured in the lambda introducer. A lambda can also capture ***this** in the introducer, this means that a copy of the current object is passed to the lambda.

Modern C++: Type Includes The Exception Spec (C++17)

Since C++17, exception handling specifications have become part of the type of a function. That is, the following two function declarations are now two different types:

```
void foo(int);
void foo(int) noexcept;                // different type, see below
```

Prior to C++17, both functions would have been the same type.

The compiler will enforce correct exception semantics for function pointers:

```
// Note: we cannot define f1 and f2 together in the same namespace
// since f1 and f2 have the same signatures even though they are
// different types

void f1();
void f2() noexcept;

// define a pointer to a function that doesn’t throw an exception
void (*funcptr)() noexcept fp;

// this assignment is OK, since the pointer does not violate exception
// semantics
fp = f2;

// Error since C++17, the pointer is trying to point to
// a different type of function
fp = f1;
```

We cannot overload a function name with the same signature and different exception semantics in the same scope:

```
void f3(int x);
void f3() noexcept;                // Compiler Error
```

Consider this example:

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

int func(int a, int b) noexcept
{
    return a + b;
}


int main()
{
  func(1, 3);

  return 0;
}

// error: redefinition of 'int func(int, int)'
```

Modern C++: Empty Enum Initialization (C++17)

C++11 implements strong enum type safety with scoped enums (*enum class*). A scoped enum is a distinct type that does not implicitly convert to an integer type.

C++17 permits empty braced initialization for scoped enums. These enums do not have any enumerators:

```
#include <iostream>
#include <string>

int main()
{
  // scoped enum does not declare any enumerators
  enum class CAPITAL { };

  // initialize
  CAPITAL capital { 3000 };
  CAPITAL kapital { 5000 };

  if (capital == kapital)
     std::cout << "capital = 5000" << std::endl;
  else
     std::cout << "capital != 5000" << std::endl;

  return 0;
}

// output
capital != 5000
```

Braced initialization always prohibits narrowing conversions for any type:

```
#include <iostream>
#include <string>

int main()
{
  enum class CAPITAL:int { };

  // cannot convert long int to int
  CAPITAL capital { 3000000000000000 };            // error
  CAPITAL kapital { 5000 };
  if (capital == kapital)
     std::cout << "capital = 5000" << std::endl;
  else
     std::cout << "capital = 3000" << std::endl;
}

// output
error: narrowing conversion of '3000000000000000' from 'long int' to
'int'
```

Modern C++: Fold Expressions (C++17)

A fold expression applies a binary operator to all of the elements of a parameter pack. Consequently, we can perform binary operations on the elements of a parameter pack without having to resort to recursion.

We can use all of the binary operators with fold expressions except for: ., ->, and [].

In the following example, the function *foldSum* receives a parameter pack argument and returns the sum of all of it's arguments:

```
#include <iostream>
#include <string>

template<class ... Ttype>
  auto sum(Ttype ... args) {
        auto sum = (... + args);
        std::cout << "sum = " << sum << std::endl;
        return sum;
  };

int main()
{
  sum(1, 5, 11, 18, 9);
}

// output
sum = 44
```

(... + args) is called a *fold expression*, it applies the + operator to all of the parameter pack arguments starting from the left.

The parentheses around the fold expression is mandatory..

Using Fold Expressions

Let *operator* be some binary operator, then:

- The following syntax implements a left fold, meaning that it starts processing the parameter pack arguments from the left:

  (... operator args)

  This expands to: ((arg1 operator arg2) operator arg3) operator . . .

- The following is the syntax for a right fold, meaning that it starts processing the parameter pack arguments from the right:

  (args operator ...)

  This expands to: ((argN operator argN-1) operator argN-2) operator ....

The parentheses are part of the fold expression and are required.

Handling Empty Parameter Packs

For an empty parameter pack, the following rules apply:

- If the binary operator && is used, the fold expression is true.
- If the binary operator || is used, the fold expression is false.
- For all other operators the call is ill-formed.

Modern C++: string_view (C++17)

The C++17 Standard Library implements a new string class called *std::string_view*. This class allows us to interact with character sequences and strings without allocating memory for them. A *string_view* object provides a view of some other character sequence or string. In particular a *std::string_view* object refers to a character sequence or string without owning it.

In contrast to *std::string, std::string_view* objects have the following properties:

- The underlying character sequence or string is read-only.
- Character sequences are not guaranteed to be null terminated. That is, a *string_view* of a char sequence may not be null terminated.
- A *string_view* can be a *nullptr*. *nullptr* is returned by the *data*() member function when a *string_view* is created with the default constructor.

Because there is the possibility that a *string_view* object does not have a null terminator, we must always use *size()* before accessing characters through the [] *operator* or the *data()* member function. The member function *data()* returns a pointer to the underlying *string_view* array.

Here is an example:

```
#include <iostream>
#include <string>
#include <string_view>

int main()
{
  std::string str("hello world");
  std::string_view sv(str);

  std::cout << sv << std::endl;
  std::cout << sv[6] << std::endl;

}
```

```
// output
hello world
w
```

Modifying the underlying string can give rise to undefined behaviour in a *string_view* (this will occur when memory in the string is re-allocated):

```
#include <iostream>
#include <string>
#include <string_view>

int main()
{
  std::string str("hello world");
  std::string_view sv(str);

  std::cout << sv << std::endl;
  std::cout << sv[6] << std::endl;

  str = str + " goodbye";             // memory reallocated

  std::cout << str << std::endl;
  std::cout << sv << std::endl;
  std::cout << sv[6] << std::endl;

}

// output
hello world
w
hello world goodbye
rld
```

string_view And Scope

A std::string_view's lifetime is dependent on the string that it is viewing. If the viewed string goes out of scope, *std::string_view* has nothing to observe and accessing it causes undefined behaviour. Thus a *string_view* object does not extend the lifetime of the string being viewed.

Consider the following code:

```
#include <iostream>
#include <string>
#include <string_view>

std::string_view greet()
{
  std::string str {};
  str = "hello";

  std::string_view view {str};
  return view;

} // str is deallocated, and so the string_view of str does not exist


int main()
{
  std::string_view view {greet()};

  // view is observing a string that has already died.
  // Undefined behavior
  std::cout << "greeting is " << view << '\n';

  return 0;
}

// output
greeting is
```

Best Practise

When a function declaration specifies a const string as a function parameter use a *string_view* instead. Example:

```
Instead of:

int ret = foo(const string mirror) { ... }

use:

int ret = foo(string_view mirror) { ... }
```

The *string_view* will not call a constructor and allocate memory for a new string.

A string_view Anti-Pattern

Consider the following code fragment:

```
#include <iostream>
#include <string>
#include <string_view>

int main() {
  std::string s = "Hello ";
  std::string_view sv = s + "World\n";
  std::cout << sv;
}
```

This will create a temporary right value: *s + "World\n"*. This right value will be copy assigned to the *string_view sv*. The problem is that the right value will disappear immediately after the assignment.

Modern C++: Initialization Of Aggregates (C++17)

An aggregate is an array or a simple C++ class that has the following properties:

- no user-declared or explicit constructor.
- no inherited constructor.
- no virtual member functions.
- All non-static data members are public.

For arrays and classes that satisfy these constraints we can initialize a class instance or array using brace initialization:

```
#include <iostream>
#include <string>

int main()
{
  struct Foo
  {
    int x;
    double y;
    std::string z;
  };

  Foo foo  { 5, 0.99, "hello world" };

  std:: cout << foo.x << " " << foo.y << " " << foo.z << std::endl;

  return 0;
}

# output

5 0.99 hello world
```

The sequence of the initializers in the braces must match the declaration order of the data members of the aggregate class.

In C++17, aggregate derived classes that derive from aggregate base classes, can be initialized through brace initialization:

```
#include <iostream>
#include <string>

int main()
{
  struct Foo
  {
    int x;
    double y;
    std::string z;
  };

  struct Bar : Foo
  {
    float p;
  };


  Bar bar;

   bar = { {5, 0.99, "hello world"}, 3.12 };

  std::cout << bar.x << " " << bar.y;
  std::cout << " " << bar.z << << “ ” << bar.p << std::endl;

  return 0;
}

//output

5 0.99 hello world 3.12
```

Notice that aggregate class initialization supports nested braces to pass values to the implicit base class constructor. If the derived class has only one data member then the inner braces can be omitted.

Zero Initialization Of Aggregates

Consider the case where we use brace initialization to initialize an aggregate but skip giving initialization values to some of the members of the base or derived aggregate. In this case, the compiler will initialize the missing elements to the zero values corresponding to their types. For example:

```
struct Foo
{
  const char* name;
  double value;
};

struct Bar : Foo
{
  std::string state;
  void print() const
  {
    std::cout << ' [ ' << name << ' , ' << state << "]\n";
  }
};
```

- Bar bar {};   // zero-initialize all elements
- Bar b{{"me"}};   // same as {{"me", 0.0}, “”}
- Bar b{{}, “hello};   // same as {{nullptr, 0.0}, “hello”}
- Bar d;   // error: values of fundamental types are unspecified

Modern C++: [[nodiscard]] Attribute (C++17)

Attributes allow us to express additional constraints, give the compiler additional optimization possibilities or give the developer additional information. We can use attributes on types, variables, classes, functions, identifiers and lambdas.

An attribute is specified by enclosing a keyword in a pair of brackets: *[[keyword]]*

Multiple attributes can be specified in any of the following ways:

```
[[attribute1]] [[attribute2]] [[attribute3]]
int func1();

or:

[[attribute1, attribute2, attribute3]]
int func2();
```

The [[nodiscard attribute]]

The **[[nodiscard]]** attribute informs the compiler that the return value of a function must not be ignored. The compiler will issue a warning if the return value is not used subsequently.

Here is an example showing how [[nodiscard]] is used with a function implementation:

```
[[nodiscard(“return value of sqlAppend cannot be discarded”)]]
sqlAppend() {
    ....
    return ret;
}
```

Note: the custom error string is only supported in C++20.

Using [[nodiscard]] With Scoped Enums

We can also use the *[[nodiscard]]* attribute with scoped enums:

```
enum class [[nodiscard]] result {ok, no_data, error};
```

This means that any function that returns a scoped enum *result* value cannot have it's return ignored.

Modern C++: [[maybe_unused]] Attribute (C++17)

Consider the function declaration: *int ret = foo(int x, int y);* where the parameter *y* never receives an argument value. Though this declaration is syntactically correct, the declaration is bad practise since it introduces ambiguity into the meaning of the function and the reason why *y* is included as a parameter in the declaration.

However there may be cases where we expect y to receive a value only infrequently or in edge cases. For example *y* may receive a value in certain template specializations, in some conditional compilation cases, or due the nature of the application domain.

The attribute *[[maybe_unused]]* informs the compiler and the programmer that *y* will receive a value in certain infrequent cases.

[[maybe-unused]] can be used to annotate the *foo* function as follows:

ret = foo(int x, ***[[maybe_unused]]*** *int y);*

Modern C++: __has_include Preprocessor Directive (C++17)

The *__has_include* preprocessor directive determines whether an include file is available:

```
#if __has_include(<optional>)
  #include <optional>
#else
  #include <experimental/optional>
#endif
```

This directive is useful to determine if an include file is available for a particular compiler version or with a particular operating system.

Modern C++: The Spaceship Operator (C++20)

The spaceship operator <=> (also called the three-way comparison operator) can be used to do the following types of comparisons: a < b, a <= b, a == b, a != b, a > b and a >= b. The return type of the spaceship operator is type *bool*.

Suppose that we have the following struct:

```
struct MyInt {

   int value;
   MyInt(int val) { value = val };

};
```

Now suppose, that we want to compare two *MyInt* objects, we will have to at least implement the operator function *operator<* for this struct:

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

struct MyInt {

   int value;
   MyInt(int val) { value = val; };
   bool operator<(MyInt rhs) { return value < rhs.value; };
};

int main() {

  MyInt myint1(10), myint2(0);

  bool ret = myint1 < myint2;           // myint1.operator<(myint2)
  cout << boolalpha << ret << endl;

}

// output
false
```

In fact, we may need to code all of the comparison operators *MyInt::operator<, MyInt:: operator<=, MyInt::operator==, MyInt::operator!=, MyInt::operator> and MyInt:: operator>=.*

We can request the compiler to generate all six comparison operators: == , != , < , <= , > and >= by declaring a default three-way comparison (spaceship) operator for the class.

In the following example, the compiler automatically generates all of the comparison operators for *MyInt*:

```
#include <iostream>
#include <compare>

using namespace std;

struct MyInt {

   int value;
   MyInt(int val) { value = val; };
   bool operator<=>(const MyInt& rhs) const = default;
};

int main() {

  MyInt myint1(10), myint2(0);

  bool ret = myint1.value < myint2.value;
  cout << boolalpha << ret << endl;

}

// output
false
```

Note: The *operator<=>* must be declared as a *const* function that takes a reference parameter.

The compiler treats <=> as an implicitly *constexpr* function and *noexcept.*

<=> performs lexicographic comparisons. For objects, lexicographical comparison,

means that all base classes are compared left to right and for each class all non-static members of the class are compared lexicographically in their declaration order.

The following example uses the spaceship operator in a class template

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

template<typename T>
struct MyObject {
   T value;
   MyObject(T val) : value(val) {};
   bool operator<=>(const MyObject& rhs) const = default;
};

int main() {

  MyObject<float> myobject1(10.5), myobject2(30.4);
  bool ret = myobject1.value < myobject2.value;
  cout << boolalpha << ret << endl;

  MyObject<const char*> strobject1("hello"), strobject2("hello");
  ret = strobject1.value < strobject2.value;
  cout << boolalpha << ret << endl;

}

// output
true
false
```

Modern C++: consteval And constinit (C++20)

C++20 has two new keywords: *consteval* and *constinit* . A function that is declared as a *consteval* function is always executed at compile-time.

consteval Functions

The syntax for a *consteval* function is:

```
consteval int multiply(int n) {
   return n*n;
}
```

All of the dependencies of a *consteval* function must resolve to constants at compile-time. Thus a *consteval* function is also a *constexpr* function. *consteval* functions are sometimes called immediate functions.

Note: a function that allocates or deallocates memory cannot be a *consteval* function.

A *constexpr* function can be evaluated at compile-time or run-time depending upon whether it's arguments are compile-time constants. A *consteval* function must always be evaluated at compile-time.

Consider:

```
consteval int multiply(int n) {
   return n*n;
}


int main() {
  multiply(5);

  const int x = 10;
  multiply(x);

  int z = 5;
```

```
    multiply(z)
  }
```

Both 5 and x are compile-time constants, so they can be used in the *consteval* function. z is not a compile-time constant, therefore *multiply(z)* will not compile.

constinit Variables

Global variables, static variables, static class members and thread-local variables have static storage duration. These objects are allocated when the program starts and deallocated when the program ends.

A variable with static storage duration may be initialized at (i) compile-time, or (ii) when the declaration is encountered at run-time (for example when a static variable is inside a function block).

A *constinit* variable is a variable with static storage that is guaranteed to be initialized at compile-time. The constinit attribute can only be applied to static storage variables.

A *const* variable is not necessarily a *constinit* variable since it does not have have to be initialized at compile-time.

constexpr try-catch Blocks

C++20 allows *try-catch* blocks inside *constexpr* functions but *throw* is not allowed. The catch block in a *constexpr* function is simply ignored at compile-time.

```
constexpr void f(A_t x, A_t y){

    try{
         bool ret = (x == y);
        // ...

    }
```

```
        catch(...){     // ignored at compile-time
            // ...
        }
    }
```

std::is_constant_evaluated()

std::is_constant_evaluated() is a type trait that permits us to check whether a call to a *constexpr* function occured at compile-time:

```
#include <iostream>
#include <type_traits>
#include <string>

constexpr const char* getEval(int x){
    x = x;
    if(std::is_constant_evaluated()) {
        return "compile-time evaluated";
    }
    return "run-time evaluated";
}

int main()  {
  const int x = 5;
  std::string ret = getEval(x);
  std::cout << ret << std::endl;
  return 0;
}

Output
run-time evaluated
```

asm Declaration In constexpr Functions

C++20 permits *asm* code to appear inside *constexpr* functions. This allows compile and run time *constexpr* functions to have *asm* code inside a single function:

```
    constexpr int add(int a, int b) {

        if (std::is_constant_evaluated()){
            return a + b;
        }
        else{
            asm("asm code");
            //...
        }
    }
```

Modern C++: Initializers In Range-Based Loops (C++20)

C++20 permits initializers to be used in range-based loops:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    for (vector<int> vec {1, 2, 3}; auto v : vec)
        {
          cout << v << " ";
        }
}

// output
1 2 3
```

Modern C++: Functions With auto Parameters (C++20)

In C++20, *auto* can be used in the parameter list of a class member function or free function. The function automatically becomes a function template and the *auto* parameter becomes a template parameter.

Modern C++: Non-Parameter Types In Templates (C++20)

Prior to C++20, non-parameters in templates had to be one of the following types:

- integers
- enumerators
- pointers or references
- nullptr

C++ permits floating point and literal type non-template parameters. The most important literal type is a string literal.

Modern C++: Virtual Constexpr Functions (C++20)

A *constexpr* function will execute at compile time if all of it's arguments are compile-time constants. A *constexpr* function will be executed at run time if some of it's arguments do not exist at compile-time or are not compile-time constants.

C++20 permits a *constexpr* member function to be virtual. A virtual *non-constexpr* function can override a *constexpr* virtual function, and a virtual *constexpr* function can override a virtual *non-constexpr* function.

Here is a simple example:

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

struct X1
{
   virtual int f() const = 0;
};


struct X2: public X1
{
    constexpr int f() const override { return 2; }
};


struct X3: public X2
{
    int f() const override { return 3; }
};


struct X4: public X3
{
    constexpr int f() const override { return 4; }
};

int main()
{
   X2  x2;
   X3  x3;
   X4  x4;

   cout << "x2: " << x2.f() << "  x3: " << x3.f() << "  x4: " << x4.f();
}

// output
x2: 2  x3: 3  x4: 4
```

A *consteval* virtual function cannot override a *constexpr* virtual function. A *constexpr* virtual function can override a *consteval* virtual function.

Modern C++: Concepts (C++20)

There are two approaches to compile-time polymorphism. Firstly, we can explicitly code all of the overloaded functions that we need. Alternatively, we can define generic function templates and generic class templates. Both of these approaches are problematic. The first approach is tedious and error-prone. The second approach is too expansive because we may want to prohibit some template specializations.

C++20 introduces concepts to handle the previous issues. Concepts are predicate expressions that constrain the template parameters of a class or function template. These expressions are evaluated at compile time to determine whether template parameters satisfy the given constraints. Only template specializations that meet the constraints imposed by a concept are permissible.

Definition (concept): A concept is a named compile-time constraint on template types such as their supported operations and semantics.

There are a number of syntactical constructs available to implement concepts; the following syntax is one of the ways to specify a concept:

```
template<typename T, typename X, ...>
concept concept_name = <constraint>
rest of the template declaration;
```

Concepts have the following advantages:

- The requirements imposed on template parameters are part of the declaration. This means that concepts are self-describing.
- We can use predefined concepts or define our own concepts
- If a function declaration uses a concept, it automatically becomes a function template.

The Four Ways To Specify A Concept

There are four ways to implement concepts. These are discussed below.

Preface A Typename With A Concept

In the template declaration, we preface a typename with a concept:

```
#include <concepts>
#include <iostream>

template<std::integral T>
T add(T a, T b) {
return (a + b);
}
```

Here the constraint on the template parameter T is part of the type declaration itself. This makes the declaration of the concept in the *add* function template self-documenting.

Note: *std::integr*al is a predefined type trait that is true only if T is an integer type.

Implement Concepts With A Requires Clause

```
#include <concepts>
#include <iostream>

template<typename T>
requires std::integral<T>
T add(T a, T b) {
  return (a + b);
}
```

This syntax uses the requires keyword in conjunction with a pre-defined concept. The syntax for custom user-defined concepts is different.

Implement Concepts With A Trailing Requires Clause

A concept can be specified in a trailing requires clause:

```
#include <concepts>
#include <iostream>

template<typename T>
T add(T a, T b) requires std::integral<T> {
  return (a + b);
}
```

Here the concept is specified as an attribute of the generic function.

Implement Concepts With Abbreviated Function Templates

```
#include <concepts>
#include <iostream>

auto add(std::integral auto a, std::integral auto b) {
  return (a + b);
}
```

Notice that the *template<typename T>* portion of the declaration is missing. *add* is automatically a function template because the concept constraints are attached to the parameters of the *add* function.

The Requires Clause

The keyword *requires* introduces a clause which specifies constraints on template parameters. *requires* must be followed by a concept name and then a constraint:

```
concept concept_name = <constraint>
```

The constraint can be omitted if we are using a concept that is predefined by the standard library.

Applying Concepts To The Return Type Of A Function

The implementation is simple, we preface the return type of the function template with a concept. For example:

```
#include <concepts>
#include <iostream>

std::integral auto add(std::integral auto a, std::integral auto b) {
      return (a + b);
}
```

Using More Than One Concept

Multiple concepts can be joined together using conjunction (&&) and disjunction (||):

```
template<typename Iter, typename Val>
requires std::input_iterator<Iter>
&& std::equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
```

Valid join operators are: *and (&&)*, *or (||)*, and *not (!)*.

Writing A Custom Concept

The syntax for a custom concept is:

```
template<typename T, typename X, ...>
concept concept_name = <constraint>
```

The constraint expression has the following syntax:

```
requires (optional_parameter_list) { requirements; ...;}
```

The optional_parameter_list is a comma-separated list of template parameters. The variables specified in this list are used to specify compile-time constraints on the parameters. They don't have any lifetime, initialization, storage, and so forth. If no variables are required, the parentheses can be omitted.

Example:

```
template<typename T>
concept addable = requires (T a, T b) { a + b; };
```

This concept requires that we must be able to add the template parameters.

Validate That A Template Specialization Can Compile

To verify that type parameters can be used as intended by a class or function template specialization, we introduce variables in the optional parameter list and write the intended usage as a requirements expression. This is exactly what the previous *addable* concept does.

Once we have declared the concept *addable*, we can complete our template declaration for the *add* function is follows:

```
template<addable T>
T add(T a, T b);
```

Note: the a + b in the requirements expression is not executed; the compiler only verifies if it can be compiled successfully.

Constrain A Template Parameter Type To Types With Certain Properties

A requirements expression can be used to verify that a type has a required property. For example:

```
template <typename T>
concept is_vectorizable = requires { typename std::vector<T>; };
```

This concept only permits specializations where type T is a vector type:

Here is a complete function template specification using the vectorizable concept:

```
Template <typename T>
concept is_vectorizable = requires { typename std::vector<T>; }
std::vector vector_add(is_vectorizable a, is_vectorizable b) { };
```

Constrain Template Parameters To Boolean Expressions That Evaluate To True

It is possible to evaluate boolean expressions inside a requires clause. The following syntax is used for evaluating logical expressions:

```
requires { boolean_expression };
```

As an example, consider a concept that checks if the size of the template parameter T is equal to 4 bytes. The syntax for this is:

```
template <typename T>
concept C1 = requires {requires sizeof(T) == 4;};
```

or we can also write:

```
template <typename T>
concept C1 = ( sizeof(T) == 4 );
```

Note that the following does not work:

```
template <typename T>
concept C2 = requires { sizeof(T) == 4; };
```

C2 doesn’t actually execute *sizeof(T) == 4*. It just verifies whether the statement compiles. Therefore *C1<int_64t>* will return *false*, but *C2<int_64t>* will return *true*.

Constrain auto With Concepts

We have already seen an example of this previously. We constrain an auto type by prefacing the auto declaration with the name of the concept. For example:

```
std::unsigned_integral auto foo {1};
```

This syntax can be used anywhere where *auto* can be used, including lambdas.

In the example above the type of `foo` is constrained to an unsigned integral type. Since *foo* is initialized to 1, which by default is a *signed int*, the compiler will throw an error for this declaration.

Modern C++: Modules (C++20)

Frailties In The Traditional Compilation Process

An implementation file typically needs one or more header files before compilation can proceed. These header files contain the declaration of the functions used in the implementation file. The compiler pre-processes the implementation file by first including the header files specified in the implementation file (for example, *#include <iostream>*). After this, the compiler performs any macro substitutions specified directly or indirectly in the implementation file. The result is a translation unit. This translation unit is then compiled by the compiler into an object file.

There are a number of problems with this build process. Firstly, a header file can be included a multiple number of times since it can be included in more than one translation unit. Consider for example the header file *iostream* which contains over half a million lines of code. It is redundant to include it in several translation units.

Secondly problems may arise because of the one definition rule. This rule states that a variable or a function can be defined only once. If a variable or a function is defined differently in two header files, the inclusion of these header files during the compilation process is going to cause a compiler error.

The third potential problem with header files is that two header files may define a macro differently. Macro substitutions are simply text substitutions that are performed at compile-time. The compiler will silently perform a macro substitution whenever it encounters a macro. The problem here is that if we have different definitions of a macro in two different translation units then this error will be masked by the macro substitutions that are performed. This is a difficult to trace error.

Modules are designed to solve these problems. In particular, modules enable:

- faster compilation times by preventing repeated inclusion of include files
- isolation from preprocessor macros
- prevention of violations of the one definition rule.

Modules are imported only once and the order in which they are imported does not matter.

The Structure Of A Module File

A module file has the following canonical structure:

```
module;                          // declare a global module fragment

// header files required by the module
#include <numeric>
#include <vector>

export module math;              // module declaration

// import <import other modules>

// non-exported declarations>
// names that are only visible inside the module


// exported functions

export int add(int first, int second) {
    return first + second;
}


export int multiply(int first, int second) {
    return first*second;
}
```

The global module fragment starting with the keyword *module* is optional. Header files should be included between it and the module declaration.

The module declaration *export module math* names the module. This declaration section is sometimes called the *module purview*. Imported modules as well as exported and non-exported names are specified in this section of the module file.

Ways To Export Module Symbols

We have already seen one way to export a top-level name (symbol) using an export specifier:

```
export module math;

export int add(int hour, int second);
```

A second way to export module symbols is by using an export group:

```
export module math;

export {
    int add(int first, int second);
    int multiply(int first, int second);
}
```

Thirdly, we can export a namespace:

```
export module math;

export namespace math {

    int multiply(int hour, int second);
    void makePolar();
}
```

In the last case, we use the exported names by prefixing them by their namespace (*math::multiply(5, 7);*).

Using Exported Module Symbols

A client program uses the exported functionality of a module by importing the module:

```
#include <iostream>
import math;

int main() {
  std::cout << '\n';
  std::cout << "add(3, 4): " << add(3, 4) << '\n';
  std::cout << "mul(3, 4): " << multiply(3, 4) << '\n';

}
```

Module Interface Units And Module Implementation Units

For a large module file, it may be beneficial to divide the file into a module interface unit and a module implementation unit. This mimics the division between a header file and an implementation file. This following example shows how this is done:

```
// mathInterfaceUnit.ixx

module;

#include <vector>

export module math;

// import <import other modules>

// non-exported declarations>
// names only visible inside the module


// exported names
export namespace math {

     // only the function declarations are specified
     int add(int, int);
     int getProduct(int x, int y);

}
```

The corresponding implementation unit is:

```
// mathImplementationUnit.cpp

module math;

#include <numeric>

namespace math {

   int add(int hour, int seconds) {
       return hour*3600 + seconds;
   }

   int multiply(int hour, int seconds) {
       return hour * 3600 * seconds;
   }

}
```

A module interface unit can have multiple implementation units.

Sub-modules

A module that imports another module can re-export this module as follows:

```
module;                          // global module fragment

// header files required by the module
#include <numeric>
#include <vector>

export module math;              // module declaration

// import a module and then export it
export import module trig_module

...
```

In this example, the *math module* imports the *trig_module* and re-exports it's functionality as part of the *math module*. The trig*_module* is said to be a sub-module of the *math module*.

Module Partitions

A large module can be partitioned. Each partition consists of a module interface unit (the partition interface file) and zero or more module implementation units. The following example shows how module partitioning is implemented:

```
// mathPartition.ixx – primary interface module

  export module math;       // module declaration

  export import :math1;
  export import :math2;
```

The *math* module imports symbols from *math1* and *math2* and re-exports these symbols as it’s own symbols (notice the colons).

The implementation files are:

```
// mathPartition1.ixx

export module math:math1;

export int add(int first, int seconds) {
    return first + seconds;
}
```

The statement *export module math:math1* exports the symbols of the *math1* partition to the *math* module. For the second partition, we have:

```
// mathPartition2.ixx

export module math:math2;

export {
    int multiply(int hour, int seconds) {
          return hour * 3600 + seconds;
    }
}
```

We use the partition as follows:

```
import math;

int main() {

  std::cout << '\n';
  std::cout << "add(3, 4): " << add(3, 4) << '\n';
  std::cout << "multiply(3, 4): " << multiply(3, 4) << '\n';

}
```

A Note On Templates

Templates should be declared in the module interface files:

```
// mathModuleTemplate.ixx

export module math;

export namespace math {

    template <typename T, typename T2>
    auto sum(T hour, T2 seconds) {  return hour*3600 + seconds;  }

}


// program file
#include <iostream>

import math;

using namespace std;

int main() {

  cout << "math::sum(2000, 11): " << math::sum(2000, 11) << endl;
  cout << "math::sum(2013.5, 0.5): " << math::sum(2013.5, 0.5) << endl;
  cout << "math::sum(2017, false): " << math::sum(2017, false) << endl;

}
```

Linkage

Prior to C++20, C++ supported two types of linkage:

- **Internal Linkage**: names with internal linkage are not accessible outside the translation unit.
- **External Linkage:** Names with external linkage are accessible outside the translation unit.

C++20 also implements **module linkage**. Names with module linkage are only accessible inside a module.

Header Units

Header units are a technique to smoothly transition from headers to modules. We just have to replace the *#include* directive with the new *import* directive:

```
#include <vector>      replace by    import <vector>;
#include "myHeader.h"  replace by    import "myHeader.h";
```

The compiler generates an import module from the header file.

Modern C++: Lambda Improvements (C++20)

Template Parameters In Lambdas

C++20 permits lambdas to have template parameters:

```
auto sumTem = []<typename T>(T x, T y) { return x + y; };
```

All of the template parameters must have the same type.

Allow lambda-capture `[=, this]`

Until C++20 *this* was always captured by reference in lambdas, even with the introducer [=]. To remove this confusion, C++20 deprecates such behaviour and allows the more explicit *[=, this]*:

```
[=]{};          // captures this by reference, deprecated since C++20
[*this]{};      // OK since C++17, captures this by value
[this]{};       // OK since C++20, captures this by reference
```

Modern C++: Designated Initialization (C++20)

Aggregate objects do not have constructors. This means that the initialization of an aggregate object must be effected by other means.

Designated initialization is a special case of aggregate initialization. Designated initialization implements the initialization of the data members of an aggregate using their names. Here is an example:

```
#include <iostream>

struct Point2D
{
  int x;
  int y;
};

int main
{
  Point2D point2D {.x = 1, .y = 2};

  std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
}
```

The initializers *.x* and *.y* are called designators.

For the initialization of a union, only one initializer can be provided.

If a designated initializer is missing for a data member of an aggregate, the member will be given it’s default value. The order of the designated initializers must match the order in the aggregate object. Here is an example:

```
#include <iostream>

class Point
{
  public:
  int x;
  int y;
  int z;
};

int main()
{
   Point point {.x = 5, .z = 20};

  std::cout << "point: " << point.x << " " << point.y << " "
      << point.z << '\n';


  // error because the designated initializer order is wrong.
  // Point point2 {.z = 20, .y = 1};

  //  std::cout << "point2: " << point2.x << " " << point2.y << " "
  //    << point2.z << '\n';
}


// Result
point: 5 0 20
```

Designated initializers prohibit narrowing conversions.

Modern C++: Using enum (C++20)

When the type alias *using enum enum_name* is specified in a local scope, we can use an enumerator value without it's scope qualifier.

Here is an example:

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

enum class Color {
  red,
  green,
  blue
};

string_view toString(Color col) {

  switch (col) {
     using enum Color;
     case red: return "red";
     case green: return "green";
     case blue: return "blue";
  }

  return "unknown";
}


int main() {

std::cout << "toString(Color::red): " << toString(Color::red) << '\n';

using enum Color;
std::cout << "toString(green): " << toString(green) << '\n';

}

// output
toString(Color::red): red
toString(green): green
```

Modern C++: Conversion From ptr To bool (C++20)

A type conversion from a pointer or a pointer-to-member to `bool` is narrowing and cannot be used unless an explicit cast is done. The exception is that a `nullptr` can be cast to *bool* when used in a direct initialization.

Modern C++: boolalpha (C++20)

The *boolalpha* keyword can be inserted into streams. bool values in the stream will be converted to their textual representations: *true* or *false,* instead of integral values.

```
#include <iostream>

int main() {
   using namespace std;

   cout << boolalpha << "hello " << true << endl;
   cout << boolalpha << (10 < 1) << endl;

}

// output
hello true
false
```

Modern C++: [[no_unique_address]] (C++20)

The attribute *[[no_unique_address]]* states that a data member of a class need not have a unique address. Note that in unions, the data members of a union ordinarily share a memory address.

This attribute is applied to a potentially empty data member so that the compiler can optimize it to occupy no memory. A data member with the *[[no_unique_address]]* attribute may share the address of another member. The syntax is as follows:

```
[[no_unique_address]] int x {};
```

Here is a complete example:

```
struct Empty {}; // empty strut

struct X {
    int i;
    Empty e;
};

struct Y {
    int i;
    [[no_unique_address]] Empty e;
};

int main()
{
    // the size of any object of empty class type is at least 1
    static_assert(sizeof(Empty) >= 1);

    // at least one more byte is needed to give e a unique address
    static_assert(sizeof(X) >= sizeof(int) + 1);

    // empty member optimized out
    static_assert(sizeof(Y) == sizeof(int));
    return 0;
}

Output: 0
```

Modern C++: [[likely]] and [[unlikely]] (C++20)

These attributes give a hint to the compiler regarding the path of execution that is expected to be more or less likely. This allows the compiler to optimize the code.

Modern C++: Coroutines (C++20)

Introduction

Coroutines are a major language feature in C++20. They are very useful in the implementation of applications that process asynchronous events. A coroutine is a function that can be suspended and optionally resumed at some later point in time. This can be done a multiple number of times.

An ordinary function cannot suspend it’s operation and be resumed at some later point in time. Once an ordinary function starts execution, it will continue execution until a return statement or the closing brace of the function is encountered.

The C++ standard requires that a coroutine function must use at least one of the following:

- co_await operator,
- co_yield keyword,
- co_return keyword

The main additional restriction is that a coroutine function cannot use the *return* statement but must instead use a *co_return* statement to return from the function. A function that abides by these restrictions will be automatically compiled as a *coroutine* by the C++ compiler. Note that a *coroutine* is not required have a *co_return* statement. There are some additional restrictions which we will discuss later.

For more information see: https://en.cppreference.com/w/cpp/language/coroutines

Why Is co_return Used Instead Of return?

*T*he stack is not used in a call or return from a coroutine function. Instead the state of the function is stored on the heap.

Considerable simplification is obtained when coroutines do not maintain their state on the stack. Consider the scenario where a number of coroutines suspend themselves in sequence. These suspended coroutines can resume in any order and not necessarily in the order of their suspension. Since coroutines do not pass parameters on the stack, the complexities of maintaining the stack when these coroutines resume in any order are entirely avoided. Thus coroutines can be described as stackless functions.

Coroutine Return Objects

A *coroutine* must return an object that has a specific well-defined structure and semantics. Consider the following declaration of the simplest object that a coroutine can return:

```
struct ReturnObject {

    struct promise_type {

      promise_type() = default;
      ReturnObject get_return_object() { return {}; };
      std::suspend_never initial_suspend() { return {}; };
      std::suspend_always final_suspend() noexcept { return {}; };
      void unhandled_exception() { };
      void return_void(void) {};
    };

};
```

ReturnObject can be a class or a struct. Similarly *promise_type* can be a class or a struct. The name *promise_type* must be spelled exactly as indicated. *promise_type* is not related to *std::async, std::future* or *std::promise*. A *promise_type* object is simply an object that controls the behavior of the coroutine.

The name *ReturnObject* is not special, it is simply the name of the class or struct that has the special syntax and semantics described above. It could be named *Task* or *Job*, for example.

The *promise_type* struct (or class) must have a constructor. In the declaration above, we are using the default constructor.

The *promise_type* object declares and defines the *get_return_object* method. The compiler automatically calls this method after the *promise_type* struct has been instantiated. This method instantiates and returns an object of type *ReturnObject* that contains the *promise_type* object. In our example, this function instantiates a *ReturnObject* using the default constructor (because of return {};).

Then there is a method called *initial_suspend* that returns an *awaiter* object and a method called *final_suspend* that also returns an *awaiter* object. Both of these objects are instantiated using their default constructors. *final_suspend* cannot throw an exception and thus must be declared with the *noexcept* attribute.

The C++ Standard Library provides us with two ready to use *awaiter* objects:

- `std::suspend_always: instructs the C++ runtime to suspend the coroutine`
- `std::suspend_never: instructs the C++ runtime to not suspend the coroutine`

We will examine the method *return_void(void)* shortly.

The final method in *promise_type* is the function *unhandled_exception.* It specifies the code that is to be executed if the coroutine throws an exception. The body of this function will be empty if we do not want to handle exceptions.

Instantiating A Coroutine

When a coroutine is called the compiler executes the following sequence of events:

1. Allocates memory for the coroutine using the *new* operator (this memory is called the activation frame of the coroutine).

2. Copies any function arguments passed by the calling function into the activation frame.

3. The C++ run-time then calls the constructor for the *promise_type* struct.

4. After the *promise_type* object has been constructed, the run-time calls the *get_return_object* method. This method constructs an object of type *ReturnObject*.

5. The method *initial_suspend* is then called before the first line of code in the body of the coroutine is executed. If the return value of *initial_suspend* is *std::suspend_never*, the coroutine starts execution of the code in the body of the function. If the return value is *std::suspend_always*, the coroutine goes into a suspended state without executing any code in the body of the coroutine.

6. The method *final_suspend* is executed when the coroutine is about to exit. This function also returns an *awaiter* object.

Note that an object of type *ReturnObject* will be returned to the caller when the coroutine suspends itself or the coroutine exits through a *co_return* or the closing brace of the function. The calling function uses *ReturnObject* to resume a suspended coroutine, destroy a coroutine or obtain information about it’s state.

Returning From A Coroutine

co_return statements have two forms. A *co_return* statement can return no value or it can return a value of some arbitrary type. In both cases the *co_return* statement is specified as:

```
co_return;
```

However a corresponding special function has to be defined to handle each *co_return* case. For the case where the *co_return* statement does not return a value, this special function is declared in the *promise_type* struct as:

```
void return_void();
```

The body of this function can be empty or implement some clean-up code.

In the example below, we implement a *co_routine* that simply prints *hello world* and then exits:

```
#include <coroutine>
#include <iostream>

struct ReturnObject {

    struct promise_type {

      promise_type() = default;
      ReturnObject get_return_object() { return {}; };
      std::suspend_never initial_suspend() { return {}; };
      std::suspend_never final_suspend() noexcept { return {}; };
      void unhandled_exception() { };

      void return_void() {}

    };

};

// declaration of coroutine foo
struct ReturnObject foo() {
  std::cout << "hello world" << std::endl;
  co_return;
```

```
      }

      int main(void) {
        foo();
      }

      // output
      hello world
```

Returning A Value From A Coroutine

If *co_return* returns a value then the coroutine must in addition to the declaration of *return_void()*, declare a function called *return_value* in *ReturnObject* in order to return a value.

In the following example, the coroutine returns the string value "*hello, I am returning*" when it returns:

```
#include <coroutine>
#include <iostream>

using namespace std;

struct ReturnObject {

 struct promise_type {

   promise_type() = default;

    ReturnObject get_return_object() {
      return ReturnObject
      {};
   }

   std::suspend_never initial_suspend() {
      return {};
   };

   std::suspend_never final_suspend() noexcept {
      return {};
   };

   void return_void() {}
   void unhandled_exception() { };

    };

    string return_value() const { return "hello, I am returning\n"; };

};


// declaration of the coroutine foo

struct ReturnObject foo() {
  co_return;
}


int main(void) {

  ReturnObject ret = foo();
  cout << ret.return_value();

  cout << "exiting main";
}


// output
hello, I am returning
exiting main
```

Suspending A Coroutine

A coroutine can suspend it's execution. It cannot be suspended by the caller. When a coroutine suspends itself, it saves it's state on the heap and returns a handle to the *ReturnObject* (to the function that called the coroutine). This handle can then be used by the calling function to resume or destroy the coroutine.

The points of the coroutine at which the coroutine is suspended are identified by the *co_await* and *co_yield* keywords. A coroutine can have multiple suspend points.

The *co_await* operator takes an *awaiter* object as it's sole parameter. For example:

```
co_await std::suspend_always {};
or
co_await std::suspend_never {};
```

In the example below, our *foo* coroutine suspends itself:

```
#include <coroutine>
#include <iostream>

struct ReturnObject {

    struct promise_type {

      promise_type() = default;
      ReturnObject get_return_object() { return {}; };
      std::suspend_never initial_suspend() { return {}; };
      std::suspend_never final_suspend() noexcept { return {}; };
      void unhandled_exception() { };
      void return_void() {}

    };

};

// declaration of coroutine foo
struct ReturnObject foo() {
  std::cout << "hello world" << std::endl;

  co_await std::suspend_always {};

  std::cout << "good-bye" << std::endl;
  co_return;
}


int main(void) {
  foo();
}


// output
hello world
```

The text *good-bye* is not output because we have suspended the coroutine.

Note that *co-await* is just a unary operator function, so we can write it as:

```
co_await(std::suspend_always) {
};
```

The body of the operator function can provide code that is to be executed prior to suspension.

Resuming A Suspended Coroutine

A calling function controls a coroutine through a *coroutine_handle* object.

See: https://en.cppreference.com/w/cpp/coroutine/coroutine_handle

std::coroutine_handle is a template object with the following declaration:

```
template< struct promise_type = void > struct coroutine_handle;
```

It has two specializations of interest. A primary specialization based on *promise_type* and the specialization *template<void> struct coroutine_handle*. The latter specialization is the same as *template<> struct coroutine_handle*, a template with no template parameters.

coroutine_handle holds a pointer to the coroutine data on the heap and implements a number of functions to control the coroutine. The compiler can implicitly cast a coroutine handle based on the primary specialization to the void specialization. This performs type erasure (removes the *promise_type* from the coroutine handle.);

We will now show the specification of the coroutine handle in two steps.

Firstly, the *coroutine_handle object* must be a data member of *ReturnObject*:

```
struct ReturnObject {
    struct promise_type {
      promise_type() = default;
      ReturnObject get_return_object() { return {}; };
      std::suspend_never initial_suspend() { return {}; };
      std::suspend_never final_suspend() noexcept { return {}; };
      void unhandled_exception() { };
      void return_void() {std::cout << "good-bye" << "\n"; }

    };

    std::coroutine_handle<promise_type> handle;

    ReturnObject(std::coroutine_handle<promise_type>
        _handle) : handle(_handle)
   {

   }
};
```

Note that *ReturnObject* has a constructor that receives a coroutine handle.

The second step uses the static member function *coroutine_handle::from_promise*. This function constructs a coroutine handle from a *promise_type* object. The *from_promise* method takes a *promise_type* object as it's only parameter and returns a coroutine handle.

Thus the declaration of *returnObject* is:

```
struct ReturnObject {

   struct promise_type {

   promise_type() = default;

   ReturnObject get_return_object() {
      return ReturnObject
      { std::coroutine_handle<promise_type>::from_promise(*this) };
   }

   std::suspend_never initial_suspend() {
      cout << "ready to execute coroutine" << endl;
      return {};
   };

   std::suspend_never final_suspend() noexcept {
      cout << "ready to exit coroutine" << endl;
      return {};
   };

   void return_void() {std::cout << "good-bye, coroutine has exited\n";}

   void unhandled_exception() { };

    };
```

std::coroutine_handle<promise_type> handle;

```
   ReturnObject(std::coroutine_handle<promise_type> _handle) :
          handle {_handle} {}
};
```

Here *get_return_object* creates a coroutine handle from the *promise_type* object and the constructor for *ReturnObject* uses this handle to instantiate a *ReturnObject.*

The *coroutine_handle* object provides the ***resume()*** method to resume a coroutine. The ***done()*** method is a predicate function that checks whether a coroutine is finished. *done()* returns true if the coroutine is suspended at it's final suspend point and false otherwise.

In the following example, the coroutine suspends itself twice and the calling function resumes the coroutine twice:

```
#include <coroutine>
#include <iostream>

using namespace std;

struct ReturnObject {

   struct promise_type {

   promise_type() = default;

   ReturnObject get_return_object() {
      return ReturnObject
      {std::coroutine_handle<promise_type>::from_promise(*this)};
   }

   std::suspend_never initial_suspend() {
      cout << "ready to execute coroutine" << endl;
      return {};
   };
   std::suspend_never final_suspend() noexcept {
      cout << "exiting coroutine" << endl;
      return {};
   };

   void return_void() {std::cout << "returning no value to caller\n";}

   void unhandled_exception() { };

    };

    std::coroutine_handle<promise_type> handle;
    ReturnObject(std::coroutine_handle<promise_type> _handle) :
           handle {_handle} {}

};


// declaration of the coroutine foo

struct ReturnObject foo() {
  cout << "coroutine is executing\n";
  co_await std::suspend_always {};
  cout << "coroutine has resumed\n";
  co_await std::suspend_always {};
  cout << "coroutine has resumed\n";
  co_return;
}


int main(void) {

  ReturnObject ret = foo();

  if (ret.handle.done() == false) {
    cout << "resuming coroutine\n";
    ret.handle.resume();
  }

  if (ret.handle.done() == false) {
    cout << "resuming coroutine again\n";
    ret.handle.resume();
  }

  if (ret.handle.done()) {
      cout << "coroutine has finished";
  }

}


// output
ready to execute coroutine
```

```
coroutine is executing
resuming coroutine
coroutine has resumed
resuming coroutine again
coroutine has resumed
returning no value to caller
exiting coroutine
coroutine has finished
```

Note: Instead of resuming with *ret.handle.resume()* we can use *ret.handle()*.

Passing Intermediate Values From A Suspended Coroutine

A coroutine can pass an intermediate value to the calling function when it suspends itself. This can be done any number of times. The *co_yield* keyword is used to pass an intermediate value. The syntax for *co_yield* is:

```
co_yield val;
```

where val is some object or scalar value. In order for *co_yield* to be used, the *promise_object* as well as *ReturnObject* have to be modified as follows:

```
struct promise_type {

   promise_type() = default;

   T val_;

   std::suspend_always yield_value(T val) {
      this->val_ = val;
      return std::suspend_always {};
   }

   ReturnObject get_return_object() {
      return ReturnObject
      {std::coroutine_handle<promise_type>::from_promise(*this)};
   }

   std::suspend_never initial_suspend() {
      return {};
   };

   std::suspend_never final_suspend() noexcept {
      return {};
   };

   void return_void() {}

   void unhandled_exception() { };

    };


    std::coroutine_handle<promise_type> handle;

    ReturnObject(std::coroutine_handle<promise_type> _handle) : handle
      {_handle}
   {
   }

    T get_value() const { return handle.promise().val_; };

};
```

The *promise_type* struct must have a method called *yield_value* those parameter *T* has the type of the value that will be returned to the calling function. When the coroutine executes the *co_yield* statement, the *yield_value* function is called. *yield_value* returns an awaiter *std::suspend_always* object which will suspend the coroutine and then enables the calling function to fetch the intermediate value.

To enable the calling function to fetch the intermediate value, *ReturnObject* defines a utility method *get_value* which returns the *promise_type* data member *val_*. *handle.promise()* gets the promise type object.

In the example below we fetch the squares of the first 12 natural numbers:

```
#include <coroutine>
#include <iostream>

using namespace std;

struct ReturnObject {

 struct promise_type {

   promise_type() = default;
   int val_;

   std::suspend_always yield_value(int val) {
      this->val_ = val;
      return std::suspend_always {};
   }

   ReturnObject get_return_object() {
      return ReturnObject
      {std::coroutine_handle<promise_type>::from_promise(*this)};
   }

   std::suspend_never initial_suspend() {
      return {};
   };

   std::suspend_never final_suspend() noexcept {
      return {};
   };

   void return_void() {}

   void unhandled_exception() { };

    };

    std::coroutine_handle<promise_type> handle;

    ReturnObject(std::coroutine_handle<promise_type> _handle) : handle
      {_handle}
   {
   }

    int get_value() const { return handle.promise().val_; };

};


// declaration of the coroutine square

struct ReturnObject square() {

  int square;
  for (int i = 0; i < 12; ++i) {
     square = i*i;
     co_yield square;
  }

  co_return;
}


int main(void) {
  ReturnObject ret = square();

  while (!ret.handle.done()) {
    cout << ret.get_value() << " ";
    ret.handle.resume();
  }

}


// output
0 1 4 9 16 25 36 49 64 81 100 121
```

Destroying A Coroutine

A coroutine can be destroyed with the *destroy* method. In the context of the previous square coroutine example, the syntax to destroy the coroutine is:

```
if (!ret.handle.done()) {
   ret.handle.destroy();
}
```

Coroutine Restrictions

The following types of functions cannot be coroutines:

- Functions with the auto return type
- Functions with variadic parameters,
- Functions which are constexpr or consteval
- Constructors and destructors

Modern C++: #pragma once

#pragma once is not part of the ISO standard, but it is implemented by all modern C++ compilers.

The use of *#pragma* once in a header file can reduce build times, as the compiler won't open and read an include file again after the first *#include* of the file in a translation unit. It has an effect similar to the **include guard idiom**, which uses a preprocessor macro definition to prevent multiple inclusions of the contents of a file. *#pragma once* also helps to prevent violations of the **one definition rule**, the requirement that all templates, global functions and objects have no more than one definition in our code.

There is no advantage in using both the *#pragma once* and the include guard idiom in a header file.

#pragma once is supported in GCC since version 5.4.

Header Management And Include Guards

Why Does C And C++ Require Header Files

In C and C++, identifiers in a program such as variables, function names, class names, and so on must be declared before they can be used.

For example, we can't just write *foo = 42* without first declaring *foo*:

```
    int foo;
```

This declaration tells the compiler whether the element is an *int*, a *double*, a *function*, a *class* or some other thing. Besides providing a name for the identifier, the declaration informs the compiler about the amount of memory that the identifier requires.

Every identifier used in an implementation file must be declared (directly or indirectly) in a header file. When we compile a program, every implementation file is compiled independently after a translation unit is generated from the file. The compiler has no knowledge about what identifiers have been declared in other translation units. This means that if we use a class or function or global variable, we must provide a declaration of that thing in each *.cpp* file that uses it. Each such declaration of an identifier must be exactly the same in all of these files. A slight inconsistency in these declarations of an identifier will cause errors, or unintended behaviour, when the linker attempts to merge all of the object files into a single executable program.

To minimize the potential for such declaration errors, C and C++ have adopted the convention of using header files to contain declarations. We make declarations in a header file, and then use the *#include* directive in every implementation file (*.cpp file*) or other header file that requires the declaration. The *#include* directive inserts a copy of the header file directly into the *.cpp* file (or other header file) prior to compilation.

C and C++ also use forward declarations. A forward declaration tells the compiler that a variable, function, struct or class is in some other header file.

Effective Header Management

The problem with the C++ compilation model is as follows. We make one tiny, innocuous change in a header file and then 200 implementation files that include this header file have to to be recompiled. This consumes a lot of time. In addition, we can have circular dependencies in header files. Header A includes header B, which includes header C. Header C includes header A.

How can we effectively manage header files and header dependencies in order to make our build times quicker?

The first rule is that an implementation file should only include the header files that it actually needs.

Forward Declarations

The second rule is to use forward declarations in preference to including header files whenever possible.

Forward declarations introduce an identifier to the compiler. In certain cases, an identifier is all that is required, we do not need to include the header file containing the declaration of the identifier. The compiler only needs to know that a specific identifier is declared in some other header file. Details about the identifier, such as how much memory it requires or what methods it has, are generally not relevant until the identifier is actually used in an implementation.

A forward declaration to an identifier will be sufficient when the include or implementation file only uses a pointer to the identifier and this file does not need to know the size of the identifier and does not need to call a member of a class or struct.

The following example shows how forward declarations work:

```
// elephant.h

// trunk.h is required because an Elephant object has a Trunk object.
#include "trunk.h"

class Elephant
{
public:
    // constructor
    Elephant(const Trunk& trunk);

private:
    // Trunk object
    Trunk m_trunk;

}
```

The problem with this implementation is that every time *trunk.h* changes, every source file that includes *elephant.h* will have to be recompiled. We can remove this dependency by including a forward reference to the *trunk* class in *elephant.h*.

```
// elephant.h
// trunk.h is not required because the compiler
// already knows everything about a pointer.

// However, the compiler needs to know that something
// called a `Trunk` does indeed exist.

class Trunk;  // Forward Declaration to the Trunk class

class Elephant
{
public:
    // We are using a reference (implicitly a pointer), so the
    // compiler does not need to know the details of the Trunk class
     Elephant(const Trunk& trunk);


private:
    // Pointer to a Trunk object, so no specifics about
    // the trunk class are required.
    Trunk* ptr_trunk;


}
```

This works because to the compiler, a pointer or reference is always the same size and can always have the same operations performed on it, regardless of the identifier that it is pointing or referring to.

In our example, the elephant class contains a member *ptr_trunk* that points to a *Trunk* class. All that the compiler needs to know at this time is the fact that *ptr_trunk* is a pointer and that the identifier name *Trunk* exists somewhere else. Similarly all that the compiler needs to know about the reference *&trunk* is that it is a reference (implicit pointer) to a *Trunk* object.

Now changes made to the *trunk.h* file do not effect *elephant.h*.

Therefore to prevent triggering lengthy recompilation of implementation files, we should prefer forward declarations.

Include Guards

A global variable can only be defined once. Because of this, including the same header file in a translation unit more than once will generate a compiler error when a global variable is defined more than once.

A common way to prevent multiple include errors is to use an include guard. An include guard is created by enclosing the whole header file in an *#ifndef* section that checks for the existence a macro that is specific to the header file only. If the macro is not defined in the translation unit, the header file is included in the translation unit. If the macro has already been defined in the translation unit, it is not included. This ensures that a header file is included only once in a translation unit.

Here is an example of the use of an include guard in a header file:

```
// store.h
#ifndef __STORAGE_HEADER__
```

```
#define __STORAGE_HEADER__

   declarations etc.

#endif // __STORAGE_HEADER__

// store.cpp
#include “store.h”
#include “store.h”      // OK, store.h is included only once
...
```

pragma once directive

The *pragma once* directive was discussed previously. Many compilers support the *pragma once* directive which can be placed at the top of an include file instead of an include guard:

#pragma once

__*has_include* Preprocessor Directive

C++17 has added the __*has_include* preprocessor directive. This lets us test whether an include file exists. The preprocessor evaluates to true if the header file is found:

```
#if __has_include("store.h")
#include "store.h"
#endif
```

C++ Standard Library: std::any

std::any implements a type safe variable that can dynamically hold values of any type. In other words, a *std::any* variable is dynamically typed. The template declaration is:

```
template<class T> class any<T>;
```

Here is an example of defining a *std::any* variable:

```
#include <any>

int main()
{
  std::any<int> tmp {5}
  // or
  // std::any  tmp {5};          // type int is inferred
}
```

This will create a *std::any* variable integer type variable *tmp* with value 5.

We can also initialize an *any* variable by assignment:

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

int main()
{
  any  var = 2.174;       // type double is inferred
}
```

We can dynamically assign values with different types to an *any* variable:

```
std::any<float>  tmp = 3.14
tmp = “hello world”
```

Useful *std::any* member functions include:

reset	Destroys the value contained in an *any* variable
has_value	Tests whether a *std::any* variable contains a value
type	Returns a typeid object for an *any* variable

The function ***std::any_cast*** takes the address of an *any* variable and returns a pointer to the contained value. For example:

```
int* ptr = std::any_cast<int>(&a);     // a is an any type variable
```

The following *any_cast* will throw an exception because we are trying to cast an any variable of type *int* to a double type:

```
double* ptr = std::any_cast<double>(&a)         // a is an int type
```

Here is an example of usage:

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

int main()
{
  cout << "placing 3.12 in an any container" << endl;
  any  icontainer {3.12};

  if (icontainer.has_value())
    cout << "container has value" << endl;
  else {
    cout << "icontainer does not have a value" << endl;
    return -1;
    }

  cout << "contained type = " << icontainer.type().name() << endl;

  double* p = any_cast<double>(&icontainer);
  cout << "contained value = " << *p << endl;
}

// output
placing 3.12 in an any container
container has value
contained type = d
contained value = 3.12
```

C++ Standard Library: std::async, std::future

Note: *std::async* requires knowledge of *std::thread*. Review of *std::ref* is recommended.

Consider the case where we want to pass a value from a child thread to the parent thread. This can be done by using a shared variable that is protected by a mutex and a condition variable that sends a notification when the variable is changed by the child thread. A much simpler implementation is available with the *std::async* function.

The following program shows canonical usage of the *std::async* function:

```
#include <iostream>
#include <future>


// compute the factorial of a number
size_t factorial(size_t num)
{
   size_t result = 1;
   for (size_t ctr = num; ctr > 0; ctr--)
     result *= ctr;
   return result;
}

int main()
{
  std::future<size_t> f = std::async(std::launch::async, factorial, 20);
  size_t fac = f.get();
  std::cout << fac << std::endl;
}

      # output
      2432902008176640000
```

In this program, *std::async* is a function template that returns an *std::future object* ***f*** of type *size_t* sometime in the future. The template parameter specifies the type of future object that will be returned. The *std::launch::async* parameter creates a thread and executes the *factorial* function in this thread. The remaining parameters, if any, of *std::async* are the arguments of the factorial function.

The factorial function will be launched in a new thread and the main thread will continue processing.

When the main thread encounters the statement ***f.get()***, it will wait for the factorial function to return.

The *f.get()* function will return the value computed by the factorial function. This value will be wrapped in the object of type *std::future<size_t>*.

Note: The parent thread can call *f.get* only once (since it gets it's value only when the factorial function returns). Calling it more than once will result in an exception.

std::async Launch Types

std::async supports different launch types:

std::launch::async	Starts the target function in a new child thread.
std::launch::deferred	Defers the launching of the target function until *f.get()* is encountered by the parent thread. Then executes the target function in the parent thread.
std::launch::async | std::launch::deferred	Defers the launching of the target function until *f.get()* is encountered by the parent thread. The target function may be executed in the main thread or a new thread depending upon the compiler's implementation.

The default value of the first parameter is *std::launch::deferred* | *std::launch::async*. The first parameter can be omitted if we are using the default value.

Using std::promise

std::promise lets a parent thread send a value to a child thread sometime in the future. The child thread can pause during it's processing, waiting for the parent thread to send a value to it.

The following canonical example shows how promises are implemented. A parent thread creates a child thread and promises to send the factorial function a value in the future. The child thread begins it's processing and waits for the parent to send a value. When the child gets a value from the parent, it resumes processing. Finally, the factorial function sends it's return value to the parent:

```
#include <iostream>
#include <future>

int factorial(std::future<int>& fu)
{
   int start = fu.get();
   int result = 1;

   for (int ctr = start; ctr > 0; ctr--)
     result *= ctr;

  return result;
}

int main()
{
  std::promise<int>  p;
  std::future<int> fu = p.get_future();

  // std::ref required since future objects cannot
  // be copied or assigned
  std::future<int> f = std::async(std::launch::async,
    factorial, std::ref(fu));

  std::this_thread::sleep_for(std::chrono::seconds(10));
  p.set_value(10);

  size_t fac = f.get();
  std::cout << fac << std::endl;

}
```

The statement ***std::promise<int>* p** creates a promise object of type *int*. This is a promise to send a value of type *int* in the future.

The expression **p.get_future()**, returns a future object of type *int* which will be used to send the promised value.

We pass **std::ref<fu>** as an argument to the child function since future objects cannot be passed by value (copied).

p.set_value() sends the promised value to the child function.

The child function has a reference to an *std::future<int>* as a parameter. In the body of the function, **fu.get()** causes the child function to wait until it receives the promised value.

Note again that *f.get()* can be called only once.

std::shared_future

Consider the situation where we want to execute our factorial function in ten threads and each thread is to receive the same value through a promise. Since a future object cannot be copied, we would have to create a distinct future object for each thread and then pass this future object to a factorial instance as a reference. However there is a simpler solution to this problem using *std::shared_future* objects.

The following program shows how to use shared futures:

```
#include <iostream>
#include <future>

int factorial(std::shared_future<int> fu)
{
  int start = fu.get();
  int res = 1;

  for (int ctr = start; ctr > 0; ctr--)
```

```
    res *= ctr;

  return res;
}

int main()
{
  std::promise<int> p;
  std::future<int> fu = p.get_future();
  std::shared_future<int> sfu = fu.share();

 std::future<int> f1 = std::async(std::launch::async, factorial, sfu);
 std::future<int> f2 = std::async(std::launch::async, factorial, sfu);

  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  p.set_value(10);

    size_t fac1 = f1.get();
    std::cout << fac1 << std::endl;

  size_t fac2 = f2.get();
  std::cout << fac2 << std::endl;

}
```

```
Output:
3628800
3628800
```

Once the promise has returned a future in which it will send a value to a child function, the function **fu.share()** converts this object into a *std::shared_future object*.

The shared future object is passed as an argument to the child function. Since a shared object can be copy constructed and copy assigned, we do not need to pass a reference to the child object.

The statement **p.set_value(10)** sends the same value 10 to all child functions through the shared future object.

C++ Standard Library: std::atomic

The declaration for the std::atomic class template is:

```
template <class T> struct atomic {  }
```

An object of type *std::atomic<class T>* changes the value of a scalar variable of type T atomically. That is, a variable of this type is guaranteed to provide read and write access without any data races. Atomic variables can be changed without requiring a mutex lock. An integer atomic type variable can be defined as follows:

```
atomic<int> x;
```

The atomic class template supports types such as: *bool*, *char*, *int*, *long*, *unsigned int*, *unsigned long*, *short*, *unsigned short*, *wchar_t*, *unsigned long long* and so forth.

Atomic variables cannot be copied or moved.

The statement *atomic<int> x*; will create an uninitialized atomic variable *x* of type int.

Some of the atomic template class methods are: *store*, *load*, *exchange*, *operator++ and operator--*.

Here is an example of usage:

```
#include <iostream>
#include <atomic>

int main()
{
  std::atomic<int>  val(5);
  std::cout << "value of atomic variable val with constructor 5: ";
  std::cout << val << std::endl;

  int x = 3400;
  std::cout << "value of variable x: " << x << std::endl;
```

```
    x = val.load();
    std::cout << "value of x after x = val.load(): ";
    std::cout << x << std::endl;

    val++;
    std::cout << "value of atomic variable after val++: ";
    std::cout << val << std::endl;

    val.store(99);
    std::cout << "value of atomic variable after val.store(99): ";
    std::cout << val << std::endl;

    --val;
    std::cout << "value of atomic variable after --val: ";
    std::cout << val << std::endl;

  }

  // output
  value of atomic variable val with constructor 5: 5
  value of variable x: 3400
  value of x after x = val.load(): 5
  value of atomic variable after val++: 6
  value of atomic variable after val.store(99): 99
  value of atomic variable after --val: 98
```

Atomic Type Aliases

std::atomic<bool>	std::atomic_bool
std::atomic<int>	std::atomic_int

Other type aliases are similar.

C++ Standard Library: std::bind

Note: also see *std::function*.

The *std::bind* template implements a function adaptor. It converts a function into an object. Some of the function parameters can be optionally bound to fixed values.

The declaration of the *std::bind* class template is:

```
#include <functional>
template<class T, class ... args>  bind;
```

std::bind takes a function pointer as it’s first parameter T and the function parameters as a parameter pack. For example:

```
auto Adder = std::bind(add, std::placeholders::_1, std::placeholders::_2);
```

Here the function pointer is *add*, it takes two parameters. ***std::placeholders::_1*** and ***std::placeholders::_2*** are two variables that symbolically represent the corresponding parameters of *add*.

If we want the second parameter of *add* to be fixed to 15, we declare our *Adder* object as:

```
auto Adder = std::bind(add, std::placeholders::_1, 15);
```

Now we can call the function *add* with it’s second parameter fixed to 15, as follows:

```
Adder(5);
```

The function call will return 20;

Here is a complete example:

```
#include <iostream>
#include <functional>

int adder(int x, int y)
{
  return x+y;
}

int main()
{
   auto Adder = std::bind(adder, std::placeholders::_1, std::placeholders::_2);
   std::cout << Adder(5, 15) << std::endl;

   auto Adder2 = std::bind(adder, std::placeholders::_1, -15);
   std::cout << Adder2(15) << std::endl;
}

// output
20
0
```

C++ Standard Library: std::chrono

The C++ standard library class template *std::chrono* provides implementations for time objects. *std::chrono* consists three components: clocks, durations and time points.

Clocks

A clock consists of a starting time called an **epoch** and a tick rate. The tick rate is the smallest discrete time unit of the clock.

There are three types of clocks:

- std::chrono::system_clock
- std::chrono::steady_clock
- std::chrono::high_resolution_clock

std::chrono::system_clock is a wrapper for the real-time clock provided by the operating system. *std::chrono::steady_clock* is a monotone increasing clock that cannot be adjusted once it has been started. A high_resolution clock is a user defined clock. It is frequently cast to a *system_clock* or a *steady_clock*.

Clocks have the following member functions:

now()	gets the current *time_point* since the start of the clock's epoch
to_time_t	converts a system clock time to a *std::time_t* structure
from_time_t	Converts a *std::time_t* to a *std::chrono::system_clock*

All of these member functions are static.

The std::ratio Class Template

In order to use *std::chrono* effectively, we must understand the class template *std::ratio*.

std::ratio is a class template for rational numbers. A rational number is the ratio of two integers where the number in the denominator is not zero. *std::ratio* is declared as:

```
#include <ratio>

template <intmax_t n, intmax_t m = 1>
class ratio;
```

intmax_t is a signed integer type with a maximum and minimum value. *n* represents the numerator of the rational number and *m* the denominator. *m* cannot be zero.

Here is an example of defining a ratio:

```
class ratio<5, 2>   rational_num;
```

Each instantiation of a *std::ratio* object represents a rational number.

Note: the data member *m* may not be zero and may not be equal to the most negative value of *std::intmax_t*.

Two *std::ratio<n, m>* objects with different *n* or *m* values are distinct types even if they represent the same rational number after reduction.

A *std::ratio<n, m>* object has two constant public static data members *num* and *den*. *num* is the numerator and *den* is the denominator.

Here is an example of usage:

```
#include <iostream>
#include <ratio>

int main()
{
  std::ratio<3,7> rl;
  std::cout << rl.num << " : " << rl.den << std::endl;

  typedef std::ratio <4, 8>  r2;
  std::cout << r2::num << " : " << r2::den << std::endl;

}

// Output
3 : 7
1 : 2
```

Notice that in the declaration: *ratio <4, 8>*, *r2::num* and *r2::den* are reduced to the largest common denominator.

Durations

std::chrono::duration is a class template that represents a time interval. It is declared as follows:

```
#include <chrono>

template <class Type, class Period = std::ratio<1>>
class duration;
```

A duration class template has two parameters. *Type* is an arithmetic type such as *int*, *long*, *double*, *float* and so forth. This represents the duration in ticks (the number of ticks since the clock's epoch). *Period* is a parameter of type *std::ratio*. It represents the number of ticks in a second or the number of seconds from one tick to the next if a tick is greater than a second. *Period* is a compile-time rational constant object.

Period<1,5000> means that there are five thousand ticks in a second. *Period<1000,1>* (or *Period<1000>*) means that there is a tick every 1000 seconds.

The default period is *Period<1>* (one tick every second).

Suppose that we have 5000 ticks in a second then a duration of twenty seconds can be defined as follows:

```
#include <iostream>
#include <chrono>
#include <ratio>

using namespace std;

int main() {
  // 100000 is the duration in ticks
  chrono::duration<int, ratio<1,5000>>  dur {100000};

  cout << (double) dur.count()/5000;
}
```

The constructor of a duration object takes a value in ticks.

Duration objects have a member function *count()*. *count* returns a constant number of type *Type*. This number is the duration of the object in ticks or the number of ticks since the clock's epoch. Suppose that our *count* function returns 1000. This is the duration as the number of ticks. This value is interpreted in terms of the period of the duration object. Suppose that the period is the ratio<1,1000>. This means that there are a thousand ticks in a second. So *count* is interpreted as returning a duration of one second. If the period is ratio<1, 10> then there are 10 ticks in a second. So *count* is interpreted as returning a duration of 100 seconds.

To summarize, a duration class contains the compile-time tick period, which is a *std::ratio* type. The tick period specifies the number of ticks in a second. The default period is *ratio<1,1>* one tick per second *(represented as ratio<1>)*. A period such as *ratio<1, 1000>* means that there are a thousand ticks in a second so one tick is a millisecond. A period of *ratio<1, 1000000>* means that there are a million ticks in a second, so one tick is a microsecond. A period of *ratio<3600,1> (or ratio<3600>)*

means that there is a tick every 3600 seconds.

The standard library defines some **common duration types**:

```
typedef std::chrono::duration<signed int, ratio<1,1000000000>>
std::chrono::nanoseconds;
typedef std::chrono::duration<signed int, ratio<1,1000000>>
std::chrono::microseconds;
typedef std::chrono::duration<signed int, ratio<1,10>>
std::chrono::milliseconds;
typedef std::chrono::duration<signed int, ratio<1,1>> std::chrono::seconds;
typedef std::chrono::duration<signed int, ratio<60,1>> std::chrono::minutes;
typedef std::chrono::duration<signed int, ratio<3600,1>> std::chrono::hours;
```

An example of the use of these predefined duration types is:

```
#include <iostream>
#include <chrono>

int main()
{
    // define a duration object with duration of 1 second
    std::chrono::seconds sec(1);

   // create a microseconds object that receives a one
   // second duration object
   std::chrono::microseconds  ms(sec);

   std::cout << ms.count() << std::endl;
}

// output
1’000’000
```

The duration class overloads the +, -, *, /, %, ==, <=, >, >=, !=, <=> operators. These operators permit arithmetic operations and comparisons on duration objects that have different periods. For the arithmetic operators, both duration operands are converted to their greatest common period.

Here is another duration example:

```
#include <iostream>
#include <chrono>
#include <ratio>

int main()
{
  std::chrono::duration<long, std::ratio<1>>  dr1(100);
  std::cout << "duration = " << dr1.count()  << std::endl;

  std::chrono::duration<long, std::ratio<60, 1>>  dr2(50);
  std::cout << "duration = " << dr2.count()  << std::endl;

  std::chrono::duration<double, std::ratio<1, 2>>  dr3(100);
  std::cout << "duration = " << dr3.count()  << std::endl;

  // add two duration objects
  auto dr4 = dr1 + dr2;
  std::cout << "duration = " << dr4.count() << std::endl;

  // subtract duration objects
  auto dr5 = dr2 - dr1;
  std::cout << "duration = " << dr5.count() << std::endl;

  return 0;
}

// output
duration = 100
duration = 50
duration = 100
duration = 3100
duration = 2900
```

The function template **std::duration_cast** converts a duration object to a duration object with a different period. Since the compiler will implicitly convert the periods of duration objects to a common period, this function is only needed if there is a loss of precision when converting between duration types. The *duration_cast* function template is declared as follows:

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
```

ToDuration is the duration type that we are casting to. The parameter inside duration cast is some duration object.

Here is an example of using a *duration_cast*:

```
#include <iostream>
#include <chrono>

int main()
{
  // declare a duration object with duration 144 minutes
  std::chrono::minutes mins(144);

 // create a hours object that receives the mins duration object
 std::chrono::hours hrs =
       std::chrono::duration_cast<std::chrono::hours>(mins);

   std::cout << hrs.count() << std::endl;

}

      # output
      2
```

The duration cast is needed because when the minutes duration object is converted into an hours duration object there will be a loss of precision. This means that the compiler will refuse to convert the minutes duration to an hour duration implicitly.

Time Points

A *time_point* represents a point in time since the start of a clock (the epoch of the clock). Thus a *time_point object* is a duration since the clock's epoch. *time_point<system_clock>* represents a point in time since the system clock started.

The template class *time_point* is declared as follows:

```
#include <chrono>

template <
           class Clock,
           class Duration
         >
class time_point;
```

Clock is a Clock class such as *system_clock*, *steady_clock* or some other clock type. The second template parameter is a duration type.

Some member functions of a *time_point* object are:

time_since_epoch	The duration since the clock's epoch (number of ticks since epoch)
operator++, operator--	Increment or decrement the clock's duration
operator+=, operator-=	Modify the time point by a given duration
operator+, operator-	Add or subtract durations

time_point objects can be compared with: *operator==*, *operator!=*, *operator>*, *operator <*, *operator>=* and the *operator <=*

time-point_cast converts a time point to another time point on the same clock, with a different duration.

System Time Computations

We can get a friendly representation of the current system time as follows:

```
#include <iostream>
#include <ctime>
#include <ratio>
#include <chrono>

int main ()
{
  using namespace std::chrono;

  duration<int, std::ratio<60*60*24> > one_day (1);

  auto today = system_clock::now();
  auto tomorrow = today + one_day;

  time_t time_struct;
```

```
    time_struct = system_clock::to_time_t (today);
    std::cout << "today is: " << ctime(&time_struct);

    time_struct = system_clock::to_time_t (tomorrow);
    std::cout << "tomorrow is: " << ctime(&time_struct);

  }

  Output:
  today is: Thu Aug 18 06:01:48 2022
  tomorrow will be: Fri Aug 19 06:01:48 2022
```

Elapsed Time Calculation

Here is an example of an elapsed time calculation:

```
#include <iostream>
#include <chrono>

int main()
{
  std::chrono::steady_clock  clock;
  std::chrono::time_point start  = clock.now();

  // do some work
  double sum= 0;
  for (int ctr = 0; ctr < 1'000'000'000; ctr++)
       sum += ctr;

  std::chrono::time_point end = clock.now();
  std::chrono::duration dur = end - start;

  std::chrono::milliseconds msecs =
          std::chrono::duration_cast<std::chrono::milliseconds>(dur);

   std::cout << msecs.count() << std::endl;

}

# output
3731
```

Put The Current Thread To Sleep

Note: review *std::thread*

```
#include <thread>
#include <chrono>

std::this_thread::sleep_for(std::chrono::seconds(15));
```

Time In Seconds Since Unix Epoch

```
#include <stdio.h>
#include <time.h>

long seconds = time(NULL);
printf("Seconds since January 1, 1970 = %ld\n", seconds);
```

We can alternatively use *std::chrono*:

```
#include <chrono>
#include <ratio>

size_t time_now =
   duration_cast<seconds>(system_clock::now().time_since_epoch()).count();
```

C++ Standard Library: std::copy

The function template *std::copy*, copies one container to a second container of the same type. *std::copy* is declared as:

```
template <class InputIterator, class InputIterator, class OutputIterator>
         OutputIterator copy(InputIterator first, InputIterator last,
             OutputIterator res);
```

The copy function receives an iterator to the start of the first container, an iterator to the end of the first container and an output iterator to a second container. The function copies the values of the first container from the *first* iterator value up to and including the *last* iterator value to the second container beginning at the *res* iterator. The function returns an iterator to the second container that is one item beyond the last item copied:

```
#include <iostream>
#include <vector>

int main()
{
  std::vector vec1  = {0,1,2,3,4,5,6,7,8,9};
  std::vector<int> vec2(10);
  std::cout << "vec1 size = " << vec1.size() << std::endl;

  std::copy(vec1.begin(), vec1.end(), vec2.begin());

  std::cout << "vec2 size = " << vec2.size() << std::endl;

  for (auto item : vec2)
    std::cout << item << " ";
}

// output
vec1 size: 10
vec2 size = 10
0 1 2 3 4 5 6 7 8 9
```

Important Note: *std::copy* does not automatically allocate memory for the destination vector, if needed.

C++ Standard Library: std::format

std::format is a variadic function template that implements a modern formatting facility based upon *printf*. *std::format* also implements easy string interpolation. It is declared as follows:

```
      template<class... Args>
      string format(“fmt”, Args&&... args);
```

The function argument *fmt* is a string consisting of text and placeholders. This is followed by a variable number of function arguments. The *format* function returns a string.

Aside from any placeholders, the text in the *fmt* argument is copied to the output string verbatim. The placeholders in *fmt* are designated by braces {}. These designated braces are substituted by the function arguments corresponding to the brace positions. For example:

```
#include <iostream>
#include <string>
#include <format>


int main() {

 string str = std::format("this is an example of {} and {}", "dogs", "cats");
 cout << str;
}

  #output
  “this is an example of dogs and cats”
```

A placeholder can contain an integer index which specifies that the argument at the index position is to be used in the substitution. For example:

```
       string str = std::format(" {2} and {1}", "dogs", "cats");

       produces the output:
```

```
“cats and dogs”
```

There can be more arguments than placeholders in a *format* function.

A format specifier can be used inside a placeholder. This will format the value that is substituted into the placeholder. For example:

```
std::format(“{1:.2f}”, 2.71423)

# output
2.71
```

Note that the format specifier is separated from the index by a colon. Format specifiers have the same syntax and semantics a *printf* format specifiers.

Note: *std::format* was introduced in C++20 and requires GCC 12.2 or later.

C++ Standard Library: std::function

The class template *std::function* is a polymorphic function wrapper. If we have a free function, lambda expression, function pointer, functor, member function or data member, then *std::function* can be used to wrap this target into a callable class (*functor*). This functor has a copy constructor and is copy assignable. Furthermore this functor can be stored and passed as an argument to some other function.

The declaration is:

```
#include <functional>

template <class R, class... Args>
class function<R(Args)>
```

R is the data type of the function return value. *Args* is a comma separated list of argument types of the function. *function* is the function wrapper class.

Note: This declaration creates a family of *std::function* types.

We construct a function object for a free function as follows:

```
#include <iostream>
#include <functional>

// we will create a function object for adder
double adder(int x, double y)
{
    double z = x + y;
    std::cout << "inside adder" << std::endl;
    return z;
}

int main()
{
   // declare the functor and then initialize it
   std::function<double(int, double)> fun = adder;

   // invoke the callable target in the fun object
   auto result = fun(3, 3.12);
   std::cout << result << std::endl;
}

// output
inside adder
6.12
```

In this example, *fun* is the function wrapper. The template declaration *std::function<double(int, double)>* starts with the return type of the function *adder;* the parameters of *adder* are in parentheses. The function *adder* is assigned to the functor *fun*.

Some useful member functions for this class template are:

operator bool	Tests whether the callable object is valid
operator=	Assigns a target to the functor
operator()	Invoke the target
target	Get a pointer to the target

In the following example, we wrap a function object around a lambda expression and then call the lambda through the functor:

```
#include <iostream>
#include <functional>

int main()
{
   // create a function object for a lambda
   std::string s = "hello world";

   auto lm = [](std::string q) {
     std::cout << q << std::endl;
     return "returning from lambda";
   };

   std::function<const char*(std::string)> fun = lm;
   // invoke the lambda through the function object
   std::string p = fun(s);
   std::cout << p << std::endl;
}
```

```
// output
hello world
returning from lambda
returning from lambda
```

In this example, *fun* is the function wrapper and the lambda *lm* is assigned to *fun*.

In the following example, a member function of a class object is invoked through a function object:

```
#include <iostream>
#include <functional>

class Foo
{
public:
  void print(const char* str) const
  {
     std::cout << str << std::endl;
  }
};

int main()
{
   Foo foo;

   std::function<void(Foo&, const char*)> fun = &Foo::print;

   // invoke the print member function
   const char* p = "hello world";
   fun(foo, p);
}

// output
hello world
```

Notice the definition of the class template *fun*:

```
std::function<void(Foo&, const char*)> fun = &Foo::print;
```

The return type is void and the first parameter in the parentheses is a reference to the class (not the class instance), the member function parameter types then follow (const char * in our case). The right hand side is the offset of the member function *print* in Foo: *&Foo::print*.

The callable object is invoked as *fun(foo, "bye bye")*. The first argument is the instantiated *foo* object, then the member function's parameters follow.

We can get the value of a data member through a functor as follows:

```
#include <iostream>
#include <functional>

class Foo
{
public:
 Foo() { greeting = "hello world"; };
 std::string greeting;
};

int main()
{
    Foo foo;
    std::function<std::string(Foo&)> fun = &Foo::greeting;

   // get the data member
   std::string ret = fun(foo);

   std::cout << ret << std::endl;
}

// output
hello world
```

Type Erasure

Type erasure is a set of techniques for creating a type that provides a uniform interface to various underlying types, while hiding the underlying type information from the client.

std::function<R(A...)>, which has the ability to hold callable objects of various types, is the best known example of type erasure in C++. Another example are *std::any* variables.

C++ Standard Library: std::initializer_list

Initializer lists permit us to initialize a collection of objects using list syntax, for example: *int array[2] {1, 2};*.

An initializer list is a class template that is declared as follows:

```
#include <initializer_list>

template <class T>
class initializer_list;
```

The following is an example of the declaration for an *initializer_list* variable:

```
#include <initializer_list>

std::initializer_list<double> bar;
```

Internally, an *initializer_list* is an array of objects of type T.

An *initializer_list* object can be created from a **braced init list**:

```
#include <iostream>
#include <initializer_list>

int main()
{
  // declare an initializer list and then assign values to it
  std::initializer_list<int> ilist = {4, 1, 11};

  for (auto value : ilist)
    std::cout << value << " ";

  std::cout << std::endl;

  return 0;
}

// output
4 1 11
```

The item {4, 1, 11} is called a braced init list.

An initializer list has the following public members:

begin()	Pointer to the first initializer list item
end()	Pointer to the last initializer list item
size()	Size of the initializer list

A class permits objects to be instantiated with a braced init list, providing it has a constructor that takes an initializer list as a parameter. Consider this snippet:

```
Foo foo = {1, 2, 6, 11};
```

When the compiler encounters this statement, it creates an initializer list (right value) from the braced init list on the right side. Now *foo* can be initialized with this initializer list providing that it has a constructor that takes an initializer list as a parameter.

C++ Standard Library: std::map

Insert A Key-Value

```
iterator map_name.insert( { key, element } )
```

The parentheses is required since a pair object is being inserted.

Remove A Key-Value

```
void map_name.erase(iterator i);
```

This will delete the key-value pair pointed to by the iterator.

Test That A Map Has A Key

```
if (myMap.find("foo") == myMap.end()) {
  // not found
} else {
  // found
}
```

Test That Map Has A Value

Beginning with C++20, associative containers have the *contains* member function to check if a map container has a key or a set has an element :

```
std::map<int, char> map {{1, 'a'}, {2, 'b'}};
map.contains(2);                                        // true
map.contains(123);                                      // false

std::set<int> set {1, 2, 3};
set.contains(2);                                       // true
```

C++ Standard Library: std::move

std::move() takes a *lvalue* (left value) object as a parameter and returns a *rvalue* reference to the object. *std::move* does this by using *static_cast* on it's argument. *std::move* lets us use the more efficient move constructor and move assignment operator in object construction.

std::move() requires the *utility* header.

```
#include <iostream>
#include <utility>
#include <string>

using namespace std;

void print(string&& value) {
   cout << value << endl;
}

int main()
{
   string value("print the string");

   // print(value) fails because value is not a rvalue

   // value is converted to a rvalue
   print(std::move(value));

}

// output
print the string
```

C++ Standard Library: std::optional

Consider a function that returns the index at which the number zero is found in some text. It is perfectly acceptable and not an error if zero does not exist in the text. *std::optional* is designed to handle these types of situations. The template type *std::optional* is declared as follows:

```
#include <optional>

std::optional<class T> var;
```

std::optional<class T> consists of all possible values of type T and the value *std*::*nullopt*. *std::nullopt* represents *no value* or *nothing*. *std::nullopt* semantically represents the absence of a value.

The following example shows a typical use of *std::optional*:

```
#include <iostream>
#include <optional>

using namespace std;

optional<unsigned int> evenNumber(unsigned int num) {
       if (num%2 == 0)
          return num;

       return nullopt;
}


int main()
{
  std::optional<unsigned> opt = evenNumber(5);

if (opt.has_value())
  {
    std::cout << "The number is "
              << opt.value()
              << "\n";
  }
else
  cout << "no value returned";
}

# return value
no value returned
```

If the *evenNumber* function does not receive an even number, *evenNumber* returns *nullopt*. Otherwise the function returns an *unsigned int*.

opt.has_value() is a predicate that checks whether the *std::optional* variable has a value or is *nullopt*. ***opt.value()*** returns the value.

If *opt* is *nullopt*, *opt.value()* will throw an exception of type *std::bad_optional_access*.

The pointer dereferencing operators * and -> are implemented, but accessing an empty *std::optional* object has undefined behavior.

reset() destroys the *std::optional* object and makes it empty (*nullopt*).

C++ Standard Library: std::reference_wrapper

std::reference_wrapper

A reference wrapper is a class template that creates a reference to an object or a variable. It’s declaration is:

```
template< class T >
class reference_wrapper;
```

Note: *reference_wrapper* declares a family of reference wrapper types parameterized on T.

This class will wrap an object or variable of type T and create a reference to this object or variable. The wrapper object can be treated as if it is the object itself. A *std::reference_wrapper* object can be copied and assigned.

Here is an example:

```
#include <iostream>
#include <functional>

int main() {
   int x = 5;

   // create a reference wrapper for x
   std::reference_wrapper<int>  int_wrapper {x};

   std::cout << int_wrapper << std::endl;
   int_wrapper++;
   std::cout << int_wrapper << std::endl;
   x++;
   std::cout << int_wrapper << std::endl;
   return 0;
}

# output
5
6
7
```

std::reference_wrapper can be used to pass references to functions.

std::reference_wrapper automatically deduces the type and *const* properties of the object being wrapped.

std::ref

std::ref is a convenience template for creating a reference wrapper for a variable.

```
#include <iostream>
#include <functional>

int main() {
   int x = 5;

   // this creates a reference wrapper for variable x and
   // assigns it to wrapper

   std::reference_wrapper wrapper = std::ref(x);

   std::cout << wrapper << std::endl;
   wrapper++;
   std::cout << wrapper << std::endl;

   return 0;
}

# output
5
6
```

If *x* is a reference wrapper then *std::ref* simply makes a copy of the variable.

const Reference Wrappers

std::cref creates a *const* reference wrapper. That is, the value of the reference variable cannot be changed.

C++ Standard Library: std::sample

std::sample selects n random elements from a sequence container:

```
#include <iostream>
#include <random>
#include <iterator>
#include <algorithm>

int main() {
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    std::vector<int> out;

    std::sample(
         v.begin(),               // range start
         v.end(),                 // range end
         std::back_inserter(out), // where to put the random elements
         3,                       // no. of elements to sample
         std::mt19937{std::random_device{}()}
     );

    std::cout << "Sampled values: ";
    for (auto i : out)
        std::cout << i << ", ";
}


// output
Sampled values: 1, 4, 9,
```

The following example creates a random string:

```
#include <iostream>
#include <random>

using namespace std;

int main() {

    std::string str {"0123456789abcdefghjklmnpqrstuvwxyz"};
    std::random_device rd;
    std::mt19937 generator(rd());
    std::shuffle(str.begin(), str.end(), generator);
    string password = str.substr(0, 6);

    cout << password << endl;
}
```

C++20 String Interpolation

In C++20 we can use *std::format*:

```
#include <format>

string s = std::format(" From {1} to {0}", "a", "b");

# From b to a
```

C++17 String Interpolation

In C++17, we can use stringstreams:

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main()
{
  stringstream ss;
  string a = "hello";
  string b = "world";

  ss << "value is " << a << " And " <<  b;

  // convert the stringstream to a string
  cout << ss.str() << endl;
}

// output
value is hello And world
```

Clear the stringstream with:

ss.str("");

or

ss.str(std::string());

or:

ss.clear()

erase

Erases the portion of a string value that begins at the character position *pos* and spans *len* characters (or until the end of the string, if either the content is too short or if len is *string::npos*):

```
string& erase (size_t pos = 0, size_t len = npos);
```

returns a pointerto the erased string.

find

The *string::find(string)* function returns the index of first occurrence of a substring *substr* in a given string, if there is an occurrence of the substring in the string. If the given substring is not present in this string, find() returns -1. The syntax is:

```
str.find(substr)
```

We can also specify the search starting position in this string and the search interval for the substring.

```
str.find(substr, start, n)
```

start is optional. And if start is provided, *n* is optional.

front

```
char& string::front();
const char& string::front() const;
```

returns a reference to the first element of a string. If the string object is const-qualified, the function returns a const char&. Otherwise, it returns a char&.

Numeric To String

This string method converts a numeric argument to a *std::string*:

```
std::to_string(1.2);            // == "1.2"
std::to_string(123);            // == "123"
```

replace

Replaces the portion of a string that begins at character position *pos* and spans *len* characters by new contents:

```
string& replace (size_t pos,  size_t len,  const string& str);

string& replace (iterator i1, iterator i2, const string& str);
```

returns a pointer to the string after replacement.

Replace all occurrences of a char with another char

```
#include <algorithm>
#include <string>

void foo() {
  std::string s = "example string";
  // replace each 'x' with 'y'
  std::replace( s.begin(), s.end(), 'x', 'y');
}
```

starts_with and ends_with

Beginning with C++20, strings (and string views) have the *starts_with* and *ends_with* member functions to check if a string starts or ends with the given substring:

```
std::string str = "foobar";
str.starts_with("foo");                      // true
str.ends_with("baz");                        // false
```

String To Integer

Convert a string to an integer:

```
int num = stoi(str);
```

Integer To String

```
int my_int 5;
string str = to_string(my_int);
```

Substrings

The following string function returns a substring from a string:

```
string substr (size_t pos = 0, size_t len = npos) const;
```

pos is the position of the first character to be copied.

len is the number of characters to copy. If *len* is not specified, all characters from the starting position to the end of the string are copied. If the string has length less than *len*, only the characters to the end of the string are copied.

C++ Standard Library: std::swap

The function template *std::swap*, swaps the contents of two objects of the same type. *std::swap* is defined as:

```
template <class T>
void swap (T& a, T& b);
```

Example:

```
#include <iostream>
#include <vector>

int main()
{
  std::vector vec1  = {0,1,2,3,4,5,6,7,8,9};
  std::vector<int> vec2 {-9, -8, -7, -6, -5, -4, -3, -2, -1, 0};

  std::swap(vec1, vec2);

  for (auto& item : vec1)
    std::cout << item << " ";
  std::cout << std::endl;

  for (auto& item : vec2)
    std::cout << item << " ";
  std::cout << std::endl;

  return 0;

}
```

C++ Standard Library: std::thread

C++11 implements a threading library through *#include <thread>*. When a program begins execution, the operating system executes the program in the program's own process space. The basic idea behind threads is to sub-divide a process into a number of independently executing threads. The purpose is to improve the performance of the program by executing multiple threads concurrently.

By default when a C++ program begins execution, it starts in a default thread called *main*. If the program does not create any threads then it runs entirely in the *main* thread. The program is said to be a single threaded application.

Basic Thread Creation

The following program uses two threads to compute the number of odd and even numbers in a range:

```
#include <iostream>
#include <thread>

size_t oddsum = 0;
size_t evensum = 0;

void even_sum(int start, int stop) {
    for (int i = start; i < stop; i++)
        if (i%2 == 0 )  evensum += i;
}

void odd_sum(int start, int stop) {
     for (int i = start; i < stop; i++)
        if (i%2 != 0 )  oddsum += i;
}

int main()
{
  size_t start = 1000;
  size_t stop  =  90678124;

  // create thread objects t1 and t2
   std::thread t1(even_sum, start, stop);
   std::thread t2(odd_sum, start, stop);
   t1.join();
   t2.join();

   std::cout << "even sum = " << evensum <<  std::endl;
   std::cout << "odd sum  = " << oddsum  <<  std::endl;
   return 0;
}

// output
even sum = 2055630497451282
odd sum  = 2055630542789844
```

In this code, *std::thread t1(even_sum, start, stop)* creates a thread called *t1* and runs the function *even_sum* in this thread. The arguments *start* and *stop* are passed to the function. Note that *t1* is receiving a function pointer as it's first argument.

The statement *t1.join()* joins the *t1* child thread to the *main* thread (the parent thread). This halts the *main* thread until the *t1* thread has finished executing the function *even_sum*. In general, any thread can be joined to it's parent thread. This join halts the parent thread until the child thread has completed.

Note: when we create multiple threads there is no assurance as to which thread will start first.

When a thread object is created we should call *join()* on it in order to prevent the parent thread from exiting while the child thread is running. Joining a thread causes the parent thread to suspend execution until the child thread returns.

Can A Reference Be Passed To A Thread

The short answer is no. Even if an attempt is made to pass a reference variable to a thread, the thread will receive a copy of the variable. If we want to pass a reference variable to a thread we must use the function template *std::ref*. *std::ref* wraps a reference object around a variable. This wrapper object can be copied by value and assigned. The wrapper object can be treated as if it is the variable that it wraps.

Executing A Lamda Function In A Thread

In the previous example, we used a function pointer to create a thread. The function that is pointed to executes in the newly created thread. We can also execute a lambda in a thread:

```
#include <iostream>
#include <thread>

int main()
{
  // lambda
  auto  fn = [](int i) {
                for (int ctr = i; ctr >= 0; ctr--)
                   std::cout << ctr << " ";
  };

  std::thread t1(fn, 10);
  t1.join();
  return 0;
}

// output
10 9 8 7 6 5 4 3 2 1 0
```

Executing A Non-Static Public Member Function In A Thread

The following example shows how to execute a non-static public member function of a class in a thread:

```
#include <iostream>
#include <thread>

class Base {
public:
   void fn(int i)
   {
     for (int ctr = i; ctr >= 0; ctr--)
       std::cout << ctr << " ";
   }
};

int main()
{
  Base b;
  std::thread  t1(&Base::fn, &b, 10);
  t1.join();

  return 0;
}

// output
10 9 8 7 6 5 4 3 2 1 0
```

The first argument of the thread function is the offset of the member function *fn* in the class *Base*, the second argument is the address of the class instance *b*. The third argument is the value passed to *fn*.

Executing A Static Member Function In A Thread

A static member function of a class can also be executed in a thread:

```
#include <iostream>
#include <thread>

class Base
{
public:
  static void fn(int i) {
    for (int ctr = i; ctr >= 0; ctr--)
        std::cout << ctr << " ";
  }
};

 int main()
{
  std::thread t1(&Base::fn, 10);
  t1.join();

  return 0;
}

// output
10 9 8 7 6 5 4 3 2 1 0
```

The first argument of the thread function is the offset of the static member function in the class. The second argument is the value passed to the static member function. Note that we do not need to create an instance of a class in order to invoke a static member function.

Lastly, we can execute a functor in a thread. A functor is an object that behaves like a function. This behaviour is induced by overloading the *()* operator.

Thread Joins

Trying to join a thread that does not exist or joining the same thread more than once will throw an exception:

```
t1.join();
t1.join();        // exception thrown
```

To prevent these types of errors we should test that a thread is joinable before joining the thread:

```
if (t1.joinable())  t1.join();
```

Detaching A Thread From It's Parent Thread

A thread t1 can be detached with:

```
t1.detach();
```

This statement will cause the parent and child thread to execute independantly from one another. A *join* stops the execution of the parent thread until the child thread has finished execution. The child thread does not have to be joined to the parent thread before detaching it.

After a thread is created we must *join* or *detach* it.

Calling *t1.detach()* more than once (with no intervening *t1.join()*) will throw an exception.

Mutexes

A race condition occurs when two or more threads try to access a shared resource simultaneously. The purpose of a mutex is to prevent race conditions from occurring by locking shared resources so that only one thread can access the resource at any time. A mutex accomplishes this by locking the portion of the code that accesses a shared resource.

A **critical section** is a sequence of program statements that will be executed in their entirety by one thread at a time. Typically a critical section consists of program statements that access variables (resources) that are shared by multiple threads. Critical sections have the following type of symbolic syntax:

```
lock the mutex
   execute critical section
unlock the mutex
```

Notice that the mutex object has to be shared by all of the threads that want to execute the statements in the critical section. Therefore the mutex is defined as a global variable.

Locking Multiple Mutexes With std::lock()

std::lock() attempts to lock multiple mutexes in a deadlock-free manner. It’s syntax is:

```
std::mutex m1;

std::lock(m1);
```

std::lock() will attempt to obtain a lock on all of the mutexes that are it's arguments. If it fails to lock all of the mutexes, it will unlock the mutexes that it was able to lock and wait. Thus it is a blocking call. In the next try, it will try to lock the mutexes in a different order.

If the *std::lock* function succeeds in locking all of the mutexes, it processes the critical section. The *std::lock* function must unlock all of the mutexes that it locked after it has finished processing the critical section.

Here is an example:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex    mu;          // our mutex object

int balance = 0;           // this is our shared resource

void deposit(int amount)
{
     mu.lock();                // obtain a lock on the shared resource
     balance += amount;        // this is the critical section
     mu.unlock();              // unlock the mutex
}

int main()
 {
   std::thread t1(deposit, 100);
   std::thread t2(deposit, 200);

   t1.join();
   t2.join();

   std::cout << balance << std::endl;

  }

// output
300
```

When *t1* locks *balance* and *t2* tries to access it, the t2 thread will discover that the resource is locked. *t2* will then wait for *balance* to be unlocked. Once the resource is unlocked, *t2* will acquire a lock and begin updating *balance*. We have a deadlock if *t1* never releases the mutex.

Note: If an exception is thrown while executing a critical section, the mutex lock will not be released.

try_lock()

try_lock attempts to lock a mutex. It returns *false* if it cannot lock the mutex and *true* if a lock is acquired. The syntax is:

```
using namespace std;

mutex m1;

bool m1.try_lock();
```

try_lock has undefined behavior if it is called by a thread that already has a lock.

Look at the following thread code:

```
#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>

using namespace std;
mutex mu;

void print(void) {

   if (mu.try_lock() == false) {
      cout << "mutex is locked\n";
   }
   // critical code section
   else {
      cout << "mutex is unlocked\n";
      cout << "hello world\n";
      mu.unlock();
   }
}


int main() {
    mu.lock();

    thread t1(print);
    t1.detach();
    this_thread::sleep_for(std::chrono::seconds(3));
    mu.unlock();

    thread t2(print);
    t2.detach();
    this_thread::sleep_for(std::chrono::seconds(2));

}

output:
mutex is locked
mutex is unlocked
hello world
```

The thread tries to acquire a lock with *mu.try_lock().* If it fails, the thread returns immediately. If it succeeds, the thread acquires a lock and executes the critical section code. After it has executed the critical section, it releases the lock.

Locking Multiple Mutexes With std::try_lock(m1, m2, ... mn)

std::try_lock(m1, m2, ... mn) tries to lock all of the mutexes that are it's arguments. Two or more arguments must be supplied to this function.

The declaration syntax is:

int try_lock(m1, m2, ... mn)

If *try_lock* succeeds in locking all of the mutexes, it returns -1. If it fails to lock all of the mutexes, it unlocks all of the mutexes that it was able to lock and returns the zero-based index of the first mutex that it failed to lock.

The following code sample shows canonical usage:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex    m1, m2;          // our mutex objects

int balance = 500;             // this is our shared resource

void deposit(int amount)
{
  int ret = std::try_lock(m1, m2);
  if (ret == 0) {
     std::cout << "resource m1 is locked\n";
  }
  else if (ret == 1) {
     std::cout << "resource m2 is locked\n";
}
else {
     std::cout << "resource m1 and m2 not locked\n";
     // this is the critical code section
     balance += amount;
     std::cout << "critical section processed" << std::endl;
     m1.unlock();
     m2.unlock();
    }
}

int main()
 {

   std::cout << "balance: " << balance << std::endl;

   m2.lock();
   m1.lock();
   std::thread t1(deposit, 100);
   std::thread t2(deposit, 200);
   t1.join();
   t2.join();
   m1.unlock();
   m2.unlock();

   std::thread t3(deposit, 100);
   t3.join();
   std::thread t4(deposit, 200);
   t4.join();

   std::cout << "balance: " << balance << std::endl;
  }
```

```
output:

balance: 500
resource m1 is locked
resource m1 is locked
resource m1 and m2 not locked
critical section processed
resource m1 and m2 not locked
critical section processed
balance: 800
```

Note that after the critical section is processed, all of the mutexes must be unlocked.

If an exception is thrown, *try_lock(...)* will unlock all of the mutexes before rethrowing the exception.

Timed Mutex

The *timed_mutex::try_mutex_for(some_duration)* function tries to acquire a lock on a mutex until a specified period of time elapses. The function returns true if it acquires a lock. It returns false if the lock is not acquired in the specified time duration. It's declaration is:

```
#include <mutex>

bool std::timed_mutex::try_lock_for(const std::chrono::duration);
```

Here is a reference program:

```
#include <iostream>
#include <thread>
#include <mutex>

std::timed_mutex m1;           // our mutex object

int balance = 500;             // this is our shared resource

void deposit(int amount)
{
  using ms = std::chrono::milliseconds;

  int ret = m1.try_lock_for(ms(200));
  if (ret == false) {
     std::cout << "resource is locked\n";
     return;
  }

 std::cout << "resource is not locked\n";
 // this is the critical code section
 balance += amount;
 std::cout << "critical section processed" << std::endl;
 m1.unlock();

}

int main()
 {

   std::cout << "balance is: " << balance << std::endl;

   m1.lock();
   std::thread t1(deposit, 100);
   t1.join();

   m1.unlock();

   std::thread t2(deposit, 100);
   t2.join();

   std::cout << "balance is: " << balance << std::endl;

  }

output:
balance is: 500
resource is locked
resource is not locked
critical section processed
balance is: 600
```

In this example, we wait for up to 200 milleseconds to acquire a lock on a shared resource.

Note that the shared resource has to be unlocked after the critical section has finished.

There is another variant of a timed mutex where we try to acquire a lock until a specified period of time has expired. It's declared as:

```
bool try_lock_until(std::chrono::time_point)
```

The following is a usage snippet.

```
std::timed_mutex mu;
auto now = chrono::steady_clock::now()
...
mu.try_lock_until(now + chrono::seconds(1))
...

mu.unlock();
```

Recursive Mutex

A recursive mutex keeps a count of the number of times that it has been locked. In order for the shared resource to be unlocked, the mutex must be unlocked the exact number of times that it has been locked.

The semantics are as follows. A thread acquires a lock with *lock()* or *try_lock()*. After acquiring a lock, the thread can then lock the resource a multiple number of times by calls to *lock()* or *try_lock()*. The lock prevents other threads from accessing the shared resource. The shared resource becomes accessible by another thread after it is unlocked the exact number of times that it has been locked.

A std::recursive_mutex object is typically used in recursive functions and loops where a particular resource is locked a multiple number of times.

The declaration syntax for a recursive mutex is:

```
std::recursive_mutex mu;
```

The following example shows the use of a *recursive_mutex* in a recursive function:

```
#include <iostream>
#include <thread>
#include <mutex>

int dec = 10;
std::recursive_mutex  m1;

void rfunc(std::string thread_id)
{
  if (dec < 1) return;
    m1.lock();
    // critical section
    dec--;
    std::cout << thread_id << ":" << dec << " ";
    // recursion occurs here
    rfunc(thread_id);
    // end of critical section
    m1.unlock();
}

int main()
{
  std::thread t1(rfunc, "a");
  std::thread t2(rfunc, "b");
  t1.join();
  t2.join();

  return 0;
}

// output
a:9 a:8 a:7 a:6 a:5 a:4 a:3 a:2 a:1 a:0 b:-1
```

Using std::lock_guard<mutex>

A *lock_guard* is a class template with template declaration:

```
std:lock_guard<T> m;
```

The template type parameter must be a mutex type which implements the *lock()* and *unlock()* and functions.

A *lock_guard* automatically releases a mutex when it goes out of scope even if an exception is thrown. The following example shows how to use a *std::lock_guard<mutex>* object:

```
#include <iostream>
#include <thread>
#include <mutex>

size_t  sum = 0;
std::mutex m1;

void adder(std::string id, size_t loop_limit)
{
  // create a lock_guard object with parameter m1
  std::lock_guard<std::mutex> lock(m1);
  // critical section
  for (size_t  i = 0; i < loop_limit; i++)
       sum += i;
  std::cout << id << ":" << sum << std::endl;
}


int main()
{
  std::thread  t1(adder, "t1", 20);
  std::thread  t2(adder, "t2", 30);
  t1.join();
  t2.join();
  return 0;
}

// output
t1:190
t2:625
```

We create a *lock_guard* object *lock* for mutex *m1*. A thread entering the *adder* function will wait until a lock is acquired. The lock will be automatically released once the *lock_guard* object goes out of scope or if an exception is thrown.

Ownership of a *lock_guard* object cannot be given by a copy constructor or by assignment.

std::unique_lock<std::mutex>

The class template *std::unique_lock<std::mutex>* is a mutex wrapper that allows locking, time-constrained locking and recursive locking. The mutex template type must support one of the following functions: *lock(), try_lock_for or try_lock_until.* The template type must also support the *unlock()* function.

The following mutex types are supported:

1. lock(*some_mutex*)
2. try_lock(*some_mutex*)
3. try_lock_for(*some_mutex*)
4. try_lock_until(*some_mutex*)
6. owns_lock(*some_mutex*)   - does the object own the mutex

The function owns_lock() determines whether a std::unique object has acquired a lock.

A lock acquired by an *unique_lock* object will be automatically released if an exception is thrown or if the lock object goes out of scope.

A *std::unique_lock* object can be used with condition variables. Furthermore these objects are movable but not copyable.

Here is an example of usage:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m1;              // mutex for critical section

void print_block (int n, char c) {

     // we can use any of the following mutexes:
      // std::unique_lock<std::mutex>  lock (m1);
      // std::unique_lock<std::mutex>  try_lock (m1);
      // std::unique_lock<std::mutex>  try_lock_for (m1);
      // std::unique_lock<std::mutex>  try_lock_until (m1);

       std::unique_lock<std::mutex>  lock (m1);
      // critical section
      for (int i = 0;  i < n;  ++i)  std::cout << c;
       std::cout << '\n';
}

int main ()
{
  std::thread t1 (print_block, 20, '*');
  std::thread t2 (print_block, 20, '$');

  t1.join();
  t2.join();

  return 0;
}


// output
********************
$$$$$$$$$$$$$$$$$$$$
```

If we use type inference, we can specify the *std::unique_lock<std::mutex>* object as:

```
std::unique_lock lock(m1);
```

Condition Variables

Condition variables allow us to synchronize threads; they can only be used with *std::unique_lock* objects.

Threads can wait upon a condition variable. One or all of these waiting threads can be notified when a condition variable changes.

The following example shows the canonical use of a condition variable. In this example, we want to deposit money into a bank account and then withdraw money. Obviously, the deposit must be made before the withdrawal.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mu;
std::condition_variable cv;
size_t  balance = 0;                   // shared resource

void deposit(size_t amt)
{
     mu.lock();
     balance += amt;
     mu.unlock();
     cv.notify_one();
  }

void withdraw(size_t amt)
{
     std::unique_lock<std::mutex>  ul(mu);
     cv.wait(ul, [] { return(balance > 0 ? true : false); });
     balance -= amt;
     ul.unlock();
}

int main()
{
     std::thread t1(deposit, 500);
     std::thread t2(withdraw, 400);
     t1.join();
     t2.join();
     std::cout << "final balance: " << balance << std::endl;
     return 0;
}

// output
final balance: 100
```

Notice that use of the condition variable cv requires an unique_lock. In the *withdraw* function, we first try to lock the mutex *mu*. If we cannot lock the mutex, we keep on waiting. If the lock succeeds, we test the value of *cv.wait()*.

The first parameter of *cv.wait* is the *unique_lock* mutex. The second parameter is a lambda that implements a predicate condition. If the condition evaluates to true, processing proceeds. If the condition is not satisfied, the lock on the mutex is released and we keep on waiting.

If the condition variable object sends a notification (***cv.notify_one()***), we try to acquire the lock again. If the lock cannot be acquired we keep on waiting. If a lock is acquired, we test the condition variable wait predicate again.

In the deposit function, the thread waits until a lock is acquired. If the lock is acquired, the deposit is made. Next, the lock is released. Then one waiting thread is notified (**notify_one**) or all waiting threads are notified (**notify_all**).

Notice that in the deposit function, the mutex is unlocked and then the notification is sent.

Notice that the *ul.unlock* function call in the *withdraw* function is optional since the mutex will be automatically released once it falls out of scope.

Thread Local Storage

A thread can maintain data that is private to itself. This data is known as thread local data or thread local storage. Each thread local variable has a scope confined to the thread that created the variable. Thread local variables behave like static variables and last as long as the thread continues in existence. Thread local storage is created when a thread needs this storage, that is, the storage is created lazily.

The storage specifier ***thread_local*** is used to create a thread local variable.

Here is an example of thread local storage:

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>

std::mutex  mu;
```

```
thread_local std::string s  {"hello "};

void greet(std::string id)
{
  std::lock_guard<std::mutex> lock(mu);
  s = s + id;
  std::cout << s <<std::endl;

}

int main()
{
  std::thread t1(greet, "t1");
  std::thread t2(greet, "t2");
  std::thread t3(greet, "t3");

  t1.join();
  t2.join();
  t3.join();
}

// output
hello t1
hello t2
hello t3
```

C++ Standard Library: std::tuple

A tuple is a data structure that can store an arbitrary number of items with heterogeneous data types:

```
#include <string>
#include <tuple>

std::tuple<int, std::string, float>  t1(5, “hello world”, 5.3);
```

This creates a tuple with the given types and values. A tuple can also be constructed without any values:

```
std::tuple<int, string, float>  t2;
```

Note: Typically STL containers can only store values that are not references. Tuples have an unusual feature, unlike other STL containers, they can store references.

Tuple Assignment

We can do a tuple copy assignment as follows:

```
std::tuple<int, string, float> t1<5, “hello”, 5.5>;
std::tuple<int, string, float> t2;

t2 = t1;
```

The assignment operator performs a member by member copy.

Get Tuple Elements

We can extract the members of a tuple as follows:

```
#include <iostream>
#include <string>
#include <tuple>
```

```
int main() {
    std::tuple<int, std::string, double>  t1(5, “hello world”, 5.3);

     int x0 = std::get<0>(t1);
     std::string x1 = std::get<1>(t1);
     float x2 = std::get<2>(t1);
     std::cout << x0 << “ ” << x1 << “ “ << x2 << std::endl;
     return 0;
 }
```

The *std::get* function returns a reference to a tuple value. The index *i* in the *std::get<i>* function must be a compile-time constant.

Tuples that have the same types can be lexicographically compared:

```
if (t2 < t1) {
   ...
}
```

Tuple Concatenation

Tuples can be concatenated:

```
std::tuple<int, string, float> t1<5, “hello”, 5.5>;
std::tuple<string, char> t2;

t3 = std::tuple_cat(t1, t2);
```

t3 is a tuple of type *<int, string, float, string, char>*

Setting The Value Of A Tuple Element

std.get<i>(foo) returns a reference to the ith element of the tuple. We can set the ith element as follows:

```
std::get<i>(my_tuple) = “hello world”;
```

Tuple Type Traits

We can get the size of a tuple and the type of a tuple member as follows:

```
// get the size of the tuple t3
auto tsize = std::tuple_size<decltype(t3)>::value;

// get the type of a tuple member
auto tuple_t = std::tuple_element<1, decltype(t3)>::type d
```

std::pair

A pair is a special case of *std::tuple* which has two elements. *std::pair* is defined as:

```
template <class T1, class T2>
struct pair;
```

C++ Standard Library: std::tie

Note: see *std::tuple* first

The template ***std::tie*** creates a tuple those elements are references. *std::tie* has syntax:

```
template< class T1, class T2, class T3 >
tuple(T1&, T2&, T3&) tie( T1, T2, T3 ) noexcept;
```

The following example uses *std::tie* to create a tuple:

```
#include <iostream>
#include <string>
#include <tuple>

int main()
{
  int x = 5;
  std::string y = "hello world";
  double z = 6.8;

  std::tuple<int, std::string, double>  mytuple;

  // assign references to my_tuple
  mytuple = std::tie(x, y, z);

  std::cout << "tuple (0): " << std::get<0>(mytuple) << std::endl;
  std::cout << "tuple (1): " << std::get<1>(mytuple) << std::endl;
  std::cout << "tuple (3): " << std::get<2>(mytuple) << std::endl;

  return 0;
}

Output:
tuple (0): 5
tuple (1): hello world
tuple (3): 6.8
```

The tuple *mytuple* contains references to *x*, *y* and *z*. Changing *x*, *y* or *z* will change *mytuple* and vice versa.

We can ignore elements in a *std::tie* function by using **std::ignore** as the element value.

Unpacking A Tuple

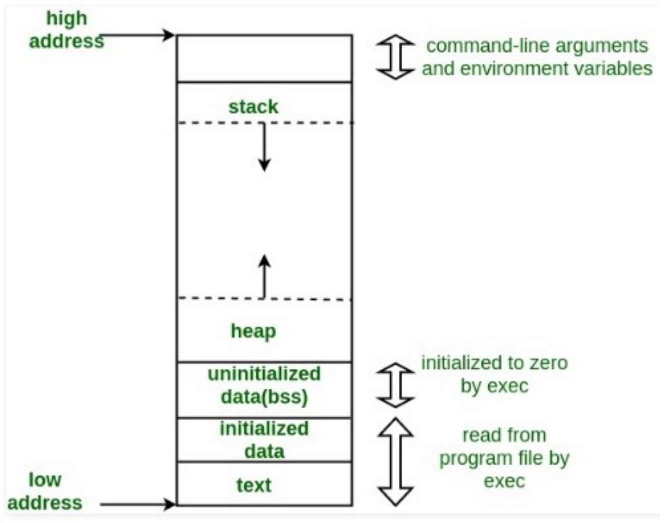
We can also use *std::tie* to unpack a tuple into variables:

```
#include <iostream>
#include <string>
#include <tuple>

int main()
{
  int x;
  std::string y;
  double z;

  std::tuple<int, std::string, double, int>  mytuple;

  // assign to mytuple
  mytuple = {5, "hello world", 90.67, 0};

  //unpack the tuple
  std::tie(x,y,z, std::ignore) = mytuple;

  std::cout << "x: " << x << std::endl;
  std::cout << "y: " << y << std::endl;
  std::cout << "z: " << z << std::endl;

  return 0;
}

# output
x: 5
y: hello world
z: 90.67
```

Memory Layout Of C And C++ Programs

A typical C/C++ program in memory consists of following sections:

- Text Segment or Code Segment
- Data Segment
- BSS Segment (Block Started by Symbol)
- Heap Segment
- Stack Segment


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

The Text Segment: also known as the Code Segment contains the machine language program code that will be executed. Typically this segment is shareable and read-only.

The Data Segment: also called the initialized data segment contains the global variables and static variables that have been initialized by the program. It may be further sub-divided into a read-only section for program constants and a read-write section for initialized variables that can be changed.

The BSS Segment: is also called the uninitialized data segment. *BSS* stands *for Block Started by Symbol*. This segment is initialized to arithmetic zero by the operating system, before the program starts executing. It contains global and static variables that have not been explicitly initialized to zero.

The Heap Segment: This segment of memory is used by the program to allocate dynamic memory using *malloc*, *calloc*, *realloc* or *new*.

Stack Segment: This segment of memory grows downward. It stores local variables and function arguments when functions are called. This memory segment is managed by a stack pointer. The stack segment behaves like a LIFO (*last in, first out*) container. The Stack is said to be exhausted when the stack pointer meets the heap.

We can get the memory layout of a program with *gcc*. Consider the program *memory.c*:

```
#include <stdio.h>

 /* Uninitialized variable stored in the BSS segment*/
int global;

// parameter z stored on the stack
int Foo(int z) {
  // Initialized static variable stored in the Data Segment
  static int i = 100;
  // local variable stored on the stack
  x = 5;
  x += z;

  // return value stored on the stackq
  return x;
}

int main() {
  Foo(5);
  // dynamic allocation on the heap
  float* y = new float;
  free y;
}
```

We can get the memory layout of this program with:

```
gcc memory.c  -o memory-layout
```

C++ GNU C++ Compiler

Check The GNU C++ Compiler Version

```
$ g++ --version
```

The Difference Between gcc and g++

GCC or GNU Compiler Collection Refers to all the different languages that are supported by the GNU compiler.

```
gcc is the GNU C Compiler
g++ is the GNU C++ Compiler
```

The main difference between these compilers is:

- gcc will compile: *.c and *.cpp files as C and C++ files respectively.
- g++ will compile: *.c and *.cpp files but the C files will all be treated as C++ files.
- if we use g++ to link the object files created by the compiler, it will automatically link in the Standard C++ library (gcc does not do this).

g++ Semantics

Compile And Run A Program

```
g++  -a helloworld helloworld.cpp foo.cpp bar.c
```

The compiler will compile the source files *helloworld.cpp*, *foo.cpp* and *bar.c* and create the executable file *helloworld*. If the output file is not specified with the -*a* option, the compiler will create the executable file *a.out*.

Compile And Output An Object File

```
g++  -c  foo.cpp
```

The compiler will compile *foo.cpp* and output the object file *foo.o*.

Compile A Program And Link A Library

```
g++ -o foo  foo.cpp -lmath
```

The -*l* option links the library file *libmath.a* or *libmath.so* to the object file *foo.o* and produces the executable file *foo.*

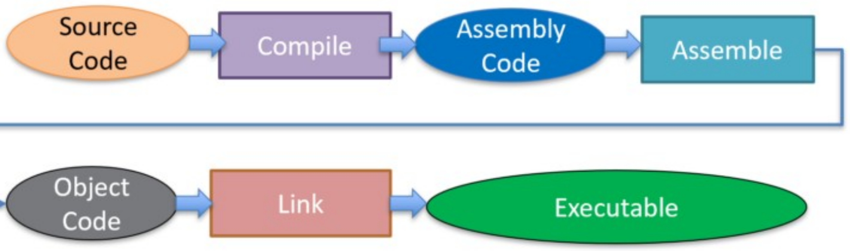
Since the -L option is not specified, the library file must be in a standard compiler defined location such as /*usr*/lib, for example.

gcc and g++ Options

-std	Specifies the C++ standard to use with g++. For example: *-std=c++11*
-std=c99	Compile using ANSI C99. For example: gcc hello.c -lsqlite3 -std=c99
-fsyntax-only	Check the source file for syntax errors only.
-S	Create an assembly language file from a source file.
-g	**Turn on gdb debugging.**

-O or -O2	**Turn on optimizations.**
-Wall	Turn on most compiler warnings.
-Wextra	Enable some extra warning flags.
-Wfatal-errors	Abort on the first error.
-Werror	Make all warnings into errors.

The Compilation Process



Unknown attribution

We can get the assembly code file from a source code file as follows:

g++ -S foo.cpp

An object code file is a file containing instructions in machine language. An executable file is a special type of machine code file. Executable files are usually in a specific format that the operating system can understand. An executable file has a section called the text section. This is where the machine code for the program is stored. In Unix and Linux, the machine language file (the object file) is called an executable and linkable format file or ELF file.

Creating Static And Shared Libraries

Static And Shared Libraries

A library is a collection of pre-compiled object files that can be linked into a program by a linker.

There are two types of libraries: *static libraries* and *shared libraries*.

A static library has the file extension "`.a`" (*archive file*) in Unix. When a program is linked against a static library, the machine code of the library functions that are used in the program is copied into the executable program.

A shared library has the file extension ".so" (*shared object*) in Unix. When a program is linked against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code of the external library functions that are used by the program. This process is known as dynamic linking. Dynamic linking makes executable files smaller and saves disk space, because only one copy of a library is shared by multiple programs. Furthermore, most operating systems allow one copy of a shared library in memory to be used by all running programs, thus, conserving memory. The shared library can be upgraded without the need to recompile the main program.

One major advantage of static libraries is speed. There is no dynamic querying of symbols in static libraries.

Header File And Library Location

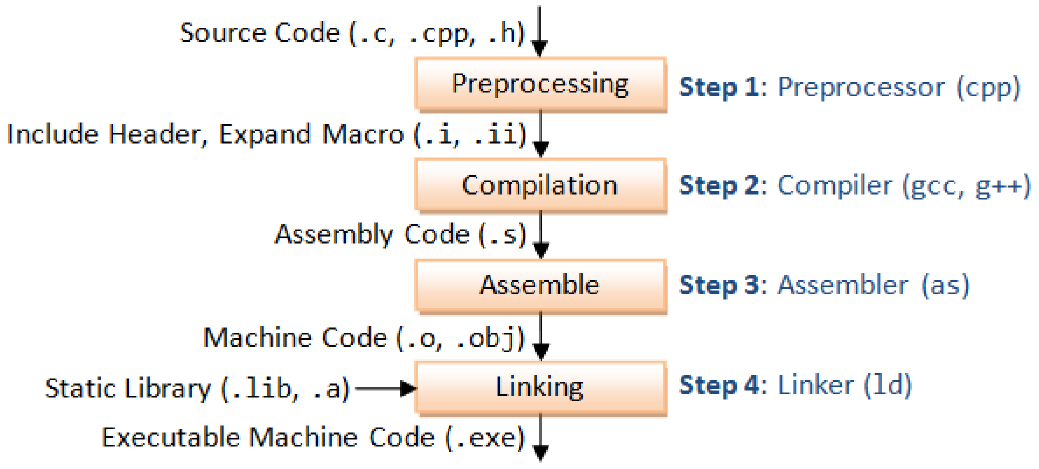
While compiling a program, the compiler needs the header files to compile the source code.

The compiler searches the include paths of header files in order to resolve external references. Include paths are specified on the compiler command line by the *-Idir* option (uppercase I followed by a directory path) or by the environment variable PATH. Since the header filenames are known (e.g., *iostream.h*, *stdio.h*), the compiler only needs the include directory paths.

The linker searches the library paths for libraries that have to be linked into an executable. The library path is specified by the *-Ldir* option (uppercase 'L' followed by the directory path) or by the environment variable LIBRARY_PATH.

In addition, we also have to specify the library names of the static libraries that are to be linked into the executable . In Unix, the static library *libxxx.a* is specified with the *-lxxx* option (lowercase letter '*l*', without the prefix "*lib*" and the "*.a*" extension). The *-l* flag indicates the name of the library. *-l* assumes that the library name starts with *lib* and ends with *.a* (so *lib* and *.a* must not be specified).

Compiling And Linking A Static Library


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Attribution unknown

When a C or C++ program is compiled, the compiler generates object code. After generating the object code, the compiler invokes the linker. One of the main tasks of the linker is to make the object code of library functions available to a program. A linker can accomplish this task in two ways. Firstly, by merging the library function code into our object code (static library), or by making some arrangement so that the complete object code of library functions is not merged, but made available at run-time (dynamically linked library).

In static linking the linker adds the object code of library functions to the executable file.

Creating A Static Library

Step 1: Create The Source Files

Create the source C++ files containing the library functions:

```
/* Filename: foo.cpp */
#include <stdio.h>

void funfoo(void)
{
  printf("called funfoo() in a library");
}

/* Filename: bar.cpp */
#include <stdio.h>

void funbar(void)
{
  printf("called funbar() in a library");
}
```

Step 2: Create The Header File For The Library

Create the include file for the library source files:

```
/* Filename: foobar.h */

 void funfoo(void);
void funbar(void);
```

Step 3: Compile The Library Source Files

Compile the library source files and produce object files:

```
gcc -c foo.cpp   -o foo.o
gcc -c bar.cpp   -o bar.o
```

If we are creating a dynamic library, the two library source files must be compiled with the *-fPIC* option. PIC is an acronym for position independent code:

```
gcc -fPIC -c foo.cpp -o foo.o
gcc -fPIC -c bar.cpp -o bar.o
```

Step 4: Create The Static Library

Create the static library by bundling all of the object files into one static library:

```
ar rcsv  some/path/libfoobar.a  path/foo.o path/bar.o
```

ar is the gnu archive program. The options are: r – replace, c – create an archive, v – verbose, s – create an index. The paths to the object files are specified on the command line.

Step 5: Link The Static Library Into A Program

We can use the library *libfoobar.a* as follows. Create a main program:

```
/* filename: driver.cpp  */
 #include "libfoobar.h"
 void main()
 {
   fun();
   bar();
 }
```

Compile *driver.cpp* and get the object file *driver.o*.

We link *driver.o* and the library file *libfoobar.a:*

```
gcc   bin/driver.o  -Lbin/static  -lfoobar   -o bin/driver
```

If the library exists in a non-standard location, the directory where the static library is located must be specified with the -*L* option.

The -l flag indicates the name of the library. -*l* assumes that the library name starts with *lib* and ends with *.o* (so *lib* and *.o* must not be specified).

The linking process creates an executable file called *driver*.

Alternative Step 4: Create A Shared Library

A shared library is created with GCC's -*shared* flag and by naming the resultant file with the filename extension *.so* rather than *.a*:

```
g++   -shared  bin/static/foo.o bin/static/bar.o    - o shared/libfoobar.so
```

Alternative Step 5: Link To The Shared Library

We dynamically link with the shared library *libfoobar.so* as follows:

```
gcc   bin/driver.o  -Lbin/shared  -lfoobar   -o bin/driver
```

The options -L and -l have the same meanings as before. If the library is in a standard location such as *usr/lib* for example, we do not need to use the -*L* option.

GCC Environment Variables

- **PATH**: search paths for executables and run-time shared libraries (.so).
- **CPATH**: search paths for include files. It is searched after paths specified in *-I<dir>* options.

  C_INCLUDE_PATH and **CPLUS_INCLUDE_PATH** can be used to specify search paths for header files in the C or C++ languages.
- **LIBRARY_PATH**: search paths for libraries. It is searched after paths specified in *-L<dir>* options.

Finding A Shared Library

On Unix systems, GCC first searches for libraries in */usr/local/lib*, then in */usr/lib*. Following that, it searches for libraries in the directories specified by the *-L* parameter, in the order specified on the command line. The GCC loader (*ld*) also looks for shared library files in these standard locations.

If we install a shared library in a standard location, make sure that the file permissions are correct. For example:

```
$ chmod 0755 /usr/lib/libfoo.so
```

Setting The Shared Library Location With rpath

rpath, or the run path, is a way of embedding the location of shared libraries in the executable itself, instead of relying on default locations or environment variables. We do this during the linking stage.

```
gcc -L/home/username/foo -Wl, -rpath=/home/username/foo -Wall -o test
main.c -lfoo
```

Notice the *-Wl, -rpath=/home/username/foo* option. The *-Wl* portion sends comma-separated options to the linker, so we tell it to send the *-rpath* option to the linker with our working directory.

Updating The Loader Config

Once we have placed our shared library in a standard location, with correct permissions; we have to tell the loader that it is available for use. We do this by updating the loader cache:

```
$ ldconfig
```

This will create a link to our shared library and update the cache so that the shared library is available for immediate use. We can test that the shared library is in the loader cache or equivalently installed as follows:

```
$ ldconfig -p | grep foo

libfoo.so (libc6) => /usr/lib/libfoo.so
```

List The Shared Libraries Used By A Program

ldd prints the shared libraries that are by a program.

```
ldd [-r] myProgram            // -v for verbose
```

This will print the paths of the shared libraries being used by *myProgram*. The -r option causes *ldd* to list shared libraries that are indirectly used by *myProgram*.

We can verify that our executable program is using the */usr/lib* instance of *libstdc++* library by using *ldd*:

```
$ ldd myProgram | grep libstdc++

libfoo.so => /usr/lib/libstdc++.so (0x00a42000)
```

C++ GDB Debugger

Basic Debugging

In order to debug a program, the program must be compiled with debugging symbols. Debugging symbols map program identifiers to memory addresses and line numbers in the source code. These debugging symbols are present in the object files generated by the compiler. The *-g* compiler option instructs g++ to generate debugging symbols:

```
$> g++ -g -std=c++20 –o hello hello.cpp
```

Once a program has been compiled with debugging symbols, we can start the GDB debugger:

```
$> gdb hello
```

The debugger will read the debugging symbols and stop at a debugger prompt.

gdb> is the debugger prompt.

Typical usage of *gdb* is as follows: After starting *gdb*, we set breakpoints. Breakpoints are locations in the source code where we want execution to pause. Each time gdb encounters a breakpoint, it suspends execution of the program at that point, giving us an opportunity to check the values of various variables. When we reach a breakpoint, we can single step forward from the breakpoint, which means that *gdb* will execute one line of the source code at a time and then pause.

The run command causes gdb to execute the program to the next breakpoint:

```
gdb> run                 (or gdb> r)
```

If we use the *start* option (*$> gdb start*) then *gdb* will stop program execution at the *main* function.

Command Completion

gdb will try to complete a command if we press the *tab key* after partially entering a command.

Debugging An Already Running Process

If a program is already running, we can debug it by attaching the gdb debugger to the program's process:

```
$ gdb

gdb> attach process_id
```

Setting A Breakpoint

The command ***break [function name]*** pauses a running program during execution when it reaches the named function. The following syntactical forms are available for the break command:

- `gdb> b`
- **`gdb> break [function name]`**
- `gdb> break [file name]:[line number]`
- `gdb> break [line number]`
- `gdb> break *[address]`
- `gdb> break ***any of the above arguments***` **`if [condition]`**

For example, if *square(int x)* is a function, we can do:

```
$> gdb break square if x == 5
```

Note: break at a scoped function by using the scope resolution operator, for example:

```
gdb> b account::balance
```

We can delete all of the breakpoints with:

```
gdb> delete
or
gdb> d
```

A particular breakpoint can be deleted with:

- *gdb> clear [line number]*
- *gdb> clear [function_name]*

We can get information on all of the breakpoints with:

```
gdb> info
```

Note: the GDB debugger can be used to debug threads.

Saving Breakpoints Between Debugging Sessions

If you are engaged in a complex debugging session, you will most likely have set up multiple breakpoints in a number of source files. Furthermore it will most likely take more than one debugging session to locate the bug. In such a situation it is completely frustrating as well as time consuming to have to set up the breakpoints whenever you start a new session. GDB provides a way to avoid this problem.

Firstly set up your breakpoints in the normal manner. Then save your breakpoints to a file:

```
gdb> save breakpoints breakpoints.txt
```

Now when you start a new debugging session, load the saved breakpoints:

```
gdb> source breakpoints.txt
```

Stepping Through A Program

While we are stopped at a breakpoint, we can step through a program, one program line at a time.

Both the *next* and *step* commands tell *gdb* to execute the next line of the program, and then pause again. If that line is a function call, then *next* and *step* will give different results. If we use *step*, then the program will pause will be at the first line of the function; if we use *next*, then the program will pause will be at the line following the function call.

```
gdb> next          // gdb n
or:
gdb> s             // gdb s
```

Stepping Out Of A Function

If we have used *step* to step into a function, we can use gdb> *finish (or gdb fin)* to step out of the function.

Listing Program Statements

When we are at a breakpoint, we can list the program line at which we are stopped along with a few lines around it:

```
gdb> l
```

We can view program lines in a different source file with:

```
gdb> l filename:line_number
```

We can list a few lines of a function with:

```
gdb> l function_name
```

Continuing The Debugging Session

After we have examined the code at a breakpoint, we can continue the debugging session with:

```
gdb> continue          (or gdb> c)

or

gdb> continue repeat count
```

If we just press the *return* key by itself then *gdb* will repeat the previous command.

Restarting A Debugging Session

We can restart an entire debugging session with:

```
gdb > restart
```

Examining And Setting Variables

When we are at a breakpoint, we can examine a variable such as *foo* with:

```
gdb> print foo

or

gdb> p foo
```

We can show the values of all the arguments of the currently executing function with:

```
gdb> info args
```

The current value of all local variables can be displayed with:

```
gdb> info locals
```

We can set the value of a variable as follows:

```
gdb> set var foo = 5

or:

gdb> p foo = 5
```

Watching Variables

Before a debugging session starts or while we are at a breakpoint, we can set a watch on a variable as follows;

```
gdb> watch balance

gdb> run or continue
```

The debugger will stop at the point where the value of *balance* changes.

Quit A Debugging Session

```
gdb> quit
or:
gdb> control-d    (control-d is the end of file character)
```

The .gdbinit File

Whenever the GDB debugger is started, it searches for the file *.gdbinit*, first in the user's home directory and next in the current working directory. If the file is found then the commands listed in the file will be executed. Place one command per line in this file.

For example, if we want the debugger to always stop at main(), we can insert the following GDB command into *.gdbinit*:

```
gdb> b main.cpp
```

Tracing Program Execution

Suppose we encounter a critical error during program execution that crashes the program. We can locate the program statement at which this error occurred with the *backtrace* command. At the debugger prompt, enter:

```
gdb> bt
```

bt traces the function calls leading to the critical error.

Debugging Core Dumps

Serious errors such as a segmentation fault will abort program execution and result in the debugger outputting a core dump. The core dump file contains all of the information necessary to create the state of the program at the point where the program aborted. We can use *gdb* to access the information in the core dump by passing the core dump file name as the second parameter to *gdb*:

```
gdb> hello core.file
```

Debugging A Segmentation Fault

Consider the following program:

```
// foo.cpp

#include<iostream>
using namespace std;

int main()
{
  int x = 7;
  int *p = 0;

  cout << "x = " << x;
  cout << "The pointer points to the value " << *p;
}
```

This program will abort with a segmentation fault when it is compiled and executed. We can debug this program as follows:

Compile the program with debugging symbols:

```
g++ -g foo.cpp
```

The compiler will produce the file ***a.out***

We run the program:

```
./a.out
```

The debugger will now emit, the following:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400824 in main () at 227.cpp:13
13 cout << "The pointer points to the value " << *p;
```

We can print the value of the pointer p:

```
gdb> print p

$1 = (int *) 0x0
```

The value of the pointer is 0.

Lets look at code near to the statement where the segmentation fault occurred:

```
gdb> list

int x = 7;
int *p = 0;
cout << "x = " << x;
cout << "The pointer points to the value " << *p;
```

Quit the debugger session:

```
gdb> quit
```

SIGSEGV Signals

When we receive the signal SIGSEGV (segmentation fault), the gdb debugger populates the *$_siginfo* structure. This structure can be examined to determine the faulting address:

```
gdb> p $_siginfo._sifields._sigfault.si_addr
```

A value such as: $1 = (void *) 0x0 indicates a null pointer.

GDB Errors

GDB Error: "The program is not being run"

Solution: This indicates that the debugging session has not begun. Start the debugging session with:

```
gdb> run or start   (or gdb> r)
```

C++ Sanitizers

In this section we look at two sanitizers that are supported by GCC, the *Address Sanitizer* and the *Undefined Behavior Sanitizer*. The *Address Sanitizer (Asan)* detects memory errors while the *Undefined Behavior Sanitizer (UBSan)* detects some undefined behavior errors. Both sanitizers require the source code to be instrumented and the appropriate sanitizer library to be linked in. These sanitizers detect errors at run-time.

Note: the chapter on smart pointers provides an extensive discussion of the types of memory errors that can occur.

The Address Sanitizer (Asan)

The address sanitizer detects the following types of memory errors:

- Failure to free a memory allocation (*memory leak*)
- Using a pointer after it's memory has been freed (*dangling pointer*)
- Writing to a memory location not allocated to a pointer (*memory corruption*)
- Deleting a pointer to memory more than once (*double free*)
- using *free* twice to deallocate memory that has been allocated with *new*.

Asan works by using shadow memory. Shadow memory is memory that tracks memory allocated or freed by a program.

In order to use Asan, the source code must be instrumented for the sanitizer. That is, the appropriate compiler flag must be used and the address sanitizer library linked in.

For GCC and CMAKE, the *CMakeLists.txt* file will have the following syntax:

```
target_compile_options(${PROJECT_NAME} PRIVATE -fno-omit-frame-pointer
                                        -fsanitize=address
                                        -g3 )

target_link_libraries(${PROJECT_NAME}

                      -fsanitize=address

                     )
```

The compiler flag *g3* sets the amount of debugging information (symbols) that the compiler will insert in the object file(s). *g, g1, g2* and *g3* are permissible flag values. *g3* produces the maximum amount of debugging information.

Undefined Behavior Sanitizer (UBSan)

The Undefined Behavior Sanitizer (UBSan) is a run-time sanitizer that tests whether an operation has unknown semantics. Some examples of such operations are: divide by zero, out of bounds access of an array, using a null pointer, integer overflow and dangerous narrowing casts. The result of such an operation may cause a segmentation error, return some garbage value or return a seemingly correct value. This type of bug is hard to detect since it may cause a program to crash only sporadically.

UBSan uses a catalogue of predefined undefined error types that it uses to check program expressions.

For GCC and CMAKE, we enable UBSan with the following syntax in the *CMakeLists.txt* file:

```
target_compile_options(${PROJECT_NAME} PRIVATE -fno-omit-frame-pointer
                                        -fsanitize=undefined
                                        -g3 )

target_link_libraries(${PROJECT_NAME}

                      -fsanitize=undefined

                     )
```

SonarLint

A linter is a static code analysis tool that examines program statements and flags statements that though syntactically correct, are not idiomatic C++ or have a better or safer syntactic constructions. Use of a good linter is an excellent way to improve the quality of a code base.

SonarLint is an excellent linter. SonarLint integrates seamlessly with Visual Code. Install it from the VisualCode Extensions menu.

SonarLint examines our code while it being written in the code editor. An expression that is not idiomatic C++ or those syntax could be improved will be indicated by an squiggly line under the expression. If we right click on the squiggly line, SonarLint will explain what the problem is with the expression and also suggest the solution.

The best practise is to use a linter such as SonarLint to monitor code quality in real-time.

C++ main Function

Every C and C++ program must contain a *main* function. Program execution starts from the *main* function. The signature of *main* is predefined by the compiler, it is:

```
int main(void)
or
int main(argc, char *argv[])
```

main cannot be called from within a C++ program and it cannot be qualified with the virtual, static or inline attributes.

argc is a count of the number of elements in *argv*. It is always greater than or equal to one.

The first element *argv[0]* is the command to start execution of the program. By convention, *argv[0]* is the name of the program. *argv[0]* does not have to be specified, it is implicitly taken from the command line. The rest of the arguments of *argv* are null-terminated strings representing command-line arguments passed by the user on the command line to the program.

Misc: How To Call A Function Before main

C and C++ programs start execution at the function *main()*. The following code will call the function *func* before *main()* is executed:

```
#include <iostream>

void func() { std::cout << "inside func" <<  std::endl; }

class Base {
 public:
    Base() { func(); }
};

Base foo;

int main() {
    std::cout << "inside main" << std::endl;
}
# output
inside func
inside main
```

The global variable *foo* will be initialized by the compiler before *main* is called.

We can also use a static global variable to call *func* before main:

```
#include <iostream>

int func()  {
   std::cout << "in func" << std::endl;
   return 0;
   }

static int foo = func();

int main() {
   std::cout << "in main" << std::endl;
   return 0;
}

# output
in func
in main
```

The compiler will initialize the global variable *foo* before calling *main()*.

Misc: Hide The Address Of An Object

We want to prevent the address of an object from being disclosed. There are two ways to do this:

- Make the address operator **&** of the class private
- delete the address operator of the object

The following example makes the address operator private:

```
#include  <iostream>

class base {
public:
    base() {  std::cout << "inside base" << std::endl;  }
private:
    base*  operator &() {return this;}
};

int main() {
   base foo;
   base*  bptr = &foo;     // will not compile
   bptr = nullptr;
   }
```

We can also delete the address operator:

```
#include  <iostream>

class base {
public:
    base() {  std::cout << "inside base" << std::endl;  }
    base*  operator &() = delete;
};

int main() {
   base foo;
   base*  bptr = &foo;     // will not compile
   bptr = nullptr;
}
```

Misc: Inheritance And Composition

Inheritance is a "*is-a*" relationship. Composition is a "*has a*" relationship. A class can relate to other classes through inheritance, composition or both.

Consider the three classes:

```
class  Mammal {
   ...
}

class Feet {
     int number_of_feet;
     int number_of_toes;
}

class Dog : public Mammal {
  class dog_feet*   new Feet();
}
```

A *Dog* is a M*ammal* and thus inherits the public variables and methods of the class Mammal. A *Dog* has feet so the *Dog* class has an instance of the class F*eet* called *dog_feet*, this is a composition relationship.

C++ Glossary

Application Binary Interface (ABI)

This refers to the manner in which program data is laid out in memory:

- The layout of function parameters
- the layout of data
- the layout of classes and structs
- the layout of virtual functions
- the return type

Consider the following example which breaks the ABI of a library. We have a struct in the library:

```
struct {
        int a;
        int f;
      }
```

Our binary executable links in the library and the application accesses the data member *a*. The binary executable gets the 4 bytes at offset 0 from the beginning of the struct.

Now suppose that we change the struct as follows:

```
struct {
        int f;
        int a;
      }
```

And link the new library (the binary library code) into our executable. Now our executable will access *f* instead of *a*. Actually we need to access the 4 bytes beginning at an offset of 4 bytes from the start of the struct. The new library has broken it's ABI. The solution is to recompile the library; but this requires access to the source code. If we do not have access to the source code, we cannot change the order of the data members *f* and *a* if we want to preserve the application binary interface. There is a way around this problem, if we implement a C++ wrapper around the library object file.

Copy Elision

Copy Elision (sometimes called return value optimization) is an optimization technique where, in certain specific circumstances, a compiler is permitted to avoid a copy or avoid moving a variable even though the C++ standard says that this must happen.

Consider the following function:

```
std::string get_string()
{
return std::string("I am a string.");
}
```

According to the strict wording of the C++ standard, this function will create and initialize a temporary *std::string* object on the stack and then copy the object onto the stack when the return statement is executed, and finally destroy the temporary object. The standard is very clear that this is how the code is implemented.

Copy elision is a rule that permits a C++ compiler to ignore the creation of the temporary object and it's subsequent copy and destruction. That is, the compiler can take the initializing expression for the temporary object and directly place it on the stack. Copy Elision is permitted because it improves performance.

Toolchain

A sequence of programs that converts source code into an executable program.

The canonical C and C++ toolchain is:

- `Preprocessor     Processes the preprocessor symbols in a source file.`

• Compiler	Generates assembly language code from a preprocessed source code file.
• Assembler	Converts an assembly language file into machine code (an object or .obj file).
• Linker	Joins object files to create an executable file.

Machine code is simply a sequence of machine language instructions that a CPU can understand without any further translation. In other words a machine language instruction is a sequence of zeros and ones.

Translation Unit

In C and C++, a translation unit is the final input into a C or C++ compiler. The compiler processes a translation unit and generates an object file from it.

A translation unit consists of a source file after it has been processed by the C or C++ preprocessor, meaning that the header files listed in the *#include* directives have been literally included, sections of code within *#ifdef* sections have been included, and macros have been expanded. A single translation unit is compiled by the compiler into a single object file.