

This document is for the sole use of Vignesh Anand of Ncsu

C++ Language Fundamentals

Course Version 24.03

Lecture Manual

Revision 1.0

cadence®

This document is for the sole use of Vignesh Anand of Ncsu

© 1990-2024 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

The publication may be used solely for personal, informational, and noncommercial purposes;

The publication may not be modified in any way;

Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and

Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence customers in accordance with, a written agreement between Cadence and the customer.

Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Table of Contents

C++ Language Fundamentals

Module 1	About This Course	2
	Lab 1-1 Executing C++ Code Using g++ Compiler/GDB Debugger and Microsoft Visual Studio	
Module 2	C Language Review	12
Module 3	Object-Oriented Programming and C++.....	38
	Lab 3-1 Organizing an OOP Project	
Module 4	C++ Basics.....	51
	Lab 4-1 Writing a Simple C++ Program	
Module 5	Constructors and Destructors	69
	Lab 5-1 Defining Constructors and a Destructor	
Module 6	References.....	82
	Lab 6-1 Experimenting with Reference Variables	
Module 7	Functions.....	93
	Lab 7-1 Defining Class Member Functions	
Module 8	Type Conversion	111
	Lab 8-1 Exploring Cast Operations	
Module 9	Operator Overloading	123
	Lab 9-1 Overloading Member Operators	
Module 10	Inheritance.....	141
	Lab 10-1 Deriving Subclasses	
Module 11	Polymorphism	155
	Lab 11-1 Defining and Using Polymorphic Classes	

Module 12 Constant Objects and Constant Functions.....	167
Lab 12-1 Experimenting with Constant and Mutable Objects	
Module 13 Templates	179
Lab 13-1 Defining a Class Template	
Module 14 Exceptions	193
Lab 14-1 Throwing and Catching Exceptions	
Module 15 Input and Output	208
Lab 15-1 Inputting and Outputting Program Data	
Module 16 Debugging	229
Lab 16-1 Debugging a Program	
Module 17 Containers and Algorithms	262
Submodule Standard Containers	264
Submodule Standard Almost-Containers	303
Submodule Standard Algorithms	319
Lab 17-1 Instantiating and Using Standard Containers	
Module 18 Introduction to System-C	343
Lab 18-1 Implementing FIR Filter in C++ and Running Stratus Tool to Generate the RTL	
Module 19 Equivalence Checking C++ for Verification	372
Lab 19-1 Equivalence Checking of a Pipelined Multiplier	
Module 20 Course Conclusions	389
Module 21 Next Steps.....	392

C++ Language Fundamentals

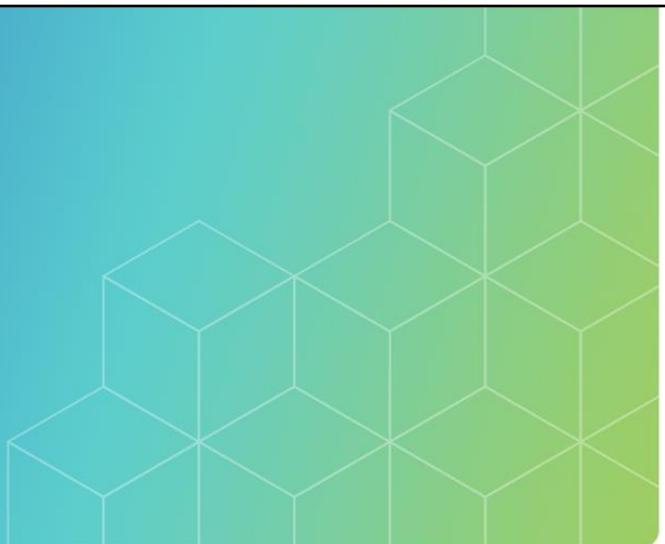
Version 24.03

Revision 1.0

Estimated time: 3 Days

cadence®

This page does not contain notes.



Module 1

About This Course

cadence®

This page does not contain notes.

Course Prerequisites

Before taking this course, you need to

- Have a basic working knowledge of the C programming language or another procedural programming language
- Have a basic knowledge of Digital IC design and verification



This page does not contain notes.

Course Objectives

In this course, you

- Develop fundamental C++ programming skills
- Run C++ designs through Electronic Design Automation (EDA) tools, namely:
 - Cadence Stratus™ – High-Level Synthesis
 - Cadence Jasper™ – C++ to RTL equivalence check



This page does not contain notes.

What Is C++?

- A high-level, general-purpose programming language.
- It has been around for 40 years.
- C++ is ISO/IEC standard 14882, which is regularly updated.
 - For example, C++11, C++14, C++17, C++20 where the number indicates year of release.
- The updates introduced significant changes to the language.
- Even inside the same company, different groups use different standards.
- This means that very different-looking C++ code gets created.
- There is no one size fits all.



This page does not contain notes.

Purpose of the Course

- There are many C++ courses available in every region of the world.
- Mainly, they are aimed at programmers creating applications.
- This course is aimed to teach programmers whose end use is in Digital Integrated Circuit (silicon chips) design and verification.
- Namely, people who create IC designs and/or verify them using C++.
 - Hence, this course differs from most others.

In this course, you will create C++ designs through EDA tools in order to:

- Create Digital IC designs.
- Verify that Digital IC designs are correct.



This page does not contain notes.

Course Agenda

- About This Course
 - Executing C++ Code Using g++ Compiler/GDB Debugger and Microsoft Visual Studio
- C Language Review
- Object-Oriented Programming and C++
 - Organizing an OOP Project
- C++ Basics
 - Writing a Simple C++ Program
- Constructors and Destructors
 - Defining Constructors and a Destructor
- References
 - Experimenting with Reference Variables
- Functions
 - Defining Class Member Functions
- Type Conversion
 - Exploring Cast Operations
- Operator Overloading
 - Overloading Member Operators
- Inheritance
 - Deriving Subclasses
- Polymorphism
 - Defining and Using Polymorphic Classes
- Constant Objects and Constant Functions
 - Experimenting with Constant and Mutable Objects
- Templates
 - Defining a Class Template
- Exceptions
 - Throwing and Catching Exceptions
- Input and Output
 - Inputting and Outputting Program Data
- Debugging
 - Debugging a Program
- Containers and Algorithms
 - Instantiating and Using Standard Containers
- Introduction to System-C
 - Implementing FIR Filter in C++ and Running Stratus Tool to Generate the RTL
- Equivalence Checking C++ for Verification
 - Equivalence Checking of a Pipelined Multiplier

7 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Software and Licenses

For the software and licenses used in the labs for this course, go to:

https://www.cadence.com/en_US/home/training/all-courses/82119.html

If there is additional information regarding the specific software, it is detailed in the lab document and/or the README file of the database provided with this course.



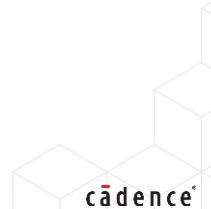
This page does not contain notes.



Lab

Lab 1-1 Executing C++ Code Using g++ Compiler/GDB Debugger and Microsoft Visual Studio

- Execute a sample C++ program based on scope, which shall be discussed in a later module (Module 4). Here, we only try to learn how to compile and debug using g++/gdb and Microsoft Visual Studio



Your objective for this lab is to execute a C++ program using g++/gdb and Microsoft Visual Studio.

For this lab, you

- Execute a sample C++ program based on scope, which shall be discussed in a later module (Module 4). Here, we only try to learn how to compile and debug using g++/gdb and Microsoft Visual Studio.

Become Cadence Certified by Earning a Digital Badge



Digital badges indicate mastery in a certain technology or skill and give managers and potential employers a way to validate your expertise.

- Cadence Training Services offers digital badges for our popular training courses.
- Your digital badge can be added to your email signature or social media platforms like LinkedIn or Facebook.

Benefits of Cadence Certified Digital Badges

- Validate expertise
 - Expand career opportunities
- Professional credibility
 - Stand apart from your peers
- For more information, go to www.cadence.com/training or email es_digitalbadge@cadence.com.



How do I register to take the exam?

- Log in to our [Learning Management System](#), click on the course in your transcript, and go to the Content tab to locate the exam.

How long will it take to complete the exam?

- Most exams take 45 to 90 minutes to complete. You may retake the exam multiple times to pass the exam.

How do I access and use the digital badge?

- After you pass the exam, you get a digital badge and instructions on how to place it on social media sites.

How is the digital badge validated?

- [Credly](#) validates the digital badge as issued to you by Cadence and includes the details of the criteria you completed to earn the badge.



10 © Cadence Design Systems, Inc. All rights reserved.

This page does not contain notes.

Icons Used in This Class



Best Practice



Language/
Command Syntax



Concept/
Glossary



Frequently Asked
Questions/
Quiz



Error Message



Problem & Solution



Quick Reference



GUI and Command



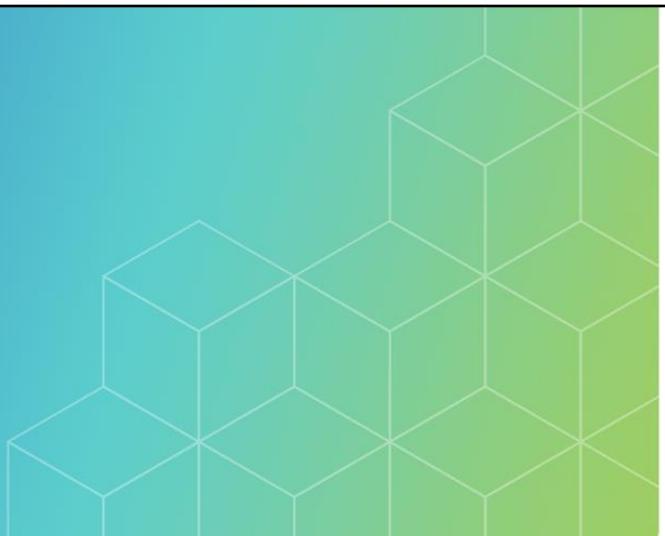
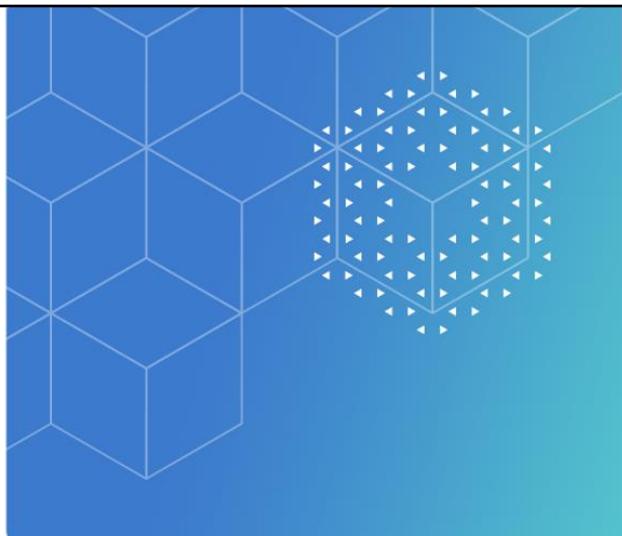
How To



Lab List

Throughout this class, we use icons to draw your attention to certain kinds of information. Here are the icons we use, and what they mean.

This page does not contain notes.



Module 2

C Language Review

cadence®

This page does not contain notes.

Module Objectives

In this module, you

- Write and execute a simple C program, demonstrating a solid understanding of the key constructs and concepts of the C programming language

Topics

- The `main()` Function
- Program Execution
- Data Types and Declaration
- Pointers and Arrays
- Structures and Unions
- Compiling and Linking with `gcc`



Your objective is that upon studying this module, your C programming skills should be sufficiently refreshed to write and execute a simple C program and then confidently continue with the C++ portion of this training. For this, the module discusses the following topics:

- The `main()` Function
- Program Execution
- Data Types and Declaration
- Pointers and Arrays
- Structures and Unions
- Compiling and Linking with `gcc`

The C main() Function



Every C program starts executing at a function called **main()**.

Every C program starts executing at a function called **main()**.

- Returns an **int** – typically an error code (0 means no errors).

```
#include <stdio.h>           // Include STD I/O library
int main(int argc, char** argv) // Entry point to program
{
    FILE *fptr;             // Pointer to FILE type
    int status, items, data[256]; // Signed integer variables
    fptr = fopen("file", "r+"); // Open "file" for update
    if (!fptr) return 1;       // Return if unsuccessful
    rw_status = fread(data, 4, 256, fptr); // Read file data
    compute_result(data, items); // User-defined function
    rw_status = fwrite(data, 4, 256, fptr); // Write modified data
    status = fclose(fptr);      // Close file
    return status;            // Return status
}
```

14 © Cadence Design Systems, Inc. All rights reserved.



The code on the slide includes the standard I/O library header file to provide definitions of, among other things, the FILE type and the fopen(), fwrite(), and fclose() functions.

The entry point to every C program is a function called **main()**. The C standard requires you to define it with an int return type and either no parameters or an int parameter and a char* parameter, in that order, to accept invocation string arguments.

The example declares two variables, and an array of 256 elements, all of the int type, opens a file called “file” for updating and reads 256 4-byte elements from the file into the array.

An array name is equivalent to a pointer of the array element type that points to the first element of the array, so passing the array to the user-defined compute_result() function is equivalent to passing the value of a pointer. Only the pointer, not the entire array, is copied to the function call's stack frame.

The example then writes the modified array data back to the file.

It is very common for the main() function to return an error status. What to do with the status is totally up to the caller. Applications typically interpret the 0 value to mean no errors.

Program Execution

C program execution is a path involving one or more functions.

- Executes as a single thread.
- Each function “call” returns to the point of its call.
- You can nest function calls to an arbitrary depth.

```
#include <stdio.h>
void compute_result(int *data_arg,int
status_arg);
int main()
{
    FILE *fptr;  int status,
    rw_status, data[256];
    fptr = fopen("file","r+");
    if (!fptr) return 1;
    rw_status = fread (data,4,256,fptr);
    compute_result(data,rw_status);
    rw_status = fwrite(data,4,256,fptr);
    status = fclose(fptr);
    return status;
}
```

```
void compute_result(int *data_arg,int
status_arg)
{
    ...
    ...
    ...
}
```

The C standard has a section on program execution. For your immediate purposes, knowing that the standard does not address multiple threads is sufficient. Absent any attempt on your part to multi-thread execution of your program, it executes in a single thread. That means that for any given input, the program takes exactly one path. Execution obeys your branch and loop statements and moves forward through the program. In the code on the slide, we see that the execution starts from the main, on-reaching function call, `compute_result` control is transferred to the function, `compute_result`. The control returns back to the main function to execute the next instruction following the function call.



Data Types

C provides a built-in set of basic data types:

- **integral types:** _Bool, char

Signed	signed char, short int, int, long int, long long
Unsigned	unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long
“size” suggested by word size of execution environment	

- **floating point:** float, double, long double
- **Void type:** void
- limits.h defines the range of values supported for any basic type.
- To ensure portability:
 - Write code that refers to these definitions.

Note: Sign of char is implementation-dependent. If important, specify signed or unsigned.



The C standard defines the basic types with sufficient ambiguity to cause portability problems if you are not careful using them. For example:

- The _Bool type must be sufficiently large to store the values 0 and 1. Do not rely upon it being any particular size.
- The char type must be sufficiently large to store the basic character set of the execution environment and, in any case, must be at least 8 bits. Implementations almost invariably use only 8 bits. The implementation may treat a char as signed or unsigned. Always specify signing for any character you use in arithmetic operations.
- The size of the int type is “suggested” by the native word size of the execution environment and, in any case, must be at least 16 bits. Implementations almost invariably use 32 bits.

The limits.h header file defines the range of values an implementation can represent using any specific basic type. To ensure the portability of your code, you should get used to writing code that refers to these definitions.

Note: sign of char is implementation-dependent. If important, specify signed or unsigned.

Enumerated Types

C supports user-defined enumerated types:

- It is a set of user-defined integer constants.
- The programmer defines the valid set of integer values.

```
enum tag-identifieroptional { enumerator-list };
enum shape_t { SQUARE, CIRCLE, TRIANGLE };
```

By default:

SQUARE=0, CIRCLE=0+1=1 and TRIANGLE=1+1=2

```
enum shape_t { SQUARE, CIRCLE, TRIANGLE };
float calc_area ( float width, shape_t shape )
{
    switch ( shape )
    {
        case SQUARE:
            return width * width;
            break;
        case CIRCLE:
            return 3.1416 * width * width / 4;
            break;
        case TRIANGLE:
            return 0.0; // you wish!
            break;
    }
}
```



An *enumerated type* is defined by its enumeration, that is, the set of named integer constant values.

The syntax for defining an enumerated type is the **enum** keyword optionally followed by an identifier tag followed by a list of enumerators within curly braces. An *enumerator* is a name, optionally assigned an integer constant value – the standard permits negative values and permits duplication of values. The enumerator values, by default, start at 0, and anywhere they are not initialized, the compiler supplies a value that is one more than that of the previous enumerator.

In the code in the slide, we see that `shape_t` is an enumerated data type that can take the values `SQUARE`, `CIRCLE`, and `TRIANGLE`; these are assigned the values 0, 1, and 2 by the compiler. We pass this enum as an argument to the function `calc_area`. `Shape` is the object of the type `enum shape_t`.

Pointers

A pointer is an object that holds an address to another object.

- Type of pointer and pointed-to object must be compatible.
- The object type can be of any complexity.
- Exchange pointers to avoid copying large objects.

```
// assume int j at location 0x1000
int i, j = 5;
int *int_p = &j; // 0x1000
*int_p = 9;
i = *int_p; // 9
```

Name	Address	Value
i	0x1004	0
j	0x1000	5
i	0x1004	0
j	0x1000	9
i	0x1004	9
j	0x1000	9

The "&" operator is called the "address" operator

```
int i = 5; int *int_p = &i; // int_p contains address of i
```

The "*" operator is called the "indirection" operator

```
int j; j = *int_p; // j gets value pointed to by int_p
```



A pointer is an object whose value references some other object. Every pointer has a type indicated by the type of the referenced object. The pointer value is an integer value representing the location in memory of the object being pointed to. As these locations have different alignments, pointer types are not necessarily compatible, and the compiler requires a type cast if you want to use a pointer to a particular object type to point to an object of some other type. Such casts may or may not produce results that you expect and thus are not portable between implementations.

The object that it points to can be of any complexity. We can avoid copying large objects by exchanging pointers.

Here, '&' is called the address operator. For example, &i mean the address of i.

The '*' operator is called the indirection operator. *p means contents stored at the address pointed by the pointer 'p.'

In the example, we declare two variables, i and j. i takes the default integer value, which is 0, and j is assigned the value 5. &j is nothing but the address of j. integer pointer, int_p is assigned with the address of j. We assign a new value 9 to the location pointed by the pointer int_p. Hence, the value of j changes to 9 as it points to the same memory location as int_p. Finally, you are assigned with the value stored at the address pointed by int_p, which is 9

Arrays

C supports arrays of any type and any number of dimensions.

- Arrays are stored in row-major order.
- You can access elements by index or by pointer.
- Treats occurrence of array name **without** subscript as a pointer.

```
float floatArray[5] = {0.0, 12.4, 4.33, 94.3, 67.4} ;
float floatVar = floatArray[3]; // 94.3
floatVar = *(floatArray+3); // 94.3
float *floatPtr = floatArray; // &floatArray[0]
floatVar = *(floatPtr+3); // 94.3
int intArray[2][5] = {0,1,2,3,4, // [rows][columns]
5,6,7,8,9};
printf("%d\n",intArray[0][4]); // 4
intArray[0][4] = 66;
int *intPtr = &intArray[0][3];
int intVar = *(++intPtr); // 66
```

intArray
(conceptually)

0	1	2	3	4
5	6	7	8	9

intArray
(as stored)

Index	Address	Val
[1][4]	0xfffffd6d0	9
[1][3]	0xfffffd6cc	8
-----	-----	--
[0][1]	0xfffffd6b0	1
[0][0]	0xfffffd6ac	0

19 © Cadence Design Systems, Inc. All rights reserved.



An array is a contiguous set of elements of a given type. Arrays can be any number of dimensions and any element type. Arrays are stored in row-major order. That means the last subscript varies fastest. For example, a 3x5 array (`a[3][5]`) is three arrays, each sub-array having five elements. You can access elements by index or by pointer. For example:

`floatArray[3]` implies 4th element of the array `floatArray`.

With few exceptions, an expression of an array type is treated as a pointer to the first element of the array.

The exceptions are:

- When used as an operand to the **size of** the operator, which yields the size in bytes of the array;
- When used as an operand to the address operator, which yields the address of the array and
- When used as a character string literal to initialize a character array.
- The example declares a float array containing five elements. We try to access the fourth element of the array using subscripts and using pointers. Both give the value of 94.3. We then declare a pointer of floating type and assign it with float array; i.e., it is assigned with the address of the first element of the array. We also access the elements using this pointer, which gives the same result as the previous one.
- Next, we declare an integer array of dimensions 2 x 5. The table shows how the elements of this array are arranged in the memory. We try to access the elements of this array using index and pointers.

Structures and Unions

- A **struct** is a collection of related members of potentially different types.

```
struct vehicle_s {  
    unsigned int num_wheels;  
    motor_t motor;  color_t color;  
    unsigned int serial_number;  
    _Bool air_conditioning;} ;
```

- A **union** is a collection of types representing one object.

```
union type_u {int i; float f;};  
enum type_t {INT, FLOAT};  
// one array with two potential uses  
Struct array_s {type_t tag; type_u data[100];};
```



A **struct** is a sequential set of members of potentially different types that are stored sequentially in increasing locations. A pointer to a structure points to the first member of the structure. The members may be of any type other than a variably modified type, such as a variable-length array.

A **union** is a sequential set of members of potentially different types that are stored overlapping in one location. The members may be of any type other than a variably modified type, such as a variable-length array. The compiler allocates sufficient storage to hold an object of the largest type.

The example depicts a structure that contains a tag member and an array member. The array element type is a union with either an int or float type. The programmer stores ints or floats in the array, but typically at different times. The programmer sets the tag member as a reminder about how the array is currently being used.

The standard provides a syntax not shown here for declaring a bit field, that is, a member having a user-specified bit width no greater than the bit width in which a member of that type is normally stored. An implementation can allocate storage for the bit field in any manner, providing it does not lose precision.

Structure and Union Member Access

- For expressions of the `struct` or `union`:
 - Use the member-access operator (`.`) to access members.
- For expressions of the `pointer-to-struct` or `pointer-to-union`:
 - Use the “pointer” operator (`>`) to access members.

Refers to *copy* of caller's vehicle

```
void my_func(my_vehicle_t vehicle)
{
    vehicle.num_wheels = 4;
    vehicle.color      = RED;
}
```

Refers *directly* to caller's vehicle

```
void my_func(my_vehicle_t *vehicle_p)
{
    vehicle_p->num_wheels = 4;
    vehicle_p->color      = RED;
}
```



For struct or union-type expressions, use the member-access (`.`) operator to access a member. The first operand of the dot operator is a struct or union object, and the second operand is a struct or union-type member. The resulting expression yields the value of the member.

For expressions of the `pointer-to-struct` or `pointer-to-union` type, use the “pointer” (`->`) operator to access a member. The first operand of the pointer operator is a pointer to a struct or union, and the second operand is a struct or union-type member. The resulting expression yields the value of the member.

Scope and Namespace



Scope is a region of program text in which a name is visible.

Namespace is syntactic context that disambiguates identifiers.

file scope	From point of declaration to end of the translation unit
function prototype scope	From point of declaration to end of prototype
function scope	Anywhere within function (but only for statement labels)
block scope	From point of declaration to end of enclosing block

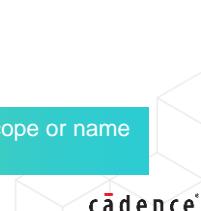
Ordinary identifiers:

- statement *labels*
- enum, struct, or union *tags*
- struct or union *members*

```
int i; // file scope
int main()
{
    int i=0; // block scope
    struct i { // tag namespace
        int i; // member name space    };
    ...
    i: // label name space
    while (i)
    { // block scope (outer)
        int i; // block scope (inner)
    } // -- hides outer decl
    return 0;
}
```

Note: The same name in a different scope or name space is a different entity!

22 © Cadence Design Systems, Inc. All rights reserved.



A scope is a region of program text in which an identifier is visible.

The various scopes are:

- File scope is from the point of declaration to the end of the translation unit;
- Function prototype scope is from the point of declaration to the end of the function prototype;
- Function scope encompasses the entire function body but applies only to labels and
- Block scope is from the point of declaration to the end of the enclosing block.

A namespace is a syntactic context that disambiguates identifiers. Most identifiers are *ordinary identifiers*, but you can unambiguously reuse the names in:

- Statement *labels*;
- enum, struct, or union *tags*; and
- struct or union *members*.

Reusing an identifier in a different scope or a different namespace designates a different entity.

In the example on the slide, variable *i* is declared in various scopes such as file scope, block scope, member name scope, label name scope, etc. Its usage in the while loop uses the inner block scope and hides all outer declarations.

Internal Linkage and External Linkage



Internal Linkage – The same entity in *one* translation unit of the program.



External Linkage – The same entity in *all* translation units of the program.

File scope function or object name with the **static** specifier.

```
static int i;  
void f (...)  
  
{  
    i = 1 ;  
    ...  
}  
void g (...)  
{  
    int j = i ;  
    ...  
}
```

```
extern int i;  
void f (...)  
{  
    i = 1 ;  
    ...  
}
```

```
extern int i;  
void g (...)  
{  
    int j = i ;  
    ...  
}
```

Linkage relates to how multiple declarations of an identifier all refer to the same entity:

- If the linkage is *internal*, the declarations refer to the same entity in *one* translation unit of the program. A translation unit is one file plus all other files included with the #include directive. An example of internal linkage is a file scope function or object name declared with the **static** specifier.
- If the linkage is external, then the declarations refer to the same entity in *all* translation units of the program. Examples of external links are all file scope functions or object names by default and other identifiers declared with the **extern** specifier if they have not already been declared with the static specifier in a still-visible scope.



Storage Duration

Storage duration determines the lifetime of an object:

- **Allocated:** persists from allocation to de-allocation.
 - Via `calloc()`, `malloc()`, `realloc()`, and `free()`
- **Automatic:** persists from block entry to exit – not implicitly initialized.
 - A block-scope object by default.
- **Static:** persists throughout the program lifetime – implicitly initialized.
 - An object declared with linkage or with `static` specifier.

```
void f ( ... )
{
    static int * ip ;           static
    if ( ip == 0 )             allocated
        ip = (int*) malloc(256) ;
    for ( int i = 0; ... )     automatic
    {
        ...
    }
    ...
}
```

24 © Cadence Design Systems, Inc. All rights reserved.



Storage duration relates to the lifetime of an object:

- Objects with *allocated* duration are those that you allocate via calls to `calloc()`, `malloc()`, or `realloc()`, and de-allocate via calls to `free()`
- Objects with *automatic* duration are those that you declare local to a function or block – they exist only while the program is executing that function or block
- Objects with *static* duration are either those that you declare with the `static` specifier or those that otherwise have internal or external linkage – they exist throughout program lifetime

Storage Class Specifiers

- static
 - Static (heap) variable. Has internal linkage if declared in the file scope.
- Extern
 - Static (heap) variable. Has external linkage if not previously declared static. File scope objects, by default, have external linkage.
- auto
 - Automatic (stack) variable. Specifiers seldom used as objects by default are automatic if not declared static and have no linkage.
- Register
 - Hint to the compiler that object may be stored in a platform register. You cannot retrieve the address of an object so declared. Specifier has an insignificant effect on most applications on most platforms.

25 © Cadence Design Systems, Inc. All rights reserved.



The **static** specifier designates an object with static storage duration and if declared in the file scope, with internal linkage.

The **extern** specifier designates an object with static storage duration and with external linkage if it has not previously been declared static. By default, objects declared in the file scope have static storage duration and external linkage.

The **auto** specifier designates an object with automatic storage duration. The specifier is almost never used because an object declared with no linkage and without the **static** storage-class specifier has automatic storage duration anyway.

The **register** specifier hints to the compiler that the object is frequently accessed so it should be stored in a platform register. Modern compilers rarely do so, but if they do, the program cannot obtain the address of a registered object.

Type Qualifiers

const

- An object the program cannot modify.

- `const unsigned WIDTH = 8;`

volatile

- An object the platform may modify (e.g., hardware timer).

- `extern const volatile unsigned clock;`

restrict

- A pointer whose value is the only value a function will directly or indirectly use to modify the pointed-to object. This facilitates optimization by the compiler.

- `void f(int n, int * restrict p, int * restrict q);`



The **const** qualifier designates an object that the program intends not to modify. The compiler may place the object in write-protected memory. Thus, any attempt to modify the object that the compiler does not detect will have undefined results.

The **volatile** qualifier designates an object that the platform may modify. The platform can modify the object at any time, thus, any reference to a volatile object must be through a volatile value to ensure that the value is current.

The **restrict** qualifier designates a pointer whose value is the only value a function will directly or indirectly use to modify the pointed-to-object. This qualifier supports dependency analysis during optimization and has no other effect.

Function Declarations



```
[inline] [linkage] type identifier ([parameter-type-list]);
```

```
[inline] [linkage] type identifier ([parameter-type-list]);
```

- **inline** “hints” to compiler to inline calls.
- **Linkage:** extern or static
 - You can declare any function **extern** and file-scope functions **static**.
- **Type:** _Bool, char, int, float, void, etc.
 - Use *pointer-to-type* to avoid passing large structures by value.
 - Convert to *pointer-to-const-type* to prevent modifying pointed-to object.
- Parameter type list: Pairs of *type identifier*
 - Can omit *identifier* in type declarations.
 - The last parameter can be ellipsis (...) to indicate the variable argument list.



You can include the **inline** function specifier as a “hint” to the compiler to inline function calls. The compiler may or may not inline some or all calls to the function.

You can include the **static** storage-class specifier to provide internal linkage for a file-scope function declaration or the **extern** storage-class specifier to provide external linkage for any function declaration.

You must specify a type for the function declaration. The type can be any built-in or user-defined type, except it cannot be an array or function type. You can instead return a pointer to the first array element or a pointer to a function definition. It is common to pass and return a pointer to a large structure instead of copying the structure into and out of the function call. If returning a pointer to a const object, then the return type must be *pointer-to-const-type*. You can also return a *pointer-to-const-type* to prevent the use of the pointer to modify the pointed-to-object.

You can omit the parameter list or, alternatively, explicitly specify a **void** parameter list. The parameter list otherwise is a list of declarations. You can omit the parameter identifier if declaring a type rather than defining a function. The standard also describes a variable argument list this training does not address.

Example: Function Declarations

```
#include <stdio.h>
int main() {
    typedef struct {int first[100]; int second[100];} struct_t;

    struct_t struct1;
    for (int i=0;i<=99;++)
    {
        struct1.first[i]=i;
        struct1.second[i]=99-i; } // -std=c99

    struct_t pass_value (struct_t param) { param.first[0] = 111; return param; }
    struct_t struct2 = pass_value ( struct1 ); // 200 int on stack
    struct2.second[0] = 222;
    printf ("%3d %3d\n",struct1.first[0],struct1.second[0]); // 0 99
    printf ("%3d %3d\n",struct2.first[0],struct2.second[0]); // 111 222

    struct_t *pass_pointer (struct_t *param) { param->first[0] = 333; return param; }
    struct_t *structp = pass_pointer ( &struct1 ); // 1 pointer on stack
    structp->second[0] = 444;
    printf ("%3d %3d\n",struct1.first[0],struct1.second[0]); // 333 444
    printf ("%3d %3d\n",structp->first[0],structp->second[0]); // 333 444
```

This example first passes and returns a large structure by value. Passing values into and out of function calls is safe because all modifications are to the copied value. However, for large objects, the attendant performance degradation is typically unnecessary. Here, the entire structure is copied into and out of the function call, but the function modifies only a very small piece of it. In the example, we see that struct1 containing two integer arrays of 100 elements each is passed into the function pass_value where it modifies only one element of an array, struct2; copying this struct into the function causes much overhead. The changes are made on the local copy, struct2 ; this doesn't affect the value of struct1.

The example next passes and returns a large structure by pointer. Passing non-const pointers into and out of function calls preserves performance but shares objects between calls, so it may inadvertently clobber a shared object. Here, we see when we change the value of struct_p, value of struct1 changes to the struct_p value and vice-versa as both pointers point to same memory location.

Example: Function Declarations (continued)

```

const struct_t *pass_const_pointer (const struct_t *param) {
    param->first[0] = 555; // illegal
    return param;
}
const struct_t *cstructp = pass_const_pointer ( &struct1 );
cstructp->second[0] = 666; // illegal
printf ("%3d %3d\n", struct1.first[0], struct1.second[0]); // 333 444
printf ("%3d %3d\n", cstructp->first[0], cstructp->second[0]); // 333 444
return 0;
}

```



The example lastly passes and returns const structure pointers. It is good programming practice to declare passed pointers **const** wherever possible, even if they point to non-const objects, to prevent inadvertent modification of the pointed-to object. Here, the compiler detects the attempt to modify the structure through a pointer-to-const type. In the example, we see that the function parameter, which is a pointer to struct_t, is declared constant. Hence, it's illegal to modify its contents. We see that the contents of struct1 and cstructp are the same.



Reference: Precedence and Associativity

In order of highest to lowest precedence.

Category	Operators	Assoc	Category	Operators	Assoc
primary	identifier constant (expr)		AND (bitwise)	&	Left
postfix	[] . > ++ func ()	Left	XOR (bitwise)	^	Left
unary	++ & * + ~ !	Right	OR (bitwise)	 	Left
cast	(type) expr	Right	AND (logical)	&&	Left
multiplicative	* / %	Left	OR (logical)	 	Left
additive	+	Left	Conditional	? :	Right
shift	<< >>	Left	Assignment	= *= /= %= += -= <=>= &= ^= =	Right
relational	< <= >= >	Left	Comma	,	Left
equality	== !=	Left			

30 © Cadence Design Systems, Inc. All rights reserved.



As you are already familiar with these operators, this table is primarily for your reference purposes.

The C programming language does not have a unary bitwise AND operator, so what you see in the unary category are the address and indirection operators.

The comma operator evaluates both expressions and the comma expression value is the value of the right expression.

Selection Statements



```
if ( expr ) stmt1 [ else stmt2 ]
```

Jumps to: if $expr \neq 0$ *stmt1* else *stmt2*

Note: *expr* must be scalar



```
switch ( expr ) statement
```

- Jumps to: first *stmt* where *const-expr==expr*, if it exists.
- Otherwise: *stmt* with "default" label, if it exists, otherwise to end of the switch.

- *expr* must be integral.
- *stmt* is typically a compound containing one or more statements labeled:
 - *case const-expr: statement*
and optionally one statement labeled:
 - *default: stmt*

```
typedef enum {WRITE,READ} transtype_t;
transtype_t transtype;
// ... code setting transtype ...
switch ( transtype ) {
  case WRITE:
    do_write();
    break;
  case READ:
    do_read();
    break;
  default:
    do_error();
    break;
}
```

```
int main() {
  float f;
  printf("Enter float value: ");
  int num = scanf("%f", &f);
  if ( num == 1 )
    printf("Value is %f\n", f);
  else
    printf("Not a float!\n");
  return 0;
}
```

31 © Cadence Design Systems, Inc. All rights reserved.



For the **if** statement, the first substatement executes if the controlling expression is not equal zero; if the **else** clause exists, the second substatement executes if the controlling expression is equivalent to zero. The controlling expression must be a scalar. In the example, we see that when num equals 1, it executes the first substatement, printing the value of f else executes the else part and prints 'not a float'.

For the **switch** statement, execution jumps to the statement following the matched **case** label or, if no match exists, the **default** label, if it exists. The controlling expression must be an integer scalar. Case expressions must be constant integer scalar and no more than one can match the controlling expression. In the example, when transtype is of the type write, it executes write case; if read type, then executes read case else; if neither read nor write, then to default where it throws out an error.

Iteration Statements



```
while ( expr ) stmt  
do stmt while ( expr ) ;  
  
while expr != 0 do stmt.  
do stmt while expr != 0
```



```
for ( [expr1]; [expr2]; [expr3] )  
stmt
```

Note: *expr* must be scalar.

```
int main()  
{  
    char str[2];  
    unsigned i=0;  
  
    do {  
        printf("%10d :Iterate? (y/n) [n] : ",i++);  
        gets(str); // dangerous!  
    } while ( *str == 'y' );  
  
    return 0;  
}
```

```
int main()  
{  
    printf("Iterations? : "); int i;  
    int num = scanf("%i", &i);  
  
    if ( num == 1 )  
        for (int j=0; j < i; ++j)  
            printf("Iteration %10d\n",j);  
    else  
        printf("int not found!");  
  
    return 0;  
}
```

32 © Cadence Design Systems, Inc. All rights reserved.



For the **while** statement, the controlling expression is evaluated *before* each loop body execution.

For the **do-while** statement, the controlling expression is evaluated *after* each loop body execution.

The **for** statement is equivalent to evaluating expression 1 followed by a while statement in which expression 2 is the controlling expression and expression three is appended to the statement. You can omit any or all expressions. If you omit expression 2, the compiler replaces it with a non-zero constant. The 1999 update to the standard (ISO/IEC Std. 9899) C Programming language permits a declaration in the expression 1 position. Your compiler may need an invocation option to accept the 1999 constructs. In the example, the for loop prints out the iteration number from 0 to i-1.



Jump Statements

`break;`

- Jump to end of loop or switch statement.
- May appear only in loop body or switch.

`continue;`

- Jump to loop body end.
- May appear only in loop body.

`goto label;`

- Jump to labeled statement.
- Label must be in enclosing function.

`return [expr];`

- Return to function caller.
- Must omit `expr` if function type void.
- Must have `expr` if function type not void.

```
while (index < 10)
{ // do some processing
  if ( valueGot == valueWant )
  {
    printf("Got value\n");
    break; //control goes out of while loop
  }

  if ( valueGot == invalid )
  {
    printf("Invalid value\n");
    continue;
  } // do more processing
}
```

33 © Cadence Design Systems, Inc. All rights reserved.



A **break** statement breaks execution from the enclosing switch or iteration. It must appear *only* in a switch body or loop body.

A **continue** statement jumps execution to the end of the loop body. It must appear *only* in a loop body.

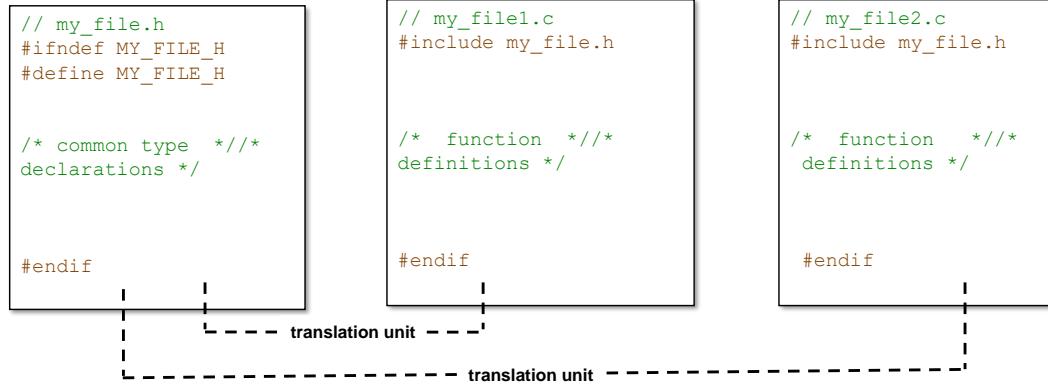
A **goto** statement jumps execution to the labeled statement. The labeled statement must reside in the function of the goto statement.

A **return** statement returns control to the calling function. You must include the expression if returning from a non-void function.

Translation Units

A *preprocessing translation unit* is the source file, and all other source and header files included with the `#include` preprocessing directive

- This suggests the separation of *declarations* and *descriptions* – an usual and good programming practice.



34 © Cadence Design Systems, Inc. All rights reserved.



You keep the C program source text in one or more source files. Usual and good programming practice places common declarations in header files, with preprocessing directives to ensure that a compilation session reads the header file at most once, even if multiple source files compiled during that session each include the header file.

The C standard refers to a source file with its included headers and potentially other included source files as a *preprocessing translation unit*. Some compilers permit you to save the preprocessed units as *translation units*.

Translation of a translation unit produces object code. People generating object code for reuse typically store it in object libraries. For example, all the object code implementing standard C functions is in object libraries. People typically do *not* place the entry point to the application, which on a hosted system is the main() function in a library.

Multiple translation units intercommunicate by utilizing names having *external* linkage. A link editor collects object code, resolves these references, and produces an executable binary image.

Compiling and Linking with gcc

- The compiler compiles source files (*.h, *.c) into object files (*.o).
- The link editor links object files into a library or an executable.
- You can do this in two steps or one:

Separate steps

```
gcc -c file1.c -o file1.o  
gcc -c file2.c -o file2.o  
gcc -o run.x file1.o file2.o  
run.x
```

Together

```
gcc -o run.x file1.c file2.c  
run.x
```

Include `-std=c99` to accept the ISO/IEC 9899:1999 features.



The GNU C compiler expects C source files to use the “.c” extension. The compiler, by default, preprocesses, compiles, assembles, and links. The compiler compiles source files (*.h, *.c) into object files (*.o). The link editor links object files into a library or an executable

According to the compiler manual:

- Use the `-c` option to only compile or assemble the source code, with no subsequent linking.
- Use the `-o` option to place the output in the specified file.

Many more options exist that these materials do not re-document, and for this, you are encouraged to refer to the compiler manual.

Makefiles

Use **make** to maintain, update, and regenerate related programs and files.

A *makefile* consists of comments, macro definitions, and *rules*, which have:

- One or more targets.
- An optional dependency list.
- An optional command list (Bourne shell) for “making” the target.

```
target ... {::|::} [dependency ...] [;
command]
[command]
[ ]-----> CC=gcc  # macros
CFLAGS=-g -Wall -I/usr/include/libxml2
LIBS=-lxml2

prog: main.o aux.o  # link
      $(CC) $(LIBS) main.o aux.o -o
prog

.c.o:$*.c  # compile all
$(CC) $*.c -c $(CFLAGS)
```

make [target] [-f makefile_name]

36 © Cadence Design Systems, Inc. All rights reserved.



Use the **make** utility to maintain, update, and regenerate related programs and files.

Makefiles may contain comments, variable definitions, and rules. A rule is a set of targets, an optional set of dependencies, and an optional set of Bourne shell commands to “make” each target. Where a target is older than a dependency or does not exist, the **make** utility executes the commands to update the target. Targets can be explicit file names or classes of files as shown here.

You specify the initial target on the command line when you invoke **make**. If you do not specify a target, **make** uses of the first target. The utility also has several rules for resolving a target that has no Makefile entry, or that has no specified rule.

As dependencies can also be targets and have dependencies of their own, **make** recursively checks targets to completely update the target hierarchy. To further understand **make**, you are encouraged to refer to the manual pages installed on your platform or any good bookstore offering technical literature.

Quiz



How does the `main()` function differ from all other functions?

- The `main()` function is the entry point to your C program.



An enumerated type is a user-specified set of values of the [____] type.

- A user-specified set of values of the `int` type.



For what reason do programmers frequently choose to exchange pointers between functions instead of exchanging objects?

- Exchange of an object requires copying the object, which is often large and thus potentially wasteful of platform resources. Exchanging a pointer to the object allows the multiple functions to use the object without copying it.

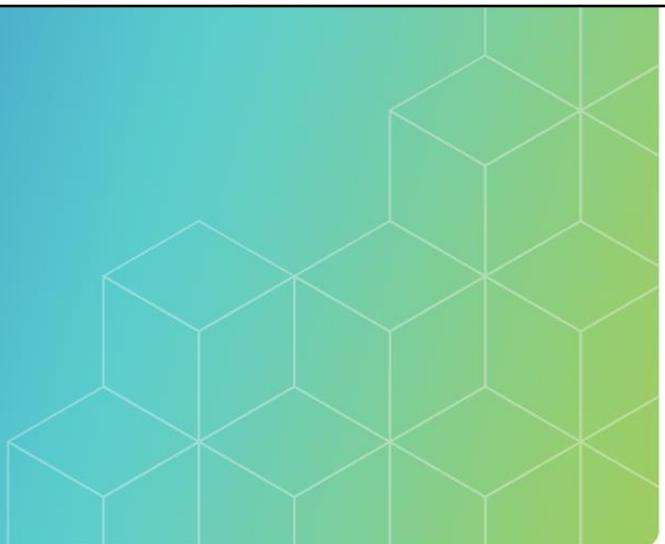
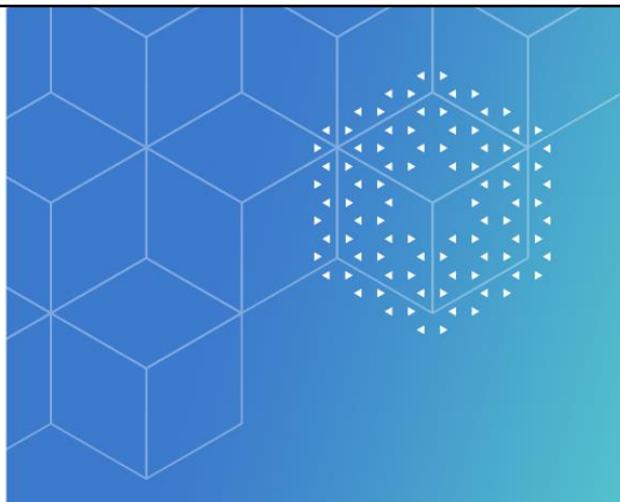


Explain how a `struct` and a `union` differ while appearing syntactically similar.

- A `struct` is a collection of objects of typically different types.
A `union` is a collection of types for representing one object

Please pause here for a minute and consider answering these questions

- How does the `main()` function differ from all other functions?
- An enumerated type is a user-specified set of values of the [____] type.
- For what reason do programmers frequently choose to exchange pointers between functions instead of exchanging objects?
- Explain how a `struct` and a `union` differ while appearing syntactically similar.



Module 3

Object-Oriented Programming and C++

cadence®

This training module introduces Object-Oriented Programming (OOP) and then introduces C++ in the OOP context.

Module Objectives

In this module, you

- Organize your programming solution in terms of objects and their interactions

This training module discusses:

- Object-Oriented Programming
- An Introduction to C++
- Classes and Objects



Your objective is to organize the solution to your programming problem in terms of objects and their interactions, instead of in terms of a sequence of actions.

To help you to achieve your objective, this training module discusses:

- Object-Oriented Programming;
- An Introduction to C++; and
- Classes and Objects.

What Is Object-Oriented Programming?



Object-Oriented Programming is a programming paradigm that focuses on objects (as apposed to process).

Data-oriented rather than procedure-oriented

- Focuses on objects instead of actions
- Defines classes to characterize the objects
- Classes declare and own:
 - properties (AKA “attributes” or “data”)
 - Class member data mostly is private – hidden from “outside” code
 - behaviors (AKA “methods” or “functions”)
 - Class member functions often are public – callable by outside code
 - Termed the “public interface methods”

class Student

PROPERTIES

- grade
- name
- number

BEHAVIORS

- eat
- play
- sleep
- study



A+
Jack
1234



A+
John
2341



A+
Jane
3412



A+
Jill
4123



40 © Cadence Design Systems, Inc. All rights reserved.

Object-Oriented Programming is a programming paradigm that focuses on *objects*, as apposed to *processes*.

It is data-oriented rather than procedure-oriented.

It focuses on objects instead of actions.

It partitions a programming problem into object types, that we refer to as “classes”.

- An object of the class has properties, the collective value of which define the object’s state.
 - The class definition can choose to individually hide or make visible its properties. Classes typically hide most properties.
- An object of the class has behaviors, that collectively define how the object is created, how it interacts with other objects, and how it reacts to applications that use it.
 - The class definition can choose to individually hide or make visible its behaviors. Classes typically expose at least some behaviors.
 - Objects interact with other objects and applications only through the visible properties and behaviors. The visible properties and behaviors are termed the “public interface”.

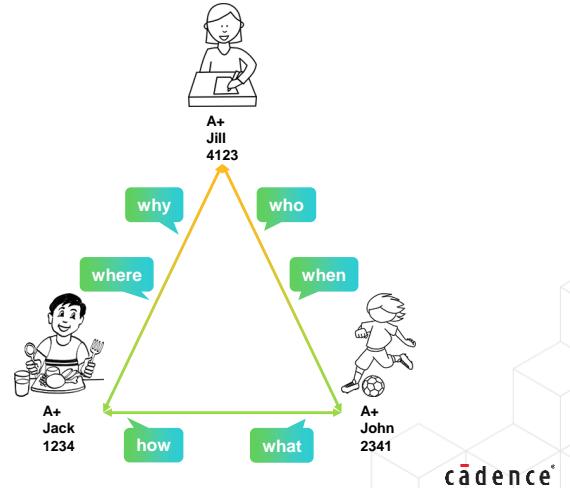
For example, let us imagine an object type that we name “student”. All objects of that type will have the properties of that type, and exhibit the behaviors of that type. Different objects can have different property values, and those values can effect their behaviors, but all objects of that type will have the properties of that type, and exhibit the behaviors of that type.

What Is Data Encapsulation in Object-Oriented Programming?



Data Encapsulation means that each object “hides” its data from external access.

- An object typically prohibits direct external access to its data.
- Objects interact by “sending messages.”
- A message can request the receiving object to change a data value or reply with a data value.
- Programmers refer to this as “calling a function.”



41 © Cadence Design Systems, Inc. All rights reserved.

Data Encapsulation means that each object “hides” its data from external access.

An object typically prohibits direct external access to its data.

In the object oriented paradigm, objects interact by “sending messages”.

A message can request the receiving object to change a data value or reply with a data value.

In C++ we simply term this “calling a function”.

Here, we illustrate students interacting by sending messages.

What Is C++?



C++ is essentially a superset of C to support object-oriented programming and other features.

- classes and objects
- new input/output interface
- overloaded functions
- overloaded operators
- references
- generic types (templates)
- exceptions
- much more!



42 © Cadence Design Systems, Inc. All rights reserved.

C++ is essentially a superset of C that supports object-oriented programming and has additional features:

- You can use a built-in library of stream classes for character-oriented terminal and file I/O.
- You can declare and define multiple functions having the same name but with different parameters. The functions would presumably perform similar tasks. The compiler automatically resolves each function call in accordance with the argument types.
- You can similarly define functions to implement operators applied to your own classes. Some few operators you cannot overload.
- You can pass references to and from functions. Passing references is equivalent to passing pointers, but the compiler automatically does the address and indirection operations to prevent you from making the mistakes that are all too common with pointers.
- You can write generic code that you can reuse with multiple different types.
 - You can define a function template with one or more placeholders for types and values that you define when you call the function. Each call with a different set of types or values generates a different function definition.
 - You can define a class template with one or more placeholders for types and values that you define when you reference the class type. Each reference with a different set of types or values generates a different class definition.
- You can use a built-in infrastructure for handling exceptions, which includes a standard exception class.

This is not a comprehensive list. C++ offers much more.



Quick Reference Guide: Terminology – OOP Versus C++

Textbook OOP	ISO/IEC C++
“attribute”	“variable”
“property”	
“behavior”	“function”
“method”	
“object”	“instance”
“sending a message”	“calling a function”



You may see these terms used interchangeably in books about Object-Oriented Programming.

When discussing a specific OOP language, to use the terminology of the language standard will promote comprehension.

- What classic OOP terms an “attribute” or a “property”, the C++ standard terms a “variable”.
- What classic OOP terms a “behavior” or a “method”, the C++ standard terms a “function”.
- What classic OOP terms an “object”, the C++ standard terms an “instance”.
- What classic OOP terms “sending a message”, the C++ standard terms “calling a function”.

Comparing Data Models: C Versus C++



C structures data into `struct` constructs that contain only variables.



C++ classifies real-world entities (person, place or thing) into class constructs that contain variables and functions to manipulate the variables.

```
struct studentData { // only data
    unsigned number;
    char grade;
    char *name;
    char *surname;
};
```

```
class StudentRecord {
public: // interface
    void set_name(const char *nm);
    char *get_name() const;
private: // data
    unsigned number;
    char grade;
    char *name;
    char *surname;
};
```

In the C programming language, you model a collection of related data using a structure (**struct**). The fields of a C structure can be only variables. Function definitions cannot reside within a C structure.

In the C++ programming language, you can still model a collection of related data with a structure. However, you can also include in the C++ structure or class, definitions of functions to manipulate the data and mediate its interchange with other objects and with application code.

Comparing Classes and Objects



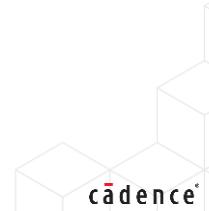
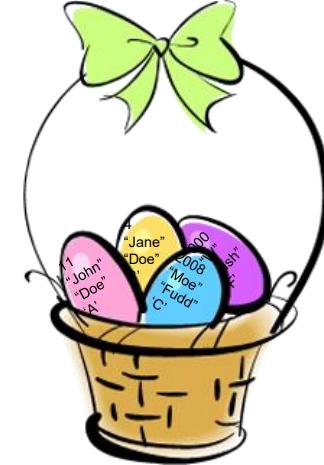
A **class** defines an entity (person, place or thing) type, having:

- Member data (attributes, properties, variables)
- Member behaviors (methods, functions)



An object is one instance of the class type.

```
class StudentRecord {  
  
public:// Public methods to access  
    private ATTRIBUTES  
    void set_name(const char *name);  
    char *get_name() const;  
  
private: // Private ATTRIBUTES  
    unsigned number;  
    char grade, name[20], surname[20];  
};
```



45 © Cadence Design Systems, Inc. All rights reserved.

A class defines an entity type, having:

- Member data, in C++ termed *variables*, and elsewhere also termed *attributes* or *properties*.
- Member behaviors, in C++ termed *functions*, and elsewhere also termed *methods*.

To declare a class, you can use either keyword “**class**” or “structure” (**struct**).

- All members of a class are by default **private**, and you can declare them **protected** or **public**.
- All members of a structure are by default **public**, and you can declare them **protected** or **private**.

An object is one instance of the class type.

An object is an area of memory allocated for your program and associated with a type. Memory allocated with the C memory allocation functions has no type, and thus does not represent an object. Memory allocated by the C++ operator “**new**” is associated with a type, and so is an object.

The example class “Student Record” declares the functions “set name” (`set_name`) and “get name” (`get_name`) as its public interface, and declares several variables as private data.

Here, we illustrate a collection of objects of the Student Record class, each with the variables *number*, *grade*, and *name*.

Comparing the C++ Constructs Struct and Class



A C++ **struct** is a C++ **class**. The *only* difference is that access to **struct** members is by default **public** and you can make it **protected** or **private**.



A C++ **class** is a C++ **struct**. The *only* difference is that access to **class** members is by default **private** and you can make it **protected** or **public**.

Struct members are by default *public*

```
struct Struct_s {  
//public:  
    int x;};
```

Class members are by default *private*

```
class Class_c {  
private:  
    int x;};
```

You can explicitly declare them *private*

```
struct Struct_s {  
private:  
    int x;};
```

You can explicitly declare them *public*

```
class Class_c {  
public :  
    int x;};
```

46 © Cadence Design Systems, Inc. All rights reserved.



Access to C++ structure (**struct**) members is by default **public** and you can declare it **protected** or **private**.

Access to C++ **class** members is by default **private** and you can declare it **protected** or **public**.

This difference exists purely for your convenience as a programmer.

C++ Vs. C



C++ is essentially a superset of C to support object-oriented programming and other features.

C++ is a “better” C

- Stronger type-checking
- Exceptions
- Much more...

Data abstraction/encapsulation/hiding

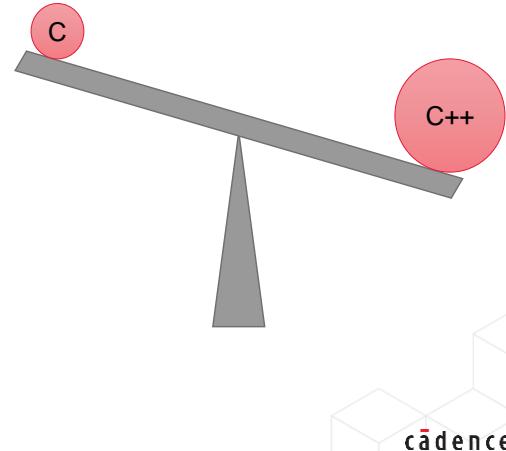
- User can hide implementation details to prevent “outside” code from modifying internal data.

Object-oriented programming

- User can extend previously defined classes and override its functions for the extended type.

Generic programming

- User can write class templates to be specialized upon instantiation.



47 © Cadence Design Systems, Inc. All rights reserved.



C++ is essentially a superset of C to support object-oriented programming and other features.

Here are some reasons why you might want to program in C++ instead of in C.

- Firstly, without getting into any C++ OOP features, C++ is simply a “better” C. Among its enhancements are that it is more strongly typed and supports exceptions.
- Secondly, C++ supports the OOP concepts that are indispensable to the organization of large projects.
 - Data abstraction – you can create classes that hide their implementation, thus isolating the user from changes of the implementation;
 - Object-oriented programming – you can create a hierarchy of classes that encapsulate common features in higher-level base classes; and
 - Generic programming – you can create function and class templates that work with a variety of suitable types.

Module Summary

In this module, you

- Organized your programming solution in terms of objects and their interactions

This training module discussed:

- Object-Oriented Programming
- An Introduction to C++
- Classes and Objects



Your objective is to organize the solution to your programming problem in terms of objects and their interactions, instead of in terms of a sequence of actions.

To help you to achieve your objective, this training module discussed:

- Object-Oriented Programming;
- An Introduction to C++; and
- Classes and Objects.

Quiz: Object-Oriented Programming and C++



If a Toyota Tacoma is an example of a class, then my old beat-up pickup truck with serial number XYZZY is an example of an [_____].



What member can a C++ **struct** have that a C **struct** cannot have?



How many objects can exist of any given class?



If you do not use the C++ object-oriented features, what C++ function-oriented features make it still worth your while to use C++ ?

Please pause here to consider these questions.



Lab

Lab 3-1 Organizing an OOP Project

Objective:

- To determine a set of classes to support transaction recording

For this lab, you:

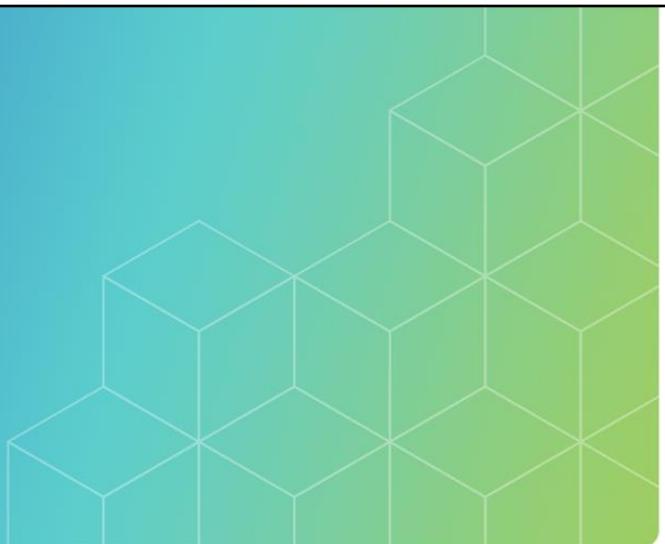
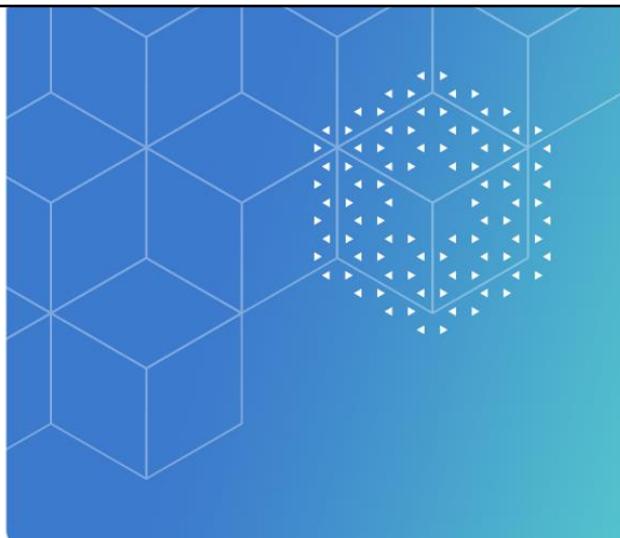
- Generate a document describing your class-based organization of these transaction recording roles – describe your proposed classes and their member data and functions
 - Maintain a transaction database
 - Maintain a transaction stream
 - Generate transactions
 - Represent transactions
 - Represent the data item being transacted

50 © Cadence Design Systems, Inc. All rights reserved.



Your objective for this lab is to determine a set of classes to support transaction recording.

For this lab, you generate a document describing how you would organize transaction recording components. Describe what classes you would propose, and what their member data and functions would be.



Module 4

C++ Basics

cadence®

This training module just briefly introduces C++. Following study of this module, you should be able to write a simple “Hello World” program.

Module Objectives

In this module, you

- Write a simple C++ program having classes and objects

This training module discusses:

- Basic C++ Input and Output
- Declarations and Scope
- Member Access Specifiers, **private** and **public**
- The Current Instance Pointer **this**
- Operators **new** and **delete**



Your objective is to write a simple C++ program having classes and objects.

To help you to achieve your objective, this training module discusses:

- Basic C++ Input and Output;
- Declarations and Scope;
- Member Access Specifiers **private** and **public**;
- The Current Instance Pointer **this**; and
- Operators **new** and **delete**.

The C++ main() Function



```
int main ( ) ;  
int main ( int, char** ) ;
```

- The main() function is the program entry point.
- The standard requires an implementation to support the two signatures displayed here.
- The standard permits an implementation to define additional signatures having additional parameters.
- The standard permits a stand-alone program to omit the main() function.

```
#include <iostream>  
  
using namespace std;  
  
int main() {  
    cout << "Hello World!" << endl;  
    return 0;  
}
```

53 © Cadence Design Systems, Inc. All rights reserved.



The “main” function is the program entry point.

The standard requires an implementation to support the two signatures displayed here.

The standard permits an implementation to define additional signatures having additional parameters.

The standard permits a stand-alone program to omit the main function.

The example program:

- Includes the input/output stream (**<iostream>**) header, which declares objects associated with the standard character-based input and output streams.
- Directs that names in the standard namespace are visible for unqualified lookup in this scope.
- Defines the function “main” with two statements:
 - The 1st statement, inserts into the built-in C output (**cout**) stream object, the string “Hello World” and a new-line character, and flushes the output stream.
 - The 2nd statement, returns the integer value 0 to the calling program. For the function “main”, this is an error code, for which the value 0 indicates the absence of error. For the function “main”, if you omit to explicitly return an error code, then the program implicitly returns the value 0.

The **<iostream>** header declares in the **std** namespace the object **cout** of type **ostream**.

Example: C++ main() Function

```
#include <fstream>      // Include definitions from fstream library
using namespace std;   // Make all in std namespace directly visible

int main(int argc, char **argv)      // Entry point to program
{                                     // (optional invocation arguments)
    fstream stream;                 // Instance of file stream
    stream.open("file");           // Open "file" for update
    if (!stream.is_open())
        return 1;                  // Return if unsuccessful
    stream.read ((char*)data,4*256); // Read file data
    compute_result(data,256);       // User-defined function
    stream.write((char*)data,4*256); // Write modified data
    stream.close();                // Close file
    return 0;                      // Program status
}
```

54 © Cadence Design Systems, Inc. All rights reserved.



The “hash include” (`#include`) preprocessing directive replaces the directive with the contents of a bracketed header file or double-quoted source file. The C++ standard does not define the search algorithm other than to state that if the search for a double-quoted source file fails then the implementation must re-attempt the search as if it were a bracketed header. This example includes the file stream (`<fstream>`) header, which contains definitions necessary for file input and output.

The “using-directive” directs the compiler to accept names from the designated namespace for unqualified lookup in the scope of the directive after the directive appearance. This example makes names within the standard (`std`) namespace directly visible.

The “*main*” function is the entry point to every C++ program.

The `<fstream>` header declares in the `std` namespace the type `fstream`.

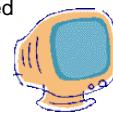
Basic C++ Input and Output with `<iostream>` Objects



```
cin // built-in input stream instance opened to standard input
cout // built-in output stream instance opened to standard output
clog // built-in output stream instance opened to standard output
cerr // built-in output stream instance opened to standard error output
```

Include the `<iostream>` header to utilize C++ input/output stream objects:

- **cin** – for standard input
 - The keyboard unless otherwise redirected
 - Comparable to C `scanf()` function
- **cout** – for standard output
 - The terminal window unless otherwise redirected
 - Comparable to C `printf()` function
- **clog** – for standard output
- **cerr** – for standard error output



```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

To instead continue to utilize the C interface include the `<stdio.h>` or `<cstdio>` headers.

55 © Cadence Design Systems, Inc. All rights reserved.



After including the input/output stream (`<iostream>`) header, you can utilize C++ input/output stream objects:

- The “C in” (**cin**) object is a pre-initialized input stream (**istream**) object associated with the standard input.
- The “C out” (**cout**) object is a pre-initialized output stream (**ostream**) object associated with the standard output.
- The “C log” (**clog**) object is a pre-initialized output stream object also associated with the standard output.
- The “C error” (**cerr**) object is a pre-initialized output stream object associated with the standard error output.
- The “C error” object differs from the “C out” and “C log” objects, in that upon initialization, it by default flushes its buffer after each output operation.

You can still utilize the C interface by including either the C or the C++ versions of the C standard I/O library headers:

- The C-style headers are placed in the global namespace.
- The C++-style headers are placed in the standard (**std**) namespace, and compilers compliant with the standard require you to make an explicit *using-directive* or *using-declaration* to make visible the names declared in the header.

The example program:

- Includes the input/output stream header, which declares objects associated with the standard C input/output streams.
- Directs that names in the standard namespace are visible for unqualified lookup in this scope.
- In the function “*main*”, inserts into the “C out” output stream, the string “Hello World” and a new-line character, and flushes the output buffer.

The `<iostream>` header declares in the **std** namespace the object **cout** of type **ostream**.

Example: Basic C++ Input and Output with <iostream> Objects

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

int main()
{
    int age;
    cout << "Enter your age: "
        << flush; // Stream manipulator flushes ostream buffer
    cin >> age; // Operator overloaded to extract from istream
    cout << "You are " << age
        << " years old" << endl; // Inserts newline and flushes
    return 0;
}
```

56 © Cadence Design Systems, Inc. All rights reserved.



For this example:

- The output stream (**ostream**) insertion operator inserts only the one next value. Here, it inserts a character string, and then a manipulator to flush the output buffer. For standard output associated with a terminal, the operating system may flush the buffer anyway, but to ensure portability, you should explicitly flush the buffer for interactive applications such as this.
 - The input stream (**istream**) extraction operator extracts only the one next value. Here it extracts an int value.
-

The insertion and extraction operators map to overloaded functions that take and return a reference to the stream object. This permits the operators to be chained, as shown here.

The stream insertion operator overload that takes a *basic_ostream&* parameter is implemented to call the function passed to it, for example the endl() function, which inserts a newline character and flushes the stream. Such operations performed mid-stream are termed “I/O manipulators”. Several such manipulators are defined.

What Is Scope?



Scope is a region of program text in which a name may be referenced without qualification.

Names are unique within a scope:

- The same name in a different scope is a different entity.
- A more local declaration of a name “hides” a less local declaration that uses the same name.

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

int i=0;           // namespace (global) scope

int main()
{
    int i = 1;      // block scope
    cout << "i = " << i << endl;
    return 0;
}
```

Scope is a region of program text in which a name may be referenced without qualification.

A declaration introduces a name into a scope. The name is visible from the point of declaration to the end of the declaration scope. Namespace scopes do not end.

Names are usually unique within a scope, though exceptions do exist that we do not discuss here.

- The same name in a different scope is a different instance.
- A more local declaration of a name hides a less local declaration using the same name.

Example: C++ Scopes

Scope	Potential Extent
Block	From point of declaration to end of block. Local or function-local variables.
Namespace	From point of declaration onward (namespaces are open). For global and anonymous namespaces from point of declaration to translation unit end.
Class	From point of declaration to end of the class definition and also within <i>member function</i> declarations and definitions.



A name declared in a block is local to the block. Its potential scope is from the point of declaration to the block end. This applies also to a name declared in a function definition.

A name declared in a namespace is a member name. Its potential scope is from the point of declaration onwards. For anonymous namespaces, that can have only one declarative region, that potential scope ends at the translation unit end. This applies also to the “global” namespace.

The potential scope of a class member name is from point of declaration to the class definition end, and also within function default arguments and bodies, and in non-static data initializers, and in exception specifications.

Example: Scope Resolution

```

int var1 = 1, var2 = 3; // global namespace
int fnc1() {return 1;} int fnc2() {return 3;};

namespace NS {
    int var1 = 5, var3 = 7; // named namespace
    int fnc1(); int fnc3() {return 7;};

    int fnc1() {
        int var1 = 9; // block (function local)
        cout << var1 << endl; // local -- prints out "9"
        cout << NS::var1 << endl; // named -- prints out "5"
        cout << ::var1 << endl; // global -- prints out "1"
        cout << ::fnc1() << endl;
        cout << var2 << endl; // global -- but not ambiguous
        cout << fnc2() << endl;
        cout << var3 << endl; // named -- but not ambiguous
        cout << fnc3() << endl;
        return 5;
    }
}

int main()
{
    NS::fnc1();
    return 0;
}

```

59 © Cadence Design Systems, Inc. All rights reserved.



This example, in the global namespace, declares the variables “variable 1” and “variable 2”, and the functions “function 1” and “function 2”.

This example, in the named namespace “NS”, declares the variables “variable 1” and “variable 3”, and the functions “function 1” and “function 3”. The named namespace declaration of “variable 1” hides the global namespace declaration of “variable 1”. Following the named namespace declaration of “variable 1”, any unqualified reference to “variable 1” refers to the “variable 1” of the named namespace. Likewise for “function 1”.

The function “function 1” also declares its own variable “variable 1”. This local declaration of “variable 1” hides the named namespace declaration of “variable 1”. Following the local declaration of “variable 1”, any unqualified reference to “variable 1” refers to the “variable 1” of the local block.

To qualify a reference, you use the scope resolution operator. The function directly refers to its own declaration of “variable 1”, and then by name qualification, refers to the named namespace declaration of “variable 1”, and then refers to the global namespace declarations of “variable 1” and “function 1”.

Best Practices for Variable Declarations



Your variable declarations can promote or impede understanding.

For readability:

- Declare your variables at the point of first use
- Give the variable a name reminiscent of its use
- Use each variable for only one purpose

To further promote readability and avoid name clashes, you might elect to:

- Prefix class member data with “m_”
 - Example: `String m_name;`
- Suffix function arguments with “_a”
 - Example: `String name_a;`

```
int age;  
cout << "Enter your age: " << flush;  
cin >> age;  
cout << "You are " << age  
<< " years old" << endl;
```

```
float pi=3.1415;  
cout << "Pi value is approximately: "  
<< pi << endl;
```

60 © Cadence Design Systems, Inc. All rights reserved.



Your variable declarations can either promote understanding or impede understanding.

To promote readability and reduce the likelihood of error:

- Declare your variables at the point of first use;
- Give each variable a name that reminds you of its use; and
- Use each variable for only one purpose.

Declaring a variable at the point of first use promotes readability. You know that the variable has not been previously set. You also know that the variable is not used in an outer scope.

To further promote readability and avoid name clashes, you might elect to consistently prefix class member data, and consistently suffix function arguments.

Member Access Specifiers: **private, protected, public**



```
private: // Only member functions of this class can access  
protected: // Only member and friend functions of this class  
// and member functions of derived classes can access  
public: // Any code can access
```

private:

- All **class** members are by default *private*.
- Only member functions can access private members.
- You explicitly specify what class members are *public*.



```
class Class_c {
```

```
    int x; // private
```

```
public:
```

```
    int y;
```

```
};
```

```
struct Struct_s {
```

```
    int x; // public
```

```
private:
```

```
    int y;
```

```
};
```

The *public interface* is those public functions through which application code manipulates private data.



61 © Cadence Design Systems, Inc. All rights reserved.

cadence

The keyword “**private**”, denotes a class region of declarations that only member functions of the class can access.

The keyword “**protected**”, denotes a class region of declarations that only member and friend functions of the class and member functions of its derived classes can access.

The keyword “**public**”, denotes a class region of declarations that any code can access.

All members of a type declared with the keyword “**class**” are by default **private**. Only members and friends can access private members. You must explicitly specify what members of your class are made **public**.

All members of a type declared with the keyword “structure” (**struct**) are by default **public**. All code can access public members. You must explicitly specify what members of your structure are made **private**.

The public interface is those public class functions through which application code or other objects access private data.

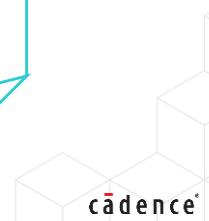
Example: Member Access Specifiers

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

class StudentRecord
{
public:
    // Public functions to access private data
    void set_name(const char *name);
    const char *get_name() const;
private:
    // Private data
    unsigned number;
    char grade, name[20], surname[20];
};

int main()
{
    StudentRecord s;
    //cout << s.name << endl;           // Illegal access to private data
    cout << s.get_name() << endl;        // Legal access using public interface
    return 0;
}
```

62 © Cadence Design Systems, Inc. All rights reserved.



The functions “set name” (set_name) and “get name” (get_name) constitute the public interface to Student Record objects. The compiler will not allow application code and other objects to directly access the private data.

Pointer to Current Instance: **this**



```
this // Implicit member function parameter points to current object
```

All class/struct non-static member functions have an implicit parameter named **this** that points to the current object. How is this useful?

- To access member data hidden by a more local declaration.
- For certain operator overloads that return a reference to the current instance.

```
class Class_c {  
public:  
    bool is_same(int i) const  
    { return this->i == i; }  
private:  
    int i;  
};  
Class_c &operator= (const Class_c &a)  
{  
... return *this; // for chaining  
}
```

63 © Cadence Design Systems, Inc. All rights reserved.

cadence®

A built-in pointer named “**this**” points to the current object.

The C++ compiler automatically and invisibly adds the pointer to the argument list when you call a non-static member function. The pointer has at least these two very useful purposes:

- You can use the **this** pointer to access member data hidden by a local declaration of the same name; and
- Certain operator overloads return a reference to the current object.
 - For example, the assignment operator returns the current object reference to allow assignment chaining.

Dynamic Storage Allocation/Deallocation Operators: new, delete



```
new      // Constructs an instance of the type and returns a pointer to it
delete // Destroys an instance created with new
```

new

- Constructs an instance of the type and returns a pointer to it
- Throws an exception upon failure (may potentially return NULL if exception handler appropriately set)

delete

- Destroys an instance created with **new**

```
// Construct instances
StudentRecord *s_ptr =
    new StudentRecord;
StudentRecord *s_ary =
    new StudentRecord[7];

// Utilize instances
s_ptr -> set_name("John");
(s_ary+5) -> set_name("Jack");

// Destroy instances
delete s_ptr;           // delete one instance
delete [] s_ary;        // delete instance array
```

64 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The *new-expression* attempts to construct an object or array of objects on the heap. If successful, it returns a pointer to the object or array of objects. If unsuccessful, it throws an exception (`std::bad_alloc`). You use the pointer with a pointer operator to access instance members.

The *delete-expression* destroys an object or array of objects created on the heap.

The example:

- Constructs a single Student Record object, and an array of 7 Student Record objects;
- Sets the name of the single Student Record object, and the name of the 6th element of the Student Record object array; and
- Deletes the single Student Record object, and the Student Record object array.

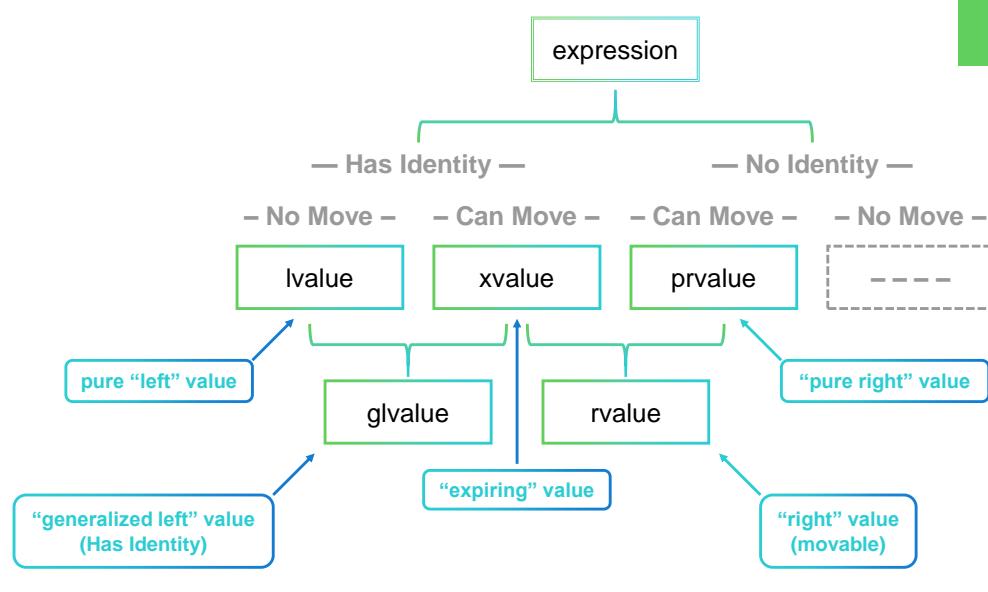
To allocate the heap storage, the *new-expression* calls a global allocation function, passing in the needed number of bytes and accepting the returned void pointer, which it then casts to the specified type and returns to the calling expression. You can call these global allocation functions yourself if all you need is a chunk of heap space representing no particular type:

- `void* operator new(std::size_t);`
- `void* operator new[](std::size_t);`

To de-allocate the heap storage, the *delete-expression* calls a global de-allocation function, passing in a void pointer. You can call these global de-allocation functions yourself to de-allocate your chunk of untyped heap space:

- `void operator delete(void*) noexcept;`
- `void operator delete[](void*) noexcept;`

Expression Categories



simplistically
`lvalue = rvalue;`

65 © Cadence Design Systems, Inc. All rights reserved.



You will see these expression category terms in almost any dissertation concerning C++.

The terms “left value” (*lvalue*) and “right value” (*rvalue*) initially appeared to mean the expressions respectively on the left side of an assignment and on the right side of an assignment.

To accommodate move semantics, C++ modifies the “right value” category and adds an “expiring value” category.

Now we categorize expressions by using two criteria:

- 1st – Does the expression have an identity? That is, does the expression identify an entity?
- 2nd – Is the expression movable? That is, is it a temporary expression whose resources can be appropriated for use elsewhere?

Every expression is in one of the three categories.

- A “left value” has an identity and cannot be moved from;
- An “expiring value” (*xvalue*) has an identity and can be moved from; and
- A “pure right value” (*prvalue*) has no identity and can be moved from.

C++ defines two non-disjoint collections of these categories:

- A “generalized left value” (*glvalue*) is an expression that has identity, either a “left value”, or an “expiring value”; and
- A “right value” is an expression that can be moved, either an “expiring value”, or a “pure right value”.

So an “expiring value” is both a “generalized left value” and a “right value”.

Module Summary

In this module, you

- Wrote a simple C++ program that uses classes and objects

This training module discussed:

- Basic C++ Input and Output
- Declarations and Scope
- Member Access Specifiers `private` and `public`
- The Current Instance Pointer `this`
- Operators `new` and `delete`



Your objective is to write and execute a simple C++ program that uses classes and objects.

To help you to achieve your objective, this training module discussed:

- Basic C++ Input and Output;
- Declarations and Scope;
- Member Access Specifiers `private` and `public`;
- The Current Instance Pointer `this`; and
- Operators `new` and `delete`.

Quiz: C++ Basics



The C++ counterpart to the C **scanf()** and **printf()** functions are overloaded operators of the [____] and [____] stream objects.



What does C++ “scope” mean?



Class members are **private** by default. What code can access private class members?



Suggest a use for the **this** pointer to the current object.

Please pause here to consider these questions.



Lab

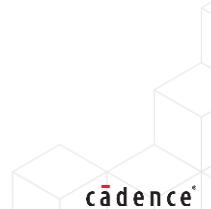
Lab 4-1 Writing a Simple C++ Program

Objective:

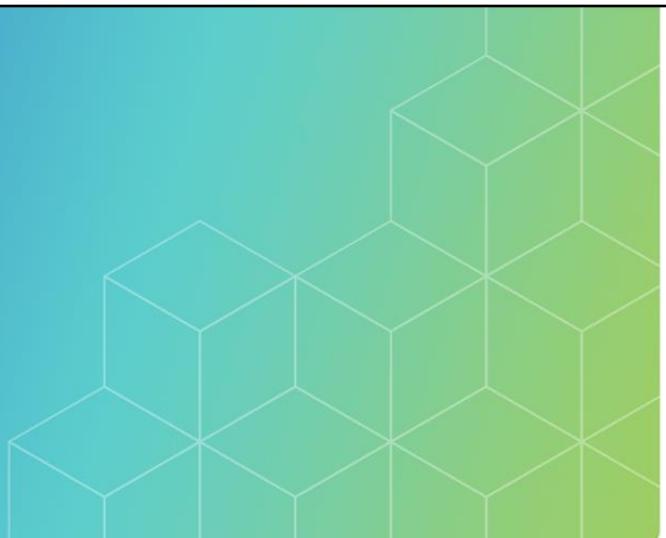
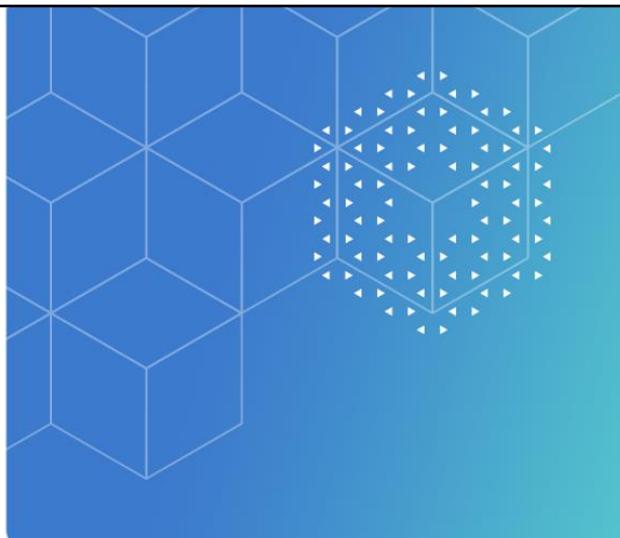
- To write and execute a simple C++ program with structured input/output

For this lab, you:

- Write and execute a simple C++ program having structured input/output



Your objective for this lab is to write and execute a simple C++ program with structured input and output.



Module 5

Constructors and Destructors

cadence®

This training module introduces constructors and destructors – special functions that construct and destroy objects.

Module Objectives

In this module, you

- Initialize class variables during object construction
 - By defining class constructors
- Return unneeded memory to the heap
 - By defining class destructors

This training module discusses:

- Constructors
- Destructors

70 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Initialize class variables during object construction, which you do by defining class constructors; and
- Return unneeded memory back to the heap, which you do by defining class destructors.

To help you to achieve your objective, this training module discusses:

- Constructors; and
- Destructors.

What Is a Constructor?



A **constructor** function allocates storage for and initializes an object.

You can provide default, copy, move, conversion, and initialization constructor functions.

A class having no user-defined constructors has an implicit default constructor.

A class having no user-defined copy constructor has an implicit copy constructor.

A constructor function:

- Has the same name as the class itself
- Does not return a value (not even **void**)
- Can accept arguments
 - No parameters – default
 - Reference parameter same type – copy/move
 - One parameter of other type – converting
 - Multiple parameters – initialization
- Cannot be called explicitly (as a function)
 - Implicitly called upon object construction
 - Explicitly invoked from parent object initialization line

```
class StudentRecord {  
public:  
    // Conversion/Initialization Constructor  
    StudentRecord (unsigned n) : number(n) {}  
    ...  
private:  
    unsigned number;  
    char grade, name[20], surname[20];  
};
```

Conversion: Compiler can implicitly call constructor to convert unsigned student number to a StudentRecord. Prevent this by prepending keyword **explicit**.

Initialization: You explicitly provide initialization parameters.

71 © Cadence Design Systems, Inc. All rights reserved.



A **constructor** function allocates storage for and initializes an object.

You can provide default, copy, move, conversion, and initialization constructor functions.

A class having no user-defined constructors has an implicit default constructor.

A class having no user-defined copy constructor has an implicit copy constructor.

In a class having a move constructor or move assignment operator, the implicit copy constructor is automatically disabled.

A constructor is a member function having the same name as the class and returning no value, not even void. A constructor can accept any number of arguments. In fact, constructor overloading is very common.

- A default constructor has no parameters. If you do not provide any constructors, the compiler will automatically provide an implicit default constructor.
- A copy constructor has one reference parameter of the class type. It may also have other parameters if these others all have default values. A copy constructor initializes the member data of an object to the values of the member data of another object.
- A converting constructor has one parameter of any other type. You more typically provide a converting constructor for initialization purposes, but the compiler can also use it for explicit conversions, and if you omit the “explicit” specifier, can also use it for implicit conversions.
- All other constructors are usable only for initialization and you must explicitly invoke them.

When you construct an object you call its constructor function:

- The call occurs when you statically declare or dynamically allocate an object; and
- The call occurs when you invoke the constructor on the member initializer line of its parent object’s constructor.

You define a constructor function to initialize objects of your class. If you have no need to initialize objects then you usually do not need to define a constructor.

For this example, the Student Record class defines an initialization constructor that accepts an unsigned argument, and in the constructor member initializer list, initializes the private student identifier variable to that argument value.

Examples: Invoking Constructors

```
// When statically declaring an object  
// Syntax: ClassName instanceName ( arg1, arg2, ... );  
StudentRecord jane (123456789);  
StudentRecord john (987654321);  
  
// When dynamically allocating an object  
// Syntax: pointerName = new ClassName ( arg1, arg2, ... );  
StudentRecord *jane_ptr = new StudentRecord (123456789);  
StudentRecord *john_ptr = new StudentRecord (987654321);
```

72 © Cadence Design Systems, Inc. All rights reserved.



You explicitly invoke a constructor to construct an object each time you declare a variable of the constructor's class or dynamically allocate the object on the heap.

The 1st example, illustrates objects statically declared.

The 2nd example, illustrates objects dynamically allocated on the heap.

You can also on a constructor's member initializer list, invoke constructors of parent classes and constructors of member variables of class types.

The compiler will also implicitly invoke constructors to construct temporary objects during type conversion and assignment operations.

Deleted Versus Defaulted Constructor Definitions



A constructor or other function defined as ***deleted*** cannot be used. You explicitly define a function *deleted* to prevent the compiler from creating an implicit definition.



A special member function or comparison operator function defined as ***defaulted*** enables the compiler to create an implicit definition when it otherwise would not.

- A class having no user-defined constructors has implicit default, copy, and move constructors, and implicit copy and move assignment operators.
- To prevent creating implicit functions, you explicitly declare the functions and define them as *deleted*.
- A class having a user-defined non-default constructor will not have an implicit default constructor.
- To enable an implicit definition, you explicitly declare the constructor and define it as *defaulted*.

```
class C {
public:
    C() = delete; // No default
    C(const C&) = delete; // No copy
    C& operator=(const C&) = delete; // No copy
    C(C&)
    C& operator=(C&)
    // ...
};
```

```
class C {
public:
    C() = default; // Default
    C(const C&); // Copy
    C& operator=(const C&); // Copy
    C(C&); // Move
    C& operator=(C&); // Move
    // ...
};
```

73 © Cadence Design Systems, Inc. All rights reserved.

cadence

You define a function as *deleted* to prevent its use.

You define a special member function or comparison operator function as *defaulted* to enable creating an implicit definition.

The default, copy, and move constructors, copy assignment and move assignment operators, and prospective destructor, are termed “special member functions”. The implementation implicitly declares special member functions for a program that does not declare them

To prevent creating implicit functions, you explicitly declare the functions and define them as deleted.

The left example explicitly declares default, copy, and move constructors, and copy and move assignment operators, and defines them as deleted. These constructors cannot be used to construct an object of the class.

A class having a user-defined non-default constructor will not have an implicit default constructor.

To enable an implicit definition, you explicitly declare the constructor and define it as defaulted.

The right example explicitly declares a default constructor and defines it as defaulted. The implicit default constructor does default initialization, while a user-provided explicit default constructor does only the initialization that the user defines.

You can likewise declare defaulted comparison operator functions, as a convenience, to omit explicitly defining them. The implicit default comparison, compares all data members, while a user-provided explicit comparison, compares only the data members that the user programs it to compare.

If the class definition includes a user-defined destructor, then the copy constructor and copy assignment operator are not implicitly defined.

If the class definition includes a move constructor or move assignment operator, then the implicitly declared copy constructor is defined as deleted.

If the class definition includes a user-declared copy assignment operator, then the copy constructor is NOT implicitly defined.

If the class definition includes a user-declared copy constructor, then the copy assignment operator is NOT implicitly defined.

Best Practice When Copying Class Instances



The default copy constructor performs a shallow copy – does not allocate duplicate subobjects!

The built-in copy constructor performs a shallow copy:

- Copies all member data.
 - Both instances' member pointers have same values.
 - The pointed-to storage will likely get clobbered.

To perform a deep copy, you must implement:

- copy constructor
- copy assignment operator

```
class StudentRecord
{
public:
    // Default Constructor
    StudentRecord();
    // Initialization Constructor
    StudentRecord (unsigned n);
    // Destructor
    ~StudentRecord();
    void set_name (const char *name);
    ...
private:
    unsigned number;
    char grade, *name;
    void init_name (const char *name);
};

StudentRecord Sue, Sam;
Sam = Sue; Sue.set_name ("Sue");
cout << Sue.get_name(); // Sue
cout << Sam.get_name(); // Sue
```

Note that here name is pointer, not array

A boy named Sue.

74 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The default copy constructor performs a shallow copy. It does not allocate duplicate sub-objects!

The built-in copy constructor performs a shallow copy. It copies all member data. Member pointers of both instances have the same values. The pointed-to storage will likely get clobbered.

To perform a deep copy you must implement your own copy constructor or copy assignment operator.

Here, as Sam and Sue go out of scope, the duplicate attempt to delete the shared allocated name aborts the program.

Example: Defining a Copy Constructor

```
class StudentRecord {
public:
    // Default Constructor
    StudentRecord();
    // Initialization Constructor
    StudentRecord (unsigned n);
    // Copy Constructor
    StudentRecord (const StudentRecord&);
    // Destructor
    ~StudentRecord();
    void set_name (const char *name);
    ...
private:
    unsigned number;
    char grade, *name;
    void init_name (const char *name);
};
```

Note that here name is pointer, not array

```
// Default constructor
StudentRecord::StudentRecord () : name(nullptr) {}
// Initialization Constructor
StudentRecord::StudentRecord (unsigned n) : name(nullptr), number(n) {}

// Copy constructor
StudentRecord::StudentRecord (const StudentRecord &other) : number(other.get_number())
{ init_name(other.get_name()); }
```

```
void StudentRecord::init_name(const char *name) {
    this->name = new char[strlen(name)+1];
    strcpy(this->name, name);
}
```

```
void StudentRecord::set_name(const char *name) {
    delete [] this->name;
    init_name(name);
}
```

copy constructor initializes number and calls init_name()

init_name() allocates and copies name

set_name() deletes existing name and calls init_name()

75 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Here, the Student Record class defines a copy constructor to initialize the name by allocating heap space and copying characters into the new heap space.

The class definition, when changing an already-existing name, must also *de-allocate* the old heap space.

Comparing Copying and Moving



Copying is making a copy of an *lvalue*, which could be a very deep copy.



Moving is changing ownership of an *rvalue*, which could be much faster than making a very deep copy.

Let's illustrate *copying* an object.

```
// Copy constructor
StudentRecord(const StudentRecord& other)
: number(other.number)
, name(new char[strlen(other.name)+1])
{strcpy(name, other.name);}

// Copy assignment operator
StudentRecord& operator=(const StudentRecord& other) {
    if (this != &other) {
        number = other.number;
        delete [] name;
        name = new char[strlen(other.name)+1];
        strcpy(name, other.name);
    }
    return *this;
}
```

lvalue
allocation

Let's illustrate *moving* an object.

```
// Move constructor
StudentRecord(StudentRecord&& other)
: number(other.number)
, name(other.name)
{other.name = nullptr;

// Move assignment operator
StudentRecord& operator=(StudentRecord&& other) {
    if (this != &other) {
        number = other.number;
        delete [] name;
        name = other.name;
        other.name = nullptr;
    }
    return *this;
}
```

rvalue
appropriation

76 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Copying is making a copy of what we must assume is a left-value (*lvalue*) also referenced elsewhere, which could be a very deep expensive copy.

Moving is changing ownership of what we know to be a right-value (*rvalue*) used nowhere else, which could be much faster than making a very deep expensive copy.

The left example makes a complete new copy of the object and its variables and subobjects.

The right example makes a complete new copy of the object and its variables, but assumes ownership of subobjects without copying them. It assigns the null pointer value to the previous owner's subobject pointers, so that when the previous owner goes out of scope and is deleted, the subobjects that it no longer owns are not also deleted.

What Is a Destructor?



A **destructor** function gracefully releases resources allocated by the object.

All types have a built-in destructor that does not know about allocated subobjects.

You can provide a destructor function, and should for ephemeral objects.

A destructor function:

- Has the same name as the class, preceded by “~”.
- Cannot be called explicitly.
- Does not accept arguments.
- Does not return anything (not even **void**). 

Object deletion executes its destructor function:

- **For a static variable** – upon program end.
- **For an automatic variable** – upon execution going “out of scope”.
- **For a dynamic variable** – when you invoke the **delete** operator for it.

```
class StudentRecord
{
public:
    // Initialization Constructor
    StudentRecord (unsigned n);
    // Copy Constructor
    StudentRecord (const StudentRecord&);
    // Destructor
    ~StudentRecord() { delete [] name; }
    ...

private:
    unsigned number;
    char grade, *name;
    void init_name (const char *name);
};

}
```

77 © Cadence Design Systems, Inc. All rights reserved.



A **destructor** function gracefully releases resources allocated by the object.

All types have a built-in destructor that does not know about allocated sub-objects.

You can provide a destructor function, and for ephemeral objects, you should.

A destructor function:

- Has the same name as the class, preceded by a tilde character (“~”);
- Cannot be called explicitly;
- Does not accept arguments; and
- Does not return anything.

Object deletion executes its destructor function:

- For a static variable, upon program end;
- For an automatic variable, upon execution going “out of scope”; and
- For a dynamic variable, when you invoke the **delete** operator for it.

The Student Record constructor dynamically constructs the “name” subobject, so the Student Record destructor must likewise dynamically destroy the “name” subobject.

Example: Class Constructors and Destructor

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

class TransAttribute {
public:
    // Default constructor
    TransAttribute ( )                                // Default constructor
    : TransAttribute ( "unknown", 0 ) {}

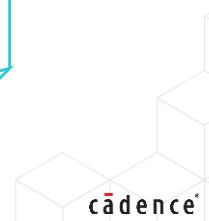
    // Copy constructor
    TransAttribute ( const TransAttribute& rhs )
    : TransAttribute ( rhs.name, rhs.value ) {}          // Copy constructor
                                                       // has ONE parameter of same type

    // Initialization constructor
    TransAttribute ( const char *name_, int value_ = 0 ) {   // Initialization constructor
        name = new char[strlen(name_)+1];
        strcpy(name, name_);
        value = value_;
    }

    // Destructor
    ~TransAttribute () {                                // Destructor
        delete [] name; }                            // has NO parameters

private:
    char *name;
    int value;
};
```

78 © Cadence Design Systems, Inc. All rights reserved.



The Transaction Attribute constructor dynamically constructs the “name” subobject, so the Transaction Attribute destructor must likewise dynamically destroy the “name” subobject.

Module Summary

In this module, you

- Initializes class variables during object construction
 - By defining class constructors
- Returned unneeded memory to the heap
 - By defining class destructors

This training module discussed:

- Constructors
- Destructors

79 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Initialize class variables during object construction, which you do by defining class constructors; and
- Return unneeded memory back to the heap, which you do by defining class destructors.

To help you to achieve your objective, this training module discussed:

- Constructors; and
- Destructors.

Quiz: Constructors and Destructors



What is the purpose of a constructor function?



What constructors does every class have even if you do not provide them?



What is the purpose of the destructor function?



Suggest a situation in which you should provide a destructor.

Please pause here to consider these questions.



Lab

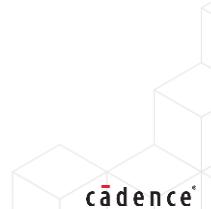
Lab 5-1 Defining Constructors and a Destructor

Objective:

- To define and test a counter class with constructors and a destructor

For this lab, you:

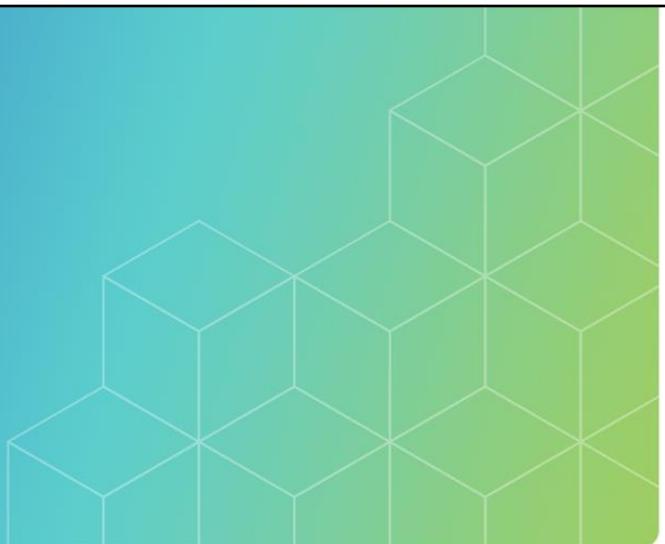
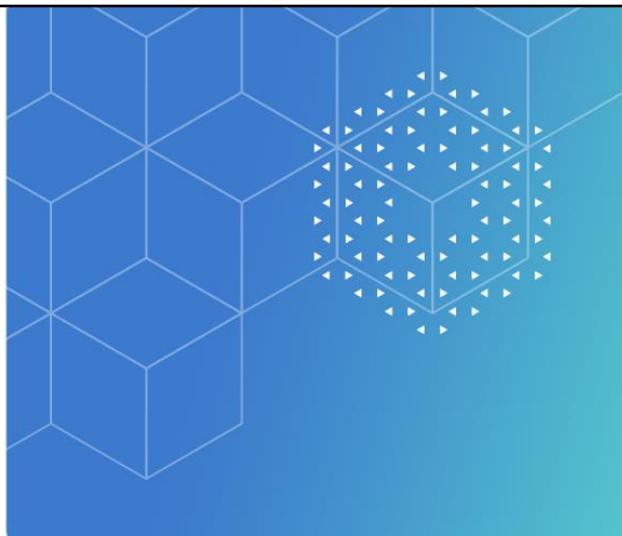
- Define a counter class with default, copy, and initialization constructors, a destructor, and functions to set, get, and increment the count
- Write a test to confirm constructor and function operation



Your objective for this lab is to define and test a counter class that has constructors and a destructor.

For this lab, you:

- Define a counter class with default, copy, and initialization constructors, a destructor, and functions to set, get, and increment the count; and
- Write a test to confirm constructor and function operation.



Module 6

References

cadence®

This training module examines C++ references.

Module Objectives

In this module, you

- Define aliases for variables
 - By defining *reference variables*

This training module discusses:

- Reference Variables
- Introduction to Pass by Reference



Your objective is to:

- Define aliases for variables, by defining reference variables

To help you to achieve your objective, this training module discusses:

- Reference Variables; and
- Introduction to Pass by Reference.

What Is a Reference Variable?



A **Reference Variable** is a *name* for an object.

Declare an lvalue reference with the “**&**” operator.

Declare an rvalue reference w/ the “**&&**” operator.

- You generally initialize a reference identifier upon declaration.
- The reference identifier and original identifier refer to the same object.
- You cannot change what the reference refers to!

```
int x = 3;
int &x_ref = x; // x_ref is a reference to x
x_ref = 7;      // x is now 7
int y = 5;
x_ref = y;      // x is now 5
```

```
int *x_ptr = &x; // address operator
// (not a reference)
```

84 © Cadence Design Systems, Inc. All rights reserved.



A reference variable is another name for a variable.

You declare a reference variable by prefixing the ampersand (**&**) token to the reference identifier.

- A single ampersand declares a left-value (*lvalue*) reference.
- A double ampersand declares a right-value (*rvalue*) reference.
- C++ treats a named right-value reference as a left-value.
- The distinction is more useful as the declaration of a function parameter or function return type.

You generally initialize a reference variable upon its declaration.

- You omit the initializer if the declaration contains an explicit extern specifier, is a class member declaration within a class definition, or is the declaration of a function parameter or a function return type.

Within its scope, the reference identifier and the original identifier refer to the same variable.

You cannot later change what variable the reference refers to.

The example declares a variable and a reference to that variable. The reference is another name for the variable. Be sure to not confuse the reference operator with the address operator.

ISO/IEC 14882:2020 7.2.1 Value category

ISO/IEC 14882:2020 9.3.3.2 References

Example: Using Reference Variables

```

int original;                                // declare original variable
int &lval_ref = original;                     // declare lvalue reference to original
int &&rval_ref = (int&&)original;           // declare rvalue reference to original

original = 2;                                 // update value of original
lval_ref = 5;                               // update value of original
rval_ref = 7;                               // update value of original

cout << original << endl;                  // output the value of original, "7"
cout << lval_ref << endl;                  // output the value of original, "7"
cout << rval_ref << endl;                  // output the value of original, "7"

cout << &original << endl;                 // output the address of original
cout << &lval_ref << endl;                 // output the address of original
cout << &&rval_ref << endl;                // output the address of original

```

85 © Cadence Design Systems, Inc. All rights reserved.



This example illustrates that you can for assignments interchangeably use the “original” variable and its references.

Notwithstanding that the term “right-value” (*rvalue*) implies that it can appear on only the right side of an assignment, C++ treats a named right-value reference as a left-value (*lvalue*), so you see here that you can make an assignment to a right-value reference.

To use reference variables in this manner is not common. People generally use reference operators for function parameters and function return, where a left-value reference implies copy semantics, and a right-value reference implies move semantics.

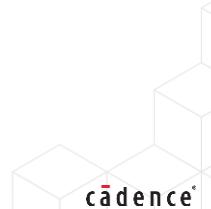
C++ has multiple uses for the ampersand (&) token. This example also shows its use as an address operator. It is also a bitwise conjunction operator.



Quick Reference Guide: Using Reference Variables

Syntax	Example	Description
<code>type &referenceName = originalName;</code>	<code>int intVar; int &intRef = intVar;</code>	Declaration and initialization
<code>originalName &referenceName</code>	<code>intVar = 1; intRef = 2;</code>	Accessing the object's <i>value</i> Accessing the object's <i>value</i>
<code>originalName.memberName &referenceName.memberName</code>	<code>Class_c classObj; Class_c &classRef = classObj; classObj.intVar = 1; classRef.intVar = 2;</code>	Accessing the object's <i>members</i> Accessing the object's <i>members</i>
<code>&originalName &referenceName</code>	<code>Class_c *classPtr1 = &classObj; Class_c *classPtr2 = &classRef;</code>	Accessing the object's <i>address</i> Accessing the object's <i>address</i>

86 © Cadence Design Systems, Inc. All rights reserved.



You declare a reference variable by prefixing the ampersand (&) token to the reference identifier. You initialize a reference variable upon its definition and cannot later change what variable it refers to.

You use the reference variable name interchangeably with the original variable name. They are two names for the same variable.

To use reference variables in this manner is not common. People generally use reference operators for function parameters and function return, where a left-value reference implies copy semantics, and a right-value reference implies move semantics.

Note that C++ has multiple uses for the ampersand (&) token. Here, we also show it taking an object's address.

Comparing Function Argument Passing by Value, Pointer, & Reference



Passing by *value* passes a *copy* of the passed object.

Declare the function like this:

```
Class_a funcVal (Class_b obj);
```

Call the function with an *object*:

```
object_a = funcVal(object_b);
```

Advantage:

- Simplicity

Disadvantage:

- Cannot modify original object.
- Inefficient to copy a **HUGE** object of which only parts may be used only temporarily.



Passing by *pointer* passes a *pointer* to the passed object.

Declare the function like this:

```
Class_a *funcPtr(Class_b *ptr);
```

Call the function with an *address*:

```
obj_a_ptr = funcPtr(&object_b);
```

Advantage:

- Efficient way to pass argument.
- Can modify a non-**const** object.
 - Prevent by declaring parameter:
 - **const** Class_c *ptr

Disadvantage:

- Easy to inadvertently abuse pointer!



Passing by *reference* passes a *reference* to the passed object.

Declare the function like this:

```
Class_a &funcRef (Class_b &ref);
```

Call the function *normally*:

```
object_a = FuncRef(object_b);
```

Advantage:

- No pointer abuse!
- Efficient way to pass argument.
- Can modify a non-**const** object.
 - Prevent by declaring parameter:
 - **const** Class_c &ptr



87 © Cadence Design Systems, Inc. All rights reserved.

By default, what you pass to a function, and return from a function, is the *value* of an object.

- You declare the function to accept an object of a type, and return an object of a type.
- When you call the function, you pass to it an object of that type. The argument object is copied to the function parameter. Upon function return, the returned object is, conceptually at least, copied to the calling expression. Almost all compilers actually optimize away that extra copy operation.
- This approach has the advantage that usage is simple and intuitive.
- This approach has the two disadvantages that the function cannot modify the original object, and to copy the object may be unnecessarily expensive to performance.

You can alternatively pass to a function, and return from a function, a *pointer* to an object.

- You declare the function to accept a pointer to a type, and return a pointer to a type.
- When you call the function, you pass to it a pointer of that type. Only the pointer is copied to the function parameter. Upon function return, only the returned pointer is copied to the calling expression.
- This approach has the two advantages that passing a pointer is very efficient, and unless you declare the pointer type as to a constant object, the function can modify the original object.
- This approach has the disadvantage that for the less-experienced programmer, use of pointers increases the likelihood of error. You need a way to pass arguments both safely *and* efficiently.

C++ provides a way to pass to a function, and return from a function, a *reference* to an object.

- You declare the function to accept a reference to a type, and return a reference to a type.
- When you call the function, you pass to it an object of that type. The compiler copies only the object reference, similar to a pointer, to the function parameter. Upon function return, the compiler copies only the returned object reference to the calling expression.
- This approach has all the advantages of passing and returning pointers, but without the programmer getting involved with pointers or with the address and indirection operators.

Comparing Function Argument Passing by lvalue Reference & rvalue Reference



Passing a function argument by *lvalue* reference (`&`) forces copy construction and assignment.



Passing a function argument by *rvalue* reference (`&&`) enables move construction and assignment.

Copying an object means copying its subobjects.

```
// Copy assignment operator
StudentRecord& operator=(const StudentRecord& other) {
    if (this != &other) {
        number = other.number;
        delete [] name;
        name = new char[strlen(other.name)+1];
        strcpy(name, other.name);
    }
    return *this;
}
StudentRecord Sam(42, "Sam");
StudentRecord Dan;
Dan = Sam;
```

lvalue

allocation

copy lvalue Sam

Moving means appropriating the subobjects.

```
// Move assignment operator
StudentRecord& operator=(StudentRecord&& other) {
    if (this != &other) {
        number = other.number;
        delete [] name;
        name = other.name;
        other.name = nullptr;
    }
    return *this;
}
StudentRecord Dan;
Dan = StudentRecord(42, "Sam");
```

rvalue

appropriation

move rvalue expr

Passing a function argument by left-value (*lvalue*) reference (`&`) forces copy construction and assignment.

Passing a function argument by right-value (*rvalue*) reference (`&&`) enables move construction and assignment.

Copying an object means copying its subobjects.

The copy assignment operator must de-allocate the target object's heap space, allocate new heap space, and copy the source's resources to the target's heap space.

Moving means appropriating the subobjects. We can do this only if we know that the source is a temporary object whose resources can be reused.

The move assignment operator must simply appropriate the sources resources and nullify any pointers the source has to those resources.

What Is a Reference-Qualified Function?



A **reference qualifier** (`&`, `&&`) selects between overloaded non-static member functions by whether the current object is an *lvalue* or an *rvalue*.

Selection assumes[†] by default that the current object is an *lvalue* requiring *copy* semantics.

Knowing that the current object is an *rvalue* enables *move* semantics.

[†]This means that you can call a function not reference-qualified and expecting an *lvalue* object on behalf of a *rvalue* object – with unintended results!

```
class Person {
public:
    Person(const char *name_) { // Constructor init
        name = new char[strlen(name_)+1];
        strcpy(name, name_);
    }
    Person(const Person &lPerson) { // Constructor copy
        name=new char[strlen(lPerson.name)+1];
        strcpy(name, lPerson.name);
    }
    Person(Person &&rPerson) { // Constructor move
        name = rPerson.name;
        rPerson.name = nullptr;
    }
private: char *name;
};

class Group {
public:
    Person &get_person() & {return person;} // lvalue ref
    Person &&get_person() && {return move(person);} // rvalue ref
private: Person person("unknown");
};

int main() {
    Group g;
    Person p1(g.get_person());
    Person p2(Group().get_person());
    return 0;
} // copy person from lvalue
// move person from rvalue
```

89 © Cadence Design Systems, Inc. All rights reserved.



A **reference qualifier** (`&`, `&&`) selects between overloaded non-static member functions by whether the current object is a left-value (*lvalue*) or a right-value (*rvalue*)

Absent a reference qualifier, selection assumes that the current object is a left-value requiring *copy* semantics.

Knowing that the current object is a right-value enables *move* semantics, which preserves execution performance.

You can inadvertently call a function *not* reference-qualified, and thus expecting by default a left-value object, on behalf of a right-value object, with unintended results! So perhaps as a habit, you should reference-qualify all your non-static member functions.

The example:

- Defines a class “Person” having constructors for initialization, copy, and move.
- Defines a class “Group” having reference-qualified functions, returning respectively, a left-value Person reference, and a right-value Person reference.
 - The left-value reference function *copies* the person properties.
 - The right-value reference function *moves* the person properties. Here, the “move” function does not perform the move, but only casts the right-value reference to an expiring-value (*xvalue*) reference.
- The application code, gets a person from a left-value object, which *copies* the person, and gets a person from a right-value object, which *moves* the person.

Module Summary

In this module, you

- Defined aliases for variables
 - By defining *reference variables*

This training module discussed:

- Reference Variables
- Introduction to Pass by Reference

90 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define aliases for variables, by defining reference variables

To help you to achieve your objective, this training module discussed:

- Reference Variables; and
- Introduction to Pass by Reference.

Quiz: References



Explain how the reference operator (&) and address-of operator (&) differ?



Suggest why you might prefer *pass-by-reference* over *pass-by-pointer*.

Please pause here to consider these questions.



Lab

Lab 6-1 Experimenting with Reference Variables

Objective:

- To declare, initialize, and test a reference variable

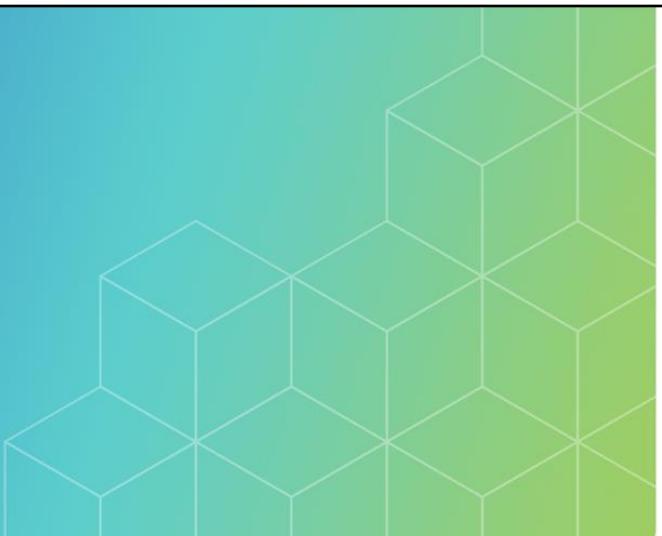
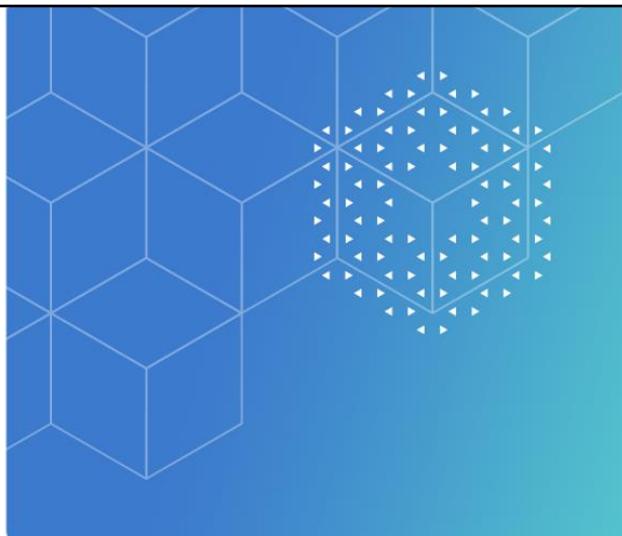
For this lab you:

- Add a reference variable to your counter class, initialize the variable, and use it in at least one class function



Your objective for this lab is to declare, initialize, and test a reference variable.

For this lab you add a reference variable to your counter class, initialize the variable, and use it in at least one of the class functions.



Module 7

Functions

cadence®

This training module examines some C++ features related to functions.

Module Objectives

In this module, you

- Choose function argument passing by *value*, *pointer*, or *reference*, to optimize program reliability and runtime performance
- Define overloaded functions, to simplify the application interface to the class
- Define *static* class data and function members independent of class objects

This training module discusses:

- Function Argument Passing
- Function Overloading
- Global and Local Scope
- Static Class Members

94 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Achieve the best compromise between program reliability and runtime performance, which you do by appropriately choosing function argument passing by value, pointer, or reference;
- Simplify the application interface to the class, which you do by defining overloaded functions; and
- Define class data and functions independent of class objects, which you do by declaring them static.

To help you to achieve your objective, this training module discusses:

- Function Argument Passing;
- Function Overloading;
- Global and Local Scope; and
- Static Class Members.

Comparing Function Argument Passing by Value, Pointer, & Reference



Passing by *value* passes a *copy* of the passed object.

Declare the function like this:

```
Class_a funcVal (Class_b obj);
```

Call the function with an *object*:

```
object_a = funcVal(object_b);
```

Advantage:

- Simplicity

Disadvantage:

- Cannot modify original object.
- Inefficient to copy a **HUGE** object of which only parts may be used only temporarily.



Passing by *pointer* passes a *pointer* to the passed object.

Declare the function like this:

```
Class_a *funcPtr(Class_b *ptr);
```

Call the function with an *address*:

```
obj_a_ptr = funcPtr(&object_b);
```

Advantage:

- Efficient way to pass argument.
- Can modify a non-**const** object.
 - Prevent by declaring parameter:
 - **const** Class_c *ptr

Disadvantage:

- Easy to inadvertently abuse pointer!



Passing by *reference* passes a *reference* to the passed object.

Declare the function like this:

```
Class_a &funcRef (Class_b &ref);
```

Call the function *normally*:

```
object_a = FuncRef(object_b);
```

Advantage:

- No pointer abuse!
- Efficient way to pass argument.
- Can modify a non-**const** object.
 - Prevent by declaring parameter:
 - **const** Class_c &ptr



95 © Cadence Design Systems, Inc. All rights reserved.

By default, what you pass to a function, and return from a function, is the *value* of an object.

- You declare the function to accept an object of a type, and return an object of a type.
- When you call the function, you pass to it an object of that type. The argument object is copied to the function parameter. Upon function return, the returned object is, conceptually at least, copied to the calling expression. Almost all compilers actually optimize away that extra copy operation.
- This approach has the advantage that usage is simple and intuitive.
- This approach has the two disadvantages that the function cannot modify the original object, and to copy the object may be unnecessarily expensive to performance.

You can alternatively pass to a function, and return from a function, a *pointer* to an object.

- You declare the function to accept a pointer to a type, and return a pointer to a type.
- When you call the function, you pass to it a pointer of that type. Only the pointer is copied to the function parameter. Upon function return, only the returned pointer is copied to the calling expression.
- This approach has the two advantages that passing a pointer is very efficient, and unless you declare the pointer type as to a constant object, the function can modify the original object.
- This approach has the disadvantage that for the less-experienced programmer, use of pointers increases the likelihood of error. You need a way to pass arguments both safely *and* efficiently.

C++ provides a way to pass to a function, and return from a function, a *reference* to an object.

- You declare the function to accept a reference to a type, and return a reference to a type.
- When you call the function, you pass to it an object of that type. The compiler copies only the object reference, similar to a pointer, to the function parameter. Upon function return, the compiler copies only the returned object reference to the calling expression.
- This approach has all the advantages of passing and returning pointers, but without the programmer getting involved with pointers or with the address and indirection operators.

Comparing Function Argument Passing by lvalue Reference & rvalue Reference



Passing a function argument by *lvalue* reference (`&`) forces copy construction and assignment.



Passing a function argument by *rvalue* reference (`&&`) enables move construction and assignment.

Copying an object means copying its subobjects.

```
// Copy assignment operator
StudentRecord& operator=(const StudentRecord& other) {
    if (this != &other) {
        number = other.number;
        delete [] name;
        name = new char[strlen(other.name)+1];
        strcpy(name, other.name);
    }
    return *this;
}
StudentRecord Sam(42, "Sam");
StudentRecord Dan;
Dan = Sam;
```

lvalue

allocation

copy lvalue Sam

Moving means appropriating the subobjects.

```
// Move assignment operator
StudentRecord& operator=(StudentRecord&& other) {
    if (this != &other) {
        number = other.number;
        delete [] name;
        name = other.name;
        other.name = nullptr;
    }
    return *this;
}
StudentRecord Dan;
Dan = StudentRecord(42, "Sam");
```

rvalue

appropriation

move rvalue expr

Passing a function argument by left-value (*lvalue*) reference (`&`) forces copy construction and assignment.

Passing a function argument by right-value (*rvalue*) reference (`&&`) enables move construction and assignment.

Copying an object means copying its subobjects.

The copy assignment operator must de-allocate the target object's heap space, allocate new heap space, and copy the source's resources to the target's heap space.

Moving means appropriating the subobjects. We can do this only if we know that the source is a temporary object whose resources can be reused.

The move assignment operator must simply appropriate the sources resources and nullify any pointers the source has to those resources.

What Is Function Overloading?



Function Overloading is declaring in the same scope multiple functions having the same name, and either different parameter type lists, or (for non-static member functions) different const-volatile or reference qualifiers. Return type and exception specification do not participate in overload resolution, but the return type *can* differ.

You can *overload* in the same scope multiple function declarations having the same name.

- The compiler conveniently matches the call to the appropriate declaration, depending upon the number and type of the arguments.
- The compiler can convert types to match a call to a function declaration.
- Not all function declarations can be overloaded.
 - Reference-qualified versus not reference-qualified
 - static versus non-static

```
// Function prototypes
bool add(int, int);           // prototype 1
int add(int, int, int);        // prototype 2
float add(double, int);        // prototype 3
struct S {
    int f()& {cout << "Object is lvalue" << endl;}
    float f()&& {cout << "Object is rvalue" << endl;}
};
...
// Function calls
double d1 = add(5, 10);        // uses prototype 1
double d2 = add(5, 10, 20);     // uses prototype 2
double d3 = add(0.5, 10);       // uses prototype 3
double d4 = add(0.5, 0.1);      // uses prototype 3
S s;
double d5 = s.f();             // Object is lvalue
double d6 = S().f();            // Object is rvalue
```

97 © Cadence Design Systems, Inc. All rights reserved.



Function Overloading is declaring in the same scope multiple functions having the same name, and either different parameter type lists, or for non-static member functions, different constant-volatile or reference qualifiers. Return type and exception specification do not participate in overload resolution.

- The compiler conveniently matches the call to the appropriate declaration, depending upon the number and type of the arguments.
- The compiler can convert types to match a call to a function declaration.
- Not all function declarations can be overloaded. For example, you cannot overload a reference-qualified non-static member function with a function that is not reference-qualified, and you cannot overload a static member function.

The example defines overloaded functions. The overload resolution utilizes the parameter type list, and for non-static member functions, a reference qualifier, if it exists.

The compiler attempts conversions in the order of precedence:

- 1st – Exact match, or trivial conversions such as non-const pointer to const pointer;
- 2nd – Matches after promotion, such as int to long and float to double;
- 3rd – Matches after standard conversions such as int to double and signed to unsigned;
- 4th – Matches using user-defined conversions; and
- 5th – Matches to functions declaring a variable number of arguments;

For functions with multiple arguments, the function called is the function with a match better than the others for at least one argument and not worse than the others for the remaining arguments. If the compiler cannot find a function meeting this criteria then it issues an error.

ISO/IEC 14882:2020 12.2 Overloadable declarations

What Is Scope?



Scope is a region of program text in which a name may be referenced without qualification.

Names are unique within a scope:

- The same name in a different scope is a different entity.
- A more local declaration of a name “hides” a less local declaration that uses the same name.

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

int i=0;           // namespace (global) scope

int main()
{
    int i = 1;      // block scope
    cout << "i = " << i << endl;
    return 0;
}
```

Scope is a region of program text in which a name may be referenced without qualification.

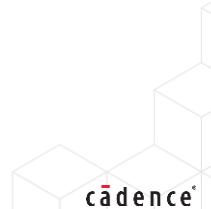
A declaration introduces a name into a scope. The name is visible from the point of declaration to the end of the declaration scope. Namespace scopes do not end.

Names are usually unique within a scope, though exceptions do exist that we do not discuss here.

- The same name in a different scope is a different instance.
- A more local declaration of a name hides a less local declaration using the same name.

Example: C++ Scopes

Scope	Potential Extent
Block	From point of declaration to end of block. Local or function-local variables.
Namespace	From point of declaration onward (namespaces are open). For global and anonymous namespaces from point of declaration to translation unit end.
Class	From point of declaration to end of the class definition and also within <i>member function</i> declarations and definitions.



A name declared in a block is local to the block. Its potential scope is from the point of declaration to the block end. This applies also to a name declared in a function definition.

A name declared in a namespace is a member name. Its potential scope is from the point of declaration onwards. For anonymous namespaces, that can have only one declarative region, that potential scope ends at the translation unit end. This applies also to the “global” namespace.

The potential scope of a class member name is from point of declaration to the class definition end, and also within function default arguments and bodies, and in non-static data initializers, and in exception specifications.

Example: Scope Resolution

```

int var1 = 1, var2 = 3; // global namespace
int fnc1() {return 1;}; int fnc2() {return 3;};

namespace NS {
    int var1 = 5, var3 = 7; // named namespace
    int fnc1(); int fnc3() {return 7;};

    int fnc1() {
        int var1 = 9; // block (function local)
        cout << var1 << endl; // local -- prints out "9"
        cout << NS::var1 << endl; // named -- prints out "5"
        cout << ::var1 << endl; // global -- prints out "1"
        cout << ::fnc1() << endl;
        cout << var2 << endl; // global -- but not ambiguous
        cout << fnc2() << endl;
        cout << var3 << endl; // named -- but not ambiguous
        cout << fnc3() << endl;
        return 5;
    }
}

int main()
{
    NS::fnc1();
    return 0;
}

```

100 © Cadence Design Systems, Inc. All rights reserved.



This example, in the global namespace, declares the variables “variable 1” and “variable 2”, and the functions “function 1” and “function 2”.

This example, in the named namespace “NS”, declares the variables “variable 1” and “variable 3”, and the functions “function 1” and “function 3”. The named namespace declaration of “variable 1” hides the global namespace declaration of “variable 1”. Following the named namespace declaration of “variable 1”, any unqualified reference to “variable 1” refers to the “variable 1” of the named namespace. Likewise for “function 1”.

The function “function 1” also declares its own variable “variable 1”. This local declaration of “variable 1” hides the named namespace declaration of “variable 1”. Following the local declaration of “variable 1”, any unqualified reference to “variable 1” refers to the “variable 1” of the local block.

To qualify a reference, you use the scope resolution operator. The function directly refers to its own declaration of “variable 1”, and then by name qualification, refers to the named namespace declaration of “variable 1”, and then refers to the global namespace declarations of “variable 1” and “function 1”.

What Is a Const-Volatile Qualified Function?



A **constant** or **volatile** qualifier (**const**, **volatile**) selects between overloaded non-static member functions by whether the current object is constant or volatile or both.

A program can (by default) change any object.

- A constant (**const**) object cannot change.

An object can be changed by (by default) only the program.

- A volatile (**volatile**) object can change unbeknownst to the program.

Only functions at least as qualified as the object can act upon the object.

- Only a **const [volatile]** function can act on a **const** object.
 - It promises to not modify the object
- Only a **[const] volatile** function can act on a **volatile** object.
 - It notifies the compiler not to optimize "away" seemingly duplicate reads of the object.
- Only a **const volatile** function can act on a **const volatile** object.

```
class myClass {  
public:  
    void func() {}  
    void func() const {}  
    void func() volatile {}  
    void func() const volatile {}  
};  
int main() {  
    myClass m ; m.func() ;  
    const myClass mc ; mc.func() ;  
    volatile myClass mv ; mv.func() ;  
    const volatile myClass mcv ; mcv.func() ;  
    return 0;  
}
```

101 © Cadence Design Systems, Inc. All rights reserved.



A “constant” (**const**) or “**volatile**” qualifier selects between overloaded non-static member functions by whether the current object is constant or volatile or both.

A program can, by default, change any object, but a constant (**const**) object cannot change.

An object can be changed by, by default, only the program, but a “**volatile**” object can change unbeknownst to the program.

Only functions at least as qualified as the object can act upon the object.

- Only a constant function can act on a constant object. It promises to not modify the object.
- Only a volatile function can act on a volatile object. It notifies the compiler to not reuse previously read values.
- An object can be both constant and volatile, in which case only a constant volatile function may act upon it.

The example defines each flavor of overloaded function. Overload resolution chooses the function in part by whether the object is constant or volatile or both.

What Is a Reference-Qualified Function?



A **reference qualifier** (`&`, `&&`) selects between overloaded non-static member functions by whether the current object is an *lvalue* or an *rvalue*.

Selection assumes[†] by default that the current object is an *lvalue* requiring *copy* semantics.

Knowing that the current object is an *rvalue* enables *move* semantics.

[†]This means that you can call a function not reference-qualified and expect an *lvalue* object on behalf of an *rvalue* object – with unintended results!

```
class Person {
public:
    Person(const char *name_) { // Constructor init
        name = new char[strlen(name_)+1];
        strcpy(name, name_);
    }
    Person(const Person &lPerson) { // Constructor copy
        name=new char[strlen(lPerson.name)+1];
        strcpy(name, lPerson.name);
    }
    Person(Person &&rPerson) { // Constructor move
        name = rPerson.name;
        rPerson.name = nullptr;
    }
private: char *name;
};

class Group {
public:
    Person &get_person() & {return person;} // lvalue ref
    Person &&get_person() && {return move(person);} // rvalue ref
private: Person person("unknown");
};

int main() {
    Group g;
    Person p1(g.get_person());
    Person p2(Group().get_person());
    return 0;
} // copy person from lvalue
// move person from rvalue
```

102 © Cadence Design Systems, Inc. All rights reserved.



A **reference qualifier** (`&`, `&&`) selects between overloaded non-static member functions by whether the current object is a left-value (*lvalue*) or a right-value (*rvalue*)

Absent a reference qualifier, selection assumes that the current object is a left-value requiring *copy* semantics.

Knowing that the current object is a right-value enables *move* semantics, which preserves execution performance.

You can inadvertently call a function *not* reference-qualified, and thus expecting by default a left-value object, on behalf of a right-value object, with unintended results! So perhaps as a habit, you should reference-qualify all your non-static member functions.

The example:

- Defines a class “Person” having constructors for initialization, copy, and move.
- Defines a class “Group” having reference-qualified functions, returning respectively, a left-value Person reference, and a right-value Person reference.
 - The left-value reference function *copies* the person properties.
 - The right-value reference function *moves* the person properties. Here, the “move” function does not perform the move, but only casts the right-value reference to an expiring-value (*xvalue*) reference.
- The application code, gets a person from a left-value object, which *copies* the person, and gets a person from a right-value object, which *moves* the person.

What Is a Function Object (functor)?



A **function object (functor)** is an object of a class that overloads the function call operator (operator()), thus enabling “calling” the object by using function call syntax.

Many iterative algorithms take as an argument a unary function, passing in turn to the function each container item.

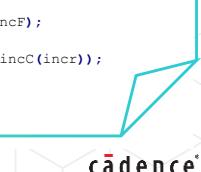
- The function takes one argument – the current item.
- To pass multiple arguments, you replace the function with a function object.
 - You construct the object with whatever number of additional arguments.

```
int incF (int i) { return i+1; } // increment function

class incC { // increment class
public: incC(int n) : n(n) {}
    int operator()(int i) const{return i+n;}
private: int n;
};

int main() {
    const unsigned int size = 8, incr = 3;
    using container_t = array<int,size>;
    container_t cont;
    int i=0;
    for (auto &elem : cont) elem=i++;
    cout << cont << endl; // 0123... user-defined operator
                           // transform copies function-transformed
                           // elements from sequence to sequence
    transform(cont.begin(),cont.end(),cont.begin(),incF);
    cout << cont << endl; // 1234...
    transform(cont.begin(),cont.end(),cont.begin(), incC(incr));
    cout << cont << endl; // 4567...
    return 0;
}
```

103 © Cadence Design Systems, Inc. All rights reserved.



A **function object (functor)** is an object of a class that overloads the function call operator (operator()), thus enabling “calling” the object by using function call syntax.

Many iterative algorithms take as an argument a unary function, passing in turn to the function each container item.

- The function takes one argument – the current item.
- To pass multiple arguments, you replace the function with a function object.
 - You construct the function object with a number of additional arguments.

The example first shows an “increment” function that returns the parameter argument plus 1. The value to add is hard-coded. The function takes only one argument.

The example then shows an “increment” class that is constructed with the value to add. The class overloads the function call operator to return its parameter argument plus the constructed value. The value to add is provided as the function object is constructed. The function still takes only one argument.

The application code initializes an array, and calls the transform function twice, first passing the “increment” function, then passing the “increment” function object.

What Is a Lambda Expression?



A **lambda expression** is an anonymous function object (*functor*) defined at the point of instantiation.

Advantage:

- Shorter than defining/instantiating a function object.
- Definition can access local scope variables.

Disadvantage:

- Definition cannot be instantiated elsewhere.

```
class incC {                                // increment class
public: incC(int n) : n(n) {}              int operator()(int i) const{return i+n;}
private: int n;
};
int main() {
    const unsigned int size = 8, incr = 3;
    using container_t = array<int,size>;
    container_t cont;
    int i=0;
    for (auto &elem : cont) elem=i++;
    cout << cont << endl;      // 0123... user-defined operator
    // transform copies function-transformed
    // elements from sequence to sequence
    transform(cont.begin(),cont.end(),cont.begin(),
              incC(incr));
    cout << cont << endl;      // 3456...
    transform(cont.begin(),cont.end(),cont.begin(),
              [](int i){return i+incr;});
    cout << cont << endl;      // 6789...
    return 0;
}
```

104 © Cadence Design Systems, Inc. All rights reserved.



A **lambda expression** is an anonymous function object (*functor*) defined at the point of instantiation.

You might choose a lambda expression, instead of a named function object, because it requires less code, and because the definition can access scope variables.

As the lambda expression has no name, you cannot instantiate it elsewhere, but you would typically use a lambda expression for only short normally-inline functions.

The example shows an “increment” class that is constructed with the value to add. The class overloads the function call operator to return its parameter argument plus the constructed value. The value to add is provided as the function object is constructed. The overloaded function takes only one argument.

The application code initializes an array, and calls the transform function twice, first passing the “increment” function object, and then passing a lambda expression.

The term “lambda expression” derives from the mathematical system “lambda calculus”.

Defining a Lambda Expression



```
[ [captures] ] [<t-parameters> [requires-clause]]C++20 [declarators] { [statements] }
declarators ::= (parameters) [specifiers] [noexcept [(const-expr)] [attributes] [->
typeid] [requires-clause]C++20
```

captures	List of comma-separated variable captures.	Capture	Description
t-parameters	Template parameters as in template function.	= id [...] init	Capture by copying all referenced vars.
parameters	Parameter list as in function declaration. Accepts type auto as of C++14. Omitting parameters implies omitting template. As of C++20 including any declarators implies including parameters. This may change.	& &id [...] init	Capture by copy specific var that optionally is either a pack expansion or has an initializer.
specifiers	mutable , constexpr ^{C++17} or consteval ^{C++20} .		Capture by reference all referenced vars.
attributes	Attribute specifications pertaining to the function call operator type.		Capture by reference specific var that optionally is either a pack expansion or has an initializer.
typeid	Returned type. Omit declarator if type inferable by return statement.	[*]this	Capture this by reference or optionally by copy.
requires	Constrains function call operator as of C++20.	[&] ... id init	Capture by copy or optionally by reference with initializer that is a pack expansion.

```
[] (double d){return d+3.14;} // implied return type
[] (double d)->double{if (d<0) return 0; else return d+3.14;} // specified type
[min] (double d) {if (d<min) return min; else return d+3.14;} // captured var
```

105 © Cadence Design Systems, Inc. All rights reserved.

cadence[®]

You define a lambda expression in four parts.

- The 1st part is a required lambda introducer. Within the introducer, you can optionally capture variables for use within the lambda expression. You can omit the captures, but must include the square brackets. Some variables, you cannot capture. Some variables, the expression can use without capture. You can capture variables by copy or by reference. You can specify to capture by default all referenced variables by copy or by reference, and then specify to capture individual variables the other way. Capture is by default by copy.
- The 2nd part is an optional template parameter list, as in a function template, and optionally with it, a requires clause, to place compile-time constraints on the template arguments.
- The 3rd part is optional declarators.
 - As of C++ 20, if you include any other declarators, then you must include the parameter declaration clause. As with any function definition, the parameter declaration clause can be empty.
 - Declaration specifiers may include the keywords “**mutable**”, “constant expression” (**constexpr**), and “constant evaluation” (**consteval**).
 - The keyword “**mutable**” enables the function to modify variables captured by copy.
 - The keyword “constant expression” (**constexpr**), explicitly declares the function to be a constant expression, that is, can be evaluated at compile time. A function is implicitly a constant expression anyway if it satisfies the requirements of a constant expression. You cannot provide both this keyword and the “constant evaluation” (**consteval**) keyword.
 - The keyword “constant evaluation” (**consteval**), declares the function to be an immediate function, that is, all calls of the function return a compile-time constant. You cannot provide both this keyword and the “constant expression” (**constexpr**) keyword.
 - Declarators may include a no-exceptions specifier, to optionally specify compile-time conditions under which the function call operator may throw exceptions.
 - Declarators may include attribute specifiers pertaining to the function call operator type.
 - Declarators may include a trailing return type, that is otherwise inferred from the **return** statement, and missing a return statement, assumed **void**; and
 - Declarators may include a requires clause, another way to place compile-time constraints on the function template arguments.
- The 4th part is a required compound statement. As with any function definition, you can omit the statements, but must include the curly braces.

What Is Static Class Member Data?



A **static** data member is one copy accessible independently of class objects.

The class controls the member's definition space and access.

The member is not a class subobject.

Declare member data **static** to share one copy among all code independently of class objects.

- Declare the member *inside* the class definition.
 - Define the member *outside* the class definition.
- The variable is “global” within the class definition.
- Safer than a truly global variable – access can be controlled.
 - If **public**: Access with class name and scope (::) operator or object name and dot (.) operator.
 - If **private**: Access only through the public interface.

```
class Myclass {  
    private: static const int sci = 5; // static const int  
    public: static int si; // static int  
};  
  
const int Myclass::sci; // define outside. don't repeat  
int Myclass::si=sci; // "static". knows about class  
// scope. can access private area.  
  
int main() {  
    cout << Myclass::si << endl; //can access without object  
    Myclass mcl, mc2;  
    mcl.si = 3; // can access with object  
    cout << mc2.si << endl; // same value all objects  
}
```

A **static** data member is one copy accessible independently of class objects.

The class controls the member's definition space and access, but the member is not a class subobject.

With the keyword “**static**”, you declare member data static, to share one copy among all code independently of class objects.

- You *declare* the member inside the class definition.
- You *define* the member outside the class definition.

The member data is “global” within the class, but safer than a truly global variable, as the class can “hide” it within a private or protected region.

This example declares a private static constant integer and initializes it. The example also declares a public static integer variable, which because it is not constant, can be initialized only outside the class definition but within the enclosing namespace. Within the enclosing namespace, it defines both variables, and initializes the non-constant variable. Initialization is performed within the class scope, so can access private data without scope qualification.

The declaration of a static data member is not a definition. You must also define the static data member outside the class definition but within the namespace enclosing the class definition. Upon its definition, the static data member exists regardless of the existence of any objects of the class. If you initialize the static data member upon its definition, the initializer expression is in the scope of the class. You can alternatively initialize a const static data member of integral or enumeration types upon its declaration within the class definition, upon which the initialized member may then appear in other constant integral expressions within the class definition.

What Are Static Class Member Functions?



A **static** member function is accessible independently of class objects.

Advantage:

- You can call it before any objects of the class exist.

Disadvantage:

- Has no implicit **this** parameter.
 - Can directly access only static data.

```
class Myclass {  
    private: static const int sci = 5;  
    public: static int si;  
    static void print_value();  
};  
  
const int Myclass::sci;  
int Myclass::si=sci;  
void Myclass::print_value() { ... }  
  
int main() {  
    // no object needed  
    Myclass::print_value();  
    Myclass::si = 3;  
    Myclass::print_value();  
}
```



A **static** member function is accessible independently of class objects.

With the keyword “**static**”, you declare member functions static, to permit calling the function without an object.

The static member function has no knowledge of the current object, so can access only the static members.

The example “main” function, calls the static “print value” function, sets the static integer member variable and then again calls the static “print value” function.

Module Summary

In this module, you

- Achieved the best compromise between program reliability and runtime performance
 - By appropriately choosing function argument passing by value, pointer, or reference
- Simplified the application interface to the class
 - By defining overloaded functions
- Defined data and functions common to all class objects
 - By defining static members

This training module discussed:

- Function Argument Passing
- Function Overloading
- Global and Local Scope
- Static Class Members

108 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Achieve the best compromise between program reliability and runtime performance, which you do by appropriately choosing function argument passing by value, pointer, or reference;
- Simplify the application interface to the class, which you do by defining overloaded functions; and
- Define data and functions that are common to all objects of the class, which you do by declaring them static.

To help you to achieve your objective, this training module discussed:

- Function Argument Passing;
- Function Overloading;
- Global and Local Scope; and
- Static Class Members.

Quiz: Functions



How can you prevent the receiving function from modifying an object for which a pointer or reference is passed?



Suggest why the compiler does not consider the return type when matching a function call to an overloaded function declaration.



What does C++ “scope” mean?



Declaring a static data member in the class definition does not define the member. Where and how do you define the member so that it actually exists.

Please pause here to consider these questions.



Lab

Lab 7-1 Defining Class Member Functions

Objective:

- To pass function arguments by reference
- To overload functions
- To use static data members

For this lab, you:

- Overload class functions to set variable values
- Use a static data member to track the number of class objects
- Use a static member function to get that number

110 © Cadence Design Systems, Inc. All rights reserved.

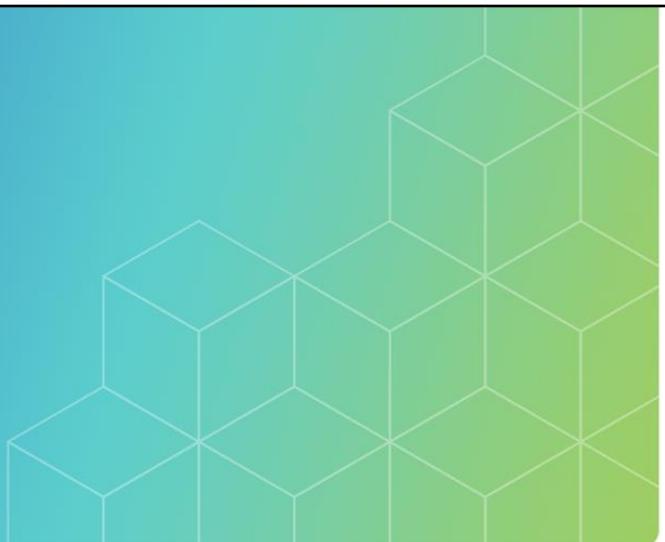
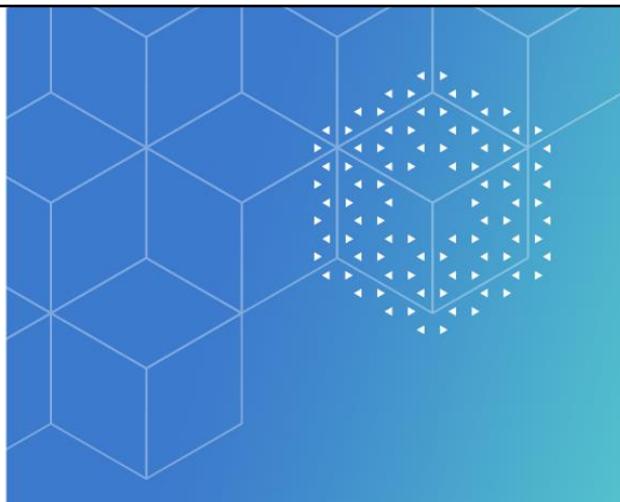


Your objectives for this lab are to:

- Pass function arguments by reference;
- Overload functions; and
- Use static data members;

For this lab, you:

- Overload class functions to set variable values;
- Use a static data member to track the number of class objects; and
- Use a static member function to get that number.



Module 8

Type Conversion

cadence®

This training module briefly examines C++ type conversion, both implicit and explicit.

Module Objectives

In this module, you

- Describe and explain implicit type conversions automatically done by the compiler
- Utilize an expression as a type other than its natural type by explicitly converting its type

This training module discusses:

- Type Checking
- Implicit Type Conversion
- Explicit Type Conversion

112 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Describe and explain implicit type conversions automatically done by the compiler; and
- Utilize an expression as a type other than its natural type by explicitly converting its type.

To help you to achieve your objective, this training module discusses:

- Type Checking;
- Implicit Type Conversion; and
- Explicit Type Conversion.

Quick Reference Guide: C++ Types

Fundamental Types	Width	Compound Types	Example Definition	CV Qualifiers	Description
<code>bool</code>	1	array	<code>using ia_t = int[42];</code>		
<code>[signed unsigned] char</code>	8+	enumeration	<code>enum E {val1,val2};</code>		
<code>[unsigned] short int</code>	16+	class	<code>class C {members};</code>		
<code>[unsigned] int</code>	16+	function	<code>void func() {}</code>		
<code>[unsigned] long int</code>	32+	pointer to non-static class member	<code>int C::* p = &C::i;</code>		
<code>[unsigned] long long int</code>	64+	pointer to others	<code>int i; int *p = &i;</code>		
<code>float</code>	$32=1+8+23^\dagger$	reference	<code>int i; int &j = i;</code>		
<code>double</code>	$64=1+11+52$	union	<code>union U {members};</code>		
<code>long double</code>	$79=1+15+63$				
<code>nullptr_t</code>	<code>sizeof(void*)</code>				
<code>void</code>	0				

[†]IEEE Std. 754 sign + exponent + fraction

113 © Cadence Design Systems, Inc. All rights reserved.



The C++ fundamental types are:

- The Boolean type, having values true and false;
- The signed or unsigned integer types, which with the Boolean type, comprise the integral types;
- The floating-point types, which with the integral types, comprise the arithmetic types;
- The null pointer type; and
- The **void** type.

The standard prescribes a minimum width for integer types.

The standard does not prescribe a minimum width for floating-point types, other than to require that the double precision be no less than the float precision, and the long double precision be no less than the double precision. Almost all implementations comply with the IEEE Std. 754.

The value range for arithmetic types is revealed by the implementation’s “limits” (<limits>) library.

A type can be a compound of other types.

- An array;
- An enumeration;
- A class;
- A function;
- A pointer to a non-static class member;
- A pointer to something other than a non-static class member;
- A reference, which can be a left-value (*lvalue*) reference or a right-value (*rvalue*) reference; and
- A union.

The object type includes any “constant-volatile” qualifiers.

- An object qualified as “constant” (**const**) is one that the program cannot change.
- An object qualified as “**volatile**” is one that may change unbeknownst to the program, for example, by hardware.

What Is Type Checking?



Type Checking determines whether a construct's type matches the expected type in the context in which the construct is used.

C++ is a relatively (compared to C) strongly-typed language.

- Every name and every expression has a type.
- The compiler statically checks the type of each expression in its usage context.
- The runtime environment dynamically checks that a base-type reference converted to an extended-type reference actually references an object of that extended type.

Type checking makes the program more likely to execute as expected.

Strong Static Checking

ADA, Cobol, Fortran, C#, F#, Java, Haskell, Rust, Scala

Strong Dynamic Checking

Clojure, Elixir, Erlang, Groovy, Lisp, Majik, MATLAB, Python, Ruby

Weak Static Checking

C, C++, C#, Modula-2, Pascal, VB

Weak Dynamic Checking

JavaScript, PHP, Perl, VB

Some languages offer multiple type-checking modes.

114 © Cadence Design Systems, Inc. All rights reserved.



Type Checking determines whether a construct's type matches the expected type in the context in which the construct is used.

C++ is a relatively (compared to C) strongly-typed language:

- Every name and every expression has a type;
- The compiler *statically* checks the type of each expression in its usage context; and
- The runtime environment *dynamically* checks that a base-type reference converted to an extended-type reference actually references an object of that extended type.

Type checking makes the program more likely to execute as expected.

The diagram attempts to categorize some languages according to their type safety, and whether the checking is performed statically or dynamically. Some languages offer multiple type-checking modes. The categorization is somewhat subjective. Relative type safety, is a subject of protracted discussion, that we shall not here further engage in.

Type safety strength is on a continuum. No programming language type safety is absolutely loose or absolutely strict. An example of weak type safety might be JavaScript. An example of strong type safety might be Ada. C is somewhere between, and C++ is somewhat more strict.

Peter van der Linden's in "Expert C Programming" states: "To this day, many C programmers believe that 'strong typing' just means pounding extra hard on the keyboard."

Some things you get "get away with" in C but not in C++ include:

- Types considered "compatible" in C, such as identically-defined structs having different tag names.
- Assigning the value of a pointer-to-void type to some other kind of pointer without an explicit type cast.
- Declaring a new type within an expression or in a parameter list or function return type.
- Implicit int type specifications, for example in function parameters or return type.

What Is Type Conversion?



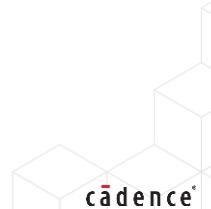
Type Conversion is converting an expression's data type from one type to another.

Type conversion can be:

- *implicit* – (AKA coercion)
 - Performed automatically by the compiler
 - According to well-defined rules
 - Mostly preserves expression value
- *explicit* – (AKA casting)
 - Expressed by code and statically performed or dynamically executed.
 - Can force reinterpretation of expression's bit pattern.

Integer Conversion Rank
for Integral Promotions
in Standard Conversions

long long int
long int
int
short int
char
bool



Type Conversion is converting an expression's data type from one type to another.

The most important type conversion aspect is whether it is done implicitly or explicitly.

- The compiler statically and quietly performs implicit conversions according to documented rules that mostly preserve the expression's value, though obviously converting from a higher-resolution type to a lower-resolution type may lose some value resolution. Implicit conversions are also termed *coercions*.
- The user can code explicit conversions, that depending on the conversion, either the compiler statically performs, or the runtime environment dynamically executes. Depending on the conversion, this can reinterpret a value's bit pattern as some other type entirely, without modifying the value in any way. Explicit conversions are also termed *casting*.

Standard conversions are defined implicit conversions. One step of the standard conversion sequence may be integral promotions. Integral promotions promote lower-ranking integral expressions to higher-ranking integral expressions, without data loss.

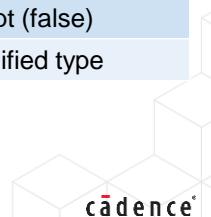
What Are the Standard Implicit Conversions?



A standard *implicit conversion* is one of the following conversions.

The compiler may choose in order at most one from each category.

Order	Standard Conversion	Example
1 st	conversion: lvalue-to-rvalue, array-to-pointer, function-to-pointer	A[]-to-A*, f() to (*f)()
2 nd	promotion: integral, floating-point; conversion: integral, floating-point, floating/integral, pointer, pointer-to-member, Boolean;	int-to-long, float-to-double long-to-int, double-to-float, float-to-int, int-to-float, derived*-to-base*, base::*-to-derived::*, int-to-bool, bool-to-int
3 rd	conversion: function pointer	(*fp)() noexcept (true) to (*fp)() noexcept (false)
4 th	conversion: qualification	less cv-qualified type to a more cv-qualified type



A standard *implicit conversion* is one of the following conversions.

The compiler may choose in order at most one from each category.

- The 1st potential implicit conversion is either a left-value (*lvalue*) to right-value (*rvalue*) conversion, an array-to-pointer conversion, or a function-to-pointer conversion.
 - A generalized-left-value (*glvalue*) of a type that is not a function or array, can be implicitly converted to a pure-right-value (*prvalue*) of that type. The result is, for a fundamental type, the value, and for a class type, a copy.
 - A left-value or right-value of type “array of some element type” can be implicitly converted to a pure-right-value of type “pointer to that element type”. It points to the first array element.
 - A left value of some function type can be implicitly converted to a pure-right-value of type “pointer to a function of that type”.
- The 2nd potential implicit conversion is numeric promotions and numeric and pointer conversions. The numeric promotions and conversions explain themselves. A pure-right-value of type “pointer to derived class” can be implicitly converted to a pure-right-value of type “pointer to the base class”. A pure-right-value of type “pointer to member of the base class” can be implicitly converted to a pure-right-value of type “pointer to member of derived class”.
- The 3rd potential implicit conversion is function pointer conversion. A pure-right-value of type “pointer to function that does not throw exceptions” can be implicitly converted to a pure-right-value of type “pointer to function that may throw exceptions”, but not the reverse.
- The 4th potential conversion is from a type that is not or only somewhat constant-volatile-qualified to a type that is more constant-volatile-qualified. For example, the value of a pointer to an integer can be assigned to a pointer of type “pointer to a *constant* integer”.

Implicit Type Conversion Versus Explicit Type Conversion



Implicit Type Conversion (AKA coercion) is performed automatically by the compiler.



Explicit Type Conversion (AKA casting) is specified by (and for classes defined by) the programmer.

A source expression type is quietly converted to a target expression type.

Loss of information may occur, for example:

- **long** to **int** truncates integer part
- **float** to **int** removes fractional part
- **double** to **float** can lose some range and/or precision

Explicit type conversion can be expressed by:

- The cast notation (looks like C)
 - (T) expression
- The functional notation
 - T (expression_list)
- Type conversion operators
 - cast_operator<T> (expression)



The C++ compiler automatically performs conversions between the built-in types. It attempts an ordered sequence of such conversions and issues an error if unsuccessful. These conversions can lose information without warning.

The programmer can explicitly convert between types by using the legacy C syntax or a C++ functional notation or type conversion operation.

For implicit type conversion:

- It 1st tries no more than one of the following conversions:
 - from lvalue to rvalue
 - from array to pointer
 - from function to pointer
- It 2nd tries no more than one of the following conversions:
 - integral promotions such as short to int
 - floating point promotion such as float to double
 - integral conversions such as int to short
 - floating point conversions such as double to float
 - floating-integral conversions such as float to int and int to float
 - pointer conversions such as pointer-to-subclass to pointer-to-superclass
 - pointer to member conversions such as pointer-to-superclass-member to pointer-to-subclass-member
 - Boolean conversions such as int to bool
- It 3rd may try one function pointer conversion, such as pointer-to-noexcept-function to pointer-to-function.
- It 4th may try one qualification conversion such as pointer-to-object to pointer-to-const/volatile-object

Explicit Type Conversion Using Type Conversion Operators



```
// Cast between related types
static_cast<T>(v)

// Cast between unrelated types
reinterpret_cast<T>(v)

// Cast reference-to-const-object
// to reference-to-object
const_cast<T>(v)

// Cast during run time
dynamic_cast<T>(v)
```

```
int_var = static_cast<int> (float_var);

int_ptr = reinterpret_cast<int*> (struct_ptr);

int_ptr = const_cast<int*> (const_int_ptr);

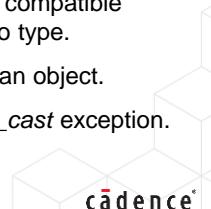
subclass_ptr = dynamic_cast<subclass*>(superclass_ptr);
```

For **reinterpret_cast**, the standard guarantees only that a cast to and from a type of sufficient size and compatible alignment preserves the cast value, and does not guarantee how the value is represented as the cast-to type.

For **const_cast**, the standard does not define the result of an attempt to “cast away” the constness of an object.

For **dynamic_cast**, failure to cast a pointer returns *nullptr*, and failure to cast a reference throws a *bad_cast* exception.

118 © Cadence Design Systems, Inc. All rights reserved.



The operator “static cast” (**static_cast**) casts between related types, such as among integral types and between integral and floating-point types and between pointers within the same class hierarchy.

The operator “reinterpret cast” (**reinterpret_cast**) casts between unrelated types, such as between pointer types and between pointers and integers. The standard guarantees only that if the target type has not lost any precision, you can reverse the cast and get the same value. How the value is represented in the cast-to type is dependent upon the implementation and thus not portable.

The operator “constant cast” (**const_cast**) “casts away” the constness of a pointer or reference type. The C++ standard does not require the operator to “cast away” the constness of an *object*. Whether your compiler does is dependent upon the implementation and thus not portable.

The operator “dynamic cast” (**dynamic_cast**) performs the cast during run time. Use this cast when the cast-to type cannot be statically determined. Failure to cast a pointer returns a null pointer. Failure to cast a reference throws a “bad cast” (*bad_cast*) exception. This operator is mostly used to cast a pointer to a virtual base class to a pointer to one of its subclasses.

Explicit Type Conversion Using Cast and Functional Notation



```
// Prefix form
// (like C)
(T) cast_expression
my_int = (int) my_float;
```



```
// Postfix Functional form
// (like a constructor call)
T (expression_list)
my_int = int (my_float);
```

Prefix form (like C):

- The cast notation will attempt the conversion sequence: `[[static_cast | reinterpret_cast]] ; [const_cast]`
 - If that does not work for you, then use the type conversion operators directly.
- If the *target* type is a class:
 - The conversion is performed by a user-defined *conversion constructor*.
- If the *expression* type is a class:
 - The conversion is performed by a user-defined *cast operator*.

Postfix functional form (like a constructor call):

- If the *expression_list* is a single expression:
 - The conversion is equivalent to that done by the cast notation.
- If the *expression_list* is more than one expression:
 - The conversion is performed by a user-defined *initialization constructor*.



The prefix form is equivalent to an optional “static cast” (`static_cast`) or “reinterpret cast” (`reinterpret_cast`), followed by an optional “constant cast” (`const_cast`). The compiler attempts an ordered selection of these combinations. For a non-dynamic cast, you can almost always just use the prefix form and let the compiler figure it out. If the *target* type is a class, the compiler will look for a suitable *conversion constructor*. If the *expression* type is a class, the compiler will look for a suitable overloaded *cast operator*.

For the functional form, if the expression list is a single expression, then the conversion is equivalent to that done by the cast notation, and if the expression list is more than one expression, the target type must be a class and a constructor must exist with a parameter list matching the provided expression list.

The prefix form of explicit type casting performs the first of the following casts that applies:

1st – `const_cast`

2nd – `static_cast`

3rd – `static_cast` followed by a `const_cast`

4th – `reinterpret_cast`

5th – `reinterpret_cast` followed by a `const_cast`

Module Summary

In this module, you

- Described and explain implicit type conversions automatically done by the compiler
- Utilized an expression as a type other than its natural type by explicitly converting its type

This training module discussed:

- Type Checking
- Implicit Type Conversion
- Explicit Type Conversion

120 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Describe and explain implicit type conversions automatically done by the compiler; and
- Utilize an expression as a type other than its natural type by explicitly converting its type.

To help you to achieve your objective, this training module discussed:

- Type Checking;
- Implicit Type Conversion; and
- Explicit Type Conversion.

Quiz: Type Conversion



The compiler does implicit type conversions, for example the standard conversion from **int** to **float** if you pass an integer value to a function expecting a floating-point value. Suggest how a standard conversion can cause an algorithm to produce wildly inaccurate results.



The prefix-form cast operation is equivalent to an optional static cast or reinterpret cast, followed by an optional const cast. Why might you want to instead use these C++ cast operators explicitly?

Please pause here to consider these questions.



Lab

Lab 8-1 Exploring Cast Operations

Objective:

- To analyze the C++ cast operators

For this lab, you:

- Experiment with the **reinterpret_cast** operator
 - Cast a pointer type to an integer type that is the same size or larger than the pointer type
 - Cast a pointer type to an integer type that is smaller than the pointer type
 - See that casting to the smaller integer type potentially loses the pointer value

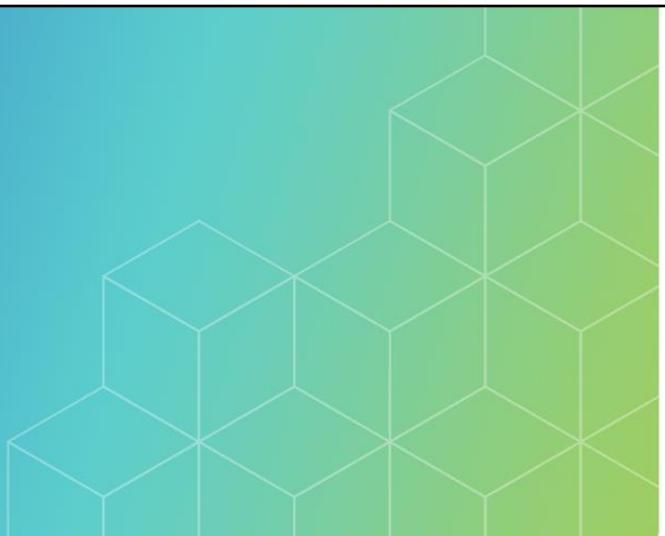
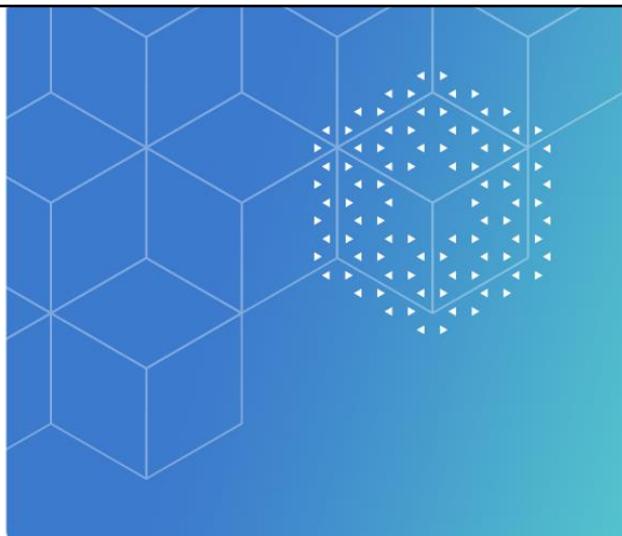
122 © Cadence Design Systems, Inc. All rights reserved.



Your objective for this lab is to explore the C++ cast operators.

For this lab, you experiment with the **reinterpret_cast** operator:

- You cast a pointer type to an integer type that is the same size or larger than the pointer type; and
- You cast a pointer type to an integer type that is smaller than the pointer type.
 - When you do this, you see that casting to the smaller integer type potentially loses the pointer value.



Module 9

Operator Overloading

cadence®

This training module examines operator overloading.

Module Objectives

In this module, you

- Define overloaded operators
 - To provide an intuitive interface to class operations

This training module discusses:

- Introduction to Operator Overloading
- Guidelines for Operator Overloading
- Friend Operators

124 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Provide an intuitive interface to class operations, by defining overloaded operators.

To help you to achieve your objective, this training module discusses:

- An Introduction to Operator Overloading;
- Guidelines for Operator Overloading; and
- Friend Operators.

What Is Operator Overloading?



Operating Overloading is defining functions to define or redefine custom behavior for the operators when one or more operands are objects of your class type.

For user-defined class types, re-purposing existing operators (+, -, etc) can make application code more readable.

- For example, using “+” to concatenate two strings may be more obvious than a call to function `strcat`.

Best Practice: Make overloads act intuitively.

- For example, don’t overload “+” to do multiplication.

Examples

- (`Class_obj + other`)
 - Implement as class member function.
 - `const Class& operator+(const Tother&)`
 - Can also directly call the class member function.
 - `Class_obj.operator+(other)`
- (`other + Class_obj`)
 - Implement as non-member function.
 - `const Class& operator+(const Tother&, const Class&)`
 - Can also directly call the non-member function.
 - `operator+(other, Class_obj)`



For your classes, it may be convenient and produce less verbose and thus more readable code, to represent some operations on objects by using existing operators. For example, a user-defined String class might use the plus (+) operator to represent a string concatenation operation. Although you can define any behavior, to define non-intuitive behavior is considered poor practice.

For expressions where the left or only operand is a class object, the compiler will attempt to find an overloaded operator. For these expressions, the most efficient implementation is a member function. For member functions, the left or only operand is the current object and any right operand is an argument to the function. Some overloaded operators must be member functions.

For binary expressions where the left operand is not a class object, the compiler will first attempt to convert that operand to an object of the same class as the right operand. Failing that, the compiler will attempt to find an overloaded operator. For these expressions, the only implementation is a non-member function. For non-member functions, both operands are arguments to the function. Non-member functions can access only the public members of the class unless the class has declared them to be “friend” functions.

Some overloaded operators can be implemented as either member functions or non-member functions.

Example: Assignment and Equality Operator Overload Code

```
#include <cstring>
using namespace std;
class TransAttribute {
public:
    TransAttribute () { initialize("unknown", 0); }
    TransAttribute ( const TransAttribute& rhs ) { initialize(rhs.name, rhs.value); }
    TransAttribute ( const char* name_, int value_=0 ) { initialize(name_, value_); }
    ~TransAttribute () { delete [] name; }
    void print();
    TransAttribute& operator= ( const TransAttribute& );
    bool operator== ( const TransAttribute& ) const;
private:
    char* name;
    int value;
    void initialize ( const char*, int );
};

// Other functions
...

TransAttribute& TransAttribute::operator= (const TransAttribute& rhs) {
    if ( this != &rhs ) {
        delete [] name;
        initialize ( rhs.name, rhs.value );
    }
    return *this;
}

bool TransAttribute::operator== ( const TransAttribute& rhs ) const
{ return !strcmp(name, rhs.name) && value == rhs.value; }
```

Don't do anything if rhs already is lhs.
Else return reference to support assignment chaining.

```
#include "TransAttribute.h"
int main() {
    TransAttribute t1;
    TransAttribute t2 ("read", 44);
    TransAttribute t3 ( t2 );
    t1.print(); // unknown / 0
    t2.print(); // read / 44
    t3.print(); // read / 44
    if (t3 == t2) {
        t1 = (t2 = TransAttribute("write", 66));
        t1.print(); // write / 66
        t2.print(); // write / 66
        (t3 = t2) = TransAttribute("read", 88);
        t2.print(); // write / 66
        t3.print(); // read / 88
    }
    return 0;
}
```

Here, we merely compare all attributes.
But customize as needed for your class.



126 © Cadence Design Systems, Inc. All rights reserved.

The example adds to the Transaction Attribute class an assignment (“=”) operator overload and an equality (“==”) operator overload.

- The assignment operator overload returns a reference to the current object to support chaining assignment operations.
- The equality operator overload simply determines whether the attribute values of the two objects are identical.

The application code:

- Constructs three transaction objects, utilizing the default, initialization, and copy constructors;
- Confirms the equality of two transactions; and
- By expression grouping, confirms that assignment returns a non-constant transaction reference.

Here we return a non-constant reference to support arcane coding such as $(x=y)=z$

Example: Member Addition Operator Overload Code

```
#include <iostream>
#include <cstring>
using namespace std;
class String {
public:
    String() {};
    String(const char *);
    void print() const;
    String operator+ (const char* ) const;
    String operator+ (const String&) const { return *this + rhs.str; }
private: char *str;
};

// Other functions
...

// Member operator+ overload
String String::operator+ (const char* rhs) const {
    String result; // new String
    result.str = new char[strlen(str) + strlen(rhs) + 1];
    strcpy(result.str, str); // copy left operand string
    strcat(result.str, rhs); // concatenate right operand string
    return result; // return result
}
```

```
#include "String.h"
int main() {
    const String s1("We ");
    const String s2("love ");
    (s1 + (s2 + "C++")).print(); //We love C++
    ((s1 + s2) + "C++").print(); //We love C+
    return 0;
}
```

Make new String.
Make new c-string.
Copy LHS c-string to new c-string.
Concatenate RHS c-string to new c-string.
Return new String.

127 © Cadence Design Systems, Inc. All rights reserved.

This example defines for the String class two plus operator (operator+) member overloads to add to a left operand String object a right operand String object or C-string.

The plus operator member overload:

- Declares a new String object result;
- Copies to the result C-string the left String object operand's C-string;
- Concatenates to the result C-string the right operand C-string; and
- Returns the result object.

The application code:

- Declares and initializes two strings;
- Tests that both plus operator overloads work; and
- Tests that the result String is non-constant.

Befriending a Non-Member Function to Allow Access



For expressions whose left operand is not an object of the class, for example:

```
(42 + Class_object) // operator+ function cannot be a member of the class
```

The operator can implemented only as a non-member function, which by default cannot access private member data.

Solution 1:

- Implement the operator as a non-member function that calls member functions to access the data.
- These extra calls may degrade performance!

Solution 2:

- Declare the non-member function a friend of the class.
- Class friends can directly access all member data – even that declared private or protected.

Befriending non-member functions:

```
friend function_prototype ; // the one function is a friend  
friend class ClassName ; // all the class functions are friends
```



For binary expressions, the left operand is not always of a class type. To accommodate those expressions, you must write a non-member operator overload. However, a non-member function cannot normally access the non-public members of the class. What to do?

One solution is for the non-member function to call a member function to perform the operation. Here, for the example expression, the non-member operator could use a public class interface function to convert the class object to an integer. This has a negative performance impact, that for this trivial example is insignificant, but might be significant for a more complex class.

Another solution, that discards the data encapsulation concept, so is arguably not a “better” solution, is for the class to befriend the non-member function so that the non-member function can directly access the class object’s private data.

Example: Non-Member Friend Addition Operator Overload Code

```
#include <iostream>
#include <cstring>
using namespace std;
class String {
public:
    String();
    String(const char *);
    void print() const;
    String operator+ (const char*) const;
    String operator+ (const String&) const { return *this + rhs.str; }
    friend String operator+ (const char*, const String&);
private: char *str;
};

// Other functions
...

// Non-member operator+ overload
String operator+ (const char* lhs, const String& rhs) {
    String result; // new String
    result.str = new char[strlen(lhs) + strlen(rhs.str) + 1];
    strcpy(result.str, lhs); // copy left operand string
    strcat(result.str, rhs.str); // concatenate right operand string
    return result; // return result
}
```

```
#include "String.h"
int main() {
    const String s1("love ");
    const String s2("C++ ");
    ("We " + (s1 + s2)).print(); // We love C++
    ((We " + s1) + s2).print(); // We love C++
    return 0;
}
```

Make new String.
 Make new c-string.
 Copy LHS c-string to new c-string.
 Concatenate RHS c-string to new c-string.
 Return new String.

129 © Cadence Design Systems, Inc. All rights reserved.



This example defines for the String class a plus operator (operator+) non-member overload to add to a left operand C-string, a right operand String object.

The plus operator member overload:

- Declares a new String object result;
- Copies to the result C-string the left operand C-string;
- Concatenates to the result C-string the right String object operand's C-string; and
- Returns the result object.

The application code:

- Declares and initializes two strings;
- Tests that the non-member plus operator overload works; and
- Tests that the result String is non-constant.

Example: Addition Assignment Operator Overload Code

```
#include <iostream>
#include <cstring>
using namespace std;
class String {
public:
    String(const char*); // constructor
    void print() const; // print function
    String& operator+=(const String&); // plus assignment operator
private: char *str;
};

// Other functions
...

// operator += overload
String& String::operator+=(const String& rhs) {
    char *temp = new char[strlen(str) + strlen(rhs.str) + 1];
    strcpy(temp, str); // copy left operand string
    strcat(temp, rhs.str); // concatenate right operand string
    delete [] str; // free left operand string memory
    str = temp; // point to new left operand memory
    return *this; // return left operand by reference
}

#include "String.h"
int main() {
    String s1("We ");
    String s2("love ");
    String s3("C++ ");
    s1 += (s2 += s3); s1.print(); // We love C++
    (s1 += s2) += s3; s1.print(); // We love C++ love C++ C++
    return 0;
}
```

Return reference to support chaining.
Making non-const supports (s1+=s2)+=s3

130 © Cadence Design Systems, Inc. All rights reserved.

cadence

This example defines for the String class a plus assignment operator (operator+=) overload, to add to a left operand String object, a right operand String object.

The plus assignment operator overload:

- Constructs a new temporary C-string;
- Copies to the temporary C-string the left String object operand's C-string;
- Concatenates to the temporary C-string the right String object operand's C-string;
- Deletes the left String object operand's C-string;
- Replaces the left String object operand's C-string with the temporary C-string; and
- Returns a non-constant reference to the left String object operand.

The application code:

- Declares and initializes three strings;
- Tests that the plus assignment operator overload works; and
- Tests that the returned String reference is non-constant.

Example: Auto Increment/Decrement Operator Overload Code

```
#include <cstring>
using namespace std;
class TransAttribute {
public:
    TransAttribute () {initialize("unknown", 0); }
    TransAttribute ( const TransAttribute& rhs ) {initialize(rhs.name, rhs.value); }
    TransAttribute ( const char* name_, int value_=0 ) {initialize( name_, value_); }
    ~TransAttribute () { delete [] name; }
    void print();
    TransAttribute& operator++ ();
    TransAttribute& operator-- ();
    TransAttribute& operator++ (int dummy);
    TransAttribute& operator-- (int dummy);
private:
    char* name;
    int value;
    void initialize ( const char*, int );
};

// Other functions
...

TransAttribute& TransAttribute::operator++ () {
    TransAttribute& TransAttribute::operator-- () {
    TransAttribute TransAttribute::operator++ (int dummy) {TransAttribute t=*this; value++; return t;}
    TransAttribute TransAttribute::operator-- (int dummy) {TransAttribute t=*this; value--; return t;}
}

```

**prefix if no extra parameter.
postfix if extra int parameter.**

postfix returns new object by value.

```
#include "TransAttribute.h"
int main() {
    TransAttribute t("read", 55);
    ((++t)--).print(); // 56
    t.print(); // 55
    (++(t--)).print(); // 56
    t.print(); // 54
    return 0;
}
```

131 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The example defines for the Transaction Attribute class the operators auto-increment and auto-decrement.

The signature with no extra parameter indicates a prefix operation.

The signature with an extra integer parameter indicates a postfix operation.

Postfix operations must return a new object having the pre-operation value.

The application code:

- Constructs a transaction object by utilizing the initialization constructor; and
- By expression grouping, confirms pre-increment and post-decrement operations, and that the returned reference or object is non-constant.

The prefix overloads return non-constant references to permit coding expressions such as `--b+=c`

Using a postfix operator passes the value 0 to the dummy parameter. When called as a function you can pass any integer value.

Example: Cast Operator Overload Code

```
#include <cstring>
using namespace std;
class TransAttribute {
public:
    TransAttribute () { initialize("unknown", 0); }
    TransAttribute ( const TransAttribute& rhs ) { initialize(rhs.name, rhs.value); }
    TransAttribute ( const char* name_, int value_=0 ) { initialize(name_, value_); }
    ~TransAttribute () { delete [] name; }
    void print();
    explicit C++11 (const-expr) C++20 operator int() const; // declaration
private:
    char* name;
    int value;
    void initialize ( const char*, int );
};

// Other functions
...
// operator int() overload
TransAttribute::operator int() const { return value; } // definition
```

```
#include "TransAttribute.h"
int main() {
    TransAttribute t("read", 55);
    t.print(); // read / 55
    cout << (int)t << endl; // 55
    return 0;
}
```

132 © Cadence Design Systems, Inc. All rights reserved.

You can define a member cast operator to return pretty much anything that you want to except an array or a function type.

The signature does not specify a return type and does not take arguments.

You can specify the function explicit to prevent the compiler from using your operator for implicit conversions.

As of C++ 20, you can make that explicit specifier conditional.

This example casts a Transaction Attribute reference to an integer.

The application code confirms the cast operator.

Example: Function Call Operator Overload Code

```
#include <cstring>
using namespace std;
class TransAttribute {
public:
    TransAttribute () { initialize("unknown", 0); }
    TransAttribute ( const TransAttribute& rhs ) { initialize(rhs.name, rhs.value); }
    TransAttribute ( const char* name_, int value_=0 ) { initialize(name_, value_); }
    ~TransAttribute () { delete [] name; }
    void print();
    void operator() ( const char*, int );
private:
    char* name; declaration
    int value;
    void initialize ( const char*, int );
};

// Other functions
...

// operator() overload -- this one updates data and returns nothing
void TransAttribute::operator () ( const char* name_, int value_ ) {
    delete [] name;
    initialize ( name_, value_ ); definition
}
```

```
#include "TransAttribute.h"
int main() {
    TransAttribute t ("read", 55);
    t.print(); // read / 55
    t ("write", 66);
    t.print(); // read / 66
    return 0;
}
```

133 © Cadence Design Systems, Inc. All rights reserved.



You can define a function call operator to return anything, and do anything, that a member function can do.

The signature can specify any return type, and can take any arguments that a member function can take.

Defining at least one function call operator, makes the class a function object (also known as a functor).

Unlike a function, a function object maintains its state between calls, without needing a static or global variable.

The example defines a function call operator that updates the Transaction Attribute data.

The application code confirms the function call operator.

Example: Subscript Operator Overload Code

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

using namespace std;
class String {
public:
    String(const char *ptr);
    void print() const;
    const char& operator[](const int&) const; // read rhs
    char& operator[](const int&); // write lhs
private: char *str;
};

// Other functions
...

// operator[] overload
const char& String::operator[](const int& n) const { return str[n]; } // read rhs
char& String::operator[](const int& n) { return str[n]; } // write lhs
```

```
#include "String.h"
int main()
{
    String s("Don't we love C++ much!");
    s.print(); // Don't we love C++ much!
    s[6] = s[21];
    s.print(); // Don't he love C++ much!
    return 0;
}
```

Constant character reference for read.
Non-constant character reference for write.

134 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The example adds to the String class two subscript (“[]”) operator overloads each returning a single string character.

- To read the subscripted element the code returns a constant (**const**) reference.
- To write the subscripted element the code returns a non-constant reference.

The application code:

- Declares and initializes a string;
- Prints the string;
- Reads the character at position 21 and writes that value to the character at position 6; and
- Again prints the string.

Example: Member Access Operator Overload Code

```
// Design
#include "Module.h"
#include "Channel.h"
class D_c {
private:
    M_c M_i;
    C_c<int> C_i;
public:
    D_c() {M_i.bind(&C_i);}
    void test(int i)
    {M_i.send(i);}
};
```

```
// Module
#include "Port"
class M_c {
private:
    P_c <int> P_i;
public:
    void bind(C_c <int> *C_p)
    {P_i.bind(C_p);}
    void send(int i)
    {P_i->write(i);}
};
```

```
// Port
#include "Channel.h"
template <class T> class P_c {
private:
    C_c<T> *C_p;
public:
    void bind(C_c<T> *C_p_)
    {C_p = C_p_;}
    C_c<T>* operator ->()
    {return C_p;}
};
```

```
#include "Design.h"
int main()
{
    D_c d_i; // instance
    d_i.test(42);
    return 0;
}
```



```
// Channel
#include <iostream>
template <class T> class C_c {
private:
    T t;
public:
    void write(T t_)
    {t = t_;}
    std::cout << t << std::endl;
};
```

(P_i.operator->())->write(i);

135 © Cadence Design Systems, Inc. All rights reserved.



The example overloads the member access operator, to enable a module to access channel functions through the module’s port, with no knowledge of what channel instance the port is bound to.

The Design class instantiates a module and a specialized channel, and in its constructor, calls the module’s function “bind” to bind the module’s port to the channel, and implements the function “test” to call the module’s function “send”.

The Module class instantiates a specialized port, implements the function “bind” to call the port’s function “bind”, and implements the function “send” to call the channel’s function “write”, which it does by calling the port’s overloaded member access operator, to retrieve a pointer to the channel.

The Port class instantiates a channel pointer, implements the function “bind” to update that pointer, and overloads the member access operator, to return the pointer to the channel.



Quick Reference Guide: Operator Overload Prototypes

Category	Expression	Prototype
Assignment	<code>x = y</code>	<code>T_x& T_x::operator=(const T_y&);</code>
Pre Inc/Dec	<code>++x, --x</code>	<code>T_x& T_x::operator++();</code>
Post Inc/Dec	<code>x++, x--</code>	<code>T_x T_x::operator++(int);</code>
Conversion	<code>(T_y)x</code>	<code>T_x::operator T_y() const;</code>
Function call	<code>x(a, ...)</code>	<code>T T_x::operator()(const T_a&, ...);</code> Note that <code>T</code> can be <i>any type</i> . Defining one or more makes <code>T_x</code> a <i>Function Object (AKA functor)</i>
Subscripting	<code>x[y]</code>	<code>T& T_x::operator[](const T_y&);</code> <code>const T& T_x::operator[](const T_y&) const; // rvalue</code> <code>T& T_x::operator[](const T_y&); // lvalue</code> Note that <code>T</code> can be <i>any type</i> .
Member access	<code>x -> m</code>	<code>T_m* T_x::operator->();</code>

136 © Cadence Design Systems, Inc. All rights reserved.



Presented here are some of the more difficult and interesting operator overload prototypes. The list is not comprehensive.

- The assignment operator overload, returns a reference to the current object, because all expressions have a value, and the value of the assignment operation is the current object.
- The increment and decrement operators, operate on the current object.
 - The prefix form, returns a reference to the modified object.
 - The postfix form, returns a new object copied from the current object before its modification.
- The conversion operator overload, returns a new object of the converted-to type. Here, we do not allow it to modify the current object.
- The function call operator, can return any type, and can modify the current object. Here, we pass the arguments by constant (**const**) reference to allow passing a “right value” (*rvalue*) without copying it.
- The subscripting operator returns a reference to what is presumably an array element. You need separate forms for a “left value” (*lvalue*) and a “right value”. Here, we pass the index by constant reference to allow passing a “right value” without copying it.
- This member access operator, converts the left object to a pointer of the type of which the right operand is a member, then evaluates the expression as a normal member access operation.



Quick Reference Guide: Operator Overloading Guidelines

- You can overload only existing operators. You cannot create new operators.
- The standard lists the operators that you can overload.
- These operators are NOT on that list.
 - `:: . .* ?: sizeof typeid` (more...)
- These operators must be member functions.
 - `= [] () ->`
- For member functions, the first operand must be an object of the class.
- Other operators may be member functions or non-member functions.

137 © Cadence Design Systems, Inc. All rights reserved.



You can overload only existing operators. You cannot create new operators.

The standard lists the operators that you can overload. Assume that you cannot overload any other operator.

So you cannot, for example, overload the scope resolution operator (`::`), one class member access operator (`.`), one pointer to member operator (`.*`), the conditional operator (`?:`), the “size of” (`sizeof`) operator, or the “type ID” (`typeid`) operator.

Some operators must be member functions. The assignment operator (`=`), subscript operator (`[]`), cast operator (`()`), function call operator, and one class member access operator (`->`), must be member operators.

A member operator overload implies that the left or only operand is an object of the class.

Other operators, such as the prefix unary operators and the binary operators, may be either member functions or non-member functions.

The scope resolution operator (`::`) and member selection operators (`. .*`) take a name, instead of a value, as their right operand, so to permit user redefinition would be problematic. The language accommodates unary and binary overloads, but not ternary (`:?`) overloads. The language does accommodate user redefinition of named operators “`new`”, “`delete`”, and ‘`co_await`”, but currently no other named operators.

Recall that from C we have pointers to functions, and now with C++ we have pointers to objects. A pointer to a member is a variable that represents an offset to a member of the object’s class. Combining a pointer-to-object variable with a pointer-to-member variable provides maximum flexibility for selecting, for example, which function of the object to call. The pointer-to-member operator (`->*`), is a simple binary operator that you can overload and the overload can be a member or non-member function.

Module Summary

In this module, you

- Provided an intuitive interface to class operations
 - By defining overloaded operators

This training module discussed:

- Introduction to Operator Overloading
- Guidelines for Operator Overloading
- Friend Operators

138 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Provide an intuitive interface to class operations, by defining overloaded operators.

To help you to achieve your objective, this training module discussed:

- An Introduction to Operator Overloading;
- Guidelines for Operator Overloading; and
- Friend Operators.

Quiz: Operator Overloading



Why overload operators?



Suggest a reason for why some operators cannot be represented by non-member functions.



How does a class make its private members accessible by non-member functions?

139 © Cadence Design Systems, Inc. All rights reserved.



Please pause here to consider these questions.



Lab

Lab 9-1 Overloading Member Operators

Objective:

- To provide an intuitive interface to class operations, by defining overloaded operators

For this lab, you:

- Overload an assignment operator to accept a const counter reference argument to support:

```
counter2 = counter1;
```

- Overload an assignment operator to accept a count argument to support:

```
counter1 = count;
```

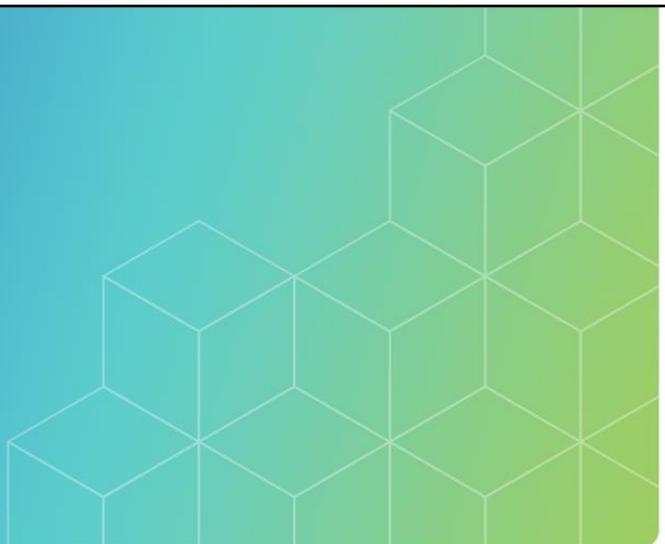
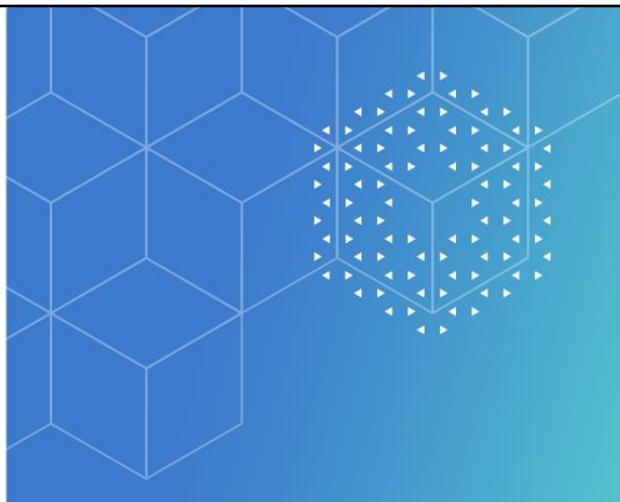
140 © Cadence Design Systems, Inc. All rights reserved.



Your objective for this lab is to provide an intuitive interface to class operations, which you do by overloading operators.

For this lab, you:

- To support assignment of one counter to another, overload an assignment operator to accept a const counter reference argument; and
- To support assignment of a value to a counter, overload an assignment operator to accept a count argument.



Module 10

Inheritance

cadence®

This training module examines class inheritance.

Module Objectives

In this module, you

- Define a class based upon an existing class definition, that inherits the base class attributes and normal functions, and optionally declares additional members

This training module discusses:

- Introduction to Class Inheritance
- Unified Modeling Language (UML) Diagrams
- Access Specifiers in Base Class Specifiers
- Multiple Inheritance and Virtual Base Classes

142 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define a class based upon an existing class definition, that inherits members of the base class and optionally declares additional members.

To help you to achieve your objective, this training module discusses:

- Introduction to Class Inheritance;
- Unified Modeling Language (UML) Diagrams;
- Access Specifiers in Base Class Specifiers; and
- Multiple Inheritance and Virtual Base Classes.

What Is Class Inheritance?



Class Inheritance is extending a class by defining a derived class that inherits members[†] of the base class and may override inherited members and/or add new members.

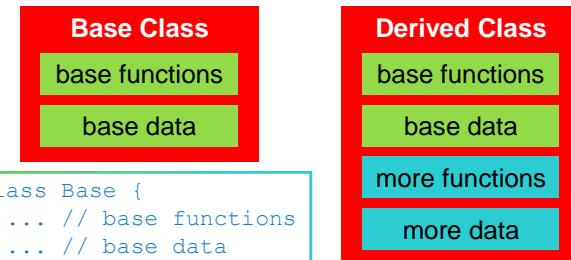
- [†] Does not inherit constructors and destructor.

Why inheritance?

- Reusability makes the software developer and user more productive.
- Encourages use of proven software (why re-invent the basic wheel?)

Terminology

- Original class is called the superclass (OO) or base class (C++).
- New class is called the subclass (OO) or derived class (C++).
- With single inheritance, the derived class inherits from a single base class.
- With multiple inheritance, the derived class inherits from multiple base classes.



```
class Base {  
    ... // base functions  
    ... // base data  
};
```

```
class Derived : Base {  
    ... // more functions  
    ... // more data  
};
```



Class Inheritance is extending a class by defining a derived class that inherits members of the base class and may override inherited members and may add new members.

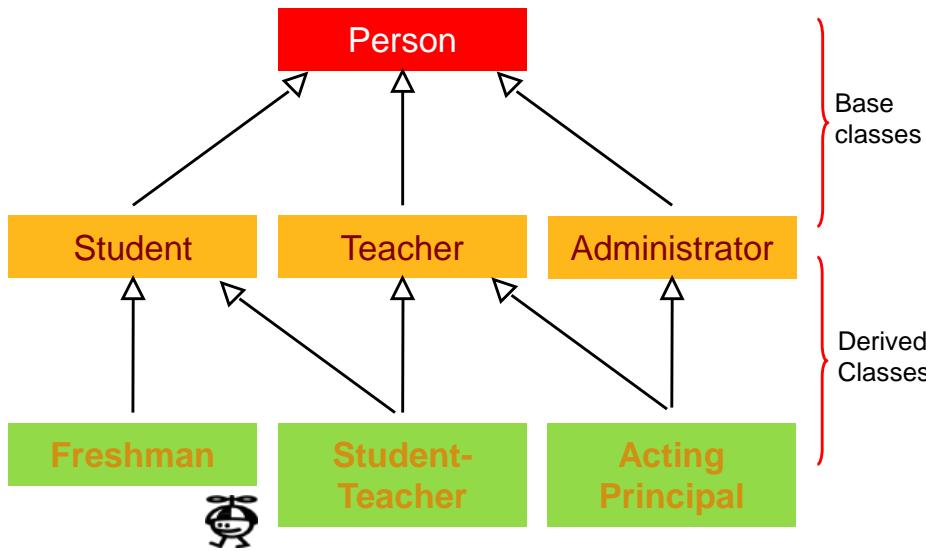
Inheritance is essential to the object-oriented paradigm. Each initial description or refinement of the class is defined in one place and easily reused by further refinements to the class.

In an inheritance hierarchy, an inherited class is a “superclass” in object-oriented terms and a “base” class in C++ terms, and the inheriting class is a “subclass” in object-oriented terms and a “derived” class in C++ terms. Base classes and derived classes are relative terms, as a derived class can be a base for a further derivation.

The C++ language permits multiple inheritance, where a derived class inherits from more than one base class.

You can explicitly inherit base class constructors via a using-declarator.

Example: Class Inheritance Hierarchy



144 © Cadence Design Systems, Inc. All rights reserved.



The class “Person” defines attributes and operations common to all persons. Here, the Person class is the base class for other classes. The Person class could also be itself a derived class, for example from a Mammal class.

The classes “Student”, “Teacher”, and “Administrator” are based upon the Person class. Each of those derived classes inherits all the Person attributes and behaviors, can override them, and can add attributes and behaviors. The Teacher class, for example, might have an behavior called “teach”, that only teachers, and not persons in general, can do.

The class “Freshman” derives from the Student class. A freshman is a special kind of student. Take a moment to imagine what attributes and behaviors a freshman might have that a student in general might not have.

The C++ programming language supports multiple inheritance. The class “Student-Teacher” derives from both the Student class and the Teacher class. A Student-Teacher has the attributes and behaviors of both a Student and a Teacher. A Student-Teacher does *not* necessarily have the incremental attributes and behaviors of a Freshman. Likewise, a Teacher may be called upon to temporarily perform the duties of a Principal.

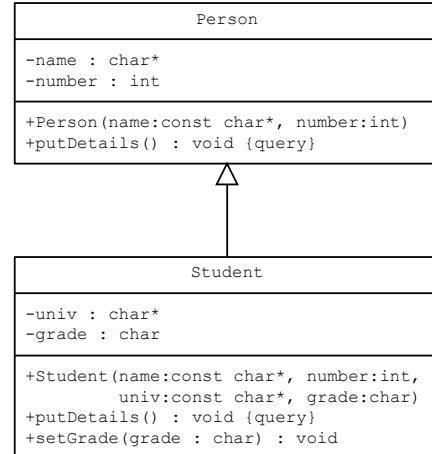
What Is the Unified Modeling Language?



The OMG's **Unified Modeling Language™** (UML®) helps you specify, visualize, and document models of software systems, including their structure and design...

<https://www.uml.org/what-is-uml.htm>

- Designers of software systems to be implemented in object-oriented languages utilize UML diagrams to rigorously document and communicate their design.
- From the UML, they use primarily the class diagrams and diagram elements expressing the relationships between classes.



145 © Cadence Design Systems, Inc. All rights reserved.



The Open Modeling Group (OMG) **Unified Modeling Language™** (UML®) helps you specify, visualize, and document models of software systems, including their structure and design.

Designers of software systems to be implemented in object-oriented languages utilize UML diagrams to rigorously document and communicate their design.

From the UML they use primarily the class diagrams and diagram elements expressing the relationships between classes.

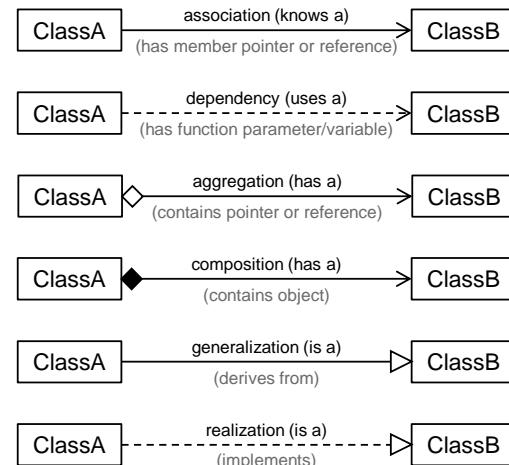
The example specifies first a class "Person" having the private attributes "name" and "number", and the public read-only process "put details" to presumably output the class object attribute values.

The example specifies next a class "Student" derived from the class "Person", having the added private attributes "university" and "grade", and overriding the public read-only process "put details" to presumably also output the added attribute values, and adding a public process to set the "grade" attribute.

The International Standards Organization (ISO/IEC) publishes the UML as ISO/IEC 19505.

Unified Modeling Language: Basic Diagrams for OOP

ClassName
<pre><<attributes>> #protected_attribute : type -private_attribute : type -private_static_attr : type</pre>
<pre><<operations>> <<constructor>> +ClassName([arg:type {,arg:type}]) +public_behavior([arg-types]) : type +public_static_behavior([arg-types]) : type <<helper>> -private_behavior([arg-types]) : type</pre>
<pre><<stereotype>> The UML defines standard stereotypes and you can add your own as comments.</pre>



Every class diagram has at least the one top section for the class name, and may have one or two following sections in their respective order for attributes and/or operations. The UML defines standard stereotypes having capitalized names, and you can enter your own uncapitalized stereotypes as a form of comment. People commonly use the stereotypes “attributes” and “operations”, which in a three-section diagram add no useful information.

Prefix all attributes and operations with a visibility kind literal:

- A private element is visible only inside the namespace that owns it.
- A protected element is visible to only elements that have a generalization relationship to the namespace that owns it.
- A public element is visible to all elements that can access contents of the namespace that owns it.
- A package element is visible to elements that are in the same package as the namespace that owns it. The namespace cannot itself be a package.

The Unified Modeling Language describes inter-class relationships:

- An association relationship (knows a), where a class object knows another class object, by virtue of having as a member attribute, a pointer or reference to that object.
- A dependency relationship (uses a), where a class object uses another class object, by virtue of having as a member function parameter or variable, a pointer or reference to that object.
- An aggregation relationship (has a), where a class object collects other class objects, by virtue of having as a member attribute, a collection of pointers or references to those objects. An aggregation relationship is an association relationship that implies collection.
- A composition relationship (has a), where a class object contains other class objects, by virtue of having as a member attribute, a container of those objects. In a composition relationship, the contained objects have no life of their own outside the containing context.
- A generalization relationship (is a), where a class is derived from another class.
- A realization relationship (is a), where a class is derived from, and implements, an abstract class.
- It also describes other relationships beyond the basic ones depicted here.



Quick Reference Guide: Access Specifiers in Member Specifications

Access Specifier	What code can access this region?			
	Class Member Functions	Friend Functions	Sub-class Member Functions	Application Code
public	yes	yes	yes	yes
protected	yes	yes	yes	no
private	yes	yes	no	no

public: // Accessible by all code including application code.

protected: // Accessible by only class member functions and friend functions and sub-class member functions.

private: // Accessible by only class member functions and friend functions.

```
member-specification ::=  
    member-declaration [member-specification]  
    | access-specifier : [member-specification]
```

```
class Person {  
public:  
    ...  
protected:  
    ...  
private:  
    ...  
};
```

147 © Cadence Design Systems, Inc. All rights reserved.



You can control access to your class members. Access control is essential to the object-oriented paradigm. You can demarcate regions within your class definition:

- A **public** class member is accessible without such restriction.
- A **protected** class member is accessible by only members and friends of the class and members of derived classes.
- A **private** class member is accessible by only members and friends of the class.

All members of a class declared with the keyword “**struct**” are by default public members.

All members of a class declared with the keyword “**class**” are by default private members.

The following is an editorial comment having much to do with style and little to do with language. You may or may not have noticed that the UML class diagram places the attribute compartment over the operation compartment. That has unfortunately led to the poor style of declaring data attributes before functions in the class definition. For C++ classes, the vast majority of the data is nobody else’s business. Functions that are not members must typically call the member functions to manipulate the data. So when writing code, you should place the public functions and the public data first, followed by the protected functions and data, and lastly the private functions and data. This allows users of your class to quickly find and examine your class’ public interface, which is the only parts they can use.



Quick Reference Guide: Access Specifiers in Base Specifiers

Access Specifier	How is access to base class members further restricted?		
	Base Class Public Region Becomes	Base Class Protected Region Becomes	Base Class Private Region Remains
public	public	protected	private
protected	protected	protected	private
private	private	private	private

public: // Base class members retain their access restrictions when accessed through sub-class objects.
protected: // Base class **public** members become **protected** when accessed through sub-class objects.
private: // Base class **public** and **protected** members become **private** when accessed through sub-class objects.

```
base-specifier ::= [attribute-specifier-seq] base-type-specifier
| [attribute-specifier-seq] virtual [access-specifier] base-type-specifier
| [attribute-specifier-seq] access-specifier [virtual] base-type-specifier
```

```
class Person { ... // base class
class Student: public Person { // Person member access unchanged in Student class
class Student: protected Person { // Person public member access becomes protected in Student class
class Student: private Person { // Person public and protected member access becomes private in Student class
```

Access specifiers in base specifiers can further limit access to inherited members, from further derivations, and from application code.

- A **public** specifier, leaves the access unchanged;
- A **protected** specifier, reduces public access, to protected access, so that inherited public members, are no longer accessible to application code; and
- A **private** specifier, reduces public and protected access, to only private access, so that inherited public and protected members, are no longer accessible to application code or further derivations.

Example: Class Inheritance Code

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

class Person { // Base class
public:
    Person(const char *name, int number);
    ~Person() { delete [] name; }
    void putDetails() const;

private:
    char *name;
    int number;
};

Person::Person(const char *name, int number)
    : number(number) {
    this->name = new char[strlen(name)+1];
    strcpy(this->name, name);
}
void Person::putDetails() const {
    cout << name << " " << number << endl;
}
```

Person parts.

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

class Student : public Person { // Derived class
public:
    Student(const char *name, int number,
            const char *univ, char grade);
    ~Student() { delete [] univ; }
    void putDetails() const; // override
    void setGrade(char grade);

private:
    char *univ;
    char grade;
};

Student::Student(const char *name, int number,
                 const char *univ, char grade)
    : Person(name, number), grade(grade) {
    this->univ = new char[strlen(univ)+1];
    strcpy(this->univ, univ);
}
void Student::putDetails() const {
    Person::putDetails();
    cout << univ << " " << grade << endl;
}
```

Student parts.

Person parts.
Then Student parts.

Person parts.
Then Student parts.

149 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The example implements the relationship between a Person and a Student.

A Person always has a name and most governments require that they also have a number. The Person constructor initializes the name and number with the arguments passed to the constructor. The function “put details” (*putdetails*) outputs the person’s name and number.

A Student is a Person, so inherits all the Person attributes and operations. A Student also has a university and a grade-point average. The Student constructor first calls the Person constructor to initialize the student’s Person attributes, then initializes the additional university and grade attributes with the arguments passed to the constructor. The Student class overrides the “put details” function to also output the university and grade-point average.

Multiple Inheritance and Virtual Base Specifiers

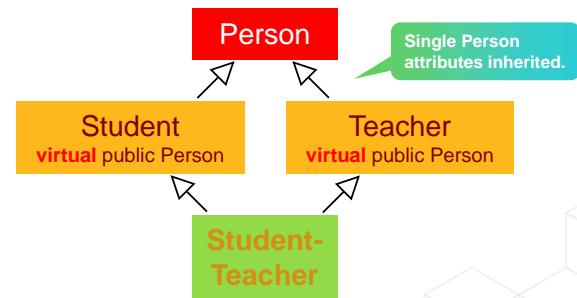
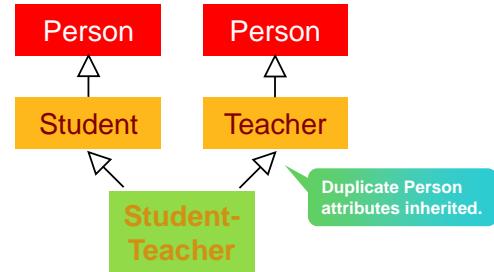


Inheriting multiple base classes that share a common higher base class results in duplicate inheritance.

The solution is to add the keyword **virtual** to the base class specifier.

- Informs the compiler of such potential multiple inheritance.

```
class Student: virtual public Person {  
class Teacher: virtual public Person {
```



150 © Cadence Design Systems, Inc. All rights reserved.



Inheriting multiple base classes that share a common higher base class results in duplicate inheritance.

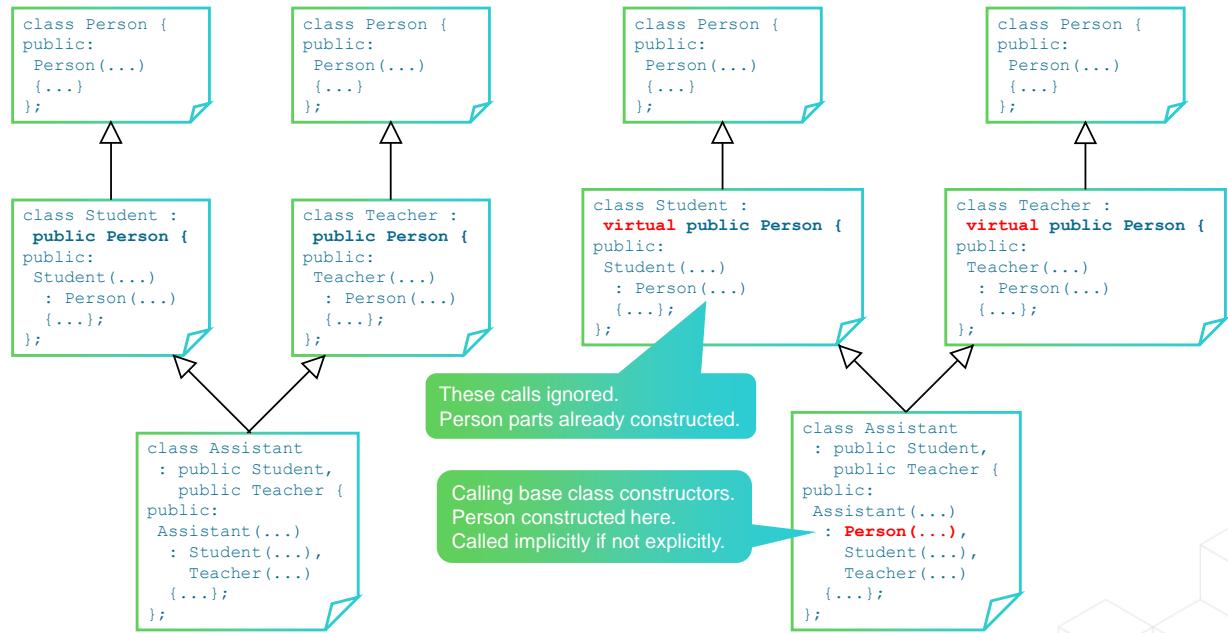
In the situation where more than one base class both inherit the same data members from a yet higher base class:

- If the inheritance is not made virtual, then more than one set of the higher base class data is constructed.
- If the inheritance is made virtual, then only one set of the higher base class data is constructed.

The example class hierarchy accommodates student-teachers. A Student-Teacher is a Person that has the attributes and behaviors of both a Student and a Teacher. Making both the Student class and the Teacher class virtual Persons, inherits and initializes the Student-Teacher class Person attributes only once.

In a situation where inheritance of more than one base class creates a naming conflict, that is, where both base classes define the same identifier, the compiler issues a name ambiguity error. This could happen, for example, if both the Student class and the Teacher class override the Person class `putDetails()` function, but the StudentTeacher class does not. The compiler does not know which base class version to select. Application code could of course qualify the function call, but good coding practice suggests that the StudentTeacher class should also override such functions.

Example: Declaring Virtual Base Classes



151 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Assume that the Person class constructor does some sort of initialization of data common to all Person objects, and the Student and Teacher classes inherit the Person class, and their constructors first call the Person constructor to initialize the common Person data, and then initialize data common to all Student objects or Teacher objects, respectively.

The Assistant class inherits both the Student class and the Teacher class, so an Assistant object normally has duplicate copies of its Person class data. The compiler demands that the Assistant class disambiguate that data when referencing it to specify whether it is the version inherited by the Student or the version inherited by the Teacher.

In situations where it is important that the base class data not be duplicated, you can make the base class a “**virtual**” base class. This notifies the compiler to make only one copy of the base class data and call the base class constructor only once.

Examine the virtual base class hierarchy. The constructors of a Student object and of a Teacher object still first call the Person constructor to initialize the common Person data and then initialize data common to all Student objects or all Teacher objects, respectively. The constructor of an Assistant object now first calls the Person constructor to initialize the common Person data and then calls the Student and Teacher constructors as before to initialize data common to all Student objects or all Teacher objects, respectively. Calls to the Person constructor from the Student and Teacher constructors are ignored, as the Person data is already initialized. Keep in mind that in this situation the Assistant constructor first calls the Person constructor regardless. If it does not explicitly call a Person constructor then the compiler implicitly calls the Person default constructor.

Module Summary

In this module, you

- Defined a class based upon an existing class definition, that inherits the base class attributes and normal functions, and optionally declares additional members

This training module discussed:

- Introduction to Class Inheritance
- Unified Modeling Language (UML) Diagrams
- Access Specifiers in Base Class Specifiers
- Multiple Inheritance and Virtual Base Classes

152 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define a class based upon an existing class definition, that inherits all members of the base class and optionally declares additional members.

To help you to achieve your objective, this training module discussed:

- Introduction to Class Inheritance;
- Unified Modeling Language (UML) Diagrams;
- Access Specifiers in Base Class Specifiers; and
- Multiple Inheritance and Virtual Base Classes.

Quiz: Inheritance



Explain the difference between class *aggregation* and class *inheritance*.



In a UML diagram, the arrow that represents a *generalization* kind of relationship points to the [____] class.



If a derived class inherits from a base class through multiple paths, for example a *StudentTeacher* is both a *Student* and a *Teacher*, and both are a *Person*, how do you prevent multiple initialization of the base class members?

Please pause here to consider these questions.



Lab

Lab 10-1 Deriving Subclasses

Objective:

- To define a class based upon an existing class definition, that inherits all members of the base class and optionally declares additional members

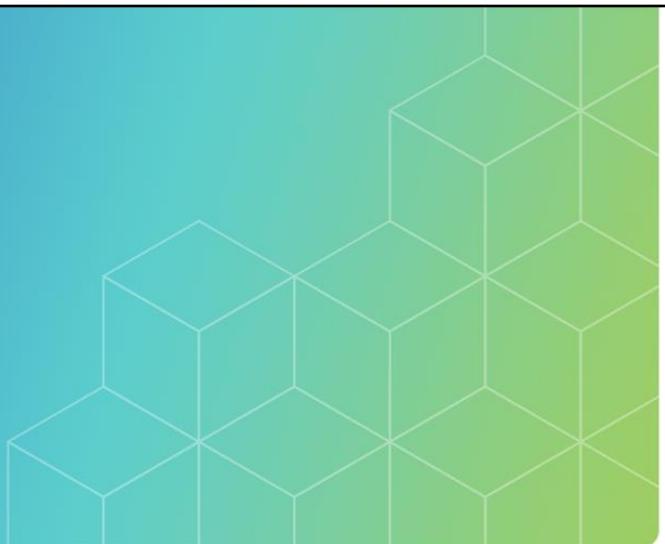
For this lab, you:

- Derive a down-counter class from your counter class and determine which functions to re-implement and which functions to remain inherited “as-is”



Your objective for this lab is to define a class based upon an existing class definition, that inherits all members of the base class, and optionally declares additional members.

For this lab, you derive from your counter class, a down-counter class, and determine which functions to re-implement, and which functions to remain inherited “as-is”.



Module 11

Polymorphism

cadence®

This training module examines object polymorphism.

Module Objectives

In this module, you

- Differentiate between function call name binding done statically and dynamically
- Utilize the virtual specifier to defer function call name binding until run time
- Create an interface common to multiple classes by defining pure virtual functions

This training module discusses:

- Introduction to Polymorphism
- Name Binding: Static & Dynamic
- Pure Virtual Functions and Abstract Classes

156 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Differentiate between early static function call name binding, and late dynamic function call name binding;
- Enable late dynamic function call name binding, by defining virtual functions; and
- Create an interface common to multiple classes, by defining pure virtual functions.

To help you to achieve your objective, this training module discusses:

- An Introduction to Polymorphism;
- Name Binding: Static & Dynamic; and
- Pure Virtual Functions and Abstract Classes.

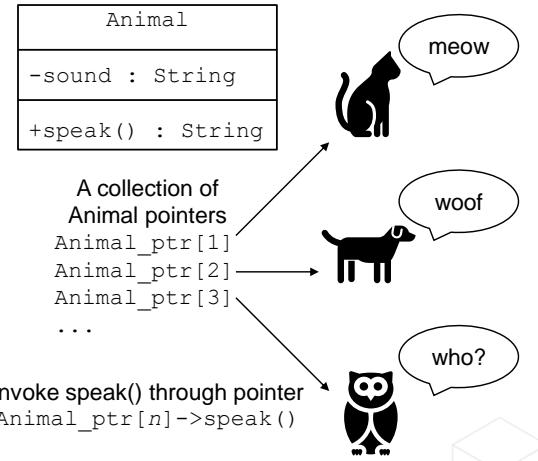
What Is Polymorphism in the OOP Context?



Polymorphism is the ability to have a meaning that depends upon context.

OOP supports three types of polymorphism:

- **Variable polymorphism:** the type of an identifier depends upon what type the expression wants it to be.
- **Function polymorphism:** the function behavior depends upon the number and types of the arguments.
- **Object polymorphism:** an object of a basic type acts like an object of a more complex related type, depending upon how you use the object.



157 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Polymorphism in object-oriented programming means the ability for a variable, function, or object to have a meaning that depends upon the context in which you use it.

- Variable polymorphism, may for example, allow you to use the same variable identifier as either a character string or a numerical value, depending upon which type the expression within which you use it expects.
- Function polymorphism, may for example, allow you to use the same function identifier with multiple sets of parameters, where the function performs different actions depending upon the number and types of the arguments.
- Object polymorphism, may for example, allow you to use an object of a basic type as if it were an object of a more complex related type, depending upon how you use the object.

The illustration depicts a class “Animal” that has a sound and a function to issue that sound. For convenience, we collect pointers to Animal objects, and through the pointers, access the function to issue the sound. Polymorphism selects the sound dynamically, depending upon what Animal subtype the pointer currently points to.

Comparing Static and Dynamic Function Name Binding

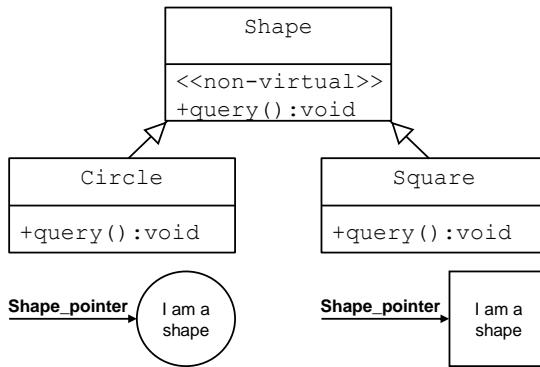


Function name binding done *statically* (“early” – by the compiler) binds according to pointer type.

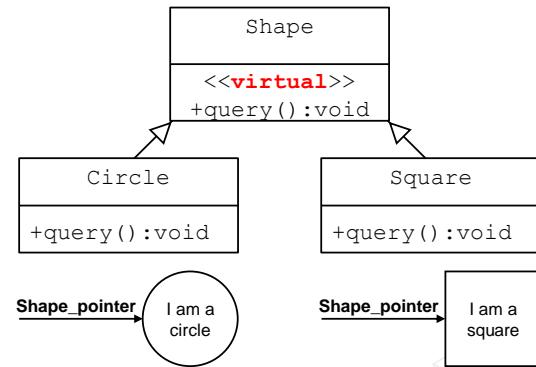


Function name binding done *dynamically* (“late” – by runtime code) binds according to object type.

For normal functions – name binding to a pointer is done statically based on the pointer type.



For **virtual** functions – name binding to a pointer is done dynamically based on the object type.



158 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Function name binding done *statically* (“early” – by the compiler) binds according to *pointer* type.

Function name binding done *dynamically* (“late” – by runtime code) binds according to *object* type.

For non-virtual functions, name binding to a pointer is done statically based on the pointer type.

- The 1st illustration is of a non-virtual function defined in a base class and overridden in the derived class. A base class pointer can point to objects of derived classes. Invoking pointer class non-virtual functions invokes the functions defined in the *pointer* class.

For virtual functions, name binding to a pointer is done dynamically based on the object type.

- The 2nd illustration is of a virtual function defined in a base class and overridden in the derived class. A base class pointer can point to objects of derived classes. Invoking pointer class virtual functions invokes the functions redefined in the *object* class.

Note that runtime function name binding occurs for only virtual functions invoked through a pointer.

- The “dot” (.) class member access operation is always resolved at compile time.
- A reference operation is resolved:
 - For non-virtual functions – at compile time, based on reference type.
 - For virtual functions – at compile time, based on object type.
- A pointer (->) class member access operation is resolved:
 - For non-virtual functions – at compile time, based on pointer type.
 - For virtual functions – at runtime time, based on object type.

Example: Static Name Binding Code

```
class Point {
    public:
        Point(int x=0, int y=0);
        void printme() const;
    private: int x,y; };
```

Base class

Normal function

```
class Circle: public Point{
    public:
        Circle(double r=0.0, int x=0, int y=0);
        void printme() const;
    private: double r; };
```

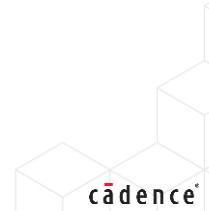
Derived class

Overridden function

```
int main() {
    Point p, *pPtr = &p;
    Circle c, *cPtr = &c;
    pPtr->printme(); // "I am a Point"
    cPtr->printme(); // "I am a Circle"
    cPtr->Point::printme(); // "I am a Point"
    pPtr = cPtr; // Points to circle
    pPtr->printme(); // "I am a Point"
}
```

Name bound at *compile time*.
Unqualified name look-up by *pointer type*.

159 © Cadence Design Systems, Inc. All rights reserved.



You can define a point by providing “x” and “y” coordinates. Here, the Point class has a constructor to initialize its “x” and “y” coordinates, and a “print me” (*printme*) function to print out “I am a Point”.

You can define a Circle by providing “x” and “y” coordinates and a radius. Here, the Circle class is based upon the already defined Point class, and has a constructor to initialize its radius and invoke the Point constructor to initialize its “x” and “y” coordinates, and has a “print me” function to print out “I am a Circle”.

The calls to the “print me” function are by default bound during compilation, so when called through a Point class pointer, execute the Point class “print me” function, even when the Point class pointer is pointing to a Circle object.

Situations exist in which you would like to use a base class pointer to point to objects of various derived classes. Imagine for example an array of such pointers. Elements of an array must all be of the same type. If the array element type is a pointer to a base class, then pointer elements can point to any object that inherits the base class.

Example: Dynamic Name Binding Code

```
class Point {
public:
    Point(int x=0, int y=0);
    virtual void printme() const;
private: int x,y; };
```

Base class

```
class Circle:public Point{
public:
    Circle(double r=0.0, int x=0, int y=0);
    void printme() const;
private: double r; };
```

Derived class

```
int main() {
    Point p, *pPtr = &p;
    Circle c, *cPtr = &c;
    pPtr->printme(); // "I am a Point"
    cPtr->printme(); // "I am a Circle"
    cPtr->Point::printme(); // "I am a Point"
    pPtr = cPtr; // Points to circle
    pPtr->printme(); // "I am a Circle"
}
```

Name bound at *run time*.
Unqualified name look-up by *object type*.

160 © Cadence Design Systems, Inc. All rights reserved.



The compiler resolves a call of a *non-virtual* member function statically, depending on the *pointer type* with which the call is made.

The compiler resolves a call of a *virtual* member function dynamically, depending on the *object type* for which the call is made.

The calls to the virtual function “print me” (*printme*) are bound during run time, so when called using a *Point* class pointer, execute the “print me” function of the object being pointed to.

You do not need to repeat the “**virtual**” keyword in derived classes. You may choose to do so to improve readability.

Pure Virtual Functions and Abstract Classes



For **virtual** functions, name binding to a pointer is done *dynamically* based on the *object* type.

The base class must implement the function. The class definition is complete, so can be instantiated.

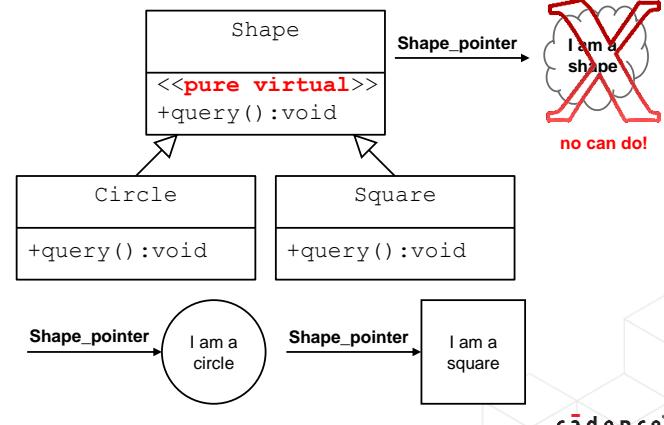
In some situations, the virtual function is merely a placeholder with a trivial implementation.

How to prevent constructing a base class object and force *derivations* to implement the function?

The solution is to make the virtual function “pure”.

```
virtual type func(args) = 0
```

- A “pure” virtual function is not implemented.
- The class definition is incomplete so cannot be instantiated.
 - An *abstract class*, also termed an *interface*
 - Provides a common base class pointer type
 - Defines public interface functions that derivations must implement
 - Supports reusability (“plug-and-play”) between objects of differently derived classes.



161 © Cadence Design Systems, Inc. All rights reserved.

cadence®

You have seen that a base class can define a function, and classes derived from the base class can override that function, and if the function is a virtual function, then a base class pointer accesses the version of the function that is correct for the type of object the pointer points to.

Programmers often use base classes only to provide such a base class pointer, and perhaps to declare some functions that they then know all derived classes will have. Perhaps the base class version of the functions does nothing meaningful and the programmer expects the derived classes to provide meaningful overrides. The programmer prefers that nobody actually instantiate objects of that base class.

A class that you cannot instantiate is termed an “abstract” class.

One way that you can prevent construction of a class object is by hiding the constructors. If you do not provide a default constructor, the compiler provides one if needed. If you provide a protected default constructor then the compiler will not provide one, and nobody can instantiate an object of the class.

Another way to prevent construction of a class object is by not completely defining the class. An incomplete type cannot be instantiated. The compiler normally complains about an incomplete type, but C++ provides a syntax to tell the compiler that you know what you are doing. The syntax is to replace the body of a virtual function with the “*pure-specifier*”, similar to a “*constant-initializer*” of value 0. This function is now termed a “pure” virtual function. No class in the derivation hierarchy can be instantiated until the one for which all such pure virtual functions have been implemented.

The illustration declares a class “Shape” pure virtual function, thus making the Shape class an abstract class that cannot be instantiated, and also guaranteeing that any derived class will in its class hierarchy somehow implement that function.

Example: Abstract Class with Pure Virtual Function Code

```
// abstract base class - cannot construct
class Shape {
public:
    ...
    // "pure" virtual function -- no implementation
    virtual void query() = 0 ;
};

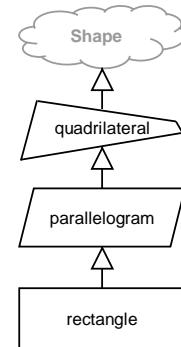
// derivation
class Quadrilateral : public Shape {
public:
    ...
    void query();
};

// implementation -- class complete -- can construct
void Quadrilateral::query() { ... }

// further derivation
class Parallelogram : public Quadrilateral {
public:
    ...
    void query();
};
// overridden implementation
void Parallelogram::query() { ... }
```

```
int main() {
    Shape *sPtr;
    //sPtr = new Shape;           // ERROR - abstract class
    sPtr = new Quadrilateral;   // OKAY
    sPtr -> query();          // Query Quadrilateral
    sPtr = new Parallelogram;
    sPtr -> query();          // Query Parallelogram
    return 0;
}
```

162 © Cadence Design Systems, Inc. All rights reserved.



Here, the class “Shape” declares the pure virtual function “query” (*query*). Hence, the Shape class is an abstract class that cannot be instantiated.

The class “Quadrilateral” derives from the Shape class, and provides an implementation of the “query” function. That class definition is complete, so can be instantiated.

The class “Parallelogram” derives from the Quadrilateral class, and re-implements the “query” function to reflect the Parallelogram class incremental properties.

You can imagine yet further derived classes of types “Rectangle” and “Square”.

Best Practice: Polymorphism also Demands Virtual Destructors!



Function name binding statically versus dynamically applies also to *destructors*.

To call the polymorphic class destructor appropriate for the *object* type, you must also declare the *destructor* function **virtual**.

- Upon declaring any base class function **virtual**, also immediately declare the base class destructor **virtual**.
 - This is to prevent later surprises!
 - Calls the destructor of the object type instead of the destructor of the pointer type.
 - Helps to prevent a “memory leak”.
- Unlike other functions, a base class *pure virtual* destructor must also be *defined* in the base class.
 - Doing this does not negate the abstract nature.

```
struct TransBase {
    TransBase() = default;
    virtual ~TransBase()=0; // Pure virtual
}; // makes abstract
TransBase::~TransBase() {} // Must define!

struct TransData:TransBase {
    TransData(unsigned n) {data = new char[n];}
    ~TransData() {delete [] data;} // Override
    char *data;
};

int main() {
    TransBase *TB_p = new TransData(32767);
    delete TB_p; // binds to object destructor
    return 0;
}
```

163 © Cadence Design Systems, Inc. All rights reserved.

cadence

Virtual function binding to base class pointer is dynamic, based upon object type instead of on pointer type.

This applies also to destructor functions.

Upon declaring any base class function **virtual**, also immediately declare the base class destructor **virtual**.

- Doing this may prevent later surprises!
- Deleting the object through the base class pointer, then binds to the destructor of the object type instead of to the destructor of the pointer type.
- Doing this helps to prevent a “memory leak”.

Making any base class **virtual** function “pure” makes the base class an abstract class that cannot be instantiated. If you still want a base-class definition of functions, then you can make only the destructor a **pure virtual** function. This makes the base class an abstract class that cannot be instantiated, and as you must also define the **pure virtual** destructor in the base class, the base class can have all functions defined, but still be an abstract class.

The example code defines a transaction base class having a **pure virtual** destructor, thus making it an abstract base class. The derived transaction data class, that contains the actual transaction payload, overrides the destructor to delete the data payload. Imagine that multiple differently-derived transactions are defined, and that application code utilizes a container of base-class pointers to handle objects of these multiple types. If the base class destructor was not **virtual**, then deleting the object through the base class pointer would not invoke the derived class destructors, thus creating a memory “leak”.

Constructors and destructors are not inherited, but a derived-class destructor does override a base-class **virtual** destructor.

Module Summary

In this module, you

- Differentiated between static and dynamic function call name binding done
- Deferred function call name binding until run time by declaring functions with the **virtual** specifier
- Created an interface common to multiple classes by defining pure virtual functions

This training module discussed:

- Introduction to Polymorphism
- Name Binding: Static & Dynamic
- Pure Virtual Functions and Abstract Classes

164 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Differentiate between early static function call name binding, and late dynamic function call name binding;
- Enable late dynamic function call name binding, by defining virtual functions; and
- Create an interface common to multiple classes, by defining pure virtual functions.

To help you to achieve your objective, this training module discussed:

- An Introduction to Polymorphism;
- Name Binding: Static & Dynamic; and
- Pure Virtual Functions and Abstract Classes.

Quiz: Polymorphism



Compare *static* name binding and *dynamic* name binding.



What does “polymorphism” mean in the C++ context?



Should you make a destructor a *virtual* function?



What is the purpose of a “pure” virtual function?

Please pause here to consider these questions.



Lab

Lab 11-1 Defining and Using Polymorphic Classes

Objective:

- To defer function call name binding until run time, thus making a class hierarchy polymorphic

For this lab, you:

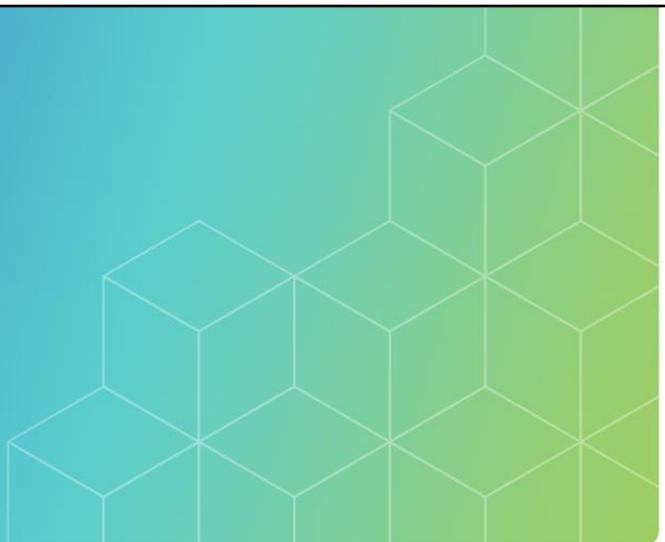
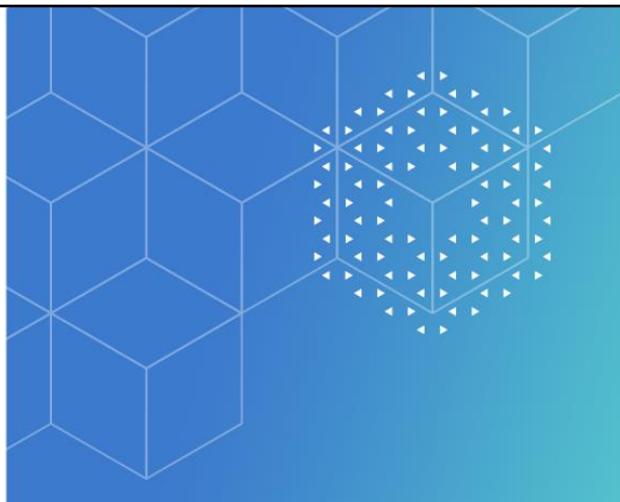
- Access a down-counter function whose name is statically bound
- Access a down-counter function whose name is dynamically bound
- Down-cast a base class pointer or reference to enable using it to access a derived class incremental member



Your objective for this lab is to defer function call name binding until run time, thus making a class hierarchy *polymorphic*.

For this lab, you:

- Access a down-counter function whose name is statically bound;
- Access a down-counter function whose name is dynamically bound; and
- Down-cast a base class pointer or reference and use it to access a derived class incremental member.



Module 12

Constant Objects and Constant Functions

cadence®

This training module more deeply explores the “constant” (**const**) specifier.

Module Objectives

In this module, you

- Declare variables with the **const** qualifier to ensure their values remain unchanged
- Declare member functions of a **const** class object with the **const** qualifier to allow their calling
- Remove the "constness" of pointers and references by performing a **const_cast** operation
- Move object and function evaluation from run time to compile time by declaring them **constexpr**

This training module discusses:

- Constant Variables and Constant Objects
- Constant Objects and Constant Member Functions
- Constant Member Functions and Constant Cast
- Constant Member Functions and Mutable Variables
- Constant Expressions

168 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Declare variables whose values cannot be changed, by declaring the variables with the “constant” (**const**) qualifier;
- Enable calling member functions of a constant class object, by declaring the functions with the “constant” qualifier; and
- Remove the “const-ness” of pointers and references, by performing a “constant-cast” (**const_cast**) operation.

To help you to achieve your objective, this training module discusses:

- Constant Variables and Constant Objects;
- Constant Objects and Constant Member Functions;
- Constant Member Functions and Constant Cast;
- Constant Member Functions and Mutable Variables; and
- Constant Expressions.

What Is a **const** Object or Variable?

The cv-qualifier **const** qualifies the variable or object type as constant.



You must initialize a **const** variable (construct the object) upon definition and cannot later change it.

```
const int ci = 5;           // "ci" is integer constant, value 5
int const ic = 5;          // "ic" is integer constant, value 5
ci = 7;                   // Error: cannot change const int value
int *ip = &ci;             // Error: int* can't point to const int
const int *cip = &ci;      // OK: const int* points to const int
(*cip) = 7;                // Error: cannot change const int value
const int ci2 = 7;          // "ci2" is integer constant, value 7
cip = &ci2;                // OK: const int* points to const int
const int * const cipc = &ci; // "cipc" is pointer constant to int constant
cipc = &ci2;               // Error: can't change pointer constant
```

Easier understood if read backward:
"constant pointer to int constant"

169 © Cadence Design Systems, Inc. All rights reserved.



With the “constant” (**const**) qualifier you specify a variable to be constant. You must initialize the variable upon definition and cannot later change its value.

You can declare a pointer to a constant type, and unless you also declare the pointer itself to be constant, you can change the pointer, but not the object it points to, not even if the pointed-to object is itself not constant. These rules concerning pointers are worth reiterating:

- You cannot set a pointer to a non-constant type to point to an object of a constant type.
- You cannot use a pointer to a constant type to modify an object, even if the object itself is not constant.

For the example, the identifiers are abbreviations of the type, so “CI” is a constant integer, “CIP” is a constant integer pointer, that is, a pointer to integer constants, and “CIPC” is a constant integer pointer constant, that is, a constant pointer to integer constants.

The example erroneously attempts to modify a constant integer, erroneously attempts to set a “pointer-to-integer” to point to a constant integer, and erroneously attempts to modify a constant pointer.

The type of an object includes the cv-qualifiers specified in the decl-specifier-seq, declarator, type-id, or new-type-id when the object is created. – ISO/IEC 14882-2020 6.8.3 CV-qualifiers

What Is a **const** Member Function?

The cv-qualifier **const** qualifies the member function as constant.



- A **const** member function can't modify non-static data members† († Unless declared mutable) or call non-**const** member functions.
- Thus, for a **const** object, you can only call the **const** and **static** member functions.

```
class Motorcycle {  
public:  
    Motorcycle (int fuel, int disp) : fuel(fuel), disp(disp){}  
    void refuel (int fuel) { this->fuel += fuel; }  
    void rebuild (int disp) { this->disp = disp; }  
    void display() const  
    { cout << "\tfuel = " << fuel  
      << "\tdisp = " << disp << endl; }  
private:  
    int fuel; int disp;  
};
```

For **const** objects can call
only **const** member functions.

```
int main()  
{  
    Motorcycle moto_m(25,300);  
    moto_m.refuel(50);  
    moto_m.rebuild(500);  
    moto_m.display();  
    const Motorcycle moto_c(84,250);  
    moto_c.refuel(50); // Error  
    moto_c.rebuild(500); // Error  
    moto_c.display(); // Okay  
    return 0; }
```

170 © Cadence Design Systems, Inc. All rights reserved.

cadence

You cannot normally modify a constant (**const**) object. So for a constant object you cannot call normal member functions. With the constant qualifier, you specify a member function that does not normally modify the current object, thus enabling calling it for a constant object!

You can apply the constant type qualifier to non-static member functions. This has the effect of making the “this” pointer implicitly passed to the function a pointer to a constant object, so the function cannot modify the object, even if the object is itself not constant. These rules concerning member functions are worth reiterating:

- You cannot invoke the non-constant functions of a constant object.
- You cannot use a constant function to modify the current object, even if the current object is itself not constant.

In the example, the “display” (*display*) function does not modify the current object. It is declared constant so that it can be invoked for constant objects. A constant object is defined and only the “display” function can be called for that object.

Casting “Away” the Const-ness of a Pointer-to-Const-Type

`const_cast < type-id > (expression)`



You can recast a Pointer-to-Constant-Type as a Pointer-to-Non-Constant-Type.

- You can then use the pointer to modify a pointed-to non-const object.
- An attempt to modify a **const** object is not defined or portable.

```
class Motorcycle {
public:
    Motorcycle (int fuel, int disp) : fuel(fuel), disp(disp) { }
    void refuel (int fuel) const {
        Motorcycle *thisNC = const_cast<Motorcycle*>(this);
        thisNC->fuel += fuel;
    }
    void rebuild (int disp) { this->disp = disp; }
    void display() const { cout << "\tfuel = " << fuel
                           << "\tdisp = " << disp << endl; }
private: int fuel, disp;
};
```

Casts away const-ness of *this*.
(not the pointed-to object)

```
int main()
{
    Motorcycle moto_m(25,300);
    moto_m.refuel(50); // Okay
    moto_m.rebuild(500);
    moto_m.display();
    const Motorcycle moto_c(84,250);
    moto_c.refuel(50); // Undefined
    moto_c.rebuild(500); // Error
    moto_c.display(); // Okay
    return 0;
}
```

171 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Perhaps you want your constant member function to modify data used only for internal purposes that does not significantly affect the “outside” world’s view of the object. With the “constant-cast” expression, you can cast “away” the constant-ness of a pointer-to-constant type, and then use the pointer to modify a non-constant object. The standard does not define the result of such an attempt to modify a constant object, so to use this “workaround” to modify a constant object is not portable.

This operation is of no valid use for the *Motorcycle* class as it is currently defined, but we use it anyway for purely illustrative purposes. The example declares the member function “refuel” (*refuel*) to be constant, so that function cannot normally modify member data. However, the function casts its pointer-to-constant “*this*” pointer to a normal “*this*” pointer and can through the normal “*this*” pointer modify the member data, but reliably only for a non-constant object.

What Is a Mutable Member Variable?



The storage-class-specifier **mutable** specifies a member object or variable that can be modified even if the object itself is **const**.

- For a **const** object, you can only call the **const** and **static** member functions.
- A **const** member function can change only the **mutable** variables of a **const** object.

```
class Motorcycle {
public:
    Motorcycle (int fuel, int disp) : fuel(fuel), disp(disp) { }
    void refuel (int fuel) const { this->fuel += fuel; }
    void rebuild (int disp) const { this->disp = disp; } // Error
    void display() const { cout << "\tfuel = " << fuel
                           << "\tdisp = " << disp << endl; }

private:
    mutable int fuel;
    int disp;
};
```

Const function can modify mutable data.

Const function cannot modify nonmutable data.

```
int main()
{
    Motorcycle moto_m(25,300);
    moto_m.refuel(50);
    moto_m.rebuild(500);
    moto_m.display();
    const Motorcycle moto_c(84,250);
    moto_c.refuel(50); // Okay to call
    moto_c.rebuild(500); // Okay to call
    moto_c.display(); // Okay to call
    return 0;
}
```

172 © Cadence Design Systems, Inc. All rights reserved.



Perhaps you *do* want your constant member function to modify the data of a constant object. With the **mutable** specifier, you specify a member variable that a constant member function can change, even if the current object is constant. A constant member function can change only the mutable variables.

This example declares the member functions “refuel” (*refuel*) and “rebuild” (*rebuild*) to be constant, so application code can call them for constant objects, and so that they can modify member data declared mutable. The example declares the fuel data mutable, but does not declare the displacement data mutable, so the compiler reports an error when the “rebuild” function attempts to change the displacement.

What Is a **constexpr** Object or Function?



The declaration specifier **constexpr** enables the value of an object or function to be usable as a constant expression where a constant expression term is expected.

Roughly meaning: MAY be evaluable at compile time.

The declaration specifier **constexpr** can apply to objects.

- **constexpr** objects are **const** and are initialized with values known during compilation.
 - Some constant-expression objects are **constexpr** anyway without so declaring them.
 - `const int ci=0; // a constant expression anyway`

The declaration specifier **constexpr** can apply to non-member or member functions, including constructors.

- **constexpr** functions produce compile-time results when called with arguments whose values are known during compilation.
 - A constant-expression function must be so declared to be usable in a constant expression.
 - Several restrictions to be used in a constant expression.
 - Can also be used in non-constant expressions.



The declaration specifier “constant expression” (**constexpr**) enables the value of an object or function to be usable as a constant expression where a constant expression term is expected.

The expression MAY be evaluable at compile time, but not necessarily.

The declaration specifier “constant expression” can apply to objects.

- Objects declared “constant expression” are constant and are initialized with values known during compilation.
 - Some objects are constant-expression anyway, without so declaring them.

The declaration specifier “constant expression” can apply to non-member or member functions, including constructors.

- Functions declared “constant expression” produce compile-time results when called with arguments whose values are known during compilation.
 - A constant-expression function must be so declared to be usable in a constant expression.
 - The standard describes several restrictions on using such functions in constant expressions.
 - You can also use such functions in non-constant expressions.

Example: Declaration Specifier **constexpr** Code

```
class C {
public:
    static const int static_const_int;
    static constexpr int static_constexpr_int=2;
};

const int C::static_const_int = 1;

int main()
{
    const int i = C::static_const_int;
    const int j = C::static_constexpr_int;
    return 0;
}
```

```
constexpr int add(int i, int j) {return i+j;}

int main() {
    const int i=1,j=2; int k=3;
    // i,j constexpr thus add(i,j) constexpr
    constexpr int l = add(i,j);
    // k not constexpr thus add(j,k) not constexpr
    constexpr int m = add(j,k); // error
    // not constexpr but not expecting constexpr
    int n = add(j,k);
    return 0;
}
```



The left example, in the class definition, declares a static constant (**const**) integer, and defines a static constant-expression (**constexpr**) integer and initializes it. For the static constant integer, the declaration is not a definition. The definition must appear outside the class definition, but in the same namespace scope.

The right example, defines a constant-expression (**constexpr**) function. This function will be evaluated at compile time if its arguments are constant expressions. The example calls the function three times:

- 1st – Successfully, as a constant-expression initializer, with constant-expression arguments.
- 2nd – Unsuccessfully, as a constant-expression initializer, but with a non-constant-expression argument.
- 3rd – Successfully, as a non-constant-expression initializer, with a non-constant-expression argument.

ISO/IEC 14882-2020 11.4.8.2 Static data members

Comparing Specifiers **const** and **constexpr**



The specifier **const** means to prevent inadvertently changing an object or variable.



The specifier **constexpr** means to move runtime evaluation to compile time.

For **const** object or variable:

- Can be of any type.
- Initialization may be executed at compile time.
- Cannot change after initialization.
- Is also **constexpr** if integral or enum type and declaration-initialized with constant expression.

For **const** function:

- Must be a non-static member function.
- Cannot call non-static non-const member functions.
- Cannot modify non-static non-mutable member data.

For **constexpr** object or variable:

- Must be of literal type and be initialized.
- Initialization is always executed at compile time.
- Cannot change after initialization.
- Is also **const** and **inline** by implication.

For **constexpr** function:

- Parameter and return types must be literal types
- Body cannot enclose a **goto** statement, identifier label, definition of a variable of non-literal type or of static or thread storage duration.
- Evaluation *will* occur at compile time if all inputs are **constexpr**, and can also occur at run time if not.
- Is also **inline** by implication.



The specifier “constant” (**const**) means to prevent inadvertently changing an object or variable.

The specifier “constant expression” (**constexpr**) means to move runtime evaluation to compile time, to improve runtime performance.

For a “constant” (**const**) object or variable:

- It can be of any type;
- Initialization *may* be executed at compile time, but may also be executed at run time;
- Its value cannot change after initialization; and
- It is also a “constant expression” (**constexpr**) if of integral or **enum** type, and declaration-initialized with a constant expression.

For a “constant expression” (**constexpr**) object or variable:

- It must be of literal type and be initialized;
- Initialization is *always* executed at compile time;
- Its value cannot change after initialization; and
- It is also “constant” (**const**) and “**inline**” by implication.

For a “constant” (**const**) function:

- It must be a non-static member function;
- It cannot call non-static non-const member functions; and
- It cannot modify non-static non-mutable member data.

For a “constant expression” (**constexpr**) function:

- Parameter and return types must be literal types;
- The body cannot enclose a **goto** statement, an identifier label, or a definition of a variable of non-literal type or of static or thread storage duration; and
- Evaluation *will* occur at compile time if all inputs are **constexpr**, and can also occur at run time if not.
- The function is **inline** by implication.

Module Summary

In this module, you

- Declared variables with the **const** qualifier to ensure their values remain unchanged
- Declared member functions of a **const** class object with the **const** qualifier to allow their calling
- Removed the “constness” of pointers and references by performing a **const_cast** operation
- Moved object and function evaluation from run time to compile time by declaring them **constexpr**

This training module discussed:

- Constant Variables and Constant Objects
- Constant Objects and Constant Member Functions
- Constant Member Functions and Constant Cast
- Constant Member Functions and Mutable Variables
- Constant Expressions

176 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Declare variables whose values cannot be changed, by declaring the variables with the “constant” (**const**) qualifier;
- Enable calling member functions of a constant class object, by declaring the functions with the “constant” qualifier; and
- Remove the “const-ness” of pointers and references, by performing a “constant-cast” (**const_cast**) operation.

To help you to achieve your objective, this training module discussed:

- Constant Variables and Constant Objects;
- Constant Objects and Constant Member Functions;
- Constant Member Functions and Constant Cast;
- Constant Member Functions and Mutable Variables; and
- Constant Expressions.

Quiz: Constant Objects and Constant Functions



Compare a **const** pointer and a *pointer-to-const-type*.



Why should you declare as **const** *any* function that does not modify the current object?



Suggest why the standard cannot guarantee that you can cast “away” the “constness” of a *pointer-to-const-type*, and then use the pointer to modify a **const** object.

Please pause here to consider these questions.



Lab

Lab 12-1 Experimenting with Constant and Mutable Objects

Objective:

- To declare class variables whose values cannot be changed
- To enable calling member functions of a **const** class object
- To remove the “constness” of pointers and references

For this lab, you:

- Cast “away” the “constness” of a **const** reference and use it to access a non-const member function of a non-const object
- Cast “away” the “constness” of a **const** reference and use it to access a non-const member function of a **const** object
- Modify a **mutable** variable of a **const** object

178 © Cadence Design Systems, Inc. All rights reserved.

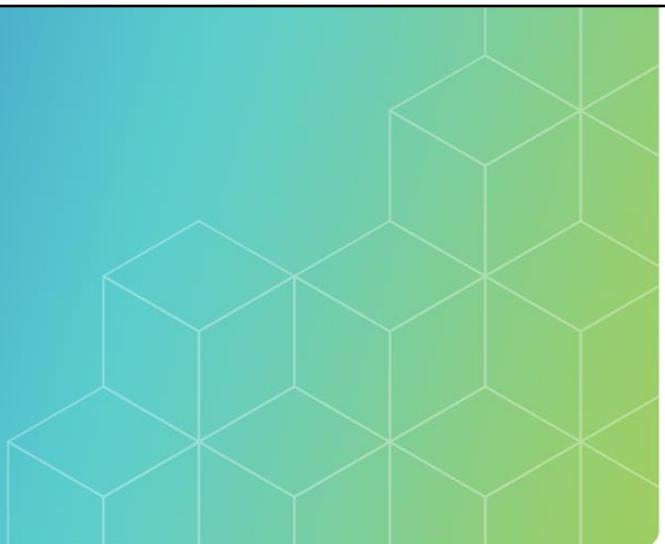
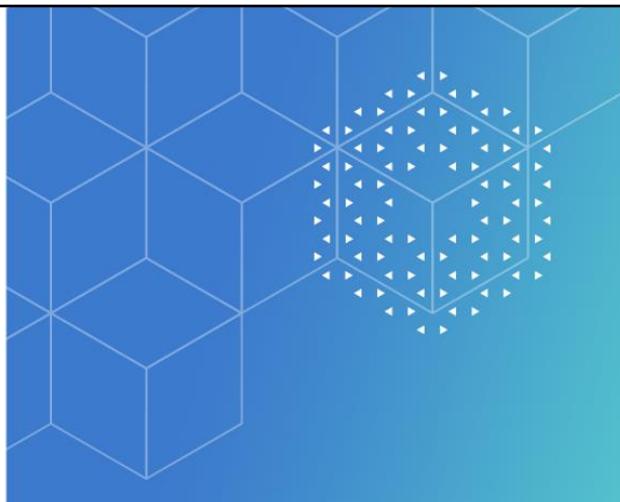


Your objective for this lab is to:

- Declare class variables whose values cannot be changed;
- Enable calling member functions of a **const** class object; and
- Remove the “constness” of pointers and references.

For this lab, you:

- Cast “away” the “constness” of a **const** reference and use it to access a non-const member function of a *non-const* object;
- Cast “away” the “constness” of a **const** reference and use it to access a non-const member function of a *const* object; and
- Modify a **mutable** variable of a *const* object.



Module 13

Templates

cadence®

This training module examines generic programming, that is, templates.

Module Objectives

In this module, you

- Define function templates and class templates to define specialized generic code for each use

This training module discusses:

- Function Templates
- Default Function Template Arguments
- Resolving References to a Function
- Class Templates
- Default Class Template Arguments
- Resolving References to a Class
- Template Parameters
- Template Constraints
- Concepts

180 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define generic code to be specialized upon each use, by defining function templates and class templates.

To help you to achieve your objective, this training module discusses:

- Function Templates;
- Default Function Template Arguments;
- Resolving References to a Function;
- Class Templates;
- Default Class Template Arguments;
- Resolving References to a Class;
- Template Parameters;
- Template Constraints; and
- Concepts.

What Is a Function Template?



A **function template** defines a *family* of functions that differ only in the value(s) and/or type(s) of the template parameter(s).

```
template < template-parameter-list > requires-clauseopt declaration
```

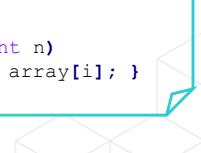
- Template parameters can be *non-type*, *type*, or *template*.
- Any template parameter except a *template parameter pack* can have a default argument.
- The compiler may deduce template types from the call arguments.
- You can explicitly specify template arguments as you make the call.
- The function definition does not need to use the template parameters.

```
template <class T> T square (const T & x)
{ return x*x; }

...
int i1, i2=5;
i1 = square(i2);           Instantiates a function taking an int arg.
float f1, f2=1.1;
f1 = square(f2);           Instantiates a function taking a float arg.
```

```
// works for any T for which the
// cout << operator is defined
template <class T>
void printArray (const T * array, int n)
{ for (int i=0; i<n; ++i) cout << array[i]; }
```

181 © Cadence Design Systems, Inc. All rights reserved.



A function template defines a family of functions that differ only in the values and/or types of the template parameters.

Template parameters can be *non-type*, *type*, or *template*.

Any template parameter except a *template parameter pack* can have a default argument.

The compiler may deduce template types from the call arguments.

The compiler examines the function call to determine which function type to instantiate and call.

- If the template has no non-type parameters, and all type parameters appear at least once in the function parameter list, the compiler can deduce the template arguments from the call arguments. In this case, it looks only at the function name and argument types, and performs no data type conversions to match the call.
- If the template has non-type parameters, or not every template type parameter appears at least once in the parameter list, the compiler cannot deduce the template arguments from the call arguments. In this case, you must specify the template arguments as you call the function.

The example “square” function template will work for any type for which the multiply operator is defined.

The example “print array” (printArray) function template will work for any type for which the ostream insertion operator is defined.

You must in general craft code in the function body to accept arguments of anticipated data types. Some operators, for example cannot be overloaded, so could in some contexts restrict the arguments to only the primitive types.

ISO/IEC 14882-2020 13.2 Template parameters (14) A template parameter pack of a function template shall not be followed by another template parameter unless that template parameter can be deduced from the parameter-type-list of the function template or has a default argument.



Example: Function Template Default Arguments

You can specify default values for the function template parameters.

Template	Instantiation
template <class T1 , int I1 , class T2 , int I2 > void TF()	TF <bool,1,int,2> () ;
template <class T1 , int I1 , class T2=int, int I2=2> void TF()	TF <bool,1,int,2> () ; TF <bool,1,int > () ; TF <bool,1 > () ;
template <class T1=bool, int I1=1, class T2=int, int I2=2> void TF()	TF <bool,1,int,2> () ; TF <bool,1,int > () ; TF <bool,1 > () ; TF <bool > () ; TF < > () ; // Brackets not needed.
template <class T1, class T2 > void TF(T1 t1, T2 t2)	TF (false, 4) ; // T1 deduced. T2 deduced.
template <class T1, class T2=int> void TF(T1 t1)	TF (true) ; // T1 deduced. T2 default.
template <class T1, class T2 > void TF(T2 t2)	query6<bool>(6) ; // T1 specified. T2 deduced.

182 © Cadence Design Systems, Inc. All rights reserved.



For function templates, you can provide default template arguments.

The parameters having default arguments must appear after any parameters not having default arguments.

Any reference to the function must specify all non-default template arguments.

When referencing a function template with an empty argument list, you can omit the list brackets.

For these templates:

- The 1st example provides no default arguments, so the instantiation must supply all arguments.
- The 2nd example provides two trailing default arguments, so the instantiation can omit either the last or both of those trailing arguments.
- The 3rd example provides all parameters with default arguments, so the instantiation can omit any trailing contiguous set of arguments.
- The 4th example illustrates type parameters deduced from function arguments.
- The 5th example illustrates a 1st type parameter deduced from a function argument, and a 2nd type parameter having a default argument.
- The 6th example illustrates a 1st type parameter specified upon instantiation, and a 2nd type parameter deduced from a function argument.

The 2011 standard first introduced default function template parameter arguments.



Quick Reference Guide: Resolving References to a Function

The precedence for resolving function calls is as follows:

1. Exact match to non-template function unless template explicitly specified.
 - o `void printArray(int *array, int n);`
 - o `printArray(int_array, int_n);`Exact match takes precedence
2. Exact match to function template if matching template specified or no matching non-template function.
 - o `template <class T> void printArray(T *array, int n);`
 - o `printArray<int*, int>(int_array, int_n); // template explicitly specified or`
 - o `printArray(int_array, int_n); // assume no non-template function`
3. Normal overload resolution is applied to other non-template functions:
 - Exact match after promoting argument types.
 - o `bool/char/short` \Rightarrow `int` \Rightarrow `long` \Rightarrow `long long`
 - o `float` \Rightarrow `double`
 - Exact match after promotions and standard conversions.
 - o `long` \Rightarrow `int`, `double` \Rightarrow `float`, `int` \Leftrightarrow `float`
 - Exact match after promotions and standard conversions and user-defined conversions.
 - o Using constructors and cast operators

183 © Cadence Design Systems, Inc. All rights reserved.



A non-template or template function can be overloaded by any number of non-template and template functions of the same name.

Explicit specializations can appear after the primary template.

The precedence for resolving a function call is this:

1. Firstly, the compiler attempts to match the call to a non-template function having an exactly matching signature.
2. Secondly, the compiler attempts to match the call to a function template that can be made to match the call.
3. Thirdly, the compiler attempts the normal overload resolution it applies to non-template functions.

What Is a Class Template?



A **class template** defines a *family* of classes that differ only in the value(s) and/or type(s) of the template parameter(s).

```
template < template-parameter-list > requires-clauseopt declaration
```

- Template parameters can be *non-type*, *type*, or *template*.
- Any template parameter except a *template parameter pack* can have a default argument.
- The compiler may deduce template types from implicit or explicit deduction guides.
- You can explicitly specify non-default template arguments as you reference the class type.
- The class definition does not need to use the template parameters.

```
template <class T> class Stack {  
public:  
    Stack(int max=50): max(max), top(0) {array=new T[max];}  
    void push(const T &element) {array[top++]=element;}  
    T pop() {return array[--top];}  
private:  
    T *array;  
    int max, top;};
```

T requires operator=() definition.

```
Stack<int> stackInt;  
Stack<char> stackChar;  
Stack<float> stackFloat;  
int i=1; char c='c'; float f=1.1;  
stackInt.push(i);  
stackChar.push(c);  
stackFloat.push(f);  
cout << stackInt.pop() << " "  
    << stackChar.pop() << " "  
    << stackFloat.pop() << endl;
```

T requires cout operator<<() definition.

184 © Cadence Design Systems, Inc. All rights reserved.

A class template defines a family of classes that differ only in the values and/or types of the template parameters.

Template parameters can be *non-type*, *type*, or *template*.

Any template parameter except a *template parameter pack* can have a default argument.

The compiler may deduce template types from implicit or explicit deduction guides.

You can explicitly specify template arguments as you reference the class type.

The class definition does not need to use the template parameters.

The compiler must have access to the definition of class template member functions, and not merely their declaration. For class templates, programmers almost invariably place the member function definitions in the header file *with* the class definition. Some compilers will search for a body file of the same name as the header file, but you should not rely upon this.

The example Stack class template generates a stack of elements of some type T. The application code creates stacks of integer (**int**), character (**char**), and **float**.

ISO/IEC 14882-2020 13.2 Template parameters (14) If a *template-parameter* of a class template, variable template, or alias template has a default *template-argument*, each subsequent *template-parameter* shall either have a default *template-argument* supplied or be a template parameter pack. If a *template-parameter* of a primary class template, primary variable template, or alias template is a template parameter pack, it shall be the last *template-parameter*.



Example: Class Template Default Arguments

You can specify default arguments for the class template parameters.

Template	Instantiation
<code>template <class T1, int I1, class T2, int I2> class TC</code>	<code>TC <bool,1,int,2> tc;</code>
<code>template <class T1, int I1, class T2=int, int I2=2> class TC</code>	<code>TC <bool,1,int,2> tc; TC <bool,1,int > tc; TC <bool,1 > tc;</code>
<code>template <class T1=bool, int I1=1, class T2=int, int I2=2> class TC</code>	<code>TC <bool,1,int,2> tc; TC <bool,1,int > tc; TC <bool,1 > tc; TC <bool > tc; TC < > tc;</code>

185 © Cadence Design Systems, Inc. All rights reserved.



For class templates, you can provide default template arguments.

The parameters having default arguments must appear after any parameters not having default arguments.

Any reference to the class must specify all non-default template arguments.

When referencing a class template with an empty argument list, you must still provide the list brackets.

For these templates:

- The 1st example provides no default arguments, so the instantiation must supply all arguments.
- The 2nd example provides two trailing default arguments, so the instantiation can omit either the last or both of those trailing arguments.
- The 3rd example provides all parameters with default arguments, so the instantiation can omit any trailing contiguous set of arguments.



Quick Reference Guide: Resolving References to a Class

The precedence for resolving class references is as follows:

- Exact match to non-template class unless template explicitly specified.

- o `class C_none {};`
- o `C_none c_none; // no template`

- Exact match to specialized class template.

- o `template <> class C_one<int> {};`
- o `C_one<int> c_int; // template type specified`

- Match to class template.

- o `template <typename T=bool> class C_one {};`
- o `C_one<float> c_float; // use template`
- o `C_one<> c_one; // use template default`

Non-template and template cannot co-exist in same scope.

More specialized takes precedence over less specialized.



A class template name must be unique within its scope.

Explicit specializations can appear after the primary template.

The precedence for resolving a class reference is this:

- Firstly, a non-template instantiation will never match a class template and vice-versa, and cannot have the same name within the same scope.
- Secondly, a match to a more-specialized template is chosen over a match to a less-specialized template.
- Thirdly, a match to a partially-specialized template is chosen over a match to a non-specialized template.

Quick Reference Guide: Template Parameters

Category	Syntax	Description	Example	Example Usage
non-type	<code>type name_{opt}</code>	optional name	<code>template<int A> struct S{};</code>	<code>S<1> s;</code>
	<code>type name_{opt} = default</code>	default value	<code>template<int A=1> struct S{};</code>	<code>S<> s;</code>
	<code>type ... name_{opt}</code>	parameter pack ^{C++11}	<code>template<int... A> struct S{};</code>	<code>S<0,1,2,3> s;</code>
	<code>placeholder name</code>	placeholder type ^{C++17}	<code>template<auto A> struct S{};</code> <code>template<auto... A> struct S{};</code>	<code>S<1> s;</code> <code>S<true,1,3.14,nullptr> s;</code>
type	<code>class name_{opt}</code>	optional name	<code>template<class T> struct S{};</code>	<code>S<int> s;</code>
	<code>class name_{opt} = default</code>	default type	<code>template<class T=int> struct S{};</code>	<code>S<> s;</code>
	<code>class ... name_{opt}</code>	parameter pack ^{C++11}	<code>template<class... T> struct S{};</code>	<code>S<bool,int,float,nullptr_t> s;</code>
	<code>type-constraint^{C++20} name_{opt}</code>	optional name	<code>template <C T> struct S {};</code>	<code>S<int> s;</code>
	<code>type-constraint^{C++20} name_{opt} = default</code>	default type	<code>template <C T=int> struct S {};</code>	<code>S<> s;</code>
	<code>type-constraint^{C++20} ... name_{opt}</code>	parameter pack	<code>template <C... T> struct S {};</code>	<code>S<bool,int,float,nullptr_t> s;</code>
template	<code>template<parameters> class name_{opt}</code>	optional name	<code>template <class U, template<class> class V> struct S {};</code>	<code>S<int,Sa> s;</code>
	<code>template<parameters> class name_{opt} = default</code>	default type	<code>template <class U, template<class> class V=S> struct S {};</code>	<code>S<int> s;</code>
	<code>template<parameters> class ... name_{opt}</code>	parameter pack ^{C++11}	<code>template <class U, template<class> class... V> struct S {};</code>	<code>S<int,Sa,Sb,Sc,Sd> s;</code>

187 © Cadence Design Systems, Inc. All rights reserved.



A template parameter can be a non-type parameter, a type parameter, or itself a template.

Each can omit the parameter name, can have a default value, and can be a parameter pack.

A template having a parameter pack is a variadic template.

A non-type parameter must have a structural type. The structural type can be a placeholder for a type to be deduced.

A type parameter can alternatively be a named concept specifying a constraint on the type.

A template parameter that is itself a template, introduces a nesting of typically type parameters.

In a template the keywords **typename** and **class** are interchangeable, except that until C++17, a template template parameter declaration could use only the keyword **class** and not the keyword **typename**.

What Is a Template Constraint?



A **Template Constraint**^{C++20} constrains the value of one or more template arguments, to select template specialization and select between function overloads.

requires-clause ::= requires constraint-logical-or-expression

```
template<unsigned W> requires W<=64 class my_int
```

- A template constraint may appear in the template parameter list, may alternatively or also follow the template parameter list, and for functions, may alternatively or also follow the declarator.
- Each constraint is a logical expression evaluated left-to-right with short-circuit evaluation.
- The constraints are conjoined (`&&`) and evaluated in this order:
 1. A type-constrained parameter
 2. A *requires-clause* after the template parameter list
 3. A type-constrained placeholder type in an abbreviated function template declaration
 4. A *requires-clause* after a function declaration

```
// An introspective data class whose
// underlying type is long long int
template<unsigned W> requires W<=64

class my_int : public my_int_base {
public:
    // refine my_int_base constructors
    // refine my_int_base operators
private:
    long long int my_value;
};

// Application code
my_int<64> my_int64; // okay
my_int<65> my_int65; // error
```

188 © Cadence Design Systems, Inc. All rights reserved.

cadence®

A **Template Constraint** constrains the value of one or more template parameter arguments, to select template specialization and select between function overloads.

A *template constraint* may follow the template parameter list, and for functions, may alternatively or also follow the declarator.

Each constraint is a logical expression evaluated left-to-right with short-circuit evaluation.

The constraints are conjoined (`&&`) to form the overall constraint, and evaluated in order.

Imagine that you are creating an introspective data class whose underlying type is a single 64-bit integer.

You might replicate the usual initialization, conversion, and operators, and also add introspective features such as name and width.

Because the underlying type is a single 64-bit integer, you constrain the width parameter argument to at most 64.

What Is a Concept?



A **Concept** is a template defining constraints on its template arguments.

```
template <template-parameter-list> requires-clauseopt concept-definition  
concept-definition ::= concept concept-name = constraint-expression;
```

- You define a concept in a namespace scope.
- You cannot further constrain, specialize, or instantiate a concept.
- You use the concept as a constraint on template arguments.

```
// Concept "Hashable" is a requirement that for 'k' of type  
// "Key" functor hash<Key>{}(k) is defined with return type  
// constrained by concept "convertible to type 'size_t'"  
template<typename Key> concept Hashable = requires (Key k) {  
    std::convertible_to<std::size_t>{  
};  
  
// Constrain the parameter directly  
template <Hashable Key>  
void f1(Key) {}  
  
// Constrain via requires-clause after template  
template <typename Key> requires Hashable<Key>  
void f2(Key) {}  
  
// Constrain via requires-clause after function declarator  
template <typename Key>  
void f3(Key) requires Hashable<Key> {};
```

189 © Cadence Design Systems, Inc. All rights reserved.

cadence®

A Concept is a template defining constraints on its template arguments.

You define a concept in a namespace scope.

You cannot further constrain, specialize, or instantiate a concept.

You use the concept as a constraint on template arguments.

The example defines the concept “Hashable” as a requirement that a functor named “hash” is defined, taking an argument of type “Key”, and returning a result convertible to the standard “size” type.

The standard guarantees that class template “hash” specializations are enabled for the null pointer type (nullptr_t) and all constant-volatile-unqualified arithmetic, enumeration, and pointer types.

The example then demonstrates using the concept in three ways:

- To constrain a template parameter directly;
- To constrain a template parameter with a “requires” clause after the template; and
- To constrain a template parameter with a “requires” clause after the function declarator.

ISO/IEC 14882-2020 13.7.8 Concept definitions

ISO/IEC 14882-2020 20.14.18 Class template hash

Module Summary

In this module, you

- Defined generic code to be specialized upon each use by defining function templates and class templates

This training module discussed:

- Function Templates
- Default Function Template Arguments
- Resolving References to a Function
- Class Templates
- Default Class Template Arguments
- Resolving References to a Class
- Template Parameters
- Template Constraints
- Concepts

190 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define generic code to be specialized upon each use, by defining function templates and class templates.

To help you to achieve your objective, this training module discussed:

- Function Templates;
- Default Function Template Arguments;
- Resolving References to a Function;
- Class Templates;
- Default Class Template Arguments;
- Resolving References to a Class;
- Template Parameters;
- Template Constraints; and
- Concepts.

Quiz: Templates



For each different set of template arguments, the compiler creates a function or class from generic code. Suppose that for a specific set of template arguments, you want a function or class to do something differently than that generic way. Can you do this, and if so, how?



Please pause here to consider this question.



Lab

Lab 13-1 Defining a Class Template

Objective:

- To define generic code to be specialized upon each use by defining a class template

For this lab, you:

- Define an object pool class template
- Instantiate and test a pool of objects

192 © Cadence Design Systems, Inc. All rights reserved.

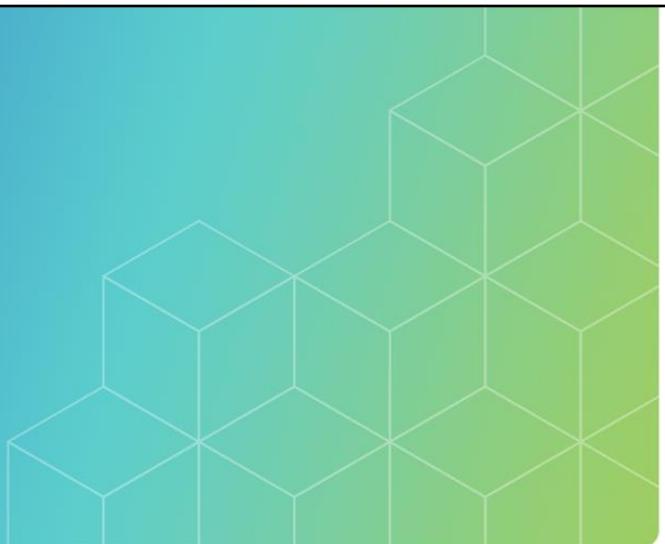
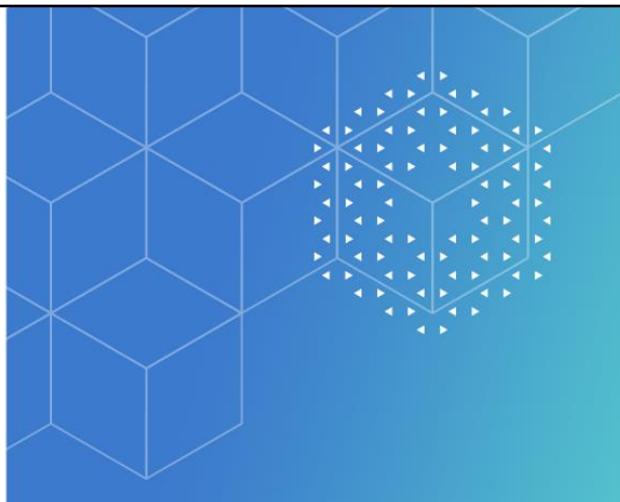


Your objective for this lab is to:

- To define generic code to be specialized upon each use, by defining a class template.

For this lab, you:

- Define an object pool class template; and
- Instantiate and test a pool of objects.



Module 14

Exceptions

cadence®

This training module examines graceful handling of “exceptional” conditions, that is, exceptions.

Module Objectives

In this module, you

- Define the handling of “exceptional” conditions

This training module discusses:

- Introduction to Exceptions
- Throwing and Catching Exceptions
- Indicating Whether a Function May Throw
- Standard Exceptions
- Nested Exceptions

194 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define graceful handling of “exceptional” conditions.

To help you to achieve your objective, this training module discusses:

- Introduction to Exceptions;
- Throwing and Catching Exceptions;
- Indicating Whether a Function May Throw;
- Standard Exceptions; and
- Nested Exceptions.

Problem and Solution: The Exception Mechanism



A function (perhaps a library routine) detects an “exceptional” condition.

The low-level function is unaware of the programmer’s intent – what to do?



- Say nothing and hope for the best?
- Terminate the program?
- Return an error flag?
- Call some specified error routine?

C++ provides a *standard* way for a function to react to an “exceptional” condition.



- An exception handling infrastructure.
- Standard exception base classes.
- An exception methodology that:
 1. Finds a problem – (detects an exception)
 2. Informs that an error occurred – (**throws** an exception object)
 3. Receives the error information – (**catches** the exception object)
 4. Takes corrective actions – (handles the exception)

The exception infrastructure is particularly useful for constructors, which cannot return a status flag.

Failure to catch an exception calls std::terminate()!

A low-level routine may be best positioned to detect an exceptional condition, but not best positioned to determine the most appropriate reaction.

- It could choose to take no action, or to simply terminate the program, both rather drastic steps.
- It could return an error flag, but many functions already return something useful that does not accommodate an error flag.
- At best, the function could call one of a set of error reporting routines, but this requires that every function call pass along an extra function pointer argument collection for the various reporting routines.

The C++ language provides a way for a function encountering a problem to gracefully pass control back to the calling function. The C++ standard provides only the exception handling infrastructure and some standard exception base classes. The infrastructure suggests, but does not define, a process that:

- 1st – Finds a problem, that is, encounters exceptional behavior;
- 2nd – Informs that exceptional behavior occurred, that is, throws an exception object;
- 3rd – Receives the error information, that is, catches the exception object; and
- 4th – Takes corrective actions, that is, handles the exception.

Much of what the exception infrastructure does is what you could do yourself in probably a somewhat more clumsy way. For example, it is very common for functions that could potentially fail, to return an error status to their caller. This approach is well and good and should be done, even in an exception handling environment, but is particularly not applicable to constructors, which cannot return a status.

It is important to remember that this process is almost entirely user-defined. For an exception that is not caught anywhere in the call chain, the standard handler simply terminates the program.

Throwing and Catching Exceptions



```
try {
    statements
    function calls
    etcetera
}
```

throw somewhere in here



```
catch (what) {
    analyze
    report
    rethrow?
}
```

catch must be lexically
after try and at or before
stack frame of throw

From within a **try{} block**, a function can throw an object of any defined data type.

- The remainder of the block does not execute.

One or more **catch(){} blocks** may (and usually do) follow a try block.

- You provide each catch block with a single parameter of the type that it catches or an ellipsis (...) to indicate it catches all types.
- Execution resumes at the catch block, accepting the thrown type that is nearest on the call stack to the throwing function – the stack is unwound as needed.
- A catch block can rethrow the object to allow another catch block earlier on the call stack to continue processing it.
- Execution continues normally after the last catch block in the source text containing the catch block that caught the object.

Failure to catch an exception calls std::terminate()

196 © Cadence Design Systems, Inc. All rights reserved.

A function can define any number of **try blocks**. From within a try block a function can **throw** an object of any defined data type. The remainder of the try block after the throw does not execute.

One or more **catch blocks** may, and usually do, follow a try block:

- You provide each catch block with a single parameter of the type that it catches, or an ellipsis (...) to indicate it catches all types;
- Execution resumes at the catch block accepting the thrown type that is nearest on the call stack to the throwing function. That can be in the throwing function or in any function in the call chain. The stack is unwound as needed;
- You code the catch block to fully or partially process the thrown object;
 - A catch block can rethrow the object to allow another catch block earlier on the call stack to continue processing it.
- Execution continues normally after the last catch block in the source text containing the catch block that caught the object.

For a thrown object that is nowhere caught, the “terminate” function is called.

Example: Throwing and Catching Exceptions

The FPU will detect an attempt to divide to zero and signal (SIGFPE) the executing process.

You may want to more gracefully react to this exception.

```
// DEFINE EXCEPTION CLASS
class DivideByZeroException {
public:
    DivideByZeroException() noexcept {};
    virtual ~DivideByZeroException() noexcept {};
    virtual const char *what() const noexcept
    { return message; }
private:
    static const char message[];
};

const char DivideByZeroException::message[] = "Divide by Zero";

// THROW EXCEPTION OBJECT
double divide(double numerator,double denominator)
{
    if ( denominator==0.0 ) throw DivideByZeroException();
    return numerator/denominator;
}
```

197 © Cadence Design Systems, Inc. All rights reserved.

```
int main()
{
    // TRY EXCEPTION-PRONE CODE
    try
    {
        while (true)
            cout << "Enter numbers: " << flush;
        double num, den;
        cin >> num >> den;
        cout << divide(num,den) << endl;
    }
    // CATCH EXCEPTION HERE
    catch(DivideByZeroException x){
        cout << x.what() << endl;
        return 1;
    }
    // RESUME HERE IF NOT
    // THROWN OR IF CAUGHT
    return 0;
}
```

The operating system detects an attempt to divide by zero and signals the offending process (SIGFPE). The default signal handler in response terminates the process. You can write code to handle the signal, but that requires knowledge of Unix signaling. You may prefer to trap such behavior yourself using C++ exceptions.

The example defines a class “Divide By Zero Exception”, and a global function “divide” that throws a “Divide By Zero Exception” object if the denominator is zero. The “main” function, from within a try block, calls the “divide” function. Immediately after the try block, it defines a catch block to catch “Divide By Zero Exception” objects and call their “what” function to output information about the exception object.

Indicating Whether a Function Might Throw an Exception



type-specifier[†] identifier (parameter-declaration-clause) noexcept-specifier_{opt}
noexcept-specifier ::= noexcept (constant-expression)_{opt}

[†]This syntax is greatly simplified.

- A function declaration whose qualifiers do not include an *exception-specification* might throw an exception.
- A function declaration whose qualifiers include a *noexcept-specifier* will not throw an exception unless the optional constant Boolean expression value is false.
 - Used primarily to define function templates that may throw exceptions for some template arguments and not for others.
 - The compiler does no checking but may further optimize a noexcept function.

```
// Might throw exception
void f();

// Will not throw exception
void f() noexcept;

// Might throw exception if expr false
void f() noexcept(const-expr);

// Can throw if expression can throw
void f() noexcept(noexcept(expression));
```

If the search for a handler encounters the outermost block of a *noexcept true* function, then `std::terminate` is called.

A function declaration whose qualifiers do not include an *exception-specification* might throw an exception.

A function declaration whose qualifiers include a “*no-exception-specifier*” will not throw an exception unless the optional constant Boolean expression value is false.

- The “*no-exception-specifier*” is used primarily to define function templates that may throw exceptions for some template arguments and not for others. It could also be used by a base class to prevent virtual functions overridden in the derived class from throwing an exception.
- The compiler does no checking but may further optimize a function that will not throw an exception.

The illustration shows four declarations of a function.

- The 1st will potentially throw an exception;
- The 2nd will not throw an exception; and
- The 3rd will potentially throw an exception if the constant expression evaluates to false.
- The 4th may throw an exception if the expression may throw an exception.

Note that “no-exception” (**noexcept**) is also an operator, which evaluates an expression, and returns the Boolean “pure right value” (*prvalue*) true or false whether that expression may thrown an exception.

If for example you define a non-member function that calls the equivalent member function, instead of repeating the no-exception conditions, you can instead make the non-member function no-exception if the member function is no-exception.

```
template<> void swap(T &x, T &y) noexcept(noexcept(x.swap(y)));
```

The member function controls the no-exception conditions, which could be for example that the containers are guaranteed swappable, or the containers have identical contents, thus no swap would be done.

Standard Exception Class Code

```
// <exception>
namespace std {
    class exception {                                // Exception base class
        public:
            exception() noexcept;                  // Constructor (default)
            exception(const exception&) noexcept; // Constructor (copy)
            exception& operator=(const exception&) noexcept; // Assignment (copy)
            virtual ~exception();                  // Destructor
            virtual const char* what() const noexcept; // Message
    };
}
```

```
// <stdexcept>
namespace std {
    class logic_error : public exception {          // Example exception subclass
        public:
            explicit logic_error(const string& what_arg); // Constructor (conversion)
            explicit logic_error(const char* what_arg);   // Constructor (conversion)
    };
}
```

199 © Cadence Design Systems, Inc. All rights reserved.



The base class “exception” defines default and copy constructors, the copy assignment operator, the destructor, and the function “what” (*what*) to return a character string message. None of these functions is permitted to throw any type of object. The implementation defines the message for most built-in exception subclasses.

The standard defines a representative set of exception subclasses for various purposes. Displayed here is a generalized “logic error” class for reporting errors detectable before run time. This is a base class for pre-defined and user-defined “logic error” subclasses.



Quick Reference Guide: Standard Exceptions

Library	Exception Class Hierarchy	Thrown by	Description
<exception>	exception	*** incomplete list ***	
<exception>	l_bad_exception	current_exception()	if unable to copy exception
<new>	l_bad_alloc	new()	if unable to allocate storage
<new>	l_bad_array_new_length	new()	if length < 0 or length > limit
<typeid>	l_bad_cast	dynamic_cast<>()	if expression not valid
<functional>	l_bad_function_call	function::operator()	if function wrapper object has no target
<optional>	l_bad_optional_access	optional<>::value() &	if optional object has no value
<typeid>	l_bad_typeid	typeid()	if the pointer value is nullptr
<variant>	l_bad_variant_access	variant<>::get()	if invalid access to variant object
<memory>	l_bad_weak_ptr	shared_ptr<const weak_ptr<Y>>&	if weak_ptr<> has expired
<stdexcept>	l_logic_error		errors detectable before execution
<stdexcept>	l_domain_error		– domain errors
<stdexcept>	l_invalid_argument		– invalid argument
<stdexcept>	l_length_error		– length exceeds limit
<stdexcept>	l_out_of_range		– value outside acceptable range
<stdexcept>	l_runtime_error		errors detectable only during execution
<stdexcept>	l_range_error		– computed value outside acceptable range
<stdexcept>	l_overflow_error		– arithmetic overflow
<stdexcept>	l_underflow_error		– arithmetic underflow
<system_error>	l_system_error		– runtime errors returning error code
<ios>	l_ios_base::failure	ios_base::clear(), setstate()	– ios operation failure
<filesystem>	l_filesystem_error	file system operations	– file system errors

200 © Cadence Design Systems, Inc. All rights reserved.

The standard defines a hierarchy of exception classes derived from the exception class. This hierarchy is an attempt by the committee to clearly separate logic exceptions from runtime exceptions, thus providing a framework for further exception derivations.

These derived standard exceptions override the virtual function “*what*”, but do not add functions, except for the “filesystem error” exception, which defines functions with which to retrieve the error code (*code*) and the one or two paths (*path1*) (*path2*) with which it was constructed.



Quick Reference Guide: Exception-Related Functions

Library	Type	Function	Parameters	Description
<code><exception></code>	<code>exception_ptr</code>	<code>current_exception</code>	<code>()</code>	Make the pointer to currently handled exception or copy.
<code><exception></code>	<code>exception_ptr</code>	<code>template<class E> make_exception_ptr</code>	<code>(E)</code>	Make the pointer to exception object.
<code><exception></code>	<code>int</code>	<code>uncaught_exceptions</code>	<code>()</code>	Retrieve the number of outstanding uncaught exceptions.
<code><exception></code>	<code>void</code>	<code>rethrow_exception</code>	<code>(exception_ptr)</code>	Rethrow exception.
<code><exception></code>	<code>void</code>	<code>terminate</code>	<code>()</code>	Called upon abandoning exception handling. Calls current <code>terminate_handler</code> function.
<code><exception></code>	<code>void</code>	<code>unspecified</code>	<code>()</code>	The default <code>terminate_handler</code> calls <code>abort()</code> .
<code><exception></code>	<code>terminate_handler</code>	<code>set_terminate</code>	<code>(terminate_handler)</code>	Set <code>terminate_handler</code> function. Returns previous <code>terminate_handler</code> function.
<code><cstdlib></code>	<code>void</code>	<code>abort</code>	<code>()</code>	Aborts program ungracefully.

201 © Cadence Design Systems, Inc. All rights reserved.



The function “current exception”, returns an exception pointer to the currently handled exception object or copy thereof, or if none, then a null exception pointer object.

The function template “make exception pointer”, returns an exception pointer to the copy of the exception object passed by value.

The function “uncaught exceptions”, returns the number of uncaught exceptions in the current thread. An exception is uncaught if thrown and not yet completely handled.

The function “rethrow exception”, rethrows an exception object.

The function “terminate”, called directly by user code, or called by the implementation when exception handling cannot continue, calls the current terminate handler function.

The default unnamed terminate handler function calls the function “abort”.

You can define your own terminate handler.

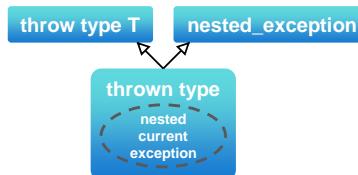
The function “abort” terminates the program ungracefully. It does not execute destructors and does not call functions registered to be called back at exit.

What Is a Nested Exception?



The class “**nested_exception**” is a polymorphic mix-in class in which you can capture the current exception, thus incrementally adding diagnostic information up the call chain.

Return Type	Member Function	Parameters	Description
<code>exception_ptr</code>	<code>nested_ptr</code>	<code>()</code>	Pointer to nested (captured) exception.
<code>void</code>	<code>rethrow_nested</code>	<code>()</code>	If nested exists rethrow nested else terminate.
Return Type	Non-Member Function	Parameters	Description
<code>void</code>	<code>template<class T> throw_with_nested</code>	<code>(T&&)</code>	Throws exception derived from <code>nested_exception</code> and <code>decay_t<T></code> and constructed from <code>forward<T>(t)</code>
<code>void</code>	<code>template<class E> rethrow_if_nested</code>	<code>(const E&)</code>	If E polymorphic and derived from <code>nested_exception</code> calls that base class function <code>rethrow_nested()</code>



202 © Cadence Design Systems, Inc. All rights reserved.



The class “**nested exception**” is a polymorphic mix-in class in which you can nest the current exception, thus incrementally adding diagnostic information up the call chain.

Without nested exceptions, a programmer might catch an exception at a lower level, and throw a new exception at a higher level, perhaps adding diagnostic information. The lower-level exception is lost, and only its information explicitly copied to the new exception still remains.

With nested exceptions, a programmer can nest a lower-level exception within a newly-thrown higher-level exception. All of the lower-level exception information is retained within the nested exception.

The member function “**nested pointer**” (`nested_ptr`), returns a pointer to the captured exception. You can use the pointer to access the captured exception’s members. To drill down further, you must discard the outermost layer, which you do by rethrowing the nested exception and catching it.

The member function “**rethrow nested**”, if an exception is nested within, rethrows it, else calls the “**terminate**” function.

The non-member function “**throw with nested**”, throws an exception of a type derived from both “**nested exception**” and the template class “**decay**” specialized for the argument type (`decay_t<T>`), and nesting within it the exception being currently handled.

The non-member function “**rethrow if nested**”, if the exception argument is polymorphic and derived from the “**nested exception**” class, calls the “**nested exception**” class function “**rethrow nested**”, to rethrow the exception.

The illustration is of the resulting thrown type when you throw an exception with the currently handled exception nested within. The thrown type inherits both the type that you threw, and the “**nested exception**” class, from which it gets the member functions for accessing the nested current exception.

Nested Exception Class Code

```
// <exception>
namespace std {
    class nested_exception {
        public:
            nested_exception() noexcept;
            nested_exception(const nested_exception&) noexcept = default;
            nested_exception& operator=(const nested_exception&) noexcept = default;
            virtual ~nested_exception() = default;
            [[noreturn]] void rethrow_nested() const; // Rethrow nested exception
            exception_ptr nested_ptr() const noexcept; // Pointer to nested exception
    };
    template<class T> [[noreturn]] void throw_with_nested(T&& t); // Embed t and throw
    template<class E> void rethrow_if_nested(const E& e); // Call e.rethrow_nested()
                                                               // if has embedded
}
```

203 © Cadence Design Systems, Inc. All rights reserved.



The class “nested exception” is a polymorphic mix-in class in which you can nest the current exception, thus incrementally adding diagnostic information up the call chain.

The class defines default and copy constructors, the copy assignment operator, a virtual destructor, and the functions “rethrow nested” and “nested pointer”. Other than “rethrow nested”, none of these functions is permitted to throw any type of object.

- The function “rethrow nested” rethrows the nested exception; and
- The function “nested pointer” returns a pointer to the nested exception.

The standard namespace defines:

- The function “throw with nested”, to nest an exception within a nested exception object, and throw the nested exception object; and
- The function “rethrow if nested”, to call the exception’s “rethrow nested” function if the exception argument is polymorphic and derived from the “nested exception” class.

Example: Using Nested Exceptions

```
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
using namespace std;

const unsigned levels = 3;

// Print exception what
void print_what (const exception &e)
{
    // Print exception what
    cerr << e.what();
    // Rethrow nested exception if any
    try { rethrow_if_nested(e); }
    // Catch nested exception and recurse
    catch (const exception &e)
    { cerr<<"("; print_what(e); cerr<<") ";
    }
}
```

```
void throw_it(unsigned level) {

try {
    if (level==levels)
        throw logic_error(to_string(level)); // innermost
    else
        throw_it(level+1); // recurse
}
catch (const exception &e) {
    if (level > 1) // nest and rethrow
        throw_with_nested(logic_error(to_string(level-1)));
    else
        { print_what_arg(e); cerr << "\n"; }
}
}

int main () {
    throw_it(1);
    return 0;
}
```

Output: 1(2(3))

204 © Cadence Design Systems, Inc. All rights reserved.



The example first defines the function “print what”, to recursively print the exception’s “what” string, from the outermost exception, to the innermost nested exception.

The example then defines the function “throw it”, to recursively nest and throw exceptions. The innermost frame, in the **try** block, throws the initial un-nested exception. Inner frames, in the **catch** block, nest and rethrow the caught exception. The outermost frame’s **catch** block, calls the “print what” function.

The “what” strings are a string representation of the nesting level. The innermost un-nested exception is here at the defined maximum level 3. It is nested in an exception at level 2, which is nested in an exception at level 1.

I chose a logic error exception simply because it offers a constructor that accepts a string initializer.

Module Summary

In this module, you

- Defined handling of “exceptional” conditions

This training module discussed:

- Introduction to Exceptions
- Throwing and Catching Exceptions
- Indicating Whether a Function May Throw
- Standard Exceptions
- Nested Exceptions

205 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Define graceful handling of “exceptional” conditions.

To help you to achieve your objective, this training module discussed:

- Introduction to Exceptions;
- Throwing and Catching Exceptions;
- Indicating Whether a Function May Throw;
- Standard Exceptions; and
- Nested Exceptions.

Quiz: Exceptions



Where is the **catch** block in relationship to the **try** block.



Suppose that an exception is caught three stack frames earlier than where it was thrown. Where does program execution resume?



What standard exception does the **new** operator throw upon failure?



How can you indicate that a function will not throw an exception?

Please pause here to consider these questions.



Lab

Lab 14-1 Throwing and Catching Exceptions

Objective:

- To gracefully handle “exceptional” conditions

For this lab, you:

- Define an exception class
- Throw an object of your exception class type upon an attempt to check an object into the pool that was not checked out of the pool
- Catch and display the exception object

207 © Cadence Design Systems, Inc. All rights reserved.

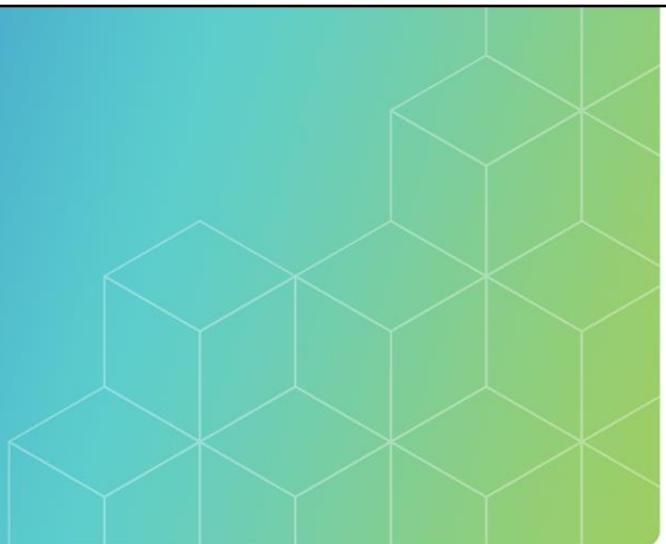
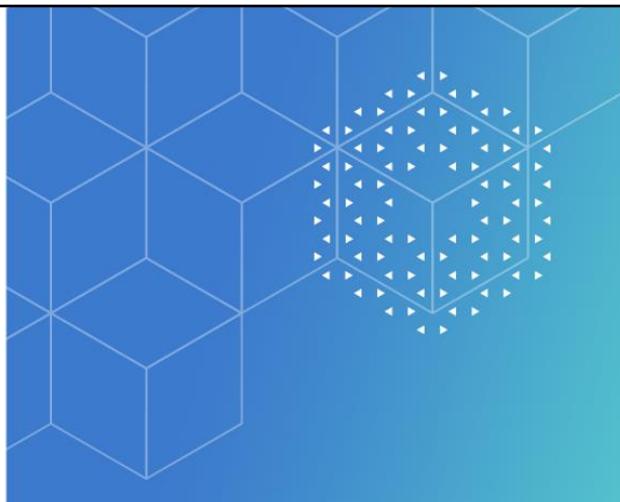


Your objective for this lab is to:

- Gracefully handle “exceptional” conditions.

For this lab, you:

- Define an exception class;
- Throw an object of your exception class type upon an attempt to check an object into the pool that was not checked out of the pool; and
- Catch and display the exception object.



Module 15

Input and Output

cadence®

This training module examines character-oriented I/O streams.

Module Objectives

In this module, you

- Input/output data to/from programs

This training module discusses:

- Input/Output Libraries
- Managing I/O State
- Opening and Closing Files
- Input/Output Formatting

209 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Input and output data to and from your programs.

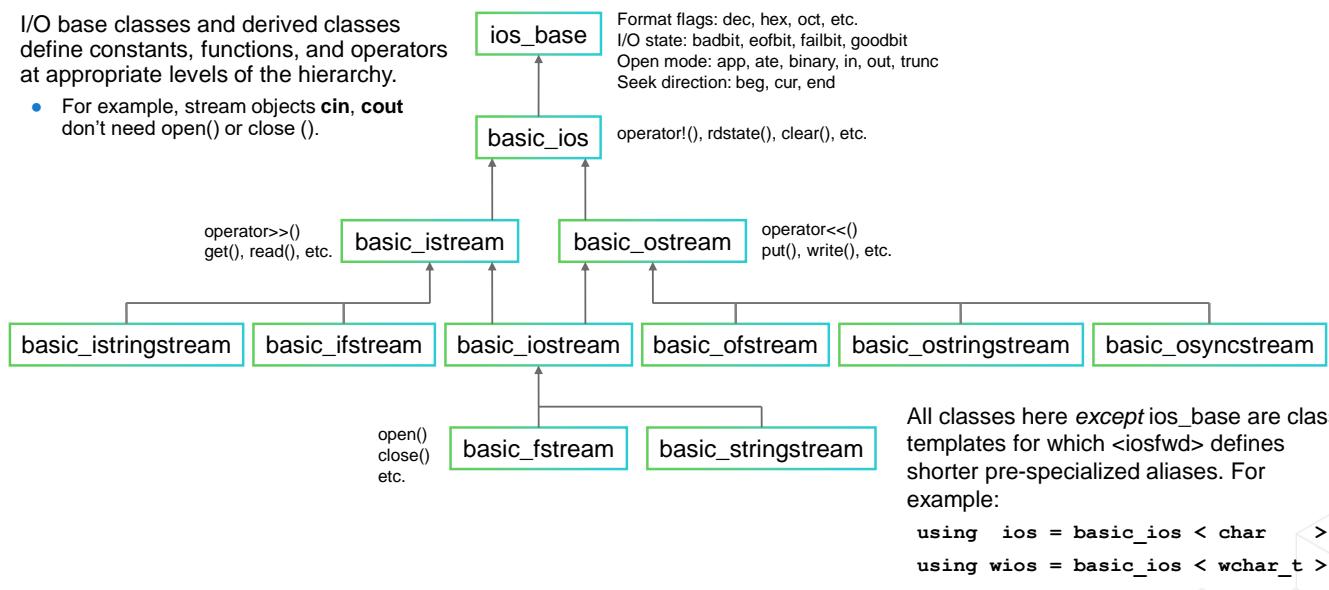
To help you to achieve your objective, this training module discusses:

- Input/Output Libraries;
- Managing I/O State;
- Opening and Closing Files; and
- Input/Output Formatting.

Input/Output Library Hierarchy

I/O base classes and derived classes define constants, functions, and operators at appropriate levels of the hierarchy.

- For example, stream objects `cin`, `cout` don't need `open()` or `close()`.



210 © Cadence Design Systems, Inc. All rights reserved.



Input and output base classes and derived classes define constants, functions, and operators at appropriate levels of the hierarchy. For stream input and output, you can almost always just use the “input-output stream” (`<iostream>`) library. For streams associated with files, you can simply use the “file stream” (`<fstream>`) library.

The class `ios_base` defines:

- Format flag constants, such as `boolalpha`, `dec`, `fixed`, `hex`, `internal`, `left`, `oct`, `right`, etc;
- I/O state constants such as `badbit`, `eofbit`, `failbit`, and `goodbit`;
- Open mode constants, such as `app`, `ate`, `binary`, `in`, `out`, and `trunc`;
- Seek direction constants, such as `beg`, `cur`, and `end`; and
- Functions to set and get the flags, such as `flags()`, `setf()`, `unsetf()`, `precision()`, and `width()`.

The class `ios`:

- Converts a stream object to a Boolean;
- Overloads the not ('!) operator; and
- Defines functions for setting and getting the I/O state, such as `rdstate()`, `clear()`, `setstate()`, `good()`, `eof()`, `fail()`, and `bad()`.

The class `istream`:

- Overloads the extraction operator for built-in and stream types;
- For unformatted input, defines the functions `get()`, `getline()`, `ignore()`, `peek()`, `read()`, `readsome()`, `putback()`, and `unget()`; and
- For file position manipulation, defines the functions `tellg()` and `seekg()`.

The class “output stream” (`ostream`):

- Overloads the insertion operator for built-in and stream types;
- For unformatted output, defines the functions `put()`, `write()`, and `flush()`; and
- For file position manipulation, defines the functions `tellp()` and `seekp()`.

The classes `ifstream` and `ofstream` define functions for opening and closing files.



Quick Reference Guide: Managing I/O State

Header	Class	Type	Function	Parameters	Description
<iostream>	basic_ios	iostate	rdstate	()	Read I/O state
<iostream>	basic_ios	void	clear	(iostate state=goodbit)	Make rdstate()==state
<iostream>	basic_ios	void	setstate	(iostate state)	clear(rdstate() state)
<iostream>	basic_ios	bool	good	()	rdstate()==0
<iostream>	basic_ios	bool	eof	()	rdstate() & eofbit
<iostream>	basic_ios	bool	bad	()	rdstate() & badbit
<iostream>	basic_ios	bool	fail	()	rdstate() & (badbit failbit)
<iostream>	basic_ios	bool	operator!	()	fail()
<iostream>	basic_ios	bool	operator bool	()	!fail()

ios_base::iostate	Description
ios_base::goodbit	0
ios_base::eofbit	Encountered EOF
ios_base::failbit	Operation failed
ios_base::badbit	Device failed

```

while (fin) // assume fstream object "fin"
    char c = fin.get();
    if (fin.bad())
        cerr << "device failed";
    else
        cerr << "op failed";

```

211 © Cadence Design Systems, Inc. All rights reserved.



Every I/O stream has a state bitmask with bad, fail, and EOF bits. An operation failure for any reason sets the fail bit. An operation failure due to an attempt to read past the end of the file also sets the EOF bit. A device failure also sets the bad bit. You can get and reset the flag as a whole, and you can get and reset individual bits.

The “ios” class defines functions for setting, getting, and testing the I/O state.

- The function “read state” (*rdstate*) returns the I/O state flag.
- The function “*clear*” sets the I/O state flag, by default to the “good bit” (*goodbit*) value.
- The function “*set state*” (*setstate*) sets individual bits of the I/O state flag.
- The function “*good*” returns the truth of whether no state flags are set.
- The function “*EOF*” (*eof*) returns the truth of whether the “EOF” flag is set.
- The function “*bad*” returns the truth of whether the “bad” flag is set.
- The function “*fail*” returns the truth of whether the “bad” flag is set or the “fail” flag is set.
- The not (!) operator overload allows you to use a stream reference as a Boolean expression. It returns true if the last operation failed, and false if it was successful. To promote readability, you might want to instead explicitly call the “*fail*” member function.
- The Boolean conversion operator allows you to use a stream reference as a Boolean expression. It returns false if the last operation failed, and true if it was successful. To promote readability, you might want to instead explicitly call the “*fail*” member function.

The example utilizes this conversion operator, and then directly calls the “*bad*” function.



Quick Reference Guide: Reading Stream Objects

Header	Class	Type	Function	Parameters	Description
<iostream>	basic_istream	istream&	operator>>	(built-in-types)	Extract element(s)
<iostream>	basic_istream	int_type	peek	()	Peek char
<iostream>	basic_istream	int_type	get	()	Get char
<iostream>	basic_istream	istream&	get	(char_type &c)	Get char
<iostream>	basic_istream	istream&	get	(char_type* s, streamsize n, char_type delim);	Get up to n-1 char, stop at delim and leave it, terminate with null
<iostream>	basic_istream	istream&	ignore	(streamsize n = 1, int_type delim = traits::eof());	Ignore up to n-1 char, stop at delim and discard
<iostream>	basic_istream	istream&	getline	(char_type *s, streamsize n, char_type delim='\\n');	Get up to n-1 char, stop at delim and discard it, terminate with null
<iostream>	basic_istream	istream&	read	(char_type *s, streamsize n);	Read binary up to n char
<iostream>	basic_istream	pos_type	tellg	()	Tell file position
<iostream>	basic_istream	istream&	seekg	(pos_type)	Seek to file position (absolute)
<iostream>	basic_istream	istream&	seekg	(off_type, ios_base::seekdir);	Seek to file position (relative)

ios_base::seekdir	Offset
ios_base::beg	Beginning
ios_base::cur	Current
ios_base::end	End

212 © Cadence Design Systems, Inc. All rights reserved.



For stream input, the “input stream” (*istream*) class:

- Overloads the extraction operator for extracting elements from the stream;
- For unformatted input, defines the functions “*peek*” to look at the next character, “*get*” to get one or more characters, “*ignore*” to ignore characters, “*get line*” (*getline*) to get a line, and “*read*” to read binary data; and
- For stream position manipulation, defines the functions “*tell get*” (*tellg*) to tell the stream position and “*seek get*” (*seekg*) to seek to a stream position. The seek can be to an absolute position, or to a position relative to either the current position or to the beginning or end of the stream.

Not displayed here are also the functions “*unget*” to put the previous got character back into the stream, “*put back*” (*putback*) to put any character back into the stream, and “*read some*” (*readsome*) to read some characters while disregarding delimiters.

basic_istream& unget();

basic_istream& put_back(Ch c);

streamsize readsome(Ch* p, streamsize n);



Quick Reference Guide: Writing Stream Objects

Header	Class	Type	Function	Parameters	Description
<ostream>	basic_ostream	ostream&	operator<<	(fundamental types and ostreams)	Insert element(s)
<ostream>	basic_ostream	ostream&	put	(char_type &c)	Put char
<ostream>	basic_ostream	ostream&	write	(const char_type *s, streamsize n);	Write binary n char
<ostream>	basic_ostream	ostream&	flush	()	Flush buffer
<ostream>	basic_ostream	pos_type	tellp	()	Tell stream position
<ostream>	basic_ostream	ostream&	seekp	(pos_type)	Seek stream position
<ostream>	basic_ostream	ostream&	seekp	(off_type, ios_base::seekdir);	Seek stream position

Manipulators	Action
std::ends	Calls os.put('\0')
std::endl	Calls os.put('\n'), os.flush()
std::flush	Calls os.flush()

ios_base::seekdir	Offset
ios_base::beg	Beginning
ios_base::cur	Current
ios_base::end	End



213 © Cadence Design Systems, Inc. All rights reserved.

For stream output, the “output stream” (*ostream*) class:

- Overloads the insertion operator for inserting elements into the stream;
- For unformatted output, defines the functions “*put*” to put a character, “*write*” to write binary data, and “*flush*” to flush the output buffer; and
- For stream position manipulation, defines the functions “*tell put*” (*tellp*) to tell the stream position and “*seek put*” (*seekp*) to seek to a stream position. The seek can be to an absolute position, or to a position relative to the current position or to the beginning or end of the stream.
- For output stream manipulation, the standard (*std*) namespace defines the functions “end string” (*ends*) to terminate a character string, “end line” (*endl*) to terminate a line, and “*flush*” to flush the output buffer.



Quick Reference Guide: Opening and Closing Files

Header	Class	Type	Function	Parameters	Description
<fstream>	basic_fstream	-	basic_fstream	(const char* s, ios_base::openmode mode = ios_base::in ios_base::out);	Constructor
<fstream>	basic_fstream	void	open	(const char* s, ios_base::openmode mode = ios_base::in ios_base::out);	Open file
<fstream>	basic_fstream	bool	is_open	()	Is open
<fstream>	basic_fstream	void	close	()	Close

basic_istream and basic_ostream duplicate these functions with the default open mode respectively only input and only output.

ios_base::openmode	Action
ios_base::in	Open for input
ios_base::out	Open for output
ios_base::app	Seek end before each write
ios_base::ate	Seek end upon opening
ios_base::trunc	Truncate upon opening
ios_base::binary	Binary mode (Unix ignores)

Examples

```
ofstream fout;
fout.open("output_file",ios::app);
ifstream fin;
fin.open("input_file");
fstream file("inout_file");
file.close();
```

214 © Cadence Design Systems, Inc. All rights reserved.



You can open a file either as you construct the stream object, or later with its function “open”.

You can determine whether a file is open on a stream, and you can close the file open on a stream.

An “input file stream” (*ifstream*) object is by default in the input mode, an “output file stream” (*ofstream*) object is by default in the output mode, and a “file stream” (*fstream*) object is by default in both the input and output modes.

Some of the other open modes are less obvious:

- The mode “*in*”, does not create the file if it does not exist, opens an existing file for input, and sets the read pointer to the file begin.
- The mode “*out*”, creates the file if it does not exist and the “*in*” mode is not included, opens the file for output, and sets the write pointer to the file begin.
- The mode “append” (*app*), creates the file if it does not exist, regardless of whether the “*in*” mode is included, opens the file for output, and sets the write pointer to the file end. The “append” mode first seeks to the file end upon each write operation, regardless of where you pre-position the write pointer. Including the “*out*” mode has no further effect.
- The mode “at end” (*ate*), initially positions the file pointers at the file end, and has no other effect.
- The mode “truncate” (*trunc*) truncates an existing stream upon opening. You cannot include the “truncate” mode when opening a file with the “append” mode or for only input and not output.
- The mode “*binary*” applies to operating systems that differentiate between binary files and text files.

The standard leaves it to the implementation to determine exactly what the open modes mean.

The example:

- 1st – Constructs an output file stream object, and for it opens an output file for appending;
- 2nd – Constructs an input file stream object, and for it opens an input file; and
- 3rd – Constructs a file stream object, simultaneously opening a file for input and output, and then closes the file.

Examples of Textual I/O Application Code

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    cout << "Enter string: " << flush;
    char string[81];
    cin >> string; // No white space!
    ofstream ofile;
    ofile.open("myfile");
    for (int i=0; i<=80; ++i) {
        ofile.put(string[i]);
        if (string[i]=='\0') break;
    }
    ofile.close();
    ifstream ifile;
    ifile.open("myfile");
    while (ifile) {
        char ch;
        ifile.get(ch);
        if (ch=='\0') break;
        cout << ch;
    }
    cout << endl;
    ifile.close();
    return 0;
}
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    fstream file("myfile",ios_base::in
                |ios_base::out
                |ios_base::trunc);
    file << "some output" << endl;
    file.seekp(0,ios_base::beg);
    while (file) { // !fail()
        char line[81];
        file.getline(line,81);
        cout << line << endl;
    }
    file.close();
    return 0;
}
```

215 © Cadence Design Systems, Inc. All rights reserved.



Textual I/O is sensitive to the delimiter character.

The left example:

- Inserts a character string into the “character output” (**cout**) stream, and without a new line, flushes the buffer;
- Extracts a character string from the “character input” (**cin**) stream;
- Individually writes the character string characters to a file named “my file”; and
- Individually reads the character string characters from the file “my file”, and inserts them into the “character output” stream.

The right example:

- Opens a file named “my file” for reading and writing, truncates it if it already exists, and creates it if it does not already exist;
- Inserts some output into the file stream;
- Repositions the “put” pointer to the beginning of the file; and
- Individually reads lines from the file and inserts them into the “character output” stream.

The **get()** member function is available in three forms. The form displayed here gets the one next character from the input stream into the character referenced, and returns the current object. Another form without parameters returns the one next character as an int. A third form reads up to n-1 characters or to a terminating character, into a character array, appends a null character, and returns a reference to the current object. It does not extract the terminating character.

The **getline()** member function reads up to n-1 characters or to a terminating character, into a character array, appends a null character, and returns a reference to the current object. It extracts and discards the terminating character.

Example: Binary I/O Application Code

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

int main() {
    MyData_s data;           // Assume struct MyData_s { ... };
    fstream file("myfile",ios::in|ios::out);           // open
    file.read( (char*) &data, sizeof(data));           // read
    message(data);           // operate
    file.seekp(0,ios_base::end);           // sync
    file.write( (char*) &data, sizeof(data));           // write
    file.close();           // close
    return 0;
}
```

216 © Cadence Design Systems, Inc. All rights reserved.



Binary I/O has no notion of a delimiter character.

This example:

- Opens a file named “my file” for input and output;
- Reads binary data from the file into a structure;
- Calls a function to presumably modify the structure data;
- Repositions the “put” pointer to the end of the file;
- Writes binary data from the structure into the file; and
- Closes the file.

The read() member function reads up to n characters from the input stream into a character array, and returns a reference to the current object.

The write() member function writes n characters to the output stream from a character array, and returns a reference to the current object.

Between the read and write operations the implementation requires a positioning operation.

What Is Stream Formatting?



Stream Formatting is setting a format flag or parameter that formats stream expression (interpretation) of inserted (extracted) stream elements.

For example: signage, radix, whitespace, character case, width, notation, precision ...

- You specify the formatting by setting format flags and by calling the functions `fill()`, `precision()`, and `width()`.
- The stream insertion (extraction) operators accordingly format the expressed (interpreted) stream.

```
cout.setf(ios::scientific, // scientific notation
          ios::floatfield);
cout.setf(ios::right,      //right justify (default)
          ios::adjustfield);
cout.fill('_')            // fill with underscore
cout.precision(3)         // 3 digits precision
cout.width(10)            // minimum width 10
cout << (float)3.14       // value 3.14 cast to float
                           << '\n';
cout.flush();              // Result: _3.140e+00
```

Stream Formatting is setting a format flag or parameter that formats expression of inserted elements, or interpretation of extracted elements.

You specify the formatting by setting format flags, and by calling the functions “*fill*”, “*precision*”, and “*width*”.

The stream insertion or extraction operators accordingly format the expressed or interpreted stream.

The example sets the standard character-based output (`cout`) stream formatting, and then inserts into the stream a floating-point value and a new line character, and flushes the output buffer.



Quick Reference Guide: Formatting Functions

Header	Class	Type	Function	Parameters	Description
<iostream>	ios_base	fmtflags	flags	() (fmtflags fmtfl)	Get format flags Set format flags
<iostream>	ios_base	fmtflags	setf	(fmtflags fmtfl) (fmtflags fmtfl, fmtflags mask)	Get format flag bits Set format flags bits
<iostream>	ios_base	void	unsetf	(fmtflags mask)	Clear mask
<iostream>	ios_base	streamsize	precision	() (streamsize prec)	Get precision Set precision
<iostream>	ios_base	streamsize	width	() (streamsize wide)	Get minimum field width Set minimum field width
<iostream>	basic_ios	char_type	fill	() (char_type ch)	Get fill character Set fill character

In most cases, setting the width applies to only the one next inserted or extracted object.



Format flags determine how stream output is displayed and stream input is interpreted. Most of these functions have two versions. Calling the appropriate functions without an argument, is *querying* the format flags, precision, width, or fill character. Calling the appropriate functions with an argument, is *setting* the format flags, precision, width, or fill character. In most cases, setting the width applies to only the one next insertion or extraction, which then resets to the default width.

- The function “flags”, returns the format flags, and if provided with an argument, sets the format flags to that provided as the argument.
- The functions “set flags” (*setf*), return the format flags, and if provided with an argument, logically OR the argument into the format flags, and if provided with a second mask argument, replaces that masked field of the format flags with that masked field of the argument.
 - For those flags that require a mask, *always* use the two-argument “set flags” function to ensure that the other mutually-exclusive flag bits are cleared.
- The function “unset flags” (*unsetf*), clears the masked field of the format flag.
- The function “precision”, gets or sets the precision for insertion operations. The precision value is by default 6 decimal digits of precision.
- The function “width”, gets or sets the width for the next insertion or extraction operation. The width value is by default 0, which means to use the default width for the operation.
- The function “fill”, gets or sets the fill character for when the width is more than required for the output operation. The fill character value is by default the space character.



Quick Reference Guide: Format Flag Descriptions

Header	Class	Type	Flag	Mask	Mode	Description
<iostream>	ios_base	fmtflags	internal, left, right	adjustfield	O	Where to add fill
<iostream>	ios_base	fmtflags	dec, oct, hex	basefield	I/O	How to show integers
<iostream>	ios_base	fmtflags	fixed, scientific	floatfield	O	How to show floating point
<iostream>	ios_base	fmtflags	boolalpha		I/O	Bool as “true” or “false”
<iostream>	ios_base	fmtflags	showbase		O	Show numeric base
<iostream>	ios_base	fmtflags	showpoint		O	Show float decimal always
<iostream>	ios_base	fmtflags	showpos		O	Show + if positive
<iostream>	ios_base	fmtflags	skipws		I	Skip white space
<iostream>	ios_base	fmtflags	unitbuf		O	Flush each operation
<iostream>	ios_base	fmtflags	uppercase		O	Show radix/values uppercase

219 © Cadence Design Systems, Inc. All rights reserved.



Format flag “adjust field” (*adjustfield*) is a bit-wise OR of the flags “internal”, “left”, and “right”. The function “set flag” (*setf*) uses this value as a mask to first clear all the adjust-field bits, before setting the one specified adjust-field bit.

- Format flag “internal” adds fill characters at a designated internal point. For numerical values, the fill characters appear after the signage and radix and before the numerical value. For non-numerical values, or if no sign or radix is displayed, this format behaves identically to format flag “right”.
- Format flag “left” left-justifies the output and adds fill characters on the right.
- Format flag “right” right-justifies the output and adds fill characters on the left.

Format flag “base field” (*basefield*) is a bit-wise OR of the flags “decimal”, “octal”, and “hexadecimal”. The “set flag” (*setf*) function uses this value as a mask to first clear all the base-field bits, before setting the one specified base-field bit.

- Format flag “decimal” (*dec*) inputs or outputs integral values in base decimal.
- Format flag “octal” (*oct*) inputs or outputs integral values in base octal.
- Format flag “hexadecimal” (*hex*) inputs or outputs integral values in base hexadecimal.

Format flag “float field” (*floatfield*) is a bit-wise OR of the flags “fixed” and “scientific”. The “set flag” (*setf*) function uses this value as a mask to first clear all the float-field bits, before setting the one specified float-field bit.

- Format flag “fixed” outputs floating-point values in fixed-point notation.
- Format flag “scientific” outputs floating-point values in scientific notation.

Format flag “Boolean alphabetic” (*boolalpha*) inputs or outputs Boolean values in alphabetic format.

Format flag “show base” (*showbase*) outputs integral values with a radix prefix.

Format flag “show point” (*showpoint*) outputs floating point values with a decimal character.

Format flag “show positive” (*showpos*) outputs non-negative numbers with a plus (+) sign.

Format flag “skip whitespace” (*skipws*) skips leading whitespace before some input operations.

Format flag “unit buffer” (*unitbuf*) flushes the output buffer after each output operation.

Format flag “uppercase” (*uppercase*) outputs radix and values in upper case.

Example: Output Stream Formatting Application Code

```
#include <iostream>
using namespace std;
int main () {

    cout.setf(ios::boolalpha);    cout << (bool)0 << endl;    // false
                                cout << (bool)1 << endl;    // true
    cout.unsetf(ios::boolalpha);  cout << (bool)1 << endl;    // 1
    cout.setf(ios::showpos);     cout << (int)33 << endl;    // +33
                                cout << (int)55 << endl;    // +55
    cout.unsetf(ios::showpos);   cout << (int)77 << endl;    // 77

    cout.setf(ios::scientific, ios::floatfield);
    cout.precision(3); cout.width(10);           // width applies once!
    cout << (float)3.14 << endl;                // 3.140e+00 (width 10)
    cout << (float)1.67 << endl;                // 1.670e+00
    cout.setf(ios::fixed, ios::floatfield);
    cout << (float)0.99 << endl;                // 0.990

    cout.setf(ios::showbase|ios::uppercase);
    cout.setf(ios::hex,ios::basefield);
    cout << (unsigned)255 << endl;              // 0xFF
    cout.unsetf(ios::uppercase);
    cout << (unsigned)0x3F << endl;              // 0xef
    cout.setf(ios::oct, ios::basefield);
    cout.unsetf(ios::showbase);
    cout << (unsigned)0xAFc << endl;              // 5374
    return 0;
}
```

220 © Cadence Design Systems, Inc. All rights reserved.



The example illustrates the use of some stream formatting.

- For the Boolean output, it outputs values with and without the “Boolean alphabetical” (*boolalpha*) flag.
- For the integral output, it outputs values with and without the “show positive” (*showpos*) flag.
- For the floating-point output, it sets the format to scientific, the precision to three digits after the decimal point, and the minimum field width to 10. The minimum field width automatically resets to 0 after inserting the subsequent value, so you should see an extra fill character, by default a space character, before the first floating-point number, but not before the second. The example then outputs a floating-point value in fixed-point format.
- For another integral output, the example first outputs a value in upper-case hexadecimal with the radix prefix, then outputs a value in lower-case hexadecimal with the radix prefix, and then outputs a value in lower-case octal without the radix prefix.

What Is a Stream Manipulator?



A **Stream Manipulator** is a stream insertion (extraction) operation term that manipulates stream expression (interpretation) of other terms of the operation.

For example: signage, radix, whitespace, character case, width, notation, precision ...

- You enter the manipulators as terms in the stream insertion (extraction) operation.
- The manipulators participate in stream composition (chaining) by taking and returning a stream reference.
- Overloaded stream insertion (extraction) operators take these stream references and accordingly format the expressed (interpreted) stream.

```
cout << scientific      // scientific notation
<< right                // right justify (default)
<< setfill('_')          // fill with underscore
<< setprecision(3)       // 3 digits precision
<< setw(10)              // minimum width 10
<< (float)3.14           // value 3.14 cast to float
<< endl;                 // Result: _3.140e+00
```

A *Stream Manipulator* is a stream insertion or extraction operand that manipulates stream expression or interpretation of other operands.

You enter the manipulators as operands in the stream insertion or extraction operation.

The manipulators participate in stream composition, that is, operation chaining, by taking and returning a stream reference.

Overloaded stream insertion or extraction operators take these stream references and accordingly format the expressed or interpreted stream.

The example inserts into the standard character-based output (**cout**) stream a selection of manipulators to format a floating-point value, and then a final manipulator to insert a newline character and flush the output buffer.



Quick Reference Guide: Stream Manipulators

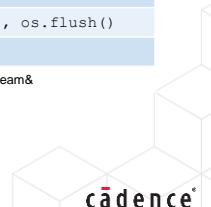
Manipulators <iostream>	Mode	Equivalent Function Call	
<code>internal, left, right</code>	O	<code>str.setf(ios_base::internal, ios_base::adjustfield)</code>	etc.
<code>dec, oct, hex</code>	I/O	<code>str.setf(ios_base::dec, ios_base::basefield)</code>	etc.
<code>fixed, scientific, defaultfloat</code>	O	<code>str.setf(ios_base::fixed, ios_base::floatfield)</code>	etc.
<code>boolalpha, noboolalpha</code>	I/O	<code>str.setf(ios_base::boolalpha)</code>	<code>str.unsetf(ios_base::boolalpha)</code>
<code>showbase, noshowbase</code>	O	<code>str.setf(ios_base::showbase)</code>	<code>str.unsetf(ios_base::showbase)</code>
<code>showpoint, noshowpoint</code>	O	<code>str.setf(ios_base::showpoint)</code>	<code>str.unsetf(ios_base::showpoint)</code>
<code>showpos, noshowpos</code>	O	<code>str.setf(ios_base::showpos)</code>	<code>str.unsetf(ios_base::showpos)</code>
<code>skipws, noskipws</code>	I	<code>str.setf(ios_base::skipws)</code>	<code>str.unsetf(ios_base::skipws)</code>
<code>unitbuf, nounitbuf</code>	O	<code>str.setf(ios_base::unitbuf)</code>	<code>str.unsetf(ios_base::unitbuf)</code>
<code>uppercase, nouppercase</code>	O	<code>str.setf(ios_base::uppercase)</code>	<code>str.unsetf(ios_base::uppercase)</code>

These manipulators are functions taking and returning an `ios_base`&

Manipulators <iomanip>	Mode	Equivalent Function Call	Manipulators <ostream>	Equivalent Function Call
<code>setprecision(n)</code>	I/O	<code>str.precision(n)</code>	<code>ends</code>	<code>os.put('\0')</code>
<code>setbase(n)</code>	I/O	<code>str.setf(ios_base::oct) etc.</code>	<code>endl</code>	<code>os.put('\n'), os.flush()</code>
<code>setfill(c)</code>	O	<code>str.fill(c)</code>	<code>flush</code>	<code>os.flush()</code>
<code>setw(n)</code>	I/O	<code>str.width(n)</code>	These manipulators are functions taking and returning a <code>basic_ostream</code> &	
<code>setiosflags(mask)</code>	I/O	<code>str.setf(mask)</code>		
<code>resetiosflags(mask)</code>	I/O	<code>str.setf(ios_base::fmtflags(0), mask)</code>		

These manipulators are functions returning an implementation-defined type.

222 © Cadence Design Systems, Inc. All rights reserved.



The standard defines functions, termed “manipulators”, that you place directly in the chain of extraction or insertion operations. The manipulators almost exactly duplicate format flag features.

- Manipulator “*internal*” adds fill characters at a designated internal point. For numerical values, the fill characters appear after the signage and radix and before the numerical value. For non-numerical values, or if no sign or radix is displayed, this format behaves identically to manipulator “*right*”.
- Manipulator “*left*” left-justifies the output and adds fill characters on the right.
- Manipulator “*right*” right-justifies the output and adds fill characters on the left.
- Manipulator “decimal” (*dec*) inputs or outputs integral values in base decimal.
- Manipulator “octal” (*oct*) inputs or outputs integral values in base octal.
- Manipulator “hexadecimal” (*hex*) inputs or outputs integral values in base hexadecimal.
- Manipulator “fixed” outputs floating-point values in fixed-point notation.
- Manipulator “scientific” outputs floating-point values in scientific notation.
- Manipulator “default float” (*defaultfloat*) clears both manipulators “fixed” and “scientific”.
- Manipulator “Boolean alphabetic” (*boolalpha*) inputs or outputs Boolean values in alphabetic format.
- Manipulator “show base” (*showbase*) outputs integral values with a radix prefix.
- Manipulator “show point” (*showpoint*) outputs floating point values with a decimal character.
- Manipulator “show positive” (*showpos*) outputs non-negative numbers with a plus (+) sign.
- Manipulator “skip whitespace” (*skipws*) skips leading whitespace before some input operations.
- Manipulator “unit buffer” (*unitbuf*) flushes the output buffer after each output operation.
- Manipulator “uppercase” (*uppercase*) outputs radix and values in upper case.

This works because the stream extraction (`<<`) and insertion (`>>`) operators are overloaded to also accept as an argument a pointer to a function returning the `ios_base` type or the `basic_ios` type or the `basic_ostream/basic_istream` types.

You include the `<iostream>` header to obtain definitions of the manipulators here that do not take arguments.

You include the `<iomanip>` header to obtain definitions of the manipulators here that do take arguments.

You include the `<ostream>` or `<iostream>` header to obtain definitions of the manipulators `ends`, `endl`, and `flush`.

Example: Output Stream Manipulators Application Code

```
#include <iostream>
#include <iomanip>
using namespace std;
int main () {

    cout << " " << boolalpha << (bool)0 // false
    << " " << (bool)1 // true
    << " " << noboolalpha << (bool)1 // 1
    << endl ;

    cout << " " << showpos << (int)33 // +33
    << " " << (int)55 // +55
    << " " << noshowpos << (int)77 // 77
    << endl ; // width once!

    cout << " " << scientific << setprecision(3) << setw(10) << (float)3.14 // 3.140e+00
    << " " << (float)1.67 // 1.670e+00
    << " " << fixed << (float)0.99 // 0.990
    << endl ;

    cout << " " << showbase << hex << uppercase << (unsigned)255 // 0xFF
    << " " << nouppercase << (unsigned)0357 // 0xef
    << " " << noshowbase << oct << (unsigned)0xAFC // 5374
    << endl ;
    return 0 ;
}
```

223 © Cadence Design Systems, Inc. All rights reserved.



The example illustrates the use of some stream manipulators.

- For the Boolean output, it outputs values with and without the “Boolean alphabetical” (*boolalpha*) flag.
- For the integral output, it outputs values with and without the “show positive” (*showpos*) flag.
- For the floating-point output, it sets the format to scientific, the precision to three digits after the decimal point, and the minimum field width to 10. The minimum field width automatically resets to 0 after inserting the subsequent value, so you should see an extra fill character, by default a space character, before the first floating-point number, but not before the second. The example then outputs a floating-point value in fixed-point format.
- For another integral output, the example first outputs a value in upper-case hexadecimal with the radix prefix, then outputs a value in lower-case hexadecimal with the radix prefix, and then outputs a value in lower-case octal without the radix prefix.

Example: Transaction Recording File Output [1/2]

```
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <fstream>
#include <iomanip>
using namespace std;

// Address and data ranges
const unsigned A[] = {0,1023};
const unsigned D[] = {0,255};

// Forward reference
void write_trans(unsigned addr,
                 unsigned data);

int main() {
    srand((unsigned)time(0));
    for (int i=0; i<=9; i++)
        write_trans(rand()% (A[1]-A[0]+1)+A[0],
                    rand()% (D[1]-D[0]+1)+D[0]);
    return 0;
}
```

```
// Transaction structure
struct Trans {
    const char *phase;
    unsigned value;
    Trans *next;
};

// Forward reference
void record_trans(const char *filename,
                  const char *trans_type,
                  Trans *trans_ptr);

void write_trans(unsigned addr,
                 unsigned data) {
    Trans *trans = new Trans;
    trans->next = new Trans;
    trans->phase = "Addr";
    trans->next->phase = "Data";
    trans->value = addr;
    trans->next->value = data;
    trans->next->next = NULL;
    // ... some code to send transaction ...
    record_trans("transrecord.txt",
                 "Write", trans);
    delete trans->next; delete trans;
}
```

224 © Cadence Design Systems, Inc. All rights reserved.



This is an example of a transaction recording application. The code is presented in a top-down manner so that you will understand its purpose before you examine its details.

- On the left, the function “main”, in a loop, calls the function “write transaction”, multiple times with constrained randomized address and data arguments.
- On the right, the function “write transaction”, constructs address and data transaction objects, populates their phase and value fields, and after presumably communicating the transaction, calls the function “record transaction”, with the arguments “file name”, “transaction type”, and “transaction”.

Example: Transaction Recording File Output [2/2]

```

// / Write \
// .....
// Addr :     801
// Data :      43
// .....
void record_trans ( const char *filename,
                     const char *trans_type,
                     Trans *trans_ptr_a) {
    ofstream stream ( filename, ios::app );
    stream << " " << setfill('_') << setw(strlen(trans_type)+2) << "" << endl; // / Write \
    stream << "/" << trans_type << "\\" << endl; // / Write \
    stream << "....." << endl; // .....
    Trans *trans_ptr = trans_ptr_a;
    do {
        stream << trans_ptr->phase << " : "
            << right << setfill(' ') << setw(10) << trans_ptr->value
            << endl;
    while ((trans_ptr=trans_ptr->next) != NULL);
    stream << "....." << endl; // .....
    stream.close();
}

```

225 © Cadence Design Systems, Inc. All rights reserved.



The function “record transaction”:

- Accepts arguments for “file name”, “transaction type”, and “transaction”;
- Opens the file for appending;
- Formats and outputs the transaction information to the file; and
- Closes the file.

The formatting involves setting the fill character and field width and right-justifying the address and data values.

Module Summary

In this module, you

- Input/outputted data to/from programs

This training module discussed:

- Input/Output Libraries
- Managing I/O State
- Opening and Closing Files
- Input/Output Formatting

226 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Input and output data to and from your programs.

To help you to achieve your objective, this training module discussed:

- Input/Output Libraries;
- Managing I/O State;
- Opening and Closing Files; and
- Input/Output Formatting.

Quiz: Input and Output



The `fail()` function reveals whether the stream operation failed for any reason. How would determine more precisely why the operation failed?



The **istream** and **ostream** classes overload extraction and insertion operators (respectively) for convenient somewhat formatted I/O, but what functions would you use for binary reads and writes?



Opening a file for a **fstream** object sets the open mode to by default **ios::in|ios::out**. What does this imply for the initial get and put positions?



As well as a bitmask format flag, streams also have functions to specify the precision, width, and fill character. Which of these automatically reverts to the default setting after each use?

Please pause here to consider these questions.



Lab

Lab 15-1 Inputting and Outputting Program Data

Objective:

- To input/output data to/from programs

For this lab, you:

- Define an ostream insertion operator for objects of type pointer-to-pool
- Define an ostream insertion operator for objects of type pointer-to-node
- Define a new **void** function to display the pool contents by inserting the pool pointer into the **cout** output stream

228 © Cadence Design Systems, Inc. All rights reserved.

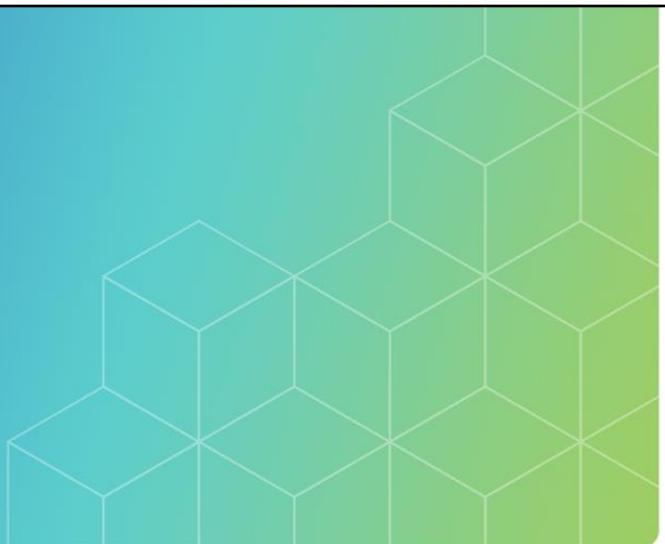
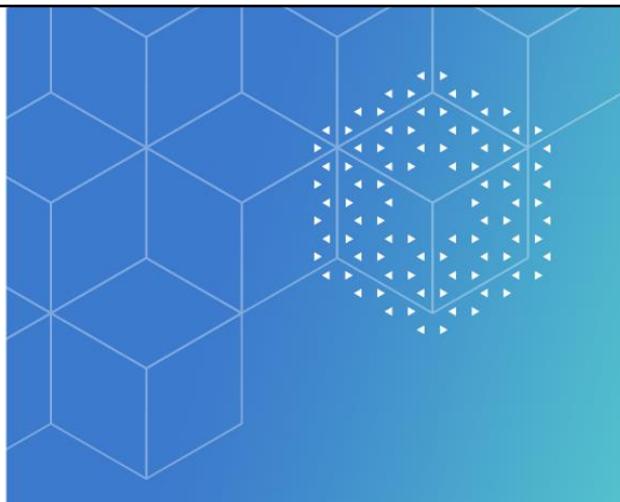


Your objective for this lab is to:

- Input/output data to/from programs.

For this lab, you:

- Define an “output stream” (*ostream*) insertion operator for objects of type pointer-to-pool;
- Define an “output stream” (*ostream*) insertion operator for objects of type pointer-to-node; and
- Define a new void function that displays the pool contents by inserting the pool pointer into the “character-based output” (**cout**) stream.



Module 16

Debugging

cadence®

This training module just briefly examines program debugging.

Module Objectives

In this module, you

- Diagnose incorrect program behavior

This training module discusses:

- What Is a Bug?
- What Is Debugging?
- Approaches to Debugging
- Debugging with GDB

230 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Diagnose incorrect program behavior.

To help you to achieve your objective, this training module discusses:

- What Is a Bug?
- What Is Debugging?
- Approaches to Debugging; and
- Debugging with GDB.

What Is a Bug?



A **Bug** is a source code defect manifest at run time[†].

[†] For this definition we ignore compiler-reported grammatical errors.

Bugs occur due to primarily two reasons.

- Programmer misunderstands the specification
- Programmer mis-implements the specification

Bugs present themselves in primarily three ways.

- A hardware exception, if not caught by software
 - For example, a segmentation fault
- A software exception, if the software checks itself
- Unexpected program output

110100100 101001001 010010011
001001101 010011010 100110100
011010010 110100100 101001001
100100110 001001101 010011010
001101001 010010010 110100100
010010011 001 0 101 001001101
100110101 001 01 110100101
10100100 010100110 001001100
01001101 101 01 01 010101001
110100100 010011010 010010011
001001101 010011010 101010100
011010010 110100100 101 01001
100100110 001001101 010011010
001101001 011010010 110100100



231 © Cadence Design Systems, Inc. All rights reserved.

A **Bug** is a source code defect manifest at run time. For this definition, we ignore compiler-reported grammatical errors, for which the compiler directly identifies the offending code.

Bugs occur due to primarily two reasons. Either:

- The programmer misunderstands the specification; or
- The programmer mis-implements the specification.

Bugs present themselves in primarily three ways.

- A hardware exception, if not caught by software, for example, a segmentation fault;
- A software exception, if the software checks itself; or
- Unexpected program output.

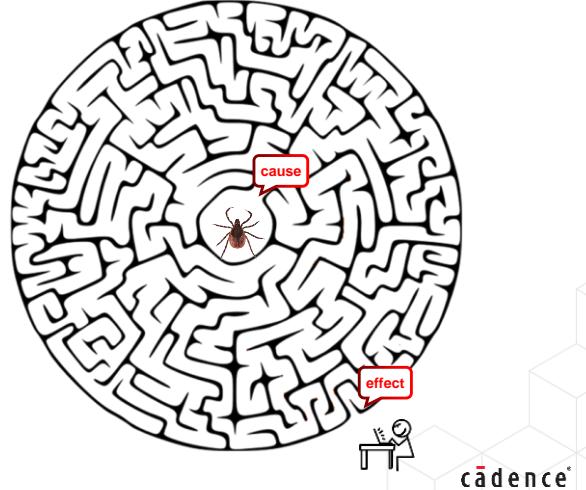
What Is Debugging?



Debugging is methodically finding and removing bugs, and confirming their removal.

The generic bug-handling flow is this:

- The programmer (or user!) encounters unexpected program behavior.
- The programmer, applying the exposing inputs, confirms the presence of the unexpected program behavior.
- The programmer locates and corrects the culprit software segment.
 - Understands the bug – links cause to effect.
 - Select the best of potentially multiple corrections.
 - Implements the selected correction.
- The programmer, applying the exposing inputs, confirms the absence of the unexpected program behavior.
- The programmer confirms that the correction does not cause other unexpected program behaviors.
 - This is termed “regression testing”.



232 © Cadence Design Systems, Inc. All rights reserved.

Debugging is methodically finding and removing bugs, and confirming their removal.

The overall bug-handling flow is this:

- 1st – The programmer or user encounters unexpected program behavior;
- 2nd – The programmer, applying the exposing inputs, confirms presence of the unexpected program behavior;
- 3rd – The programmer locates and corrects the culprit software segment. This is in three steps: To understand the bug, that is, to conclusively link cause to effect; To select the best of potentially multiple corrections; and to implement the selected correction.
- 4th – The programmer, applying the exposing inputs, and perhaps also other tests, confirms absence of the unexpected program behavior; and
- 5th – The programmer confirms that the correction does not cause other unexpected program behaviors. This is termed “regression testing”, as the test confirms that the change does not cause the program behavior to regress.

The Debugging Effort

Debugging is “hard” because bugs like to “hide”.

- Only a specific hardware and/or software platform exposes the bug.
- Only a specific input combination or sequence exposes the bug.
 - The specific input combination or sequence may be due to infrequent random (and perhaps not repeatable!) factors.
- The bug observation may be far removed in text and time from the bug execution.

You can choose from two debugging strategies.

- Manual code inspection – problematic!
 - Volume of code.
 - Your inspection is subject to the same assumptions as your authoring was subject to.
 - Other people have other things to do, so may give your code only a perfunctory nod.
- Code execution:
 - What debugging is all about!
 - Modify inputs as needed to make errors maximally reproducible.
 - Narrow the debug scope by making and testing a sequence of hypotheses eliminating potential causes.



Debugging takes effort because the bug existence is likely not obvious.

- Perhaps only a specific hardware and/or software platform exposes the bug;
- Perhaps only a specific input combination or sequence exposes the bug;
 - That specific input combination or sequence may be due to infrequent random factors that might not be reliably repeated.
- The bug observation may be far removed in text and time from the bug execution.

Debugging strategies include:

- Firstly, manual code inspection, but this is problematic, due to:
 - The volume of code to inspect;
 - That your inspection is subject to your same assumptions that your authoring was subject to; and
 - That other people have other things to do, so may give your code only a perfunctory nod.
- Secondly, to execute the code.
 - This is what debugging is all about!
 - First, you modify the inputs as needed to make the error maximally reproducible;
 - Then, you iteratively narrow the debug scope, by making and testing a sequence of hypotheses eliminating potential causes.

Best Practice: Avoid Debugging



The best debugging strategy is to avoid the need to debug!

Don't you create errors

- Don't program while you are below your par
- Research and avoid common programming mistakes
- Utilize best programming practices
 - Prefer standard library code wherever practical
 - Maximally constrain types and loosen only as needed
 - Code simply rather than complexly
 - Factor functions sufficiently to examine without scrolling
 - Comment judiciously
 - *At least 100 more...*
- Employ static and dynamic code analysis tools

Don't let the user create errors

- Program defensively – assume the user will imaginatively try to break the application

Find your errors early and often

- Test your program unit continually
 - After each significant change
- Test regression daily



234 © Cadence Design Systems, Inc. All rights reserved.

The best debugging strategy is to avoid the need to debug.

That starts with you not creating errors. For example:

- Don't program while you are in any way below your par, either stressed or tired or “below the weather” in any way;
- Research and avoid common programming mistakes, such as the failure to initialize data;
- Research and comply with industry-accepted best programming practices; and
- Employ static and dynamic code analysis tools.

Next, don't let the user create errors. For example:

- Program defensively, by assuming that the user will imaginatively try to break the application, perhaps by providing inputs of unexpected values and at unexpected times.

Finally, find your errors early and often.

- Test your program unit continually, after each significant change; and
- Test program regression daily.

Approaching the Debug Effort



Here we suggest 5 tactics to your debug efforts, generally from the simplest to the more complex.

1. Selectively bypass suspect code segments
2. Code statements issuing debug messages
3. Code statements asserting invariant truths
4. Utilize a logger
5. Utilize a source-level debugger



Here we suggest 5 approaches to your debug efforts, generally from the simplest to the more complex.

Those tactics are, in order, to:

- 1st – Selectively bypass suspect code segments;
- 2nd – Code statements issuing debug messages;
- 3rd – Code statements asserting invariant truths;
- 4th – Utilize a logger; and
- 5th – Utilize a source-level debugger.

Tactic 1: Bypass Suspect Code Segments

If you suspect a certain code segment of harboring a bug, perhaps you can temporarily “patch around” that segment, perhaps by replacing it with a less efficient but better understood algorithm.

```
#include <random>
#include <cstdlib>
...
mt19937 gen ( 42 ) ;
uniform_int_distribution<unsigned> dist(0,255);
srand ( 42 ) ;
...
//unsigned i = dist ( gen ) ;
unsigned i = rand()%256 ;
```



The 1st tactic is to selectively bypass suspect code segments.

Here, the programmer imagines that they incorrectly utilize the random generator, so temporarily replaces it with an inferior algorithm with which they are more comfortable.

Tactic 2: Issue Debug Messages

This tried-and-true tactic involves surrounding a suspect code segment with message-issuing statements advising you of their execution and outputting useful values.

Good practices include to:

- Print sufficient information to support debugging efforts.
- Provide means to link output back to issuing a statement.
- Conditionalize message issuance with verbosity level or conditional compilation.
- Send messages to **cerr** rather than **cout** or be sure to flush the message output!

Issues to consider include such temporary statements:

- Clutter the source code and program output.
- Provide the opportunity to inject yet more bugs!
- If not conditionalized, must be safely removed.

```
// Error if lowest next-time element in past
if ( time_next < time_now ) {
    ostringstream ostream;
    ostream << "Time: "
        << time_now
        << " scheduler.start() "
        << node_ptr -> get_name()
        << " Found for earlier time "
        << time_next << "\n";
    cerr << ostream.str();
    return;
} // end if
```



The 2nd tactic is to code statements issuing debug messages.

This tried-and-true tactic involves surrounding a suspect code segment with message-issuing statements advising you of their execution and outputting useful values.

Good practices include to:

- Print sufficient information to support your debugging efforts;
- Provide a means to link message output back to the issuing statement;
- Conditionalize message issuance with verbosity level or conditional compilation; and
- Send messages to the character-based error stream (**cerr**) rather than character-based output stream (**cout**), or be sure to flush the message output!

Issues to consider include that such temporary statements:

- Clutter the source code and program output;
- Provide the opportunity to inject yet more bugs; and
- If not conditionalized, must later be safely removed.

Here, the programmer codes a check for a condition that should never occur. The programmer atomized the output to accommodate multiple threads.

Tactic 3: Assert Invariant Truths

Anywhere in problematic code segments where an expression will always have some known value, you can assert that invariant truth.

Advantages of utilizing built-in assertions include:

- Assertion success executes efficiently.
- Assertion failure automatically displays the asserted expression, source file and line number, and enclosing function.
- Assertions are automatically conditionalized by whether the **NDEBUG** macro is defined at the point of their compilation.

```
// Fatal if lowest next-time element in past
assert ( time_next >= time_now ) ;
```

238 © Cadence Design Systems, Inc. All rights reserved.



The 3rd tactic is to code statements asserting invariant truths.

Anywhere in problematic code segments where an expression will always have some known value, you can assert that invariant truth.

Advantages to utilizing built-in assertions include that:

- Assertion success executes efficiently;
- Assertion failure automatically displays the asserted expression, source file and line number, and enclosing function; and
- Assertions are automatically conditionalized by whether the “no debug” (**NDEBUG**) macro is defined at the point of their compilation.

Here, the programmer asserts a condition that must always be true when the statement is executed. Assertion failure is a fatal failure that terminates execution.

Tactic 4: Utilize a Logger

A **Logger** is a device or computer program that systematically records events, measurements, or observations.

- For example: data, events, messages, transactions.

The logs are useful for a variety of purposes.

- For example, auditing, diagnostics, profiling, statistics.

Here, we are most interested in *diagnostics*, for which a *message logger* is most appropriate.

- Their output might include attributes such as *message id*, *timestamp*, *process id*, *thread id*, message *verbosity* and *category*, *file name* and *line number*, *function name*, and a user string providing more data and explanation.
- The logger would ideally be from a standard library, and its output be both easily human-readable and easily machine-parsed.

```
// Error if lowest next-time element in past
if ( time_next < time_now ) {
    BOOST_LOG_SEV(lg,error) << "Found for past time";
    return;
}
```

239 © Cadence Design Systems, Inc. All rights reserved.



The 4th tactic is to utilize a logger.

A *Logger* is a device or computer program that systematically records events, measurements, or observations, for example: data, events, messages, or transactions.

The logs are useful for a variety of purposes, for example: auditing; diagnostics; profiling; and statistics.

Here, we are most interested in diagnostics, for which a message logger is most appropriate.

- Their output might include a message identifier, time stamp, process identifier, thread identifier, message verbosity and category, file name and line number, function name, and a user string providing more data and explanation.
- The logger would ideally be from a standard library, and its output be both easily human-readable and easily machine-parsed.

Here, if the scheduler finds an element scheduled for an earlier than current time, the programmer issues a log message having severity “error”. Assume that the logger back-end has been configured with filters, attributes to add to the message, and the destination output stream.

Configuring the Boost logger requires several code lines not displayed here.

Tactic 5: Utilize a Source-Level Debugger

A source-level debugger is a symbolic debugger that can show the source code line associated with the current program counter.

A typical source-level debugger can:

- Pause a program at a particular instruction or line
- Pause a program on an expression's value change
- Pause a program upon its receiving a signal
- Examine the call stack
- Examine and edit source files
- Examine and set variable values
- Step a program by instruction or by line
- Invoke a program function

```
$ g++ *.cc -g -o system -pthread
$ gdb system
(gdb) info functions start
(gdb) break \
hidden::temp_sched<hidden::dummy>::start(unsigned int)
Breakpoint 1 at 0x804dcd8: file pnet_sched.h, line 93.
(gdb) run
Starting ...
Breakpoint 1, ...
93 while ( time_now <= time_max ) {
(gdb) printf "%d %d\n", time_now, time_max
0 1
(gdb)
```

240 © Cadence Design Systems, Inc. All rights reserved.



The 5th tactic is to utilize a source-level debugger.

A source-level debugger is a symbolic debugger that can show the source code line associated with the current program counter.

Among the majorly useful things that a typical source-level debugger can do are to:

- Pause a program at a particular instruction or line;
- Pause a program on an expression's value change;
- Pause a program upon its receiving a signal;
- Examine the call stack;
- Examine and edit source files;
- Examine and set variable values;
- Step a program by instruction or by line; and
- Invoke a program function.

Here, the programmer runs the program up to entering the scheduler function “start”, and prints the value of two pertinent variables.

What Are GDB and DDD?



The **GNU Debugger (GDB)** is a portable debugger that works on many operating systems and works for many programming languages.



The **GNU Data Display Debugger (DDD)** is a graphical interface to textual-interface debuggers such as GDB.

The GNU debugger is a free debugger with a textual user interface that supports debugging tasks such as:

- Start your program with your specified arguments.
- Pause your program at specified lines, instructions, conditions, events, or on function entry.
- Examine the program's source, data, and call stack.
- Continue or step the program.

The GNU debugger fully supports C/C++ and partially supports multiple other languages.

The GNU data display debugger is a free debugger with a graphical user interface to wrap around debuggers having only a textual interface.

For debugging executable binaries, the GNU data display debugger fully supports the textual-interface debuggers DBX, GDB, XDB.

241 © Cadence Design Systems, Inc. All rights reserved.



The *GNU Debugger (GDB)* is a portable debugger that works on many operating systems and works for many programming languages.

The *GNU Data Display Debugger (DDD)* is a graphical interface to textual-interface debuggers such as GDB.

The GNU debugger is a free debugger with a textual user interface that supports debugging tasks such as to:

- Start your program with your specified arguments;
- Pause your program at specified lines, instructions, conditions, events, or upon function entry;
- Examine the program's source, data, and call stack; and
- Continue or step the program.

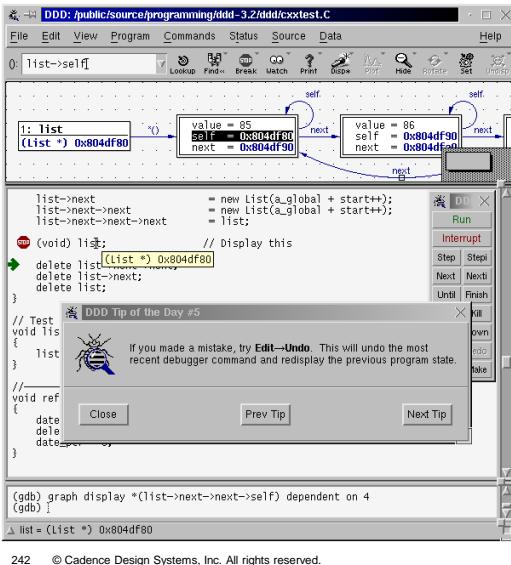
The GNU debugger fully supports C and C++, and partially supports multiple other languages, for which you should see the GDB User Guide.

The GNU data display debugger is a free debugger with a graphical user interface to wrap around debuggers having only a textual interface.

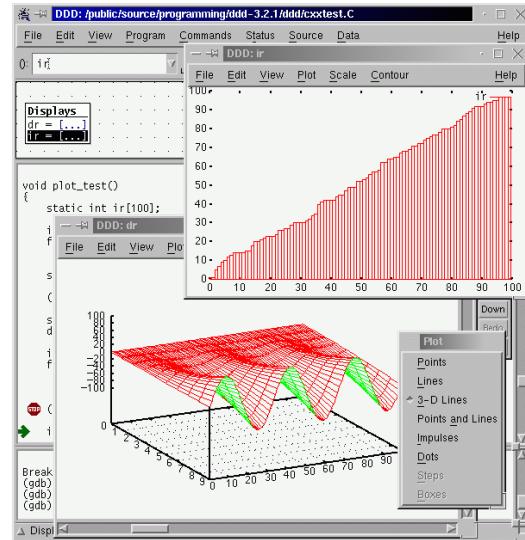
For debugging executable binaries, the GNU data display debugger fully supports the textual-interface debuggers DBX, GDB, XDB.

Example: DDD Plots

This DDD main window shows data, source, console.
<http://www.gnu.org/software/ddd/all.png>



The DDD offers some advanced data visualization.
<http://www.gnu.org/software/ddd/plots.png>



cadence®

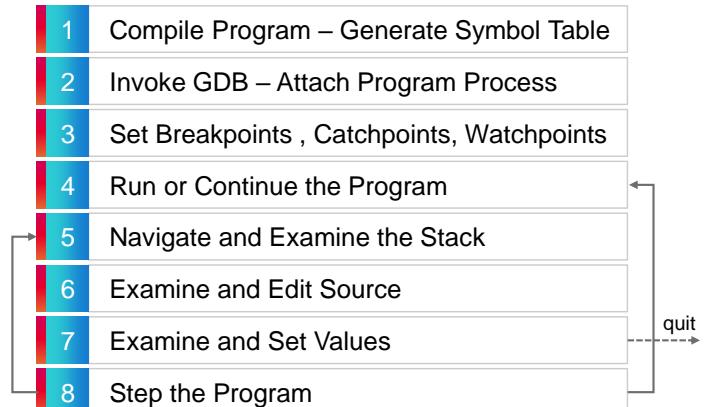
The left image displays the main window, having a data display pane, an interactive source browser, and a console, and displays a separate command tool window of buttons associated with commonly-issued debugger commands.

The right image displays graphical plots of the element values in 2-dimensional and 3-dimensional arrays.

Debugging with GDB



The GNU debugger offers approximately 1400 commands, functions, and variables. Many or most commands have multiple options and arguments, and many functions can take multiple argument combinations. Generally, debugging with GDB can be classified into 8 steps.



cadence®

243 © Cadence Design Systems, Inc. All rights reserved.

Debugging with GDB is in these general steps.

- 1st – Compile with GCC as you normally would, but also at least generate a symbol table.
- 2nd – Invoke GDB and optionally specify an executable file and optionally specify either a core dump file or the process identifier number.
- 3rd – Set stops on source code locations, occurrence of program events, and expression value changes.
- 4th – Override as needed your program's arguments, environment, working directory, and terminal, then start or run your program.
- 5th – Navigate and examine the call stack and individual frames and that frame's variables.
- 6th – Search and list source lines, and edit the source.
- 7th – Print variable and memory values with a multitude of formats, either on demand or at each stop, and set variable or memory values.
- 8th – Step by line or by instruction, stepping into or over function calls, or to the function call return.

The GNU debugger offers approximately 1400 commands, functions, and variables. Many or most commands have multiple options and arguments, and many functions can take multiple argument combinations. You are strongly encouraged to refer to the user guide, and perhaps even to read it entirely just once so that you know what capabilities are available to you.

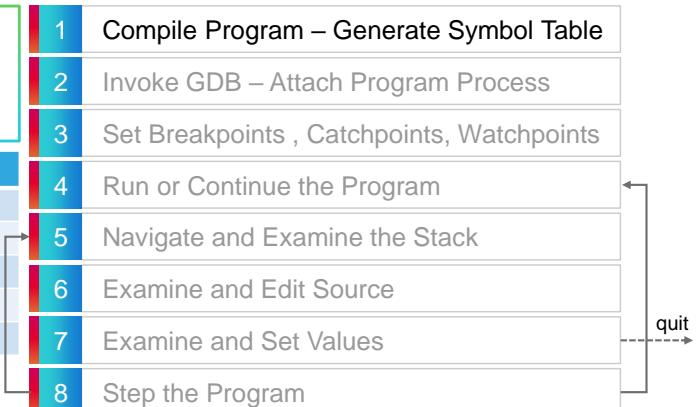
Debugging with GDB: Compiling the Program



Compile with GCC as you normally would, but also at least generate a symbol table.

Option	Description
<code>-g [level]</code>	Generate debugging information 0-3 [2]
<code>-ggdb [level]</code>	Generate debug info with GDB extensions
<code>-gdwarf [-version]</code>	Make it DWARF information 2-5 [4]
<code>-O [level]</code>	Manage optimization 0-3,fast,g,s [0]
<code>-W [flag]</code>	Manage warnings, for example <code>-Wall</code>

DWARF information may be the default and also may be not supported.



```
g++ *.cc -g -o system -pthread
```

244 © Cadence Design Systems, Inc. All rights reserved.



When debugging with GDB, the 1st step is to compile with GCC as you normally would, but also at least generate a symbol table.

- You must generate debug information. To debug without these symbols may be possible, but is extremely impractical. People mostly use the unadorned option, which generates some information specific to the GNU debugger, and which other debuggers might react unkindly to. A multitude of option variations are available, with which to control what information is generated, for which you can refer to the compiler documentation.
- You can, and should, manage compiler optimizations. Some optimizations interfere with debugging. Without this option, the default value is 0, meaning no optimizations. With this option, the default value is 1, meaning a relatively low level of optimizations. Optimization options may be buried in the invocation script, so you should explicitly override them to level 0. You can alternatively override them to the level ‘g’, which enables only those options guaranteed to not effect subsequent debug efforts.
- A multitude of options are available for managing warnings, either to suppress warnings, or to request additional warnings, for which you can refer to the compiler documentation. People mostly use only the flag “all”, which does not really enable all warnings, but rather, those for an agreed-upon set of questionable and easily-corrected constructs.

The example compiles all the local C++ sources, creating a symbol table, and enabling threads.

Debugging with GDB: Invoking GDB

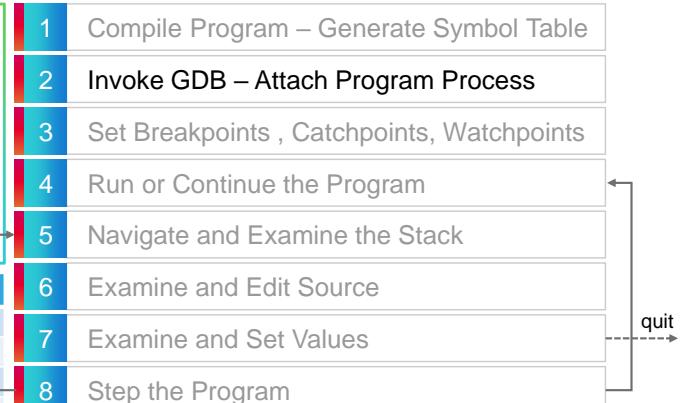


Invoke GDB and optionally specify executable file and optionally specify core dump file or process identifier number.

- `gdb [options] [program [core | pid]]`
- `gdb [options] --args program [arg(s)]`

Optionally later specify and attach program.

Command	Description
<code>file [-readnow] name</code>	Load/Unload executable and symbols
<code>exec-file [file]</code>	Load/Unload executable file
<code>core[-file] [name]</code>	Load/Unload core file
<code>symbol-file [[-readnow] name]</code>	Load/Unload symbols file
<code>info {files target}</code>	Print current target files
<code>directory [name]</code>	Reset or prepend source path
<code>set directories path</code>	Set source path for search
<code>show directories</code>	Show source search path
<code>attach PID</code>	Attach to running process
<code>detach</code>	Detach attached process



`gdb system`



245 © Cadence Design Systems, Inc. All rights reserved.

When debugging with GDB, the 2nd step is to invoke GDB, optionally specify the executable file, and optionally specify either a core dump file or the process identifier number.

You can alternatively provide these parameters later, as debugger commands.

- The command “**file**”, without an argument, unloads any loaded memory image and symbols, and with an argument, loads the memory image and symbols.
- The command “executable-file” (**exec-file**) without an argument, unloads any loaded memory image, and with an argument, loads the memory image.
- The command “symbol-file”, without an argument, unloads any symbols, and with an argument, loads symbols.
- The command “information” (**info**), with its argument “*files*” or “*target*”, displays information about the currently loaded target files.
- The command “**directory**”, without an argument, resets the source search path to only the compilation directory, if recorded, and the current working directory, and with an argument, prepends the argument to the front of the source search path.
- The command “**set directories**”, sets the source search path to the argument, and if the argument omits the compilation directory or current working directory, adds them to the path.
- The command “**show directories**”, shows the current source search path.
- The command “**attach**” attaches the specified running process. If the program file is not loaded, the debugger searches first the current working directory, and then utilizing the source search path.
- The command “**detach**” detaches the attached process.

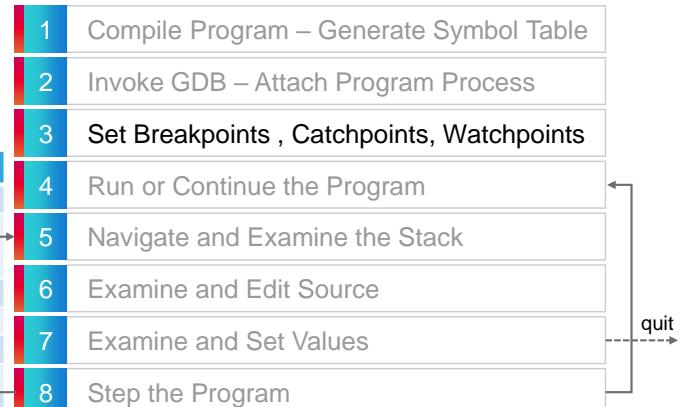
The example command invokes the debugger to debug the program named “*system*”.

Debugging with GDB: Setting Breakpoints

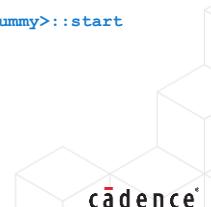


Set stops on source code locations.

Command	Description
info program	Display program info and if stopped why.
break [location] [thread n] [if cond]	Stop at next instruction or specified location, optionally only in specified thread, optionally only when cond true.
tbreak [location] [if cond]	Delete breakpoint upon activation.
[t]hbreak [location] [if cond]	Hardware-assisted break or tbreak.
rbreak [file:]regex	Stop on entering matching functions.
clear [location]	Clear current or specified breakpoint.
*** Following apply to breakpoints, catchpoints, watchpoints ***	
commands [range] [commands] end	Commands to execute upon activation.
info breakpoints [n ...]	Display all or specified b/c/w points' info.
disable [breakpoints] [range]	Disable all or specified b/c/w points.
enable [breakpoints] [delete once count n] [range]	Enable all or specified b/c/w points, optionally delete after activation, optionally disable after activation(s).
delete [breakpoints] [range]	Delete all or specified b/c/w points.



`break hidden::temp_sched<hidden::dummy>::start`



When debugging with GDB, the 3rd step is to set stops on source code locations, program events, and expression value changes. Here, we set stops on source code locations.

- The command “**info program**”, displays the program status, and if it is stopped, why.
- The command “**break**”, sets a stop, at the next instruction, or the specified location, optionally in only the specified thread, and optionally to activate only when a specified condition is true.
- The command “temporary break” (**tbreak**), sets a breakpoint that automatically deletes upon its activation.
- The commands “hardware break” (**hbreak**), and “temporary hardware break” (**thbreak**), set hardware-assisted breakpoints, for platform architectures that support them.
- The command “regular expression break” (**rbreak**), sets an unconditional breakpoint on functions that match the regular expression, optionally restricted to a specified file.
- The command “**clear**”, clears the breakpoint at the current stopped location or other specified location.
- The command “**commands**”, specifies a list of commands to execute upon breakpoint, catchpoint, or watchpoint activation. It applies by default to the points set by the most recently-issued command, and you can alternatively specify a range of points. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.
- The command “**info breakpoints**”, displays information about all or the specified breakpoints, catchpoints, and watchpoints. The information it displays includes: The breakpoint number, type, and disposition; whether it is enabled; its address; and if a source-line breakpoint, its file name and line number.
- The command “**disable breakpoints**”, disables all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.
- The command “**enable breakpoints**”, enables all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order. You can optionally specify that the points be deleted upon their next activation, or enabled for only some number of next activations.
- The command “**delete breakpoints**”, deletes all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.

The example command sets a breakpoint upon entering the function named “*start*”.

What Is a GDB Location?



A GDB **location**, also known as a **linespec**, specifies a source code location for commands such as **advance**, **break**, **clear**, **dprintf**, **edit**, **info line**, **info macros**, **list**, **jump**, **skip**, **trace**, and **until**.¹

Location	Description	Example
{+ -} offset	Line offset from current line.	<code>list +5</code>
[filename :] linenum	Line in current or specified file.	<code>list pnet_sched.h:93</code>
[filename :] function	Entry point of specified function.	<code>list pnet_sched.h:\hidden::temp_sched<hidden::dummy>::start</code>
[[filename :] function :] label	Label of current or specified function.	<code>list pnet_sched.h:\hidden::temp_sched<hidden::dummy>::start:\CLONE</code>
* expression	Address.	<code>list *0x804dcdb</code>
*[' file ' ::] function	First instruction of specified function.	<code>list *'pnet_sched.h':\hidden::temp_sched<hidden::dummy>::start</code>

Table omits static probes for which see GDB User Guide.

247 © Cadence Design Systems, Inc. All rights reserved.



A GDB *location*, also known as a *line specification*, specifies a source code location for commands such as **advance**, **break**, **clear**, dynamic formatted print (**dprintf**), **edit**, **info line**, **info macros**, **list**, **jump**, **skip**, **trace**, and **until**.

This table lists the six location formats that do not involve static probes.

- Use the *offset* format, to specify an offset from the current line. When used as the 2nd line specification, the offset is from the 1st line specification.
- Use the *line number* format, to specify a line number of a specified file.
- Use the *function* format, to specify the first executable line of the function, and optionally include the file name, to disambiguate between multiple files defining that function.
- Use the *label* format, to specify a line label, either in the current stack frame, or optionally in some other specified function.
- Use the *address* format, to specify a numerical instruction address. Specifying a function name, means the function's first instruction. Optionally include the file name, to disambiguate between multiple files defining that function.

What Is a GDB Expression?



A GDB **expression** includes any literals, variables, and operators native to the programming language, plus preprocessor macros for which debug information is compiled, plus additional operators, functions, and variables defined within GDB.

Operator	Example	Description
@	*array@length	Interpret memory as array
::	'mfile.cc' ::mvar	Qualify variable scope
{type}addr	{int}0x123456789	Interpret address as type

Convenience Function	Description
\$_memeq(buf1, buf2, len)	Whether buffers are equal
\$_regex(str, regex)	Whether string matches regex
\$_streq(str1, str2)	Whether strings are equal
\$_strlen(str)	String length

Requires Python support.

Convenience Variable	Description
\$__	Most recently examined address
\$__	Value at most recently examined address
\$bpnum	Most recently set breakpoint
\$cdir	Compilation directory (if recorded)
\$cwd	Current working directory
\$_exitcode	Program exit code
\$_probe_argc	Static probe argument count.
\$_probe_argn	Static probe argument
\$_siginfo	Most recent signal information
\$_thread	Current thread

init-if-undefined \$variable = expression
show convenience

Create convenience variable
Show convenience variables

Incomplete list – see GDB User Guide.

248 © Cadence Design Systems, Inc. All rights reserved.



GDB accepts all literals, variables, and operators native to the programming language, plus preprocessor macros for which debug information is compiled, plus operators defined by GDB.

- GDB defines the operator “at” (@), which is a binary operator forming an artificial array. Its left operand is the array first element, and its right operand is the number of elements of that type to include in the expression. You can alternatively, using the cast notation, simply cast the memory location to a fixed-size array. Expressions so formed appear most frequently as arguments to the command “print”.
- GDB extends the scope resolution operator (::), to disambiguate static variables declared in multiple files or multiple functions, and to access automatic scope variables hidden by a more local declaration.
- GDB defines a cast operator ({}), to refer to any memory location as holding any specified object type.

GDB version 7.6 defines 4 convenience functions. The convenience functions are available only if GDB was configured with Python support.

GDB version 7.6 defines somewhat more than 20 convenience variables. You can also define your own convenience variables, and you can show all the defined convenience variables. Displayed here is a selection of arguably the most frequently used.

Debugging with GDB: Setting Catchpoints



Set stops on occurrence of program events.

Command	Description
<code>info program</code>	Display program info and if stopped why.
<code>catch event, tcatch event</code>	Stop on event, optionally delete.
*** Following apply to breakpoints, catchpoints, watchpoints ***	
<code>commands [range] [commands] end</code>	Commands to execute upon activation.
<code>info breakpoints [n ...]</code>	Display all or specified b/c/w points' info.
<code>disable [breakpoints] [range]</code>	Disable all or specified b/c/w points.
<code>enable [breakpoints] [delete once count n] [range]</code>	Enable all or specified b/c/w points, optionally delete after activation, optionally disable after activation(s).
<code>delete [breakpoints] [range]</code>	Delete all or specified b/c/w points.

Event	Description
<code>throw, catch</code>	Throwing or catching a C++ exception.
<code>exec, fork, vfork</code>	System call "exec", "fork", "vfork".
<code>syscall [name n]</code>	Any or specified system call.
<code>{load unload} [regexp]</code>	Loading or unloading any or specified library(s).
<code>signal [signals] 'all'</code>	Delivery of specified or all signals.

Signals do not include SIGINT or SIGTRAP, used by GDB internally, unless explicitly specified by name or 'all'.

249 © Cadence Design Systems, Inc. All rights reserved.

When debugging with GDB, the 3rd step is to set stops on source code locations, program events, and expression value changes. Here, we set stops on program events.

- The command “**info program**”, displays the program status, and if it is stopped, why.
- The command “**catch**”, sets a stop on a program event. The event can be:
 - Throwing or catching a C++ exception;
 - Making a system call;
 - Loading or unloading a library; or
 - Delivering signals.
- The command “temporary catch” (**tcatch**), sets a catch point that automatically deletes upon its activation.
- The command “**commands**”, specifies a list of commands to execute upon breakpoint, catchpoint, or watchpoint activation. It applies by default to the points set by the most recently-issued command, and you can alternatively specify a range of points. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.
- The command “**info breakpoints**”, displays information about all or the specified breakpoints, catchpoints, and watchpoints. The information it displays includes: The break point number, type, and disposition; whether it is enabled; its address; and if a source-line breakpoint, its file name and line number.
- The command “**disable breakpoints**”, disables all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.
- The command “**enable breakpoints**”, enables all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order. You can optionally specify that the points be deleted upon their next activation, or enabled for only some number of next activations.
- The command “**delete breakpoints**”, deletes all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.

The example command sets a catchpoint on the system call “clone”.

1 Compile Program – Generate Symbol Table

2 Invoke GDB – Attach Program Process

3 Set Breakpoints , Catchpoints, Watchpoints

4 Run or Continue the Program

5 Navigate and Examine the Stack

6 Examine and Edit Source

7 Examine and Set Values

8 Step the Program

catch syscall clone

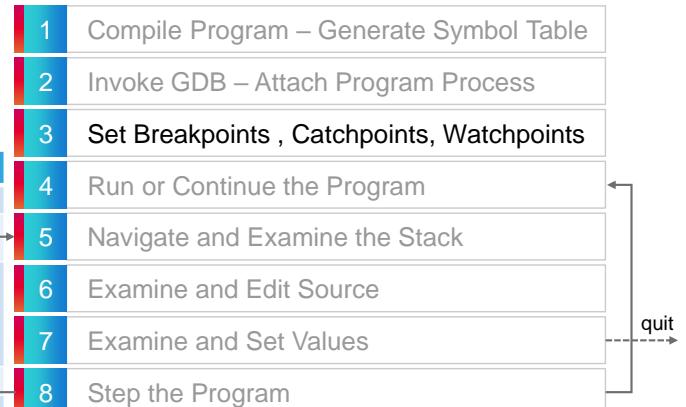


Debugging with GDB: Setting Watchpoints



Set stops on expression value changes.

Command	Description
<code>info program</code>	Display program info and if stopped why.
<code>set can-use-hw-watchpoints {0 1}</code>	Disable default enabled HW watchpoints. Show whether HW watchpoints enabled.
<code>show can-use-hw-watchpoints</code>	
<code>watch [-location] expr [thread thread_number] [mask mask_value]</code>	Set a stop on expression value change. Optionally only in specified thread. Optional bit-mask for use with -location. Like <code>watch</code> but stop on read.
<code>rwatch ...</code>	Like <code>watch</code> but stop on any access.
<code>awatch ...</code>	
<code>info watchpoints [n ...]</code>	Display all or specified watchpoints' info.
*** Following apply to breakpoints, catchpoints, watchpoints ***	
<code>commands [range] [commands] end</code>	Commands to execute upon activation.
<code>info breakpoints [n ...]</code>	Display all or specified b/c/w points' info.
<code>disable [breakpoints] [range]</code>	Disable all or specified b/c/w points.
<code>enable [breakpoints] [delete once count n] [range]</code>	Enable all or specified b/c/w points, optionally delete after activation, optionally disable after activation(s).
<code>delete [breakpoints] [range]</code>	Delete all or specified b/c/w points.



watch time_now

Additional notes:

- The `-location` option watches the location of the `expression` instead of its `terms`.
- Few architectures support `mask`.
- Only HW watchpoints support `thread` and `awatch` and `rwatch`.
- The debugger automatically deletes watchpoints that go out of scope.

250 © Cadence Design Systems, Inc. All rights reserved.



When debugging with GDB, the 3rd step is to set stops on source code locations, program events, and expression value changes. Here, we set stops on expression value changes.

- The command “**info program**”, displays the program status, and if it is stopped, why.
- Use the command “set can use hardware watchpoints” (**set can-use-hw-watchpoints**), to disable setting hardware watchpoints. The debugger by default uses hardware watchpoints when it can.
- The command “**watch**”, sets a stop on a write that changes the expression value.
 - The option “location” (`-location`), watches the location of the expression instead of the expression terms.
 - The argument “`thread`”, restricts the watch to only the specified thread. Only hardware watchpoints can do this.
 - The argument “`mask`”, restricts the watch to only the masked bits. This argument implies use of the “`location`” option. Few architectures support this.
- The command “read watch” (**rwatch**), sets a watch that activates on read. Only hardware watchpoints can do this.
- The command “access watch” (**awatch**), sets a watch that activates on read or write. Only hardware watchpoints can do this.
- The command “**info watchpoints**”, displays information about all or the specified watchpoints. The information it displays includes: The watchpoint number, type, and disposition; whether it is enabled; and its address.
- The debugger automatically clears watchpoints that go out of scope.
- The command “**commands**”, specifies a list of commands to execute upon breakpoint, catchpoint, or watchpoint activation. It applies by default to the points set by the most recently-issued command, and you can alternatively specify a range of points. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.
- The command “**info breakpoints**”, displays information about all or the specified breakpoints, catchpoints, and watchpoints. The information it displays includes: The breakpoint number, type, and disposition; whether it is enabled; its address; and if a source-line breakpoint, its file name and line number.
- The command “**disable breakpoints**”, disables all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.
- The command “**enable breakpoints**”, enables all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order. You can optionally specify that the points be deleted upon their next activation, or enabled for only some number of next activations.
- The command “**delete breakpoints**”, deletes all or the specified breakpoints, catchpoints, or watchpoints. The range list is space-separated single numbers and hyphenated ranges between two numbers in ascending order.

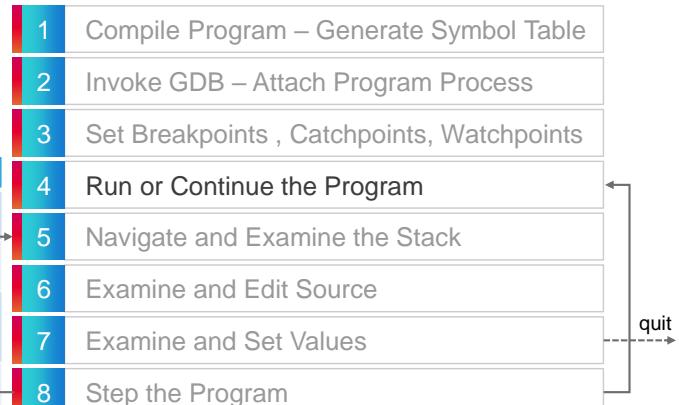
The example command sets a watch point on the variable “time now”. You can set a watch point on only in-scope expressions, and the debugger automatically deletes the watchpoint when the expression goes out of scope.

Debugging with GDB: Running or Continuing the Program



Override as needed your program's arguments, environment, working directory, and terminal, then start or run your program.

Command	Description
<code>set args [args]</code> <code>show args</code>	Clear or set program arguments. Show program arguments.
<code>path directory</code> <code>show paths</code>	Insert directory at PATH front. Show the PATH environment variable.
<code>set environment var[=value]</code> <code>unset environment var</code> <code>show environment [var]</code>	Set an environment variable. Unset an environment variable. Show all or one environment variable(s).
<code>cd [directory]</code> <code>pwd</code>	Change working directory. Print working directory
<code>set inferior-tty device</code> <code>show inferior-tty device</code> <code>info terminal</code>	Set program's terminal device. Show program's terminal device. Display terminal information.
<code>run [args] [redirection]</code>	Run program, optionally set arguments.
<code>start [args] [redirection]</code>	Start program, optionally set arguments. Execution breaks at main procedure entry.
<code>continue [count]</code>	Continue, optionally ignore count breaks.
<code>jump {line location}</code>	Continue at specified location.



251 © Cadence Design Systems, Inc. All rights reserved.



When debugging with GDB, the 4th step is to override as needed your program's arguments, environment, working directory, and terminal, and then start or run your program.

- The command “set arguments” (**set arg**), without arguments, clears any existing arguments, and with arguments, sets program arguments. Issuing this command, overrides any arguments you provided as you invoked the debugger.
- The command “**path**” inserts the specified directory at the front of the program’s environment variable “PATH”, thus partially prefixing the PATH inherited upon invoking GDB.
- The command “**set environment**” sets an environment variable for the program. Be aware that for Unix systems, the debugger invokes the shell named in the environment variable “SHELL”, to run the program, so shell startup commands can potentially reset the environment variables. You can work around this by moving such commands from the shell run commands script to the shell login script.
- The command “change directory” (**cd**), changes the program’s working directory, without an argument, to your HOME directory, and with an argument, to the specified directory. Issuing this command, overrides the working directory inherited as you invoked the debugger.
- The command “set inferior teletype” (**set inferior-tty**) sets the program’s terminal device. The command “teletype” (**tty**) is an alias for the command “**set inferior teletype**”.
- The command “**run**”, runs the program, optionally first setting the program arguments to any specified arguments. The new arguments persist until again changed. You can with this command, pass input and output redirection tokens to the program, in whatever is the shell’s syntax. The redirection tokens apply to only the program’s input and output, and do not effect the program’s controlling terminal.
- The command “**start**”, sets a breakpoint on the main procedure entry point, and runs the program, optionally first setting the program arguments to any specified arguments. The new arguments persist until again changed. You can with this command, pass input and output redirection tokens to the program, in whatever is the shell’s syntax. The redirection tokens apply to only the program’s input and output, and do not effect the program’s controlling terminal. You can use this command to examine the results of elaboration before running the program.
- The command “**continue**”, continues execution from the current point. If stopped due to a breakpoint, the optional count argument ignores the next count breakpoint activations at this address.
- The command “**jump**” continues execution at the specified point. It changes the program counter, and nothing else, so should be used carefully by only advanced users.

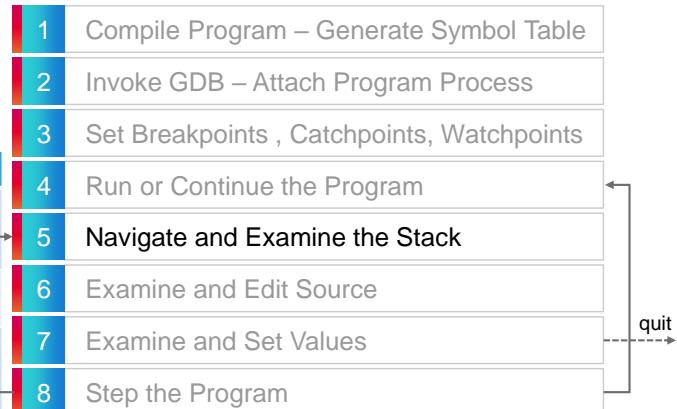
The example command runs the program.

Debugging with GDB: Navigating and Examining the Stack



Navigate and examine the call stack and individual frames and that frame's variables.

Command	Description
<code>backtrace [full] [[-]n]</code> <code>info stack and where</code> are aliases for <code>backtrace</code> .	Print call stack. Optionally with variables. Optionally limit to n +/- frames.
<code>frame [{addr n}]</code>	Print current frame (brief). Optionally set current frame.
<code>up n</code> <code>up-silently n</code> <code>down n</code> <code>down-silently n</code>	Move n frames up and print. Move n frames up silently. Move n frames down and print. Move n frames down silently.
<code>info frame [addr]</code>	Describe current or specified frame.
<code>info args</code>	Print current frame's arguments.
<code>info locals</code>	Print current frame's variables.



backtrace



When debugging with GDB, the 5th step is to navigate and examine the call stack.

- The command “back trace” (**backtrace**), prints the call stack, optionally including the values of local variables, and optionally printing only a specified number of innermost (n) or outermost ($-n$) frames. The commands “information stack” (**info stack**) and “where” are aliases for the command “back trace”.
- The command “frame”, with no argument, describes the selected frame, and with an argument, selects a frame, as specified by address or by number. An address argument depends on the CPU architecture, and could involve a program counter, one or more stack pointers, and one or more frame pointers.
- The command “up”, moves one or more frames up, that is, toward earlier, higher-numbered frames.
- The command “down”, moves one or more frames down, that is, toward later, lower-numbered frames.
- The command “information frame” (**info frame**), verbosely describes the selected or specified frame. An address argument depends on the CPU architecture.
- The command “information arguments” (**info args**), prints the selected frame’s arguments and values.
- The command “information locals (**info locals**), prints the selected frame’s variables that are visible without qualification, and their values.

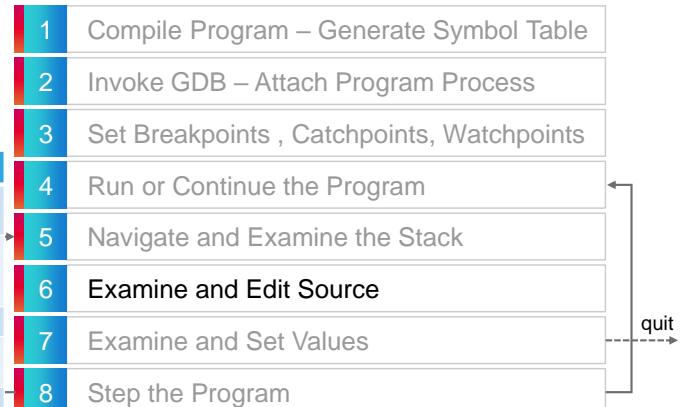
The example command prints minimal call stack information.

Debugging with GDB: Examining and Editing the Source



Search and list source lines, and edit the source.

Command	Description
<code>set listsize n</code> <code>show listsize</code>	Number of lines to list
<code>list [+ -]</code> <code>list [linespec]</code> <code>list [m, ,n m,n]</code>	List lines before/after current line List according to <i>linespec</i> List lines from/to <i>linespec(s)</i>
<code>edit location</code>	Edit file specified by <i>location</i>
<code>forward-search regexp</code> <code>reverse-search regexp</code>	Search for <i>regexp</i>
<code>directory dirname(s)</code> <code>set directories path</code> <code>show directories</code>	Reset or add to source search path Set source search path Show source search path
<code>set substitute-path from to</code> <code>unset substitute-path [path]</code> <code>show substitute-path [path]</code>	Substitute path for relocated sources



`list 93,105`

List size 0 means all lines.

Editor is /bin/ex or EDITOR environment variable.
Source search path always includes \$cdir:\$cwd

253 © Cadence Design Systems, Inc. All rights reserved.



When debugging with GDB, the 6th step is to examine and edit the source.

- The command “set list size” (**set listsize**), sets the line count for list commands. A count of 0 means to list all lines,
- The command “show list size” (**show listsize**), shows the current line count for list commands.
- The command “**list**”, lists source lines. You can list lines before or after the current line, or according to one or more line specifications. You can imply that one of two line specifications be the first or last line of the file.
- The command “**edit**”, loads the file of the specified location into the editor specified by the EDITOR environment variable.
- The command “**forward-search**”, searches the current file forward for a regular expression, and the command “**reverse-search**”, searches the current file backward for a regular expression.
- The command “**directory**”, without an argument, resets the source search path to its default value, and with an argument, adds the argument to the front of the source search path.
- The command “**set directories**”, sets the source search path to the path argument, and includes the default search path.
- The command “**show directories**”, shows the source search path.
- The command “**set substitute-path**” substitutes some other directory for a directory in the source search path, to enable finding source that has been relocated.
- The command “**unset substitute-path**”, unsets one or all substitute paths.
- The command “**show substitute-path**”, shows one or all substitute paths.

The example command lists source lines 93 through 105 in the current file.

Debugging with GDB: Examining and Setting Values

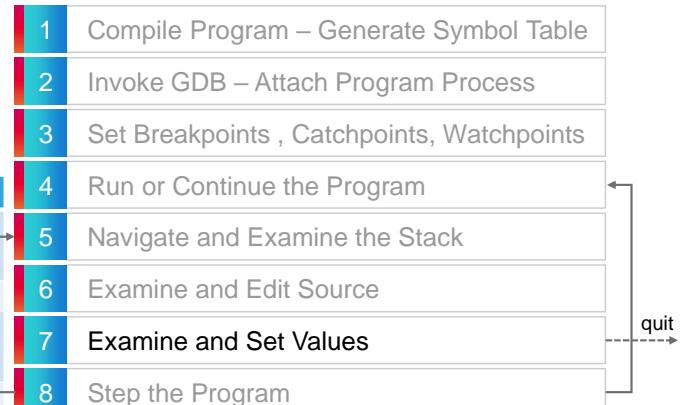


Print variable and memory values with a multitude of formats either on demand or at each stop, and set variable or memory values.

Command	Description
<code>print [/f] [expr] - OR -</code>	Print expression value. Optionally specify format. Optionally specify expression.
<code>printf "format" expr{,expr}</code>	Print formatted expression(s) value.
<code>x[/nfu] [addr]</code>	Examine memory. Optionally specify size, format, units. Optionally specify address.
<code>display</code> <code>display[/f] [expr]</code> <code>display/nfu addr</code>	Display everything on display list. Add <code>expr</code> to list to display at each stop. Add <code>addr</code> to list to display at each stop.
<code>info display</code> <code>disable display {n n-m}</code> <code>enable display {n n-m}</code> <code>delete display {n n-m}</code> <code>undisplay</code> is an alias for <code>delete display</code>	List expressions on display list. Disable display numbers. Enable display numbers. Delete display numbers.
<code>set [var] expr=value</code>	Set variable or memory value.

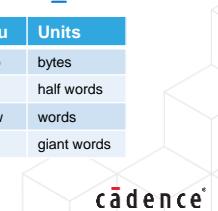
Also many commands not displayed here for print settings.

254 © Cadence Design Systems, Inc. All rights reserved.



`printf "%d %d\n", time_now, time_max`

f	Format	f	Format	u	Units
a	address	o	octal	b	bytes
i	instruction	x	hexadecimal	h	half words
t	binary	f	floating point	w	words
c	character	r	raw (not pretty)	g	giant words
d, u	signed/unsigned	s	string		



When debugging with GDB, the 7th step is to print variable and memory values, in your choice of a multitude of formats, either on demand, or at each stop, and set variable or memory values.

The command “**print**”, with no expression argument, prints what it most recently printed, perhaps with a different specified format, and with an expression argument, prints the expression value, with a default format appropriate for the type, or with your specified format.

The command “**examine**” (**x**), with no address argument, examines memory locations starting at the default memory address, perhaps a different number of locations, and perhaps with a different specified format, and with an expression argument, examines memory locations starting at the specified memory address.

- The size defaults to 1 unit upon each command re-issuance.
- The format initially defaults to hexadecimal, and is retained between command issuance unless re-specified. If you specify a format, then you must specify a repeat count.
- The unit initially defaults to 4-byte words, and is normally retained between command issuance, unless re-specified, with two exceptions. The exceptions are that for the “*instruction*” format, the units are ignored, and for the “*string*” format, the unit default resets to single-byte after each command issuance. If you specify the units, then you must specify a unit count.
- The default memory address is one after the most recently examined. However, the command “**info breakpoints**”, leaves it at the last listed breakpoint, the command “**info line**” leaves it at the line first instruction, and the command “**print**”, when used to print a memory location value, leaves it at that memory location.
- The command “**display**”, with no arguments, displays everything on the display list, and with an argument, adds that expression or memory location to the list of things to display upon each stop.
- The command “**info display**”, lists the expressions on the display list, while the commands “**disable display**”, “**enable display**”, and “**delete display**”, manage the display list.
- The command “**set**”, with its subcommand “**variable**”, sets an expression to a value. You can omit the “variable” subcommand if the expression prefix does not match any other subcommand. Be aware that the command “**set**” has approximately 170 subcommands that can potentially match an expression prefix.

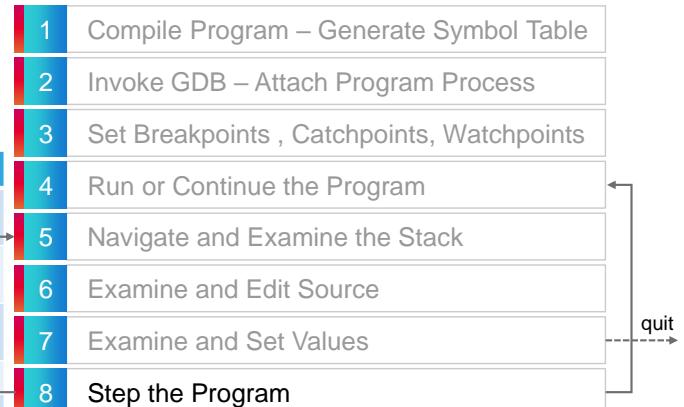
The example command prints in decimal format the value of two expressions. This command is equivalent to the C function of the same name.

Debugging with GDB: Stepping the Program



Step by line or by instruction, stepping into or over function calls, or to their return.

Command	Description
step [count]	Step <i>count</i> lines <i>into</i> calls.
next [count]	Step <i>count</i> lines <i>over</i> calls.
stepi [count]	Step <i>count</i> instructions <i>into</i> calls.
nexti [count]	Step <i>count</i> instructions <i>over</i> calls.
set step-mode [on off]	If on , for function having no line info, step to 1 st instruction instead of <i>over</i> .
show step-mode	Print step-mode
finish	Stop after selected frame returns.
return [expr]	Return from selected frame.
until	Run until after current line but stop before current frame returns.
until location	Run to specified location but stop after current frame returns.
advance location	Run to specified location but stop after current frame exits.



next

cadence

255 © Cadence Design Systems, Inc. All rights reserved.

When debugging with GDB, the 8th step is to step the program by line or by instruction, stepping into or over function calls.

- The command “**step**”, steps one or more lines, stopping at the first instruction of the target source line, and entering only those functions that have line number information. You can modify this behavior by setting the step mode “on”. The command also stops stepping upon encountering a breakpoint.
- The command “**next**”, steps one or more lines, stopping at the first instruction of the target source line, and executing function calls as one step. The command also stops stepping upon encountering a breakpoint.
- The command “step instruction” (**stepi**), steps one or more instructions. The command also stops stepping upon encountering a breakpoint.
- The command “next instruction” (**nexti**), steps one or more instructions, executing function calls as one instruction. The command also stops stepping upon encountering a breakpoint.
- The command “set step mode on” (**set step-mode on**), causes the step command, for functions not having line number information, to step to the first function instruction, instead of over the function. The argument “on” is assumed. To set the step mode off, you must specify the argument “off”.
- The command “show step-mode” (**show step-mode**), shows the current step mode.
- The command “**finish**”, runs to the point at which the selected frame has returned.
- The command “**return**”, returns immediately from the selected frame, optionally updating the return value, and discarding the selected and inner frames.
- The command “**until**”, with no argument, runs until after the current line, for example, to complete multiple loop iterations, but does not exit the current frame.
- The command “**until**”, with an argument, runs until the specified location, for example, to complete recursive function calls, but if the location is not reached, stops upon the current frame return.
- The command “**advance**”, runs until the specified location, but if the location is not reached within the current frame, stops upon the current frame exit.

The example command steps one source line, treating a function call as one step.



Quick Reference Guide: GDB Options Choosing Files

gdb [other options] [executable-file [core-file | process-id]]
gdb [other options] --args executable-file [executable's-arguments]

Option (long)	Short	Argument	Description
-core	-c	file	Core dump file
-directory	-d	directory	Add directory to source/script search path
-eval-command	-ex	command	Execute inline command after loading executable
-exec	-e	file	Executable file. Can combine with -s as -se
-init-eval-command	-iex	command	After init eval inline command before loading executable
-init-command	-ix	file	After init eval file commands before loading executable
-pid	-p	number	Connect to process
-readnow	-r		Read symbol table now (rather than as needed)
-symbols	-s	file	Symbol table file. Can combine with -e as -se
-command	-x	file	Execute file commands after loading executable

Processing order is: interpreter; system-wide .gdbinit (if configured); ~/.gdbinit; -i[e]x files, other options; ./gdbinit; program scripts; -[e]x files; command history file.

256 © Cadence Design Systems, Inc. All rights reserved.



These options, upon invocation, specify debugger inputs.

People mostly invoke the GNU debugger with an executable file, though you can alternatively load files later.

If debugging a core dump, you can also specify a core-dump file, or if debugging a running program, you can also specify the program's process identifier number.

Arguments after the double-dash arguments option are not processed, but instead passed to the program being debugged.

You can abbreviate any option to its shortest unique string, or simply enter instead the short option form, which is not always a proper abbreviation.

Free Software Foundation "Debugging with GDB" Tenth Edition, for gdb version 7.6



Quick Reference Guide: GDB Options Choosing Modes

Option (long)	Short	Argument	Description
<code>-annotate</code>		<code>level</code>	Set annotation level (0-3) for control programs [0]
<code>-baud</code>	<code>-b</code>	<code>bps</code>	Remote serial interface line speed
<code>-batch</code>			Run in batch mode (no command confirmation or screen manipulation)
<code>-batch-silent</code>			Run in batch mode with no standard output
<code>-cd</code>		<code>directory</code>	Use directory as CWD
<code>-data-directory</code>		<code>directory</code>	Use directory as data directory (searched for auxiliary files)
<code>-fullname</code>	<code>-f</code>		Output full file name and line number when outputting a stack frame
<code>-interpreter</code>		<code>interpreter</code>	Use interpreter as interface to control programs
<code>-l</code>		<code>timeout</code>	Remote interface timeout
<code>-nx</code>	<code>-n</code>		Not to execute any .gdbinit file
<code>-nh</code>			Not to execute ~/.gdbinit file
<code>-nowindows</code>	<code>-nw</code>		Not to use any built-in GUI
<code>-quiet, -silent</code>	<code>-q</code>		SUPPRESS header and copyright
<code>-return-child-result</code>			Return with child process exit status
<code>-statistics</code>			Output time and memory usage after each command
<code>-tty</code>	<code>-t</code>	<code>device</code>	Redirect standard input/output to device
<code>-tui</code>			Activate the Text User Interface
<code>-version</code>			Output version and exit
<code>-windows</code>	<code>-w</code>		Use built-in GUI if available
<code>-write</code>			Enable writing executable and core files

257 © Cadence Design Systems, Inc. All rights reserved.



These options, upon invocation, set various operational modes.

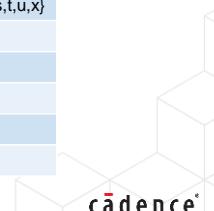
Free Software Foundation “Debugging with GDB” Tenth Edition, for gdb version 7.6



Quick Reference Guide: Common GDB Commands

Command	Abbrev	Description
<code>attach process_id</code>		Attach to a running process
<code>backtrace [full] [[-] count]</code>	<code>bt</code>	Display stack frames, optionally also variables
<code>break [*addr func line] [if condition]</code>	<code>b</code>	Set a breakpoint
<code>call expr</code>		Call a program function
<code>continue [ignore-count]</code>	<code>c</code>	Continue execution, optionally ignoring n next breaks
<code>delete [breakpoints] [range...]</code>	<code>d</code>	Delete breakpoint(s)
<code>detach</code>		Detach program
<code>file filename</code>		Executable and symbols for attached programs
<code>handle {name num range all} actions</code>		Handle signals. An action is one of [no]ignore, [no]pass, [no]print, [no]stop
<code>help [class command [subcommand]]</code>	<code>h</code>	Help with commands
<code>list [+ - m,[n] [m],n linespec]</code>	<code>l</code>	List source lines. <code>linespec::=[+ -]n *addr [file:]{n func} func[:label] label</code>
<code>{next nexti} [count]</code>	<code>n,ni</code>	Like step but treats calls as one step
<code>print [/f] [expr]</code>		Print expression (new or previous) value. Format is one of {a,c,d,f,o,r,s,t,u,x}
<code>quit [expr]</code>	<code>q</code>	Quit GDB, optionally returning status
<code>run [arguments]</code>	<code>r</code>	Run program
<code>set [var[iable]] variable = expr</code>		Assign variable. Include "variable" if var matches a subcommand.
<code>{step stepi} [count]</code>	<code>s,si</code>	Step program by line or instruction
<code>{up down} [count], frame</code>	<code>f</code>	Move up/down n frames, print current frame

258 © Cadence Design Systems, Inc. All rights reserved.



These commands are taken from the Free Software Foundation “Debugging with GDB” 10th Edition, for gdb version 7.6

This incomplete list is for your reference purposes. You should examine it briefly and note its location for future use.

Module Summary

In this module, you

- Diagnosed incorrect program behavior

This training module discussed:

- What Is a Bug?
- What Is Debugging?
- Debugging Approaches
- Debugging with GDB

259 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to:

- Diagnose incorrect program behavior.

To help you to achieve your objective, this training module discussed:

- What Is a Bug?
- What Is Debugging?
- Debugging Approaches; and
- Debugging with GDB.

Quiz: Debugging



What GCC option *must* you include to build symbol tables for use by the GDB symbolic source-level debugger?



How would you “connect” GDB to an already executing program?



GDB has over 1000 commands, most with at least some arguments or options.

What does DDD provide to greatly facilitate your debug efforts?

Please pause here to consider these questions.



Lab

Lab 16-1 Debugging a Program

Objective:

- To diagnose incorrect program behavior

For this lab, you:

- Watch nodes become added to the list as your program checks out objects
- Watch nodes become checked back into the pool



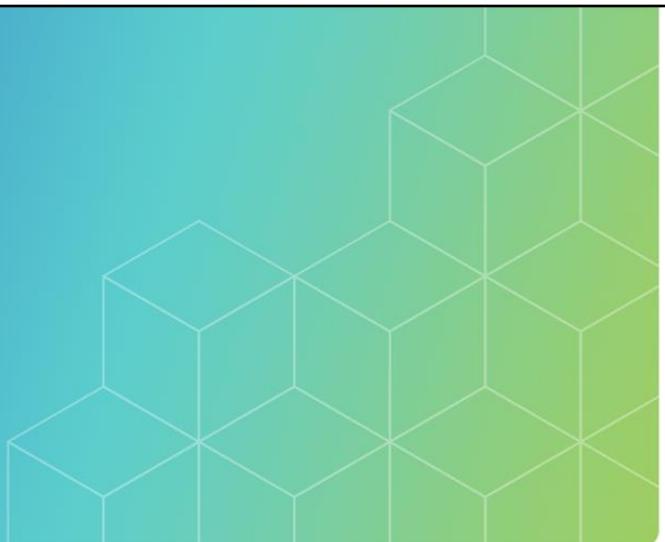
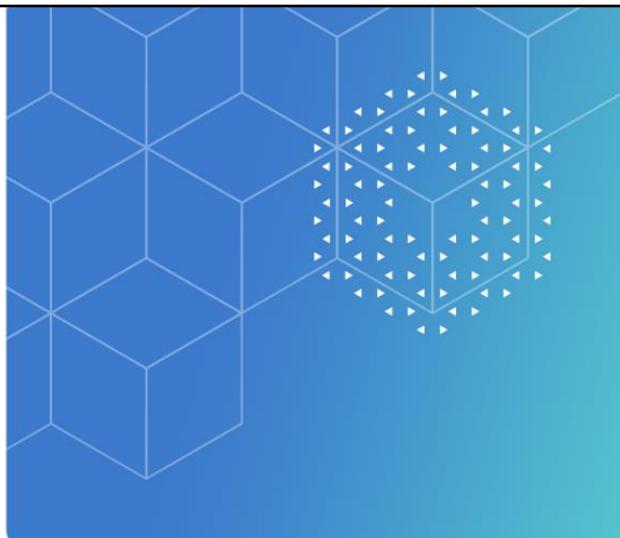
261 © Cadence Design Systems, Inc. All rights reserved.

Your objective for this lab is to:

- Diagnose incorrect program behavior.

For this lab, you:

- Watch nodes become added to the list as your program checks out objects; and
- Watch nodes become checked back into the pool.



Module 17

Containers and Algorithms

cadence®

This training module presents much information, mostly for your reference, about standard containers and standard algorithms.

Module Objectives

In this training module, you

- Store and manipulate data by using containers and algorithms

This training module discusses:

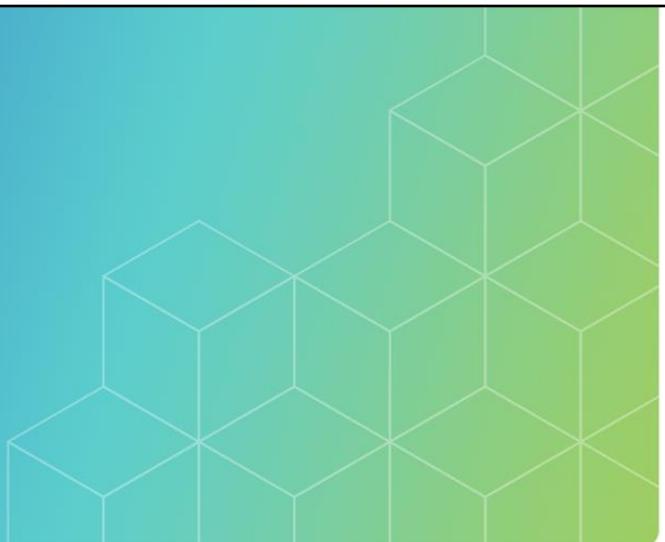
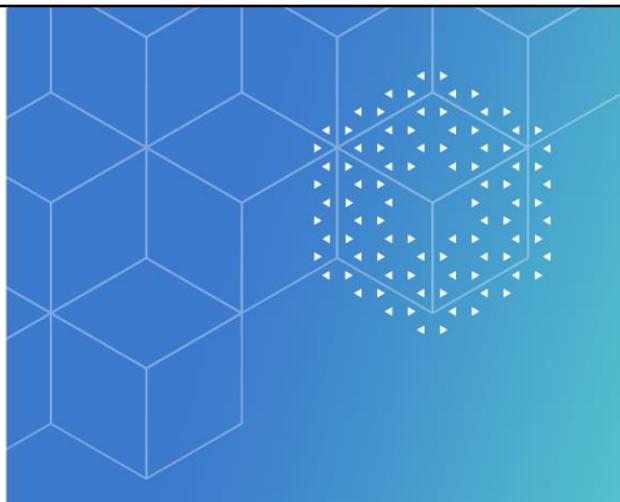
- Standard Containers
- Standard Almost-Containers
- Standard Algorithms



Your objective is to store and manipulate data by using containers and algorithms.

To help you to achieve your objective, this training module discusses:

- Standard Containers;
- Standard Almost-Containers; and
- Standard Algorithms.



Submodule 17-1

Standard Containers

cadence®

This training submodule describes standard containers.

Submodule Objectives

In this training submodule, you

- Store and manipulate data by using standard containers

This training submodule discusses:

- Sequence Containers
- Associative Containers



Your objective is to store and manipulate data by using standard containers.

To help you to achieve your objective, this training module discusses:

- Sequence Containers; and
- Associative Containers.

What Is a Standard Container?

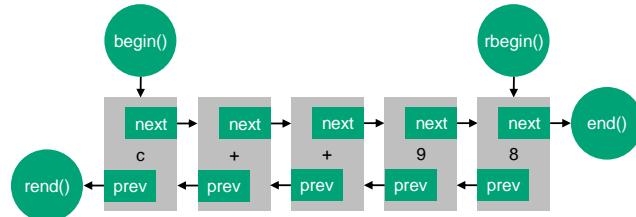


A **Standard Container** is an object that stores other objects, and controls allocation and deallocation of those objects, while conforming to a well-defined interface.

The standard provides container type templates, each optimized for different use models, and all mostly share a common interface.

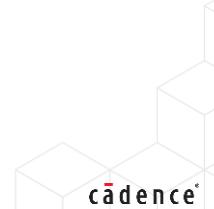
The containers are categorized broadly as:

- Sequentially-accessed containers
 - Used to model static and dynamic arrays, queues, stacks, heaps, linked lists, and trees.
- Associatively-accessed containers
 - Used to model associative arrays and sets.



The standard also provides almost-container type templates not conforming to the standard model.

266 © Cadence Design Systems, Inc. All rights reserved.



A **Standard Container** is an object that stores other objects, and controls allocation and deallocation of those objects, while conforming to a well-defined interface.

The standard provides container type templates, each optimized for different use models, and all mostly sharing a common interface.

The containers are categorized broadly as:

- Sequentially-accessed containers, used to model static and dynamic arrays, queues, stacks, heaps, linked lists, and trees; and
- Associatively-accessed containers, used to model associative arrays and sets.

The standard also provides almost-container type templates not conforming to the standard model.

The illustration is of a doubly-linked list. A doubly-linked list supports iteration in both directions and inserting and erasing at either end or anywhere between the ends.



Quick Reference Guide: Standard Container Common Features

Category	Container	Iteration	[]	Insert/Erase (at front)	Insert/Erase (at back)	Insert/Erase (at iter)	Search
Sequence Containers	array	Bidirectional	O(1)	—	—	—	O(n)
	vector	Bidirectional	O(1)	—	O(1)	O(n)	O(n)
	deque	Bidirectional	O(1)	O(1)	O(1)	O(n)	O(n)
	list	Bidirectional	—	O(1)	O(1)	O(1)	O(n)
	forward_list	Forward	—	O(1)	—	O(1)	O(n)
Sequence Container Adaptors	stack	—	—	—	O(1) push/pop	—	—
	queue	—	—	O(1) pop	O(1) push	—	—
	priority_queue	—	—	—	O(log(n))	—	—
Ordered Associative Containers	map	Bidirectional	O(log(n))	—	—	O(1)	O(log n)
	multimap	Bidirectional	—	—	—	O(1)	O(log n)
	set	Bidirectional	—	—	—	O(1)	O(log n)
	multiset	Bidirectional	—	—	—	O(1)	O(log n)
Unordered Associative Containers	unordered_map	Forward	O(1)	—	—	O(1)	O(1)
	unorder_multimap	Forward	—	—	—	O(1)	O(1)
	unordered_set	Forward	—	—	—	O(1)	O(1)
	unordered_multiset	Forward	—	—	—	O(1)	O(1)

Vector/Deque search complexity is O(log n) for algorithms requiring a sorted container.

267 © Cadence Design Systems, Inc. All rights reserved.



The standard library provides sequence containers and associative containers.

The sequence container structures differ to provide more or less efficiency for different types of operations.

Software engineers use the “big O” notation to describe algorithm complexity, that is, how much time an operation requires as a function of element count.

Assuming that the container contains arbitrarily many elements:

- Constant (O(1)) time complexity is constant regardless of the number of elements, so is least expensive, but keep in mind that the constant is higher for some structures than for others;
- Logarithmic (O(log(n))) time complexity grows proportionately to the base 2 logarithm of the number of elements, so is inexpensive relative to;
- Linear (O(n)) time complexity grows proportionately to the number of elements, so is relatively expensive.

To maximize project performance, you will want to use the simplest container that meets your needs:

- If randomly writing and reading elements, if statically sized, you might choose an **array**, and if dynamically sized, a **vector**.
- If randomly inserting and removing elements, you might choose a **list**.
- If doing only back push and pop operations, you might choose a **stack**, which is an adapter for, by default, a double-ended queue (**deque**).
- If doing both back and front push and pop operations then you might choose a **queue**, which is an adapter for, by default, a double-ended queue.
- If you need back and front operations, and also want to randomly write and read elements, then you might choose the double-ended queue itself.

Vector and double-ended queue insertion at ends complexity is an amortized constant that can degenerate to linear in the theoretical worst case that all insertions require reallocation.

Unordered associative container operation complexity can degenerate to linear in the theoretical worst case that all keys hash to the same bucket.



Quick Reference Guide: Standard Container Common Types

Type	Description
<code>allocator_type</code>	[allocator-aware] Type of memory manager
<code>difference_type</code>	Type of iterator difference
<code>iterator</code> <code>const_iterator</code>	Behaves like <code>value_type*</code> Behaves like <code>const value_type*</code>
<code>reverse_iterator</code> <code>const_reverse_iterator</code>	[reversible] Reverse order iterators
<code>pointer</code> <code>const_pointer</code>	Pointers to container elements <code>value_type*</code> <code>const value_type*</code>
<code>reference</code> <code>const_reference</code>	References to container elements <code>value_type&</code> <code>const value_type&</code>
<code>size_type</code>	Type of count and subscript
<code>value_type</code>	Type of element

268 © Cadence Design Systems, Inc. All rights reserved.



All standard containers define the types:

- Type “allocator type” (**allocator_type**), which is the memory manager type;
 - This applies to only the containers that have allocators.
- Type “difference type” (**difference_type**), which is the type of the difference between iterators;
- Types “iterator” and “constant iterator” (**const_iterator**), which is the type of the iterator;
- Types “reverse iterator” (**reverse_iterator**) and “constant reverse iterator” (**const_reverse_iterator**), which is the type of the reverse iterator;
 - This applies to only the containers that have reverse iterators.
- Types “pointer” and “constant pointer” (**const_pointer**) that behave as pointers to container elements;
- Types “reference” and “constant reference” (**const_reference**) that behave as references to container elements;
- Type “size type” (**size_type**) that is the type of the container size and index; and
- Type “value type” (**value_type**), that is the type of a container element.



Quick Reference Guide: Standard Container Common Functions

Type	Function	Parameters	Description
-	c	([const Allocator&])	Constructor (default)
-	c	(const C& [, const Allocator&]) (const C&& [, const Allocator&])	Constructor (copy) Constructor (move)
-	~c	()	Destructor
allocator_type	get_allocator	()	Get allocator
[const_]iterator const_iterator	begin, end cbegin, cend	() ()	Iterators
[const_]reverse_iterator const_reverse_iterator	rbegin, rend crbegin, crend	() ()	Iterators – reverse
bool	empty	()	Is empty
size_type	max_size	()	Size limit
size_type	size	()	Size
Class&	operator=	(const C&)	Assignment
bool	operator==, !=	(const C&)	[In]Equality
strong_ordering	operator<=>	(const_iterator, const_iterator) (const container&, const container&)	Compare iterators Optionally compare container
void	swap	(C&) (C&, C&)	Swap elements (member) Nonmember swap elements

Arrays have no user constructors and no allocators. Forward lists have no reverse iterators and no function size()
Unordered associative containers have no reverse iterators.

269 © Cadence Design Systems, Inc. All rights reserved.



All standard containers provide:

- Constructor functions to create an empty container and to initialize a new container with the contents of an existing container.
- A destructor to destroy the container.
- The function “get allocator” (**get_allocator**), for allocator-aware containers, to get a reference to the allocator object.
 - Array containers have no allocator.
- Constant and non-constant iterators, and functions returning iterators to the container beginning and ending, and for reversible containers, likewise, constant and non-constant reverse iterators, and functions returning reverse iterators to the container beginning and ending.
 - Forward lists have no reverse iterators.
- The function “empty” to return the truth of whether the container is empty.
- The function “maximum size” (**max_size**) to return the maximum supported size for that container.
- The function “size” to return the current size of the container.
 - Forward list containers have no “size” function.
- An assignment operator to make a container’s elements equal to the elements of some other container.
- Equality and inequality operators to return the truth of whether a container’s elements are equal to the elements of some other container.
- A nonmember comparison operator to determine the relationship between iterators, and optionally to determine the relationship between containers.
- Member and nonmember functions “swap” to swap all elements between two containers.

What Is a Sequence Container?



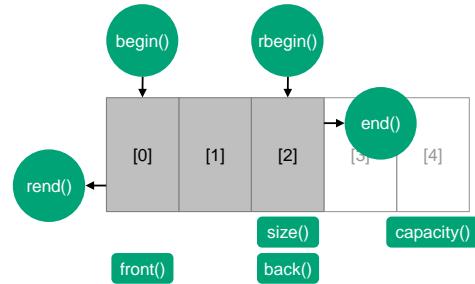
A **Sequence Container** is a standard container that stores elements sequentially and offers standard common features in addition to those common to all standard containers.

Use sequence containers to model sequentially-accessed containers.

- Static and dynamic arrays, queues, stacks, heaps, linked lists, and trees

Choose the sequence container optimized for your usage model.

- Size static versus dynamic
- Insertion at ends versus randomly
- Iteration unidirectional versus bidirectional



270 © Cadence Design Systems, Inc. All rights reserved.

A Standard Container is an object that stores other objects, and controls allocation and deallocation of those objects, while conforming to a well-defined interface.

A **Sequence Container** is a standard container that stores elements sequentially and offers standard common features in addition to those of a standard container.

Use sequence containers to model sequentially-accessed containers, such as static and dynamic arrays, queues, stacks, heaps, linked lists, and trees.

Choose the sequence container optimized for your usage model.

- Will its size remain static, or does it dynamically change?
- Will you insert elements only at the ends, or at any random location?
- Will you iterate only forward, or in both directions?

The illustration is of a **vector**. A vector is a sequence of contiguous elements, resizable, and supporting subscript operations, and insertion and deletion at its ends and randomly. Vectors are the only sequence containers for which you can reserve capacity beyond the container's current size.



Quick Reference Guide: Sequence Container Common Functions

Type	Function	Parameters	Description
-	C	(size_type, const Allocator& = Allocator())	Constructor (initializer – n elements of default value)
-	C	(size_type, const T&, const Allocator& = Allocator())	Constructor (initializer – n elements of t)
-	C	(InputIterator, InputIterator, const Allocator& = Allocator())	Constructor (initializer – from container range)
-	C	(initializer_list<T>, const Allocator& = Allocator())	Constructor (initializer – from initializer list)
void	assign	(size_type, const T&) (InputIterator, InputIterator) (initializer_list<T>)	Assign to container n copies of t Assign to container from range Assign to container from initializer list
void	clear	()	Clear the container
iterator	emplace	(const_iterator, Args&&...)	Emplace at position newly constructed element
iterator	erase	(const_iterator) (const_iterator, const_iterator)	Erase element at position Erase elements in range
iterator	insert	(const_iterator, const T&) (const_iterator, T&&) (const_iterator, size_type, const T&) (const_iterator, InputIterator, InputIterator) (const_iterator, initializer_list<T>)	Insert at position one copy of t Insert at position one move of t Insert at position n copies of t Insert at position elements from range Insert at position element list
C&	operator=	(initializer_list<T>)	Assign to container from initializer list

These functions are in addition to the functions common to all standard containers.

271 © Cadence Design Systems, Inc. All rights reserved.



All standard containers support simple constructors, some query functions, simple relational operators, and iterators.

In addition to the functions common to all standard containers, sequence containers provide additional functions.

- Initialization constructors to initialize a newly constructed container with either a number of copies of an element, from another container's range of elements, or from an initializer list;
- The function “**assign**” to assign to a container either a number of copies of an element, from another container's range of elements, or from an initializer list;
- The function “**clear**” to clear the container.
- The function “**emplace**” to emplace at a position an element newly created using the provided arguments;
- The function “**erase**” to erase either one element or a range of elements;
- The function “**insert**” to insert at a position either one or a specified number of copies of an element, or copies of elements from a range or initializer list; and
- An assignment operator to assign to the container from an initializer list.

I don't see where C(size_type n, const Allocator& = Allocator()) is stated as a requirement but all allocator-aware sequence containers implement it.

Declaring and Using Arrays



```
std::array<typename, size> identifier  
array<char,32> my_array;
```

Arrays support only some sequence container operations.

- No user constructors
- Iteration: forward/backward
- Access: front(), back(), at(), []
- Container assignment
- Insert/erase by iterator
- Emplace front/back/iterator
- Push/pop front/back
- Miscellaneous common operations
- Array-specific operations



```
const unsigned size = 32;  
using container_t = array<char,size>;  
char data;  
container_t cont;  
cout << cont.size() << endl; // 32  
for (unsigned addr=0;addr<size;++addr)  
{  
    data = (char)(97+rand()%26);  
    write(addr,data);  
    cont[addr] = data;  
}  
cout << cont << endl; // user-defined  
for (unsigned addr=0;addr<size;++addr)  
{  
    read(addr,data);  
    assert(data == cont[addr]);  
}  
cout << cont.empty() << endl; // 0
```

cadence

272 © Cadence Design Systems, Inc. All rights reserved.

An **array** is implemented as a monolithic object in contiguous memory.

You can think of it as a C array wrapper providing some container features.

Arrays support some sequence operations, but most notably not the insertion or front and back operations.

Choose an array if you randomly write and read elements and the container size is fixed during translation.

The example:

- Declares an array of 32 characters;
- Outputs the container size;
- In a loop, writes random data to a memory, while storing the data in the container;
- Outputs the container, using a user-defined output stream insertion operator;
- In a loop, reads data from the memory, while confirming that the data matches the container data; and
- Outputs the truth of whether the container is empty.



Quick Reference Guide: Array Functions

Type	Function	Parameters	Description
<code>reference</code> <code>const_reference</code>	<code>at</code>	<code>(size_type)</code>	Element at index
<code>reference</code> <code>const_reference</code>	<code>back</code>	<code>()</code>	Element at back
<code>reference</code> <code>const_reference</code>	<code>front</code>	<code>()</code>	Element at front
<code>T*</code> <code>const T*</code>	<code>data</code>	<code>()</code>	Pointer to front data
<code>void</code>	<code>fill</code>	<code>(const T&)</code>	Fill with specified value
<code>reference</code> <code>const_reference</code>	<code>operator[]</code>	<code>(size_type)</code>	Element at index

These functions are in addition to the functions common to all sequence containers.



Sequence containers generally support construction and simple emplacing, inserting, and erasing elements, and clearing and assigning the container.

As an optimized wrapper of a static C array, of these functions, an array container supports only container assignment.

In addition to other functions common to sequential containers, array containers provide additional functions.

- The function “**at**” to return the element at the specified index that is bounds-checked;
- The function “**back**” to return the element at the container back;
- The function “**front**” to return the element at the container front;
- The function “**data**” to return a pointer to the first element;
 - Only arrays and vectors have the function “data”.
- The function “**fill**” to fill the container with the specified value;
 - Only arrays have the function “fill”.
- The subscript operator to return the element at the specified index that is not bounds-checked.

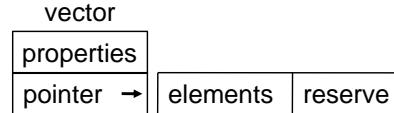
Declaring and Using Vectors



```
std::vector<typename, allocator_opt> identifier
vector<char> my_vector;
```

Vectors support all sequence container operations except front operations.

- Constructors
- Iteration: forward/backward
- Access: front(), back(), at(), []
- Container assignment
- Insert/erase by iterator
- Emplace front/back/iterator
- Push/pop front/back
- Miscellaneous common operations
- Vector-specific operations



```
const unsigned size = 32;
using container_t = vector<char>;
container_t cont; char data;
cont.resize(size);
cout << cont.size() << endl; // 32
for (int addr=0;addr<size;++addr)
{
    data = (char)(97+rand()%26);
    write(addr,data);
    cont[addr] = data;
}
cout << cont << endl; // user-defined
for (int addr=0;addr<size;++addr)
{
    read(addr,data);
    assert(data == cont[addr]);
}
cont.resize(0);
cout << cont.empty() << endl; // 1
```

cadence

274 © Cadence Design Systems, Inc. All rights reserved.

A **vector**, unless specialized for type Boolean (**bool**), is implemented as a monolithic object in contiguous memory.

You can think of it as an array that you can dynamically resize, and that resizes[†] itself as a result of assignment or insertions or erasure.

Vectors support all sequence operations except for the front operations.

Vectors are the only sequence containers that support reservation of additional uninitialized capacity for future growth. If a vector is implemented in contiguous memory, then resizing the vector really means a three-step process of allocating new heap space, copying all the elements, and then deleting the old heap space. Reserving sufficient capacity in advance eliminates this expensive reallocation. The reserve capacity is not part of the array until it is used.

Choose a vector if you randomly write and read elements and the container size changes infrequently.

The example:

- Declares an empty vector of characters;
- Resizes the container to 32;
- Outputs the container size;
- In a loop, writes random data to a memory, while storing the data in the container;
- Outputs the container, using a user-defined output stream insertion operator;
- In a loop, reads data from the memory, while confirming that the data matches the container data;
- Resizes the container to zero; and
- Outputs the truth of whether the container is empty.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.



Quick Reference Guide: Vector Functions

Type	Function	Parameters	Description
<code>size_type</code>	<code>capacity</code>	<code>()</code>	Total capacity
<code>void</code>	<code>reserve</code>	<code>(size_type)</code>	Required capacity
<code>void</code>	<code>resize</code>	<code>(size_type) (size_type, const T&)</code>	Erase or append elements of value default Erase or append elements of value t
<code>void</code>	<code>shrink_to_fit</code>	<code>()</code>	Make <code>capacity() == size()</code> – nonbinding!
<code>reference const_reference</code>	<code>at</code>	<code>(size_type)</code>	Element at index
<code>reference const_reference</code>	<code>back</code>	<code>()</code>	Element at back
<code>reference const_reference</code>	<code>front</code>	<code>()</code>	Element at front
<code>T*, const T*</code>	<code>data</code>	<code>()</code>	Pointer to front data
<code>reference const_reference</code>	<code>operator[]</code>	<code>(size_type)</code>	Element at index
<code>reference</code>	<code>emplace_back</code>	<code>(Args&&...)</code>	Emplace on back new element constructed using args
<code>void</code>	<code>flip</code>	<code>()</code>	Flip bits (vector<bool> only)
<code>void</code>	<code>push_back</code>	<code>(const T&), (T&)</code>	Push copy/move t onto back
<code>void</code>	<code>pop_back</code>	<code>()</code>	Pop element from back

These functions are in addition to the functions common to all sequence containers.

275 © Cadence Design Systems, Inc. All rights reserved.



Sequence containers generally support construction, and simple element emplacement, insertion, and erasure, and clearing and assigning the container.

In addition to other functions common to sequential containers, vector containers provide additional functions.

- The function “**capacity**” to query their total capacity;
 - Only vectors have the function “capacity”.
- The function “**reserve**” to state an anticipated required capacity;
 - Only vectors have the function “reserve”.
- The function “**resize**” to resize the container, potentially erasing or appending elements;
- The function “shrink to fit” (**shrink_to_fit**) to potentially reduce the capacity to the current size. The function is non-binding to allow the implementation to ignore it in favor of optimization.
 - Only vectors and double-ended queues have the function “shrink to fit”.
- The function “**at**” to return the element at the specified index that is bounds-checked;
- The function “**back**” to return the element at the container back;
- The function “**front**” to return the element at the container front;
- The function “**data**” to return a pointer to the first element;
 - Only arrays and vectors have the function “data”.
- The subscript operator to return the element at the specified index that is not bounds-checked;
- The function “emplace back” (**emplace_back**) to construct a new element from provided arguments and append it to the container back;
- The function “**flip**” to “flip” the bits of a vector specialized for type Boolean (bool);
- The function “push back” (**push_back**) to push an element onto the container back;
- The function “pop back” (**pop_back**) to pop an element from the container back.

Vectors do not support front operations.

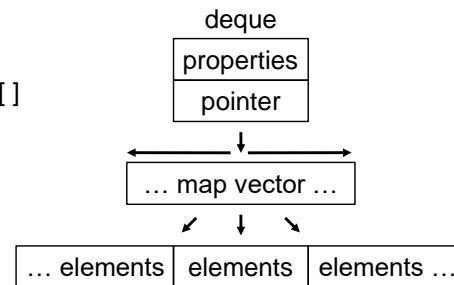
Declaring and Using Double-Ended Queues



```
std::deque<typename, allocator_opt> identifier
deque<char> my_deque;
```

Double-ended queues support all sequence container operations.

- Constructors
- Iteration: forward/backward
- Access: front(), back(), at(), []
- Container assignment
- Insert/erase by iterator
- Emplace front/back/iterator
- Push/pop front/back
- Miscellaneous common operations



```
const unsigned size = 32;
using container_t = deque<char>;
container_t cont; char data;
for (int addr=0;addr<size;++addr) {
    data = (char)(97+rand()%26);
    write(addr,data);
    cont.push_back(data);
}
cout << cont.size() << endl; // 32
cout << cont << endl; // user-defined
for (int addr=0;addr<size;++addr) {
    read(addr,data);
    assert(data == cont.front());
    cont.pop_front();
}
cout << cont.empty() << endl; // 1
```

276 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The **double-ended queue (deque)** is typically but not necessarily implemented as one or more monolithic objects in contiguous memory, that automatically resizes[†] itself as a result of assignment or insertions or erasures. The time-complexity requirements suggest multiple fixed-sized vectors indexed by a vector offset to grow at both ends.

Double-ended queues support all sequence operations. They combine relatively efficient subscript operations with relatively efficient front and back operations.

A **queue** is an adapter for a container, by default a double-ended queue, that suppresses the subscript, list, and iterator operations. A **stack** suppresses also the front operations.

Choose a double-ended queue if you randomly write and read elements and you frequently insert or remove elements only at the container's end positions.

If you do not randomly write and read elements, then you might by convention choose to instead use a **queue**. A queue is an adapter for a container, by default a double-ended queue, that suppresses the subscript, list, and iterator operations.

If you also do not use the front operations, then you might by convention choose to instead use a **stack**. A stack is an adapter for a container, by default a double-ended queue, that suppresses the subscript, list, iterator, and front operations.

The example:

- Declares an empty double-ended queue of characters;
- In a loop, writes random data to a memory, while pushing the data onto the container back;
- Outputs the container size;
- Outputs the container, using a user-defined output stream insertion operator;
- In a loop, reads data from the memory, while confirming that the data matches the container front data, and then popping the container front; and
- Outputs the truth of whether the container is empty.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.



Quick Reference Guide: Double-Ended Queue Functions

Type	Function	Parameters	Description
void	resize	(size_type) (size_type, const T&)	Erase or append elements of value default Erase or append elements of value t
void	shrink_to_fit	()	Make capacity()==size() <i>nonbinding!</i>
reference const_reference	at	(size_type)	Element at index
reference const_reference	front, back	()	Element at front, back
reference const_reference	operator[]	(size_type)	Element at index
reference	emplace_front emplace_back	(Args&&...)	Prepend on front new element constructed using arguments Append on back new element constructed using arguments
void	push_front push_back	(const T&), (T&)	Push copy/move t onto front Push copy/move t onto back
void	pop_front pop_back	()	Pop element from front Pop element from back

These functions are in addition to the functions common to all sequence containers.



Sequence containers generally support construction, and simple element emplacement, insertion, and erasure, and clearing and assigning the container.

In addition to other functions common to sequential containers, double-ended queue containers provide additional functions.

- The function “**resize**” to resize the container, potentially erasing or appending elements;
- The function “shrink to fit” (**shrink_to_fit**) to potentially reduce the capacity to the current size;
 - Only vectors and double-ended queues have the function “shrink to fit”. The function is non-binding to allow the implementation to ignore it in favor of optimization.
- The function “**at**” to return the element at the specified index that is bounds-checked;
- The function “**front**” to return the element at the container front. and the function “**back**” to return the element at the container back;
- The subscript operator to return the element at the specified index that is not bounds-checked;
- The function “emplace front” (**emplace_front**) to construct a new element from provided arguments and prepend it to the container front, and the function “emplace back” (**emplace_back**) to construct a new element from provided arguments and append it to the container back;
- The function “push back” (**push_back**) to push an element onto the container back, and the function “push front” (**push_front**) to push an element onto the container front; and
- The function “pop back” (**pop_back**) to pop an element from the container back, and the function “pop front” (**pop_front**) to pop an element from the container front.

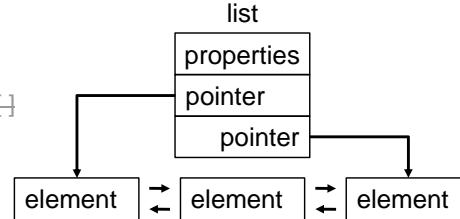
Declaring and Using Lists



```
std::list<typename, allocator_opt> identifier
list<char> my_list;
```

Lists support all sequence container operations except subscript operations.

- Constructors
- Iteration: forward/backward
- Access: front(), back(), at(), []
- Container assignment
- Insert/erase by iterator
- Emplace front/back/iterator
- Push/pop front/back
- Miscellaneous common operations
- List-specific operations



```
const unsigned size = 32;
using container_t = list<char>;
using citer_t =
container_t::const_iterator;
container_t cont; char data;
for(unsigned addr=0;addr<size;++addr){
    data = (char)(97+rand()%26);
    write(addr,data);
    cont.insert(cont.cend(),data);
}
cout << cont.size() << endl; // 32
cout << cont << endl; // user-defined
citer = cont.cbegin();
for(unsigned addr=0;addr<size;++addr){
    assert(citer!=cont.cend());
    read(addr,data);
    assert(data == *citer++);
}
cont.clear();
cout << cont.empty() << endl; // 1
```



278 © Cadence Design Systems, Inc. All rights reserved.

A **list** is typically but not necessarily implemented as a series of doubly-linked elements, that automatically resizes[†] itself as a result of assignment or insertions or erasure.

Lists support all sequence operations except for the subscripting operations.

Choose a list if you frequently insert or remove elements at positions other than the container ends, and have no need for random access.

Choose instead a forward list for higher performance if you have no need for reverse iteration.

The example:

- Declares an empty list of characters;
- In a loop, writes random data to a memory, while inserting the data at the container back;
- Outputs the container size;
- Outputs the container, using a user-defined output stream insertion operator;
- In a loop, from the container beginning to its end, reads data from the memory, while confirming that the data matches the container data;
- Clears the container; and
- Outputs the truth of whether the container is empty.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.



Quick Reference Guide: List Functions

Type	Function	Parameters	Description
<code>void</code>	<code>resize</code>	<code>(size_type)</code> <code>(size_type, const T&)</code>	Erase or append elements of value default Erase or append elements of value t
<code>reference const_reference</code>	<code>front, back</code>	<code>()</code>	Element at front, back
<code>reference</code>	<code>emplace_front</code> <code>emplace_back</code>	<code>(Args&&...)</code>	Prepend on front new element constructed using arguments Append on back new element constructed using arguments
<code>void</code>	<code>push_front</code> <code>push_back</code>	<code>(const T&), (T&&)</code>	Push copy/move t onto front Push copy/move t onto back
<code>void</code>	<code>pop_front</code> <code>pop_back</code>	<code>()</code>	Pop element from front Pop element from back

These functions are in addition to the functions common to all sequence containers.



Sequence containers generally support construction, and simple element emplacement, insertion, and erasure, and clearing and assigning the container.

In addition to other functions common to sequential containers, list containers provide additional functions.

- The function “**resize**” to resize the container, potentially erasing or appending elements;
- The function “**front**” to return the element at the container front, and the function “**back**” to return the element at the container back;
- The function “**emplace front**” (**emplace_front**) to construct a new element from provided arguments and prepend it to the container front, and the function “**emplace back**” (**emplace_back**) to construct a new element from provided arguments and append it to the container back;
- The function “**push back**” (**push_back**) to push an element onto the container back, and the function “**push front**” (**push_front**) to push an element onto the container front; and
- The function “**pop back**” (**pop_back**) to pop an element from the container back, and the function “**pop front**” (**pop_front**) to pop an element from the container front.



Quick Reference Guide: List Operations

Type	Function	Parameters	Description
void	<code>merge</code>	<code>(list&), (list&, Compare)</code> <code>(list&&), (list&&, Compare)</code>	Merge <i>sorted</i> other list elements to <i>sorted this</i> list, maintaining sort
<code>size_type</code>	<code>remove</code> <code>remove_if</code>	<code>(const T&)</code> <code>(Predicate)</code>	Remove elements matching value Remove elements making predicate true
void	<code>reverse</code>	<code>()</code>	Reverse list elements
void	<code>sort</code>	<code>() , (Compare)</code>	Sort the list
void	<code>splice</code>	<code>(const_iterator, list&)</code> <code>(const_iterator, list&&)</code> <code>(const_iterator, list&, const_iterator)</code> <code>(const_iterator, list&&, const_iterator)</code> <code>(const_iterator, list&, const_iterator,</code> <code>const_iterator)</code> <code>(const_iterator, list&&, const_iterator,</code> <code>const_iterator)</code>	Splice other list into <i>this</i> list at position Splice other list <i>element</i> into <i>this</i> list at position Splice other list <i>range</i> into <i>this</i> list at position
<code>size_type</code>	<code>unique</code>	<code>() , (BinaryPredicate)</code>	Uniquify list by removing consecutive duplicated elements



List containers support additional useful operations not common to other sequence containers.

- The function “**merge**” to move the sorted other list elements to appropriate locations of the sorted current list. Sorting can be determined by the less-than operator and can alternatively be determined by a compare object. If by compare object, then the compare class must provide a constructor accepting two iterator parameters, and must provide a conversion operator to generate a Boolean. Results are undefined if either input list is unsorted.
- The function “**remove**” to erase all matching elements. The match can be by value and can alternatively be by a unary predicate function or object. If by predicate, then the predicate class must provide a constructor accepting an iterator parameter, and must provide a conversion operator to generate a Boolean.
- The function “**reverse**” to reverse the list.
- The function “**sort**” to sort the list. Sorting can be determined by the less-than operator and can alternatively be determined by a compare object. If by compare object, then the compare class must provide a constructor accepting two iterator parameters, and must provide a conversion operator to generate a Boolean.
- The function “**splice**” to move one or more list elements in or to the current list, at a position. If moving a list as a whole then it must be some other list. If moving a range of elements then that range must not include the insertion position.
- The function “**unique**” to erase consecutive duplicated elements. Duplication can be determined by equivalence and can alternatively be determined by binary predicate. If by predicate object, then the predicate class must provide a constructor accepting two iterator parameters, and must provide a conversion operator to generate a Boolean. An unsorted list can still retain duplicated elements.

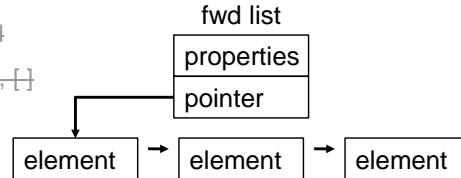
Declaring and Using Forward Lists



```
std::forward_list<typename, allocator_opt> identifier  
forward_list<char> my_forward_list;
```

Forward lists support all list operations except back operations and size().

- Constructors
- Iteration: forward/backward
- Access: front(), back(), at(), []
- Container assignment
- Insert/erase by iterator
- Emplace front/back/iterator
- Push/pop front/back
- Miscellaneous common operations
- Forward-list-specific operations



```

const unsigned size = 32;
using container_t = forward_list<char>;
using iter_t = container_t::iterator;
using citer_t = container_t::const_iterator;
container_t cont; char data;
iter_t iter = cont.before_begin();
for (unsigned addr=0;addr<size;++addr) {
    data = (char)(97+rand()%26);
    write(addr,data);
    cont.insert_after(iter,data);
    ++iter;
}
cout << cont << endl; // user-defined
citer_t citer = cont.cbegin();
for (unsigned addr=0;addr<size;++addr) {
    assert(citer!= cont.cend());
    read(addr,data);
    assert(data == *citer++);
}
cont.clear();
cout << cont.empty() << endl; // 1
  
```

281 © Cadence Design Systems, Inc. All rights reserved.

cadence

A forward list (**forward_list**) is typically but not necessarily implemented as a series of singly-linked elements, that automatically resizes[†] itself as a result of assignment or insertions or erasure.

Forward lists support all sequence operations except for the subscripting operations and back operations and the “size” function.

Forward lists, to improve performance, modify the “emplace”, “erase”, “insert”, and “splice” functions, to apply after the iterator rather than at the iterator.

Choose a list if you frequently insert or remove elements at positions other than the container ends, and have no need for random access.

Choose instead a forward list, for higher performance, if you also have no need for reverse iteration.

The example:

- Declares an empty forward list of characters;
- In a loop, writes random data to a memory, while inserting the data into the container at incremental positions;
- Outputs the container, using a user-defined output stream insertion operator;
- In a loop, from the container beginning to its end, reads data from the memory, while confirming that the data matches the container data;
- Clears the container; and
- Outputs the truth of whether the container is empty.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.



Quick Reference Guide: Forward List Functions (Modified)

Type	Function	Parameters	Description
iterator	<code>emplace_after</code>	(<code>const_iterator</code> , <code>Args&&...</code>)	Insert after position new element constructed using arguments
iterator	<code>erase_after</code>	(<code>const_iterator</code>) (<code>const_iterator</code> , <code>const_iterator</code>)	Erase element after position Erase elements in range
iterator	<code>insert_after</code>	(<code>const_iterator</code> , <code>const T&</code>) (<code>const_iterator</code> , <code>T&&</code>) (<code>const_iterator</code> , <code>size_type</code> , <code>const T&</code>) (<code>const_iterator</code> , <code>initializer_list<T></code>) (<code>const_iterator</code> , <code>InputIterator</code> , <code>InputIterator</code>)	Insert after position one element Insert after position one element Insert after position n elements Insert after position element list Insert after position elements from other container range
void	<code>splice_after</code>	(<code>const_iterator</code> , <code>forward_list&</code>) (<code>const_iterator</code> , <code>forward_list&&</code>) (<code>const_iterator</code> , <code>forward_list&</code> , <code>const_iterator</code>) (<code>const_iterator</code> , <code>forward_list&&</code> , <code>const_iterator</code>) (<code>const_iterator</code> , <code>forward_list&</code> , <code>const_iterator</code> , <code>const_iterator</code>) (<code>const_iterator</code> , <code>forward_list&&</code> , <code>const_iterator</code> , <code>const_iterator</code>)	Splice other list into <code>this</code> list at position Splice other list <i>element</i> into <code>this</code> list at position Splice other list <i>range</i> into <code>this</code> list at position

These functions replace the similar list container functions.



Forward lists, to improve performance, modify the “emplace”, “erase”, “insert”, and “splice” functions, to apply *after* the iterator rather than *at* the iterator.

- The function “emplace after” (`emplace_after`), replaces the list function “emplace”, to construct a new element from provided arguments and insert it after the provided position;
- The function “erase after” (“`erase_after`”), replaces the list function “erase”, to erase either one element after a position or a range of elements between positions;
- The function “insert after” (`insert_after`), replaces the list function “insert”, to insert after a position either one element, or a number of copies of one element, or the elements of an initializer list or a range of some other container;
- The function “splice after” (`splice_after`), replaces the list function “splice”, to move one or more list elements in or to the current list, after a position. If moving a list as a whole then it must be some other list. If moving a range of elements then that range must not include the insertion position.

Forward lists have no back operations and no “size” function.

What Is a Container Adaptor?

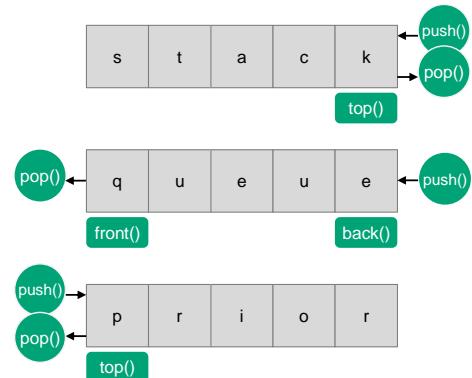


A **Container Adaptor** instantiates, protects, and wraps a container, and to simplify its usage, limits the public interface.

The standard currently defines adaptors for only sequence containers.

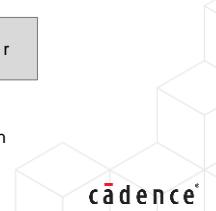
Adaptor:	stack	queue	priority_queue
Default Container	deque	deque	vector
Access	top()_back	back(), front()	top()_front
Modify	push()_back pop()_back	push()_back pop()_front	push()_back pop()_back

A priority queue is maintained in the order by default descending.
Adaptors are not containers, so don't expect container operations!



Priority queue push/pop act upon front due to element swapping.

283 © Cadence Design Systems, Inc. All rights reserved.



A **Container Adaptor** instantiates, protects, and wraps a container, and to simplify their usage, limits the public interface.

The standard currently defines adaptors for only sequence containers.

The underlying container is by default:

- For a **stack**, a double-ended queue (**deque**), but you can use any sequence container defining the functions “**back**”, “push back” (**push_back**), and “pop back” (**pop_back**).
- For a **queue**, a double-ended queue (**deque**), but you can use any sequence container defining the functions “**back**”, “**front**”, “push back” (**push_back**), and “pop front” (**pop_front**).
- For a priority queue (**priority_queue**), a **vector**, but you can use any sequence container defining the functions “**front**”, “push back” (**push_back**), and “pop back” (**pop_back**).
 - A priority queue is automatically maintained in the order by default descending, but you can provide a comparison function or function object to order the queue any way that you want,

For all of these adaptors, the underlying container type is a template parameter. For the priority queue, the comparison class is also a template parameter.

Declaring and Using Container Adaptors

```
std::stack<value_type, Container_opt> identifier
std::queue<value_type, Container_opt> identifier
std::priority_queue<value_type, Container_opt, Compare_opt> identifier
```

Adaptor:	stack	queue	priority _queue
Header	<stack>	<queue>	<queue>
Default Container	deque	deque	vector
Access	top()_back	back(), front()	top()_front
Modify	push()_back pop()_back	push()_back pop()_front	push()_back pop()_back

A priority queue is maintained in the order by default descending.
Adaptors are not containers so don't assume container operations.

```
priority_queue<char> prq; queue<char> que; stack<char> stk;
prq.push('W'); que.push('e'); stk.push('+');
prq.push('L'); que.push('u'); stk.push('v');
prq.push('C'); que.push('+'); stk.push(' ');
cout << "prq.size() is " << prq.size() << endl; // 3
cout << "que.size() is " << que.size() << endl; // 3
cout << "stk.size() is " << stk.size() << endl; // 3
cout << "prq.empty() is " << prq.empty() << endl; // false
cout << "que.empty() is " << que.empty() << endl; // false
cout << "stk.empty() is " << stk.empty() << endl; // false
cout << "prq.top() is " << prq.top() << endl; // W
cout << "que.front() is " << que.front() << endl; // e
cout << "que.back() is " << que.back() << endl; // +
cout << "stk.top() is " << stk.top() << endl; // '
cout << prq.top() << que.front() << stk.top() << endl; // We
prq.pop(); que.pop(); stk.pop();
cout << prq.top() << que.front() << stk.top() << endl; // Luv
prq.pop(); que.pop(); stk.pop();
cout << prq.top() << que.front() << stk.top() << endl; // C++
prq.pop(); que.pop(); stk.pop();
```

284 © Cadence Design Systems, Inc. All rights reserved.



The adaptors are class templates that take a “value type” argument, an optional “container type” argument, and for priority queues, an optional “compare type” argument.

- The default container type is, for stacks and queues, the double-ended queue, and for priority queues, the vector. You can alternatively use any container that implements the required functions.
- The default compare type is, for priority queues, the “less” function object. You can alternatively use any function object returning a Boolean result.

The example:

- Instantiates a priority queue, queue, and stack;
- Pushes some characters onto the adaptor instances;
- Queries the adaptor instances; and
- Iteratively, outputs the front character, and pops the adaptor.



Quick Reference Guide: Container Adaptor Functions

Type	Function	Parameters	Description
-	C	() , (const Alloc&)	Constructor – default
-	C	(const C&, const Alloc&) (C&&, const Alloc&)	Constructor – copy Constructor – move
-	C	(const Container& [, const Alloc&]) (Container&& [, const Alloc&])	Constructor – copy-init [stack, queue] Constructor – move-init [stack, queue]
bool	empty	()	Container is empty
size_type	size	()	Container size
reference const_reference	front/back	()	[queue] Element at front/back
reference const_reference	top	()	[stack] Element at top [stack, priority_queue] Element at top
void	emplace ^{C++11}	(Args&&...)	Push newly created element
void	push	(const T&), (T&&)	Push copy/move t
void	pop	()	Pop element
void	swap ^{C++11}	(C&), (C&, C&)	Swap adaptors – member/nonmember

Push/pop act upon different ends depending upon adaptor type.

285 © Cadence Design Systems, Inc. All rights reserved.



Adaptor classes provide:

- Constructors to construct and initialize adaptor objects, including other priority queue constructors not displayed here.
- The function “**empty**” to return the truth of whether the adaptor is empty.
- The function “**size**” to return the adaptor’s current size.
- For queue adaptors, the functions “**front**” and “**back**” to return the element at the adaptor’s front and back.
- For stack and priority queue adaptors, the function “**top**” to return the element at the adaptor’s top.
- The function “**emplace**” to emplace at the adaptor’s push position a new element created using the provided arguments.
- The function “**push**” to push an element, and the function “**pop**” to pop an element.
 - The push and pop functions act upon different adaptor ends, depending upon the adaptor type.
- The member and non-member functions “**swap**” to swap two identically-specialized adaptors.

Example: Declaring and Using Container Adaptors Code

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

template <class T> ostream& operator<<(ostream& os, stack<T>& cont_) {
    stack<T> cont = cont_;
    while (!cont.empty())
        { os << cont.top(); 
        cont.pop(); }
    return os;
}

int main () {
    stack<char> cont;
    for (auto &c : string("kcats"))
        {cont.push(c); }
    cout << cont << endl; // stack
    return 0;
}
```

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

template <class T> ostream& operator<<(ostream& os, queue<T>& cont_) {
    queue<T> cont = cont_;
    while (!cont.empty())
        { os << cont.front();
        cont.pop(); }
    return os;
}

int main () {
    queue<char> cont;
    for (auto &c : string("queue"))
        {cont.push(c); }
    cout << cont << endl; // queue
    return 0;
}
```

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

template <class T1, class T2, class T3>
ostream& operator<<(ostream& os,
priority_queue<T1,T2,T3>& cont_) {
    priority_queue<T1,T2,T3> cont=cont_;
    while (!cont.empty())
        { os << cont.top(); cont.pop(); }
    return os;
}

int main () {
    priority_queue<char> cont;
    for (auto &c : string("prior"))
        {cont.push(c); }
    cout << cont << endl; // rrpoi
    return 0;
}
```

286 © Cadence Design Systems, Inc. All rights reserved.



Each of these examples:

- Defines an output stream insertion operator, that makes a copy of the adaptor argument, and iteratively strips elements off the front or top of that adaptor, and outputs them to the standard output.
- In the main routine, declares an adapter, pushes a string into the adaptor, and passes the adaptor to the output stream insertion operator.

What Is an Associative Container?



An **Associative Container** is a standard container that stores elements in association with a key and offers features in addition to those common to all standard containers.

Use associative containers to model associatively-accessed containers.

- Sparse arrays, symbol tables, dictionaries, collections of any type for which a comparison expression can compare the elements.

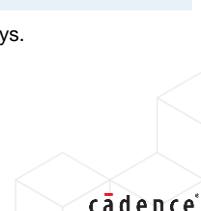
Choose the associative container optimized for your usage model.

- Finding by key is $O(\log n)$ for ordered maps/sets.
- Finding by key averages $O(1)$ for unordered maps/sets and can degrade to $O(n)$ worst-case.

<key, value>	Unique Keys	Duplicate Keys
ordered	map	multimap
unordered ^{C++11}	unordered_map	unordered_multimap

<key>	Unique Keys	Duplicate Keys
ordered	set	multiset
unordered ^{C++11}	unordered_set	unordered_multiset

The default compare object assumes numerical keys.



A Standard Container is an object that stores other objects, and controls allocation and deallocation of those objects, while conforming to a well-defined interface.

An **Associative Container** is a standard container that stores elements in association with a key and offers features in addition to those common to all standard containers.

Use associative containers to model associatively-accessed data, for example:

- Sparse arrays, symbol tables, dictionaries, and collections of any type for which a comparison expression can compare the elements.

Choose the associative container optimized for your usage model.

- First determine whether you need duplicate keys and whether you will store data associated with those keys.
- Then determine whether you should use an ordered container or an unordered container.
 - For ordered containers, operations acting on a key have logarithmic complexity.
 - For unordered containers, operations acting on a key can have constant complexity, and can degrade to linear complexity under worst-case conditions.

To choose between associative containers, you should profile your intended usage with respect to key uniqueness and how well the hash algorithm preserves that uniqueness.



Quick Reference Guide: Associative Container Common Functions [1/2]

Type	Function	Parameters	Description
void	<code>clear</code>	<code>()</code>	Clear the container
<code>pair<iterator, bool></code> <code>iterator</code> <code>iterator</code>	<code>emplace^{C++11}</code> <code>emplace^{C++11}</code> <code>emplace_hint^{C++11}</code>	<code>(Args&&...)</code> <code>(Args&&...)</code> <code>(const_iterator, Args&&...)</code>	Emplace new element [unique keys] Emplace new element [duplicate keys] Emplace new element near position
<code>size_type</code> <code>iterator</code> <code>iterator</code>	<code>erase</code> <code>erase</code> <code>erase</code>	<code>(const key_type&)</code> <code>([const_]iterator)</code> <code>(const_iterator, const_iterator)</code>	Erase element(s) having key Erase element at position Erase element(s) in range
<code>node_type</code> <code>node_type</code>	<code>extract^{C++17}</code> <code>extract^{C++17}</code>	<code>(const key_type&)</code> <code>(const_iterator)</code>	Extract first node having key Extract node at position
<code>pair<iterator,bool></code> <code>iterator</code> <code>iterator</code> <code>iterator</code> <code>pair<iterator, bool></code> <code>iterator</code> <code>iterator</code> <code>void</code> <code>void</code>	<code>insert</code> <code>insert</code> <code>insert</code> <code>insert</code> <code>insert</code> <code>insert</code> <code>insert</code> <code>insert</code> <code>insert</code>	<code>(const value_type&), (value_type&&)</code> <code>(const value_type&), (value_type&&)</code> <code>(const_iterator, const value_type&)</code> <code>(const_iterator, value_type&&)</code> <code>(</code> <code>(</code> <code>(const_iterator, node_type&&)</code> <code>(InputIterator, InputIterator)</code> <code>(initializer_list<value_type>)</code>	Insert (unique keys) Insert at range end (duplicate keys) Insert before position copied value Insert before position moved value Insert moved node (unique keys) Insert moved node (duplicate keys) Insert before position moved node Insert elements from range Insert elements from initializer list

These functions are in addition to the functions common to all standard containers.

288 © Cadence Design Systems, Inc. All rights reserved.



All standard containers support simple constructors, some query functions, simple relational operators, and iterators.

In addition to the functions common to all standard containers, associative containers provide additional functions.

- The function “**clear**” to clear the container.
- The function “**emplace**” to emplace an element newly created using the provided arguments.
 - For a container supporting duplicate keys, the emplace function places the new element at the end of any existing range having that key.
 - The function “**emplace hint**” (**emplace_hint**) places the element as close before the position as possible.
- The function “**erase**” to erase either one element or a range of elements.
- The function “**extract**” to extract a node.
- The function “**insert**” to insert either an element or a node, or elements from a range or initializer list.
 - For a container supporting duplicate keys, the insert function places the new node at the end of any existing range having that key.



Quick Reference Guide: Associative Container Common Functions [2/2]

Type	Function	Parameters	Description
void	<code>merge</code> ^{C++17}	(C1<Key, T, C2, Allocator&) (C1<Key, T, C2, Allocator&&) (C1<Key, C2, Allocator&) (C1<Key, C2, Allocator&&) (C1<Key, T, H2, P2, Allocator&) (C1<Key, T, H2, P2, Allocator&&) (C1<Key, H2, P2, Allocator&) (C1<Key, H2, P2, Allocator&&)	Merge ordered map/multimap Merge ordered set/multiset Merge unordered map/multimap Merge unordered set/multiset
bool	<code>contains</code> ^{C++20}	(const key_type&)	If contains element having key
<code>size_type</code>	<code>count</code>	(const key_type&)	Count of elements having key
[const_]iterator	<code>lower_bound</code> <code>upper_bound</code>	(const key_type&)	First element having key >= k First element having key > k
<code>pair<[const_]iterator, [const_]iterator></code>	<code>equal_range</code>	(const key_type&)	Lower and upper bounds
[const_]iterator	<code>find</code>	(const key_type&)	Find first element having key

These functions are in addition to the functions common to all standard containers.

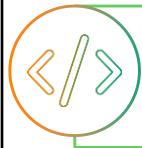
289 © Cadence Design Systems, Inc. All rights reserved.



In addition to the functions common to all standard containers, associative containers provide additional functions.

- The function “**merge**” to merge elements from another similarly-typed container.
- The function “**contains**” to return the truth of whether the specified key is contained.
- The function “**count**” to return the count of the specified key.
- The function “**lower_bound**” (**lower_bound**) to return the lower bound of the specified key, “**upper_bound**” (**upper_bound**) to return the upper bound of the specified key, and “**equal_range**” (**equal_range**) to return the lower and upper bounds of the specified key.
- The function “**find**” to return the location of the specified key.

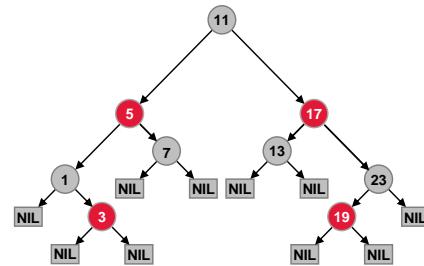
Declaring and Using Ordered Maps



```
std::map<key_type, mapped_type, key_compare_opt, allocator_opt> id
std::multimap<key_type, mapped_type, key_compare_opt, allocator_opt> id
```

Ordered maps support all associative container operations.

- Constructors
- Iteration: forward/backward
- Container assignment
- Emplacement
- Insert/erase by iterator or key
- Miscellaneous common operations
- Map-specific operations



```
using container_t = map<unsigned,char>;
const unsigned size = 32;
unsigned addr; char data;
container_t cont;
for (unsigned idx=0; idx < size; ++idx) {
    addr = ((unsigned)rand())%size;
    data = (char)(97+rand()%26);
    write(addr,data);
    cont[addr] = data;
}
cout << cont.size() << endl; // ?
cout << cont << endl; // user-defined
for (auto elem : cont) {
    addr = elem.first;
    read(addr,data);
    assert(data == elem.second);
}
```



A **map** is typically but not necessarily implemented as a red-black tree, that automatically resizes[†] itself as a result of insertions and erasure.

A red-black tree is one kind of self-balancing binary search tree. Hence, all map search operations are of logarithmic complexity.

Maps support all associative container common operations.

The example:

- Declares a map containing pairs of unsigned key and associated character data;
- In a loop, writes random data to random memory addresses, while entering the pairs of address and data into the container;
- Outputs the container, using a user-defined output stream insertion operator; and
- In a loop, from the container beginning to its end, reads pairs of address and data from the container, while confirming that the memory contains that data at that address;

If you store keys with associated values, you might choose a map.

- If your keys are not unique, choose instead a multimap.
- If key order is not important, for faster key search, choose an unordered map or unordered multimap.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.



Quick Reference Guide: Ordered Map Functions

Type	Function	Parameters	Description
[const] mapped_type&	at ^{C++11}	(const key_type&)	Value at key
mapped_type&	operator[]	(const key_type&)	Subscript operator
pair<iterator, bool> iterator	try_emplace ^{C++17} try_emplace ^{C++17}	([const] key_type&, Args&&...) (const_iterator, [const] key_type&, Args&&...)	Emplace if key absent
pair<iterator, bool> iterator	insert_or_assign ^{C++17} insert_or_assign ^{C++17}	(const key_type&, M&&) (const_iterator, const key_type&, M&&)	Insert if key absent else assign value

Only ordered and unordered maps (not multimaps) support these functions.

291 © Cadence Design Systems, Inc. All rights reserved.



Maps have a unique relationship between keys and values, so can support value-oriented operations.

- The function “**at**” to return a constant reference to the value associated with the specified key.
- The subscript operator to return a non-constant reference to the value associated with the specified key.
- The function “try emplace” (**try_emplace**), if the specified key is absent, constructs an element according to the arguments, and emplaces it.
 - The first signature, returns an iterator to the existing or emplaced element, and a Boolean indicating whether emplacement occurred.
 - The second signature, returns only an iterator to the existing or emplaced element.
- The function “insert or assign” (**insert_or_assign**), if the specified key is absent, inserts the referenced element, otherwise re-assigns the value associated with the key.

Comparing Ordered Maps and Ordered Multimaps



For a **Map**, keys must be unique. An attempt to insert a duplicate key fails.

You can retrieve the mapped type value using `at(key)` or `operator[key]`.

`insert/emplace`
return pair<iterator,bool>

```
using container_t = map<unsigned,char>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t iter1, iter2;
pair<citer_t,bool> pair1;
pair1 = cont.insert(pair<unsigned,char>(5,'e'));
cout.setf(ios_base::boolalpha);
pair1 = cont.emplace(3,'c');
cout << "Success is " << pair1.second << endl; // true
pair1 = cont.emplace(3,'d'); // Duplicate key
cout << "Success is " << pair1.second << endl; // false
iter1 = cont.emplace_hint(cont.end(),7,'g');
cout << "Success is " << (iter1 != cont.end()) << endl; // true
iter2 = cont.emplace_hint(mp, cont(),7,'h'); // Duplicate key
cout << "Success is " << (iter2 != iter1) << endl; // false
cout << "size is " << cont.size() << endl; // 3
cout << " contents is " << cont << endl; // user-defined - 357
cout << "key 7 count is " << cont.count(7) << endl; // 1
iter1 = cont.lower_bound(7); iter2 = cont.upper_bound(7);
cout << "Mapped item at key 7 lower bound exactly is "
<< ( iter1->second << endl; // g
cout << "Mapped item at key 7 upper bound minus 1 is "
<< (--iter2)->second << endl; // g
```



For a **Multimap**, keys can be duplicated. An attempt to insert a duplicate key does not fail.

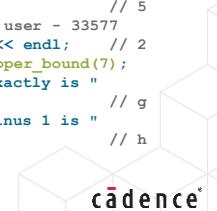
You cannot retrieve the mapped type value using its key.

`insert/emplace`
return iterator

```
using container_t = multimap<unsigned,char>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t iter1, iter2;

iter1 = cont.insert(pair<unsigned,char>(5,'e'));
cout.setf(ios_base::boolalpha);
iter2 = cont.emplace(3,'c');
cout << "Success is " << (iter2 != cont.end()) << endl; // true
iter2 = cont.emplace(3,'d');
cout << "Success is " << (iter3 != iter2) << endl; // true
iter1 = cont.emplace_hint(cont.end(),7,'g');
cout << "Success is " << (iter1 != cont.end()) << endl; // true
iter2 = cont.emplace_hint(cont.end(),7,'h');
cout << "Success is " << (iter2 != iter1) << endl; // true
cout << " size is " << cont.size() << endl; // 5
cout << " contents is " << cont << endl; // user - 33577
cout << "key 7 count is " << cont.count(7) << endl; // 2
iter1 = cont.lower_bound(7); iter2 = cont.upper_bound(7);
cout << "Mapped item at key 7 lower bound exactly is "
<< ( iter1->second << endl; // g
cout << "Mapped item at key 7 upper bound minus 1 is "
<< (--iter2)->second << endl; // h
```

292 © Cadence Design Systems, Inc. All rights reserved.



A **map** is an associative container that stores data in association with keys.

For a **map**, keys must be unique, so a map supports functions to access data by indexing with the key. An attempt to insert a duplicate key always fails, so the insert function returns a pair, whose first element is an iterator to the key location, and whose second element is a Boolean indicating whether the insertion succeeded.

For a **multimap**, keys may be duplicated, so a multimap cannot support functions to access data by indexing with the key. An attempt to insert a duplicate key never fails, so the insert function returns only an iterator to the key location.

The left example:

- Declares a map whose key type is unsigned and whose mapped type is character.
- Declares a pair whose first element type is iterator and whose second element type is Boolean.
- Inserts an element having key 5.
- Emplaces an element having key 3, and confirms success.
- Emplaces another element having key 3, and confirms failure.
- Emplaces with a hint an element having key 7. The hint improves performance, as the algorithm starts looking for an existing key at the provided location, rather than at the container beginning. Here, the previous element's key is 5, and the next element does not exist, so the search stops almost immediately. This function returns only an iterator, so to confirm success, we confirm that the returned iterator is not the container end, which is one element after the final element.
- Emplaces with a hint another element having key 7. Here, to confirm failure, we confirm that the returned iterator points to the existing element having key 7.
- Obtains iterators to the lower and upper bounds of the elements having key 7. Only one element can exist having key 7.

The right example:

- Declares a multimap whose key type is unsigned and whose mapped type is character.
- Inserts an element having key 5.
- Emplaces an element having key 3, and confirms success, by confirming that the returned element iterator is not at the container end, which is one element after the final element.
- Emplaces another element having key 3, and confirms success, by confirming that the returned element iterator does not point to the previous element having key 3.
- Emplaces with a hint an element having key 7, and confirms success.
- Emplaces with a hint another element having key 7, and confirms success.
- Obtains iterators to the lower and upper bounds of the elements having key 7. Here, two elements exist having key 7.

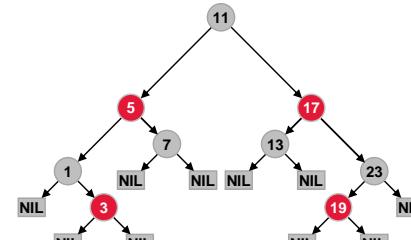
Declaring and Using Ordered Sets



```
std::set      <key_type, key_compare_opt, allocator_opt> identifier
std::multiset <key_type, key_compare_opt, allocator_opt> identifier
```

Ordered sets support all associative container operations.

- Constructors
- Iteration: forward/backward
- Container assignment
- Emplacement
- Insert/erase by iterator or key
- Miscellaneous common operations



```
struct mem {      // addr, data, constructors
    struct cmp { // mem operator<()
        using container_t = set<mem,cmp>;
        using citer_t = container_t::const iterator;
        const unsigned size = 32; mem memi[size];
        unsigned addr; char data;
        container_t cont; citer_t citer;
        for (unsigned idx=0; idx < size; ++idx) {
            addr = ((unsigned)rand())%size;
            data = (char)(97+rand()%26);
            write(memi,addr,data);
            if (cont.contains(mem(addr,data)))
                cont.erase(mem(addr,data));
            cont.insert(mem(addr,data));
        }
        cout << cont.size() << endl; // ?
        cout << cont << endl; // user-defined
        for (auto elem : cont) {
            addr = elem.addr;
            read(memi,addr,data);
            assert(data == elem.data);
        }
    }
}
```

293 © Cadence Design Systems, Inc. All rights reserved.

cadence

A **set** is typically but not necessarily implemented as a red-black tree, that automatically resizes[†] itself as a result of insertions and erasure.

A red-black tree is one kind of self-balancing binary search tree. Hence, all set search operations are of logarithmic complexity.

Sets support all associative container common operations.

The example:

- Declares a set containing memory structure objects and utilizing the comparison class;
- In a loop, writes random data to random memory addresses, while entering memory structure objects into the container;
 - The contained memory structure objects must be unique as defined by the comparison class, so first we find and erase any existing comparable object. Here, the comparison function compares only the address field of the memory element structure.
- Outputs the container, using a user-defined output stream insertion operator; and
- In a loop, from the container beginning to its end, reads memory structure objects from the container, while confirming for each object that the memory contains that data at that address;

If you store only keys and not associated with values, you might choose a set.

- If your keys are not unique, choose instead a multiset.
- If key order is not important, for faster key search, choose an unordered set or unordered multiset.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.

Comparing Ordered Sets and Ordered Multisets



For a **Set**, keys must be unique.

An attempt to insert a duplicate key fails.

```
using container_t = set<unsigned,char>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t citer1, citer2;
pair<citer_t,bool> pair1;
cout.setf(ios_base::boolalpha);
pair1 = cont.insert(5);
cout << "Success is " << pair1.second << endl; // true
pair1 = cont.emplace(3);
cout << "Success is " << pair1.second << endl; // true
pair1 = cont.emplace(3);
cout << "Success is " << pair1.second << endl; // false
citer1 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer1 != st.cend()) << endl; // true
citer2 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer2 != citer1) << endl; // false
cout << "size is " << cont.size() << endl; // 3
cout << "contents is " << cont << endl; // user-defined - 357
cout << "key 7 count is " << cont.count(7) << endl; // 1
cout << "lower_bound(7)==upper_bound(5) is " // true
<< (cont.lower_bound(7)==cont.upper_bound(5)) << endl;
cout << "upper_bound(7)==end() is " // true
<< (cont.upper_bound(7)==cont.end()) << endl;
```

insert/emplace
return pair<iter,bool>



For a **Multiset**, keys can be duplicated.

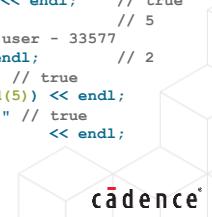
An attempt to insert a duplicate key does not fail.

```
using container_t = multiset<unsigned,char>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t citer1, citer2;

cout.setf(ios_base::boolalpha);
citer1 = cont.insert(5);
cout << "Success is " << (citer1 != cont.end()) << endl; // true
citer1 = cont.emplace(3);
cout << "Success is " << (citer1 != cont.end()) << endl; // true
citer2 = cont.emplace(3);
cout << "Success is " << (citer2 != citer1) << endl; // true
citer1 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer1 != mst.cend()) << endl; // true
citer2 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer2 != citer1) << endl; // true
cout << "size is " << cont.size() << endl; // 5
cout << "container is " << cont << endl; // user - 33577
cout << "count(7) is " << cont.count(7) << endl; // 2
cout << "lower_bound(7)==upper_bound(5) is " // true
<< (cont.lower_bound(7)==cont.upper_bound(5)) << endl;
cout << "upper_bound(7)==end() is " // true
<< (cont.upper_bound(7)==cont.end()) << endl;
```

insert/emplace
return iterator

294 © Cadence Design Systems, Inc. All rights reserved.



A **set** is an associative container that stores only keys.

For a **set**, keys must be unique. An attempt to insert a duplicate key always fails, so the insert function returns a pair, whose first element is an iterator to the key location, and whose second element is a Boolean indicating whether the insertion succeeded.

For a **multiset**, keys may be duplicated. An attempt to insert a duplicate key never fails, so the insert function returns only an iterator to the key location.

The left example:

- Declares a set whose key type is unsigned.
- Declares a pair whose first element type is iterator and whose second element type is Boolean.
- Inserts a key 5, and confirms success.
- Emplaces a key 3, and confirms success.
- Emplaces another key 3, and confirms failure.
- Emplaces with a hint a key 7. The hint improves performance, as the algorithm starts looking for an existing key at the provided location, rather than at the container beginning. Here, the previous element's key is 5, and the next element does not exist, so the search stops almost immediately. This function returns only an iterator, so to confirm success, we confirm that the returned iterator is not the container end, which is one element after the final element.
- Emplaces with a hint another key 7. Here, to confirm failure, we confirm that the returned iterator points to the existing key 7.
- Obtains iterators to the lower and upper bounds of the elements having key 7. Only one key 7 can exist.

The right example:

- Declares a multiset whose key type is unsigned.
- Inserts a key 5, and confirms success.
- Emplaces a key 3, and confirms success, by confirming that the returned element iterator is not at the container end, which is one element after the final element.
- Emplaces another key 3, and confirms success, by confirming that the returned element iterator does not point to the previous key 3.
- Emplaces with a hint a key 7, and confirms success.
- Emplaces with a hint another key 7, and confirms success.
- Obtains iterators to the lower and upper bounds of the elements having key 7. Here, two keys 7 exist.

What Is an Unordered Associative Container?



An Associative Container is a standard container that stores elements in association with a key and offers features in addition to those common to all standard containers.

Pre - C++11 Associative Containers are all ordered and all implemented using balanced binary trees, in which finding a key has logarithmic complexity ($O(\log(n))$). C++11 duplicated these containers as **Unordered Associative Containers** implemented using hash tables, in which finding a key has constant complexity ($O(1)$).

Use associative containers to model associatively-accessed containers.

- Sparse arrays, symbol tables, dictionaries, collections of any type for which a comparison expression can rank the elements.

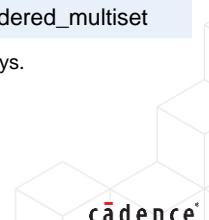
Choose the associative container optimized for your usage model.

- Finding by key is $O(\log n)$ for ordered maps/sets.
- Finding by key averages $O(1)$ for unordered maps/sets and can degrade to $O(n)$ worst-case.

<code><key, value></code>	Unique Keys	Duplicate Keys
<code>ordered</code>	map	multimap
<code>unordered C++11</code>	unordered_map	unordered_multimap

<code><key></code>	Unique Keys	Duplicate Keys
<code>ordered</code>	set	multiset
<code>unordered C++11</code>	unordered_set	unordered_multiset

The default compare object assumes numerical keys.



A Standard Container is an object that stores other objects, and controls allocation and deallocation of those objects, while conforming to a well-defined interface.

An Associative Container is a standard container that stores elements in association with a key and offers features in addition to those common to all standard containers.

Pre - C++11 Associative Containers are all ordered and all implemented using balanced binary trees, in which finding a key has logarithmic complexity ($O(\log(n))$).

C++11 duplicated these containers as **Unordered Associative Containers** implemented using hash tables, in which finding a key has constant complexity ($O(1)$).

Use associative containers to model associatively-accessed data, for example:

- Sparse arrays, symbol tables, dictionaries, and collections of any type for which a comparison expression can rank the elements.

Choose the associative container optimized for your usage model.

- First determine whether you need duplicate keys and whether you will store data associated with those keys.
- Then determine whether you should use an ordered container or an unordered container.
 - For ordered containers, operations acting upon an iterator have constant complexity, and operations acting upon a key have logarithmic complexity.
 - For unordered containers, operations acting upon an iterator have constant complexity, and operations acting upon a key can have constant complexity, and can degrade to linear complexity under worst-case conditions. The degradation is proportional to the load factor, that is, what fraction of different keys hash to the same bucket. Finding a bucket has constant complexity, and finding a key within a bucket has linear complexity.

To choose between associative containers, you should profile your intended usage with respect to key uniqueness and how well the hash algorithm preserves that uniqueness.



Quick Reference Guide: Unordered Associative Container Common Functions

Type	Function	Parameters	Description
hasher	<code>hash_function</code>	<code>()</code>	Hash function
key_equal	<code>key_eq</code>	<code>()</code>	Key comparison function
local_iterator const_local_iterator const_local_iterator	<code>begin</code> <code>begin</code> <code>cbegin</code>	<code>(size_type)</code> <code>(size_type)</code> <code>(size_type)</code>	Iterator to bucket begin
local_iterator const_local_iterator const_local_iterator	<code>end</code> <code>end</code> <code>cend</code>	<code>(size_type)</code> <code>(size_type)</code> <code>(size_type)</code>	Iterator to bucket end
size_type	<code>bucket</code>	<code>(const key_type&)</code>	Bucket containing key
size_type	<code>bucket_count</code>	<code>()</code>	Count of buckets
size_type	<code>bucket_size</code>	<code>(size_type)</code>	Size of bucket
size_type	<code>max_bucket_count</code>	<code>()</code>	Maximum bucket count
float	<code>load_factor</code>	<code>()</code>	Load factor (mean elements/bucket)
float void	<code>max_load_factor</code> <code>max_load_factor</code>	<code>()</code> <code>(float)</code>	Get maximum load factor Set maximum load factor
void	<code>rehash</code>	<code>(size_type)</code>	Rehash to at least number buckets
void	<code>reserve</code>	<code>(size_type)</code>	Rehash to at least number elements

These functions are in addition to the functions common to all associative containers.

296 © Cadence Design Systems, Inc. All rights reserved.



In addition to the functions common to all associative containers, unordered associative containers provide additional functions, related to the hash algorithm, and providing a bucket interface.

- The function “hash function” (**hash_function**) returns the function that hashes keys. This is a template parameter having a default value that supports the arithmetic, enumeration, and pointer types.
- The function “key equal” (**key_eq**), returns the function that compares keys. This is a template parameter having a default value that supports the arithmetic, enumeration, and pointer types.
- The function “**begin**”, takes a bucket number argument, and returns an iterator to the first element in that bucket.
- The function “**end**”, takes a bucket number argument, and returns an iterator to after the last element in that bucket.
- The function “**bucket**”, takes an element reference argument, and returns the bucket number containing that element.
- The function “bucket count” (**bucket_count**) returns the count of contained buckets. The initial bucket count is a constructor argument having an implementation-defined default value.
- The function “bucket size” (**bucket_size**), takes a bucket number argument, and returns the element count within that bucket.
- The function “maximum bucket count” (**max_bucket_count**), returns the maximum supported bucket count.
- The function “load factor” (**load_factor**) returns the load factor, that is, the average number of elements per bucket.
- The function “maximum load factor” (**max_load_factor**), gets and sets the maximum load factor. The maximum load factor is by default 1. The container automatically rehashes upon exceeding the maximum load factor.
- The function “**rehash**” rehashes the container to at least the specified number of buckets.
- The function “**reserve**” rehashes the container to at least the specified number of elements.

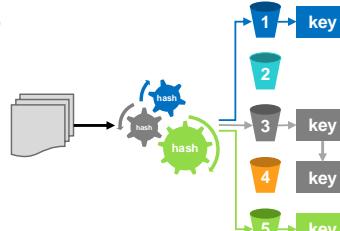
Declaring and Using Unordered Maps



```
std::unordered_map<key_type, mapped_type, Hash_opt, Pred_opt, Allocator_opt> id
std::unordered_multimap<key_type, mapped_type, Hash_opt, Pred_opt, Allocator_opt> id
```

Unordered maps support most associative container operations.

- Constructors
- Iteration: forward/backward
- Container assignment
- Emplacement
- Insert/erase by iterator or key
- Miscellaneous common operations
- Map-specific operations



```
using container_t = std::unordered_map<unsigned, char>;
using citer_t = container_t::const_iterator;
const unsigned size = 32;
unsigned addr; char data;
container_t cont; citer_t citer;
for (unsigned idx=0; idx < size; ++idx) {
    addr = ((unsigned)rand())%size;
    data = (char)(97+rand()%26);
    write(addr,data);
    cont[addr] = data;
}
cout << cont.size() << endl; // ??
cout << cont << endl; // user-defined
for (auto elem : cont) {
    addr = elem.first;
    read(addr,data);
    assert(data == elem.second);
}
```



297 © Cadence Design Systems, Inc. All rights reserved.

A unordered map (**unordered_map**), is typically but not necessarily implemented with a hash table, that automatically resizes[†] itself as a result of insertions and erasure and rehash. Hence, all unordered map search operations are of constant complexity.

Unordered maps support most associative container common operations. Unordered containers have no reverse iterators.

The example:

- Declares a map containing pairs of unsigned key and associated character data;
- In a loop, writes random data to random memory addresses, while entering the pairs of address and data into the container;
- Outputs the container, using a user-defined output stream insertion operator; and
- In a loop, from the container beginning to its end, reads pairs of address and data from the container, while confirming that the memory contains that data at that address;

If you store keys with associated values, you might choose a map.

- If your keys are not unique, choose instead a multimap.
- If key order is not important, for faster key search, choose an unordered map or unordered multimap.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.



Quick Reference Guide: Unordered Map Functions

Type	Function	Parameters	Description
[const] mapped_type&	<code>at^{C++11}</code>	(const key_type&)	Value at key
mapped_type&	<code>operator[]</code>	(const key_type&)	Subscript operator
pair<iterator, bool> iterator	<code>try_emplace^{C++17}</code> <code>try_emplace^{C++17}</code>	([const] key_type&, Args&&...) (const_iterator, [const] key_type&, Args&&...)	Emplace if key absent
pair<iterator, bool> iterator	<code>insert_or_assign^{C++17}</code> <code>insert_or_assign^{C++17}</code>	(const key_type&, M&&) (const_iterator, const key_type&, M&&)	Insert if key absent else assign value

Only ordered and unordered maps (not multimaps) support these functions.

298 © Cadence Design Systems, Inc. All rights reserved.



Maps have a unique relationship between keys and values, so can support value-oriented operations.

- The function “**at**” to return a constant reference to the value associated with the specified key.
- The subscript operator to return a non-constant reference to the value associated with the specified key.
- The function “try emplace” (**try_emplace**), if the specified key is absent, constructs an element according to the arguments, and emplaces it.
 - The first signature, returns an iterator to the existing or emplaced element, and a Boolean indicating that emplacement occurred.
 - The second signature, returns only an iterator to the existing or emplaced element.
- The function “insert or assign” (**insert_or_assign**), if the specified key is absent, inserts the referenced element, otherwise re-assigns the value associated with the key.

Comparing Unordered Maps and Unordered Multimaps



For a Map, keys must be unique. An attempt to insert a duplicate key fails.

You can retrieve a mapped type through `at (key)` or operator `[key]`.

`insert/emplace return pair<iter,bool>`



For a Multimap, keys can be duplicated. An attempt to insert a duplicate key does not fail.

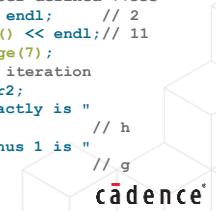
You cannot retrieve a mapped type through its potentially non-unique key.

`insert/emplace return iterator`

```
using container_t = unordered_map<unsigned,char>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t citer1, citer2;
pair<citer_t,bool> pair1;
pair1 = cont.insert(pair<unsigned,char>(5,'e'));
cout.setf(ios_base::boolalpha);
pair1 = cont.emplace(3,'c'); // Duplicate key
cout << "Success is " << pair1.second << endl; // true
pair1 = cont.emplace(3,'d'); // Duplicate key
cout << "Success is " << pair1.second << endl; // false
citer1 = cont.emplace_hint(cont.cend(),7,'g');
cout << "Success is "<<(citer1 != cont.cend())<<endl; // true
citer2 = cont.emplace_hint(cont.cend(),7,'h');// Duplicate key
cout << "Success is " << (citer2 != citer1) << endl; // false
cout << "size is " << cont.size() << endl; // 3
cout << "contents is " << cont << endl; // user-defined - 735
cout << "key 7 count is " << cont.count(7) << endl; // 1
cout << "bucket cnt is " << cont.bucket_count() << endl; // 5
pair<citer_t,citer_t> piter = cont.equal_range(7);
citer2 = citer1 = piter.first; // no reverse iteration
while (++piter.first != piter.second) ++citer2;
cout << "Mapped item at key 7 lower bound exactly is "
    << citer1->second << endl; // g
cout << "Mapped item at key 7 upper bound minus 1 is "
    << citer2->second << endl; // g
```

299 © Cadence Design Systems, Inc. All rights reserved.

```
using container_t = unordered_multimap<unsigned,char>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t citer1, citer2;
pair<citer_t,bool> pair1;
citer1 = cont.insert(pair<unsigned,char>(5,'e'));
cout.setf(ios_base::boolalpha);
citer1 = cont.emplace(3,'c');
cout << "Success is "<<(citer1 != cont.cend())<<endl; // true
citer2 = cont.emplace(3,'d'); // Duplicate key
cout << "Success is "<<(citer2 != citer1) << endl; // true
citer1 = cont.emplace_hint(cont.cend(),7,'g');
cout << "Success is "<<(citer1 != cont.cend())<<endl; // true
citer2 = cont.emplace_hint(cont.cend(),7,'h');// Duplicate key
cout << "Success is " << (citer2 != citer1) << endl; // true
cout << "size is " << cont.size() << endl; // 5
cout << "contents is " << cont << endl; // user-defined 77533
cout << "key 7 count is "<< cont.count(7) << endl; // 2
cout << "bucket cnt is "<< cont.bucket_count() << endl; // 11
pair<citer_t,citer_t> piter = cont.equal_range(7);
citer2 = citer1 = piter.first; // no reverse iteration
while (++piter.first != piter.second) ++citer2;
cout << "Mapped item at key 7 lower bound exactly is "
    << citer1->second << endl; // h
cout << "Mapped item at key 7 upper bound minus 1 is "
    << citer2->second << endl; // g
```



A **map** is an associative container that stores data in association with keys.

For a **map**, keys must be unique, so a map supports functions to access data by indexing with the key. An attempt to insert a duplicate key always fails, so the insert function returns a pair, whose first element is an iterator to the key location, and whose second element is a Boolean indicating whether the insertion succeeded.

For a **multimap**, keys may be duplicated, so a multimap cannot support functions to access data by indexing with the key. An attempt to insert a duplicate key never fails, so the insert function returns only an iterator to the key location.

The left example:

- Declares an unordered map whose key type is unsigned and whose mapped type is character.
- Declares a pair whose first element type is iterator and whose second element type is Boolean.
- Inserts an element having key 5.
- Emplaces an element having key 3, and confirms success.
- Emplaces another element having key 3, and confirms failure.
- Emplaces with a hint an element having key 7. The hint improves performance, as the algorithm starts looking for an existing key at the provided location, rather than at the container beginning. Here, the previous element's key is 5, and the next element does not exist, so the search stops almost immediately. This function returns only an iterator, so to confirm success, we confirm that the returned iterator is not the container end, which is one element after the final element.
- Emplaces with a hint another element having key 7. Here, to confirm failure, we confirm that the returned iterator points to the existing element having key 7.
- Obtains iterators to the lower and upper bounds of the elements having key 7. Only one element can exist having key 7.

The right example:

- Declares an unordered multimap whose key type is unsigned and whose mapped type is character.
- Inserts an element having key 5.
- Emplaces an element having key 3, and confirms success, by confirming that the returned element iterator is not at the container end, which is one element after the final element.
- Emplaces another element having key 3, and confirms success, by confirming that the returned element iterator does not point to the previous element having key 3.
- Emplaces with a hint an element having key 7, and confirms success.
- Emplaces with a hint another element having key 7, and confirms success.
- Obtains iterators to the lower and upper bounds of the elements having key 7. Here, two elements exist having key 7.

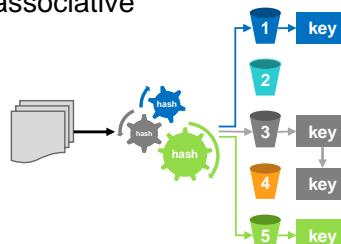
Declaring and Using Unordered Sets



```
std::unordered_set<key_type, Hash_opt, Pred_opt, Allocator_opt> identifier
std::unordered_multiset<key_type, Hash_opt, Pred_opt, Allocator_opt> identifier
```

Unordered sets support most associative container operations.

- Constructors
- Iteration: forward/backward
- Container assignment
- Emplacement
- Insert/erase by iterator or key
- Miscellaneous common operations



```
struct mem { // addr, data, constructors
    struct hsh { // conversion to unsigned op
        struct prd { // conversion to bool operator
            using container_t =
                unorderd_set<mem,hsh,prd>;
            using citer_t = container_t::const iterator;
            const unsigned size = 32; mem memi[size];
            unsigned addr; char data;
            container_t cont; citer_t citer;
            for (unsigned idx=0; idx < size; ++idx) {
                addr = ((unsigned)rand())%size;
                data = (char)(97+rand())%26);
                write(mem(addr,data));
                if (cont.contains(mem(addr,data)))
                    cont.erase(mem(addr,data));
                cont.insert(mem(addr,data));
            }
            cout << cont.size() << endl; // ??
            cout << cont << endl; // user-defined
            for (auto elem : cont) {
                addr = elem.addr;
                read(memi,addr,data);
                assert(data == elem.data);
            }
        }
    };
}
```

300 © Cadence Design Systems, Inc. All rights reserved.



A unordered set (**unordered_set**) is typically but not necessarily implemented with a hash table, that automatically resizes[†] itself as a result of insertions and erasure and rehash.

Unordered sets support most associative container common operations. Unordered containers have no reverse iterators.

The example:

- Declares a set containing structures of unsigned key and associated character data;
- In a loop, writes random data to random memory addresses, while entering the structures of address and data into the container;
- Outputs the container, using a user-defined output stream insertion operator; and
- In a loop, from the container beginning to its end, reads structures of address and data from the container, while confirming that the memory contains that data at that address;

If you store keys without associated values, you might choose a set.

- If your keys are not unique, choose instead a multiset.
- If key order is not important, for faster key search, choose an unordered set or unordered multiset.

[†]With respect to resizing a container, be sure to not confuse size and capacity. Whether and when the implementation de-allocates capacity is not specified. Only for vectors and unordered associative containers can you reserve capacity, and only for vectors and double-ended queues can you shrink capacity to size.

Comparing Unordered Sets and Unordered Multisets



For a **Set**, keys must be unique. An attempt to insert a duplicate key fails.



For a **Multiset**, keys can be duplicated. An attempt to insert a duplicate key does not fail.

```
using container_t = unordered_set<unsigned>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t citer1, citer2;
pair<citer_t,bool> pair1;
cout.setf(ios_base::boolalpha);
cont.max_load_factor(1.0); // should be default 1 anyway
pair1 = cont.insert(5);
cout << "Success is " << pair1.second << endl; // true
pair1 = cont.emplace(3);
cout << "Success is " << pair1.second << endl; // true
pair1 = cont.emplace(3);
cout << "Success is " << pair1.second << endl; // false
citer1 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer1 != cont.cend()) << endl; // true
citer2 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer2 != citer1) << endl; // false
cout << "size is " << cont.size() << endl; // 3
cout << "contents is " << cont << endl; // user-defined - 735
cout << "key 7 count is " << cont.count(7) << endl; // 1
cout << "bucket cnt is " << cont.bucket_count() << endl; // 5
cout << "max load factor " << cont.max_load_factor() << endl;
```

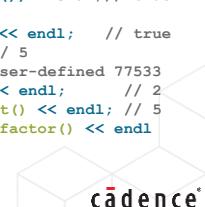
insert/emplace
return pair<iter,bool>

```
using container_t = unordered_multiset<unsigned>;
using citer_t = container_t::const_iterator;
container_t cont; citer_t citer1, citer2;

cout.setf(ios_base::boolalpha);
cont.max_load_factor(2.0); // double default loading
citer1 = cont.insert(5);
cout << "Success is " << (citer1 != cont.cend()) << endl; // true
citer1 = cont.emplace(3);
cout << "Success is " << (citer1 != cont.cend()) << endl; // true
citer2 = cont.emplace(3);
cout << "Success is " << (citer2 != citer1) << endl; // true
citer1 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer1 != cont.cend()) << endl; // true
citer2 = cont.emplace_hint(cont.cend(),7);
cout << "Success is " << (citer2 != citer1) << endl; // true
cout << "size is " << cont.size() << endl; // 5
cout << "contents is " << cont << endl; // user-defined 77533
cout << "key 7 count is " << cont.count(7) << endl; // 2
cout << "bucket cnt is " << cont.bucket_count() << endl; // 5
cout << "max load factor " << cont.max_load_factor() << endl;
```

insert/emplace
return iterator

301 © Cadence Design Systems, Inc. All rights reserved.



A **set** is an associative container that stores only keys.

For a **set**, keys must be unique. An attempt to insert a duplicate key always fails, so the insert function returns a pair, whose first element is an iterator to the key location, and whose second element is a Boolean indicating whether the insertion succeeded.

For a **multiset**, keys may be duplicated. An attempt to insert a duplicate key never fails, so the insert function returns only an iterator to the key location.

The left example:

- Declares an unordered set whose key type is unsigned.
- Declares a pair whose first element type is iterator and whose second element type is Boolean.
- Inserts a key 5, and confirms success.
- Emplaces a key 3, and confirms success.
- Emplaces another key 3, and confirms failure.
- Emplaces with a hint a key 7. The hint improves performance, as the algorithm starts looking for an existing key at the provided location, rather than at the container beginning. Here, the previous element's key is 5, and the next element does not exist, so the search stops almost immediately. This function returns only an iterator, so to confirm success, we confirm that the returned iterator is not the container end, which is one element after the final element.
- Emplaces with a hint another key 7. Here, to confirm failure, we confirm that the returned iterator points to the existing key 7.
- Obtains the count of keys 7. Only one key 7 can exist.
- Obtains the bucket count. The algorithm allocates buckets as needed to meet the maximum load factor.

The right example:

- Declares an unordered multiset whose key type is unsigned.
- Sets the maximum load factor to double the default maximum load factor.
- Inserts a key 5, and confirms success.
- Emplaces a key 3, and confirms success, by confirming that the returned element iterator is not at the container end, which is one element after the final element.
- Emplaces another key 3, and confirms success, by confirming that the returned element iterator does not point to the previous key 3.
- Emplaces with a hint a key 7, and confirms success.
- Emplaces with a hint another key 7, and confirms success.
- Obtains the count of keys 7. Here, two keys 7 exist.
- Obtains the bucket count. The algorithm allocates buckets as needed to meet the maximum load factor.

Quiz: Standard Containers



Suggest and substantiate design or verification purposes for various containers.

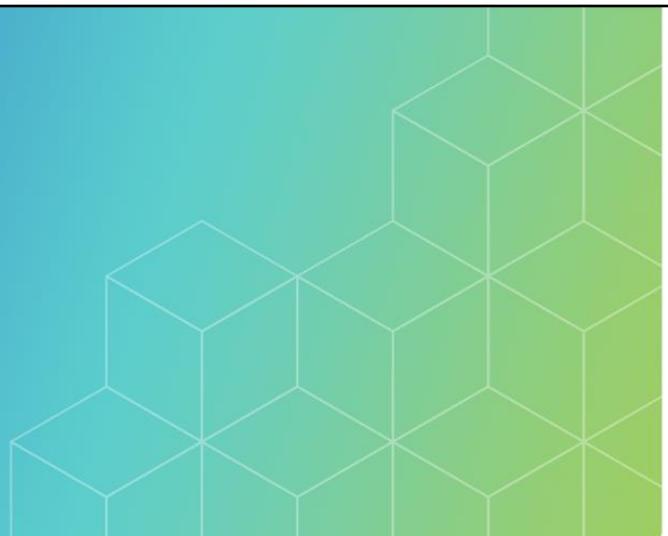
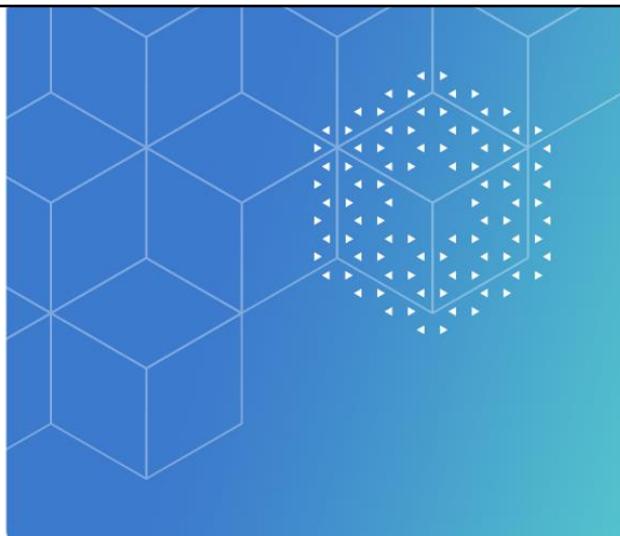


Why does the *end* iterator “point” to the position *after* the end of the sequence?



Given a sorted input sequence, can the result of unique() have adjacent duplicate values?

Please pause here to consider these questions.



Submodule 17-2

Standard Almost-Containers

cadence®

This training submodule describes some other storage classes that do not meet standard container requirements.

Submodule Objectives

In this training submodule, you

- Store and manipulate data by using standard almost-containers

This training submodule discusses:

- Bit sets
- Strings
- Value Arrays



Your objective is to store and manipulate data by using standard almost-containers.

To help you to achieve your objective, this training module discusses:

- Bit sets;
- Strings; and
- Value Arrays.

What Is a Standard Almost-Container?



A Standard Almost-Container is an informal term for template classes that offer container-like features while not quite meeting standard container requirements.

- The standard provides container type templates, each optimized for different use models, and all mostly share a common interface.
- The standard also provides almost-container type templates not conforming to the standard model.

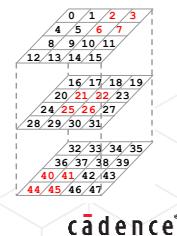
`bitset<8> bs8(42)`

7	6	5	4	3	2	1	0	pos
0	0	1	0	1	0	1	0	42

`string my_string("We Love C++")`

0	1	2	3	4	5	6	7	8	9	10	pos
W	e		L	o	v	e		C	+	+	

`valarray<unsigned> vu({0,1,2,etc.})`



305 © Cadence Design Systems, Inc. All rights reserved.

A Standard Container is an object that stores other objects, and controls allocation and deallocation of those objects, while conforming to a well-defined interface.

The standard provides container type templates, each optimized for different use models, and all mostly sharing a common interface.

The standard also provides almost-container type templates not conforming to the standard model.

A Standard Almost-Container is an informal term for the template classes that offer container-like features while not quite meeting standard container requirements.

The illustration conceptually depicts a bit set, a string, and slicing a value array.

- Bit sets are similar to the bit vectors of hardware design languages, supporting assignment from integral and string values, conversion to integral values, bitwise and reduction logical operations, inversion, and shifting.
- Strings are similar to C arrays of characters, but natively support additional operations, such as automatic resizing, and to compare, extract, and find strings.
- Numerical arrays are similar to C arrays of numbers, but natively support additional operations, such as array multiplication, element selection by slice, mask, or indirection, shifting, rotation, summing across all elements, and simultaneously applying any operation to all elements. You can treat a value array as having an arbitrary number of dimensions, cut a slice in any direction across those dimensions, and get or set the values of that cut set.

The depicted general slice is with `strides[] = {19,4,1}` and `lengths[] = {3,2,2}`

Declaring and Using Bit Sets

```
std::bitset<size_t> identifier
bitset<6> bs42;
```



Specifically intended for operations on a set of bits

- Less efficient than built-in types for $N \leq 64$
- Safer than user-defined operations for $N > 64$
- Constructor initialization from integer or string (0,1) values
- Bit references interchangeable with **bool**
- Conversion functions `to_string()`, `to_ulong()`, `to_ullong()`
- Operators `~, []`, `==`, `<<, >>`
- Assignment operators `&=`, `|=`, `^=`, `<<=`, `>>=`
- Useful functions `size()`, `all()`, `any()`, `none()`, `count()`, `set()`, `reset()`, `test()`, `flip()`

```
bitset<10> bs911 (string("1110001111"));
bitset<5> bs0030 ("11100");
bitset<9> bs420 (420); // 110100100
bitset<6> bs42; cout << bs42 << endl; // 000000
bs42.set(); cout << bs42 << endl; // 111111
bs42.reset(1); cout << bs42 << endl; // 111101
bs42[3]=0; cout << bs42 << endl; // 110101
bs42<<=1; cout << bs42 << endl; // 101010
bs42 = ~bs42; cout << bs42 << endl; // 010101
bs42.flip(3); cout << bs42 << endl; // 011101
cout.setf(ios_base::boolalpha);
cout << bs42.test(3) << endl; // true
cout << bs42.all() << endl; // false
cout << bs42.any() << endl; // true
cout << bs42.none() << endl; // false
cout << bs42.count() << endl; // 4
cout << bs42.size() << endl; // 6
```



306 © Cadence Design Systems, Inc. All rights reserved.

The standard library provides a bit set, primarily to replace those situations where a user would otherwise write their own code to manipulate multiple integer objects representing a long bit stream. Operations on bit sets are less efficient than operations on built-in integer types, but also much less error-prone and much more maintainable than typical user code.

A bit set is not a standard container and does not offer the features required of a standard container.

The example declares four bit set objects, and on one of them performs a representative set of operations.



Quick Reference Guide: Bit Set Functions

Type	Function	Parameters	Description
-	<code>bitset</code>	<code>()</code> <code>(unsigned long long)</code> <code>(const basic_string<...>....,</code> <code>(const charT* str,,,</code>	Constructors
<code>bool</code>	<code>all</code> <code>any</code> <code>none</code> <code>test</code> <code>operator[]</code> <code>operator==</code>	<code>()</code> <code>()</code> <code>()</code> <code>(size_t)</code> <code>(size_t)</code> <code>(const bitset<N>&)</code>	All bits 1 Any bits 1 No bits 1 This bit 1 This bit 1 Bits sets equal
<code>reference</code>	<code>operator[]</code>	<code>(size_t)</code>	Reference to bit
<code>size_t</code>	<code>count</code> <code>size</code>	<code>()</code> <code>()</code>	Count of bits 1 Size of bit set
<code>unsigned [long] long</code>	<code>to_[1]long</code>	<code>()</code>	Convert to [long] long
<code>bitset<N></code>	<code>operator~</code> <code>operator<<, >></code>	<code>()</code> <code>(size_t)</code>	Invert bits Shift left, right
<code>bitset<N>&</code>	<code>flip</code> <code>set</code> <code>reset</code> <code>operator<=, >=</code> <code>operator&=, =, ^=</code>	<code>([size_t])</code> <code>([size_t, [bool]])</code> <code>([size_t])</code> <code>(size_t)</code> <code>const bitset<N>&</code>	Flip all or specified bit Set all or specified bit Reset all or specified bit Shift left, right Bitwise operations

307 © Cadence Design Systems, Inc. All rights reserved.



The bit set (`bit_set`) class facilitates operations on a set of bits. The bits are ordered, so can be converted from and to integral values. They are not themselves integral values, and so do not support arithmetic operations.

- The constructor functions construct a bit set, and optionally initialize it to an integral value, or to a character string or substring.
- Boolean functions return whether all, any, or no bits are 1, whether a specific bit is 1, and whether two bit sets are equal.
- A subscript operator returns a non-constant reference to a specific bit. The reference is a class locally-defined to support bit operations, such as assignments to and from the bit.
- The function “`count`” returns the count of the number of bits that are 1.
- The function “`size`” returns the bit set size. The bit set size is statically set and cannot dynamically change.
- Functions convert the bit set to an integral value.
- Functions invert the bit set expression bits, or shift the bit set expression bits left or right.
- Functions flip, set, or reset the bit set, shift it left or right, and perform bitwise operations with another bit set.

Declaring and Using Basic Strings

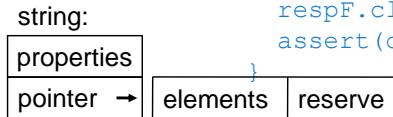


```
std::basic_string <chart, traitsopt, allocatoropt> identifier
basic_string<char> my_string; // same as string my_string
```

The `<string>` library defines the storage and manipulation of character sequences.

Strings are not true containers, but support most sequence container operations

- Constructors
- Assignment operations
- Iterators: forward/backward
- Access: `front()`, `back()`, `at()`, `[]`
- Insert/erase by iterator/position
- Push/Pop `front/back`
- Miscellaneous operations
- String-specific operations



```
const unsigned size = 32;
for (char test='1';test<='9';++test) {
    string stim=string("stim")+=test;
    string resp=string("resp")+=test;
    ifstream stimF(stim.c_str());
    ifstream respF(resp.c_str());
    stimF.read((char*)stimA,size);
    respF.read((char*)respA,size);
    stimF.close();
    respF.close();
    assert(do_test(stimA,respA,size));
}
```



308 © Cadence Design Systems, Inc. All rights reserved.

The strings library defines storage and manipulation of character sequences.

The time-complexity requirements for string operations suggest that a string is essentially a character vector. The fact that strings have reserve capacity functions identical to those of vectors also supports this suggestion. However, an additional level of indexing would also meet the time-complexity requirements.

Although strings are not true containers, they provide most of the sequence container functions. Like vectors, they provide front and back access, but no front modification.

The example, in a loop, generates stimulus and response test file names, opens and reads the files, closes the files, and performs a test that uses the file contents.

What Are Character Traits?



Character Traits are class templates defining additional names and functions of its `charT` specialization needed to implement string, string view, and iostream class templates.

For example, how to assign, compare, copy, find, and move characters of that type, and how to convert characters of that type to an integer value.

- The placeholder `charT` represents a character container class, of which the standard defines 5.
- For these 5, the standard defines specialized `char_traits<>` classes and declares string class names.

	Standard	“character traits” classes	String class names
<code>char</code>	implementation-defined	<code>char_traits<char></code>	<code>string = basic_string<char></code>
<code>char8_t</code>	ISO/IEC 10646 UTF-8	<code>char_traits<char8_t></code>	<code>u8string = basic_string<char8_t></code>
<code>char16_t</code>	ISO/IEC 10646 UTF-16	<code>char_traits<char16_t></code>	<code>u16string = basic_string<char16_t></code>
<code>char32_t</code>	ISO/IEC 10646 UTF-32	<code>char_traits<char32_t></code>	<code>u32string = basic_string<char32_t></code>
<code>wchar_t</code>	implementation-defined	<code>char_traits<wchar_t></code>	<code>wstring = basic_string<wchar_t></code>

`char` represents the basic character set and can be signed or unsigned and you can explicitly declare it signed or unsigned.
`wchar_t` represents the extended character set and can be signed or unsigned and you can explicitly declare it signed or unsigned.
ISO/IEC 10646 character types are unsigned.

- You could (with much effort!) define your own `my_Char` class and `char_traits<my_Char>` specialization for use with string, string view, and iostream class templates.



Character Traits are class templates defining additional names and functions of its “character type” (`charT`) specialization needed to implement string, string view, and input/output stream class templates.

For example, how to assign, compare, copy, find, move characters of that type, and how to convert characters of that type to an integer value.

The “character type” placeholder represents a character container class, of which the standard defines five.

For these five the standard defines specialized “character traits” (`char_traits<>`) classes and declares string class names.

- The unsized character type (`char`) is defined by the implementation as a signed or unsigned representation of the implementation’s basic character set.
- The sized character types are unsigned and required to be at least their indicated bit width.
- The wide character type (`wchar_t`) is defined by the implementation as a signed or unsigned representation of the implementation’s widest supported character set.

You could (with much effort!) define your own `my_Char` class and “character traits” specialization for use with string, string view, and input/output stream class templates.

Using Basic String Literals



```
basic_string<chart> operator""s (const chart*, size_t)C++14  
auto my_string = "my_string"s; // deduced type is string instead of char*
```

The <string> library defines overloaded operators that generate a string literal from a character array.

- Explicitly indicates desired type
- Allows embedded null characters

Borrowed (and modified) from:

https://en.cppreference.com/w/cpp/string/basic_string/operator%22%22s

```
#include <string>  
#include <iostream>  
using namespace std;  
using namespace string_literals;  
int main() {  
    string s1 = "abc\0def";  
    string s2 = "abc\0def"s;  
    cout << "s1: " << s1.size() << " \" " << s1 << "\n";  
    cout << "s2: " << s2.size() << " \" " << s2 << "\n";  
    return 0;  
}
```

s1: 3 "abc"
s2: 8 "abcdef"

310 © Cadence Design Systems, Inc. All rights reserved.



The strings library defines overloaded operators that generate a string literal from a character array. To do this has at least the two advantages that you explicitly indicate the desired result type and you can embed null characters.

This classical example illustrates this. It first initializes a string from a character array literal. It then initializes a string from a string literal. The first string has only the characters up to the first null character. The second string includes the null characters.

Borrowed (and modified) from:

https://en.cppreference.com/w/cpp/string/basic_string/operator%22%22s



Quick Reference Guide: String Construction, Copy, Capacity, Access

Type	Function	Parameters	Description
-	<code>basic_string</code>	(<code>[const Allocator&]</code>) (<code>size_type n, charT [,const Allocator&]</code>) (<code>const T& [,size_type p, size_type n] [,const Allocator&]</code>) ^{C++17} (<code>const charT*</code> [, <code>size_type n</code>] [, <code>size_type n</code>] [, <code>const Allocator&]</code>) (<code>const basic_string& [,size_type p [,size_type n]] [,const Allocator&]</code>) (<code>const basic_string&&</code> [, <code>const Allocator&</code>]) (<code>InputIterator, InputIterator</code> [, <code>const Allocator&</code>]) (<code>initializer_list<charT></code> [, <code>const Allocator&</code>]) ^{C++11}	default constructor from character from character from [substring] copy [substring] move [substring] from string range from initializer list
<code>basic_string&</code>	<code>operator=</code>	(<code>charT</code>), (<code>const T&</code>) ^{C++20} (<code>const charT*</code>) (<code>const basic_string&</code>), (<code>basic_string&&</code>) (<code>initializer_list<charT></code>)	Copy from character Copy from c-string Copy/move from string Copy from initializer list
<code>void</code> <code>size_type</code> <code>bool</code> <code>size_type</code> <code>size_type</code> <code>void</code> <code>void</code> <code>void</code> <code>size_type</code>	<code>clear</code> <code>capacity()</code> <code>empty</code> <code>length</code> <code>max_size</code> <code>reserve</code> <code>resize</code> <code>shrink_to_fit</code> <code>size</code>	() () () () () (<code>size_type</code>) (<code>size_type [,charT]</code>) () ()	Clear Capacity Is empty? Length Maximum size Reserve capacity Resize (optionally fill) Shrink capacity to size Size
<code>[const_]reference</code> <code>[const] charT&</code> <code>[const_]reference</code>	<code>at</code> <code>back/front</code> <code>operator[]</code>	(<code>size_type</code>) () (<code>size_type</code>)	Reference by function Character at front/back Reference by subscript

311 © Cadence Design Systems, Inc. All rights reserved.



Constructors construct either an empty string, or a string having the contents of a character, a subset of a string or C-string, or an initializer list.

Assignment operators update the string contents to that of a character, string, C-string, or initializer list.

Capacity functions get and set the string size, reserve and release extra capacity, and clear the string.

Access functions get a constant or non-constant reference to the character at the front, back, or specified position.



Quick Reference Guide: String Modifiers [1/2]

Type	Function	Parameters	Description
<code>basic_string&</code>	<code>append</code> , <code>assign</code>	(<code>size_type n, charT</code>) (<code>const T& [,size_type p [,size_type n]]</code>) (<code>const basic_string& [,size_type p [,size_type n]]</code>) (<code>const charT* [,size_type n]</code>) (<code>InputIterator first, InputIterator last</code>) (<code>initializer_list<charT></code>)	Character Character [Sub]-String [Sub]-C-string Between iterators Initializer list
<code>size_type</code>	<code>copy</code>	(<code>charT*, size_type n [,size_type p]</code>)	Copy [sub]string to C-string
<code>basic_string& iterator</code>	<code>erase</code>	(<code>[size_type p [,size_type n]]</code>) (<code>const_iterator [,const_iterator]</code>)	Erase at p n characters Erase between iterators
<code>basic_string&</code>	<code>insert</code>	(<code>size_type p, size_type n, charT</code>) (<code>size_type p1, const T& [,size_type p2 [,size_type n]]</code>) (<code>size_type p1, const basic_string& [,size_type p2 [,size_type n]]</code>)	Insert at p n characters Insert at p sub-string
<code>iterator</code>	<code>insert</code>	(<code>const_iterator p, charT</code>) (<code>const_iterator p, size_type n, charT</code>) (<code>const_iterator p, InputIterator, InputIterator</code>) (<code>const_iterator p, initializer_list<charT></code>)	Insert at p 1 character Insert at p n characters Insert at p from iterators Insert at p initializer list
<code>basic_string&</code>	<code>operator+=</code>	(<code>charT</code>), (<code>const T&</code>) (<code>const charT*</code>) (<code>const basic_string&</code>) (<code>initializer_list<charT></code>)	Append character Append C-string Append string Append initializer list
<code>void</code>	<code>push_back</code> <code>pop_back</code>	(<code>charT</code>) (<code>()</code>)	Push character Pop character

312 © Cadence Design Systems, Inc. All rights reserved.



The functions “**append**”, append to the string either, a number of the specified character, a subset of a string or C-string, or the characters between two iterators or from an initializer list.

The functions “**assign**”, assign to the string either, a number of the specified character, a subset of a string or C-string, or the characters between two iterators or from an initializer list.

The functions “**copy**”, copy to a C-string, a subset of the string.

The functions “**erase**”, erase from the string either, one or more characters starting at a position, or the characters between two iterators.

The functions “**insert**”, insert at a position or iterator either, a number of the specified character, a subset of a string, or the characters between two iterators or from an initializer list.

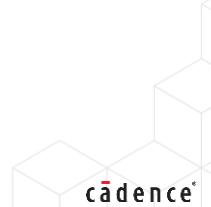
The overloaded compound assignment operator, assigns to the string either, the specified character, a string or C-string, or the characters from an initializer list.

The function “push back” (**push_back**), pushes a character onto the string back, and the function “**pop**”, pops a character from the string back. Strings do not support front operations.



Quick Reference Guide: String Modifiers [2/2]

Type	Function	Parameters	Description
<code>basic_string&</code>	<code>replace</code>	<pre>(size_type p, size_type n1, size_type n2, charT c) (size_type p, size_type n1, const charT* s [,size_type n2]) (size_type p1, size_type n1, const T& [,size_type p2 [,size_type n2]]) (const_iterator i1, const_iterator i2, size_type n, charT c) (const_iterator i1, const_iterator i2, size_type n, const T&) (const_iterator i1, const_iterator i2, const charT* s [,size_type n]) (const_iterator i1, const_iterator i2, const basic_string& s) (const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2) (const_iterator i1, const_iterator i2, initializer_list<charT>)</pre>	<p>p to p+n1 with n2 of c p to p+n1 with 0 to n2 of s i1 to i2 with n of c i1 to i2 with 0 to n of s i1 to i2 with s i1 to i2 with j1 to j2 i1 to i2 with initializer list</p>
<code>void</code>	<code>swap</code>	<code>(basic_string&)</code>	Swap with other string



The functions “**replace**”, replace characters at a position or between iterators with either, a number of the specified character, a subset of a C-string, a string, or the characters between two iterators or from an initializer list.

The function “**swap**” swaps two strings.



Quick Reference Guide: String Operations

Type	Function	Parameters	Description
const charT* [const] charT*	c_str data	() ()	C-string
int -1:this<other 0:this==other +1:this>other	compare	(const charT*) (const basic_string&), (const T&) (size_type p1, size_type n1, const charT* [,size_type n2]) (size_type p1, size_type n1, const T& [,size_type p2 [,size_type n2]]) (size_type p1, size_type n1, const basic_string& [,size_type p2, size_type n2])	string to C-string string to string [sub]string to [sub]C-string [sub]string to [sub]string
size_type	find, rfind, find_first_ [not_]of, find_last_ [not_]of	(charT c [,size_type p]) (const charT* s [,size_type p [,size_type n]]) (const basic_string& s [,size_type p]) (const T& [,size_type p])	char c starting at p first n of s starting at p s starting at p
bool	{starts ends} _with ^{C++20}	(charT) (const charT*) (basic_string_view<charT, traits>)	with character with C-string
basic_string	substr	([size_type p [,size_type n]])	substring from p tp p+n
basic_string_<>	operator basic_string_<>	()	Convert to basic string view
allocator_type	get_allocator	()	Allocator

314 © Cadence Design Systems, Inc. All rights reserved.



The functions “C-string” (**c_str**) and “**data**”, return a C-string of the string’s characters.

The functions “**compare**”, compare a string or substring to some other whole or subset of a string or C-string.

The “find” functions, search for a character or substring, optionally from some starting position.

The functions “starts with” (**starts_with**) and “ends with” (**ends_with**), reveal the truth of whether the string starts or ends with the specified character or C-string.

The function “substring” (**substr**) returns a string constructed from the specified string subset.

The “basic string view” (**basic_string_view**) conversion operator constructs and returns a “basic string view” object.

The function “get allocator” (**get_allocator**), returns a copy of the current allocator.

Declaring and Using Numeric Arrays

```
std::valarray <T> identifier
valarray<float> vf_dynamic;
```

Specifically intended to optimize numerical operations.

- Constructors and assignments accept scalars and value arrays
- Most operators apply in parallel to all elements.
 - Any operation valid for the element is valid for the array.
- Some functions apply to the array as a whole.
 - *min()*, *max()*, *sum()*, *shift()*, and *cshift()*
- The *apply()* function applies your arbitrary numeric function to each element.
- Subscripting can be by integer, by slice, or by value array.
 - Simplifies selection of regular and irregular value array subsets.

```
valarray<float> v0; // 0 elements
valarray<float> v4(1,2,4); // 4 elements 1.2
valarray<float> v5a(5); // 5 elements 0.0
float f5[]={-2,-1,0,1,2}; // 5 element c-array
valarray<float> v5b(f5,4); // 5 elements of f5[]
float f9[]={0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8};
valarray<float> v9(f9,9); // 9 elements from f9[]
v5a += 1.0; // 1.0 1.0 1.0 1.0 1.0
v5b *= 2.0; // -4.0 -2.0 0.0 2.0 4.0
v0.resize(3); // reinitialize
bool b[] = {1,0,1,0,1}; // 5 element c-array mask
v0 = v5b[valarray<bool>(b,5)]; // -4.0 0.0 4.0
size_t i[] = {8,6,4,2,0}; // 5 element c-array indices
v5a = v9[valarray<size_t>(i, 5)]; // 8.8 6.6 4.4 2.2 0.0
v0 = v5b[v5b >= 0.0f]; // 0.0, 2.0, 4.0
float f = (v5a*v5b).sum(); // -44
v0 = v9[slice(1, 3, 2)]; // slice start, length,stride
// v0 is 1.1 3.3 5.5
size_t clenLengths[] = {2,2}; // lengths
size_t cstrides[] = {3,1}; // strides
valarray<size_t> lengths(clenLengths,2);
valarray<size_t> strides(cstrides,2);
v4 = v9[gslice(0,lengths,strides)]; // 0.0 1.1 3.3 4.4
```

315 © Cadence Design Systems, Inc. All rights reserved.



You can construct an empty value array, or a value array initialized with a scalar value, or default values, or from an array of values. The array of values can be the whole or a subset of a value array.

Operations involving value arrays are typically parallel operations across all elements. Value array functions “minimum” (**min**), “maximum” (**max**), “**sum**”, “**shift**”, and “cyclic shift” (**cshift**) apply to the array as a whole.

You can subscript the value array by integers, slices, general slices, or other value arrays:

- The compiler interprets a subscript that is a value array of Boolean elements, as a mask. Here, the example uses a value array of Boolean elements to select elements from another value array.
- The compiler interprets a subscript that is a value array of “size-type” (**size_t**) elements, as an indirect selection. Here, the example uses a value array of “size-type” elements as indices to select elements from another value array.
- The compiler interprets a subscript that is a relational operation involving a value array, to be a filter. Here, the example uses a relational expression to copy only the elements of a value array that have positive values.
- Among the useful member functions is to sum the value array elements, which we use here to generate a dot-product.
- To slice a value array, you provide the starting point, the number of steps to take, and the step size. For the general slice, the number of steps and step size are themselves value arrays, thus can slice an array across any number of dimensions.

As the library developers intended value array operations to be aggressively optimized and targeted toward parallel vector processing, perhaps even implemented in hardware, runtime checking is nil – subscript abuse leads to, at best, a segmentation fault.



Quick Reference Guide: Value Array Functions

Type	Function	Parameters	Description
-	<code>valarray</code>	([[const T& T*,]size_t]]) (<code>valarray&</code>), (const <code>array<T>&</code>) (<code>initializer_list<T></code>)	Constructors <code>array</code> ::= { <code>valarray</code> <code>slice_array</code> <code>gslice_array</code> <code>mask_array</code> <code>indirect_array</code> }
<code>valarray&</code>	<code>operator=</code>	(const T&) (<code>valarray&</code>), (const <code>array<T>&</code>) (<code>initializer_list<T></code>)	Assignment
<code>valarray&</code>	<code>operator*= etc.</code>	(const T&), (const <code>valarray&</code>)	Assignment (compound)
T& <code>valarray</code> <code>slice_array<T></code> <code>valarray</code> <code>gslice_array<T></code> <code>valarray</code> <code>mask_array<T></code> <code>valarray</code> <code>indirect_array<T></code>	<code>operator[]</code>	(size_t) (slice) (const <code>gslice&</code>) (const <code>valarray<bool>&</code>) (const <code>valarray<size_t>&</code>)	Access Slice Generalized slice Mask Indirect
<code>valarray</code> <code>valarray<bool></code>	<code>operator+, -, ~</code> <code>operator!</code>	()	Unary operators
size_t T void void <code>valarray</code> <code>valarray</code>	<code>size</code> <code>min, max, sum</code> <code>resize</code> <code>swap</code> <code>[c]shift</code> <code>apply</code>	() () (size_t [,T]) (<code>valarray&</code>) (int) (T func(T const T&))	Size Minimum, maximum, sum Resize Swap two arrays Shift, cyclic shift Apply function to elements

316 © Cadence Design Systems, Inc. All rights reserved.



The value array (**valarray**) class facilitates operations on numeric arrays.

- The constructor functions construct a value array, and optionally initialize it to a numerical value or to an array or initializer list of numerical values.
- The simple assignment operator updates a value array with a numerical value or an array or initializer list of numerical values.
- The compound assignment operators update a value array with a numerical value or a value array.
- The subscript operators return either an element value, or a slice or generalized slice of a numerical array, or a mask or selection from a numerical array.
- The unary operators return a value array with each element appropriately operated on.
- Other member functions either get the array size or the minimum, maximum, or sum of its elements, resize the array, swap two value arrays, or return a shifted array, or return an array to which a specified function is applied to each array element.

Example: Numeric Array Generalized Slice Code

```
// concatenate v2 rows onto v1 rows to form vr
template<class T> void concat (
    unsigned height, const valarray<T>& v1,
    const valarray<T>& v2, valarray<T>& vr) {
    size_t width1 = v1.size() / height;
    size_t width2 = v2.size() / height;
    vr.resize(v1.size() + v2.size());
    { // insert v1
        size_t clengths[] = {height, width1};
        size_t cstrides[] = {width1 + width2, 1};
        valarray<size_t> lengths(clengths, 2);
        valarray<size_t> strides(cstrides, 2);
        vr[gslice(0, lengths, strides)] = v1;
    }
    { // insert v2
        size_t clengths[] = {height, width2};
        size_t cstrides[] = {width1 + width2, 1};
        valarray<size_t> lengths(clengths, 2);
        valarray<size_t> strides(cstrides, 2);
        vr[gslice(width1, lengths, strides)] = v2;
    }
}
```

```
// Example valarray 1
v1: { 0, 1, 2, 3,
       7, 8, 9, 10,
       14, 15, 16, 17};
```

```
// Example valarray 2
v2: { 4, 5, 6,
       11, 12, 13,
       18, 19, 20};
```

Length and stride describe placement of cols and rows.

For v1

- *length* is for cols: height; for rows: width1
- *stride* is for cols: width1+width2; for rows: 1
- start at position 0

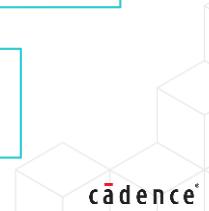
Length and stride describe placement of cols and rows.

For v2

- *length* is for cols: height; for rows: width2
- *stride* is for cols: width1+width2; for rows: 1
- start at position width1

```
// valarray result
vr: { 0, 1, 2, 3, 4, 5, 6,
       7, 8, 9, 10, 11, 12, 13,
       14, 15, 16, 17, 18, 19, 20};
```

317 © Cadence Design Systems, Inc. All rights reserved.



For the generalized slice, the number of steps and step size, are value arrays, thus can slice an array across any number of dimensions.

The example uses the generalized slice to determine the target array elements to update. The example function concatenates the rows of two input value arrays to a third value array output. Function arguments are the array height, two input array constant references, and an output array non-constant reference. The function separately copies each input array to an appropriately computed generalized slice of the output array.

Arguments to the generalized slice function are the slice starting position, an array of slice lengths, and an array of slice strides.

Quiz: Standard Almost-Containers



Suggest some features of `bitset<n>` that are not directly available with `vector<bool>`.

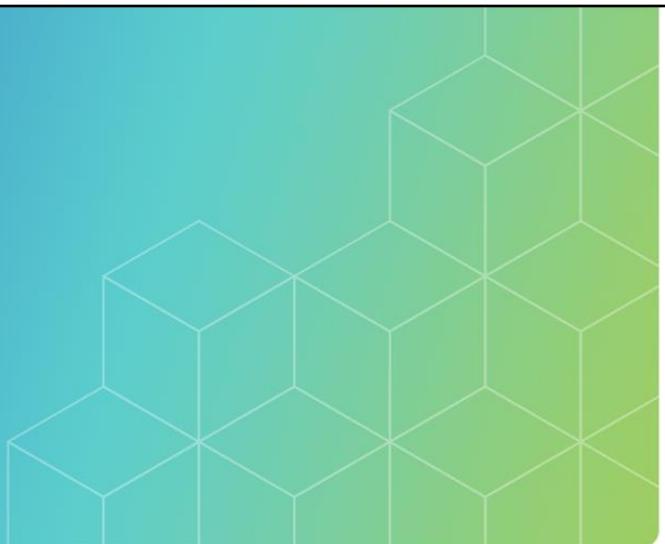
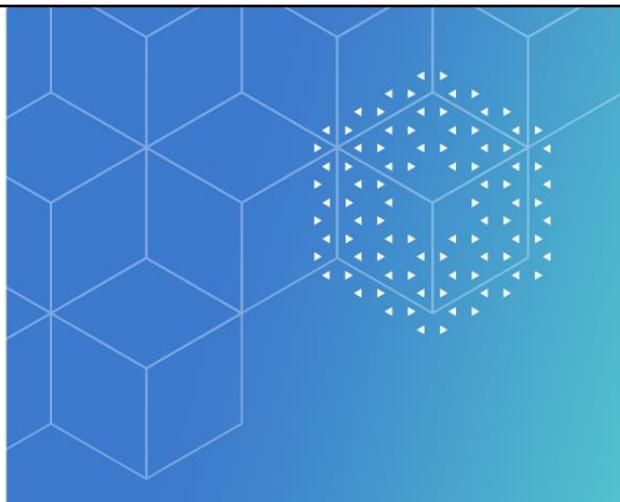


Suggest some features of `basic_string<charT>` that are not directly available with `vector<charT>`.



Suggest some features of `valarray<T>` that are not directly available with C arrays `T[]`.

Please pause here to consider these questions.



Submodule 17-3

Standard Algorithms

cadence®

This training submodule describes some standard algorithms with which you can manipulate standard containers.

Submodule Objectives

In this training submodule, you

- Manipulate data by using standard algorithms

This training submodule discusses:

- The <algorithm> Library

320 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to manipulate data by using standard algorithms.

To help you to achieve your objective, this training module discusses:

- The Algorithm Library.

This training module, describes only a few of the many available algorithms, and does not address the numeric, memory or C-standard library algorithms.

What Is a Standard Algorithm?



A **Standard Algorithm** is a function declared in the standard namespace that performs algorithmic operations on containers and other sequences.

The algorithms library provides functions for accessing and operating on sequences.

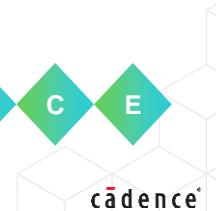
- Simplifies your code (why re-invent the basic wheel?)
- Aggressively optimized for runtime performance.
- Many algorithms, many just variations on a theme – use what you need!
- Works for any data types implementing required iterator types.

Common operators:

- Access
- Modify
- Order
- Compare
- Permute



Algorithms can move sequence elements but do not insert or delete elements.



321 © Cadence Design Systems, Inc. All rights reserved.

A **Standard Algorithm** is a function declared in the standard namespace that performs algorithmic operations on containers and other sequences.

You can write your own algorithms for manipulating sequences, but why re-invent the basic wheel? The vast majority of common operations are already available for you in the algorithm (`<algorithm>`) library, generically coded and aggressively optimized.

The algorithms work with iterators, and do not know about, or care about, the underlying sequence. That means that the algorithms cannot modify the sequence size – they cannot insert or remove elements.



Quick Reference Guide: Standard Algorithms by Category

Category	Example Algorithms
Non-modifying	<code>all_of()</code> , <code>any_of()</code> , <code>none_of()</code> , <code>count()</code> , <code>count_if()</code> , <code>equal()</code> , <code>find()</code> , <code>find_if()</code> , <code>find_if_not()</code> , <code>find_end()</code> , <code>find_first_of()</code> , <code>adjacent_find()</code> , <code>for_each()</code> , <code>mismatch()</code> , <code>search()</code> , <code>search_n()</code>
Modifying	<code>copy()</code> , <code>copy_backward()</code> , <code>copy_if()</code> , <code>copy_n()</code> , <code>fill()</code> , <code>fill_n()</code> , <code>generate()</code> , <code>generate_n()</code> , <code>move()</code> , <code>move_backward()</code> , <code>remove()</code> , <code>remove_if()</code> , <code>remove_copy()</code> , <code>remove_copy_if()</code> , <code>replace()</code> , <code>replace_if()</code> , <code>replace_copy()</code> , <code>replace_copy_if()</code> , <code>reverse()</code> , <code>reverse_copy()</code> , <code>rotate()</code> , <code>rotate_copy()</code> , <code>sample()</code> , <code>shift_left()</code> , <code>shift_right()</code> , <code>shuffle()</code> , <code>swap()</code> , <code>swap_ranges()</code> , <code>iter_swap()</code> , <code>transform()</code> , <code>unique()</code> , <code>unique_copy()</code>
Partitioning	<code>is_partitioned()</code> , <code>partition()</code> , <code>partition_copy()</code> , <code>partition_point()</code> , <code>stable_partition()</code>
Sorting	<code>is_sorted()</code> , <code>is_sorted_until()</code> , <code>sort()</code> , <code>partial_sort()</code> , <code>partial_sort_copy()</code> , <code>stable_sort()</code> , <code>nth_element()</code>
Sorted ops	<code>binary_search()</code> , <code>equal_range()</code> , <code>lower_bound()</code> , <code>upper_bound()</code> , <code>merge()</code> , <code>inplace_merge()</code>
Sorted set ops	<code>includes()</code> , <code>set_difference()</code> , <code>set_symmetric_difference()</code> , <code>set_intersection()</code> , <code>set_union()</code>
Heap ops	<code>is_heap()</code> , <code>is_heap_until()</code> , <code>make_heap()</code> , <code>pop_heap()</code> , <code>push_heap()</code> , <code>sort_heap()</code>
Min/Max	<code>max()</code> , <code>max_element()</code> , <code>min()</code> , <code>min_element()</code> , <code>minmax()</code> , <code>minmax_element()</code> , <code>clamp()</code>
Comparison	<code>equal()</code> , <code>lexicographical_compare()</code> , <code>lexicographical_compare_three_way()</code> ,
Permutation	<code>is_permutation()</code> , <code>next_permutation()</code> , <code>prev_permutation()</code>

Binary search, merge, and set operations require that the sequence be sorted.

322 © Cadence Design Systems, Inc. All rights reserved.



These are the algorithm (`<algorithm>`) library algorithms, partitioned for your convenience into seven general categories.

The library offers many algorithms for operating on sequence containers.

- Non-modifying algorithms provide the means with which to iterate through sequence elements, count elements having a specific value, find an element of a specific value, search for subsequences, and determine whether two sequences are identical;
- Modifying algorithms include those to transform, replace, or remove elements, or to copy or swap sequences;
- Ordering operations provide many variations of sort, merge, and partition operations, and an optimized binary search for sorted sequences;
- Set operations return a Boolean value indicating whether a first sequence includes a second sequence, or return a sequence that is the union, intersection, or difference between two sequences; Input sequences must be pre-sorted.
- Heap operations organize the sequence in order of highest to lowest value;
- Comparison algorithms return the minimum or maximum either of two values or of all elements in a sequence, or a Boolean value to indicate whether a first sequence is lexicographically less than a second sequence; and
- Permutation algorithms permute the sequence in lexicographical order and return a Boolean value indicating whether a next permutation or previous permutation exists.

The numeric (`<numeric>`) and memory (`<memory>`) libraries offer additional algorithms, not displayed here.



Quick Reference Guide: Standard Algorithm Iterator Requirements

Iterator Type	Supports
common requirements	<code>X a(r), ~a, X a=r, *r, X& ++r</code>
InputIterator	common plus: <code>bool a!=b, T& *a, a->m, X& ++r, (void)r++, T *r++</code>
OutputIterator	common plus: <code>X& ++r, const X& r++, *r=o, *r++=o</code>
ForwardIterator	InputIterator plus: <code>const X& r++, T& *r++</code>
BidirectionalIterator	ForwardIterator plus: <code>X& --r, const X& r--, T& *r--</code>
RandomAccessIterator	BidirectionalIterator plus: <code>X& r+=n, X& r-=n, X a+n, X a-n, X n+a, Dt a-b, T& a[n], bool a<b, bool a>b, bool a<=b, bool a>=b</code>

$T \equiv$ contained type $X \equiv$ iterator type $r \equiv$ reference of type X $m \equiv$ identifier
 $t \equiv$ value of type T $a, b \equiv$ value type X $n \equiv$ value of difference type $\circ \equiv$ output type

323 © Cadence Design Systems, Inc. All rights reserved.



The standard algorithms place incremental requirements on iterators. To apply an algorithm to a container, the container must define iterators meeting those requirements.

For example:

- To reverse a container's contents requires an iterator supporting auto-decrement operations; and
- To sort a container's contents requires an iterator supporting iterator difference.

The standard algorithms reference these iterator categories. The categories are organized here with incremental capabilities.

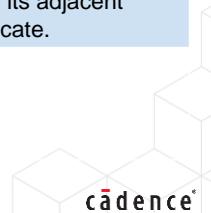
- All iterators must have a copy constructors and a destructor, be swappable with another same-type iterator, be dereferenceable, and support pre-increment.
- Input iterators meet the common requirements, plus support iterator equality, dereferencing, member access, and post-increment.
- Output iterators meet the common requirements, plus support updating contained elements, and post-increment.
- Forward iterators must meet the input iterator requirements, plus return a constant iterator reference from the post-increment operation, and an element reference from the dereferenced post-increment operation.
- Bidirectional iterators meet the requirements of forward iterators, plus support pre-decrement and post-decrement.
- Random-access iterators meet the requirements of bidirectional iterators, plus can operate with values of the difference between two iterators.



Quick Reference Guide: Some Standard Algorithm Templates for Finding an Element

Template	Description
<pre>template < class In, class T > In find (In first, In last, const T& val); template < class In, class Pred > In find_if (In first, In last, Pred p);</pre>	Find first matching element of sequence, using equality operator.
<pre>template < class For1, class For2 > For find_first_of (For1 first1, For1 last1, For2 first2, For2 last2); template < class For1, class For2 > For find_first_of (For1 first1, For1 last1, For2 first2, For2 last2, BinPred p);</pre>	Find first matching element of sequence 1 that matches any element of sequence 2, using equality operator.
<pre>template < class For > For adjacent_find (For first, For last); template < class For, class BinPred > For adjacent_find (For first, For last, BinPred p);</pre>	Find first element of sequence 1 that matches any element of sequence 2, using binary predicate.
<p>In ≡ InputIterator For ≡ ForwardIterator Pred ≡ Predicate (unary) BinPred ≡ BinaryPredicate</p>	Find first element matching its adjacent element, using equality operator. Find first element matching its adjacent element, using binary predicate.

324 © Cadence Design Systems, Inc. All rights reserved.



Some standard algorithms find a matching element.

- The function “**find**”, finds the location of the first sequence element that matches the provided value.
 - The function “**find if**” (**find_if**), similarly matches elements, but relies upon your provided unary predicate.
- The function “**find first of**” (**find_first_of**), finds the location of the first sequence 1 element that matches any sequence 2 element, using the equality operator, or your provided binary predicate; and
- The function “**adjacent find**” (**adjacent_find**), finds the location of the first sequence element that matches its adjacent element, using the equality operator, or your provided binary predicate.

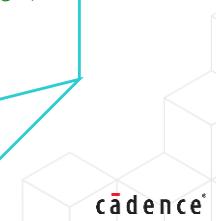
Example: Standard Algorithm “Find” Code

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

using container_t = list<int>;
using iter_t      = container_t::iterator;
using citer_t     = container_t::const_iterator;

int main() {
    container_t cont; iter_t iter; citer_t citer1, citer2;
    for (unsigned i=0; i<=9; ++i) cont.insert(cont.end(), i); // 0123456789
    iter = find (cont.begin(), cont.end(), 5); // iter to value 5
    reverse (cont.begin(), iter); // 4321056789
    citer1 = find (cont.cbegin(), cont.cend(), 2); // citer to value 2
    citer2 = find (cont.cbegin(), cont.cend(), 7); // citer to value 7
    // output maximum element between citer1 (2) and (excluding) citer2 (7)
    cout << *max_element (citer1,citer2) << endl; // 6
    // output maximum element between citer1 (2) and (including) citer2 (7)
    cout << *max_element (citer1,++citer2) << endl; // 7
    return 0;
}
```

325 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “**find**”.

The example:

- Declares a container;
- Inserts some values at the container end;
- Gets an iterator to the value 5;
- Reverses the element sequence between the sequence beginning and the iterator;
- Gets iterators to the values 2 and 7;
- Outputs the maximum element value between those two iterators; and
- Increments the 2nd iterator and again prints the maximum found value.

Example: Standard Algorithm “Find with Predicate” Code

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

using severity_t = enum {INFO, WARNING, ERROR, FATAL};
using container_t = vector<severity_t>;
using citer_t = container_t::const_iterator;

// unary predicate function
bool is_error (severity_t severity) { return severity == ERROR; }

int main () {
    container_t cont; citer_t citer;
    for (unsigned i=0;i<=9;++i)
        cont.insert(cont.cend(),(severity_t)(rand()%4));
    citer = find_if (cont.cbegin(), cont.cend(), is_error);
    if (citer != cont.cend()) cout << "ERROR was found" << endl;
    else cout << "ERROR not found" << endl;
    return 0;
}
```

326 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “find with predicate” (**find_if**).

The example:

- Declares an enumerated message severity type;
- Defines a unary predicate revealing whether a severity has the value “ERROR”;
- Declares a container of the message severity type;
- Inserts some random severity values at the container end;
- Attempts to find the location of the first element having the value “ERROR”; and
- Outputs whether the vector contains an element having the value “ERROR”.



Quick Reference Guide: Some Standard Algorithm Templates for Finding a Sequence

Template	Description
<pre>template < class For1, class For2 > For1 search (For1 first1, For1 last1, For2 first2, For2 last2); template < class For1, class For2, class BinPred > For1 search (For1 first1, For1 last1, For2 first2, For2 last2, BinPred p);</pre>	Find first occurrence in sequence 1 of sequence 2 using equality operator. Find first occurrence in sequence 1 of sequence 2 using binary predicate.
<pre>template < class For1, class For2 > For1 find_end (For1 first1, For1 last1, For2 first2, For2 last2); template < class For1, class For2, class BinPred > For1 find_end (For1 first1, For1 last1, For2 first2, For2 last2, BinPred p);</pre>	Find last occurrence in sequence 1 of sequence 2 using equality operator. Find last occurrence in sequence 1 of sequence 2 using binary predicate.
<pre>template < class For, class Size, class T > For search_n(For first, For last, Size n, const T& value); template<class For, class Size, class T, class BinPred> For search_n(For first, For last, Size n, const T& value, BinPred p);</pre>	Find first occurrence in sequence of n consecutive value using equality operator. Find first occurrence in sequence of n consecutive value using binary predicate.

327 © Cadence Design Systems, Inc. All rights reserved.

In	\equiv InputIterator	Pred	\equiv Predicate (unary)
For	\equiv ForwardIterator	BinPred	\equiv BinaryPredicate



Some standard algorithms find a matching sequence.

- The function “**search**”, finds within subsequence 1, the first occurrence of subsequence 2, using the equality operator, or your provided binary predicate;
- The function “**find end**” (**find_end**), finds within subsequence 1, the last occurrence of subsequence 2, using the equality operator, or your provided binary predicate;
- The function “**search N**” (**search_n**), finds within subsequence 1, the first occurrence of N consecutive elements having the specified value, using the equality operator, or your provided binary predicate.

Example: Standard Algorithm “Search” Code

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

using transaction_t = enum {WRITE, READ, BURST_WRITE, BURST_READ};
using container_t = vector<transaction_t>;
using citer_t = container_t::const_iterator;

int main() {
    container_t cont; citer_t citer;
    for (int i=0;i<=11;++i) cont.insert(cont.cend(),(transaction_t)(rand()%4));
    transaction_t pattern[] = {WRITE, READ};
    // pattern is array not vector so use address not iterator
    citer = search (cont.cbegin(), cont.cend(), pattern, pattern+1);
    if (citer != cont.cend())
        cout << "Pattern found at " << (unsigned)(citer-cont.cbegin()) << endl;
    else
        cout << "Pattern not found" << endl;
    return 0;
}
```

328 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “search”.

The example:

- Declares an enumerated transaction type;
- Declares a container of the transaction type;
- Inserts some random transaction values at the container end;
- Attempts to find the location of the first sequence of “WRITE” transaction followed by “READ” transaction; and
- Outputs whether the container contains that sequence, and if so, the its position in the container.

Example: Standard Algorithm Report Parser “Search” Code

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ifstream report("report.txt", ios::in);
    string line;
    while (getline(report, line) ) {
        vector<string> msgs;
        string pattern[] = { "INFO:::", "WARNING:::", "ERROR:::", "FATAL:::" };
        string::const_iterator b1 = line.cbegin(), e1 = line.cend(), b2 = b1, e2, b3;
        while (b2 != e1) { // while search region begin is not line end
            for ( unsigned i=0; i<=3; ++i ) // find a pattern within search region
                if ( (b3 = search(b2, e1, pattern[i].cbegin(), pattern[i].cend())) != e1 ) break;
            if ( b3 == e1 ) break; // no pattern found within search region
            e2 = b3; // continue search from pattern begin
            // increment iterator until line end or not white
            while ( e2!=e1 && *e2!=' ' && *e2!='\n' && *e2!='\t' ) ++e2;
            msgs.push_back(string(b3, e2)); // push message onto container
            b2 = e2; // new search region at message end
        }
        vector<string>::const_iterator citer; // output messages on this line
        for(citer=msgs.cbegin();citer!=msgs.cend();++citer) cout<<*citer<<endl;
        msgs.resize(0); // clear message container
    }
    return 0;
}
```

start of report
 NOTE::generator:start@100ns
 WARNING::channel:xmit@200ns Mary ERROR::monitor:broke@300ms
 had NOTE::channel:rcv@400ns a WARNING::generator:stop@500ns little
 lamb
 FATAL::testbench:kill@600ns
 end of report

INFO::generator:start@100ns
 WARNING::channel:xmit@200ns
 ERROR::monitor:broke@300ms
 INFO::channel:rcv@400ns
 WARNING::generator:stop@500ns
 FATAL::testbench:kill@600ns

329 © Cadence Design Systems, Inc. All rights reserved.



The example searches for and outputs messages embedded in a report.

The outer loop reads lines from the report, and for each line, outputs any messages that the line contains.

An inner loop scans each line for messages. The loop assumes that a message is a severity tag, followed by a double colon, followed by all characters to the line end or next white space.

An innermost loop scans these remaining characters to the line end or white space.

Iterators “b1” and “e1” are the line begin and end.

Iterators “b2” and “e2” are the remaining search region begin and end. When “b2” and “e1” point to the same place, the line has no further messages.

Iterator “b3” is potentially the next message begin within the search region. When “b3” and “e1” point to the same place, the line has no further messages.



Quick Reference Guide: Some Standard Algorithm Templates for Assigning Values

Template	Description	
<pre>template < class For, class T > void fill (For first, For last, const T& value); template < class For, class Size, class T > void fill_n (Out res, Size n, const T& value);</pre>	Fill all locations with value.	
<pre>template < class For, class Gen > void generate (For first, For last, Gen g); template < class Out, class Size, class Gen > void generate_n (Out res, Size n, Gen g);</pre>	Fill n locations with value.	
<pre>template < class For, class T > void replace (For first, For last, const T& val, const T& new); template < class For, class Pred, class T > void replace_if (For first, For last, Pred p, const T& new);</pre>	Replace all matching elements with new value, using equality operator.	
<pre>template < class In, class Out, class Op > Out transform (In first, In last, Out res, Op op); template < class In1, class In2, class Out, class BinOp > Out transform (In first, In last, Out res, BinOp op);</pre>	Replace all matching elements with new value, using unary predicate.	
In ≡ InputIterator Out ≡ OutputIterator	For ≡ ForwardIterator Gen ≡ Generator	Pred ≡ Predicate (unary) [Bin]Op ≡ [Binary] Operator

330 © Cadence Design Systems, Inc. All rights reserved.

Some standard algorithms assign values to elements.

- The function “**fill**”, fills all locations with the specified value, while the function “fill N” (**fill_n**), fills “N” locations with the specified value;
- The function “**generate**”, fills all locations with the generated value, while the function “generate N” (**generate_n**), fills “N” locations with the generated value;
- The function “**replace**”, replaces all matching elements with the new value, using the equality operator, while the function “replace if” (**replace_if**) replaces all matching elements with the new value, using the unary predicate; and
- The “**transform**” functions perform either a unary operation on one input sequence or a binary operation on two input sequences, and write the result to the output sequence.



Quick Reference Guide: Some Standard Algorithm Templates for Moving Elements

Template	Description
template < class For, class T > For remove (For first, For last, const T& value); template < class For, class Pred > For remove_if (For first, For last, Pred p);	Remove elements, using equality operator.
template < class Bi > void reverse (Bi first, Bi last);	Remove elements, using unary predicate.
template < class For > void rotate (For first, For middle, For last);	Reverse elements.
template < class Ran, class URBG> void shuffle (Ran first, Ran last, URBG&& g) ;	Rotate to place middle first.
template < class T > void swap (T& a, T& b); template < class For1, class For2 > void iter_swap (For1 x, For2 y); template < class For1, class For2 > For2 swap_ranges (For1 first, For1 last, For2 first2);	Shuffle elements in range.
template < class For > For unique (For first, For last); template < class For, class BinPred > For unique (For first, For last, BinPred p);	Swap elements by reference.
	Swap elements by iterator.
	Swap elements in range.
	Move unique elements to front, using equality operator.
	Move unique elements to front, using binary predicate.

331 © Cadence Design Systems, Inc. All rights reserved.

For ≡ ForwardIterator	Ran ≡ RandomIterator	Pred ≡ Predicate (unary)
Bi ≡ BidirectionalIterator	URBG ≡ UniformRandomBitGenerator	BinPred ≡ BinaryPredicate

Some standard algorithms move elements. The container size does not change.

- The function “**remove**”, moves non-matching elements to the sequence front, and returns an iterator to the non-matching elements end, using the equality operator, while the function “remove if” (**remove_if**), does the same thing, using the provided unary predicate. The container size does not change and the back elements become unspecified.
- The function “**reverse**”, reverses the elements within the specified range;
- The function “**rotate**”, rotates the elements within the range, to make the designated middle element first;
- The function “**shuffle**”, randomly permutes the elements within the range. The generator must meet the requirements of a Uniform Random Bit Generator;
- The function “**swap**”, swaps two elements by reference, while the function “iterator swap” (**iter_swap**), swaps two elements by iterator, and the function “swap ranges” (**swap_ranges**) swaps two non-overlapping ranges of elements; and
- The function “**unique**” moves all unique elements to the front of the sequence and returns an iterator to the end of those unique elements, using the equality operator, or the provided binary predicate.

The <random> std::default_random_engine meets the Uniform Random Bit Generator requirements

Example: Standard Algorithm “Remove” Code

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

using container_t = vector<unsigned>;
using iter_t      = container_t::iterator;
// Assume user-defined ostream<<(container_t)

int main() {
    container_t cont; iter_t iter;
    for (unsigned i=0;i<=8;++i) cont.insert(cont.cend(),i);
    cont.insert(cont.cend(),3);
    cout << "Size      : " << cont.size() << endl;           // 10
    cout << "Contents : " << cont << endl;                  // 0123456783
    iter = remove(cont.begin(), cont.end(), 3);                // Remove values 3
    cout << "Size      : " << cont.size() << endl;           // 10
    cout << "Contents : " << cont << endl;                  // 01245678??
    ostream_iterator<unsigned> oiter(cout);
    copy(cont.begin(), iter, oiter); cout << endl;           // 01245678
    return 0;
}
```

332 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “remove”.

The example:

- Declares a container;
- Inserts some values at the container end;
- Removes container elements having value 3. This moves other elements to the container front, and returns an iterator to the end of those elements. Elements at the returned iterator and beyond have a valid unspecified value; and
- Copies the remaining elements to the standard output stream.

Example: Standard Algorithm “Remove with Predicate” Code

```

#include <algorithm>
#include <vector>
#include <functional>
#include <iostream>
using namespace std;

using severity_t = enum {INFO, WARNING, ERROR, FATAL};
using container_t = vector<severity_t>;
using iter_t = container_t::iterator;
using citer_t = container_t::const_iterator;

// unary predicate functor
struct is_error : unary_function<severity_t, bool> {
    bool operator()(const severity_t& severity) const { return severity == ERROR; }
};

int main () {
    container_t cont; iter_t iter; citer_t citer;
    for (unsigned i=0;i<=9;++i) cont.insert(cont.end(),(severity_t)(rand()%4));
    citer = find_if (cont.begin(), cont.cend(), is_error()); // Find first ERROR
    if (citer != cont.cend()) cout << "ERROR was found" << endl;
    else cout << "ERROR not found" << endl;
    iter = remove_if (cont.begin(), cont.end(), is_error()); // Remove all ERROR
    citer = find_if (cont.begin(), iter, is_error());
    if (citer != iter) cout << "ERROR was found" << endl;
    else cout << "ERROR not found" << endl;
    return 0;
}

```

333 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “remove” with a unary predicate.

The example:

- Declares an enumerated message severity type;
- Defines a unary predicate revealing whether a severity has the value “ERROR”;
- Declares a container of the message severity type;
- Inserts some random severity values at the container end;
- Attempts to find the location of the first element having the value “ERROR”;
- Outputs whether the container contains an element having the value “ERROR”;
- Removes all elements having the value “ERROR”; and
- Again outputs whether the container contains an element having the value “ERROR”.

Example: Standard Algorithm “Shuffle” Code

```
#include <algorithm>
#include <vector>
#include <random>
#include <iostream>
using namespace std;

using container_t = vector<unsigned>;
using citer_t     = container_t::const_iterator;
// Assume user-defined ostream<<(container_t)

int main() {
    container_t cont; citer_t citer;
    for (unsigned i=0; i<=9; ++i)
        cont.insert(cont.cend(), i);
    cout << cont << endl;           // 0123456789
    shuffle(vector1.begin(),vector1.end(),random_device());
    cout << cont << endl;           // ??????????
    return 0;
}
```

Shuffle using built-in generator

334 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “shuffle”.

The example:

- Declares a vector of type integer;
- Pushes some values onto the vector back;
- Inserts the vector contents onto the standard output stream;
- Shuffles the vector contents; and
- Again inserts the vector contents onto the standard output stream.



Quick Reference Guide: Some Standard Algorithm Templates for Copying Elements

Template	Description
template < class In, class Out > Out copy (In first, In last, Out res); template < class Bi1, class Bi2 > Bi2 copy_backward (Bi1 first, Bi1 last, Bi2 res);	Copy forward to output. Copy backward to output.
template < class In, class Out, class T > Out remove_copy (In first, In last, Out res, const T& val); template < class In, class Out, class Pred > Out remove_copy_if (In first, In last, Out res, Pred p);	Copy to output while removing matching elements.
template < class In, class Out, class T > Out replace_copy (In first, In last, Out res, const T& val, const T& new); template < class In, class Out, class Pred, class T > Out replace_copy_if (In first, In last, Out res, Pred p, const T& new);	Copy to output while replacing matching elements.
template < class Bi, class Out > Out reverse_copy (Bi first, Bi last, Out res);	Copy to output while reversing element order.
template < class For, class Out > Out rotate_copy (For first, For middle, For last, Out res);	Copy to output while rotating element order.
template < class In, class Out > Out unique_copy (In first, In last, Out res); template < class In, class Out, class BinPred > Out unique_copy (In first, In last, Out res, BinPred p);	Copy to output while discarding adjacent duplicate elements.

In ≡ InputIterator For ≡ ForwardIterator Pred ≡ Predicate (unary)
Out ≡ OutputIterator Bi ≡ BidirectionalIterator BinPred ≡ BinaryPredicate

335 © Cadence Design Systems, Inc. All rights reserved.



Some standard algorithms copy elements to an output iterator. The input sequence does not change. The output sequence cannot overlap the input sequence.

- The function “**copy**”, copies the input sequence in forward order to the output sequence, and returns an iterator to the output sequence end, while the function “copy backward” (**copy_backward**) does the same in reverse order, and returns an iterator to the output sequence begin.
- The function “remove copy” (**remove_copy**), copies the input sequence to the output sequence, discarding matching elements, and returns an iterator to the output sequence end, using the equality operator, while the function “remove copy if” (**remove_copy_if**), does the same, using the provided unary predicate.
- The function “replace copy” (**replace_copy**), copies the input sequence to the output sequence, replacing matching elements with the new value, and returns an iterator to the output sequence end, using the equality operator, while the function “replace copy if” (**replace_copy_if**), does the same, using the provided unary predicate.
- The function “reverse copy” (**reverse_copy**), copies the input sequence to the output sequence, reversing the sequence order, and returns an iterator to the output sequence end.
- The function “rotate copy” (**rotate_copy**), copies the input sequence to the output sequence, rotating the sequence order to place the middle position first, and returns an iterator to the output sequence end.
- The function “unique copy” (**unique_copy**), copies the input sequence to the output sequence, discarding adjacent identical elements, and returns an iterator to the output sequence end, using the equality operator, or the provided binary predicate.



Quick Reference Guide:

Some Standard Algorithm Templates for Sorting Elements

Template	Description
<pre>template < class Ran > void sort (Ran first, Ran last); template < class Ran, class Cmp > void sort (Ran first, Ran last, Cmp cmp);</pre>	Sort (basic) $O(N \log(N))$, using less-than operator. Sort (basic) $O(N \log(N))$, using function object.
<pre>template < class Ran > void stable_sort (Ran first, Ran last); template < class Ran, class Cmp > void stable_sort (Ran first, Ran last, Cmp cmp);</pre>	Sort (stable) $O(N \log(N))$ up to $O(N \log^2(N))$
<pre>template < class Ran > void partial_sort (Ran first, Ran middle, Ran last); template < class Ran, class Cmp > void partial_sort (Ran first, Ran middle, Ran last, Cmp cmp);</pre>	Sort (partial)
<pre>template < class In, class Ran > Ran partial_sort_copy (In first1, In last1, Ran first2, Ran last2); template < class In, class Ran, class Cmp > Ran partial_sort_copy (In first1, In last1, Ran first2, Ran last2, Cmp cmp);</pre>	Sort (partial/copy)
<pre>template< class For > bool is_sorted (For1 first, For2 last); template< class For, class Cmp > bool is_sorted (For1 first, For2 last, Cmp cmp);</pre>	Is sorted sequence?

Some standard algorithms sort a container. Some operations, such as merge operations and set operations, are implemented directly.

The sort algorithms require random-access iterators. Recall that only the vector and double-ended queue containers have random iterators. For a list, you can use the list's sort function.

For all sort algorithms, you can choose between the default less-than operator or your provided comparison function object.

- The function “**sort**”, leaves duplicate elements in an undefined order.
 - The function “stable sort” (**stable_sort**), leaves duplicate elements in input order.
 - The function “partial sort” (**partial_sort**), sorts only the elements between the first and middle iterators, and leaves later elements in an unspecified order.
 - The function “partial sort copy” (**partial_sort_copy**), copies the input sequence to the output sequence, while sorting elements, and returns an iterator to the output sequence end.
 - It sorts and copies only the lesser of the number of input elements or the number of output elements.
 - The function “is sorted” (**is_sorted**), reveals whether the sequence between the two iterators is sorted.

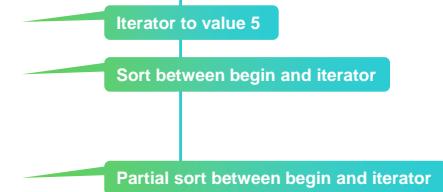
The “sort” and “partial sort” algorithms can both do partial sorts. They may be implemented differently, thus have competing average complexity.

Example: Standard Algorithm “Sort” Code

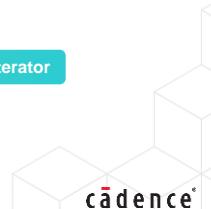
```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

using container_t = vector<unsigned>;
using iter_t      = container_t::iterator;
// Assume user-defined ostream<<(container_t)

int main() {
    container_t cont1, cont2; iter_t iter;
    for (unsigned i=0;i<=9;++i) cont1.insert(cont1.cend(), 9-i);
    cout << cont1 << endl;           // 9876543210
    cont2 = cont1;
    iter = find (cont2.begin(), cont2.end(), 5);
    sort (cont2.begin(), iter);
    cout << cont2 << endl;           // 6789543210
    cont2 = cont1;
    iter = find (cont2.begin(), cont2.end(), 5);
    partial_sort (cont2.begin(), iter, cont2.end());
    cout << cont2 << endl;           // 0123??????
    return 0;
}
```



337 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “sort”.

The example:

- Declares a container;
- Inserts some values at the container end;
- Displays the container contents;
- Partially sorts the container with the “sort” algorithm, and again displays it; and
- Restores the unsorted contents, partially sorts the container with the “partial sort” algorithm, and again displays it.

The “partial sort” algorithm leaves the unsorted elements in an unspecified order.



Quick Reference Guide: Some Standard Algorithm Templates for Merging Sequences

Template	Description
<pre>template < class In1, class In2, class Out > Out merge (In1 first1, In1 last1, In2 first2, In2 last2, Out res); template < class In1, class In2, class Out, class Cmp > Out merge (In1 first1, In1 last1, In2 first2, In2 last2, Out res, Cmp cmp);</pre>	Merge two sorted input sequences to an output sequence.
<pre>template < class Bi > void inplace_merge (Bi first, Bi middle, Bi last); template < class Bi, class Cmp > void inplace_merge (Bi first, Bi middle, Bi last, Cmp cmp);</pre>	Merge in-place two sorted subsequences of an input sequence. <p style="text-align: center;">In ≡ InputIterator Cmp ≡ CompareObject Out ≡ OutputIterator Bi ≡ BidirectionalIterator</p>

338 © Cadence Design Systems, Inc. All rights reserved.



Some standard algorithms merge two ranges.

The merge algorithms merge two pre-sorted ranges, using either the default less-than operator, or your provided comparison function object.

In the output, elements of the first input range appear before matching elements of the second input range.

- The function “**merge**”, merges two pre-sorted input sequences, to an output sequence.
- The function “in-place merge” (**inplace_merge**), merges in-place, two pre-sorted subsequences of one input sequence.

Example: Standard Algorithm “Merge” Code

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

using container_t = vector<unsigned>;
using iter_t      = container_t::iterator;
// Assume user-defined ostream<<(container_t)

int main() {
    container_t cont1, cont2, cont3(10); iter_t iter;
    for (unsigned i=0;i<=4;++i)
        {cont1.insert(cont1.cend(),8-2*i); cont2.insert(cont2.cend(),9-2*i);}
    cout << cont1 << endl; // 86420
    cout << cont2 << endl; // 97531
    sort (cont1.begin(),cont1.end());
    sort (cont2.begin(),cont2.end());
    cout << cont1 << endl; // 02468
    cout << cont2 << endl; // 13579
    (void) merge (cont1.begin(),cont1.end(),cont2.begin(),cont2.end(),cont3.begin());
    cout << cont3 << endl; // 0123456789
    iter = copy (cont1.begin(),cont1.end(),cont3.begin()); //2
    iter = copy_backward (cont2.begin(),cont2.end(),cont3.end()); //3
    cout << cont3 << endl; // 0246813579
    inplace_merge (cont3.begin(), iter, cont3.end()); //4
    cout << cont3 << endl; // 0123456789
    return 0;
}
```

Merge c1 and c2 into c3

Copy c1 forward into c3 at begin

Copy c2 backward into c3 at end

Merge two c3 sections in place

339 © Cadence Design Systems, Inc. All rights reserved.



The example demonstrates use of the standard algorithm “**merge**”.

The example:

- Declares three vectors of type integer;
- Updates the first two vectors’ contents to have descending values;
- Sorts the first two vectors;
- Merges the first two vectors into a third vector;
- Copies the first vector forward into the third vector, starting at its begin;
- Copies the second vector backward into the third vector, starting at its end; and
- In-place merges the two third vector sections.

Quiz: Standard Algorithms



Do standard algorithms resize the container (that is, perform insertion or erasure)?



What capabilities does a random-access iterator have that bidirectional iterators do not?



Can you apply standard algorithms to associative containers?

Please pause here to consider these questions.

Module Summary

In this training module, you

- Stored and manipulated data by using containers and algorithms

This training module discussed:

- Standard Containers
- Standard Almost-Containers
- Standard Algorithms



Your objective is to store and manipulate data by using containers and algorithms.

To help you to achieve your objective, this training module discussed:

- Standard Containers;
- Standard Almost-Containers; and
- Standard Algorithms.



Lab

Lab 17-1 Instantiating and Using Standard Containers

Objective:

- To store and manipulate data by using standard containers and standard algorithms

For this lab, you:

- Determine which container can most appropriately replace the list structure of your object pool
- Modify your pool class to utilize that standard container

342 © Cadence Design Systems, Inc. All rights reserved.

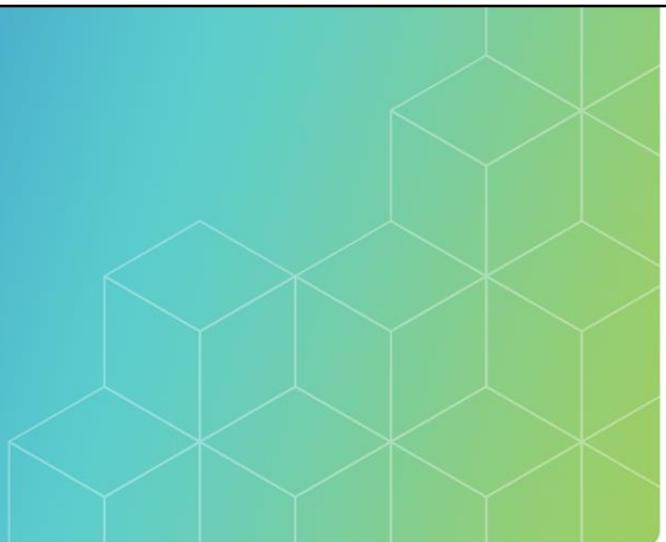
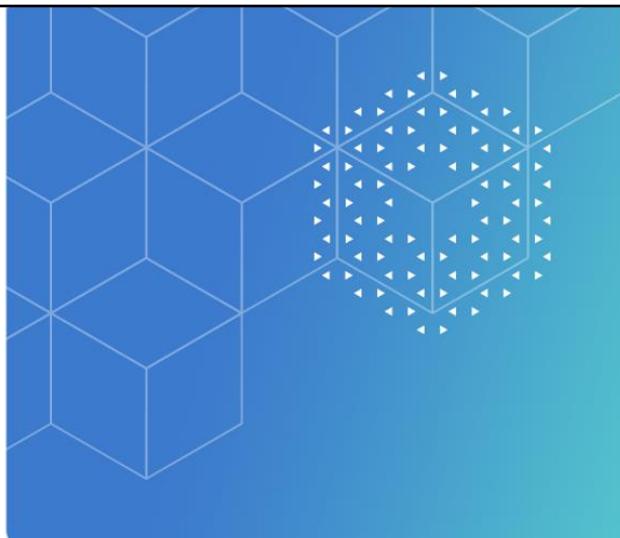


Your objective for this lab is to:

- Store and manipulate data by using standard containers and standard algorithms.

For this lab, you:

- Determine which container can most appropriately replace the list structure of your object pool; and
- Modify your pool class to utilize that standard container.



Module 18

Introduction to System-C

cadence®

This module discusses the basics of System-C language required to put a System-C wrapper on C++ code so as to run the stratus tool.

Module Objectives

In this module, you

- Create a System-C wrapper around C++ code
- Execute the wrapper with accuracy and confidence to demonstrate comprehension of System-C fundamentals

Topics

- What is system-C
- System-C macros and expansions
- System-C hardware modeling constructs
- Data types
- Processes



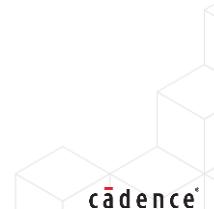
Your overall objective is to use System-C for some combination of hardware or software design or verification. To accomplish this objective requires that you know the System-C language fundamentals and be able to debug your work.

What Is System-C?



System-C is an ANSI standard C++ class library for system and hardware design for designers and architects who need to address complex systems that are a hybrid between hardware and software.

- System-C is a C++ class library:
 - Compiled with any C++ compiler (e.g., GNU gcc).
- Although it is not strictly classified as a hardware description language, it can be utilized in a similar manner.
- Capable of modeling:
 - Hardware and software co-design environments.
 - Operate at multiple levels of abstraction.
- More abstract way of defining systems than HDLs.
- Accellera open source (www.accellera.org)
- Large set of tools available:
 - modeling, debugging, synthesis
- IEEE Standard 1666.



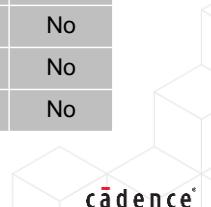
System-C is an ANSI standard C++ class library for system and hardware design for designers and architects who need to address complex systems that are a hybrid between hardware and software. It is used to provide a unified environment usable by design and verification personnel at all levels of abstraction.

System-C supports hardware and software co-design, algorithmic exploration, behavioral modeling and testing, and model refinement to a synthesizable hardware representation. It provides a more abstract way to define systems than HDLs. Accellera, the open-source group, works on defining this language standard. Large sets of tools are available for modeling, debugging, and synthesizing system-C code. In this course, we discuss one such tool called stratus, which is used for high-level synthesis. System-C is defined under the IEEE-1666 standard.

Why Choose System-C?

Combines a High Level of Abstraction with Required Hardware Constructs	Required for Control	System-C	C++	ANSI C
Compatible with algorithm languages		Yes	Yes	Yes
Object-oriented features for managing complexity		Yes	Yes	No
Bit-exact data types		Yes	No	No
Fixed-point data types		Yes	No	No
Explicit concurrency	✓	Yes	No	No
Custom Interfaces – synthesis and simulation	✓	Yes	No	No
Structural hierarchy	✓	Yes	No	No
Synchronous logic and asynchronous logic	✓	Yes	No	No
Same simulation and synthesis semantics	✓	Yes	No	No
Cycle-accurate protocol modeling	✓	Yes	No	No
Multiple levels of abstraction (behavioral + RTL)	✓	Yes	No	No

346 © Cadence Design Systems, Inc. All rights reserved.



The table compares the features of C, C++, and system-C languages. We see that System-C combines high level of abstraction with required hardware constructs.

Fixed Point Data Types



```
sc_fixed<wl, iwl, q_mode, o_mode,
n_bits> Object_Name<...>;
sc_ufixed<wl, iwl, q_mode, o_mode,
n_bits> Object_Name<...>;
```

wl : Total width (total number of bits used).

iwl : Integer word length. Number of bits to *left* of the decimal point.

q_mode : Quantization mode. Represents the process to be done when the value can not be represented precisely.

o_mode : Overflow Processing (SC_WRAP, SC_SAT, ,)

n_bits: is used in overflow processing.

It is possible to simulate fixed point arithmetic using **sc_fixed<>/sc_ufixed<>**.

- The Stratus™ installation includes compatible synthesizable classes.
 - By default, if you #include <cynw_fixed.h> all sc_fixed and sc_ufixed variables will be simulated and synthesized as cynw_fixed and cynw_ufixed(stratus datatypes) instead.

```
sc_fixed<16,8,SC_RND,SC_SAT> value;
sc_in< sc_fixed<32,8> > data;
```



Your digital hardware can represent real numbers in floating format or fixed-point format. Suppose that the real numbers your design will encounter have a known and limited range and precision. Do you really want to embed a full floating-point unit (FPU), or would you rather use a smaller, faster, cheaper, and cooler fixed-point representation?

The System-C fixed-point representation has an integer and a fractional part, which can be signed or unsigned.

For more about the floating-point format you can refer to:

IEEE Std. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic

ANSI/IEEE Std. 854-1987 IEEE Standard for Radix-Independent Floating-Point Arithmetic

IEC 559:1989 Binary floating-point arithmetic for microprocessor systems

Data Types

- System-C provides data types to facilitate digital hardware design.
- It places these declarations in the `sc_dt` namespace.
- The system.h header file makes these types directly visible
- All System-C data types implement the `length()` member method.
- You can use equality and bitwise operators on all System-C data types.
- Some categories do not support some operations
- You can use arithmetic and relational operators on only the numeric types.
- The following pages further describe the use of additional operators.

System-C Data Types	Description
<code>sc_int<N></code>	Integer up to 64 bits
<code>sc_uint<N></code>	Unsigned integer up to 64 bits
<code>sc_bigint<N></code>	Integer w/ any number of bits
<code>sc_bignum<N></code>	Unsigned w/ any number of bits
<code>sc_fixed<N, N></code>	Signed fixed point
<code>sc_ufixed<N, N></code>	Unsigned fixed point
<code>sc_lv,</code> <code>sc_lv(vector)</code>	bit (2-state), logic (4-state)
<code>sc_logic (scalar)</code>	bit (2-state), logic (4-state)

Stratus Specific Data Types	Description
<code>cynw_cm_float</code>	Flexible floating-point
<code>cynw_fixed</code>	Compatible with <code>sc_fixed</code>
<code>cynw_ufixed</code>	Compatible with <code>sc_ufixed</code>

System-C provides specific data types to facilitate digital hardware design.

It places these declarations in the System-C data type (`sc_dt`) namespace.

The SystemC.h header file contains using directives, making the types directly visible. All System-C data types implement the `length()` member method. You can use equality and bitwise operators on all System-C data types.

System-C categorizes the data types as numeric, vector, and scalar. Some categories do not support some operations. For example, only the numeric types support the arithmetic and relational operators.

All System-C data types support the equality, bitwise operators, and `length()` member method.

Notably missing from this table is a scalar 2-state type. The IEEE Std. 1666 System-C deprecates the System-C bit (`sc_bit`) type. For future work, you should use the `bool` type instead of the System-C bit type. These training materials do not further discuss the System-C bit type. The table at the bottom right lists a few Stratus-specific data types.

Hardware Operators Supported for sc_int/sc_uint

Category	Operator					
C++ operators	All the usual C++ operators					
Bit Select	[x]					
Part Select	.range(left, right)					
Concatenation	(,)					
Conversion to C++ types	to_int	to_long	to_int64	to_uint	to_uint64	to_ulong



The table lists the set of hardware operators supported for sc_int and sc_uint template classes. It supports all the C++ operators, bit select, part select, concatenation, and conversion to C++ type operators.

How to Model System-C Hardware Constructs



These are different steps required to build a simulation model.

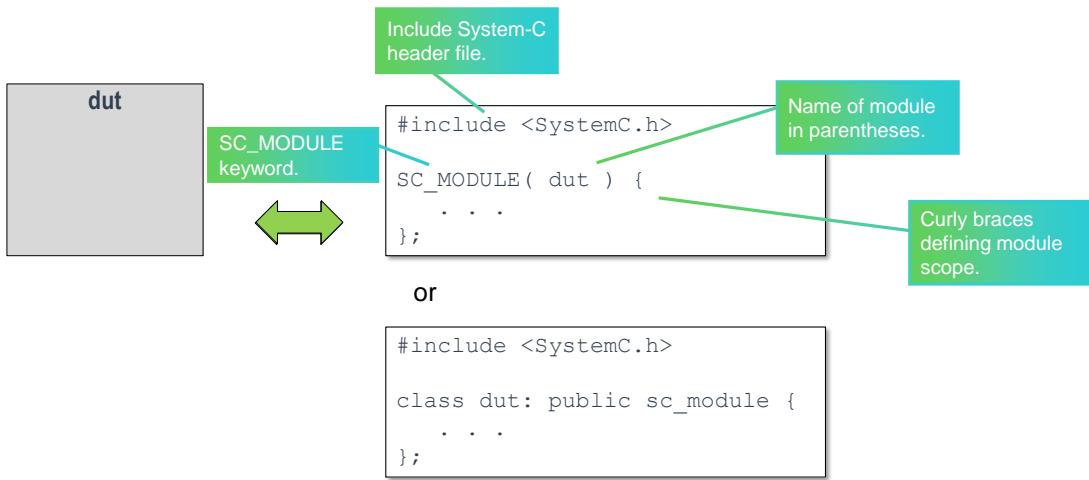
- 1 Declaring a module
- 2 Declaring Clock and Reset input ports
- 3 Declaring Data input ports
- 4 Declaring Data output ports
- 5 Performing Read/Write through i/o ports
- 6 Declaring System-C constructors

In this module, we will discuss different steps required to build a simulation model. Lets get started.

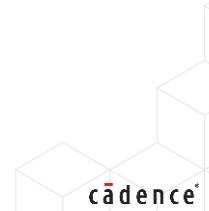
Step 1: Declaring a Module

System-C has classes that describe hardware.

- Let's start with modules.



351 © Cadence Design Systems, Inc. All rights reserved.



The module is the basic building block of your simulation model.

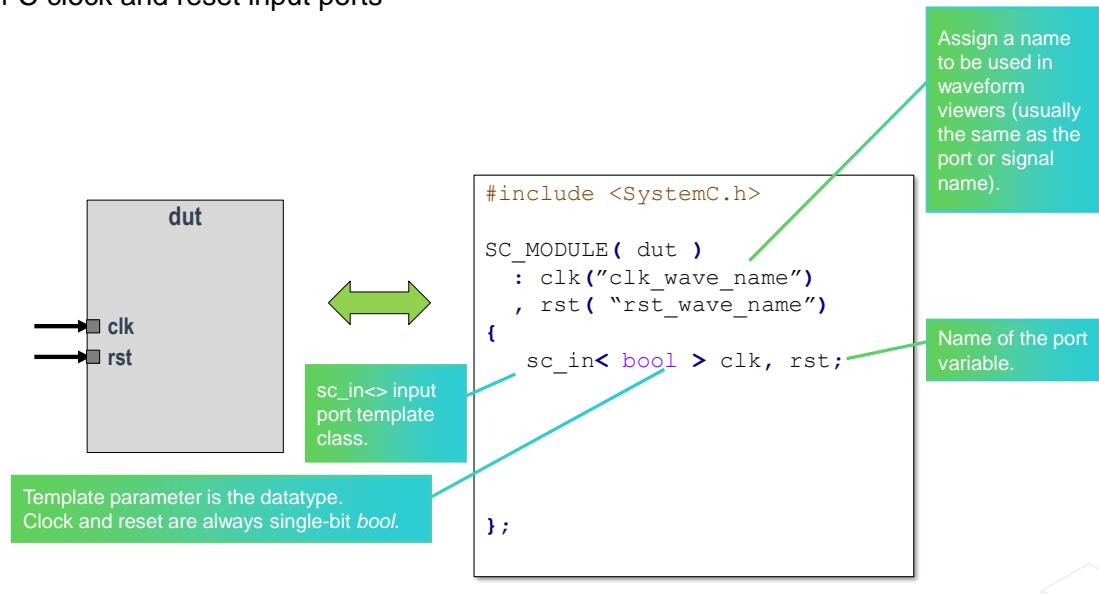
A module may contain, that is, instantiate, other System-C objects, such as processes, ports, channels, and other modules.

A module is a System-C object, so it inherits the attributes and behaviors common to all System-C objects.

`SC_MODULE` is the keyword to declare the module. The slide shows the syntax to declare a System-C module.

Step 2: Declaring Clock and Reset ports

System-C clock and reset input ports



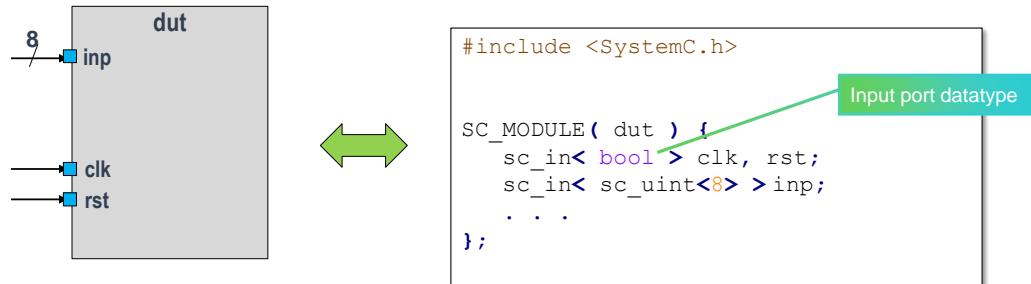
352 © Cadence Design Systems, Inc. All rights reserved.



sc_in is the pre-specialized input port template class. You can pass the input data type as the template parameter to the sc_in class. This is followed by the input port variable names. We can also assign names to the inputs to be used in waveform viewers. It should be generally matching with the port or signal name.

Step 3: Declaring Data Input Ports

System-C data input ports



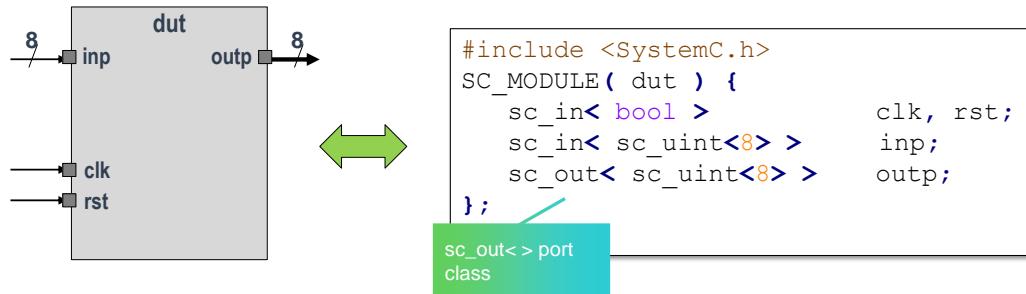
353 © Cadence Design Systems, Inc. All rights reserved.



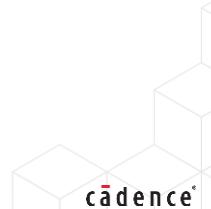
We can similarly declare all other inputs using the `sc_in` input port template class, providing the input data type as a template parameter.

Step 4: Declaring Data Output Ports

System-C data output ports



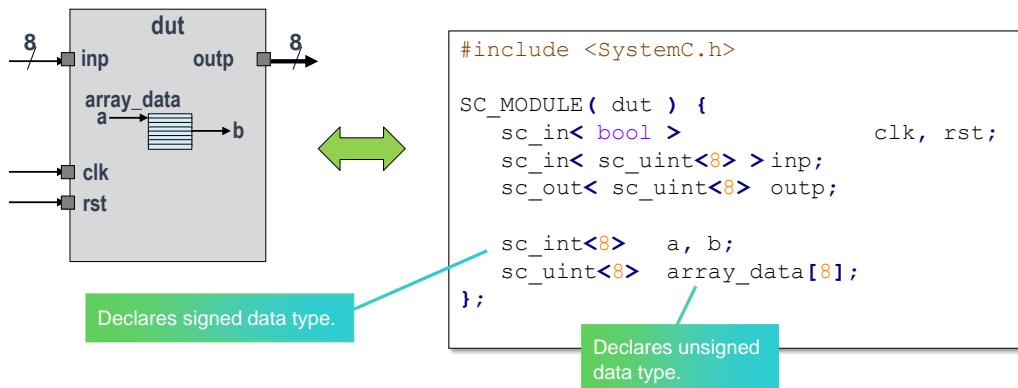
354 © Cadence Design Systems, Inc. All rights reserved.



Like `sc_in`, `sc_out` is the output port template class. We declare all the output using this class. The datatype of the output is passed as the template parameter.

Step 4: Declaring Data Output Ports (continued)

Signed and Unsigned Integer Types



355 © Cadence Design Systems, Inc. All rights reserved.



We also have template classes defined for different datatypes. For example, the `sc_int` and `sc_uint` template classes declare signed and unsigned integer types, respectively.

Step 4: Declaring Data Output Ports (continued)

Signed and Unsigned Integer Types

Most of the time, you will use bit-accurate integers.

- Unsigned sc_uint<N>

```
sc_uint<3> x;
```

Template parameter is the variable's bitwidth.

X	Value
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

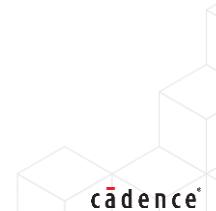
- Signed sc_int<N>

For signed types, width template parameter includes the sign bit.

```
sc_int<3> x;
```

X	Value
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	-4
1 0 1	-3
1 1 0	-2
1 1 1	-1

356 © Cadence Design Systems, Inc. All rights reserved.

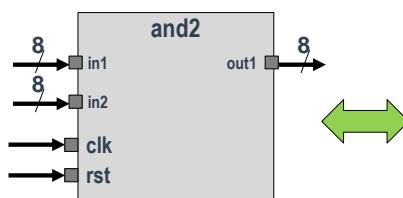


In the unsigned and signed integer template class, the template parameter is the variable's bit width. For signed types, the width template parameter includes the sign bit.

Step 5: Performing Read/Write Through I/O Ports

How do you read and write ports in System-C?

- Use the read() member to read an sc_in<> port.
- Use the write() member to write a sc_out<> port.



```
#include <SystemC.h>
SC_MODULE( and2 ) {
    sc_in< sc_uint<8> > in1, in2;
    sc_out< sc_uint<8> > out1;
    sc_uint<8> val;
public:
    void func() {
        val = in1.read() & in2.read();
        out1.write( val );
    }
    SC_CTOR( and2 ) {
    }
};
```

Input ports are read with .read() member function of sc_in<>.

Output ports are written with .write() member function of sc_out<>.

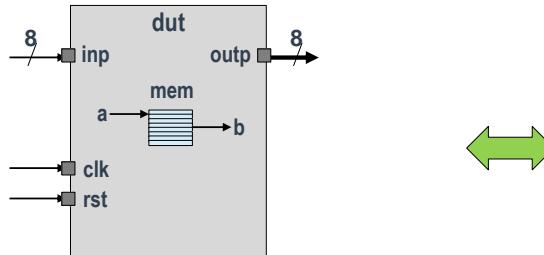


In the example on slide, in1 and in2 are two inputs to be read and processed to get the output. We use the .read() member function of sc_int<> to read the input ports. The output is written onto the output port using the .write member function of sc_out.

Step 6: Declaring Module Constructors

System-C constructors (SC_CTOR)

Note: Constructors are where you deal with the clock and reset the behavior of threads.



```
...
MyMod ( sc_module_name name, double frequency,
         unsigned width, unsigned depth);
...
```

```
#include <SystemC.h>
SC_MODULE( dut ) {
    sc_in < bool > clk, rst;
    sc_in < sc_uint<8> > inp;
    sc_out< sc_uint<8> > outp;
    sc_int<8> a, b;
    sc_uint<8> mem[8];
    SC_CTOR( dut ) {
    }
};
```

SC_CTOR is the keyword for a module constructor.

Constructor has the same name as the module, just like C++ classes.

358 © Cadence Design Systems, Inc. All rights reserved.



SC_CTOR is the keyword for a module constructor. The constructor of every module must contain at least the first System-C module name (sc_module_name) parameter. This parameter accepts the instance name upon instantiation. The SC_CTOR macro automatically includes this parameter. If you elect to declare the constructor yourself, you can add additional parameters to parameterize your module on a per-instance basis.

Here is an example of a constructor that accepts additional arguments, in this case, for width, depth, and frequency. Note that the System-C module name parameter is first. This is a good convention to follow, as it will never have a default value, and other parameters may very likely have default values.

Macros SC_CTOR and SC_HAS_PROCESS

- The SC_CTOR macro takes one `sc_module_name` parameter.
- What if you need to parameterize your module (e.g., width, depth)?
- If you write your own constructor and you have processes, you must insert SC_HAS_PROCESS.

```
#define SC_HAS_PROCESS ( user_module_name ) \
|   typedef user_module_name SC_CURRENT_USER_MODULE |
```

```
SC_MODULE (mux)
{
    sc_in <bool> sel, ina, inb;
    sc_out <bool> out;
    SC_CTOR (mux)
    {
        SC_METHOD (mux_method);
        sensitive<<sel<<ina<<inb;
    }
    void mux_method()
    {
        out = sel ? inb : ina;
    }
};
```



```
SC_MODULE (mux) {
    sc_in <bool> sel, ina, inb;
    sc_out <bool> out;
    SC_HAS_PROCESS (mux);
    mux(sc_module_name nm, ...): sc_module(nm)
    {
        SC_METHOD (mux_method);
        sensitive<<sel<<ina<<inb;
    }
    void mux_method()
    {
        out = sel ? inb : ina;
    };
}
```

The SC_CTOR macro takes one parameter, the System-C module name (`sc_module_name`).

What if you need to parameterize your module, for example, for memory width and depth?

In this case, you cannot use SC_CTOR and need to write your own constructor.

If you cannot use the SC_CTOR macro for your module and your module has processes, you must use the SC_HAS_PROCESS macro to set the current user module for the process registration macros.

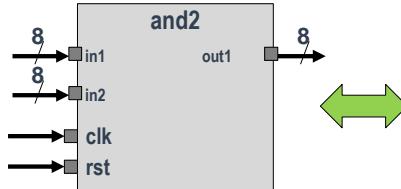
The SC_HAS_PROCESS macro takes the same single argument as the SC_CTOR macro. If you use the SC_HAS_PROCESS macro, you can define the module constructor with whatever additional arguments you need. By convention, the first argument should still be the System-C module name.

What Are System-C Processes?



System-C processes describe the actual functionality of the module.

- System-C processes are implemented as:
 - Member functions of a module.
- They can be made sensitive to events like in RTL.



```
#include <SystemC.h>
SC_MODULE( and2 ) {
    sc_in< sc_uint<8>> in1, in2;
    sc_out< sc_uint<8>> out1;

    sc_uint<8> val;
    void func() {
        val = in1.read() & in2.read();
        out1.write( val );
    }

    SC_CTOR( and2 ) {
        //what goes in here?
    }
};
```

func() is a thread function that ANDs the two input ports and writes the result to the output port, out1.

360 © Cadence Design Systems, Inc. All rights reserved.



A process defines some or all of the behavior of your simulation model.

A process is a module method that you register as a process. That means that you tell the System-C kernel to treat the method as a process, and you can optionally specify what events trigger the process execution. The kernel schedules process execution in response to any of those events.

A process is a System-C object, so it inherits the attributes and behaviors common to all System-C objects.

These processes can be made sensitive to any events like in RTL. In the example, the module and2 has the member function, func as the thread function which reads and ands the two input ports, in1, and in2, and writes the result to the output port,out1.

System-C Processes (continued)

So, how do processes model hardware?

- They run concurrently.
 - Not serially like functions in regular C or C++.
- They can be made sensitive to:
 - Signals, clock edges, or fixed amounts of simulation time.
- They are not called by the user.
 - They are always active based on how you define them in a SC_CTOR constructor.

System-C has three different process types:

- SC_CTHREAD
- SC_METHOD
- SC_THREAD

SC_CTHREADs are what you primarily use in HLS.

SC_METHODs are occasionally useful as a helper processes.

SC_THREAD is more flexible than SC_CTHREAD. It's primarily used in modeling, not HLS.

The processes are used to model hardware as they run concurrently, not serially, as in regular C or C++. They can be made sensitive to signals, clock edges, or for a fixed amount of simulation time, as in real hardware. They are never called by the user but based on how we define them in the constructor.

System-C has three different macros to register processes: SC_THREADS, which we mainly use in HLS; SC_METHODs, which is occasionally useful as helper processes; and SC_THREAD, which is more flexible than SC_CTHREAD. It is mainly used in modeling, not in HLS.

System-C Processes: SC_METHOD

- Runs continuously.
 - Normally, it first executes during the initialization phase.
 - If `dont_initialize()`, then first executes when first triggered.
 - Thereafter executes from beginning to end upon each trigger.
- Cannot suspend for any reason.
 - Cannot call `wait()` directly or indirectly.
- Uses the high-performance *method* process to model.
 - Hardware that only reacts to transitions of its inputs/external events, such as a clock.
- Synthesizable.
 - Useful for combinational logic or simple sequential logic.
- Analogous to a Verilog `@always` block.

Use sensitive on the line directly after to describe the thread sensitivity, listing signals with the “<<” operator.

```
SC_MODULE ( dff ) {
    // Declarations
    sc_out <bool> q;
    sc_in <bool> c,d,r;
    void dff_method() {
        q = r ? 0 : d ;
    }
    // Constructor
    SC_CTOR ( dff ) {
        // Process registration
        SC_METHOD(dff_method);
        sensitive << c.pos();
        dont_initialize();
    } };
```

Use `clk.pos()` to make `SC_METHOD` rising edge sensitive.


```
SC_MODULE ( dut ) {
    ...
    void func();
    SC_CTOR( dut ) {
        SC_METHOD( func );
        sensitive << in1 << in2;
    } };
```

To make `func()` an `SC_METHOD`, list it as the argument of `SC_METHOD` in the `SC_CTOR`.

`SC_METHOD` executes every sensitive event once in its entirety. It runs continuously throughout program execution. A process you register with the `SC_METHOD` macro does *not* have its own thread of execution. The kernel executes the process method in its own thread. That means the process cannot be suspended; it cannot “consume time.” This is similar to the function construct in hardware description languages.

You would likely use a method process to represent cycle-based hardware- hardware that reacts to external events, such as a clock, and does not internally generate events. It is synthesizable and useful to generate combinational logic or simple sequential logic. It is analogous to Verilog always block. In the example on the slide, upon every occurrence of a positive edge of the clock input port, the kernel calls the method process, `dff_method`, to update the “`q`” output port. The `dff_method` assigns the output port, `q`, with 0 or input, `d`, depending on the reset value, `r`. Consider another example: method `func` is registered using the `SC_METHOD` macro and made sensitive to the changes in the input signals. The method process is the computationally least expensive process, so you may choose it wherever it makes sense to do so.

System-C Processes: SC_THREAD

- Use the flexible *thread* process.
 - To model system or testbench behavior.
- Expensive. Hence, use only where needed.
- Can suspend for any reason.
 - Can wait for a change in the sensitivity list, such as a clock edge.
 - Can call `wait()` directly or indirectly.
 - Can wait for a time interval.
 - Can wait for an abstract `sc_event`.
- Normally, it first executes during the initialization phase.
 - If `don't_initialize()`, then first executes when first triggered.
- Thereafter (when triggered) executes.
 - From the statement after suspension to the next suspension.
- Can specify any number of resets.
 - New feature with IEEE 1666-2011.
 - `reset_signal_is(signal,bool)`;
 - `async_reset_signal_is(signal,bool)`;
 - Restarts process.
- Synthesizable.
 - Only when sensitive to a clock edge (like `SC_THREAD`).
 - AND if there's a reset specified.

```
SC_MODULE ( dff ) {
    // Declarations
    sc_out <bool> q;
    sc_in <bool> c,d,r;
    void dff_thread() {
        q = 0;
        wait();
        while (true) {
            q = d ;
            wait();
        }
    }
    // Constructor
    SC_CTOR ( dff ) {
        // Process registration
        SC_THREAD(dff_thread);
        sensitive << c.pos();
        reset_signal_is(r,1);
        dont_initialize();
    }
}
```

To make `dff_thread` a `SC_THREAD`, list it as the argument of `SC_THREAD` in the `SC_CTOR`.

Use `clk.pos()` to make `SC_METHOD` rising edge sensitive.

363 © Cadence Design Systems, Inc. All rights reserved.



You would most likely use a thread process to represent abstract hardware or a testbench. The thread process is the computationally most expensive process, so you may want to choose it only where you need its capabilities.

A process you register with the `SC_THREAD` macro *does* have its own thread of execution. That means the process can suspend, which can “consume time”. This is similar to the procedure or task construct in hardware description languages. It can suspend itself for any reason, i.e., it can wait for a change in the sensitivity list, or wait for `sc_event`, or wait for a time interval.

A thread process typically has an initialization part followed by a loop. The initialization part executes once during the initialization phase. The loop completes, suspends, and then resumes execution when triggered again.

The IEEE Std. 1666-2011 added the capability to provide a thread process with one or more reset inputs and levels. The reset forces the process to restart.

Here, upon the first occurrence of a positive edge of the clock input port, the kernel schedules the thread process to initialize the “`q`” output port, and, after that, upon every occurrence of the positive edge, schedules the thread process to *resume* from its suspended state. It is synthesizable only when it is sensitive to the clock edge and a reset is specified. In the code on the slide, we register the method `dff_thread` as a thread process in the constructor. The method calls the `wait` statement. The process is made sensitive to the positive edge of the clock, and an active high reset is declared.

System-C Processes: SC_CTHREAD

Use the *clocked thread* process to behaviorally model clocked hardware.

Register a clocked thread process with the SC_CTHREAD macro.

- Sensitive to a single event.
 - Second macro argument.
 - Ignores static sensitivity.
- Can specify a synchronous reset signal (and level).
 - To restart the process.
 - IEEE 1666-2011 adds.
 - Async reset and multiple resets.
- Does not initialize.
 - First executes when triggered.
 - Implies dont_initialize().
- Synthesizable.
 - Can take one or more clock cycles for a single iteration.
 - This is what you will use 99% of the time in Stratus HLS.

Process function can be a large algorithm to be scheduled over multiple clock cycles.

SC_CTHREAD parameters are process name and clock edge.

reset_signal_is() goes online after SC_CTHREAD to describe the reset signal and assertion level.

```
SC_MODULE (dff) {
  // Declarations
  sc_out <bool> q;
  sc_in <bool> c,d,r;
  void dff_cthread() {
    q = 0;
    wait();
    while (true) {
      q = d;
      wait();
    }
  }
  // Constructor
  SC_CTOR (dff) {
    // Process registration
    SC_CTHREAD(dff_cthread,c.pos());
    reset_signal_is(r,1);
    async_reset_signal_is(arst,
    false);
  }
};
```

A process you register with the SC_CTHREAD macro is conceptually a stripped-down version of a thread process. We use the *clocked thread* process to model clocked hardware behaviorally.

You make a clocked thread process sensitive to a single event, as the System-C event finder (sc_event_finder) references the second macro argument. You cannot later change the event to which the process is sensitive. Optionally provide a clocked thread process with a reset input and level. This acts as a synchronous reset, which, when active during the clock event, restarts the process. The IEEE Std. 1666-2011 added asynchronous reset and multiple resets.

The kernel does not execute clocked thread processes at simulation time 0. It schedules clocked thread processes only when triggered. Accordingly, this example initializes when first clocked, regardless of the state of the reset input. This construct is synthesizable. It may take one or more clock cycles for a single iteration. This is the construct we mostly use in Stratus.

Using Templates for SC_MODULEs

Stratus HLS supports using templates to specify SC_MODULEs.

- Useful for parameterizing the module.
- Arguments like datatype (class T) and array size (int LEN) can be passed.

```
template <class T, int LEN = 4>
SC_MODULE(shift) {
    sc_in<bool> clk;
    sc_in<bool> reset;
    sc_in<T> din;
    sc_out<T> dout;
}

// Array
T sr[LEN];

SC_CTOR(shift) {
    SC_METHOD(method0);
    sensitive_pos << clk;
}
void method0();
};
```

```
void shift::method0() {
    if( !reset.read() )
    {

        for(int i = 0; i < LEN; i++)
            sr[i] = 0;
    }
    else
    {
        dout.write( sr[LEN-1] );
        for(int i = LEN-1; i>0; i--)
            sr[i] = sr[i-1];
        sr[0] = din.read();
    }
}
```

Stratus HLS supports using templates to specify SC_MODULEs. This makes the SC_MODULE more generic. It is useful for parameterizing the module. Arguments like datatype (class T) and array size (int LEN) can be passed. In the example on the slide, we see that we have parameterized the module shift to receive different datatypes and lengths for the array on which it performs the shift operation.

System-C File Structure

Module declaration goes in the header file.

Design file includes the header file.

Use the double-colon “::” scoping operator to specify that thread func() belongs to module dut.

```
#include <SystemC.h>
SC_MODULE( dut ) {
    sc_in< sc_uint<8> >    in1, in2;
    sc_out< sc_uint<8> >   out1;
    sc_uint<8> val;
    void func();
    SC_CTOR( dut ) {
        SC_METHOD( func );
        sensitive << clk.pos();
    }
};
```

dut.h

```
#include "dut.h"

void dut::func() {
    val = in1.read() &
    in2.read();
    out1.write( val );
}
```

dut.cc

*.cpp extension can also be used.

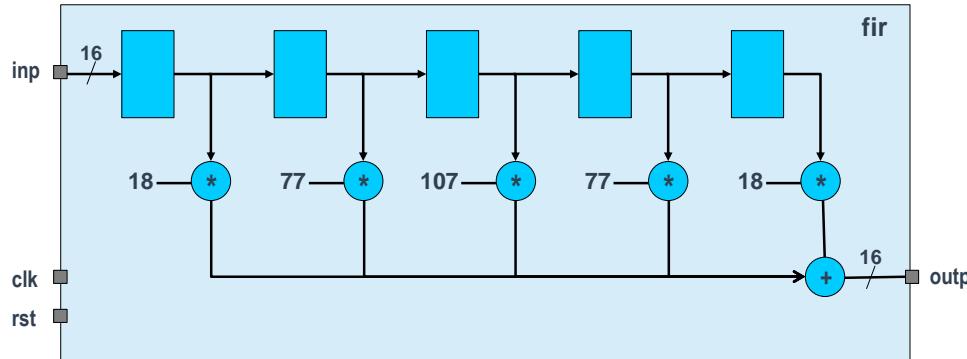
The header file only has thread declaration. The source file has thread function definition.



The diagram on the slide depicts a typical System-C file structure. It has two files: the header file with *.h extension and the design file with *.cc/* .cpp extension. All the module declarations go into the header file. The design file includes the header file. Note that the header file only has the thread declaration, and the source file has the thread definition. We use the scope resolution operator(::) along with the function to specify which module the process belongs to.

Example: System-C Clocked Threads

Let us implement a small 5-tap FIR filter using the System-C clocked threads.



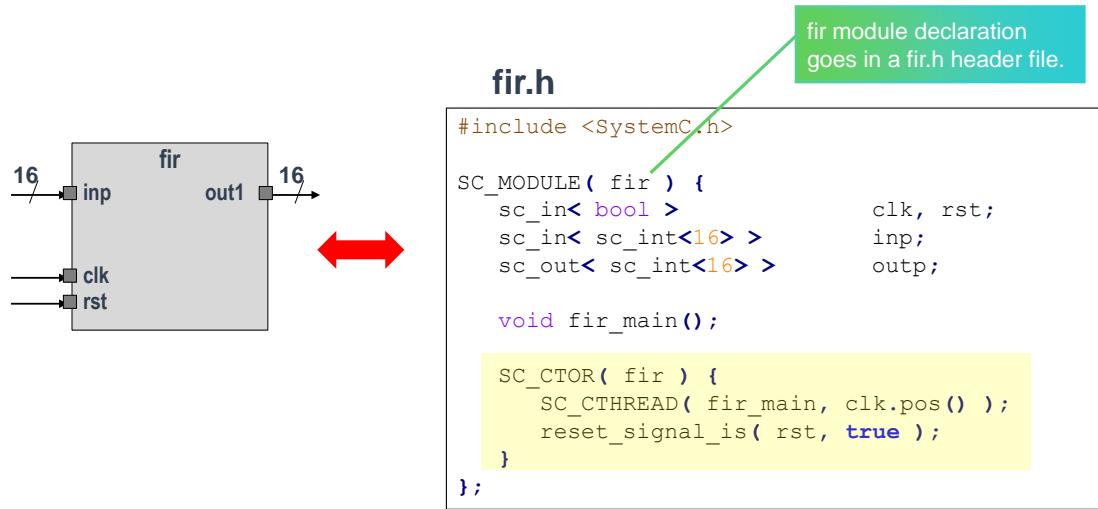
367 © Cadence Design Systems, Inc. All rights reserved.



Let us try to implement a simple FIR filter using the System-C clocked threads. We see that the fir filter has clk, reset, single input, and output. The implementation just shifts the input, multiplying the shifted values with certain coefficients and adding them to produce the filtered output.

Example: System-C Clocked Threads (continued)

fir module declaration in the header file.



368 © Cadence Design Systems, Inc. All rights reserved.



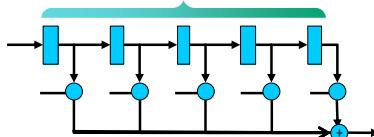
We declare the module fir with its input, output, clock, and reset along with the process fir_main in the header file. We also register this process as an SC_THREAD in the constructor SCCTOR.

Example: System-C Clocked Threads (continued)

Thread definition in design file.

coefs[] is a constant array with the 5 tap multiplier values.

taps[] is a variable array to represent the shift register of tap values.



Load first tap with the value read on the input port.

Multiply each tap by its coefficient and accumulate on variable val.

Wait one cycle and repeat while loop.

```
#include "fir.h"                                     fir.cc

const sc_uint<7> coefs[5] = {18,77,107,77,18};

void fir::fir_main() {
    sc_int<16> taps[5];
    outp.write( 0 );
    wait();

    while( true ) {
        for( int i = 4; i > 0; i-- ) {
            taps[i] = taps[i - 1];
        }
        taps[0] = inp.read();
        sc_int<16> val = 0;
        for( int i = 0; i < 5; i++ ) {
            val += coefs[i] * taps[i];
        }
        outp.write( val );
        wait();
    }
}
```

Reset output port and wait one cycle.

Shift tap values.

Write an accumulated value to output.

We put the thread definition in the design file, fir.cc. We declare the coefficients as a constant array with five tap multiplier values. Taps[] is a variable array to represent the shift register of tap values. Initially, we reset the output and wait for a cycle. In the while loop, we load the first tap with the value read on the input port and shift the tap values. We multiply each tap by its co-efficient and accumulate on a variable, val. We write the val value onto the output. We next wait for a cycle and repeat the same process, which continues forever.

Example: System-C Clocked Threads (continued)**Resetting a Clocked Thread**

This is all handled by the reset_signal_is() back in the SC_CTOR.

Why is there no check of the clk or rst ports here?

Unlike a Verilog @always block, no IF statement is needed to test the value of the rst port.

In a clocked thread, the reset behavior is everything from the start of the thread function to the first wait().

The while(true) loop will repeat indefinitely until rst port asserts. Then, the thread starts over.

This is all handled by the reset_signal_is() back in the SC_CTOR.

```
#include "fir.h"
const sc_uint<7> coefs[5] =
{18,77,107,77,18};

void fir::fir_main() {
    sc_int<16> taps[5];

    outp.write( 0 );
    wait();

    while( true ) {
        for( int i = 4; i > 0; i-- ) {
            taps[i] = taps[i - 1];
        }
        taps[0] = inp.read();

        sc_int<16> val = 0;
        for( int i = 0; i < 5; i++ ) {
            val += coefs[i] * taps[i];
        }
        outp.write( val );
        wait();
    }
}
```

fir.cc

We don't check clk or rst ports in the implementation/definition of the process in the design file, as the declaration in the constructor takes care of it. Unlike Verilog, we don't need @always block to check the reset. In a clocked thread, the reset behavior is everything from the start of the thread function to the first wait(). The while loop will loop indefinitely until the reset port asserts. All this is handled by the reset signal declared in the constructor.



Lab

Lab 18-1 Implementing FIR Filter in C++ and Running Stratus Tool to Generate the RTL

- Access a down-counter function whose name is statically bound
- Access a down-counter function whose name is dynamically bound
- Down-cast a base class pointer or reference to enable using it to access a derived class incremental member

371 © Cadence Design Systems, Inc. All rights reserved.

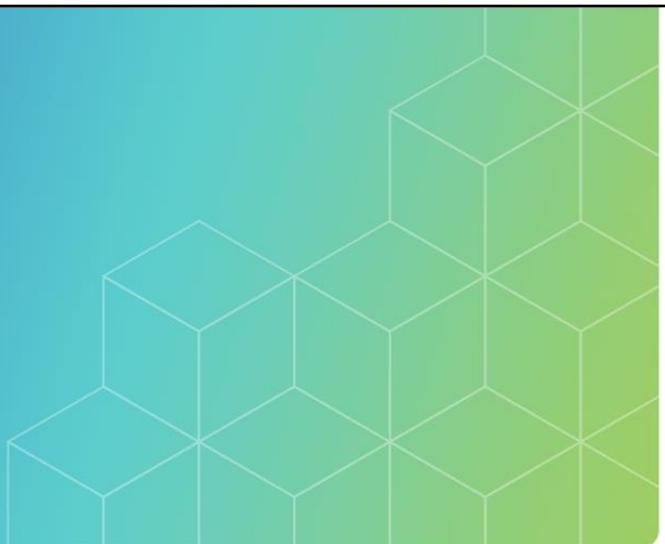
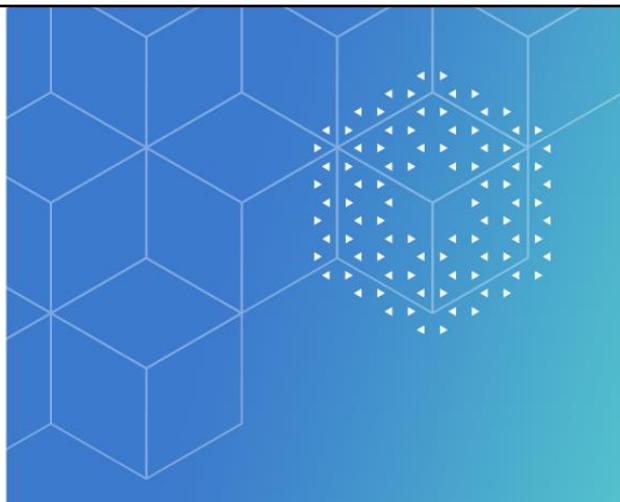


The objective for this lab includes:

- To implement FIR filter using C++.
- Create a System-C wrapper around C++ code, and
- To perform high-level synthesis of simple FIR filter implemented in C++ using the Stratus tool.

For this lab, you:

- Access a down-counter function whose name is statically bound.
- Access a down-counter function whose name is dynamically bound.
- Down-cast a base class pointer or reference to enable using it to access a derived class incremental member.



Module 19

Equivalence Checking C++ for Verification

cadence®

This training module discusses the verification of a C++ model against various other implementations by using formal equivalence checking.

Module Objectives

By the end of this module, you will be able to

- Define datapath verification and identify the challenges associated with it
- Draw a diagram that demonstrates the verification of C++ equivalence against RTL
- Explain the process of setting up, running, and debugging a C++ to RTL equivalence check using Jasper™
- Utilize the Jasper C2RTL tool to verify the equivalence of a pipelined multiplier



This slide shows the module's objectives, which are to show the motivation for and the details of how to perform a C++ to RTL equivalence check.

Why Is Datapath Verification Hard?



Formal → conceptually ideal given input combinations, but historically intractable

Simulation → infeasible to explore all input combinations, but often, the only solution

Intel's infamous **Pentium FP Division bug (1994)**.

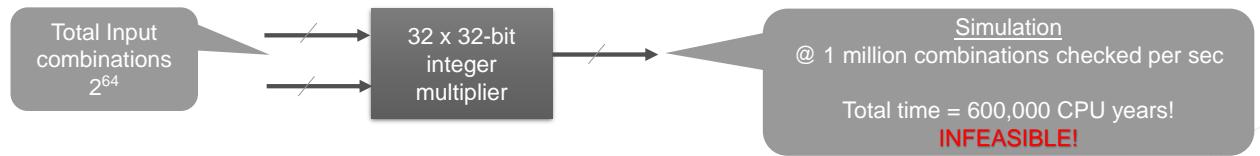
- Corner case: 1 in 9 billion random simulations would produce an inaccurate result.
- Intel recalled faulty processors → cost Intel \$475 million. (source: https://en.wikipedia.org/wiki/Pentium_FDIV_bug)

What's new?

- New Solvers and Techniques are making it possible to mathematically verify datapath designs.

Jasper Datapath engines – mathematically prove the correctness of all cases implicitly.

- Takes seconds to prove several integer multiplier implementations.
- Takes minutes to a few hours for Floating-point multipliers.



374 © Cadence Design Systems, Inc. All rights reserved.

cadence®

As far as data path verification goes, it is definitely not easy. Let's take an example of a small 32 cross 32-bit integer multiplier. The total input combinations that we can have for this input multiplier for this integer multiplier is about 2 to the power of 64. If operant a and operant b are 32 bits each. So, it means that we have to try two to three powers of 64 input combinations to be exhaustive. Assuming we could do one million combinations of checking per second, We would still require 600,000 CPUs. It is definitely invisible, and you don't need to check for all possible values. But again, then, which values do we check? Well, it's a difficult question. Remember the infamous Pentiumberg of Intel, which happened in 1994? It was a corner case bug because of a missing row in our lookup table. Estimates say it would take almost nine hundred billion random submissions to catch this corner's scenario once. This bug cost almost four seventy-five million dollars in 1994. This means which values to check for is not a simple question to answer. Ideally, one would want to do exhaustive verification and cover all possible scenarios. With the emergence of AI and the building of many GPUs, EDA companies have invested more in this domain, which has led to new solvers and techniques in the forward domain, making it possible to verify these data path designs mathematically.

Today, we have a particular data path instance in Jasper™, which can prove the mathematical correctness of many arithmetic circuits. Data path verification refers to blocks that do arithmetic transformations with some control logic. Particular data path engines take seconds to prove several teachers' multiple implementations. Even floating-point modifications are typically more complex than your integer multiplier implementations. So, we can prove those in a matter of minutes to a few hours, depending on the implementation.

Datapath Problem Space and Required Solution



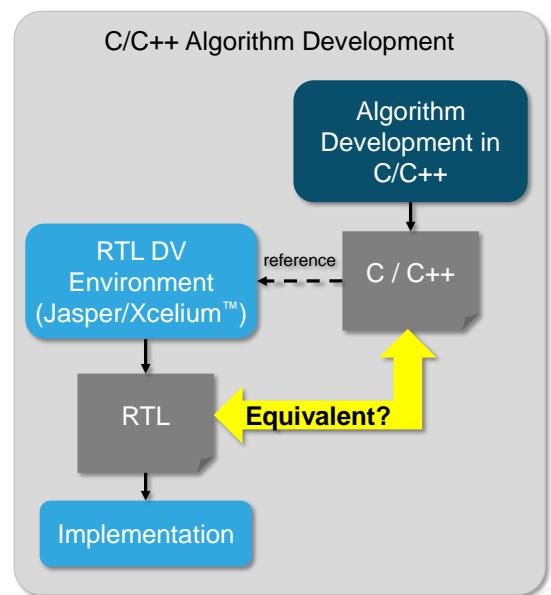
How to handle both control and data paths together?

Most datapath algorithms were first developed at a high level in C++.

- RTL designers then use C++ models as a reference while implementing RTL.

A formal tool that can verify RTL to be functionally equivalent to the reference C++ model needs:

- C++ language compiler and C++ standard library support.
- Datapath-specific formal solvers.
- Specialized debugging for C vs. RTL equivalence checking context.
- Verify control + datapath logic interactions.
- Advanced proof management.



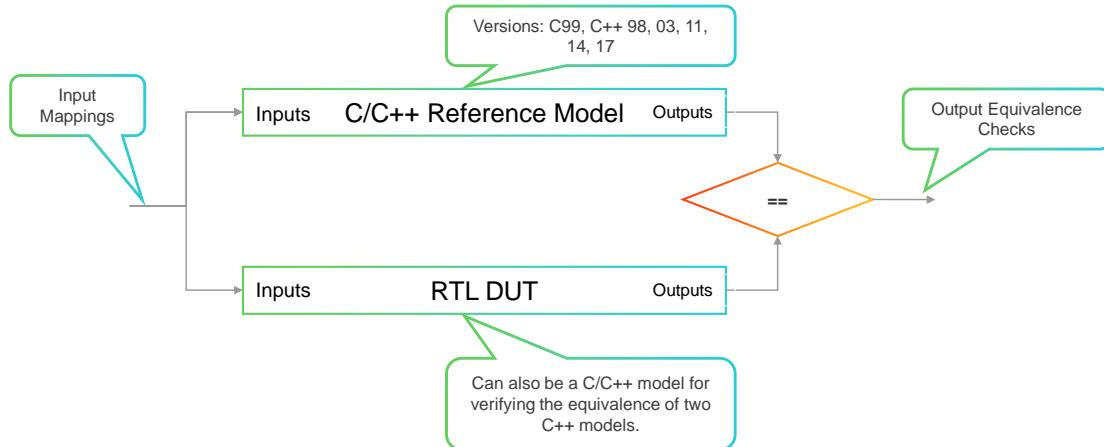
375 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Look at the terrapad domain as a whole. Typically, most terrapad algorithms are first developed in C++, and then these C++ models are taken as reference models by the RTL designers while they are implementing the RTL. Today, most of the verification is centered around submission, where you have to create test benches, and then you go about verifying this implemented idea. Ideally, it would be good to have a formal tool that can mathematically prove the function requirements of this implemented RT against the golden CC++ reference model. But to have such a tool that can consume a C++ model in the Huddl and do it means checking. We would need a tool to support the C++ language, especially the standard library and other such libraries like your boost, which are used heavily in the area. This domain is very different from control logic telepath verification. We need specific formal solvers that can handle complexity arising from mathematical operations. We realized early on when we were developing this tool that since we are doing equivalence between a C++ model and RTL, a specialized debug that works in the context of checking between these two models would make the use of a tool more productive for the users. Ideally, some solutions would allow you to control parts separately. But then you could possibly miss out on bugs that could be there in the design but are only exposed when bugs lie in the area where your control logic and data path transformation logic are intact with each other. So, no data path design is typically just a data path. Usually, you have control logic, which controls the data path pipeline and, let's say, handshake stalls. If it could handle both control and data paths together, a tool would be desirable. Typically, if you're coming from a formal background, even for the control path, you would know that many times, people use advanced, proven the composition techniques to get truths for more complex problems. Until recently, all this was done manually by the users having multiple setups and then aggregating results from all these error-prone and bulk setups. We now have a unique method to manage all these techniques, such as assumed guarantee case splits, stop ads, and cut points, etc.

Solution: Jasper C2RTL App (Datapath Verification)

Enables efficient datapath design verification by mathematically proving the functional equivalence of implemented RTL against a golden C++ reference model.



376 © Cadence Design Systems, Inc. All rights reserved.



The solution we are providing for this domain is this app, which is JasperC to the Huddl app, given a C++ reference model. It mathematically proves that the IT model does the same transformation for the input data as the C++ model does. So, for the same input values, we validate that the output values from the two models are the same. This app can also do C versus C or C versus R, which we already discussed. We support the C++ version starting from C 1999 until 2017. The sweet spots for this app would be areas around you. Blocks switched to your unit of arithmetic operations. Including your integer arithmetic, fixed-point arithmetic, and floating-point arithmetic. High-level image processing algorithms like your A's, FFTs, DFTs, or even your compression-decompression algorithms. So, you can do piecewise verification of such huge algorithms as well.

Cryptography is another area to use JasperC to attend because most of these algorithms are very standard, like the advanced encryption standard or data encryption standard. There are a lot of golden C++ reference models available, which are easy to choose. Have the same A or plain text and keys. Then, validate the ciphertext that comes out of the C++ model, and the implemented RTL are the same.

Type of Designs C2RTL App Can Verify

Unit arithmetic operations

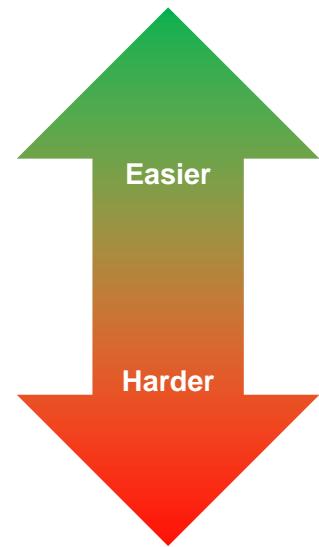
- Integer arithmetic
- Fixed Point arithmetic
- Floating Point arithmetic

Higher level image processing operations/algorithms:

- ACE/LACE (Automatic Color Enhancement / Localized Contract Adaptive Enhancement)
- Matrix multipliers
- FFTs/DFTs (Fourier Transforms)
- Compression/Decompression
- Portions of Video codecs

Encryption/Decryption models

- Several algorithms like AES, DES, RSA, MD5, etc.



377 © Cadence Design Systems, Inc. All rights reserved.

Here are the kinds of datapath designs upon which C2RTL can be deployed.

Jasper C2RTL: C/C++ to RTL Verification App



C2RTL is an equivalence checker that has solvers and techniques to prove equivalence formally.

System-level model (golden) in C/C++

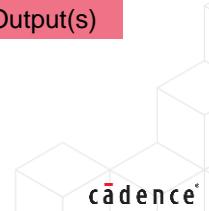
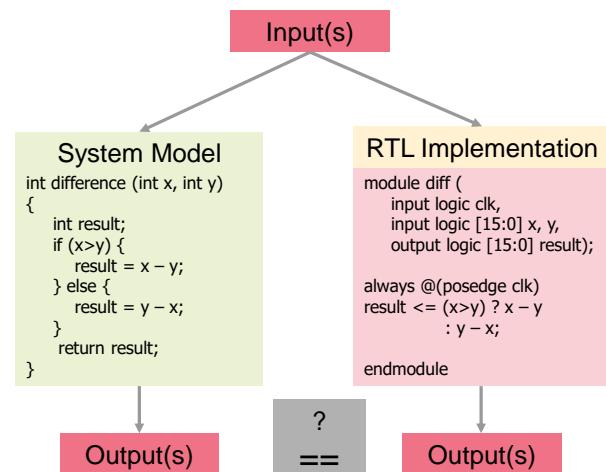
- Untimed
- Transaction-level model

DUT

- Hand-coded RTL implementation
- Timed

Equivalence goal: For the same inputs, both models compute the same output.

- Challenges: Internal structures, data sizes, data flow, and order of operations can differ significantly.
- Needs special solvers and techniques for datapath problems.



378 © Cadence Design Systems, Inc. All rights reserved.

C2RTL is an equivalence checker, so the scenario is typically shown in the slide. The system-level model is straightforward and abstract to create and simulate.

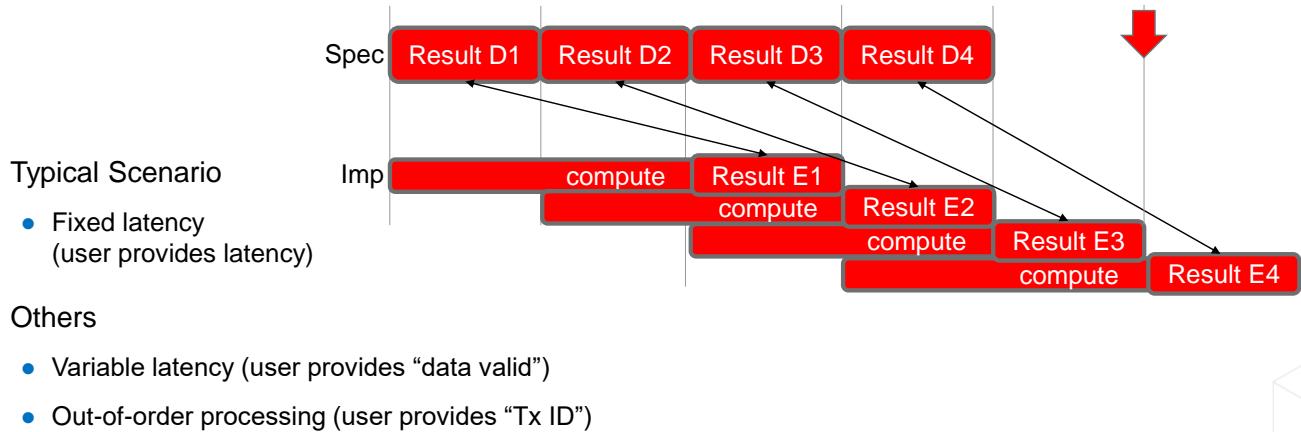
However, to design an IC, we need to implement the design in RTL, most often SystemVerilog.

The goal of equivalence is to prove that if each implementation has the same inputs, then their outputs always agree with each other.

This is not a trivial matter; special solvers and techniques are required to prove equivalence formally.

Example: How C2RTL Extracts Results from Both Models

Given a reference model in C/C++ and MANUAL implementation in RTL, do they have the same functional behavior?



Moving back, the way we do this verification is on the specification side, we have a C++ model, which we always compile to be a combinational circuit. So, there are no props or latches, which means that the data is transformed as soon as the inputs are available, and we have the result available immediately. While on the article side or the implementation side, the article could be pipelined. This means it might take a certain number of cycles before the input data is transformed or computed. Or the results are computed and available on the outputs of the hardware. In this example, we’re just checking a simple two-cycle pipeline that is a fixed latency case. On cycle one, we provide the same inputs to both the spec and M side. The result is on the spec side since its combinational circuit is available immediately on the same cycle. At the same time, RTL, a two-cycle pipeline, takes two cycles to compute the result. We compare the result from cycle one of the spec to cycle three of M. Cycle two from the spec side versus cycle four from the impartial side, and so on. This is the most basic scenario where we have a fixed delay. We validated that it could extract the results from both models for infrared cycles. There could also be scenarios with variable latency or out-of-order processing. Just because there are stalls or, let’s say, handshakes happening on the interfaces. So, that could make this delay between the inputs and the computer available on the output wearable. So, we support all these models in Jasper two at the lab. You could have a fixed latency setup, variable latency setup, and even out-of-order processing setup. All these are doable with CHASPro C2 R2. This is how we handle it in terms of how we set this up.

Jasper C2RTL App Components

Three major components of the app include the following.

Versions: C 99, C++ 98, 03, 11, 14, 17

Support for Softfloat

Datapath-specific proof engines

State-of-the-art debug features for C2RTL

C++ Frontend



C++ Synthesis engine

Broadest Range of Formal Solvers



Leading Performance & Capacity

Visualize™ Interactive Environment



Leading Ease-of-Use

Jasper™ Formal Verification Platform



In terms of the app, it is being developed on the industry-leading Jasper formal verification platform. This means most of the features already available on this platform come for free for this app. In terms of what we are developing specifically for this app, there are three pillars. One is a C++ front end that consumes your C++ model and converts it into a format that formula engines can understand and crunch. The second area second pillar that we have been working on for the last couple of years is the data path solvers or engines. As mentioned earlier, this problem differs significantly from the typical control logic problem. Typically, the complexity does not typically arise from the pipeline depth over here, but it's more due to the arithmetic operators themselves. We now have specialists who could handle these data path problems and prove your RTR to be trained to the golden C++ reference model for all possible inputs. We have also been putting much investment in the debug area. What we are trying to do here is all the excellent debug features that you have in Jasper for that new site. We are trying to bring everything into the C++ domain, which means you can do driver load tracing or analysis on the C++ code itself. This discussion is about all three pillars we are developing specifically for this app to give you a glimpse of what it looks like. The first one is the C++ front end. We have support for all C++ versions right up until 2017. We support your standard C and C++ library and all templated functions, even STL containers, which are standard template library containers. These are handled seamlessly, and we do compile many of them.

Since its release, this app has compiled all 450+ customer models belonging to GPU CPU and some cryptograph domains. Another challenge for compiling the C++ model, especially for formal, is the dynamic nature of the language itself. We could have unbounded loops request a function call and a dynamic memory allocation during runtime variable entries similar to dynamic mobile location. This makes it harder to limit the size of the hardware that you are creating out of it. We handle all of these seamlessly. You don't have to change anything in the code since we create certain safety assertions that check when we synthesize the code. Let's say the results are still valid when we compile the code with all these constructs inside your C++ model. They are not invalid due to certain assumptions that we made about the model. We also generate many checks looking at the C++ model itself, so we already have many success stories where people find bugs in their C++ model. Some examples include catching your unresolved function pointers or `c l r m`, which are not well defined for certain scenarios. The simplest cases are your shift operators' left and right shift operators. So if any of the options is negative, or let's say the shifting is happening. The value by which a number is being shifted is more than the size of the number itself. So, C++ URL leaves it either undefined or implementation-defined, which could mean you could see different results with different compilers. So, we create assertions to check such scenarios and present them to the user. We also support compiling asserts written in the C++ portal itself. These would appear in your Jasper property table, and you can prove them just like any other property you prove for your auto today.

Component 1: Jasper C++ Frontend

Excellent C/C++ language constructs and Library support

- C99 and C++98, 03, 11, 14, 17 are all supported well, including templated functions and classes.
- C/C++ standard library (including STL containers).
- 400+ customer C++ models synthesized without any code change (mostly GPU and CPU).

Dynamically sized structures handled well:

- Unbounded Loops, recursive function calls.
- Dynamic memory allocation, Variable Length Arrays (VLAs).

Autogenerated checks for C++ model.

- Helps catch bugs in the reference model itself.
- Examples: Unresolved function pointers, C++ undefined behaviors, etc.

Support for C/C++ user Asserts.



We now discuss the three pillars we are developing specifically for this app. The first one is the C++ front end. Regarding follow-up, this is one of the best C++ compilers. It supports the standard C++ library and template functions, even STL containers, which are standard template library containers. So, all of those are handled seamlessly, and we compile many of them. Another challenge in compiling the C++ model is the dynamic nature of the language itself. There could be unbounded loops, recursive function calls, dynamic memory allocation during runtime, and variable entries, which are similar to dynamic communication. This makes it harder to limit the size of the hardware that you are creating out of it. This is handled seamlessly without changing anything in the code. Certain safety assertions check that the results are still valid when we synthesize or compile the code with all these constraints inside your C++ model. We also generate a lot of checks by looking at the C++ model itself. Some examples can catch unresolved function pointers. Or, a C++ alarm is poorly defined for specific scenarios such as left and right shift operators. If any of the options are negative or shifting is happening, the value by which a number is being shifted is more than the size of the number itself. A C++ alarm leaves it either undefined or implementation-defined, which could mean that you could see different results with different compilers. Assertions are created to check such scenarios and present them to the user. It also supports compiling asserts written in the C++ portal that appear in your Jasper property table.

Component 2: Jasper C2RTL Platform Proof Features

Machine Learning based proof orchestration:

- Auto engine/solver selection.
- ML-based engine optimizations.

Optimized usage of compute results:

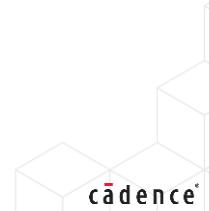
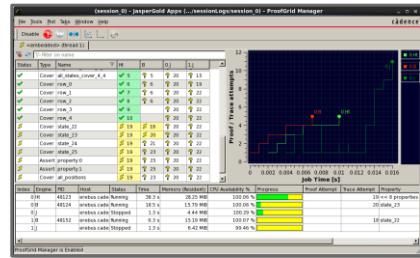
- Caching of proof results between runs (**provecache**).
- Use learnings from previous runs for engine choices (**PPD – Proof Profiling Data**).

Running hundreds of parallel-proof jobs on server farms:

- **Proofgrid** efficiently manages these parallel runs under user-provided resource constraints.

Powerful Jasper interactive-proof cockpit:

- Assume guarantee, cutpoints, and intermediate helper lemmas.
- All environment modifications without the need to recompile.



382 © Cadence Design Systems, Inc. All rights reserved.

Proof features for this app are being developed on the Jasper Group platform with features like ML-based proof orchestration, which auto-selects engines, solvers, and different optimizations for engines. We have a machine learning-based selection of these engines and heuristics, which will choose the best solver engine depending on the test case under verification that allows you to optimize the usage of your computer sources. This is something we call a proof cache and proof of filing data. Proof cache caches results from your previous runs. If the changes in your environment or your article do not impact certain properties, the results of those properties are picked up from the cache. Only the properties that get affected are rerun. PPD is an ML-based proof orchestration that automatically chooses the best engines. It keeps track of which engines work better on certain kinds of properties in previous runs. In the next runs, ML will prioritize those engines or heuristics for the properties they have already shown to work better on. The proof grade allows you to run hundreds and thousands of j paddle jobs. A unique feature of Jasper is an interactive proof kit, proof cockpit, which allows you to write your assertions assumes on the fly, cutpoints, intermediate helper sessions, and everything on the fly in Jasper Consul, without having to recompile either your RTL or C++ model again and again. This is really powerful and allows you to do a lot of experiments on the fly without having to recompile.

Component 2: Jasper C2RTL Platform Proof Features (continued)

Proof Engines

State-of-the-art datapath proof stack:

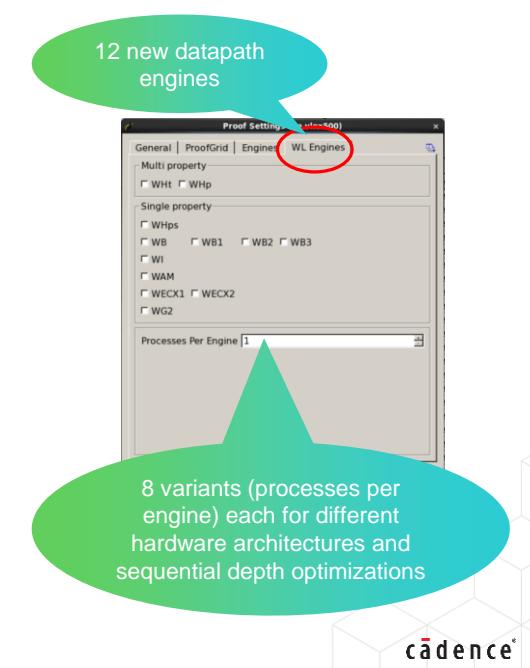
- Powerful bit-level and word-level solvers.
- Industry-leading powerful engines.

Datapath-specific optimizations:

- Support for floating-point multiplication to minimize manual case splits.
- Dedicated handling of large integer multipliers implemented using smaller multipliers and adders.
- Word-level arithmetic abstractions, term-rewriting, normalization, reductions in the size of vectors.

Specialized bit-level multiplier handling:

- Special engines to verify bit-level implementations of array multipliers, radix2, radix4, radix8 booth multipliers with variants like Baugh Wooley.



383 © Cadence Design Systems, Inc. All rights reserved.

There is a new tab for data path engines.

Component 3: Jasper C2RTL Debug

Dual debug for C++ vs. RTL

RTL-like waveform debug for C++ code

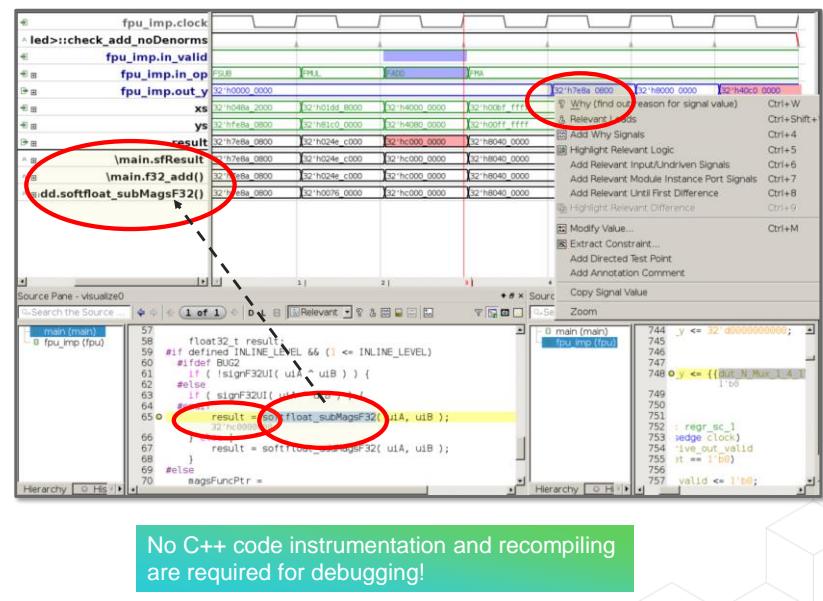
- Why tracing in C++
- Driver/load tracing
- Value annotation in source

Specialized debug for C/C++

- All variables/function-calls uniquely identifiable and plottable
- Full call stack in variable names

Powerful Visualize features integrated

- What-if analysis
- Quiet trace
- Freeze and extend, etc.



384 © Cadence Design Systems, Inc. All rights reserved.

There is a specialized debug for the C++ side so that all variables and function calls are uniquely identifiable. We can plot each of them into the waveforms and check their values. In Jasper C2 audio, we don't have to put additional hooks to look at values of intermediate variables on the C++ side. All the powerful and unique Jasper features like water analysis, quiet trace, and freeze extents all work. With a what-if analysis, you can choose an output variable or any variable in the C++ model and then tell Jasper to show certain behaviors on that variable. It reduces the amount of toggling into the model to get to the failure. Essentially, only its variable toggles are necessary to get a count example. Only those are allowed. The rest of the design is completely silent. So it will enable you to focus on the root cause of the problem quickly.

Component 3: Jasper C2RTL Debug (continued)

Floating-Point Support

Special radices for IEEE 754 representations:

- Half precision
- Single precision
- Double precision

xs	32'h048a_2000	32'h01dd_8000	32'h4000_0000	32'h00bf_ffff
ys	32'hfe8a_0800	32'h81c0_0000	32'h4080_0000	32'h00ff_ffff
result	32'h7e8a_0800	32'h024e_c000	32'hc000_0000	32'h8040_0000
xs	3.247e-36	8.137e-38	2.000e+00	1.763e-38
ys	-9.174e+37	-7.053e-38	4.000e+00	2.351e-38
result	9.174e+37	1.519e-37	-2.000e+00	-8.816e-39

Custom Floating-Point Numbers Support:

- Defined in JSON format

```
{
  "header": {
    "type": "visualize_radix_file",
    "version": 1
  },
  "radices": [
    "bfloating16": {
      "type": "floating_point",
      "mapping": {
        "16": {"exponent": 8, "mantissa": 7}
      }
    },
    "GPU_tensor": {
      "type": "floating_point",
      "mapping": {
        "16": {"exponent": 5, "mantissa": 10},
        "32": {"exponent": 8, "mantissa": 23},
        "64": {"exponent": 11, "mantissa": 52}
      }
    },
    "CPU_tensor": {
      "type": "floating_point",
      "mapping": {
        "16": {"exponent": 5, "mantissa": 10},
        "32": {"exponent": 8, "mantissa": 23},
        "64": {"exponent": 11, "mantissa": 52}
      }
    }
  ],
  "usages": {
    "sig0": "bfloating16",
    "sig1": "GPU_tensor",
    "sig2": "GPU_tensor",
    "sig3": "GPU_tensor",
    "sig4": "CPU_tensor"
  }
}
```

385 © Cadence Design Systems, Inc. All rights reserved.



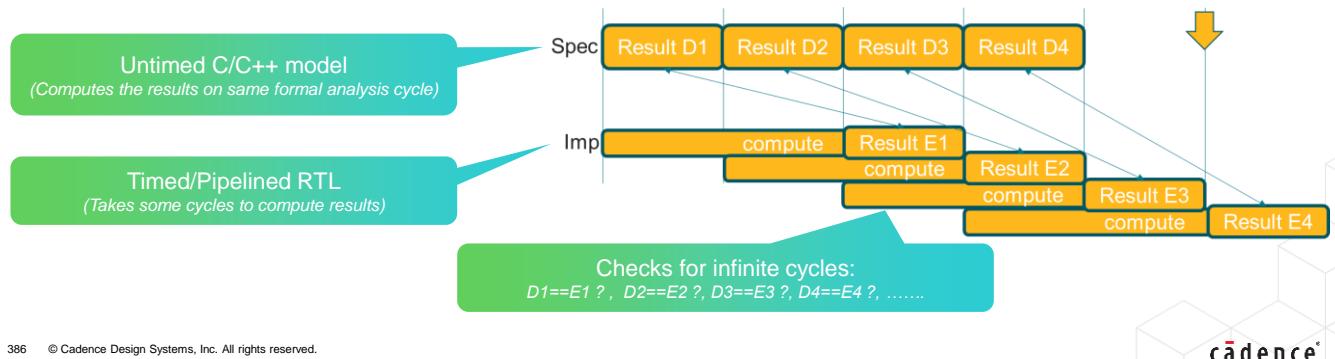
The floating point unit is the most used use case for this app. There is special support for protocol numbers. It supports a triple half signal and double position, which means we can either look at the hexadecimal value for the signal called result on cycle three, or you can look at the equivalent decimal value for the floating point hexadecimal representation. To debug, we don't need to rely on an external calculator to keep converting the hacks accessible to a value representation to your decimal value and vice versa. Additional user-defined forty-point redis is supported, wherein you can define the size of the redis redics. These new retics defined will be available in the reform debugger to choose for any of the signals, which makes your development much more productive.

Component 3: Jasper C2RTL Debug (continued)

Verification Setup

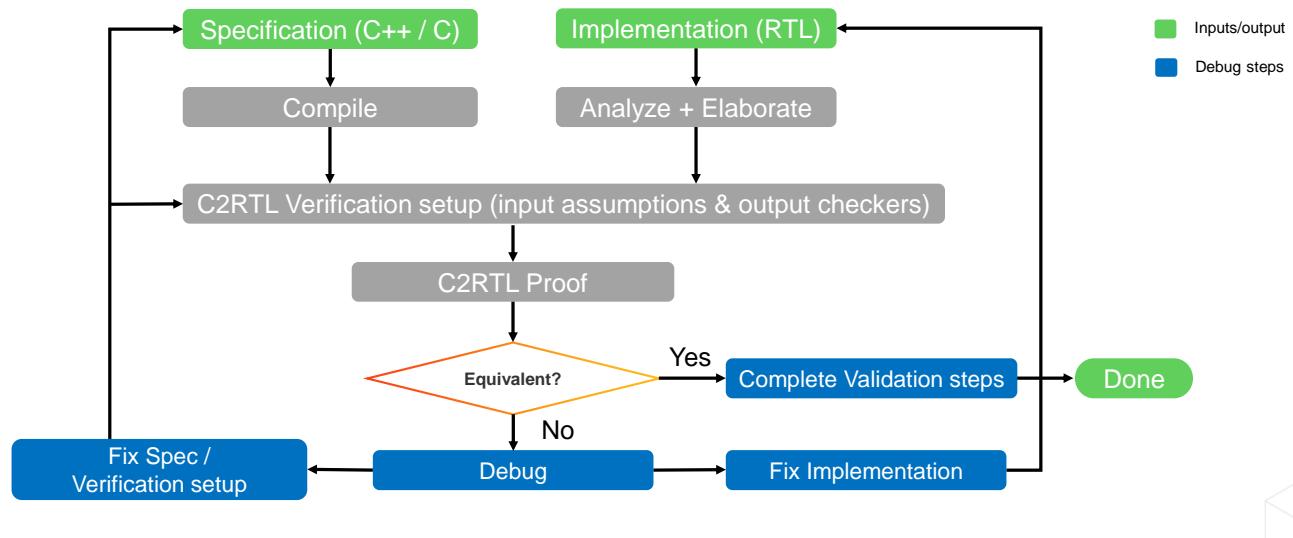
Jasper C2RTL verification setup is similar to any model checking setup => easy ramp up!

- Inputs assumed to be the same between two models, asserts to check for output equivalence with appropriate pipeline delays.
- Checks operations for all cycles on an infinite timeline.
- Allows simultaneous verification of control and datapath logic.
- Supports SVA style constraints/assertions; portable between simulation/FPV environments.



In terms of verification setup, we assume the inputs to be the same, which could be a cycle or onboarding cycle spread over multiple cycles. We check the results to be the same between the two models on specific cycles or after taking care of the delay, which could be both fixed and variable. So, this setup is similar to a typical model checking setup where we have resumes on the inputs, and we have asserts to check the behavior of the outputs. This app will check all operations in the evening to produce an exhaustive proof. It allows simultaneous verification of control, which means you don't have to tie off your control logic to the active values. You can leave them free as they would be in your real-life scenario, and Jasper will still give you clues on your data path designs. It supports SPS trial constraints and decisions, making it easy to port them between your submission and FPD amendments.

Jasper C2RTL App – Flow Diagram



387 © Cadence Design Systems, Inc. All rights reserved.



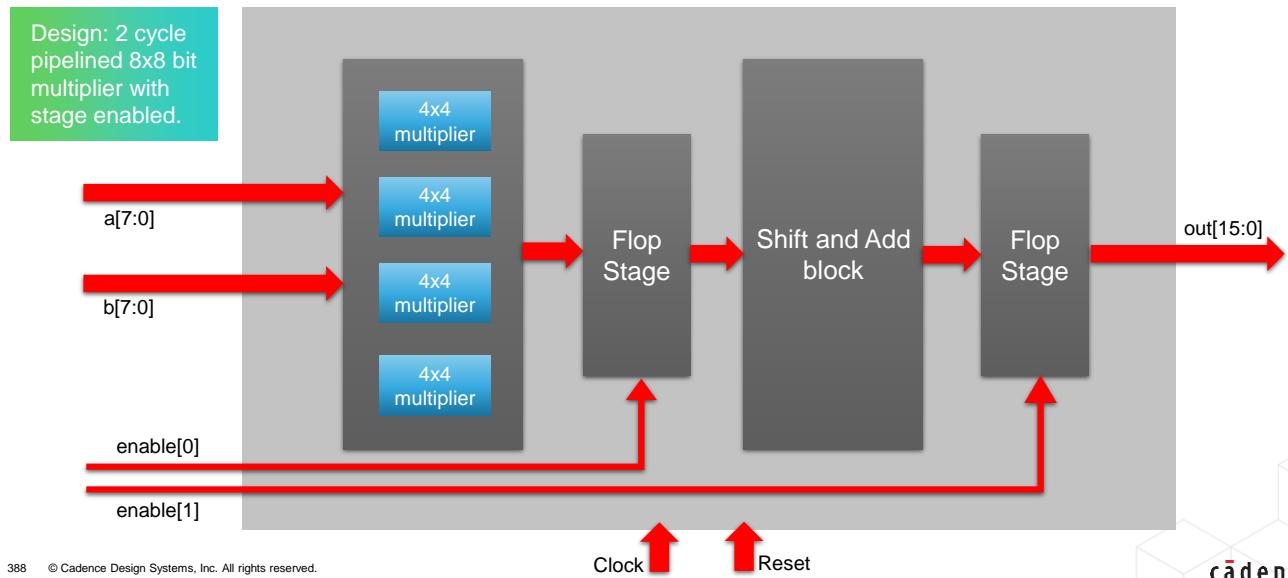
We compile the C+ model on the specification side. On the implementation side, it could either be a C model when we are doing C+C versus C, or typically, it would be an ideal model. Compile this on the implementation side and then create a verification setup, which means defining the assumptions on the inputs, checkers, and outputs. Once that is done, we run our proof engines. If we get a proof, then we go through some validation steps, and if everything is green, we are done. If we get a counterexample, then we debug, and depending on where the problem is, we either fix the specification, the setup, or the implementation.



Lab

Lab 19-1 Equivalence Checking of a Pipelined Multiplier

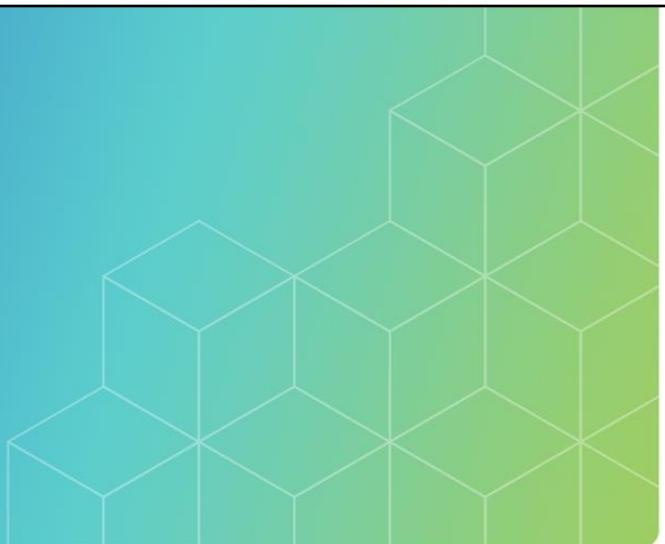
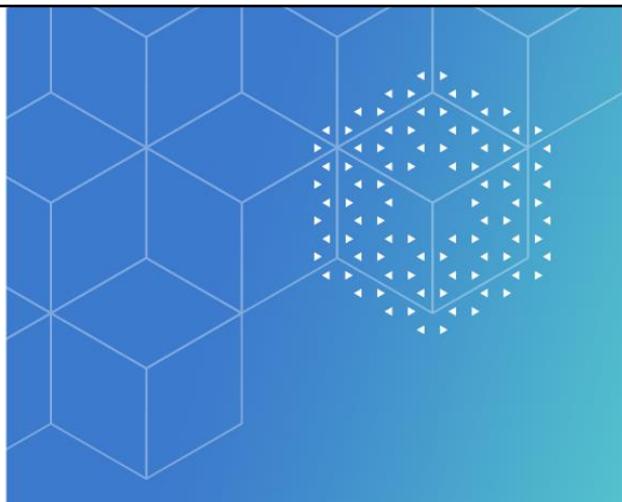
Spec: $\text{out} = \text{a} * \text{b}$



In the lab, we will perform an equivalence check between a C++ and RTL implementation of a pipelined multiplier, as shown in the diagram.

There are various implementations of the multiplier, which we will equivalence check.

Details of how to do that are in the Lab Manual.



Module 20

Course Conclusions

cadence®

This page does not contain notes.

Summary

In this training course, you:

- Developed fundamental C++ programming skills

This training course discussed:

- Object-Oriented Programming and C++
- C++ Basics
- Constructors and Destructors
- References
- Functions
- Type Conversion
- Operator Overloading
- Inheritance
- Polymorphism
- Constant Objects and Constant Functions
- Templates
- Exceptions
- Input and Output
- Debugging
- Containers and Algorithms



This page does not contain notes.

References

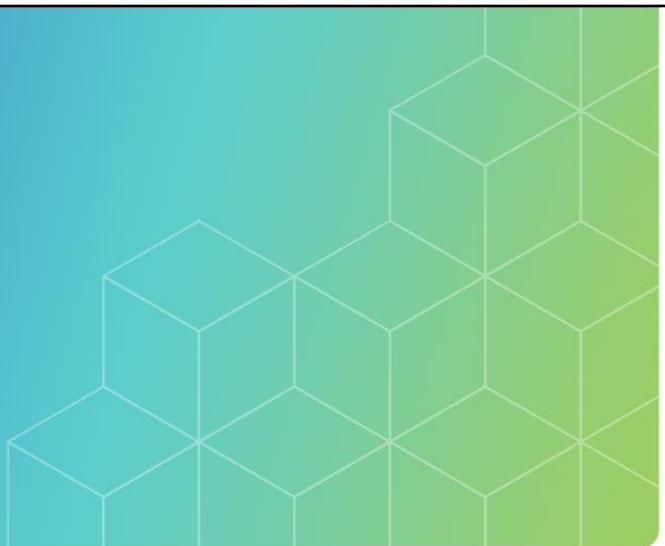
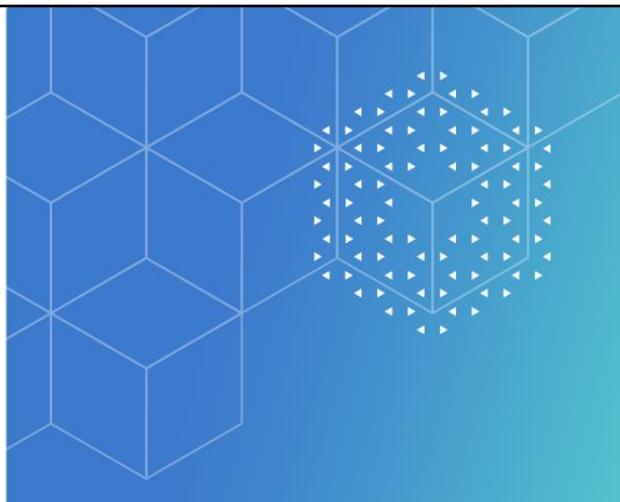
You can obtain additional information from the following sources:

- Standards
 - ISO/IEC 14882:2020 Programming languages – C++ (<https://www.iso.org/store.html>)
- Net Advocacy
 - C++ Super-FAQ (<https://isocpp.org/faq>)
 - newsgroup: <https://groups.google.com/g/comp.lang.c++>
- Publications
 - “The C++ Programming Language”. Bjarne Stroustrup, Pearson Education, 2013 (<https://www.pearson.com/us/higher-education.html>)
- Other
 - Frequently asked questions – cppreference.com (<https://en.cppreference.com/w/>)
 - Stack Overflow – Where Developers Learn, Share, & Build Careers (<https://stackoverflow.com/>)
 - C++ Programming Language - GeeksforGeeks (<https://www.geeksforgeeks.org/c-plus-plus/>)

391 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Module 21

Next Steps

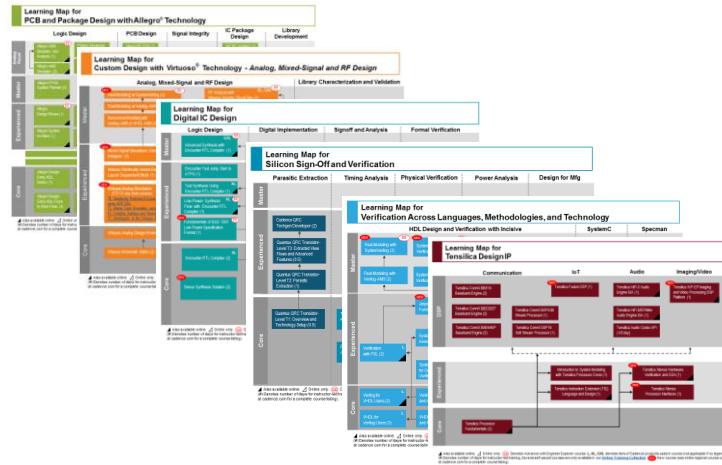
cadence®

This page does not contain notes.

Learning Maps

Cadence® Training Services learning maps provide a comprehensive visual overview of the learning opportunities for Cadence customers.

Click [here](#) to see all our courses in each technology area and the recommended order in which to take them.



393 © Cadence Design Systems, Inc. All rights reserved.



Go here to view the learning maps:

http://www.cadence.com/Training/Pages/learning_maps.aspx

Cadence Learning and Support

The screenshot shows the Cadence Learning and Support website. At the top, there's a navigation bar with links for Cases, Tools, IP, Resources, Learning, Software, My Support, and Contribute Content. To the right of the navigation are a bell icon and a user profile icon. The main header features the Cadence logo and the text "LEARNING & SUPPORT". Below the header is a search bar with placeholder text "Start your search here...". Underneath the search bar are two buttons: "View History" and "Documents Liked". A large play button icon is overlaid on the center of the page. Below the search bar, there's a message: "Know more about a product: Choose a product...". Further down, there are six categories with icons: "Installation & Licensing" (gear), "Product Manuals" (book), "Training Courses" (document), "What's New" (lightbulb), "Troubleshooting Information" (wrench), and "Video Library" (play). A banner below these categories states: "Customer Support now includes over 2000 product/language/methodology videos ("Training Bytes")!". The footer contains the text "394 © Cadence Design Systems, Inc. All rights reserved." and the Cadence logo.

Click the play button in the figure on this slide to view the demo of Cadence Learning and Support.

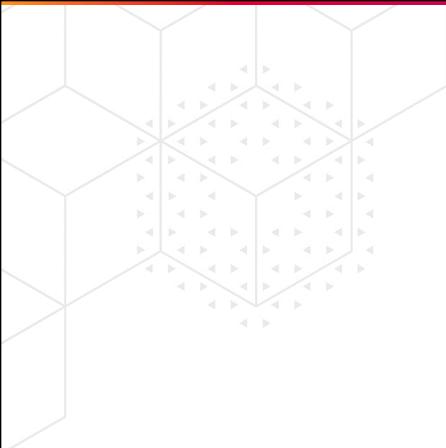
Wrap Up

- Complete Post Assessment, if provided
- Complete the Course Evaluation
- Get a Certificate of Course Completion

Thank you!



This page does not contain notes.



cadence®

© Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, the Cadence logo, and the other Cadence marks found at <https://www.cadence.com/go/trademarks> are trademarks or registered trademarks of Cadence Design Systems, Inc. Accellera and SystemC are trademarks of Accellera Systems Initiative Inc. All Arm products are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All MIPI specifications are registered trademarks or service marks owned by MIPI Alliance. All PCI-SIG specifications are registered trademarks or trademarks of PCI-SIG. All other trademarks are the property of their respective owners.

This page does not contain notes.