# C++ PROGRAMMING I (MCCLANAHAN)

*Patrick McClanahan*
San Joaquin Delta College

LibreTexts™

# San Joaquin Delta College

# C++ Programming I (McClanahan)

Patrick McClanahan

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

This text was compiled on 07/15/2025

# TABLE OF CONTENTS

# Licensing

*A detailed breakdown of this resource's licensing can be found in* ***Back Matter/Detailed Licensing****.*

# CHAPTER OVERVIEW

## Front Matter

# CHAPTER OVERVIEW

## 1: Building and Running C++ Code

# 1.1: Building a C++ Program

## What's in a File?

There is nothing special about a file that contains computer source code. It is simply a text file that contains text written in a specific syntax that defines what language is being used. So, in reality you can use any text editor to write your code - something as simple as Notepad, or for you Linux fans out there I used to write all my code in Vi (now known mostly as vim). However, I prefer to use an integrated development environment, more commonly referred to as an IDE. My personal favorite right now is MS Visual Studio Code (this is the community edition - its FREE). There are several IDE's to choose from, you should look around, and find one you like and use it.

It is the convention that source code have a specific extension on the filename to signify what language the code is written in. For C++ the extension is `.cpp`. Now if you have C++ code in a file with an extension of, lets say .xyz, it will still compile and run, but anyone looking at your files would have no idea it is C++ code. You may also have header files which have a `.h` extension, but more on that in a bit.

Back to our source code file...you may be thinking (at least I hope you are), "Wait, the computer doesn't execute a text file??!!". You are correct. The file that contains the computer source code must go through a translation process that creates the file that we can then run on our computer. The term "compiling" is often used to refer to the entire process of translating the source doe to executable code. However, there is an entire series of steps that actually occur during this process. We will briefly review the three major steps individually: preprocessing, compiling, and linking.

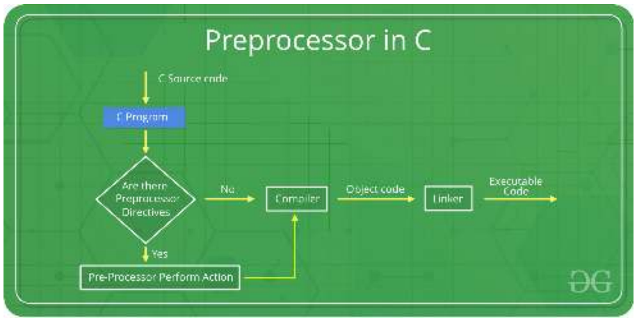
From a broad perspective the compiling process can include multiple source code files, perhaps one or more header files, and generates an object file for each of the source files. These object files have their own extension: a `.o` in Linux, or a `.obj` file for Windows. Once we have the object files created they are all pulled together and linked to the system libraries, which include certain functions necessary for the program to run. The output is a file that is able to execute on the specific processor and operating system it needs to run on.

Let's move along and take a deeper dive into this entire proces...

# 1.2: Understanding Preprocessing

## C/C++ Preprocessors

As the name suggests Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++. Let us have a look at these steps before we actually start learning about Preprocessors.



You can see the intermediate steps in the above diagram. The source code written by programmers is stored in the file program.c. This file is then processed by preprocessors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program .obj. Finally, the linker links this object code file to the object code of the library functions to generate the executable file program.exe.

Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that, whatever statement start with #, is going to preprocessor program, and preprocessor program will execute this statement. Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef* etc. Remember that # symbol only provides a path that it will go to preprocessor, and command such as include is processed by preprocessor program. For example include will include extra code to your program. We can place these preprocessor directives anywhere in our program.

**There are 4 main types of preprocessor directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

Let us take a brief look at each of these preprocessor directives.

### Macros

- In C++ macros are always proceeded by **#define**. Macros are a piece of code which is given some name. Whenever this name is encountered by the preprocessor the preprocessor replaces the name with the actual piece of code. The '#define' directive is used to define a macro. Let us now understand the macro definition with the help of a program:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
#include <iostream>
#define MAX 15

std::cout << "The max value is " << MAX << std::endl;
```

The word ‘MAX’ in the macro definition is called a macro template and ‘15’ is macro expansion. In the above program, when the pre-processor part of the compiler encounters the word MAX it replaces it with 15. It is as if you typed the code with the number 15 instead of the word MAX. The idea is that if you change your code, you only have to make a single edit, you do NOT have to go through your code and find every place where you have used that value. This may not seem like that big of a deal - until you have a coding project with 100,000 lines of code - you really don't want to sit and go through our code or even do a find-and-replace operation.
**Note**: There is no semi-colon(‘;’) at the end of macro definition. Macro definitions do not need a semi-colon to end.

**Macros with arguments**: We can also pass arguments to macros. Macros defined with arguments works similarly as functions. In the following example there are 2 macros - one for AREA which takes 2 arguments and multiplies them. The other macro is DOUBLE_IT, which takes 2 arguments and multiplies them together, but this macro is used to show how macros can look straight forward, but actually be broken.

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
#include <iostream>

// macro with parameter
#define AREA(l, b) (l * b)
#define DOUBLE_IT(arg) (arg * arg)

int main()
{
    int l1 = 10, l2 = 5, area, bad_result;
    area = AREA(l1, l2);
    std::cout << "Area of rectangle is: " << area << std::endl;

    // The compiler sees this as:
    // int bad_result = (4 + 1 * 4 + 1);
    // due to mathematical order of precedence this gives wrong answer
    bad_result = DOUBLE_IT(4 + 1);
    std::cout << "Bad Results: " << bad_result << std::endl;
    return 0;
}
```

```
main();
```

As you can see in the second part, DOUBLE_IT causes a problem - the input arguments are not evaluated properly and produces an incorrect answer. The solution to this particular problem is to add a set of parenthesis in the macro definition:

```
#define DOUBLE_IT(arg) ((arg) * (arg))
```

One more mention of macros - they can be a bit complicated and even extend over multiple lines - BEWARE - this makes it much more difficult to debug

```
#include <iostream>
#define PRINT(i, limit) while (i < limit) \
                        { \
                            std::cout <<"CIS 31A Quiz " << std::endl;
                            i++; \
                        }
int main()
{
    int i = 0;
    PRINT(i, 3);
    return 0;
}
```

Adapted from: "C/C++ Preprocessors" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 1.2.1: Understanding Preprocessing- File Inclusion

his type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files which can be included by the user in the program:

1. **Header File or Standard files**: These files contains definition of pre-defined functions like cout & cin, getline() etc. These files must be included for working with these functions. Different function are declared in different header files. For example: standard Input/Output functions are in ‘iostream’ file, whereas functions which perform string operations are in 'string’ file.
   **Syntax**:

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

   where *iostream* and *string* are both names of files that the system will include. The ‘<‘ and ‘>’ brackets tells the compiler to look for the file in standard set of directories. The contents of these files are literally imported into the source code as part of the pre-processor step so that when the compiler goes to locate the function definitions they are found.

   A full list of C++ header files shows all of the currently supported header files and a brief description of what they are used for.

2. **User Defined Header Files**: Users have the ability define their own functions that are used across several files or projects. These files can be included into the source code as:

   ```
   #include "filename"
   ```

where *filename* is the name of the actual file that contains the code. Notice that user defined files are enclosed in double quotes as opposed to the < > brackets. Normally these files exist in the same folder as the .cpp file. Certain tools that are used to build large software projects have the capability to define other locations for header files...but that is beyond the scope of this module.

One more note: many header files in the past, especially in the C programming world, ended in a .h - some of these files continue to exist in code. So if you see a file name such as <stdio.h> - know that is a valid name, just an older format.

Adapted from: "C/C++ Preprocessors" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 1.2.2: Understanding Preprocessing- Conditional Compilation

## Conditional Compilation

Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions. This can be done with the help of several preprocessing commands  #if, #ifdef, #ifndef, #if, #endif, #else and #elif. (#ifdef - is 'if defined', and #ifndef - is 'if not define')

Using these commands is relatively simple. The #ifdef directive says, "If this is defined" - in the following code it says, "If the macro TABLE_SIZE is defined, then declare an integer array called table, that is TABLE_SIZE in length". If TABLE_SIZE has not been defined, then the integer array table will NOT be defined.

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

The #if, #ifdef, and #ifndef directives MUST end with an #endif statement, otherwise the condition continues until one is found.

A little more complex example is shown below:

Login with LibreOne to run this code cell interactively.

If you have already signed in, please refresh the page.

Login

```
#include <iostream>

#define TABLE_SIZE 500

#if TABLE_SIZE > 200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE < 50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

```
std::cout << "The table will hold: " << TABLE_SIZE << " values" << std::endl;
```

So, if the macro TABLE_SIZE is greater than 200, the macro is undefined (by the #undef directive), the defined once more with a value of 200. Else if the macro has a value less than 50 it is undefined, and redefined with a value of 50. Otherwise it is set to 100.

Try changing the value of TABLE_SIZE in the #define statement at the top, and see if you can predict the correct output.

Adapted from: "C/C++ Preprocessors" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 1.3: Header Files

## A Bit More About Header Files

Just wanted to provide a bit more clarification about header files - to show what actually happens with the preprocessor.

Let's say we have some code for our big project, and we have some customized functions that we declare in the file foo-1.h. In the code example below we call the function testFoo(), which is is defined in the file foo-1.

```
// file: main.cpp
#include <iostream>
#include "foo-1"
int main() {
  int myFoo;

  myFoo = testFoo();
  cout << "Returned from testFoo: " << myFoo << endl;
}
```

So, the foo-1 file looks like and simply returns the value defined by our macro which is 99.

```
#define FOOVAL 99

int testFoo() {
  return FOOVAL;
}
```

The main.cpp outputs:

```
Returned from testFoo: 99
```

The preprocessor then actually passes the following code to the compiler to convert into an executable program.

```
// file: main.cpp
// The iostream file code has been ommitted in this example but it would be here.

#define FOOVAL 99

int testFoo() {
  return FOOVAL;
}


int main() {
  int myFoo;

  myFoo = testFoo();
  cout << "Returned from testFoo: " << myFoo << endl;
}
```

This is a simple example and there is a bit more detail that is left out of this example, but hopefully this calrifies what the preprocessor actually does with header files.

# 1.4: Compiling

## The Compile Process

The process then to take the preprocessed code and turn it into executable code is a bit more complex. Depending on who you are talking to the steps may be a called something a bit different - but the actual process is the same, we will take a brief look at the steps in the next several pages.

- **High Level Language –** refers to a programming language such as C++, Java, C#, etc. They are typically much easier for humans to read an understand than low level languages like machine code and assembly language.
- **Pre-Processor –** The pre-processor removes all the directives and performs: file inclusion, macro-processing, performs conditional compilation as previously discussed
- **Assembly Language –** Its neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler –** For every platform (Hardware + OS) we will have a assembler. They are not universal since for each platform we have one. The output of assembler is called object file. Its translates assembly language to machine code.
- **Relocatable Machine Code –** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate for the program movement.
- **Loader/Linker –** It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.
- **Absolute Machine Code -** This is the executable file, based on the system architecture and operating system it is to run on.

Adapted from: "Introduction of Compiler Design" by Rajesh Kr Jha, Geeks for Geeks is licensed under CC BY-SA 4.0

# 1.5: Compiling- Analysis Phase

## Lexical Analyzer - also called *tokenization.*

The phase of the compiler splits the source code into tokens. A *lexical token* or simply *token* is a string with an assigned and thus identified meaning. It is structured as a pair consisting of a *token name* and an optional *token value*. Common token names are

**identifier**: names the programmer chooses;

**keyword**: names already in the programming language;

**separator** (also known as punctuators): punctuation characters and paired-delimiters;

**operator**: symbols that operate on arguments and produce results;

**literal**: numeric, logical, textual, reference literals;

**comment**: line, block.

Examples of token values

| Token name | Sample token values |
|---|---|
| identifier | `x` , `color` , `UP` |
| keyword | `if` , `while` , `return` |
| separator | `}` , `(` , `;` |
| operator | `+` , `<` , `=` |
| literal | `true` , `6.02e23` , `"music"` |
| comment | `/* Retrieves user data */` , `// must be negative` |

Consider this expression in the C programming language:

```
x = a + b * 2;
```

The lexical analysis of this expression yields the following sequence of tokens:

```
[(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b),
(operator, *), (literal, 2), (separator, ;)]
```

## Syntax Analyzer

This is the process of checking a string of symbols, created by the lexical analysis stage,  to see how well the symbols are how they need to be. This is determined by the use of the rules of a formal grammar. For C++, as with any programming language, is a well defined syntax of the language, just as there is a defined syntax for all written languages.

If the symbols do NOT follow the rules of the grammar the compiler generates a "syntax error" and compilation stops, and attempts to communicate to the user what and where the error is.

## Semantic Analyzer

Semantic analysis performs semantic checks such as type checking (makes sure that mathematical operations are being performed on variables declared as int or float), or object binding (making sure that declarations match and function calls and types are correct), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Adapted from: "Introduction of Compiler Design" by Rajesh Kr Jha, Geeks for Geeks is licensed under CC BY-SA 4.0
"Lexical analysis" by Numerous contributors is licensed under CC BY-SA 3.0

# 1.6: Intermediate steps

## Intermediate Code Generator

Most compilers translate a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code specifically for the machine architecture and operating system that the code is being written for.

The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced.For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.

In a compiler that uses an intermediate language, there may be two instruction selection stages—one to convert the parse tree (that information that comes from the previous stages of the compiling process) into intermediate code, and a second phase much later to convert the intermediate code into instructions from the instruction set of the target machine.

Adapted from: "Intermediate Code Generation in Compiler Design" by Aaqib Bashir is licensed under CC BY-SA 4.0

# 1.7: Linking and Libraries

## Now We Link

The linker is a program in a system which helps to link a object modules of program into a single object file. It performs the process of linking. Linker are also called link editors. Linking is process of collecting and maintaining piece of code and data into a single file. Linker also link a particular module into system library. It takes object modules from assembler as input and forms an executable file as output for loader.

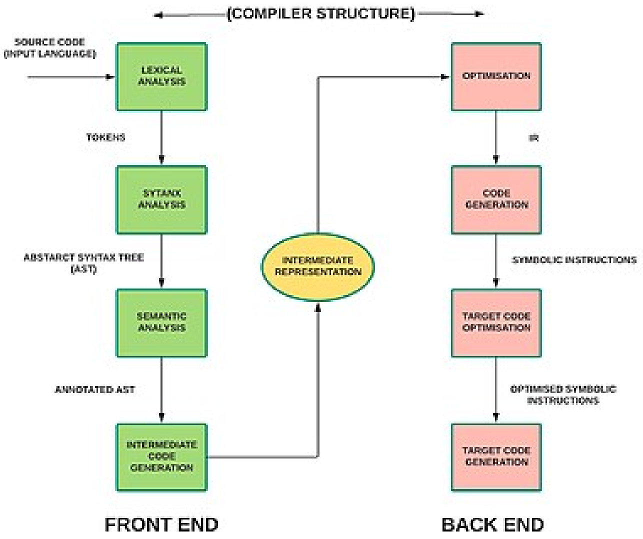


Figure 1.7.1: Compiler Structure Explanation. ("COMPILER STRUCTURE EXPLANATION" by Sharmila chandrakanth, WikiMedia is licensed under CC BY-SA 4.0)

There are two types of linking

**1. Static Linking –**
Linking is performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory by the loader. Linking is performed at the last step in compiling a program.

- **Symbol resolution –** It associates each symbol reference with exactly one symbol definition .Every symbol have predefined task.
- **Relocation –** It relocate code and data section and modify symbol references to the relocated memory location.

The linker copy all library routines used in the program into executable image. As a result, it require more memory space. As it does not require the presence of library on the system when it is run . so, it is faster and more portable. No failure chance and less error chance.

**2. Dynamic linking –** Dynamic linking is performed at run time. This linking is accomplished by placing the name of a shareable library in the executable image. There is more chances of error and failure chances. It require less memory space as multiple program can share a single copy of the library.

Here we can perform code sharing. it means we are using a same object a number of times in the program. Instead of linking same object again and again into the library, each module share information of a object with other module having same object. The shared library needed in the linking is stored in virtual memory to save RAM. In this linking we can also relocate the code for the smooth running of code but all the code is not relocatable.It fixes the address at run time.

The final executable file output by the linker (see dieagram to the left) will consist of both our compiled program and the standard library if the linking was static. On the other hand, if the linking is dynamic, the linker marks the symbols that need to be found at runtime.

When we run the program, the loader will load the library that was dynamically linked to our program. It loads the contents of the standard library into the memory and then resolves the actual location of any function calls that are necessary. The same library that is now loaded into memory can also be accessed by other programs that need to resolve their function calls.

Adapted from:
"Intermediate Code Generation in Compiler Design" by Aaqib Bashir is licensed under CC BY-SA 4.0

# 1.8: Standard Libraries

## Overview of Standard Libraries

A brief explanation of what is being linked to our code, that is, the system's standard libraries, and if necessary we have project specific libraries as well.

Many common or **standard functions**, whose definitions have been written, are ready to be used in any program. They are organized into a group of functions (think of them as several books) and are collectively called a **Standard Library** There are many function organized into several libraries For example, within C++ many math functions exist and have been coded (and placed into libraries). These functions were written by programmers and tested to insure that they work properly. In most cases the functions were reviewed by several people to double and triple check to insure that they did what was expected. We have the advantage of using these functions with **confidence** that they will work properly in our programs, thus saving us time and money.

A main program must establish the existence of functions used in that program. In C++ there is a formal way to:

- define a function
- declare a function (a prototype is a declaration to a compiler)
- call a function

When we create functions in our program, we usually see them in the following order in our source code listing:

- declare the function (this is referred to as a prototype of the function)
- call the function (actually use it in our code)
- define the function (This is the actual code for the function)

When we use functions created by others that have been organized into library, we include a header file in our program which contains the prototypes for the functions. Just like functions that we create, we see them in the following order in our source code listing:

- declaring the function (prototype provided in the include file)
- call the function (with parameter passing of values)
- define the function (it is either defined in the header file or the linker program provides the actual object code from a Standard Library object area)

In most cases, the user can look at the prototype and understand exactly how the communications (parameter passing) into and out of the function will occur when the function is called. Let's look at the math example of absolute value. The prototype is:

```
int abs(int number);
```

Not wanting to have a long function name the designers named it: **abs** instead of "absolute". This might seem to violate the identifier naming rule of using meaningful names, however when identifier names are established for standard libraries they are often shortened to a name that is easily understood by all who would be using them. The function is of data type int, meaning that the function will return an integer value. It is obvious that the integer value returned is the answer to the question, "What is the absolute value of the integer that is being passed into the function". This function is passed only one value; an int number. If I had two integer variables named apple and banana; and I wanted to store the absolute value of banana into apple; then a line of code to call this function would be:

```
apple = abs(banana);
```

This statement says to pass the function abs the value stored in variable banana and assign the returning value from the function to the variable apple. Thus, if you know the prototype you can usually properly call the function and use its returning value (if it has one) without ever seeing the definition of the code.

Adapted from: "Standard Libraries" by Kenneth Leroy Busbee is licensed under CC BY 4.0

## CHAPTER OVERVIEW

# 2: C++ Basics

# 2.1: The "main" concern....

## Basics of C++

C++ is a general-purpose programming language and widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platform like Windows, Linux, Unix, Mac, etc. C++ is  also an object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function. All of these pieces will be discussed in this course.

Learning C++ programming can be simplified into a few steps:

- Writing your program in a text-editor and saving it with correct extension(.CPP, .C, .CP)
- Compiling your program using a compiler or online IDE
- Understanding the basic terminologies.

The “Hello World” program has become the traditional first program in many programming courses. All the code does is display the message “Hello World” on the screen.So, to help us begin our journey here is the code:

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include <iostream>
using namespace std;

// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    cout << "Hello World" << endl;

    return 0;
 }
```

Lets take a look at this code and learn a bit about the terminology we will be using in this course:

1. // **Simple C++ program to display “Hello World”** : This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic. When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with // is a C++ comment **OR** multi line comments can be in between /* **and** */  .
2. **#include**: In C++,  all lines that start with pound (#) sign are called directives and are processed by preprocessor which is a program invoked by the compiler. We have discussed these concepts in the previous chapter. The **#include** directive tells the compiler to include a file and **#include <iostream>** . It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.
3. **int main()**: This line is used to declare a function named “main”, this function returns an integer data type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. **So, every C++ program must have a main() function.**
4. **{ and }**: The opening braces ‘{‘ indicates the beginning of the main function and the closing braces ‘}’ indicates the ending of the main function. Everything between these two comprises the body of the main function. Braces enclose a code block, we will see more of these as we go along.
5. **cout** << **“Hello World”** << **endl;**:  This line tells the compiler to display the message “Hello World” on the screen. This line is called a statement in C++, specifically this is an output statement. Every statement is meant to perform some task. A semi-colon

‘;’ is used to end a statement. Semi-colon character at the end of statement is used to indicate that the statement is ending there. The std::cout is used to identify the standard cout function, we will talk a bit more about this shortly. Everything followed by the character “<<” is displayed to the output device. Notice there are multiple << in the statement. Users can build complex output statements in this manner.

6. **return 0;** : This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.
7. **Indentation**: As you can see the cout and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance seems but as the programs become more complex, it makes the code more readable, less error-prone. Therefore, you must always use indentations and comments to make the code more readable.

It is key that you get the fact that EVERY C++ program needs a main() function. Without main() your code won't even compile. Main should always be declared as returning an int, but it is allowable to return a different type, such as void.

Adapted from:

"Writing first C++ program : Hello World example" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

---

# 2.2: The Importance of Good Looking Code

## Why Looking Good is Important

There is a contest - *The International Obfuscated C Code Contest* - who's goal is to: 1) to write the most Obscure/Obfuscated C program within the rules; 2) to show the importance of programming style, in an ironic way; 3) to stress C compilers with unusual code; 4) to illustrate some of the subtleties of the C language; and 5) to provide a safe forum for poor C code. :-). The term *obfuscate* means to make something obscure, not easy to understand. Go and look at their web site - the code is there. This is how you should NOT!! write your code. Code written to be obscure will not get you very far as a software developer, so lets set down some ground rules for this class.

As a software professional you sit in front of a computer day after day crunching out code. A true professional will take pride in their work, and ensure that the code is more than just some characters typed into a file that gets the job done. It takes a true "code master" to write code with style. As affirmed by the Obfuscated code contest, we can write code that does something, and at the same time is difficult to read, almost impossible for anyone else but the original programmer to debug, and is a maintenance nightmare.

If you continue to pursue the world of software engineering as career, you will find opportunities for promotions as you gain experience. The code you worked on ever since you started your current position will now become the responsibility of someone else to continue to develop and to maintain once the project is released for public use. If you wrote good code, followed some of the ideas we are going to talk about, that person will be grateful for your extra effort to write readable, understandable code.

Writing good code has several concepts that will discuss in the next few pages. These concepts will be required in all the code you write this class as well:

- Documentation
- Decomposition
- Naming
- Use of the language
- Formatting

# 2.3: Comments are Required

## You Have to Make a Comment

Comments are VERY useful in attempting to read someone else's code. There are many different methods for writing comments, and for the most part it depends on what your organization requires. Most organizations have "coding standards", a document that explains exactly how ALL software developed by any programmer working for that organization is to look.

As previously mentioned, comments have various formats depending on your organizations requirements.

- **Explain how a function is used**. Provide a brief explanation of what the function does. This simply states what the function does, what the argument list is, and what the function returns. Notice the multi-line format of the comment. The asterisk before each line is not required, but this is a common format. Usually, unless the code in this function is overly complex, you do not have to go into detail about what the function does.

```
/*
* Read an image from disk.
*
* fileName the file to read. Must be either absolute or relative to
*     the program working directory.
*
* return the image stored in `fileName`. If the image on disk does not
*     have `double` pixels, they will be cast to `double`.
*
* throws IoError Thrown if `fileName` does not exist or is not readable.
*/
lsst::afw::image::Image<double> loadImage(std::string const & fileName);
```

- **Explain complicated code** - I have NO idea what this comment even says, but am sure the code is pretty gnarly.

```
/*
 * The FFT is a fast implementation of the discrete Fourier transform:
 * @f[ X(e^{j\omega } ) = x(n)e^{ - j\omega n} @f]
 */
```

- **Communicate meta-information** - meta-information is information about the overall file or project...not specifically about the code. You usually see such comments at the beginning of the main packages of a large project.

```
/*
 * This file is part of Telescope Directional Drive package.
 *
 * Developed for the LSST Data Management System.
 * This product includes software developed by the LSST Project
 * (https://www.lsst.org).
 * See the COPYRIGHT file at the top-level directory of this distribution
 * for details of code ownership.
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */
```

- **Ad-hoc Comments**. Simple comments, this does not really add anything to the code, unless it is required by documentation standards should not be included.

```
// Sum two numbers.
double add(double a, double b);

These next two are the same - comments can be on the same line as the code.
Anything from the // to the end of the line is a comment
/// Flag set if background subtraction should not be done.
const int NO_BACKGROUND = 1 << 3;
OR
const int NO_BACKGROUND = 1 << 3;        ///< Skip background subtraction
```

Documentation examples: "Documenting C++ Code" by LSST DM staff, Legacy Survey of Space and Time , Rubin Observatory is licensed under CC BY 4.0

# 2.4: Do I Need to Decompose a Bit

## Decomposition

Decomposition is the concept of breaking code up into small chunks of code in order to make them more easily understood. When you open up some source code to find a 300-line function and huge, nested blocks of code it is a bit overwhelming. In an ideal coding world, each function or method should accomplish a single task. If a subtasks is of significant complexity it should be broken into multiple functions or methods. Let's say somebody asks you what a function or method your coded does and your answer is something like, “First it does this, then it does that; then, if one thing is true, it does A; otherwise, it does B,” you should probably have separate helper methods for This, That, One-thing, A and B.

Decomposition does not have a specific set of rules, by is guided by the code you are writing. You will find programmers who say that a function should not be longer than a page of printed code. That can be a good guide, but you can definitely find a quarter-page of code that is badly in need of decomposition.

For this class, we will focus on a few ideas:

1. Any function/method should do one thing. Once it has accomplished this one task it should end.
2. Once you have written a function/method and have decided it accomplishes its task, do NOT go back later and add something else to it just because it makes your life easier.
3. Look for standard library functions and class methods that already do what you are wanting to do - you may find that the problem has already been solved.

# 2.5: Naming Concepts

## Naming

Naming variables is a big deal. Anyone looking at your code should be able to understand what the variable is by the name you give it.

The C++ compiler has a few naming rules:

- Names cannot start with a number (for example, 9to5).
- Names that contain a double underscore (such as my__name) are reserved and shall not be used.
- Names that begin with an underscore (such as _name or _Name) are reserved and shall not be used.

Below are a few examples of good and bad naming.

| **GOOD NAMES** | **BAD NAMES** |
|---|---|
| `firstName, lastName`<br>Distinguishes two objects | `name1, name2`<br>Too general |
| `gSettings`<br>Conveys global status | `globalUserSpecificSettingsAndPreferences`<br>Too long |
| `calculateFTemp()`<br>Simple, specific | `calcTemp()`<br>Too general, unknown |
| `myTypeString`<br>Easy to read | `typeSTR256`<br>What does it mean??!! |
| `errorMessage`<br>Descriptive name | `myString`<br>Non-descriptive name |
| `sourceFile , destinationFile`<br>No abbreviations | `srcFile , dstFile`<br>Abbreviations |

The idea is to use names that make sense, that mean something in the context of the code, and that someone looking at your code doesn't have to spend an hour tracing things back through the code to figure it out.

One of the common places we see problems is in loops. Programmers love to use things like i or x as counting variables in loops. Don't be lazy, name it something simple like 'counter', or 'loopCount' - yes, it takes a bit more typing, but its easier to read and to understand.

Named Constants

If the your code reqiures the use of some mathematical or scientific constant it is best to define that variable as a constant instead of littering your code with some odd number. It would be confusing to see the number 299792458 scattered throughout your code, but if you were to declare a meaningful name

```
const int speedOfLight = 299792458;  // Speed of light in meters per second
```

it makes for much more readable code by using the variable name  in your code, instead of scattering some odd number all over your code.

## What Case Should I Use?

This is one of those questions that will get you responses about how you should only do it one way and never the other, and other people will tell you do it the other way and never the one way.

We are talking about what is known as snake case and camel case.

- in snake case words are separated by an underscore: first_name, last_name

- in camel case you use lowercase on the first word, and upper case on the first letter of all other words: firstName, lastName

As has been stated before, when you are writing code in the real world, whoever is paying you will usually have rules about how you will write your code. It is advisable that you follow their requirements if you enjoy your job.

## What should I do?

For naming in this class:

- All naming requirements for C++ must be followed
- Camel case is required for all variable, function and method names.
- Meaningful names are required - no use of single letter variable names unless it is a well known mathematical usage (i.e. $c^2 = a^2 + b^2$ you could use variables named a, b, and c in this case)

# 2.6: The Great Bracket Debate

## The Great Bracket Debate

It really isn't that big a deal - but it continues to be a point of disagreement across the software developers' world: Where do the curly brackets belong when I write my code. Once again, for you as a software developer in the world, someone else will probably make that decision for the project you are working one.

But, let us take a few moments to examine what the options are.

The following code is a common example of how people write the code. Notice that the first { is on the line AFTER the function definition. On the If/else statements the opening bracket is at the end of the line. Both of these are valid ways to code the brackets.

```
void someFunction()
 {
     if (condition()) {
         cout << "condition was true" << endl;
     } else {
         cout << "condition was false" << endl;
     }
 }
```

Another frequent formatting is that brackets are always on a line by themselves. Some say this make code easy to read and debug as you can see the matching brackets quite easily.

```
void someFunction()
 {
     if (condition())
     {
         cout << "condition was true" << endl;
     }
     else
     {
         cout << "condition was false" << endl;
     }
 }
```

The code below shows how some programmers seem to go make it difficult to read their code....this is a valid way to code, but does make it a bit it more difficult to read.

```
void someFunction()
 {
     if (condition())
        {
            cout << "condition was true" << endl;
        }
     else
        {
            cout << "condition was false" << endl;
        }
}
```

For this class, I will not tell you which method to use, but I suggest you NOT make it difficult for me or my grader to grade your code...do so may cost you points!!

# 2.7: Spaces and Such

## Readability

It may seem a bit picky, but you will be required to use appropriate spaces to improve the readability of your code.

The two lines of code are the same:

```
if(myAgeValue>60 && myCarSpeed>75) {
   myInsuranceRate=i+k/m;
}

if (myAgeValue > 60 && myCarSpeed > 75) {
   myInsuranceRate = i + k / m;              //Yes - this violates the meaningful var
}
```

Both of these statements are valid C++ code, both will work. The first one is just much more difficult to read, because there are very few spaces in the code. To help those who are having to read the code, and perhaps change it or maintain it, add spaces after every key word, before and after a variable name, before and after any operator (=, <, etc).

## One more...

There is the issue of parenthesis, they can be your friend, use them to not only make sure the compiler knows what you want but also those looking at the code can be sure you know what you are doing.

```
   myInsuranceRate = i + k / m;              //Yes - this violates the meaningful var
   myInsuranceRate = i + (k / m);              //Yes - this violates the meaningful va
```

Again - both of these statements work. The mathematical order of precedence says left to right, multiplication, division, addition then subtraction. Put the parenthesis around the division to show that is what you intended to happen first.  Do not simply assume that everyone understands what you are doing.

It is about being professional and taking the time and effort to write good code that is readable and understandable. After 40 years in the software development world, I know this to be true - and you will find promotions and bonuses easier to come by if you do a few simple things.

# CHAPTER OVERVIEW

## 3: Program Planning and Design

# 3.1: What is Software Development

While software development is a very important topic, we will not delve very deep into this concept in the class. However, it is important to understand what software design is.

Over many years there have been several different software design models. One model, that seemed to catch on, was the waterfall model.

## The Waterfall Model

The first formal description of this method is often cited as an article published by Winston W. Royce in 1970 although Royce did not use the term "waterfall" in this article. Royce presented this model as an example of a flawed, non-working model.

The basic principles are:

- Project is divided into sequential phases, with some overlap and splashback acceptable between phases.
- Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time.
- Tight control is maintained over the life of the project via extensive written documentation, formal reviews, and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase. Written documentation is an explicit deliverable of each phase.

The waterfall model is a traditional engineering approach applied to software engineering. A strict waterfall approach discourages revisiting and revising any prior phase once it is complete. This "inflexibility" in a pure waterfall model has been a source of criticism by supporters of other more "flexible" models. It has been widely blamed for several large-scale government projects running over budget, over time and sometimes failing to deliver on requirements. Except when contractually required, the waterfall model has been largely superseded by more flexible and versatile methodologies developed specifically for software development.

Part of the problem with the waterfall model was that there was no process to stop and go back to a previous step if it was discovered that something wasn't quite right. Developers had to go all the way to the end of the process before they could return to the first step, and work all the way through to resolve issues. This can create long time frames for the development of software. At this point in history, software projects were long - 12 to 18 months - was not uncommon, even longer if the project got delayed.

As software development became more refined over time, and learning from previous models of development,various other models were introduced and used at various points in the entire history of software development.

## Agile vs. waterfall

One of the differences between agile software development methods and waterfall is the approach to quality and testing. In the waterfall model, there is always a separate ***testing phase*** after a ***build phase***; however, in agile software development testing is completed in the same iteration as programming.

Another difference is that traditional "waterfall" software development moves a project through various Software Development Lifecycle (SDLC) phases. One phase is completed in its entirety before moving on to the next phase.

Because testing is done in every iteration—which develops a small piece of the software—users can frequently use those new pieces of software and validate the value. After the users know the real value of the updated piece of software, they can make better decisions about the software's future. Having a value retrospective and software re-planning session in each iteration—Scrum typically has iterations of just two weeks—helps the team continuously adapt its plans so as to maximize the value it delivers. This follows a pattern similar to the PDCA cycle, as the work is *planned*, *done*, *checked* (in the review and retrospective), and any changes agreed are *acted* upon.

This iterative approach supports a *product* rather than a *project* mindset. This provides greater flexibility throughout the development process; whereas on projects the requirements are defined and locked down from the very beginning, making it difficult to change them later. Iterative product development allows the software to evolve in response to changes in business environment or market requirements.

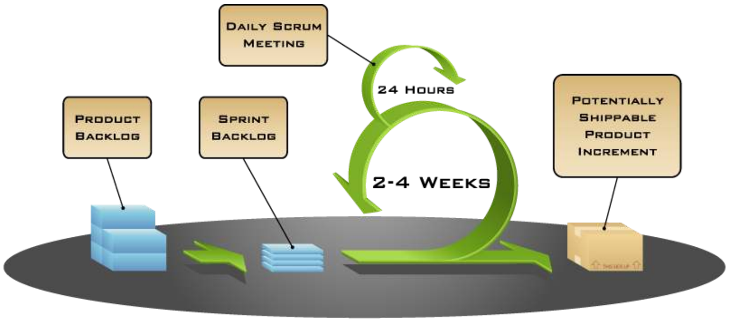


Figure 3.1.1: Scrum Diagram. ("File:Scrum diagram" by Multiple Contributors, Wikimedia is licensed under CC BY-SA 2.5)

Notice that in this model the sprint cycle is 1 to 4 weeks, not the months we saw with the waterfall model. The end date for the sprint and what will be delivered at the end of the sprint does NOT change. Also, there is a daily review, where changes can be made in order to stay on track to provide the agreed upon solution by the end of the sprint.

This model is used widely in the software industry today, because it is flexible, it is allows for quick changes, and a short time frame.

Adapted from:
"Software development process" by Various contributors, Wikipedia is licensed under CC BY 4.0
"Agile software development" by Various contributors, Wikipedia is licensed under CC BY 4.0

# 3.2: Why is Design Important

## Why Design?

So often, especially as we are learning software development, we tend to look at a problem and simply start coding. That is a very poor habit that almost every student falls into. We should first think about what the problem is we are attempting to solve, what are the requirements we are given, how can we break the problem into the smallest pieces that make sense.

Why do we need to worry about design? As a software developer throughout my career, I get it that writing code is a whole lot more fun than writing some design document that is going to tell me what I have to code. However, as we craft our code we must know what we are doing, where we are going, and when we have arrived. This becomes even more important when we begin to develop code with a team, where each person is working on a different portion of the project, and the pieces all need to work together.

Let's take building a house as an example. I have paid contractors on a couple of different occasions to build a house for me. He always shows me the blueprints, explains how all the rooms are situated in accordance with one another, how the electrical wiring is going to be run, how the plumbing all ties together. All the components are defined as to location and purpose. I would never let a contractor tell me, "I have done lots of houses, I know how this all works. We will start from the foundation and just build it up from there." That is a recipe for disaster. What happens if they forget to put in the bathroom - they have to tear down some of their work and redo it so that it is correct according to what is was suppose to be.

Let's say the builder shows up, and you ask to take a look at the blueprints. “I don't need any blueprints?” he responds. “I have been doing this a while and I know what I’m doing. I don’t need to plan every little detail ahead of time. This is a two-story house - right? Not a problem - I just completed a one-story house last week, I’ll just start with that idea and work from there.”

So, let's just say you are able to get past your amazement and allow the builder to begin building your house. As he progresses with the project you begin to notice things aren't quite right, the plumbing appears to run outside the house instead of inside the walls. You immediately question the contractor about this oddity. He replies, “Oh - yeah. I sorta forgot to leave space in the walls for the plumbing. I got all excited about this new drywall technology that I completely spaced out the plumbing. But hey, it works just as well outside, and that's what matters, right?".

As you walk through the completed home, you notice that the kitchen lacks a sink. The builder again excuses himself by saying, “We were almost done with the kitchen when we noticed there wasn’t enough room for the sink. Instead of starting over, we just added a separate sink room over here. That works okay, right?”

You would never allow that to happen when building a house - and we should not make similar mistakes with the software we are building. Even simple programs need to be deliberately designed so that we don't have to go back and redo portions of the code because we forgot something. We will take a few pages to discuss some design concepts and ideas, before we start to talk about coding.

# 3.3: Program Design

## Design Introduction

Program Design consists of the steps a programmer should do before they start coding the program in a specific language. These steps when properly documented will make the completed program easier for other programmers to maintain in the future. There are three broad areas of activity:

- Understanding the Program
- Using Design Tools to Create a Model
- Develop Test Data

## Understanding the Program

If you are working on a project as a one of many programmers, the system analyst may have created a variety of documentation items that will help you understand what the program is to do. These could include screen layouts, narrative descriptions, documentation showing the processing steps, etc. If you are not on a project and you are creating a simple program you might be given only a simple description of the purpose of the program. Understanding the purpose of a program usually involves understanding it's:

- Inputs
- Processing
- Outputs

This **IPO** approach works very well for beginning programmers. Sometimes, it might help to visualize the programming running on the computer. You can imagine what the monitor will look like, what the user must enter on the keyboard and what processing or manipulations will be done.

## Using Design Tools to Create a Model

At first you will not need a hierarchy chart because your first programs will not be complex. But as they grow and become more complex, you will divide your program into several modules (or functions).

The first modeling tool you may have learned is **pseudocode** (we will talk a bit more about this in a moment)**.** This allows you to document the logic or algorithm of each function in your program. At first, you usually have only one function, and thus your pseudocode will follow closely the IPO approach mentioned above.

There are several methods or tools for planning the logic of a program. They include: flowcharting, hierarchy or structure charts, pseudocode, HIPO, Nassi-Schneiderman charts, Warnier-Orr diagrams, etc. Some classes make you do flowcharting and pseudocode. While these can be useful, in my 35+ years of working in the computer industry there was only one project where any type of flowchart was attempted. Several standards exist for flowcharting and pseudocode and most are very similar to each other. However, most companies have their own documentation standards and styles. Programmers are expected to be able to quickly adapt to whatever software development standards the company they work for uses. The others methods that are less universal require some training which is generally provided by the employer that chooses to use them.

Understanding the logic and planning the algorithm on paper before you start to code is very important concept. Many students develop poor habits and skipping this step is one of them. A good concept that I have found useful is to write simple comments as to how the code should flow - this is somewhat of a psuedocode approach, but you kind of develop your own terminology and ideas.

It is important to think about your code BEFORE you start writing it....

## Definitions

(click on each word)

**IPO**

Inputs - Processing - Outputs

**Psuedocode**

English-like statements used to convey the steps of an algorithm or function.

**Test Data**

Providing input values and predicting the outputs.

Adapted from: "Program Design" by Kenneth Leroy Busbee is licensed under CC BY 4.0

1.

# 3.4: Psuedocode

## Overview

Pseudocode is one method of designing or planning a program. It is used by a lot of instructors to teach software design. In all my years as a software developer I never used pseudocode and therefore do not teach it. It is covered briefly here in case you run into this concept at some point in your coding career.

**Pseudo** means false, thus pseudocode means false code. A better translation would be the word fake or imitation. Pseudocode is fake (not the real thing). It looks like (imitates) real code but it is NOT real code. It uses English statements to describe what a program is to accomplish. It is fake because no compiler exists that will translate the pseudocode to any machine language. Pseudocode is used for documenting the program or module design (also known as the algorithm).

The following outline of a simple program illustrates pseudocode. We want to be able to enter the ages of two people and have the computer calculate their average age and display the answer.

```
Input
  display a message asking the user to enter the first age
  get the first age from the keyboard
  display a message asking the user to enter the second age
  get the second age from the keyboard
Processing
  calculate the answer by adding the two ages together and dividing by two
Output
  display the answer on the screen
  pause so the user can see the answer
```

After developing the program design, we use the pseudocode to write code in a language (like Pascal, COBOL, FORTRAN, "C", " C++", etc.) where you must follow the rules of the language (syntax) in order to code the logic or algorithm presented in the pseudocode. Pseudocode usually does not include other items produced during programming design such as identifier lists for variables or test data.

As I have stated, I do not spend much time on pseudocode, instead I like the idea of simply creating your comments, and then come back and actually write your code. For simple programs this looks overly simple, but for more complex code involving functions it is helps a lot to track the flow of your code.

```
// Preprocessor directives

int main()
  // Ask the user for the first age
  // Accept input into integer variable age1

  // Ask the user for the second age
  // Accept input into integer variable age2

  // Calculate average, and assign to integer variable: myAnswer = (age1 + age2) / 2

  // Output the results
```

## Definitions

Adapted from: "Pseudocode" by Kenneth Leroy Busbee is licensed under CC BY 4.0 (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2)

**pseudo**

Means false and includes the concepts of fake or imitation.

# 3.5: Test Data

## Overview

Test data consists of the user providing some input values and predicting the outputs. This can be quite easy for a simple program and the test data can be used twice.

1. to check the model to see if it produces the correct results (**model checking**)
2. to check the coded program to see if it produces the correct results (**code checking**)

Test data is developed by using the algorithm of the program. This algorithm is usually documented during the program design with either flowcharting or pseudocode. Here is the comment method I described in the previous section describing the flow - the IPO - for a program used for painting rectangular buildings.

```
// Preprocessor directives
// Namespace

int main()
  // ask user for the length of the building
  // accept length into variable bldgLength
  // ask user for the width of the building
  // accept width into variable bldgWidth
  // ask user for the height of the building
  // accept height iunto variable bldgHeight
  // ask user for the price per gallon of paint
  // accept price per gallon of paint into variable pricePerGal
  // ask user for the sq ft coverage of a gallon of paint
  // accept sq ft coverage of a gallon of paint into variable coveragePerGal

   // calculate the total area of the building:
      // area1 = (bldgLength * bldgHeight) *2
      // area2 = (bldgWidth * bldgHeight) *2
      // bldgArea = area1 + area2            You could combine all of this into one s
  // calculate the number of gallons of paint needed:
      // totalGallons = bldgArea / coveragePerGal
      // round totalGallons up to the next whole gallon     Can do this using the C++
  // paintCost =  totalGallons * pricePerGal

  // output the number of gallons needed
  // display the total cost of the paint
```

## Creating Test Data and Model Checking

Test data is used to verify that the inputs, processing and outputs are working correctly. As test data is initially developed it can verify that the documented algorithm (comment code in the example we are doing) is correct. It helps us understand and even visualize the inputs, processing and outputs of the program.

Inputs: The building is 100 feet long by 40 feet wide and 10 feet in height and I selected paint costing $28.49 per gallon that will cover 250 square feet per gallon. We should verify that the comment code is prompting the user for this data.

Processing: Using my solar powered hand held calculator, I can calculate (or predict) the total area would be: (100 x 10 x 2 plus 40 x 10 x 2) or 2,800 sq ft. The total gallons of paint would be: (2800 / 250) or 11.2 gallons. But rounded up, I would need twelve (12) gallons of paint. The total cost would be: (28.49 times 12) or $341.88. We should verify that the comment code is performing the correct calculations.

Output: Only the significant information (number of gallons to buy and the total cost) are displayed for the user to see. We should verify that the appropriate information is being displayed.

## Testing the Coded Program - Code Checking

The test data can be developed and used to test the algorithm that is documented (in our case our comment code) during the program design phase. Once the program is code with compiler and linker errors resolved, the programmer gets to play user and should test the program using the test data developed. When you run your program, how will you know that it is working properly? Did you properly plan your logic to accomplish your purpose? Even if your plan was correct, did it get converted correctly (coded) into the chosen programming language (in our case C++)? The answer (or solution) to all of these questions is our test data.

By developing test data we are predicting what the results should be, thus we can verify that our program is working properly. When we run the program we would enter the input values used in our test data. Hopefully the program will output the predicted values. If not then our problem could be any of the following:

1. The plan (IPO outline or other item) could be wrong
2. The conversion of the plan to code might be wrong
3. The test data results were calculated wrong

Resolving problems of this nature can be the most difficult problems a programmer encounters. You must review each of the above to determine where the error is lies. Fix the error and re-test your program.

## Definitions

Adapted from: "Test Data" by Kenneth Leroy Busbee is licensed under CC BY 4.0

**Model Checking**

Using test data to check the design model (usually done in pseudocode).

**Code Checking**

Using test data to check the coded program in a specific language (like C++).

## 3.6: Two Good Ideas to Keep in Mind

What we do want to focus on in C++ are two concepts: 1) Abstraction; and 2) Reuse. These are 2 concepts that we want to get in the habit of thinking about BEFORE we start writing code. You may have to do some searching, "Does C++ provide a way to calculate the square root of a number?" - you would find that yes it does, and therefore you do not have to write the code to determine a square root. As we learn more about object oriented programming, we will get better at the abstraction piece of this.

### Abstraction

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

**Abstraction using Classes:** We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

**Abstraction in Header files:** One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class, can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

**Advantages of Data Abstraction**:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

### Reuse

Code reuse aims to save time and resources and reduce redundancy by taking advantage of assets that have already been created in some form within the software product development process. The key idea in reuse is that parts of a computer program written at one time can be or should be used in the construction of other programs written at a later time.

Code reuse may imply the creation of a separately maintained version of the reusable assets. While code is the most common resource selected for reuse, other assets generated during the development cycle may offer opportunities for reuse: software components, test suites, designs, documentation, and so on.

The software library is a good example of code reuse. Programmers may decide to create internal abstractions so that certain parts of their program can be reused, or may create custom libraries for their own use.

For newly written code to use a piece of existing code, some kind of interface, or means of communication, must be defined. These commonly include a "call" or use of a subroutine, object, class, or prototype. In organizations, such practices are formalized and standardized by domain engineering, also known as software product line engineering.

The general practice of using a prior version of an extant program as a starting point for the next version, is also a form of code reuse.

Some so-called code "reuse" involves simply copying some or all of the code from an existing program into a new one. While organizations can realize time to market benefits for a new product with this approach, they can subsequently be saddled with many of the same code duplication problems caused by cut and paste programming.

Many researchers have worked to make reuse faster, easier, more systematic, and an integral part of the normal process of programming. These are some of the main goals behind the invention of object-oriented programming, which became one of the most common forms of formalized reuse. A somewhat later invention is generic programming.

Another, newer means is to use software "generators", programs which can create new programs of a certain type, based on a set of parameters that users choose.

Adapted from: "Abstraction in C++" by Harsh Agarwal, Geeks for Geeks
and: "Code reuse" by Various contributors, Wikipedia

## CHAPTER OVERVIEW

# 4: Data and Operators

# 4.1: We Begin to Code...

As we step into this module we will start to see examples of C++ code. I wanted to give just a brief example so we can also start to write some code.

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

int main()
{
   int newVal = 99;
   // cout is just a way we can get output to the screen
   // we will talk more about this later.
   cout << "This is output" << newVal << endl;

   return 0;
}
```

For your first few program you can use this code to get started. We have already discussed the preprocessor line (the #include line). There is a brief discussion of the namespace entry just below. You may not understand all of the coding syntax, but you should be able to understand the namespace concept

C++ requires every program to have a main() function, it returns an integer value - often times you will see it returns 0.

The term "cout", is used to output data to the screen. the << is separates the different values to output: anything between "" is a string value; we can specify variable names; and we can send an "end of line" with the endl.

## Namespace in C++

**Definition and Creation:**

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

- Namespace is a feature added in C++ and not present in C.
- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
- Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
   int x, y; // code declarations where
             // x and y are declared in
             // namespace_name's scope
}
```

- Namespace declarations appear only at global scope. (we will talk about scope later)
- Namespace declarations can be nested within another namespace.
- Namespace declarations don’t have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.
- We can split the definition of namespace over several units.

```
// Creating namespaces
#include <iostream>
using namespace std;

namespace ns1
{
    int value() { return 5; }
}
namespace ns2
{
    const double x = 100;
    double value() { return 2*x; }
}

int main()
{
    // Access value function within ns1
    cout << ns1::value() << endl;

    // Access value function within ns2
    cout << ns2::value() << endl;

    // Access variable x directly
    cout << ns2::x << endl;

    return 0;
}
```

The statement "using namespace std" tells C++ that we are using the standard C++ namespace, so that we can use statements like cout without having to specifically code std::cout or std::endl.

We also define two other namespaces for our own variables, ns1 and ns2.  When we use these variables/functions, we are required to prefix the variables and functions with the namespace name. In the example above we have ns1 we have an function that returns an integer named value(), in ns2 we also have a function named value(), which returns a double. These are duplicate names, BUT, are defined in different namespaces and therefore operate as separate functions.

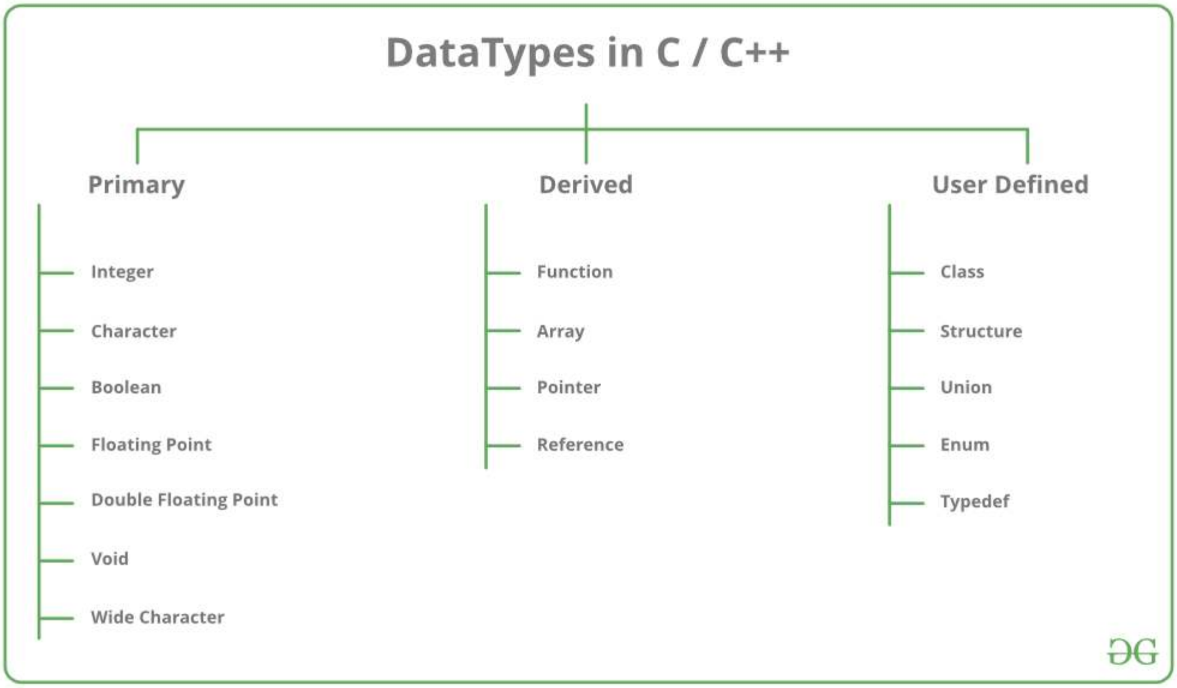
Using namespaces is very helpful in large projects where we have lots of pieces and several different teams working on different pieces of the project. We can create variables and functions and not have to worry about duplicate names, just need to specify which namespace we are using in each situation.

Adapted from:
"Namespace in C++ | Set 1 (Introduction)" by Abhinav Tiwari, Geeks for Geeks is licensed under CC BY-SA 4.0

# 4.2: Data Types in C++

## C++ Data Types

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.



Data types in C++ is mainly divided into three types:

1. **Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:
   - Integer - a whole number, no fractional part
   - Character - any printable character (some languages require a wide character)
   - Boolean - a value that is either True or False, True = 1, False = 0
   - Floating Point - a numerical value that has a fractional portion
   - Double Floating Point - a floating point value that contains greater precision due to the internal size (number of bytes)
   - Valueless or Void - meaning a empty value
   - Wide Character - certain languages require more bytes to represent all of the possible characters
2. **Derived Data Types:** The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
   - Function
   - Array
   - Pointer
   - Reference
3. **Abstract or User-Defined Data Types**: These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
   - Class
   - Structure
   - Union
   - Enumeration

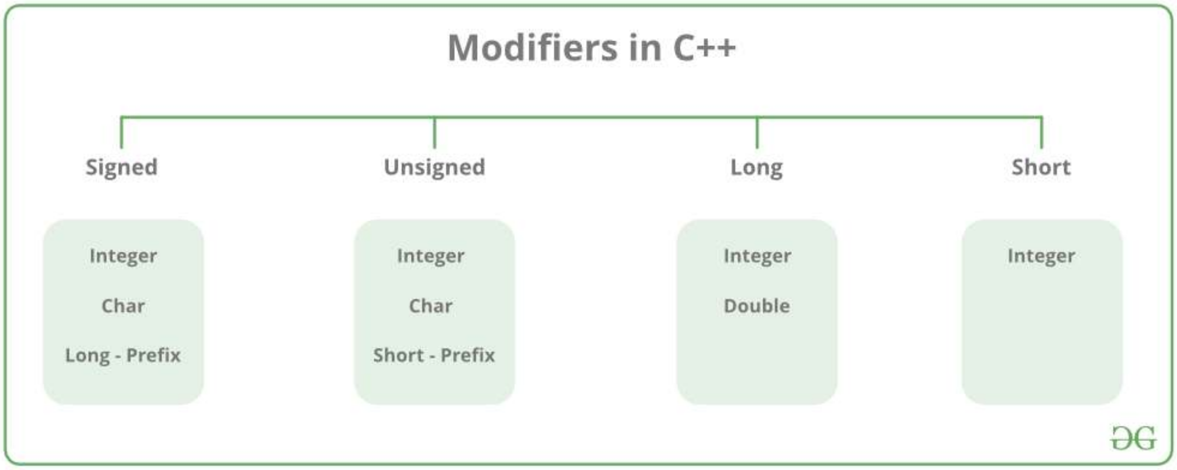
- Typedef defined DataType

This article discusses **primitive data types** available in C++.

- **Integer**: Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
- **Character**: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
- **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. Keyword used for boolean data type is **bool**.
- **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.
- **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.
- **void**: Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
- **Wide Character**: Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar_t**. It is generally 2 or 4 bytes long.

**Datatype Modifiers**

As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

- **Signed**
- **Unsigned**
- **Short**
- **Long**

Below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

| DATA TYPE | SIZE (IN BYTES) | RANGE |
|---|---|---|
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |

| | | |
|---|---|---|
| long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |
| long long int | 8 | -(2^63) to (2^63)-1 |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 |
| signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| float | 4 | 1.17549e-038 to 3.40282e+038 |
| double | 8 | 2.22507e-308 to 1.79769e+308 |
| long double | 12 | 2.22507e-308 to 1.79769e+308 |
| wchar_t | 2 or 4 | 1 wide character |

**Note** : Above values may vary from compiler to compiler. In above example, we have considered GCC 64 bit.

Adapted from: "C++ Data Types" by Harsh Agarwal, Geeks for Geeks

# 4.3: Identifier Names

## Identifier Names

Within programming a variety of items are given descriptive names to make the code more meaningful to us as humans. These names are called "Identifier Names". Constants, variables, type definitions, functions, etc. when declared or defined are identified by a name. These names follow a set of rules that are imposed by:

- the language's technical limitations
- good programming practices

## Technical to Language

An *identifier* is an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters, and most Unicode characters (see below for details). A valid identifier must begin with a non-digit character. Identifiers are case-sensitive (lowercase and uppercase letters are distinct), and every character is significant.

There are certain rules that should be followed while naming c identifiers:

- They must begin with a letter or underscore(_).
    - the identifiers with a double underscore anywhere are reserved;
    - the identifiers that begin with an underscore followed by an uppercase letter are reserved;
    - the identifiers that begin with an underscore are reserved in the global namespace.
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only first 31 characters are significant.

Some examples of C++ identifiers:

| NAME | REMARK |
|---|---|
| _A9 | Valid |
| Temp.var | Invalid as it contains special character other than the underscore |
| void | Invalid as it is a keyword |

These attributes are specific to C++ - the requirements vary from one programming language to another. The allowable characters and reserved words will be different. The length limit refers to how many characters are allowed in an identifier name and often is compiler dependent and may vary from compiler to compiler for the same language. However, all programming languages have these three technical rules.

## Good Programming Techniques

- Meaningful
- Be case consistent - making sure you are coding according to the standards your organization has in place
- CONSTANTS IN ALL UPPER CASE - not a requirements - but pretty much an industry standard concept

Meaningful identifier names make your code easier for another to understand. After all what does "p" mean? Is it pi, price, pennies, etc. Thus do not use cryptic (look it up in the dictionary) identifier names. This has been discussed previously - you will lose points in my class if you use variables that are NOT meaningful.

Some programming languages treat upper and lower case letters used in identifier names as the same. Be sure you know the coding requirements for whatever organization you are coding for.

## Definitions

**reserved word**

Words that cannot be used by the programmer as identifier names because they already have a specific meaning within the programming language.

Adapted from: "C/C++ Tokens" by I.HARISH KUMAR, Geeks for Geeks
"Identifiers" by Cubbi, cppreference.com
"Identifier Names" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 4.4: Constants and Variables

## Understanding Constants vs Variables

Various textbooks describe constants using different terminology. Added to the complexity are the explanations from various industry professionals will vary greatly. Let's see if we can clear it up.

A **constant** is a data item whose value cannot change during the program's execution. Thus, as its name implies – their value is constant.

A **variable** is a data item whose value can change during the program's execution. Thus, as its name implies – their value can vary.

Constants are used in two ways within C++. They are:

1. defined constant
2. memory constant

Most text books refer to either symbolic constants or named constants but these two refer to the same concept. A symbolic constant is represented by a name similar to how we name variables. Let's say it backwards; the identifier name is the symbol that represents the data item. Within C++ identifier names have some rules. One of the rules says those names should be meaningful. Another rule about using ALL CAPS FOR CONSTANTS is an industry rule. There are two ways to create symbolic or named constants:

```
#define PI 3.14159
```

Called a **defined constant**, we have discussed this concept when we talked about preprocessor directives in Lesson 1**,** because it uses a textual substitution method controlled by the compiler pre-processor command word "define".

```
const double PI = 3.14159;
```

The second one is called sometimes called **constant variable** but that name is contradictory all by itself. How can it be constant and vary at the same time? The better name for the second one is a **memory constant** because they have a "specific storage location in memory".

As the name suggests the name constants is given to such variables or values in C++ programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any types of constants like integer, float, octal, hexadecimal, character constants etc. Every constant has some range. The integers that are too big to fit into an int will be taken as long. Now there are various ranges that differ from unsigned to signed bits. Under the signed bit, the range of an int varies from -128 to +127 and under the unsigned bit, int varies from 0 to 255.

## Defining Constants:

In C/C++ program we can define constants in two ways as shown below:

1. Using *#define* preprocessor directive
2. Using a *const* keyword

Let us now learn about above two ways in details:

1. **Using *#define* preprocessor directive:** This directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:

   ```
   #define identifierName value
   ```

   - **identifierName:** It is the name given to constant.
   - **value:** This refers to any value assigned to identifierName.

   **Example**

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

```
#define val 10
#define floatVal 4.5
#define charVal 'G'

int main() {
        cout << "Integer Constant: " << val << "\n";
        cout << "Floating point Constant: " << floatVal << "\n";
        cout << "Character Constant: "<< charVal << "\n";

        return 0;
}
```

**Output:**

```
Integer Constant: 10
Floating point Constant: 4.5
Character Constant: G
```

Refer Macros and Preprocessors in C for details.

2. **using a *const* keyword**: Using *const* keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a *const* keyword.

You must set a value on a const variable at the time you create it:

```
RIGHT: You MUST set the value at the time of creation
const int myAmount = 10;

WRONG: you can not attempt to set a const variable after it is defined
const int myAmount;
myAmount = 10;
```

Below program shows how to use const to declare constants of different data types:

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

int main() {
        // int constant
        const int intVal = 10;

        // Real constant
        const float floatVal = 4.14;

        // char constant
        const char charVal = 'A';

        // string constant
        const string stringVal = "ABC";
        cout << "Integer Constant: " << intVal << "\n";
        cout << "Floating point Constant: " << floatVal << "\n";
        cout << "Character Constant: "<< charVal << "\n";
        cout << "String Constant: "<< stringVal << "\n";

        return 0;
}
```

**Output:**

```
Integer constant: 10
Floating point constant: 4.14
Character constant: A
String constant: ABC
```

## Limits

Each of the different data types have upper and lower limits depending on the data type, the processor and the compiler. The C++ standard has a defined minimum and maximum for all the different data types.

The **limits.h** header determines various properties of the various variable types. The macros defined in this header, limits the values of various variable types like char, int and long.

These limits specify that a variable cannot store any value beyond these limits, for example an unsigned character can store up to a maximum value of 255.

### Library Macros

The following values are implementation-specific and defined with the #define directive, but these values may not be any lower than what is given here.

| Macro | Value | Description |
|---|---|---|
| CHAR_BIT | 8 | Defines the number of bits in a byte. |
| SCHAR_MIN | -128 | Defines the minimum value for a signed char. |
| SCHAR_MAX | +127 | Defines the maximum value for a signed char. |
| UCHAR_MAX | 255 | Defines the maximum value for an unsigned char. |
| CHAR_MIN | -128 | Defines the minimum value for type char and its value will be equal to SCHAR_MIN if char represents negative values, otherwise zero. |
| CHAR_MAX | +127 | Defines the value for type char and its value will be equal to SCHAR_MAX if char represents negative values, otherwise UCHAR_MAX. |
| MB_LEN_MAX | 16 | Defines the maximum number of bytes in a multi-byte character. |
| SHRT_MIN | -32768 | Defines the minimum value for a short int. |
| SHRT_MAX | +32767 | Defines the maximum value for a short int. |
| USHRT_MAX | 65535 | Defines the maximum value for an unsigned short int. |
| INT_MIN | -2147483648 | Defines the minimum value for an int. |
| INT_MAX | +2147483647 | Defines the maximum value for an int. |

| | | |
|---|---|---|
| UINT_MAX | 4294967295 | Defines the maximum value for an unsigned int. |
| LONG_MIN | -9223372036854775808 | Defines the minimum value for a long int. |
| LONG_MAX | +9223372036854775807 | Defines the maximum value for a long int. |
| ULONG_MAX | 18446744073709551615 | Defines the maximum value for an unsigned long int. |

## Definitions

**Constant**

A data item whose value cannot change during the program's execution.

**Variable**

A data item whose value can change during the program's execution.

Adapted from:
"Constants in C/C++" by Chinmoy Lenka, Geeks for Geeks
"Understanding Constants" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 4.4.1: Literals and Constants - Integers

## Integer Literals

These are used to represent and store the integer values. Integer literals are expressed in two types i.e.,

1. **Prefixes:** The Prefix of the integer literal indicates the base in which it is to be read.
   **For example:**

   ```
   0x10 = 16

   Because 0x prefix represents a HexaDecimal base.
   So 10 in HexaDecimal is 16 in Decimal.
   Hence the value 16.
   ```

   There are basically represent in four types.

   1. **Decimal-literal(base 10):** A **non-zero decimal digit** followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

      **For example:**

      ```
      56, 78
      ```

   2. **Octal-literal(base 8):** a **0** followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7).

      **For example:**

      ```
      045, 076, 06210
      ```

   3. **Hex-literal(base 16):** **0x** or **0X** followed by one or more hexadecimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).

      **For example:**

      ```
      0x23A, 0Xb4C, 0xFEA
      ```

   4. **Binary-literal(base 2):** **0b** or **0B** followed by one or more binary digits(0, 1).

      **For example:**

      ```
      0b101, 0B111
      ```

2. **Suffixes:** The Prefix of the integer literal indicates the type in which it is to be read.

   **For example:**

   ```
   12345678901234LL
   indicates a long long integer value 12345678901234
   because of the suffix LL
   ```

   These are represented in many ways according to their data types.

   1. **int:** No suffix is required because integer constant is by default assigned as an int data type.
   2. **unsigned int:** character u or U at the end of an integer constant.
   3. **long int:** character l or L at the end of an integer constant.
   4. **unsigned long int:** character ul or UL at the end of an integer constant.

5. **long long int:** character ll or LL at the end of an integer constant.
6. **unsigned long long int:** character ull or ULL at the end of integer constant.

**Example:**

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

int main()
{   // constant integer literal

    const int intVal = 10;

    cout << "Integer Literal: " << intVal << "\n";
    return 0;
}
```

```
Integer Literal: 10
```

Adapted from:
"Types of Literals in C/C++ with Examples" by Chinmoy Lenka, Geeks for Geeks

# 4.4.2: Literals and Constants - Floating Point

## Floating-Point Literals

These are used to represent and store real numbers. The real number has an integer part, real part, fractional part and an exponential part. The floating-point literals can be stored either in decimal form or exponential form. While representing the floating-point decimals one must keep two things in mind to produce valid literals:

- In the decimal form, one must include the decimal point, exponent part or both, otherwise, it will lead to an error.
- In the exponential form, one must include the integer part, fractional part or both, otherwise, it will lead to an error.

Few floating-point literal representations are shown below:

**Valid Floating Literals:**

```
10.125
1.215-10L
10.5E-3
```

**Invalid Floating Literals:**

```
123E
1250f
0.e879
```

**Example:**

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

int main()
{   // Real literal
    const float floatVal = 4.14;

    cout << "Floating-point literal: " << floatVal << "\n";
    return 0;
}
```

**Output:**

```
Floating point literal: 4.14
```

Adapted from:
"Types of Literals in C/C++ with Examples" by Chinmoy Lenka, Geeks for Geeks

# 4.4.3: Literals and Constants - Characters

## Character Literal

This refers to the literals that are used to store a single character within a single quote. To store multiple characters, one needs to use a character array. Storing more than one character within a single quote will throw a warning and displays just the last character of the literal. It gives rise to the following two representations:

- **char type:** This used to store normal character literal or the narrow-character literals. This is supported by both C and C++.

  **Example:**

  ```
  // For C++
  char chr = 'G';
  ```

- **wchar_t type:** This literal is supported only in C++ and **not in C**. If the character is followed by L, then the literal needs to be stored in wchar_t. This represents wide-character literal.

  **Example:**

  ```
  // For C++
  wchar_t chr = L'G';
  ```

**Example:**

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

int main()
{   // constant char literal
    const char charVal = 'A';

    // wide char literal
    const wchar_t charVal1 = L'A';

    cout << "Character Literal: " << charVal << "\n";
    cout << "Wide_Character Literal: " << charVal1 << "\n";

    return 0;
}
```

**Output:**

```
Character Literal: A
```

**Escape Sequences:** There are various special characters that one can use to perform various operations.

Adapted from:
"Types of Literals in C/C++ with Examples" by Chinmoy Lenka, Geeks for Geeks

# 4.4.4: Literals and Constants - Strings and Booleans

1. **String Literals:** String literals are similar to that of the character literals, except that it can store multiple characters and uses a double quote to store the same. It can also accommodate the special characters and escape sequences mentioned in the table above.

   **Example:**

   ```
   // For C++
   string stringVal = "GeeksforGeeks"
   ```

   **Example:**

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

   int main()
   {
       const string str = "Welcome\nTo\nGeeks\tFor\tGeeks";
       cout << str;
       return 0;
   }
   ```

2. **Output:**

   ```
   Welcome
   To
   Geeks    For    Geeks
   ```

3. **Boolean Literals:** This literal is provided **only in C++** and **not in C**. They are used to represent the boolean datatypes. These can carry two values:
   - **true:** To represent True value. This must not be considered equal to int 1.
   - **false:** To represent False value. This must not be considered equal to int 0.

**Example:**

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

int main()
{
    const bool isTrue = true;
    const bool isFalse = false;

    cout << "isTrue? "
         << isTrue << "\n";
    cout << "isFalse? "
         << isFalse << "\n";
```

```
    return 0;
}
```

**Output:**

```
isTrue? 1
isFalse? 0
```

Adapted from:
"Types of Literals in C/C++ with Examples" by Chinmoy Lenka, Geeks for Geeks

# 4.5: Data Manipulation

## Data Manipulation

Single values by themselves are important; however we need a method of manipulating values (processing data). Scientists wanted an accurate machine for manipulating values. They wanted a machine to process numbers or calculate answers (that is compute the answer). Prior to 1950, dictionaries listed the definition of computers as " humans that do computations". Thus, all of the terminology for describing data manipulation is math oriented. Additionally, the two fundamental data type families (the integer family and floating-point family) consist entirely of number values.

## An Expression Example with Evaluation

Let's look at an example: 2 + 3 * 4 + 5 is our expression but what does it equal?

> the symbols of + meaning addition and * meaning multiplication are our operators
>
> the values 2, 3, 4 and 5 are our operands
>
> precedence says that multiplication is higher than addition
>
> thus, we evaluate the 3 * 4 to get 12
>
> now we have: 2 + 12 + 5
>
> the associativity rules say that addition goes left to right, thus we evaluate the 2 +12 to get 14
>
> now we have: 14 + 5
>
> finally, we evaluate the 14 + 5 to get 19; which is the value of the expression

Parentheses would change the outcome. (2 + 3) * (4 + 5) evaluates to 45.

Parentheses would change the outcome. (2 + 3) * 4 + 5 evaluates to 25.

## Precedence of Operators Chart

Each computer language has some rules that define precedence and associativity. They often follow rules we may have already learned. Multiplication and division come before addition and subtraction is a rule we learned in grade school. This rule still works. The precedence rules vary from one programming language to another. You should refer to the reference sheet that summarizes the rules for the language that you are using. It is often called a Precedence of Operators Chart. You should review this chart as needed when evaluating expressions.

A valid expression consists of operand(s) and operator(s) that are put together properly. Why the (s)? Some operators are:

> Unary – that is only have one operand
>
> Binary – that is have two operands, one on each side of the operator
>
> Trinary – which has two operator symbols that separate three operands

Most operators are binary, that is they require two operands. Within C++ there is only one trinary operator, the conditional (we will talk about this later in the semester). All of the unary operators are on the left side of the operand, except postfix increment and postfix decrement. Some precedence charts indicate of which operators are unary and trinary and thus all others are binary.

## Definitions

**Expression**

A valid sequence of operand(s) and operator(s) that reduces (or evaluates) to a single value.

**Operator**

A language-specific syntactical token (usually a symbol) that causes an action to be taken on one or more operands.

**Operand**

A value that receives the operator's action.

**Precedence**

Determines the order in which the operators are allowed to manipulate the operands.

**Associativity**

Determines the order in which the operators of the same precedence are allowed to manipulate the operands.

**Evaluation**

The process of applying the operators to the operands and resulting in a single value.

**Parentheses**

Change the order of evaluation in an expression. You do what's in the parentheses first.

Adapted from: "Data Manipulation" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 4.6: Assignment Operator

## Assignment Operator

The assignment operator allows us to change the value of a modifiable data object (for beginning programmers this typically means a variable). It is associated with the concept of moving a value into the storage location (again usually a variable). Within C++ programming language the symbol used is the equal symbol. But bite your tongue, when you see the = symbol you need to start thinking: assignment. The assignment operator has two operands. The one to the left of the operator is usually an identifier name for a variable. The one to the right of the operator is a value.

```
Simple Assignment

int    age;    // variable set up
// then later in the program
age = 21;
```

The value 21 is moved to the memory location for the variable named: age. Another way to say it: age is assigned the value 21.

```
Assignment with an Expression

int total_cousins;    // variable set up
// then late in the program
total_cousins = 4 + 3 + 5 + 2;
```

The item to the right of the assignment operator is an expression. The expression will be evaluated and the answer is 14. The value 14 would assigned to the variable named: total_cousins.

```
Assignment with Identifier Names in the Expression

int    students_period_1 = 25;    // variable set up with initialization
int    students_period_2 = 19;
int    total_students;
// then late in the program
total_students = students_period_1 + students_period_2;
```

The expression to the right of the assignment operator contains some identifier names. The program would fetch the values stored in those variables; add them together and get a value of 44; then assign the 44 to the total_students variable.

As we have seen, assignment operators are used to assigning value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the variable on the left side otherwise the compiler will raise an error.
Different types of assignment operators are shown below:

- **“=”**: This is the simplest assignment operator, which was discussed above. This operator is used to assign the value on the right to the variable on the left.
  For example:

  ```
  a = 10;
  b = 20;
  ch = 'y';
  ```

- **+=**: This operator is combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
  Example:

  ```
  (a += b) can be written as (a = a + b)
  ```

  If initially the value 5 is stored in the variable a,  then:  (a += 6) is equal to 11.  (the same as: a = a + 6)

- **-=** This operator is combination of ‘-‘ and ‘=’ operators. This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left.
  Example:

  ```
  (a -= b) can be written as (a = a - b)
  ```

  If initially value 8 is stored in the variable a, then (a -= 6) is equal to  2. (the same as a = a - 6)

- ***=** This operator is combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
  Example:

  ```
  (a *= b) can be written as (a = a * b)
  ```

  If initially value 5 is stored in the variable a,, then (a *= 6) is equal to 30. (the same as a = a * 6)

- **/=** This operator is combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example:

  ```
  (a /= b) can be written as (a = a / b)
  ```

  If initially value 6 is stored in the variable a, then (a /= 2) is equal to 3. (the same as a = a / 2)

Below example illustrates the various Assignment Operators:

```
// C++ program to demonstrate
// working of Assignment operators
#include <iostream>
using namespace std;

int main()
{
    // Assigning value 10 to a
    // using "=" operator
    int a = 10;
    cout << "Value of a is "<< a <<"\n";

    // Assigning value by adding 10 to a
    // using "+=" operator
    a += 10;
    cout << "Value of a is "<< a <<"\n";

    // Assigning value by subtracting 10 from a
```

```
    // using "-=" operator
    a -= 10;
    cout << "Value of a is "<< a <<"\n";

    // Assigning value by multiplying 10 to a
    // using "*=" operator
    a *= 10;
    cout << "Value of a is "<< a <<"\n";

    // Assigning value by dividing 10 from a
    // using "/=" operator
    a /= 10;
    cout << "Value of a is "<< a <<"\n";

    return 0;
}
```

**Output:**

```
Value of a is 10
Value of a is 20
Value of a is 10
Value of a is 100
Value of a is 10
```

## Definitions

**assignment**

An operator that changes the value of a modifiable data object.

Adapted from: "Assignment Operator" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 4.7: L Value and R Value

## What am L (or R) Value?

They refer to on the left and right side of the assignment operator. The **Lvalue** (pronounced: L value) concept refers to the requirement that the operand on the left side of the assignment operator is modifiable, usually a variable. **Rvalue** concept pulls or fetches the value of the expression or operand on the right side of the assignment operator. Some examples:

```
int age;    //variable set up
// then later in the program
age = 39;
```

The value 39 is pulled or fetched (Rvalue) and stored into the variable named age (Lvalue); destroying the value previously stored in the age variable.

```
int age;    //variable set up
int voting_age = 18;    // variable set up with initialization
// then later in the program
age = voting_age;
```

If the expression has a variable or named constant on the right side of the assignment operator, it would pull or fetch the value stored in the variable or constant. The value 18 is pulled or fetched from the variable named voting_age and stored into the variable named age.

```
age < 17;
```

If the expression is a **test expression** or **Boolean expression**, the concept is still an Rvalue one. The value in the identifier named age is pulled or fetched and used in the relational comparison of less than.

```
const int JACK_BENNYS_AGE = 39; // constant set up
// then later in the program
JACK_BENNYS_AGE = 65;
```

This is illegal because the identifier JACK_BENNYS_AGE does not have Lvalue properties. It is not a modifiable data object, because it is a constant.

## Definitions

**LValue**

The requirement that the operand on the left side of the assignment operator is modifiable, usually a variable.

**RValue**

Pulls or fetches the value stored in a variable or constant.

Adapted from: "Lvalue and Rvalue" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 4.8: Sizeof Operator

## Sizeof What??

Every data item, constants and variables, not only have a data type, but the data type determines how many bytes the item will use in the memory of the computer. The size of each data type varies with the compiler being used and the computer. This effect is known as being **machine dependent**. Additionally, there have been some size changes with upgrades to the language. In "C" the int data type was allocated 2 bytes of memory storage on an Intel compatible central processing unit (cpu) machine. In "C++" an int is allocated 4 bytes.

There is an operator named "sizeof (… )" that is a unary operator, that is it has only one operand. The operand is to the right of the operator and is placed within the parentheses if it is a data type. The operand may be any data type (including those created by typedef). If the operand is an identifier name it does not need to go inside a set of parentheses. It works for both variable and memory constant identifier names. This operator is unique in that it performs its calculation at compile time for global scoped items and at run time for local scoped items. Examples:

```
cout << "The size of an integer is: " << sizeof (int);
```

The compiler would determine the byte size of an integer on the specific machine and in essence replaces the sizeof operator with a value. Integers are usually 4 bytes long, thus the line of code would be changed to:

```
cout << "The size of an integer is: " << 4;
```

If you place an identifier name that represents a data storage area (variable or memory constant), it looks at the definition for the identifier name. NOTE: the parentheses are not needed and often not included for an identifier name.

```
Sizeof with a Variable
double money;     // variable set up with initialization
// then later on in the program
cout << "The size of money is: " << sizeof money;
```

The compiler would determine the byte size of money by looking at the definition where it indicates that the data type is double. The double data type on the specific machine (usually 8 bytes) would replace the code and it would become:

```
cout << "The size of money is: " << 8;
```

## Definitions

**sizeof**

An operator that tells you how many bytes a data type occupies in storage.

Adapted from: "Sizeof Operator" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 4.9: Arithmetic Operators

## Basic Operators

An operator performs an action on one or more operands. The common arithmetic operators are:

| Action | C++ Operator Symbol |
|---|---|
| Addition | + |
| Subtraction | - |
| Multiplication | * |
| Division | / |
| Modulus (associated with integers) | % |

These arithmetic operators are binary that is they have two operands. The operands may be either constants or variables.

age + 1

This expression consists of one operator (addition) which has two operands. The first is represented by a variable named age and the second is a literal constant. If age had a value of 14 then the expression would evaluate (or be equal to) 15.

These operators work as you have learned them throughout your life with the exception of division and modulus. We normally think of division as resulting in an answer that might have a fractional part (a floating-point data type). However, division when both operands are of the integer data type act differently. Please refer to the supplemental materials on "Integer Division and Modulus".

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

int main()
{
    int myNum1 = 10, myNum2 = 4, result;

    // printing a and myNum2
    cout<<"myNum1 is "<< myNum1 <<" and b is "<<b<<"\n";

   // addition
   result = myNum1 + myNum2;
   cout << "myNum1 + myNum2 is: "<< result << "\n";

   // subtraction
   result = myNum1 - myNum2;
   cout << "myNum1 - myNum2 is: "<< result << "\n";

   // multiplication
   result = myNum1 * myNum2;
   cout << "myNum1 * myNum2 is: "<< result << "\n";

   // division
   result = myNum1 / myNum2;
```

```
    // since
    cout << "myNum1 / myNum2 is: "<< result << "\n";

    // modulus
    result = myNum1 % myNum2;
    cout << "myNum1 % myNum2 is: "<< result << "\n";

    return 0;
}
```

**Output:**

```
myNum1 is 10 and myNum2 is: 4
myNum1 + myNum2 is: 14
myNum1 - myNum2 is: 6
myNum1 * myNum2 is: 40
myNum1 / myNum2 is: 2
myNum1 % myNum2 is: 2
```

Adapted from:
" Arithmetic Operators" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY-SA 4.0
"Operators in C | Set 1 (Arithmetic Operators)" by Unknown author, Geeks for Geeks is licensed under CC BY-SA 4.0

# 4.10: Data Type Conversions

## Conversions...

Changing a data type of a value is referred to as "type conversion". There are two ways to do this:

1. **Implicit** – the change is implied
2. **Explicit** – the change is explicitly done with the cast operator

The value being changed may be:

1. **Promotion** – going from a smaller domain to a larger domain
2. **Demotion** – going from a larger domain to a smaller domain

## Implicit Type Conversion

Automatic conversion of a value from one data type to another by a programming language, without the programmer specifically doing so, is called implicit type conversion. It happens when ever a binary operator has two operands of different data types. Depending on the operator, one of the operands is going to be converted to the data type of the other. It could be promoted or demoted depending on the operator.

```
Implicit Promotion

55 + 1.75
```

In this example the integer value 55 is converted to a floating-point value (most likely double) of 55.0. It was promoted.

```
Implicit Demotion

int money;    // variable set up
// then later in the program
money = 23.16;
```

In this example the variable money is an integer. We are trying to move a floating-point value 23.16 into an integer storage location. This is demotion and the floating-point value usually gets truncated to 23.

## Promotion

Promotion is never a problem because the lower data type (smaller range of allowable values) is sub set of the higher data type (larger range of allowable values). Promotion often occurs with three of the standard data types: character, integer and floating-point. The allowable values (or domains) progress from one type to another. That is the character data type values are a sub set of integer values and integer values are a sub set of floating-point values; and within the floating-point values: float values are a sub set of double. Even though character data represent the alphabetic letters, numeral digits (0 to 9) and other symbols (a period, $, comma, etc.) their bit pattern also represent integer values from 0 to 255. This progression allows us to promote them up the chain from character to integer to float to double.

## Demotion

Demotion represents a potential problem with truncation or unpredictable results often occurring. How do you fit an integer value of 456 into a character value? How do you fit the floating-point value of 45656.453 into an integer value? Most compilers give a warning if it detects demotion happening. A compiler warning does not stop the compilation process. It does warn the programmer to check to see if the demotion is reasonable.

If I calculate the number of cans of soup to buy based on the number of people I am serving (say 8) and the servings per can (say 2.3), I would need 18.4 cans. I might want to demote the 18.4 into an integer. It would truncate the 18.4 into 18 and because the value 18 is within the domain of an integer data type, it should demote with the **truncation** side effect.

If I tried demoting a double that contained the number of stars in the Milky Way galaxy into an integer, I might have a get an **unpredictable result** (assuming the number of stars is larger than allowable values within the integer domain).

## Explicit Type Conversion

Most languages have a method for the programmer to change or cast a value from one data type to another; called **explicit type conversion**. Within C++ the cast operator is a unary operator; it only has one operand and the operand is to the right of the operator. The operator is a set of parentheses surrounding the new data type.

**Explicit Demotion with Truncation**

```
(int) 4.234
```

This expression would evaluate to: 4.

## Definitions

**Implicit**

A value that has its data type changed automatically

**Explicit**

Changing a value's data type with the cast operator.

**Promotion**

Going from a smaller domain to a larger domain.

**Demotion**

Going from a larger domain to a smaller domain.

**Truncation**

The fractional part of a floating-point data type that is dropped when converted to an integer.

Adapted from: "Data Type Conversions" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY-SA 4.0

# 4.11: Operator Overloading

## What is Overloading?

C++ allows the specification of more than one definition for an **operator** in the same scope, which is called **operator overloading.**

An overloaded declaration is a declaration that is within the same scope and with the same name as a previous declaration, except that both declarations have different arguments and obviously different definition (implementation).

When an overloaded operator is called, the compiler determines the most appropriate definition to use, by comparing the argument types that have used to call the operator with the parameter types specified in the definitions.

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Example:

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

// Define a Class, a private variable: myVal, and 3 Class Methods: MyClass() - this i
// and Showem()
class MyClass
{
   private:
      int myVal;

   public:
       MyClass()
       {
           myVal = 99;
       }

       void operator ++()
       {
          myVal = myVal + 5;
       }

       void Showem()
       {
           cout<<"Value is: "<< myVal << endl;
       }
};


int main()
{   // Create an instance of Class MyClass
    MyClass myInstance;
    // calls our overloaded ++() operator function INSTEAD of the system ++ operator
    // which adds 5 to the value of myVal
    ++myInstance;
    // Print out the new value of myVal
    myInstance.Showem();

    return 0;
}
```

Output:

```
Value is: 104
```

Overloading allows you to create your own operators that behave in ways different than what C++ operators do, which provides a huge amount of flexibility in your code.

# 4.12: Unary Positive and Negative Operators

## General Discussion

Unary positive also known as plus and unary negative also known as minus are unique operators. The plus and minus when used with a constant value represent the concept that the values are either positive or negative. Let's consider:

```
+5 + -2
```

We have three operators in this order: unary positive, addition, and unary negative. The answer to this expression is a positive 3. As you can see, one must differentiate between when the plus sign means unary positive and when it means addition. Unary negative and subtraction have the same problem. Let's consider:

```
-2 - +5
```

The expression evaluates to negative 7. Let's consider:

```
7 - -2
```

First constants that do not have a unary minis in front of them are assumed (the default) to be positive. When you subtract a negative number it is like adding, thus the expression evaluates to positive 9.

## C++ Code Examples

The above examples work within the C++ programming language. What happens if we put a unary positive or unary negative in front of a variable or a named constant?

### Negation - Unary Negative

The concept of negation is to take a value and change its sign, that is: flip it. If it positive make it negative and if it is negative make it positive. Mathematically, it is the following C++ code example, given that money is an integer variable with a value of 6:

```
-money
```

```
money * -1
```

The above two expressions evaluate to the same value. In the first line, the value in the variable money is fetched and then it's negated to a negative 6. In the second line, the value in the variable money is fetched and then it's multiplied by negative 1 making the answer a negative 6.

### Unary Positive - Worthless

Simply to satisfy symmetry, the unary positive was added to the C++ programming language as on operator. However, it is a totally worthless or useless operator and is rarely used. However don't be confused the following expression is completely valid:

```
6 + +5
```

The second + sign is interpreted as unary positive. The first + sign is interpreted as addition.

```
money
```

```
+money
```

```
money * +1
```

For all three lines, if the value stored in money is 6 the value of the expression is 6. Even if the value in money was negative 77 the value of the expression would be negative 77. The operator does nothing, because multiplying anything by 1 does not change its value.

### Possible Confusion

Do not confuse the unary negative operator with decrement. Decrement changes the value in the variable and thus is an Lvalue concept. Unary negative does not change the value of the variable, but uses it in an Rvalue context. It fetches the value and then negates that value. The original value in the variable does not change.

Because there is no changing of the value associated with the identifier name, the identifier name could represent a variable or named constant.

## Exercises

```
Questions
    1. +10 - -2
    2. -18 + 24
    3. 4 - +3
    4. +8 + - +5
    5. +8 + / +5
Answers
    1. 12
    2. 6
    3. 1
    4. It's 3. Surprised, but it works. The middle plus sign is addition and the rest
    positive or unary negative
    5. Error, no operand between addition and division.
```

## Definitions

**Unary Positive**

A worthless operator almost never used.

**Unary Negative**

An operator that causes negation.

**Plus**

Aka unary positive.

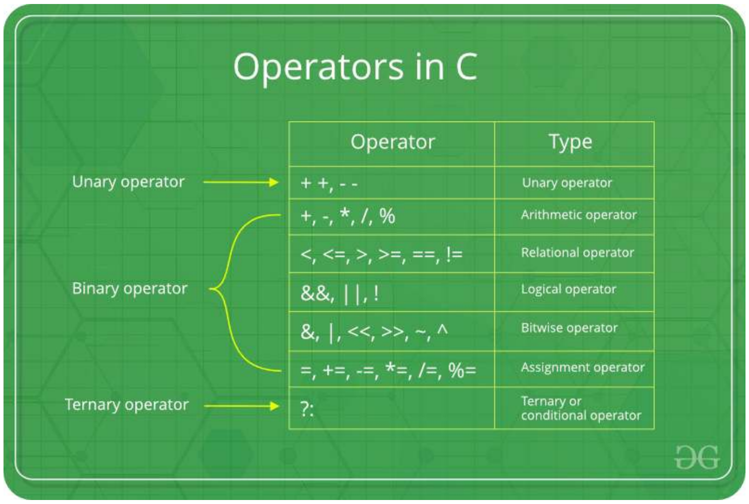
**Minus**

Aka unary negative.

# 4.13: Bitwise Operators

## Bitwise Operators

The thing about bitwise operators is they work at the bit level, and it takes some practice to get used to them.



Operators in C

| Operator | Type |
|---|---|
| + +, - - | Unary operator |
| +, -, *, /, % | Arithmetic operator |
| <, <=, >, >=, ==, != | Relational operator |
| &&, \| \|, ! | Logical operator |
| &, \|, <<, >>, ~, ^ | Bitwise operator |
| =, +=, -=, *=, /=, %= | Assignment operator |
| ?: | Ternary or conditional operator |

Unary operator
Binary operator
Ternary operator

So, if we take 5 - we get the 4 bits 0101. 9 is the 4 bits 1001

1. The **& (bitwise AND)** takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
   To AND the bits - a 1 AND 1 is 1 - EVERYTHING else is 0.
   0101
   1001
   0001  - which is 1
2. The **| (bitwise OR)**  takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
   To OR the bits - a 1 OR 0 is 1 - only two 0's is 0.
   0101
   1001
   1101  - which is 13
3. The **^ (bitwise XOR)** takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
   To XOR the bits we find where the bits are different
   0101
   1001
   1100  - which is 12
4. The **<< (left shift)**  takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
   To shift we need to take an 8 bit number - 9 in 16 bits is 00001001. Shift it one bit to the left we get 00010010, which is w 18
5. The **>> (right shift)**  takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
   To shift we need to take an 8 bit number - 9 in 16 bits is 00001001. Shift it one bit to the right we get 00000100, which is now 4
6. The **~ (bitwise NOT)** takes one number and inverts all bits of it. Take our 16 bit 5, 00000101 and invert every bit give us 11111010 which is 250

The following code shows this.

```
// C++ Program to demonstrate use of bitwise operators
#include <iostream>
```

```
using namespace std;

int main()
{
    // a = 5(00000101), b = 9(00001001)
    unsigned char a = 5, b = 9;

    cout <<  "a = " << +a << " b = " << +b << endl;

    //The result is 00000001
    cout << "a & b = " << (a & b) << endl;

    // The result is 00001101
    cout <<  "a | b =   " << (a | b) << endl;

    // The result is 00001100
    cout << "a^b =   " << (a ^ b) << endl;

    // The result is 11111010
    a = ~a;
    cout << "~a =   " << +a  << endl;

    // The result is 00010010
    cout << "b<<1 =   " << (b << 1) << endl;

    // The result is 00000100
    cout << "b>>1 =   " << (b >> 1) << endl;

    return 0;
}
```

Output:

```
a = 5 b = 9
a & b = 1
a | b =   13
a^b =   12
~a =   250
b<<1 =   18
b>>1 =   4
```

1. **The left shift and right shift operators should not be used for negative numbers**. If any of the operands is a negative number, it results in undefined behavior. For example results of both -1 << 1 and 1 << -1 is undefined. Also, if the number is shifted more than the size of integer, the behavior is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits.
2. **The bitwise XOR operator is the most useful operator from technical interview perspective.** It is used in many problems. A simple example could be “Given a set of numbers where all elements occur even number of times except one number, find

the odd occurring number” This problem can be efficiently solved by just doing XOR of all numbers.

3. **The bitwise operators should not be used in place of logical operators.** The result of logical operators (&&, || and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1. For example, consider the following program, the results of & and && are different for same operands.
4. **The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.** As mentioned in point 1, it works only if numbers are positive.
5. **The & operator can be used to quickly check if a number is odd or even.** The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.
6. **The ~ operator should be used carefully.** The result of ~ operator on a small number can be a big number if the result is stored in an unsigned variable. And the result may be a negative number if the result is stored in a signed variable (assuming that the negative numbers are stored in 2’s complement form where the leftmost bit is the sign bit)

Adapted from:
"Bitwise Operators in C/C++" by Multiple Contributors, Geeks for Geeks is licensed under CC BY-SA 4.0

# CHAPTER OVERVIEW

## 5: Common Data Types

# 5.1: Integer Data Type

## Integers

The integer data type has two meanings:

- The integer data type with its various modifiers that create different domains
- The integer family which also includes the Boolean and character data types

The integer data type basically represents whole numbers (no fractional parts). The integer values jump from one value to another. There is nothing between 6 and 7. It could be asked why not make all your numbers floating point which allow for fractional parts. The reason is twofold. First, some things in the real world are not fractional. A dog, even with only 3 legs, is still one (1) dog not ¾ of a dog. Second, integer data type is often used to control program flow by counting, thus the need for a data type that jumps from one value to another.

The integer data type has the same attributes and acts or behaves similarly in all programming languages. The most often used integer data type in C++ is the simple integer.

| | |
|---|---|
| C++ Reserved Word | int |
| Represent | Whole numbers (no fractional parts) |
| Size | Usually 4 bytes |
| Normal Signage | Signed (negative and positive values) |
| Domain (Values Allowed) | -2,147,483,648 to 2,147,483,648 |
| C++ syntax rule | Do not start with a 0 (zero) |
| C++ syntax rule | No decimal point |

We have seen the table below in Lesson 4 - it summarizes the modified size and range of built-in integer when combined with the type modifiers. Notice that there are signed and unsigned data types. The unsigned types start at zero, and only contain posititve values, these types also have a higher range since they have an additional bit (since the sign bit is NOT used) to use to contain the value.

Within C++ there are various reserved words that can be used to modify the size or signage of an integer. They include: long, short, signed and unsigned. Signed is rarely used because integers are signed by default – you must specify unsigned if you want integers that are only positive. Possible combinations are:

| DATA TYPE | SIZE (IN BYTES) | VALUE RANGE |
|---|---|---|
| short int | 2 | -32,768 to 32,767 |
| unsigned short int | 2 | 0 to 65,535 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |
| long long int | 8 | -(2^63) to (2^63)-1 |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 |

The domain of each of the above data type options varies with the compiler being used and the computer. The domains vary because the byte size allocated to the data varies with the compiler and computer. This effect is known as being **machine**

**dependent**. Additionally, there have been some size changes with upgrades to the language. In "C" the int data type was allocated 2 bytes of memory storage on an Intel compatible central processing unit (cpu) machine. In "C++" an int is allocated 4 bytes.

These variations of the integer data type are an annoyance in C++ for a beginning programmer. For a beginning programmer it is more important to understand the general attributes of the integer data type that apply to most programming languages.

## Definitions

**Machine Dependent**

An attribute of a programming language that changes depending on the computer's CPU.

Adapted from:
"C++ Data Types" by Harsh Agarwal, Geeks for Geeks
"Integer Data Type" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 5.2: Floating-Point Data Type

## Floating Point

The floating-point data type is a family of data types that act alike and differ only in the size of their domains (the allowable values). The floating-point family of data types represent number values with fractional parts. They are technically stored as two integer values: a **mantissa** and an **exponent**. The floating-point family has the same attributes and acts or behaves similarly in all programming languages. They can always store negative or positive values thus they always are signed; unlike the integer data type that could be unsigned. The **domain** for floating-point data types varies because they could represent very large numbers or very small numbers. Rather than talk about the actual values, we mention the **precision**. The more bytes of storage the larger the mantissa and exponent, thus more precision.

The most often used floating-point family data type used in C++ is the **double**. By default, most compilers convert floating-point constants into the double data type for use in calculations. The double data type will store just about any number most beginning programmers will ever encounter.

| | |
|---|---|
| C++ Reserved Word | double |
| Represent | Numbers with fractional parts |
| Size | Usually 8 parts |
| Storage | two parts (always treated together) a mantissa and an exponent |
| Normal Signage | Signed (negative and positive values) |
| Domain (Values Allowed) | ±1.7E-308 to ±1.7E308 |
| C++ syntax rule | the presence of a decimal point means it's floating-point data |

Within C++ there are various reserved words that can be used to establish the size in bytes of a floating-point data item. More bytes mean more precision: (these values are from g++ 7.5.0)

| DATA TYPE | SIZE (IN BYTES) | RANGE |
|---|---|---|
| float | 4 | 1.17549e-38 to 3.40282e+38 |
| double | 8 | 2.22507e-308 to 1.79769e+308 |
| long double | 12 (some machines use only 10 bytes) | 3.3621e-4932 to 1.18973e+4932 |

The domain of each of the above data type options varies with the compiler being used and the computer. The domains vary because the byte size allocated to the data varies with the compiler and computer. This effect is known as being **machine dependent**.

These variations of the floating-point family of data types are an annoyance in C++ for a beginning programmer. For a beginning programmer it is more important to understand the general attributes of the floating-point family that apply to most programming languages.

### Definitions

**Double**

The most often used floating-point family data type used in C++

**Precision**

The effect on the domain of floating-point values given a larger or smaller storage area in bytes.

**Mantissa Exponent**

The two integer parts of a floating-point value.

Adapted from:
"C++ Data Types" by Harsh Agarwal, Geeks for Geeks
"Floating-Point Data Type" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 5.3: String Data Type

## Strings

Technically, there is no string data type in the C++ programming language. However, the concept of a string data type makes it easy to handle strings of character data. A single character has some limitations. Many data items are not integers or floating-point values. The message **Hi Mom!** is a good example of a string. Thus, the need to handle a series of characters as a single piece of data (in English correctly called a datum).

In the "C" programming language all string were handled as an array of characters that end in an ASCII null character (the value 0 or the first character in the ASCII character code set). Associated with object oriented programming the string class has been added to C++ as a standard part of the programming language. This changed with the implementation with strings being stored as a length controlled item with a maximum length of 255 characters. Included in the C++ string class is the reserved word of **string** as if it were a data type. Some basics about strings include:

| | |
|---|---|
| C++ Reserved Word | String |
| Represent | Series of Characters (technically an array) |
| Size | Varies in length |
| Normal Signage | N/A |
| Domain (Values Allowed) | Extended ASCII Character Code Set |
| C++ syntax rule | Double quote marks for constants |

For now, we will address only the use of strings as constants. There are numerous methods to process strings with, and we will work with some of those as we prgress through the semester.

## std::string class in C++

C++ has in its definition a way to represent **sequence of characters as an object of class**. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing **access to single byte character**.

### std:: string vs Character Array

- A character array is simply an **array of characters** can terminated by a null character. A string is a **class which defines objects** that be represented as stream of characters.
- Size of the character array has to **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is wasted** in case of character array. In case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated, **no memory is wasted**.
- There is a **threat of array decay** in case of character array. As strings are represented as objects, **no array decay** occurs.
- Implementation of **character array is faster** than std:: string. **Strings are slower** when compared to implementation than character array.
- Character array **do not offer** much **inbuilt functions** to manipulate strings. String class defines **a number of functionalities** which allow manifold operations on strings.

### Operations on strings

### Input Functions

- **getline()** :- This function is used to **store a stream of characters** as entered by the user in the object memory.
- **push_back()** :- This function is used to **input** a character at the **end** of the string.
- **pop_back()** :- Introduced from C++11(for strings), this function is used to **delete the last character** from the string.

### Capacity Functions

- **capacity()** :- This function **returns the capacity** allocated to the string, which can be **equal to or more than the size** of the string. Additional space is allocated so that when the new characters are added to the string, the **operations can be done**

**efficiently**.

- **resize()** :- This function **changes the size of string**, the size can be increased or decreased.
- **length()**:-This function **finds the length of the string**
- **shrink_to_fit()** :- This function **decreases the capacity** of the string and makes it equal to its size. This operation is **useful to save additional memory** if we are sure that no further addition of characters have to be made.

### Iterator Functions

- **begin()** :- This function returns an **iterator** to **beginning** of the string.
- **end()** :- This function returns an **iterator** to **end** of the string.
- **rbegin()** :- This function returns a **reverse iterator** pointing at the **end** of string.
- **rend()** :- This function returns a **reverse iterator** pointing at **beginning** of string.

## Definitions

**string**

A series or array of characters as a single piece of data.

Adapted from:
"String Data Type" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0
"std::string class in C++" by Manjeet Singh, Geeks for Geeks is licensed under CC BY-SA 4.0

# 5.4: Character Data Type

## Overview of Character Data Type

The **character** data type basically represents individual or single characters. Characters comprise a variety of symbols such as the alphabet (both upper and lower case) the numeral digits (0 to 9), punctuation, etc. All computers store character data in a one byte field as an integer value. Because a byte consists of 8 bits, this one byte field has 28 or 256 possibilities using the positive values of 0 to 255.

Most microcomputers use the **ASCII** (stands for American Standard Code for Information Interchange and is pronounced "ask-key") Character Set which has established values for 0 to 127. For the values of 128 to 255 they usually use the Extended ASCII Character Set. When we hit the capital A on the keyboard, the keyboard sends a byte with the bit pattern equal to an integer 65. When the byte is sent from the memory to the monitor, the monitor converts the integer value of 65 to into the symbol of the capital A to display on the monitor.

The character data type attributes include:

| | |
|---|---|
| C++ Reserved Word | char |
| Represent | Single characters |
| Size | 1 byte |
| Normal Signage | Unsigned (positive values only) |
| Domain (Values Allowed) | Values from 0 to 127 as shown in the standard ASCII Character Set, plus values 128 to 255 from Extended ASCII Character Set |
| C++ syntax rule | Single quote marks - Example: 'A' |

Notice that char and unsigned char are both 1 byte, a wide char is 2 to 4 bytes.

| DATA TYPE | SIZE (IN BYTES) | RANGE |
|---|---|---|
| signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| wchar_t | 2 or 4 | 1 wide character |

Since some languages cannot represent all of their alphabet's characters in an 8 bit value, it was decided to create wide characters to solve this issue. In 1989, the International Organization for Standardization began work on the Universal Character Set (UCS), a multilingual character set that could be encoded using either a 16-bit (2-byte) or 32-bit (4-byte) value. These larger values required the use of a datatype larger than 8-bits to store the new character values in memory. Thus the term wide character was used to differentiate them from traditional 8-bit character datatypes.

## Character arithmetic in C++

As already known character known character range is between -128 to 127 or 0 to 255. This point has to be kept in mind while doing character arithmetic. To understand better let’s take an example.

Look at this example to understand better.

```
// A C++ program to demonstrate character
// arithmetic in C++.
#include <bits/stdc++.h>
using namespace std;
```

```
int main()
{
    char ch = 65;
    // The numerical value is 65, BUT...this is declared as a char, so it outputs the
    // See https://www.ascii-code.com/ - scroll down to 65 and look at the 5th column
    cout << ch << endl;

    // Now we add zero and C++ will see it as an integer value - it gets promoted.
    cout << ch + 0 << endl;

    // We add 32 but force it back to a char with "char(ch + 32)" 65 + 32 = 97
    // Look again at the https://www.ascii-code.com/ table for 97
    cout << char(ch + 32) << endl;
    return 0;
}
```

Output:

```
A
65
a
```

Without a ‘+’ operator character value is printed. But when used along with ‘+’ operator behaved differently. Use of ‘+’ operator implicitly typecasts it to an ‘int’. So to conclude, in character arithmetic, typecasting of char variable to ‘char’ is explicit and to ‘int’ it is implicit.

Adapted from:
"C++ Data Types" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0
"Character Data Type" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0
"Character arithmetic in C and C++" by Parveen Kumar, Geeks for Geeks is licensed under CC BY-SA 4.0

# 5.5: Interger Division and Modulus

## Overview of Integer Division and Modulus

By the time we reach adulthood, we normally think of division as resulting in an answer that might have a fractional part (a floating-point data type). This type of division is known as **floating-point division**. However, division when both operands are of the integer data type acts differently on most computers and is called: **integer division**. Within the C++ programming language the following expression does not give the answer of 2.75 or 2 ¾.

```
11/4
```

Because both operands are of the integer data type the evaluation of the expression (or answer) would be 2 with no fractional part (it gets thrown away). Again, this type of division is call **integer division** and it is what you learned in grade school the first time you learned about division.

**Figure 5.5.1** Integer division as learned in grade school.

In the real world of data manipulation there are some things that are always handled in whole units or numbers (integer data type). **Fractions just don't exist.** To illustrate our example: I have 11 dollar coins to distribute equally to my 4 children. How many do they each get? Answer is 2 with me still having 3 left over (or with 3 still remaining in my hand). The answer is not 2 ¾ each or 2.75 for each child. The dollar coins are not divisible into fractional pieces. Don't try thinking out of the box and pretend you're a pirate. Using an axe and chopping the 3 remaining coins into pieces of eight. Then, giving each child 2 coins and 6 pieces of eight or 2 6/8 or 2 ¾ or 2.75. If you do think this way, I will change my example to cans of tomato soup. I dare you to try and chop up three cans of soup and give each kid ¾ of a can. Better yet, living things like puppy dogs. After you divide them up with an axe, most children will not want the ¾ of a dog.

What is **modulus**? It's the other part of the answer for integer division. It's the remainder. Remember in grade school you would say, "Eleven divided by four is two remainder three." In C++ programming language the symbol for the modulus operator is the percent sign (%).

```
11 % 4
```

Thus, the answer or value of this expression is 3 - 11 divided by 4 is 2 with a remainder of 3. Modulus simply returns the remained portion of the division.

Many compilers require that you have integer operands on both sides of the modulus operator or you will get a compiler error. In other words, it does not make sense to use the modulus operator with floating-point operands.

Don't let the following items confuse you.

```
6 / 24 which is different from 6 % 24
```

How many times can you divide 24 into 6? 6 divided by 24 is zero. This is different from: What is the remainder of 6 divided by 24. The remainder portion is 6, the remainder part is what is returned by modulus.

**Questions**

1. 14 / 4

```
    2. 5 / 13
    3. 7 / 2.0
    4. 14 % 4
    5. 5 % 13
    6. 7 % 2.0
Answers
    1. 3
    2. 0
    3. 3.5 because one of the operands is a floating-point value, it is not integer d
    4. 2
    5. 5
    3. "error" because most compilers require both operands to be of the integer data
```

Adapted from:
"Integer Division and Modulus" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 5.6: Typedef - An Alias

## General Discussion

The typedef statement allows the programmer to create an **alias**, or synonym, for an existing data type. This can be useful in documenting a program. The C++ programming language syntax is:

typedef <the real data type> <the alias identifier name>;

Let's say a programmer is using a double data type to store the amount of money that is being used for various purposes in a program. He might define the variables as follows:

```
Regular Definition of Variables
double    income;
double    rent;
double    vacation;
```

However, he might use the typedef statement and define the variables as follows:

```
Using typedef when Defining Variables
typedef double cash;
    the typedef must be defined before its use
cash  income;
cash  rent;
cash  vacation;
```

So, we can use typedef to create an alias. In the above example we create a new type of 'cash' - and then we can use this new type just as we would use int, or float, or double to define a variable. the variable income is actually a double, because we stated this using the typedef instruction.

The typedef statement is not used very often by beginning programmers. It usually creates more confusion than needed, thus stick to using the normal data types at first.

## Definitions

**typedef**

Allows the programmer to create an alias, or synonym, for an existing data type.

Adapted from:
"Typedef - an alias" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 5.7: Sequence Operator

## General Discussion

The **sequence** (or comma) operator is used to separate items. It has several uses, four of which are listed then demonstrated:

1. To separate identifier names when declaring variables or constants
2. To separate several parameters being passed into a function
3. To separate several initialization items or update items in a for loop
4. Separate values during the initialization of an array

This first example is often seen in textbooks, but this method of declaring variables is not preferred. It is difficult to quickly read the identifier names.

```
int pig, dog, cat, rat
```

The following vertical method of declaring variables or constants is preferred.

```
Preferred Vertical Method of Defining Variables
int     pig;
int     dog;
int     cat;
int     rat;
```

The data types and identifier names (known as parameters) are separated from each other. This example is a function prototype.

```
double area_trapezoid(double base, double height, double top);
```

In the syntax of a for loop you have three parts each separated by a semi-colon. The first is the initialization area which could have more than one initialization. The last is the update area which could have more than one update. Multiple initializations or updates use the comma to separate them. This example is only the first line of a for loop. (we will talk about loops in more detail later)

```
for(x = 1, y = 5; x < 15; x++, y++)
```

The variable ages is an array of integers (an array is a list - more later). Initial values are assigned using block markers with the values separated from each other using a comma.

```
int ages[] = {2,4,6,29,32};
```

## Definitions

**Sequence**

An operator used to separate multiple occurrences of an item.

Adapted from:
"Sequence Operator" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

CHAPTER OVERVIEW

# 6: Conditional Execution

# 6.1: Conditional Execution

## Decision Making in C++

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. For example, in C++ if x occurs then execute y else execute z. There can also be multiple conditions like in C++ if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions.

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in C or C++ are:

1. if statement
2. if..else statements
3. nested if statements
4. if-else-if ladder
5. switch statements
6. Jump Statements:
    1. break
    2. continue
    3. goto
    4. return

We will take a look at all of these possibilities.

Adapted from:
"Decision Making in C / C++ (if , if..else, Nested if, if-else-if )" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 6.2: If - Else

## Conditional Execution

We are going to introduce the control structure from the selection category that is available in every high level language. It is called the **if - else** structure. Asking a question that has a true or false answer controls the `if` statement. It looks like this:

```
if the answer to the question is true
  then do this
```

The questions, called the condition, is a **Boolean expression**. The Boolean data type has two values – TRUE and FALSE. Let's rewrite the structure to consider this:

```
if expression is true
  then do this
```

Some languages use reserved words of: `if`, `then` and `else`. Many eliminate the `then`. Additionally the "do this" can be tied to true and false. You might see it as:

```
if expression is true
  action true
```

And most languages infer the "is true" you might see it as:

```
if expression
  action true
```

The above four forms of the control structure are saying the same thing. The else word is often not used in our English speaking today. However, consider the following conversation between a mother and her child.

Child asks, "Mommy, may I go out side and play?"

Mother answers, "If your room is clean then you may go outside and play or else you may go sit on a chair for five minutes as punishment for asking me the question when you knew your room was dirty."

Let's note that all of the elements are present to determine the action (or flow) that the child will be doing. Because the question (your room is clean) has only two possible answers (true or false) the actions are **mutually exclusive**. Either the child 1) goes outside and plays or 2) sits on a chair for five minutes. One of the actions is executed; never both of the actions.

### if statement in C/C++

The `if` statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not. If you remember, anything enclosed in the curly brackets is considered a block of code.
**Syntax**:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Here, **condition** after evaluation will be either true or false. If the value is true then it will execute the block of statements below it otherwise not.

**TAKE NOTE:** If we do not provide the curly braces '{' and '}' after `if (condition)` then by default ONLY the first statement is part of the `if` statement.

For example - the code below will see only statement1 as part of the if statement. In this case statement2 will get executed whether the condition is TRUE or FALSE. Just because a line of code is indented DOES NOT mean it is considered part of the `if` statement

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true
// if considers statement1 to be inside its block.
```

A very simple piece of code shows an `if` statement. In this case the condition ( myNum > 10) is FALSE, so we would not see the "10 is less than 15" output to the terminal.

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

int main()
{
        int myNum = 10;

        if (myNum > 15)
        {
                cout << "10 is less than 15" << endl;
        }

        cout << "I am Not in if" << endl;
}
```

## Two Way Selection within C++

The syntax for the if then else control structure within the C++ programming language is:

```
if (expression)
  statement;
else
  statement;
```

Note: The test expression is within the parentheses, but this is not a function call. The parentheses are part of the control structure. Additionally, there is no semicolon after the parenthesis following the expression.

Adapted from:
"If Then Else" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0
"Decision Making in C / C++ (if , if..else, Nested if, if-else-if )" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 6.2.1: Conditional Execution - If-Else

## The if-else Statement

There is also an if statement that has 2 parts to it. The statements that execute when the condition is TRUE, and the statements that execute when the statements is FALSE. The syntax for the if-else control structure:

```
// C++ program to illustrate if-else statement
#include<iostream>
using namespace std;

int main()
{
        int myNum = 20;

        if (myNum < 15)
            cout << "myNum is smaller than 15" << endl;
        else
            cout << "myNum is greater than 15" << endl;

    return 0;
}
```

Notice that we do NOT have the curly braces. This is allowed since we only have a single statement in the if portion and the else portion. You MAY include the brackets if you want to.

```
// C++ program to illustrate if-else statement
#include<iostream>
using namespace std;

int main()
{
        int myNum = 20;

        if (myNum < 15)
        {
            cout << "myNum is smaller than 15" << endl;
            myNum = 99;
        }
        else
            cout << "myNum is greater than 15" << endl;

    return 0;
}
```

This again is valid, you only need the curly brackets if you want to include more than a single statement in your block of code. In this case if you left out the brackets you would get an error on the `else` statement.

Adapted from:
"If Then Else" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0
"Decision Making in C / C++ (if , if..else, Nested if, if-else-if )" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 6.3: Boolean Data Type

## Discussion

The **Boolean** data type is also known as the logical data type and represents the concepts of true and false. The name "Boolean" comes from the mathematician George Boole; who in 1854 published: An Investigation of the Laws of Thought. Boolean algebra is the area of mathematics that deals with the logical representation of true and false using the numbers 0 and 1. The importance of the Boolean data type within programming is that it is used to control programming structures (if then else, while loops, etc.) that allow us to implement "choice" into our algorithms.

The Boolean data type has the same attributes and acts or behaves similarly in all programming languages. The rules within the C++ programming language are:

| | |
|---|---|
| C++ Reserved Word | bool |
| Represent | Logical concepts of true and false |
| Size | Usually 1 byte |
| Normal Signage | Unsigned |
| Domain (values allowed) | 0 meaning false, and 1 meaning true |
| C++ syntax rule | true and false are reserved words that can be used as values in expressions |
| C++ concept/rule | Any value from any data type can be demoted into a Boolean data type with zero representing false and all non-zero values representing true |

## Bool data type in C++

The C++ Standard has added the bool data types to the C++ specifications.They are provided to provide better control in certain situations as well as for providing conveniences to C++ programmers.

As mentioned above, bool values actually evaluate to either 0, which is FALSE, or 1, which is TRUE.

```
bool b1 = true;      // declaring a boolean variable with true value
```

In C++, the data type bool has been introduced to hold a boolean value, *true* or *false*.The values *true* or *false* have been added as keywords in the C++ language.
**Important Points:**

- The default numeric value of true is 1 and false is 0.
- We can use bool type variables or values *true* and *false* in mathematical expressions also.For instance,

  ```
  int x = false + true + 6;
  ```

  is valid and the expression on right will evaluate to **7** as false has value 0 and true will have value 1.
- It is also possible to convert implicitly the data type integers or floating point values to bool type

  ```
  bool x = 0; // false
  bool y = 100; // true
  bool z = 15.75; // true
  ```

Adapted from:
"Boolean Data Type" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0
"Bool data type in C++" by Mrigendra Singh, Geeks for Geeks is licensed under CC BY-SA 4.0

# 6.4: Relational Operators

## Overview of the Relational Operators

The relational operators are often used to create a **test expression** that controls program flow. This type of expression is also known as a **Boolean expression** because they create a Boolean answer or value when evaluated. There are six common relational operators that give a Boolean value by comparing (showing the relationship) between two operands. If the operands are of different data types, implicit promotion occurs to convert the operands to the same data type.

Operator symbols and/or names vary with different programming languages. The C++ programming language operators with their meanings are:

| C++ Operator | Meaning |
|---|---|
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| == | equality (equal to) |
| != | inequality (not equal to) |

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

// Main function
int main()
{
    int num1, num2;
    num1 = 33;
    num2 = 99;

    if(num1 != num2)
        cout << num1 << " is not equal to " << num2 << endl;

    if(num1 >= num2)
        cout << num1 << " is greater than " << num2 << endl;
    else
        cout << num1 << " is smaller than " << num2 << endl;
    return 0;
}
```

The answers to Boolean expressions within the C++ programming language are a value of either 1 for true or 0 for false. So, when we look at the first condition, it evaluates the 2 values, using the relational operator, and then returns a value of true , 1, or false, 0. This then determines whether the `if` statement is true or false. If we were using string variables, the comparison are based on the numerical value of the letters.

Be careful. In math you are familiar with using this symbol = to mean equal and ≠ to mean not equal. In the C++ programming language the ≠ is not used and the = symbol means assignment.

Adapted from: "Boolean Data Type" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 6.5: Compound Statement

## The Compound Statement

Often, we will want to have more than a single statement within our if statements. Some texts refer to this as a **compound statement**. The brace symbols – the opening { and the closing } - are used to create a compound statement. For example:

```
if(expression)
{
  statement;
  statement;
}
else
{
  statement;
  statement;
}
```

You can indeed use the braces with only a single statement, and many programmers do that just to be consistent with all of their code. Just be aware that some people refer to this as a compound statement. Others simply refer to it as a block of code. Either is correct, you just need to understand the terminology.

## Other Uses of a Compound Statement

"A compound statement is a unit of code consisting of zero or more statements. It is also known as a **code block** or a **block of code**. The compound statement allows a group of statements to become one single entry. You used a compound statement in your first program when you formed the body of the function main. All C++ functions contain a compound statement known as the function body.

A compound statement consists of an opening brace, optional declarations, definitions, and statements, followed by a closing brace. Although all three are optional, one should be present."[1]

## Definitions

**Compound Statement**

A unit of code consisting of zero or more statements.

**Block**

Another name for a compound statement.

## Footnotes

[1] Behrouz A. Forouzan and Richard F. Gilberg, Computer Science A Structured Approach using C++ Second Edition (United States of America: Thompson – Brooks/Cole, 2004) 100.

# 6.6: Ternary operator

## Conditional or Ternary Operator (?:) in C/C++

The conditional operator is kind of similar to the if-else statement as it does follow the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.

The conditional operator is of the form

```
variable = Expression1 ? Expression2 : Expression3
```

In this statement Expression1 is the condition. If the condition is true, then variable is assigned the value from Expression2, else variable is assigned the value of Expression2. It would be the same as:

```
if(Expression1)
{
    variable = Expression2;
}
else
{
    variable = Expression3;
}
```

Since the Conditional Operator ‘?:’ takes three operands to work, it is also called **ternary operators**.

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

int main()
{
    // variable declaration
    int num1 = 5, num2 = 10, maxNum;

    // If num1 > num2
    //    maxNum = num1
    // else
    //    maxNum = num2
    maxNum = (num1 > num2) ? num1 : num2;

    // Print the largest number
    cout << "Largest number between "
        << num1 << " and "
        << num2 << " is "
        << maxNum << endl;
    return 0;
}
```

In this case the output would be:

```
Largest number between 5 and 10 is 10
```

Now, if we wanted to get really crazy we could do something like the following. Decide the variable with the max value.

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

int main()
{
    int firstNum, secondNum, thirdNum, maxNum;
    firstNum = 55;
    secondNum = 33;
    thirdNum = 99;

    //the following statement replaces the whole if-else
    //statement and makes the code more concise
    maxNum = (firstNum > secondNum) ? (firstNum > thirdNum ? firstNum : thirdNum) : (
    cout << maxNum << "  is the largest number of given three numbers";

    return 0;
 }
```

```
6 is the largest number of given three numbers
```

Lets unpack this seemingly crazy C++ statement where nested ternary operators are used. *(firstNum > secondNum)* compares the value of *firstNum* and *secondNum.* If the value of *firstNum* is greater than value of *secondNum*, then *(firstNum > thirdNum ? firstNum : thirdNum)* will be executed that further compares the value of *firstNum* with the value of *thirdNum.* If the value of *firstNum* is greater than the value of *thirdNum*, the whole expression will become equivalent to the value of *firstNum*and assigned to *maxNum.* But if the value of *firstNum* is smaller than the value of *thirdNum* the whole expression will become equivalent to the value of *thirdNum* and assigned to *maxNum.*

And if *(firstNum > secondNum)* becomes false, then *(secondNum > thirdNum ? secondNum : thirdNum)* will be executed that further compares the value of *secondNum* with the value of *thirdNum.* If the value of *secondNum* is greater than the value of *thirdNum*, the whole expression will become equivalent to the value of *secondNum* and assigned to *maxNum.* But if the value of *secondNum* is smaller than the value of *thirdNum* the whole expression will become equivalent to the value of *thirdNum* and assigned to *maxNum.*

If it makes it easier to understand you could write this out with an if/else type of statement...but we won't do that here.

Adapted from:
"Conditional or Ternary Operator (?:) in C/C++" by khalidbitd, Geeks for Geeks is licensed under CC BY-SA 4.0

# 6.7: Nested If Then Else

## Introduction to Multiway Selection

### Nested Control Structures

We are going to first introduce the concept of nested if statements. Nesting is a concept that places one item inside of another. Consider:

```
if expression
  true action
else
  false action
```

This is the basic form of the if - else control structure. Now consider:

```
if age is less than 18
  you can't vote
  if age is less than 16
    you can't drive
  else
    you can drive
else
  you can vote
  if age is less than 21
    you can't drink
  else
    you can drink
```

As you can see we simply included as part of the "true action" a statement and another if then else control structure. We did the same (nested another if then else) for the "false action". In our example we nested if then else control structures. Nesting could have an if then else within a while loop. Thus, the concept of nesting allows the mixing of the different categories of control structures.

### Multiway Selection

One of the drawbacks of two way selection is that we can only consider two choices. But what do you do if you have more than two choices. Consider the following which has four choices:

```
if age equal to 18
  you can now vote
else
  if age equal to 39
    you are middle aged
  else
    if age equal to 65
      you can consider retirement
    else
      your age is unimportant
```

You get an appropriate message depending on the value of age. The last item is referred to as the default. If the age is not equal to 18, 39 or 65 you get the default message. In some situations there is no default action. Consider:

```
if age equal to 18
  you can now vote
else
  if age equal to 39
    you are middle aged
  else
    if age equal to 65
      you can consider retirement
```

The last if then else control structure has no "else". It’s implied "else do nothing". Without the default the multiway selection could be written as a series of "if then without the else" structures. Consider:

```
if age equal to 18
  you can now vote
if age equal to 39
  you are middle aged
if age equal to 65
  you can consider retirement
```

We have shown two ways to accomplish multiway selection. The choice of using nested if then else control structures or a series of if then control structures is decided on the existence of a default action (you must use nested if then else) or programmer preference if there is not a default action (you may use nested if then else or a series of if then control structures).

## If - Else Syntax within C++

The syntax for the if then else control structure within the C++ programming language is:

```
C++ source code: Layout of an if then else
if (expression)
{
   statement;
}
else
{
    statement;
}
```

The test expression is within the parentheses, but this is not a function call. The parentheses are part of the controlstructure. Additionally, there is no semicolon after the parenthesis following the expression.

## C++ Example

Multiway selection is often needed to cover all possibilities. Assume that the user has been prompted for the ages of two people with the answers stored in variables named age1 and age2. Consider:

```
C++ Source Code:
if(age1 > age2)
{
  cout << "\n\nThe first person is older.";
}
else
{
  cout << "\n\nThe second person is older.";
}
```

What if the two persons are the same age? The program incorrectly says the second person is older. To solve this we must handle all three possibilities. Consider this mulitway selection example:

```
C++ Source Code:
if(age1 == age2)
{
  cout << "\n\nThey are the same age.";
}
else
{
  if(age1 > age2)
  {
    cout << "\n\nThe first person is older.";
  }
  else
  {
    cout << "\n\nThe second person is older.";
  }
}
```

## Definitions

**Nested Control Structures**

Placing one control structure inside of another.

**Multiway Selection**

Using control structures to be able to select from more than two choices.

Adapted from:
"Nested If Then Else" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 6.8: Logical Operators

## Overview of the Logical Operators

Within most languages, expressions that yield Boolean data type values are divided into two groups. One group uses the relational operators within their expressions and the other group uses logical operators within their expressions.

A **logical operator** is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. This type of expression also creates a **Boolean expression** because they create a Boolean answer or value when evaluated. The answers to Boolean expressions within the C++ programming language are a value of either 1 for true or 0 for false. There are three common logical operators that give a Boolean value by manipulating other Boolean operand(s). Operator symbols and/or names vary with different programming languages. The C++ programming language operators with their meanings are:

| C++ Operator | Meaning | Comment | Typing |
|---|---|---|---|
| && | Logical and | | two ampersands |
| \|\| | Logical or | | two vertical dashes or piping symbols |
| ! | Logical not | unary | the exclamation point |

The vertical dashes or piping symbol is found on the same key as the backslash \. You use the SHIFT key to get it. It is just above the Enter key on most keyboards. It may be a solid vertical line on some keyboards and show as a solid vertical line on some print fonts.

The && - meaning AND - simply says that both sides must be true. The || - meaning or - simply says that one or the other must be true. The ! - meaning not - reverses the meainng of the variable.

In most languages there are strict rules for forming proper logical expressions. An example is:

```
6 > 4 && 2 <= 14
```

This expression evaluates the 6 > 4 to determine a true or false, and then check the operator. Since we are working with && - if the first value is true then it has to evaluate the second condition to see if they are both true. If the first value is false, it never looks at the second one, since for && BOTH have to be true.

This expression has two relational operators and one logical operator. Using the precedence of operator rules the two "relational comparison" operators will be done before the "logical and" operator. Thus:

```
1 && 1
```

or

```
true && true
```

The final evaluation of the expression is: 1 meaning true.

We can say this in English as: It is true that six is greater than four AND that two is less than or equal to fourteen.

When forming logical expressions programmers often use parentheses (even when not technically needed) to make the logic of the expression very clear. Consider the above complex Boolean expression rewritten:

```
(6 > 4) && (2 <= 14)
```

## Truth Tables

A common way to show logical relationships is in truth tables.

| **x** | **y** | **x && y** |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

| **x** | **y** | **x \|\| y** |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

| **x** | **!x** |
|---|---|
| false | true |
| true | false |

## Examples

I call this example of why I hate "and" and love "or".

Everyday as I came home from school on Monday through Thursday; I would ask my mother, "May I go outside and play?" She would answer, "If your room is clean and your homework is done then you may go outside and play." I learned to hate the word "and". I could manage to get one of the tasks done and have some time to play before dinner, but both of them… well, I hated "and".

On Friday my mother took a more relaxed view point and when asked if I could go outside and play she responded, "If your room is clean or your homework is done then you may go outside and play." I learned to clean my room quickly on Friday afternoon. Well needless to say, I loved "or".

For the next example, just imagine a teenager talking to their mother. During the conversation mom says, "After all, your Dad is reasonable!" The teenager says, "Reasonable. (short pause) Not."

Maybe college professors will think that all their students studied for the exam. Ha ha! Not. Well, I hope you get the point.

```
Questions
    1. 25 < 7 || 15 > 36
    2. 15 > 36 || 3 < 7
    3. 14 > 7 && 5 <= 5
    4. 4 > 3 && 17 <= 7
    5. ! false   - obviously NOT false is true
    6. ! (13 != 7)  - 13 is not equal to 7, so we have true, but then we have NOT tru
    7. 9 != 7 && !0
    8. 5 > && 7
Answers
    1.false
    2. true
    3. true
    4. false
    5. true
    6. false
    7. trru
    8. Error, there needs to be an operand between the operators > and &&.
```

# 6.9: Case Control Structure

## Traditional Case Control Structure

### Multiway Selection Using the Case Structure

One of the drawbacks of two way selection is that we can only consider two choices. But what do you do if you have more than two choices. Consider the following which has four choices:

```
if age equal to 18
  you can vote
else
  if age equal to 39
    you're middle aged
  else
    if age equal to 65
      consider retirement
  else
      age is un-important
```

You get an appropriate message depending on the value of age. The last item is referred to as the default. If the age is not equal to 18, 39 or 65 you get the default message. In some situations there is no default action.

### C++ Code to Accomplish Multiway Selection

Using the same example as above, here is the C++ code to accomplish the case control structure. A **case** or switch statement is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via a multiway branch.

```
//C++ source code - case structure with intergers
#include <iostream>
using namespace std;

int main()
{
   int age = 33;
   switch (age)
   {
      case 18: cout << "\nYou can vote.";
           break;
      case 39: cout << "\nYou're middle aged.";
           break;
      case 65: cout << "\nConsider retirement.";
           break;
      default: cout << "\nAge is un-important.";
    }

    return 0;
 }
```

The value in the variable age is compared to the first "case" (note: **case** is one of the C++ reserved words) which is the value 18 (also called the listed value) using an equality comparison or is "age equal to 18". If it is true, the cout is executed which displays “You can vote.” and the next line of code (the break) is done (which jumps us to the end of the control structure). If it is false, it moves on to the next case for comparison.

Most programming languages, including C++, require the listed values for the case control structure be of the integer family of data types. This basically means either an integer or character data type. Consider this example that uses character data type (choice is a character variable):

```
C++ source code - case structure with characters
switch (choice)
{
  case 'A': cout << "\nYou are an A student.";
            break;
  case 'B': cout << "\nYou are a B student.";
            break;
  case 'C': cout << "\nYou are a C student.";
            break;
  default:  cout << "\nMaybe you should study harder.";
}
```

A couple of things to note:

- A default value is NOT required
- if you leave out a break your code falls through

```
    switch (age)
    {
       case 18: cout << "\nYou can vote.";

       case 39: cout << "\nYou're middle aged.";
            break;
       case 65: cout << "\nConsider retirement.";
            break;
       default: cout << "\nAge is un-important.";
     }
```

This is valid C++ code. If age were set to 18, it falls through to the next case and the would simply output:

```
You can vote
You're middle aged.
```

- Most programming languages, including C++, do not allow ranges of values for case like structures.

Adapted from:
"Case Control Structure" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 6.10: Branching Control Structures

## Discussion

The branching control structures allow the flow of execution to jump to a different part of the program. The common branching control structures that are used with other control structures are: break, and continue. These control structures should be used with great care. Usually there is a better way to write your code than to have to rely on one of these. The break statement is valid, and NEEDS to be used in the switch statement that we discussed in the last page. There is one other branching control structure that is often not viewed as branching control structure. It is: return; which is used with functions. Thus, there are two commonly used branching control reserved words used in C++; break and return. Additionally, we will add to our list of branching items a pre-defined function commonly used in the C++ programming language of: exit; that is part of the C standard library (cstdlib).

## Examples

### break

The following is not a good use of break in a loop and it gives the appearance that the loop will execute 8 times, but the break statement causes it to stop during the fifth iteration.

```
C++ source code
counter = 0;
while(counter < 8)
{
  cout << counter << endl;
  if (counter == 4)
  {
      break;
   }
  counter++;
}
```

```
C++ source code
counter = 0;
while((counter < 8) && (count != 4))
{
  cout << counter << endl;
  counter++;
}
```

In this manner, we can tell in the loop statement exactly what is happening, and do not have to search through the rest of the code in the loop to figure out why we are not looping 8 times.

### continue

The continue statement says, "skip the rest of the loop and go back to the top of the loop for next iteration". The following gives the appearance that the loop will print to the monitor 8 times, but the continue statement causes it not to print number 4. There are times when this it totally acceptable, depending on the coding standards of the project you are working on.

```
C++ source code
for(counter = 0; counter < 8; counter++)
  {
  if (counter == 4)
```

```
    {
    continue;
    }
  cout << counter << endl;
  }
```

### return

The return statement simply says, "go back to where you came from". in the following example (we haven't talked about functions yet) we reach the end of the function get_data() and it is good practice to place a return statement here...and we simply go back to the place in our code where this function was called.

```
C++ source code
//******************************************************
// get data
//******************************************************

void get_data(void)
{
  // Input - Test Data - 5678.9, 5432.1
  cout << "\nEnter the length of the property in feet --->: ";
  cin >> property_length;
  cout << "\nEnter the width of the property in feet ---->: ";
  cin >> property_width;
  return;
}
```

### Exit

Although exit is technically a pre-defined function, it is covered here because of its common usage in programming. A good example is the opening a file and then testing to see if the file was actually opened. If not, we have an error that usually indicates that we want to pre-maturely stop the execution of the program. Within the C++ programming language the exit function terminates the running of the program and in the process returns an integer value back to the operating system. It fits the definition of branching which is to jump to some other place in the program. In our example the value returned to the operating system is the value of the constant named: EXIT_FAILURE.

```
C++ source code
inData.open(filename);  //Open input file
if (!inData)            //Test to see if file was opened
{
  cout << "\n\nError opening file: " << filename << "\n\n";
  pause();              //Pause - user reads message
  exit(EXIT_FAILURE);   //Allows a pre-mature jump to OS
 }
```

## Definitions

**Branching Control Structures**

Allow the flow of execution to jump to a different part of the program.

**Break**

A branching control structure that terminates the existing structure.

**Continue**

A branching control structure that causes a loop to stop its current iteration and begin the next one.

**Goto (SHOULD NEVER BE USED)**

A branching control structure that causes the logic to jump to a different place in the program.

**Return**

A branching control structure that causes a function to jump back to the function that called it.

**Exit**

A pre-defined function used to prematurely stop a program and jump to the operating system.

Adapted from:
"Branching Control Structures" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# CHAPTER OVERVIEW

## 7: Conditional Loops

# 7.1: Do While Loop

## Introduction to Test After Loops

There are two commonly used test after loops in the iteration (or repetition) category of control structures. They are: do while and repeat until. This module covers the: do while.

### Understanding Iteration in General - do while

The concept of iteration is connected to possibly wanting to repeat an action. Like all control structures we ask a question to control the execution of the loop. The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the do while loop is as follows:

```
do
  some statements or action
  some statements or action
  some statements or action
  update the flag
while the answer to the question is true
```

In every language that I know the question (called a **test expression**) is a **Boolean expression**. The Boolean data type has two values – true and false. Let's rewrite the structure to consider this:

```
do
  some statements or action
  some statements or action
  some statements or action
  update the flag
while expression is true
```

Within the do while control structure there are three attributes of a properly working loop. They are:

- Action or actions
- Update of the flag
- Test expression

The English phrasing is, "You do the action while the expression is true". This is looping on the true. When the test expression is false, you stop the loop and go on with the next item in the program. Notice, because this is a test after loop the action will always happen **at least once**. It is called a **test after loop** because the test comes after the action. It is also sometimes called a post-test loop, meaning the test is post (or Latin for after) the action and update.

## The do while Structure within C++

### Syntax

The syntax for the do while control structure within the C++ programming language is:

```
do
{
    statement;
    statement;
    statement;
    statement;    // This statement updates the flag;
}
while (expression);
```

The test expression is within the parentheses, but this is not a function call. The parentheses are part of the control structure. Additionally, there is a semicolon after the parenthesis following the expression.

### An Example

```
C++ source code: do while loop
do
{
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
  cout << "\nDo you want to do it again? y or n ";
  cin >> loop_response;
}
while (loop_response == 'y');
```

The three attributes of a test after loop are present. The action part consists of the 6 lines that prompt for data and then displays the total of the two ages. The update of the flag is the displaying the question and getting the answer for the variable loop_response. The test is the equality relational comparison of the value in the flag variable to the lower case character of y.

This type of loop control is called an event controlled loop. The flag updating is an event where someone decides if they want the loop to execute again.

Using indentation with the alignment of the loop actions and flag update is normal industry practice within the C++ community.

### Infinite Loops

At this point it's worth mentioning that good programming always provides for a method to insure that the loop question will eventually be false so that the loop will stop executing and the program continues with the next line of code. However, if this does not happen then the program is in an infinite loop. Infinite loops are a bad thing. Consider the following code:

```
C++ source code: infinite loop
loop_response = 'y';
do
{
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
}
while (loop_response == 'y');
```

The programmer assigned a value to the flag before the loop and forgot to update the flag. Every time the test expression is asked it will always be true. Thus, an infinite loop because the programmer did not provide a way to exit the loop (he forgot to update the flag).

Consider the following code:

```
C++ source code: infinite loop
do
{
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
  cout << "\nDo you want to do it again? y or n ";
  cin >> loop_response;
}
while (loop_response = 'y');
```

No matter what the user replies during the flag update, the test expression does not do a relational comparison but does an assignment. It assigns 'y' to the variable and asks if 'y' is true? Since all non-zero values are treated as representing true within the Boolean concepts of the C++ programming language, the answer to the text question is true. Viola, you have an infinite loop.

## Definitions

**Do While**

A test after iteration control structure available in C++.

**Action Item**

An attribute of iteration control structures.

**Update Item**

An attribute of iteration control structures.

**Test Item**

An attribute of iteration control structures.

**At Least Once**

Indicating that test after loops execute the action at least once.

**Infinite Loop**

No method of exit, thus a bad thing.

# 7.2: Flag Concept

## Concept Discussion

For centuries flags have been used as a signal to let others know something about the group or individual that is displaying, flying or waving the flag. There are country flags and state flags. Ships at sea flew the flag of their country. Pirates flew the skull and cross bones. A yellow flag was used for quarantine, usually the plague. Even pirates stayed away. Today, some people might recognize the flag used by scuba divers. The Presidents of most countries have a flag. At a race car event they use the checkered flag to indicate the race is over.

**Figure 7.2.1**

Computer programming uses the concept of a flag in the same way that physical flags are used. A flag is anything that signals some information to the person looking at it.

## Computer Implementation

Any variable or constant that holds data can be used as a flag. You can think of the storage location as a flag pole. The value stored within the variable conveys some meaning and you can think of it as being the flag. An example might be a variable named: gender which is of the character data type. The two values normally stored in the variable are: 'F' and 'M' meaning female and male. Then, somewhere within a program we might look at the variable to make a decision:

```
flag controling an if then control substance
if gender equals 'F'
  display "Are you pregnant?"
  get answer from user store in pregnant variable
```

Looking at the flag implies comparing the value in the variable to another value (a constant or the value in another variable) using a relational operator (in our above example: equality).

Control structures are "controlled" by using a **test expression** which is usually a **Boolean expression**. Thus, the flag concept of "looking" at the value in the variable and comparing it to another value is fundamental to understanding how all control structures work.

## Two Flags with the Same Meaning

Sometimes we will use an iteration control structure of do while to allow us to decide if we want to do the loop action again. A variable might be named "loop_response" with the user prompted for their answer of 'y' for yes or 'n' for no. Once the answer is retrieved from the keyboard and stored in our flag variable of "loop_response" the test expression to control the loop might be:

```
Simple Flag Connection
loop_response equals 'y'
```

This is fine but what if the user accidentally has on the caps lock. Then his response of 'Y' would not have the control structure loop and perform the action again. The solution lies in looking at the flag twice. Consider:

```
Complex Flag Comparison
loop_response equals 'y' or loop_response equals 'Y'
In code this would look like:
while ((loop_response == 'y') || (loop_response == 'Y'))
```

We look to see if the flag is either a lower case y or an upper case Y by using a more complex Boolean expression with both relational and logical operators.

## Multiple Flags in One Byte

Within assembly language programming and in many technical programs that control special devices; the use of a single byte to represent several flags is common. This is accomplished by having each one of the 8 bits that make up the byte represent a flag. Each bit has a value of either 1 or 0 and can represent true and false, on or off, yes or no, etc.

## Definitions

**flag**

A variable or constant used to store information that will normally be used to control the program.

Adapted from:
"Flag Concept" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 7.3: Assignment vs Equality within C++

## General Discussion

Most control structures use a **test expression** that executes either selection (as in the: if then else) or iteration (as in the while; do while; or for loops) based on the truthfulness or falseness of the expression. Thus, we often talk about the **Boolean expression** that is controlling the structure. Within many programming languages, this expression must be a Boolean expression and is governed by a tight set of rules. However, in C++ every data type can be used as a Boolean expression, because every data type can be demoted into a Boolean value by using the rule/concept that zero represents false and all non-zero values represent true.

Within C++ we have the potential added confusion of the equals symbol as an operator that does not represent the normal math meaning of **equality** that we have used for most of our life. The equals symbol with C++ means: **assignment**. To get the equality concept of math within C++ we use two equal symbols to represent the relational operator of equality. Let's consider:

```
if (pig = 'y')
{
  cout << "\nPigs are good";
}
else
{
  cout << "\nPigs are bad.";
}
```

The test expression of the control structure will always be true, because the expression is an assignment (not the relational operator of ==). It assigns the 'y' to the variable pig, then looks at the value in pig and determines that it is not zero; therefore the expression is true. And it will always be true and the else part will never be executed. This is not what the programmer had intended. Let's consider:

```
do
{
  cout << "\nPigs are good";
  cout << "\nDo it again, answer y or n: ";
  cin >> do_it_again
}
while (do_it_again = 'y');
```

The loop's test expression will always be true, because the expression is an assignment (not the relational operator of ==). It assigns the 'y' to the variable do_it_again, then looks at the value in do_it_again and determines that it is not zero; therefore the expression is true. And it will always be true and you have just created an infinite loop. As a reminder, infinite loops are not a good thing.

These examples are to remind you that you must be careful in creating your test expressions so that they are indeed a question usually involving the relational operators.

Don't get caught using assignment for equality.

# 7.4: Increment and Decrement Operators

## General Discussion

The idea of increment or decrement is to either add or subtract 1 from a variable that is usually acting as a flag. Using a variable named counter; in generic terms, for example:

```
counter = counter + 1
```

That is you fetch the existing value of the counter and add one then store the answer back into the variable counter. Many programming languages allow their increment and decrement operators to only be used with the integer data type. Programmers will sometimes use inc and dec as abbreviations for increment and decrement respectively.

Operator symbols and/or names vary with different programming languages. The C++ programming language operators with their meanings are:

| C++ Operator | Meaning |
|---|---|
| ++ | increment, two plus signs |
| -- | decrement, two minus signs |

## C++ Code Examples

### Basic Concept

Within the C++ programming language the increment and decrement are often used in this simple generic way. The operator of increment is represented by two plus signs in a row. Examples:

```
counter = counter + 1;
counter += 1;
counter++;
++counter
```

As C++ statements, the four examples all do the same thing. They add 1 to the value of whatever is stored in counter. The decrement operator is represented by two minus signs in a row. They would subtract 1 from the value of whatever was in the variable being decremented.

```
counter = counter - 1;
counter -= 1;
counter--;
--counter
```

So also these 4 statements all do the same, its decrements the value of the variable counter by one.

The precedence of increment and decrement depends on if the operator is attached to the right of the operand (postfix) or to the left of the operand (prefix). Within C++ postfix and prefix do not have the same precedence.

### Postfix Increment

Postfix increment says to use my existing value then when you are done with the other operators; increment me. An example:

```
int age, oldest = 44; // variable set up with initialization
//  then later on in the code
age = oldest++;
```

The first use of the oldest variable is an Rvalue context where the existing value of 44 is pulled or fetched and then assigned to the variable age; then the variable oldest is incremented with its value changing from 44 to 45. This seems to be a violation of precedence because increment is higher precedence than assignment. But that is how postfix increment works within the C++ programming language.

### Prefix Increment

Prefix increment says to increment me now and use my new value in any calculation. An example:

```
int oldest = 44; // variable set up with initialization
  then later on in the code
age = ++oldest;
```

The variable oldest is incremented with the new value changing it from 44 to 45; then the new value is assigned to age.

In postfix age is assigned 44 in prefix age is assigned 45. One way to help remember the difference is to think of prefix, which is **before** the variable - so you increment **before** it is assigned. Likewise, the postfix is **after** the variable, so it is incremented **after** the variable is assigned.

### Allowable Data Types

Within some programming languages, increment and decrement can be used only on the integer data type. C++ however, expands this not only to all of the integer family but also to the floating-point family (float and double). Incrementing 3.87 will change the value to 4.87. Decrementing 'C' will change the value to 'B'. Remember the ASCII character values are really one byte unsigned integers (domain from 0 to 255).

## Exercises

```
Questions
    1. True or false: x = x +1 and x += 1 and x++ all accomplish increment?
    2. Given: int y = 19; and int z; what values will y and z have after: z = y--;
    3. Given: double x = 7.77; and int y; what values will x and y have after: y = ++x
    4. Is this ok? Why or why not? 6 * ++(age -3)
Answers
    1. True
    2. y is: 18 and z is: 19
    3. x is: 8.77 and y is: 8 Note: truncation of 8.77 to 8 upon demotion.
    4. Not ok. Error, the item incremented must have Lvalue attributes, usually a vari
```

## Definitions

**Increment**

Adding one to the value of a variable.

**Decrement**

Subtracting one from the value of a variable.

**Postfix**

Placing the increment or decrement operator to the right of the operand.

**Prefix**

Placing the increment or decrement operator to the left of the operand.

Adapted from:
"Increment and Decrement Operators" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 7.5: While Loop

## Introduction to while Loops

There are two commonly used loops where the condition is tested before entering loop. These loops have the posibility of NEVER entering the body of the loop, depending on the condition. They are: while loop and for loop. This lesson covers the: while.

### Understanding Iteration in General - while

The concept of iteration is connected to possibly wanting to repeat an action. Like all control structures we ask a question to control the execution of the loop. The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the while loop is as follows:

```
initialization of the flag
while the answer to the question is true then do
  some statements or action
  some statements or action
  some statements or action
  update the flag
```

In almost all languages the condition is a **Boolean expression**. The Boolean data type has two values – true and false. Let's rewrite the structure to consider this:

```
initialization of the flag
while the condition is true then do
  some statements or action
  some statements or action
  some statements or action
  update the flag
```

Within the while control structure there are four attributes to a properly working loop. They are:

- Initializing the flag - **this is important**, with while loops you must initialize the flag (some people call this **priming the loop**).
- Test expression
- Action or actions
- Update of the flag - **this also is important** - if we never update the flag we have no way to exit the loop.

The initialization of the flag is not technically part of the control structure, but a necessary item to occur before the loop is started. The English phrasing is, "While the expression is true, do the following actions". This is looping on the true. When the test expression is false, you stop the loop and go on with the next item in the program. Notice, because this is a test before loop the action **might not happen**. It is called a **test before loop** because the test comes before the action. It is also sometimes called a pre-test loop, meaning the test is pre (or Latin for before) the action and update.

### Human Example of the while Loop

Consider the following one-way conversation from a mother to her child.

Child: The child says nothing, but mother knows the child had Cheerios for breakfast and history tells us that the child most likely spilled some Cheerios on the floor.

Mother says: "While it is true that you see (As long as you can see) a Cheerio on floor, pick it up and put it in the garbage."

Note: All of the elements are present to determine the action (or flow) that the child will be doing (in this case repeating). Because the question (can you see a Cheerios) has only two possible answers (true or false) the action will continue while there are Cheerios on the floor. Either the child 1) never picks up a Cheerio because they never spilled any or 2) picks up a Cheerio and keeps picking up Cheerios one at a time while he can see a Cheerio on the floor (that is until they are all picked up).

## The while Structure within C++

### Syntax

The syntax for the while control structure within the C++ programming language is:

```
statement;        // This statement initializes the flag;
while (expression)
{
  statement;
  statement;
  statement;
  statement;    // This statement updates the flag;
}
```

The test expression is within the parentheses, but this is not a function call. The parentheses are part of the control structure. Additionally, there is not a semicolon after the parenthesis following the expression.

### An Example

```
C++ source code: while
loop_response = 'y';   // initialize the flag value
while (loop_response == 'y')
{
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
  cout << "\nDo you want to do it again? y or n ";
  cin >> loop_response;     // reset the flag - changing the value that we are testin
}
```

The four attributes of a test before loop are present. The initialization of the flag. The test is the equality relational comparison of the value in the flag variable to the lower case character of y. The action part consists of the 6 lines that prompt for data and then displays the total of the two ages. The update of the flag is the displaying the question and getting the answer for the variable loop_response.

This type of loop control is called an event controlled loop. The flag updating is an event where someone decides if they want the loop to execute again.

Using indentation with the alignment of the loop actions and flag update is normal industry practice within the C++ community.

### Infinite Loops

At this point it's worth mentioning that good programming always provides for a method to insure that the loop question will eventually be false so that the loop will stop executing and the program continues with the next line of code. However, if this does not happen then the program is in an infinite loop. Infinite loops are a bad thing. Consider the following code:

```
C++ source code: infinite loop
loop_response = 'y';
while (loop_response == 'y')
```

```
{
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
}
```

The programmer assigned a value to the flag before the loop which is correct. However, he forgot to update the flag. Every time the test expression is asked it will always be true. Thus, an infinite loop because the programmer did not provide a way to exit the loop (he forgot to update the flag). Consider the following code:

```
C++ source code: infinite loop
loop_response = 'y';
while (loop_response = 'y')   // assignment NOT a check for equality
  {
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
  cout << "\nDo you want to do it again? y or n ";
  cin >> loop_response;
  }
```

No matter what the user replies during the flag update, the test expression does not do a relational comparison but does an assignment. It assigns 'y' to the variable and asks if 'y' is true? Since all non-zero values are treated as representing true within the Boolean concepts of the C++ programming language, the answer to the test expression is true. Viola, you have an infinite loop.

```
C++ source code: infinite loop
loop_response = 'y';
while (loop_response == 'y');  // accidently placed a ; at the end of the
                               // while statement - this ends the loop
  {
  cout << "\nWhat is your age? ";
  cin >> age_user;
  cout << "\nWhat is your friend's age? ";
  cin >> age_friend;
  cout >> "\nTogether your ages add up to: ";
  cout >> (age_user + age_friend);
  cout << "\nDo you want to do it again? y or n ";
  cin >> loop_response;
  }
```

The undesirable semi-colon on the end of while line causes the action of the while loop to be the "nothingness" between the closing parenthesis and the semi-colon. The program will infinitely loop because there is no action (that is no action and no update). If this is the first item in your program it will appear to start but there will be no output.

## Counting Loops

The examples above are for an event controlled loop. The flag updating is an event where someone decides if they want the loop to execute again. Often the initialization sets the flag so that the loop will execute at least once.

Another common usage of the while loop is as a counting loop. Consider:

```
C++ source code: while loop that is counting
counter = 0;
while (counter < 5)
{
  cout << "\nI love ice cream!";
  counter++;
}
```

The variable counter is said to be controlling the loop. It is set to zero (called initialization) before entering the while loop structure and as long as it is less than 5 (five); the loop action will be executed. But part of the loop action uses the increment operator to increase counter's value by one. After executing the loop five times (once for counter's values of: 0, 1, 2, 3 and 4) the expression will be false and the next line of code in the program will execute. A counting loop is designed to execute the action (which could be more than one statement) a set of given number of times. In our example, the message is displayed five times on the monitor. It is accomplished my making sure all four attributes of the while control structure are present and working properly. The attributes are:

- Initializing the flag
- Test expression
- Action or actions
- Update of the flag

Missing an attribute might cause an infinite loop or give undesired results (does not work properly).

## Infinite Loops

Consider:

```
C++ source code: infinite loop
counter = 0;
while (counter < 5)
{
  cout << "\nI love ice cream!";
  // OOPS - we never update the value of the counter variable - INFINITE LOOP
}
```

Missing the flag update usually causes an infinite loop.

## Variations on Counting

In the following example, the integer variable age is said to be controlling the loop (that is the flag). We can assume that age has a value provided earlier in the program. Because the while structure is a test before loop; it is possible that the person’s age is 0 (zero) and the first time we test the expression it will be false and the action part of the loop would never be executed.

```
C++ source code: while as a counting loop
while (0 < age)
{
  cout << "\nI love candy!";
  age--;     // we can count down to zero...
}
```

Consider the following variation assuming that age and counter are both integer data type and that age has a value:

```
C++ source code: while as a counting loop
age = 5;
counter = 0;
while (counter < age)
{
  cout << "\nI love corn chips!";
  counter++;   // we count up from zero to the appropriate value
}
```

This loop is a counting loop similar to our first counting loop example. The only difference is instead of using a literal constant (in other words 5) in our expression, we used the variable age (and thus the value stored in age) to determine how many times to execute the loop. However, unlike our first counting loop example which will always execute exactly 5 times; it is possible that the person’s age is 0 (zero) and the first time we test the expression it will be false and the action part of the loop would never be executed.

## Definitions

**While**

A test before iteration control structure available in C++.

**Loop Attributes**

Items associated with iteration or looping control structures.

**Initialize Item**

An attribute of iteration control structures.

**Might not Happen**

Indicating that test before loops might not execute the action.

**Event Controlled**

Using user input to control a loop.

**Counting Controlled**

Using a variable to count up or down to control a loop.

Adapted from:
"While Loop" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# CHAPTER OVERVIEW

## 8: Counting Loops

# 8.1: For Loop

## Introduction to Test Before Loops

We have discussed the while loop in the previous module. We now turn out attention to the for loop. The for loop is a great way to process a loop for a finite number of times, meaning that you can programmatically determine how many times you need to loop.

### Understanding Iteration in General - for

In most programming languages the for loop is used exclusively for counting; that is to repeat a loop action as it either counts up or counts down. There is a starting value and a stopping value. The question that controls the loop is a **condition** that compares the starting value to the stopping value. This expression is a Boolean expression and is usually using the relational operators of either less than (for counting up) or greater than (for counting down). The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the for loop (counting up) is as follows:

```
for
  initialization of the starting value
  starting value is less than the stopping value
  some statements or action
  some statements or action
  some statements or action
  increment the starting value
```

It might be best to understand the for loop by understanding a while loop acting like a counting loop. Let's consider;

```
initialization of the starting value
while the starting value is less than the stopping value
  some statements or action
  some statements or action
  some statements or action
  increment the starting value
```

Within the for control structure there are four attributes to a properly working loop. They are:

- Initializing the flag – done once
- Test condition
- Action or actions
- Update of the flag

The initialization of the flag is not technically part of the while control structure, but it is usually part of the for control structure. The English phrasing is, "For x is 1; x less than 3; do the following actions; increment x; loop back to the test expression". This is doing the action on the true. When the test expression is false, you stop the loop and go on with the next item in the program. Notice, because this is a test before loop the action **might not happen**. It is called a **test before loop** because the test comes before the action. It is also sometimes called a pre-test loop, meaning the test is pre (or Latin for before) the action and update.

## The for Structure within C++

### Syntax

The syntax of the for loop control structure within the C++ programming language is:

```
for (initializations; condition; updates)
{
  statement;
  statement;
  statement;
}
```

The initializations, test expression and updates are within the parentheses (each separated by a semi-colon), but this is not a function call. The parentheses are part of the control structure. Additionally, there is not a semicolon after the parenthesis following the expression.

### An Example

```
C++ source code: for
for (counter = 0; counter < 5; counter++)
{
  cout << "Counter is now set to: " << counter << endl;
}
```

The four attributes of a test before loop (remember the for loop is one example of a test before loop) are present.

- The initialization of the counter variable (this is our flag) to a value of 0. **THIS ONLY HAPENS ON THE VERY FIRST ITERATION OF THE LOOP**
- The test is the less than relational comparison of the value in the counter variable to the constant value of 5.
- The cout statement is output
- The update of the counter variable is done with the increment operator.

Using indentation with the alignment of the loop actions is normal industry practice within the C++ community.

### Infinite Loops

At this point it's worth mentioning that good programming always provides for a method to insure that the loop question will eventually be false so that the loop will stop executing and the program continues with the next line of code. However, if this does not happen then the program is in an infinite loop. Infinite loops are a bad thing. Consider the following code:

```
C++ source code: infinite loop
for (counter = 0; counter < 5;)
{
  cout << "\nI love ice cream!";
}
```

The programmer assigned a value to the counter variable (our flag) during the initialization step which is correct. However, he forgot to update the flag (the update step is missing). Every time the test expression is asked it will always be true. Thus, an infinite loop because the programmer did not provide a way to exit the loop (he forgot to update the flag).

### Multiple Items in the Initialization and Update

The following shows the use of the sequence operator to separate the multiple initializations and multiple updates. This is not available in most languages, thus is more unique to the C++ programming language.

```
C++ source code: for with multiple initializations and updates
for (x = 0, y = 10; x < 10; x++, y--)
{
  cout << "The product of x * y is: " << x * y << endl;
}
```

## Counting Loop Conversion - a while into a for

Below is a color coded the conversion of a while loop that displays a message exactly three times (which is a counting loop) into a for loop using C++ programming language syntax. The four loop attributes are color highlighted as follows:

```
blue is the initialize<
orange is the test
green is the action
red is the update

counter = 0;
while (counter < 3)
{
     cout << "Please wrote good code" << endl;
     counter++;
 }

// The for loop that does this same thing is:
 for (counter = 0; counter < 3; counter++)
 {
    cout << "Please wrote good code" << endl;
 }
```

## Definitions

**for**

A test before iteration control structure typically used for counting.

Adapted from:
"For Loop" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 8.2: Range -Based Loops

## Range-based for loop in C++

The concept of a range-based for loop in C++ is added in 2011. It executes a for loop over a range of vallues. Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in an array.

**Syntax :**

```
for ( range_declaration : range_expression )
    loop_statement

Parameters :
range_declaration :
a declaration of a named variable, whose type is the
type of the element of the sequence represented by
range_expression, or a reference to that type.
Often uses the auto specifier for automatic type
deduction.

range_expression :
any expression that represents a suitable sequence
or a braced-init-list.

loop_statement :
any statement, typically a compound statement, which
is the body of the loop.
```

This is a new concept to many programmers who have never programmed in a language that has a range based loop.

Here is a simple piece of code to start with.

```
// Illustration of range-for loop
// using CPP code
#include <iostream>
#include <vector>
using namespace std;

//Driver
int main()
{
    // Iterating over whole array
    vector<int> myVec = {0, 1, 2, 3, 4, 5};
    for (auto countNum : myVec)
        cout << countNum << ' ';

    return 0;
}
```

Several things to pay attention to:

- A new #include statements - <vector>
- We declare a vector - basically an array - that contains 6 elements
- Our for statement is different:
  - we create a variable countNum - each time through the loop this variable is set to the next value in the array - this is similar to the way Python can do a for loop.
- We do not have our typical update of our flag variable

A long, more complex example is below

```
// Illustration of range-for loop
// using CPP code
#include <iostream>
#include <vector>
#include <map>

//Driver
int main()
{
    // This is what we just looked at above Iterating over whole array
    vector<int> myVec = {0, 1, 2, 3, 4, 5};
    for (auto countNum : myVec)
        cout << countNum << ' ';

   cout << '\n';

    // the initializer may be a braced-init-list - so we specify the vector in the fo
    for (int arrNum : {0, 1, 2, 3, 4, 5})
        cout << arrNum << ' ';

    out << '\n';

    // Declare the array then iterate over the array
    int intArr[] = {0, 1, 2, 3, 4, 5};
    for (int arrNum : intArr)
        cout << arrNum << ' ';

    cout << '\n';

    // Just running a loop for every array element
    for (int arrNum : intArr)
        cout << "In loop" << ' ';

    cout << '\n';

    // Printing string characters
    string myStr = "CSP 31A";
    for (char c : myStr)
```

```
        cout << c << ' ';

    cout << '\n';

    return 0;
}
```

Adapted from:
"Range-based for loop in C++" by Rohit Thapliyal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 8.3: Circular Nature of the Interger Data Type Family

## General Discussion

There are times when character and integer data types are lumped together because they both act the same (often called the integer family). Maybe we should say they act differently than the floating-point data types. The integer family values jump from one value to another. There is nothing between 6 and 7 nor between 'A' and 'B'. It could be asked why not make all your numbers floating-point data types. The reason is twofold. First, some things in the real world are not fractional. A dog, even with only 3 legs, is still one dog not three fourths of a dog. Second, the integer data type is often used to control program flow by counting (counting loops). The integer family has a circular wrap around feature. Using a two byte integer, the next number bigger than 32767 is negative 32768 (character acts the same way going from 255 to 0. We could also reverse that to be the next smaller number than negative 32768 is positive 32767. This can be shown by using a normal math line, limiting the domain and then connecting the two ends to form a circle.

**Figure 8.3.1**

This circular nature of the integer family works for both integer and character data types. In theory, it should work for the Boolean data type as well; but in most programming languages it does not for various technical reasons.

"In mathematics, modular arithmetic (sometimes called clock arithmetic) is a system of arithmetic for integers where numbers "wrap around" after they reach a certain value — the modulus. …

A familiar use of modular arithmetic is its use in the 12 hour clock the arithmetic of time-keeping in which the day is divided into two 12 hour periods. If the time is 7:00 now, then 8 hours later it will be 3:00. Usual addition would suggest that the later time should be 7 + 8 = 15, but this is not the answer because clock time "wraps around" every 12 hours; there is no "15 o'clock". Likewise, if the clock starts at 12:00 (noon) and 21 hours elapse, then the time will be 9:00 the next day, rather than 33:00. Since the hour number starts over when it reaches 12, this is arithmetic modulo 12.

**Figure 8.3.2**

Time-keeping on a clock gives an example of modular arithmetic." (Modular arithmetic from Wikipedia)

The use of the modulus operator in integer division is tied to the concepts used in modular arithmetic.

## Implications When Executing Loops

If a programmer sets up a counting loop incorrectly, usually one of three things happen:

- Infinite loop – usually caused by missing update attribute.

- Loop never executes – usually the text expression is wrong with the direction of the less than or greater than relationship needing to be switched.
- Loop executes more times than desired – update not properly handled. Usually the direction of counting (increment or decrement) need to be switched.

Let’s give an example of the loop executing for what appears to be for infinity (the third item on our list).

```
C++ source code
for (int x = 0; x < 10; x--)
{
  cout << x << endl;
}
```

The above code accidently decrements and the value of x goes in a negative way towards -2147483648 (the largest negative value in a normal four byte signed integer data type). It might take a while (thus it might appear to be in an infinite loop) for it to reach the negative 2 billion plus value, before finally decrementing to positive 2147483647 which would, incidentally, stop the loop execution.

## Definitions

**Circular Nature**

Connecting the negative and positive ends of the domain of an integer family data type.

**Loop Control**

Making sure the attributes of a loop are properly handled.

**Modular Arithmetic**

A system of arithmetic for integers where numbers "wrap around".

Adapted from:
"Circular Nature of the Integer Data Type Family" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 8.4: Formatting Output

## Formatted I/O in C++

C++ helps you to format the I/O operations like determining the number of digits to be displayed after the decimal point, specifying number base etc.

**Example:**

- If we want to add + sign as the prefix of out output, we can use the formatting to do so:

```
stream.setf(ios::showpos)
If input=100, output will be +100
```

- If we want to add trailing zeros in out output to be shown when needed using the formatting:

```
stream.setf(ios::showpoint)
If input=100.0, output will be 100.000
```

**Note:** Here, stream is referred to the streams defined in c++ like cin, cout, cerr, clog.

There are two ways to do so:

1. Using the ios class or various ios member functions.
2. Using manipulators(special functions)

1. **Formatting using the ios members:**

   The stream has the format flags that control the way of formatting it means Using this setf function, we can set the flags, which allow us to display a value in a particular format. The ios class declares a bitmask enumeration called fmtflags in which the values(showbase, showpoint, oct, hex etc) are defined. These values are used to set or clear the format flags.

   Few standard ios class functions are:

   1. **width():** The width method is used to set the required field width. The output will be displayed in the given width
   2. **precision():** The precision method is used to set the number of the decimal point to a float value
   3. **fill():** The fill method is used to set a character to fill in the blank space of a field
   4. **setf():** The setf method is used to set various flags for formatting output
   5. **unsetf():** The unsetf method is used To remove the flag setting

Additional ios functions can be found at [std::ios - ios - C++ Reference](std::ios - ios - C++ Reference)

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

int main()
{
        char c = 'A';

        // Adjusting width will be 5.
        cout.width(5);
        cout << c <<"\n";

        int temp = 10;

        // Width of the next value to be
        // displayed in the output will
        // not be adjusted to 5 columns.
        cout << temp << endl;

    return 0;
}
```

The output spaces the 'A' character 5 spaces to the right. The spacing is NOT valid in subsequent cout statements

```
     A
10
```

Following is an expanded program that uses the ios class functions. You can see the impact of the various ios functions

```
#include<iostream>
using namespace std;
// The width() function defines width
// of the next value to be displayed
// in the output at the console.
int main()
{

    cout << "Implementing ios::precision\n\n";
    cout << "Implementing ios::width\n";
    cout.setf(ios::fixed, ios::floatfield);
    cout.precision(2);
    cout<<3.1422;
    cout << "\n--------------------------\n";

    // The fill() function fills the unused
    // white spaces in a value (to be printed
    // at the console), with a character of choice.
    cout << "Implementing ios::fill\n\n";
    char ch = 'a';

    // Calling the fill function to fill
    // the white spaces in a value with a
    // character our of choice.
    cout.fill('*');

    cout.width(10);
    cout<<ch <<"\n";

    int i = 1;
    // Once you call the fill() function,
    // you don't have to call it again to
    // fill the white space in a value with
    // the same character.
    cout.width(5);
    cout<<i;
    cout << "\n--------------------------\n";

    cout << "Implementing ios::setf\n\n";
    int val1 = 100,val2 = 200;
    cout.setf(ios::showpos);
    cout<<val1<<" "<<val2;
    cout << "\n--------------------------\n";


    cout << "Implementing ios::unsetf\n\n";
    cout.setf(ios::showpos|ios::showpoint);
    // Clear the showflag flag without
    // affecting the showpoint flag
    cout.unsetf(ios::showpos);
    cout << 200.0;
    cout << "\n--------------------------\n";

    return 0;
}
```

The output show the formatting that can be accomplished...

```
Implementing ios::precision
Implementing ios::width

3.14
--------------------------

Implementing ios::fill

*********a
****1
--------------------------

Implementing ios::setf

+100 +200
--------------------------

Implementing ios::unsetf
200.00
--------------------------
```

Adapted from:
"Formatted I/O in C++" by Somil Singh, Geeks for Geeks is licensed under CC BY-SA 4.0

# 8.4.1: Formatting Output Continued

## Formatting using Manipulators

The second way you can alter the format parameters of a stream is through the use of special functions called manipulators that can be included in an I/O expression.
The standard manipulators are shown below:

1. **boolalpha:** The boolalpha manipulator of stream manipulators in C++ is used to turn on bool alpha flag
2. **dec:** The dec manipulator of stream manipulators in C++ is used to turn on the dec flag
3. **endl: The endl manipulator of stream manipulators in C++ is used to Output a newline character.**
4. **and:** The and manipulator of stream manipulators in C++ is used to Flush the stream
5. **ends:** The ends manipulator of stream manipulators in C++ is used to Output a null
6. **fixed:** The fixed manipulator of stream manipulators in C++ is used to Turns on the fixed flag
7. **flush:** The flush manipulator of stream manipulators in C++ is used to Flush a stream
8. **hex:** The hex manipulator of stream manipulators in C++ is used to Turns on hex flag
9. **internal**: The internal manipulator of stream manipulators in C++ is used to Turns on internal flag
10. **left**: The left manipulator of stream manipulators in C++ is used to Turns on the left flag
11. **noboolalpha**: The noboolalpha manipulator of stream manipulators in C++ is used to Turns off bool alpha flag
12. **noshowbase**: The noshowbase manipulator of stream manipulators in C++ is used to Turns off showcase flag
13. **noshowpoint**: The noshowpoint manipulator of stream manipulators in C++ is used to Turns off show point flag
14. **noshowpos**: The noshowpos manipulator of stream manipulators in C++ is used to Turns off showpos flag
15. **noskipws**: The noskipws manipulator of stream manipulators in C++ is used to Turns off skipws flag
16. **nounitbuf**: The nounitbuf manipulator of stream manipulators in C++ is used to Turns off the unit buff flag
17. **nouppercase**: The nouppercase manipulator of stream manipulators in C++ is used to Turns off the uppercase flag
18. **oct**: The oct manipulator of stream manipulators in C++ is used to Turns on oct flag
19. **resetiosflags(fmtflags f)**: The resetiosflags manipulator of stream manipulators in C++ is used to Turns off the flag specified in f
20. **right**: The right manipulator of stream manipulators in C++ is used to Turns on the right flag
21. **scientific**: The scientific manipulator of stream manipulators in C++ is used to Turns on scientific flag
22. **setbase(int base)**: The setbase manipulator of stream manipulators in C++ is used to Set the number base to base
23. **setfill(int ch)**: The setfill manipulator of stream manipulators in C++ is used to Set the fill character to ch
24. **setiosflags(fmtflags f):** The setiosflags manipulator of stream manipulators in C++ is used to Turns on the flag specified in f
25. **setprecision(int p):** The setprecision manipulator of stream manipulators in C++ is used to Set the number of digits of precision
26. **setw(int w):** The setw manipulator of stream manipulators in C++ is used to Set the field width to w
27. **showbase**: The showbase manipulator of stream manipulators in C++ is used to Turns on showbase flag
28. **showpoint**: The showpoint manipulator of stream manipulators in C++ is used to Turns on show point flag
29. **showpos**: The showpos manipulator of stream manipulators in C++ is used to Turns on showpos flag
30. **skipws**: The skipws manipulator of stream manipulators in C++ is used to Turns on skipws flag
31. **unitbuf**: The unitbuf manipulator of stream manipulators in C++ is used to turn on unitbuf flag
32. **uppercase**: The uppercase manipulator of stream manipulators in C++ is used to turn on the uppercase flag
33. **ws**: The ws manipulator of stream manipulators in C++ is used to skip leading white space

To access manipulators that take parameters (such as setw( )), you must include “iomanip” header file in your program.

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

int main()
{
        // performs same as setf( ) - adds the = sign to the output
        cout << setiosflags(ios::showpos);
        cout << 123 << endl;

        // This takes the value 100 and turns it into a hex value (which is 64)
        cout << hex << 100 << endl;

        // Set the field width to 10 using setw()
        cout << setfill('*') << setw(10) << 2343.0 << endl;

        return 0;
}
```

Adapted from:
"Formatted I/O in C++" by Somil Singh, Geeks for Geeks is licensed under CC BY-SA 4.0

# 8.5: Nested For Loops

## General Discussion

### Nested Control Structures

We have looked at nested if statements in prior lesson, where one conditional statement was nested within the code black of another if statement. For example:

```
if age is less than 18
  you can't vote
  if age is less than 16
    you can't drive
  else
    you can drive
else
  you can vote
  if age is less than 21
    you can't drink
  else
    you can drink
```

As you can see we simply included as part of the "true action" a statement and another if then else control structure. We did the same (nested another if then else) for the "false action". In our example we nested if then else control structures. Nesting could have an if then else within a while loop. Thus, the concept of nesting allows the mixing of the different categories of control structures.

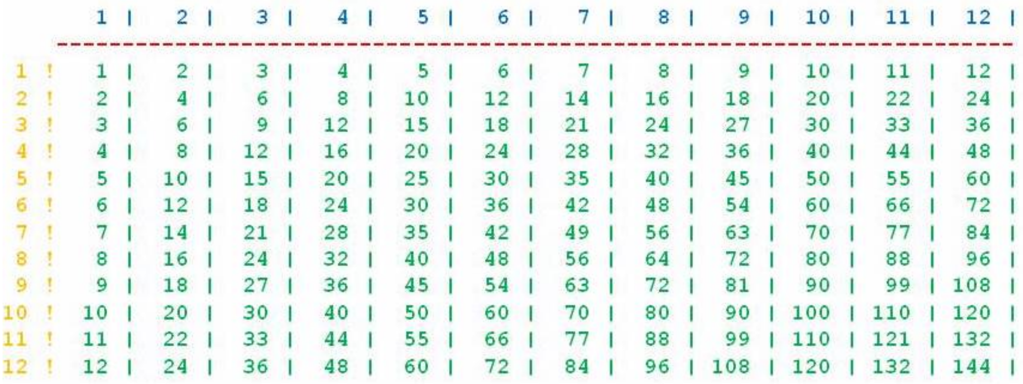
Many complex logic problems require using nested control structures. By nesting control structures (or placing one inside another) we can accomplish almost any **complex logic** problem.

## An Example - Nested for loops

We might also see that the answers could be designed as a collection of cells (each cell being exactly six spaces wide). The C++ source code below shows 4 different loops - the last one is nested inside of another loop. This allows us to process table data by row and column. This also exhibits usage of the setw() function with the cout function.

```
C++ source code: nested for loops - multiplication table
// Prints the top line of blue numerals 1 through 12 separated by |
cout << "      ";
for(across=1; across <13; across++)
{
  cout << setw(4) << across << " |";
}
cout << endl;

// Prints the red linebelow th eblue numerals
cout << "      ";
for(across=1; across <13; across++)
{
  cout << "------";
}
```

```
cout << endl;

// The outer loop prints the yellow numbers down the left side followed by the exclama
for(down=1; down <13; down++)
{
  cout << setw(4) << down << " !";
  // The inner loop prints out the product of each column (down) multiplied be each r
  for(across=1; across <13; across++)
  {
    cout << setw(4) << down*across << " |";
  }
  cout << endl;
}
```



```
         1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |  11 |  12 |
     -------------------------------------------------------------------------
   1 !   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |  11 |  12 |
   2 !   2 |   4 |   6 |   8 |  10 |  12 |  14 |  16 |  18 |  20 |  22 |  24 |
   3 !   3 |   6 |   9 |  12 |  15 |  18 |  21 |  24 |  27 |  30 |  33 |  36 |
   4 !   4 |   8 |  12 |  16 |  20 |  24 |  28 |  32 |  36 |  40 |  44 |  48 |
   5 !   5 |  10 |  15 |  20 |  25 |  30 |  35 |  40 |  45 |  50 |  55 |  60 |
   6 !   6 |  12 |  18 |  24 |  30 |  36 |  42 |  48 |  54 |  60 |  66 |  72 |
   7 !   7 |  14 |  21 |  28 |  35 |  42 |  49 |  56 |  63 |  70 |  77 |  84 |
   8 !   8 |  16 |  24 |  32 |  40 |  48 |  56 |  64 |  72 |  80 |  88 |  96 |
   9 !   9 |  18 |  27 |  36 |  45 |  54 |  63 |  72 |  81 |  90 |  99 | 108 |
  10 !  10 |  20 |  30 |  40 |  50 |  60 |  70 |  80 |  90 | 100 | 110 | 120 |
  11 !  11 |  22 |  33 |  44 |  55 |  66 |  77 |  88 |  99 | 110 | 121 | 132 |
  12 !  12 |  24 |  36 |  48 |  60 |  72 |  84 |  96 | 108 | 120 | 132 | 144 |
```

Figure 8.5.1: Colorized Code - multiplication table

## Definitions

**Complex Logic**

Often solved with nested control structures.

Adapted from:
"Nested For Loops" by Kenneth Busbee, Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2 is licensed under CC BY 4.0

# 8.6: Practice Counting Loops

## Learning Objectives

With 100% accuracy during a: memory building activity, exercises, lab assignment, problems, or timed quiz/exam; the student is expected to:

1. Define the terms on the definitions as listed in the modules associated with this chapter.
2. Identify which selection control structures are commonly used a counting loops.
3. Be able to write pseudo code or flowcharting for the for control structure.
4. Be able to write C++ source code for a for control structure.
5. When feasible, be able to convert C++ source code from while loop acting like a counting loop to a for loop and and vice versa.

## Memory Building Activities

Link to: MBA 15

## Exercises

```
Questions
    1. Only for loops can be counting loops.
    2. The integer data type has modular arithmetic attributes.
    3. The escape code of \n is part of formatting output.
    4. Nested for loops is not allowed in the C++ programming language.
    5. Counting loops use all four of the loop attributes.
Answers
    1. False
    2. True
    3. True
    4. False
    5. True
```

## Lab Assignment

### Creating a Folder or Sub-Folder for Chapter 15 Files

Depending on your compiler/IDE, you should decide where to download and store source code files for processing. Prudence dictates that you create these folders as needed prior to downloading source code files. A suggested sub-folder for the **Bloodshed Dev-C++ 5 compiler/IDE** might be named:

- Chapter_15 within the folder named: Cpp_Source_Code_Files

If you have not done so, please create the folder(s) and/or sub-folder(s) as appropriate.

### Download the Lab File(s)

Download and store the following file(s) to your storage device in the appropriate folder(s). You may need to right click on the link and select "Save Target As" in order to download the file.

Download from Connexions: Lab_15a.cpp

### Detailed Lab Instructions

Read and follow the directions below carefully, and perform the steps in the order listed.

- Compile and run the Lab_15a.cpp source code file. Understand how it works.
- Copy the source code file Lab_15a.cpp naming it: Lab_15b.cpp
- Convert the code that is counting (all four attributes) to a for loop.

- Build (compile and run) your program.
- After you have successfully written this program, if you are taking this course for college credit, follow the instructions from your professor/instructor for submitting it for grading.

## Problems

### Problem 15a - Instructions

Using proper C++ syntax, convert the following for loop to a while loop.

**C++ source code**

```
for (x = 0; x < 10; x++)
  {
  cout << "Having fun!";
  }
```

CHAPTER OVERVIEW

# 9: Introduction to Arrays

# 9.1: Array Data Type

## Overview

An **array** is a sequenced collection of elements of the same data type with a single identifier name. As such, the array data type belongs to the "Complex" category or family of data types. Arrays can have multiple axes (more than one axis). Each axis is a **dimension**. Thus a single dimension array is also known as a **list**. A two dimension array is commonly known as a **table** (a spreadsheet like Excel is a two dimension array). In real life there are occasions to have data organized into multiple dimensioned arrays. Consider a theater ticket with section, row and seat (three dimensions). This module will only cover the single dimension array. Most single dimension arrays are visualized vertically and are often called a list.

Most programmers are familiar with a special type of array called a **string**. Strings are basically a single dimension array of characters. Unlike other single dimension arrays, we usually envision a string as a horizontal stream of characters and not vertically as a list. Within C++ the string data type is a length-controlled array and is a pre-defined data class.

We refer to the individual values as members (or elements) of the array. Programming languages implement the details of arrays differently. Because there is only one identifier name assigned to the array, we have operators that allow us to reference or access the individual members of an array. The operator commonly associated with referencing array members is the **index** operator. It is important to learn how to define an array and initialize its members. Additionally, the **sizeof** operator is often used to calculate the number of members in an array.

## Defining an Array in C++

Example:

```
int ages[5] = {49,48,26,19,16};
```

This is the **defining of storage space**. The square brackets (left [ and right ]) are used here to create the array with five integer members and the identifier name of ages. The assignment with braces (that is a block) establishes the initial values assigned to the members of the array. Note the use of the sequence or comma operator. We could have done it this way:

```
int ages[] = {49,48,26,19,16};
```

By leaving out the five and having initial values assigned, the compiler will know to create the array with five storage spaces because there are five values listed. This method is preferred because we can simply add members to or remove members from the array by changing the items inside of the braces. We could have also done this:

```
int ages[5];
```

This would have declared the storage space of five integers with the identifier name of ages but their initial values would have been unknown values (actually there would be values there but we don’t know what they would be and thus think of the values as garbage). We could assign values later in our program by doing this:

```
ages[0] = 49;
ages[1] = 48;
ages[2] = 26;
ages[3] = 19;
ages[4] = 16;
```

The members of the array go from 0 to 4; **NOT** 1 to 5. This is explained in more detail in another module that covers accessing array members.

## Definitions

**Dimension**

An axis of an array.

**List**

A single dimension array.

**Table**

A two dimension array.

Adapted from:
"Array Data Types" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 9.2: Array Index Operator

## Array Index Operator

Example:

```
int ages[5] = {49, 48, 26, 19, 16};
int my_age;
my_age = ages[2];
```

This second usage of the square brackets is as the **array notation of dereference** or more commonly called the **index operator**. As an operator it either provides the value held by the member of the array (Rvalue) or changes the value of member (Lvalue). In the above example the member that is two offsets from the front of the array (the value 26) is assigned to variable named my_age. The dereference operator of [2] means to go the 2nd **offset** from the front of the ages array and get the value stored there. In this case the value would be 26. The array members (or elements) are referenced starting at zero. The more common way for people to reference a list is by starting with one. Many programming languages reference array members starting at one, however for some languages (and C++ is one of them) you will need to **change your thinking**. Consider:

| Position | C++ | Miss America | Other Contests |
|---|---|---|---|
| zero offsets from the front | ages[0] | Winner | 1st Place |
| one offsets from the front | ages[1] | 1st Runner Up | 2nd Place |
| two offsets from the front | ages[2] | 2nd Runner Up | 3rd Place |
| three offsets from the front | ages[3] | 3rd Runner Up | 4th Place |
| four offsets from the front | ages[4] | 4th Runner Up | 5th Place |

Saying that my cousin is the 2nd Runner Up in the Miss America contest sounds so much better than saying that she was in 3rd Place. We would be talking about the same position in the array of the five finalists.

```
ages[3] = 20;
```

This is an example of changing an array’s value by assigning 20 to the 4th member of the array and replacing the value 19 with 20. This is an Lvalue context because the array is on the left side of the assignment operator.

The C++ operator name is called the array index or simply the index operator and it uses the square brackets as the operator symbols.

## Definitions

**Array Member**

An element or value in an array.

**Index**

An operator that allows us to reference a member of an array.

**Offset**

The method of referencing array members by starting at zero.

Adapted from:
"Array Index Operator" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 9.3: Displaying Array Members

## Accessing Array Members in C++

```
Accessing the members of an array
int ages[] = {49,48,26,19,16};
int counter;

for (counter = 0, counter < 5, counter++)
  {
  cout << ages[counter] << endl;
  }
```

This second usage of the square brackets is as the **array notation of dereference** or more commonly called the **index operator**. As an operator it provides the value held by the member of the array. For example, during one of the iterations of the for loop the index (which is an integer data type) will have the value of 3. The expression ages[counter] would in essence be: ages[3]. The dereference operator of [3] means to go the 3rdoffset from the front of the ages array and get the value stored there. In this case the value would be 19. The array members (or elements) are referenced starting at zero. The more common way for people to reference a list is by starting with one. Many programming languages reference array members starting at one, however for some languages (and C++ is one of them) you will need to **change your thinking**. Consider:

| **Position** | **C++** | **Miss America** | **Other Contestents** |
|---|---|---|---|
| zero offsets from the front | ages[0] | Winner | 1st Place |
| one offsets from the front | ages[1] | 1st Runner Up | 2nd Place |
| two offsets from the front | ages[2] | 2nd Runner Up | 3rd Place |
| three offsets from the front | ages[3] | 3rd Runner Up | 4th Place |
| four offsets from the front | ages[4] | 4th Runner Up | 5th Place |

Saying that my cousin is the 2nd Runner Up in the Miss America contest sounds so much better than saying that she was in 3rd Place. We would be talking about the same position in the array of the five finalists.

Rather than using the for loop to display the members of the array, we could have written five lines of code as follows:

```
cout << ages[0] << endl;
cout << ages[1] << endl;
cout << ages[2] << endl;
cout << ages[3] << endl;
cout << ages[4] << endl;
```

## Using the Sizeof Operator with Arrays in C++

```
Using the sizeof operator
int ages[] = {49,48,26,19,16};
int counter;

for (counter = 0, counter < sizeof ages / sizeof ages[0], counter++)
  {
  cout << ages[counter] << endl;
  }
```

Within the control of the for loop for the displaying of the grades, note that we calculated the number of the members in the array by using the **sizeof** operator. The expression is:

```
sizeof ages / sizeof ages[0]
```

When you ask for the sizeof an array identifier name the answer is how many total bytes long is the array (or in other words – how many bytes of storage does this array need to store its values). This will depend on the data type of the array and the number of elements. When you ask for the sizeof one of its members, it tells you how many bytes one member needs. By dividing the total number of bytes by the size of one member, we get the answer we want: the number of members in the array. This method allows for **flexible coding**. By writing the for loop in this fashion, we can change the declaration of the array by adding or subtracting members and we don't need to change our for loop code.

## Definitions

**Flexible Coding**

Using the sizeof operator to calculate the numer of members in an array.

Adapted from:
"Displaying Array Memebers" by Kenneth Leroy Busbee, (Download for free at http://cnx.org/contents/303800f3-07f...93e8948c5@22.2) is licensed under CC BY 4.0

# 9.4: Finding a Specific Member of an Array

Given an array, find first and last elements of it.

In C++, we can use sizeof operator to find number of elements in an array. Notice we have to calculate the size based on the entire size of memory taken up by the array divided by the sizeof one element of the array.

```
// C++ Program to print first and last element in an array
#include <iostream>
using namespace std;
int main()
{
        int myArr[] = { 4, 5, 7, 13, 25, 65, 98 };
        int first, last, numberOf;

        // Calculate how many elements are in the array
        numberOf = sizeof(myArr) / sizeof(myArr[0]);

        first = myArr[0];
        // use [numberOf - 1] because arrays count from 0
        last = myArr[numberOf - 1];
        cout << "First element: " << first << endl;
        cout << "Last element: " << last << endl;
        // OR you could simply write
        cout << "First element: " << myArr[0] << endl;
        cout << "Last element: " << myArr[numberOf - 1] << endl;
        return 0;
}
```

This is pretty simple...and the output is:

```
First element: 4
Last element: 98
```

We could have made the code simpler by coding the cout statements as:

```
cout << "First element: " << myArr[0] << endl;
cout << "Last element: " << myArr[n - 1] << endl;
```

There is no need to use other variables...but it was an example of how to access elements of the array.

A little more complex code here as an example of how to access array elements.

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

int main()
{
   int newArr[] = { 74, 5, 37, 13, 2, 15, 9 };
   int low, arrSize = sizeof(newArr) / sizeof(newArr[0]);

   // Assign the first element to the variable low
   low = newArr[0];

   // Loop over every element in the array
   for(int index = 0; index < arrSize; index++)
   {
      // If the current array element (newArr[index]) is less than temp
      if(low > newArr[index])
      {
         // Then we assign that current array element to be the new lowest
         low = newArr[index];
      }
   }

   cout << "Smallest Element is: " << low << endl;
   return 0;
}
```

The smallest number is found when we look at every elements and simply track the smallest value.

```
Smallest Element is: 2
```

Adapted from:
"Get first and last elements from Array and Vector in CPP" by ishwarya.27, Geeks for Geeks

# 9.5: Multidimensional Arrays

In C++, we can define multidimensional arrays, an array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

I general, the form of declaring multidimensional arrays:

```
data_type  array_name[size1][size2]....[sizeN];

data_type: This is the type of data to be stored in the array.
           data_type MUST be a valid C++ data type
array_name: Name of the array
size1, size2,... ,sizeN: The size of each of the dimensions
```

**Examples**:

```
Two dimensional array:
int two_d[10][20];

Three dimensional array:
int three_d[10][20][30];
```

## Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
The above array named **two_d[10][20]** can store total (10*20) = 200 elements.
Similarly the array **three_d[10][20][30]** can store total (10*20*30) = 6000 elements.

## Two-Dimensional Array

A two dimensional array is the simplest form of a multidimensional array. We can see a two dimensional array as an array of one dimensional array for easier understanding.

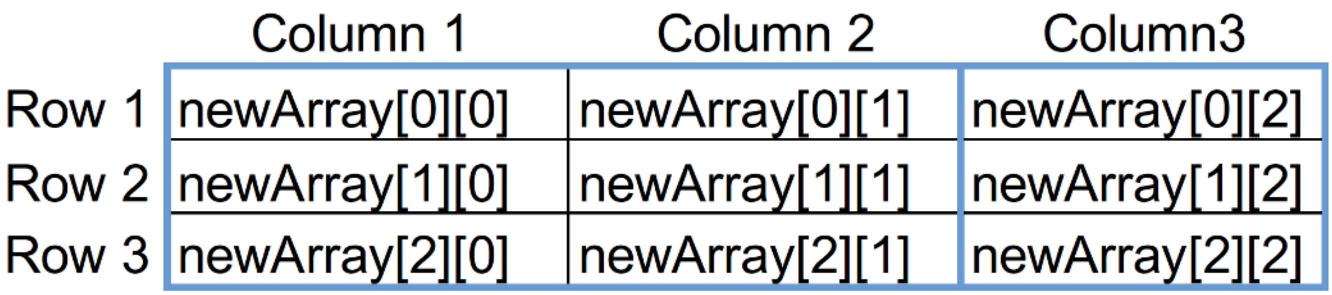
- The basic form of declaring a two-dimensional array of size row, col.
  **Syntax:**

  ```
  data_type array_name[row][col];
  data_type: Type of data to be stored. Valid C/C++ data type.
  ```

- We can declare a two dimensional integer array say 'newArray' of size 10,20 as:

  ```
  int newArray[10][20];
  ```

- Elements in two-dimensional arrays are commonly referred with a syntax of newArray[row][col].
- A two dimensional array can be seen as a table with rows and columns where the row number ranges from 0 to (row-1) and column number ranges from 0 to (col-1). A two dimensional array 'newArray' with 3 rows and 3 columns is shown below:



|  | Column 1 | Column 2 | Column3 |
|---|---|---|---|
| Row 1 | newArray[0][0] | newArray[0][1] | newArray[0][2] |
| Row 2 | newArray[1][0] | newArray[1][1] | newArray[1][2] |
| Row 3 | newArray[2][0] | newArray[2][1] | newArray[2][2] |

Figure 9.5.1: Two dimensional array ("Two dimensional array" by Pat McClanahan, Wikimedia Commans is licensed under CC BY-SA 4.0)

## Initializing Two Dimensional Arrays

There are two ways in which a Two-Dimensional array can be initialized.

**First Method**:

```
int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
```

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be fill the array, the first 4 elements from the left in first row, the next 4 elements in second row and so on.

This is a valid approach, but it can be difficult to read and to debug in the case of a large array.

**Better Method**:

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

This type of initialization make use of nested braces. Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

## Accessing Elements of Two-Dimensional Arrays

Elements in two dimensional arrays are accessed using the both of the indexes.

```
int newArray[2][1];
```

The above example represents the element present in third row and second column.

**Remember**: Arrays always count from 0 therefore, row index 2 is actually the third row, and column index 1 is actually the second column.

To output all the elements of two dimensional array we can use nested for loops. One loop to traverse the rows and another to traverse columns.

```
// C++ Program to print the elements of a
// Two-Dimensional array
#include<iostream>
using namespace std;

int main()
{
    // an array with 3 rows and 2 columns.
    int newArrray[3][2] = {{0,1}, {2,3}, {4,5}};
    // output each array element's value
    for (int row = 0; row < 3; row++)
    {
        for (int col = 0; col < 2; col++)
        {
            cout << "Element at newArrray[" << row
                << "][" << col << "]: ";
            cout << newArrray[row][col]<<endl;
        }
    }

    return 0;
}
```

Output:

```
Element at newArrray[0][0]: 0
Element at newArrray[0][1]: 1
Element at newArrray[1][0]: 2
Element at newArrray[1][1]: 3
Element at newArrray[2][0]: 4
Element at newArrray[2][1]: 5
```

### Three-Dimensional Array

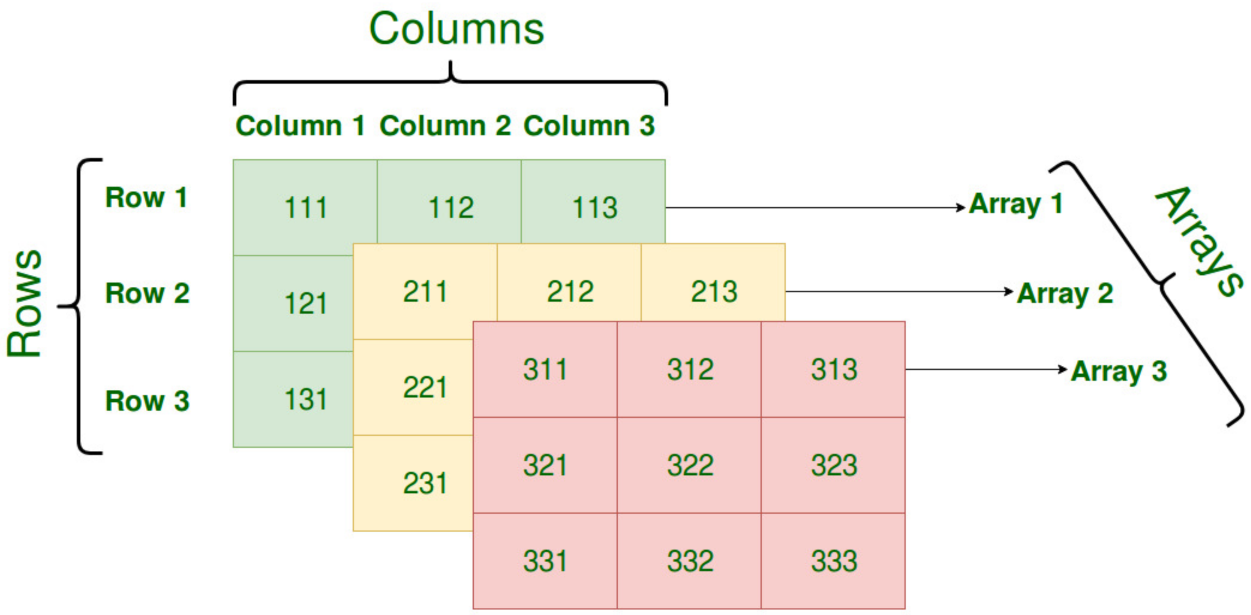


Figure 9.5.1: Three-Dimensional Array ("Three-Dimensional Array" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0)

### Initializing Three-Dimensional Array

Initialization in three dimensional array is same as that of two dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

As with the tewo dimensional arrays, it is allowable to simply initialize a series of values that equals the length of the array.

**Method 1**:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17, 18, 19,
                 20, 21, 22, 23};
```

**Better Method**:

```
int x[2][3][4] =
 {
   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
   { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };
```

### Accessing elements in Three-Dimensional Arrays

Accessing elements in three dimensional arrays is also similar to accessing two dimensional array elements. The difference is we have to use three loops instead of two loops for one additional dimension in three dimensional rrays.

```
// C++ program to print elements of Three-Dimensional
// Array
#include<iostream>
using namespace std;

int main()
{
    // initializing the 3-dimensional array
    int newArray[2][3][2] =
    {
        { {0,1}, {2,3}, {4,5} },
        { {6,7}, {8,9}, {10,11} }
    };

    // output each element's value
    for (int level1 = 0; level1 < 2; ++level1)
    {
        for (int level2 = 0; level2 < 3; ++level2)
        {
            for (int level3 = 0; level3 < 2; ++level3)
            {
                cout << "Element at newArray[" << level1 << "][" << level2
                    << "][" << level3 << "] = " << newArray[level1][level2][level3]
                    << endl;
            }
        }
    }
    return 0;
}
```

Output:

```
Element at newArray[0][0][0] = 0
Element at newArray[0][0][1] = 1
Element at newArray[0][1][0] = 2
Element at newArray[0][1][1] = 3
Element at newArray[0][2][0] = 4
Element at newArray[0][2][1] = 5
Element at newArray[1][0][0] = 6
Element at newArray[1][0][1] = 7
Element at newArray[1][1][0] = 8
Element at newArray[1][1][1] = 9
Element at newArray[1][2][0] = 10
Element at newArray[1][2][1] = 11
```

In similar ways, we can create arrays with any number of dimension. However the complexity also increases as the number of dimension increases.

Adapted from:
"Multidimensional Arrays in C / C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# CHAPTER OVERVIEW

## 10: Functions

# 10.1: Functions - Declarations and Definitions

## Functions in C++

A function is a set of statements that take zero or more arguments, does some specific computation and returns some value. All C++ programs have at least one function - that is the main() function.

The idea is to put code that is repeated multiple times into a function so that instead of writing the same code again and again for different inputs, we can call the function.

**Why do we need functions?**

- Functions help us in reducing code redundancy. If the same functionality is necessary at multiple places in the project, then rather than writing the same code, again and again, we create a function and call it whenever needed. This also helps with code maintenance as we only have to change the code in one place and that change is good for every time the function gets called.
- Consider a big project with thousands of lines of code. Using functions helps to reduce the number of lines of code, makes the code easier to read and make code maintenance easier.
- Consider library functions, we can call them without worrying about how they work, we can simply call the function, pass any necessary arguments and accept the returned results.

There are two tasks that are required when coding functions: provide a function prototype and provide the function definition.

### Prototypes

The function prototype only contains the declaration of the function. For instance take the following example of a prototype for a function named Sum().

```
int Sum( int, int );
```

First of all, function prototypes include the function signature, which includes, the name of the function, return type and argument number and type In this case the name of the function is "Sum". The function signature determines the number of parameters, in this instance two, and their types, both are int values. In the above example, the return type is "int", meaning the function returns an integer value back to the calling function. Notice that in the function prototype you are NOT required to provide variable names, just the types of the arguments is required. The purpose of the prototype is to tell the compiler that there is a function named, in our example "Sum", and it accepts, in this example 2 integer arguments, and returns a value, in our example a "void" value, meaning we don't pay attention to the return value.

Any number of arguments (also called parameters) can be passed, and you can mix any of the valid types in the list.

```
int myFunction(float, char, int);
```

### Function Definition

Since the prototype does not contain any actual code, we must also prvide the function definition, which is where the code is that gets executed when the function is called. Functions should do something specific, and then return to the calling location in the code. Your functions should not add a bunch of other processing that is not part of the specific processing. This takes practice to know when to return to the calling code and continue the processing there.

The following is a simple example and includes the prototype and the definition, as well as a function call from main():

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

// Prototype is here, BEFORE it is used
// This is the function declaration
```

```
void output(char);

int main()
{
    char newCh = 'A';

    output(newCh);

    return 0;
}

// Function definition - includes the code for
// the function
void output(char inCh)
{
    cout << "The value is " << inCh << endl;
    return;
}
```

Some of this code looks familiar. On line 6 we have the function prototype - simply tells the compile that there is a function called output, takes a single character argument, and return no value -  it is type void. There is no code for this prototype yet.

In main() we declare a character variable named newCh and assign it the value of 'A'. Then we actually make the function call, passing the proper type of argument - a character.

At the bottom of our code we have the function definition, where the code for this function is provided. Now, on the first line of the function we have a variable type AND a variable name - inCh. **A WORD OF WARNING** - do not use the same variable name here that you used when you called the function - for instance do NOT call both of your variables newCh or inCh. While this is actually allowed in C++ it will generate confusion and in this class you will lose points for coding in that manner. Also - notice the return statement does not actually return a value - because this is a void function. If you put a value in the return statement you will get a compile error.

When the function returns, control of the program returns the the line of code in main() where the function was called, and execution continues from that point.

# Definitions

**User Defined Library**

A file containing specific task functions created by individuals to be used in many programs.

Adapted from:
"Functions in C/C++" by Unknown, Geeks for Geeks CC BY-SA 4.0
"Function prototype" by Numerous Contributors, Wikipedia is licensed under CC BY-SA 4.0

# 10.2: Scope - Global vs Local

## Scope of Variables in C++

**(UNDERSTAND THAT IN THESE SIMPLE EXAMPLES WE HAVE NOT PROVIDED PROTOTYPES IN ORDER TO SAVE SPACE)**

In general, the scope is defined as the extent up to which something can be worked with. In programming also the scope of a variable is defined as the extent of the program code within which the variable can we accessed or declared or worked with. There are mainly two types of variable scopes:

1. Local Variables
2. Global Variables

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

// This is a global variable
// It is defined outside of any block of code
int myGlobal = 5;

int main()
{
    // This is a local variable
    int myLocal = 2;

    return 0;
}
```

### Local Variables

Variables defined within a function or block are said to be local to those functions.

- Anything between ‘{‘ and ‘}’ is said to inside a block.
- Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block.
- **Declaring local variables**: Local variables are declared inside a block.

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void thisFunc()
{
    // this variable is local to the
    // function thisFunc() and cannot be
    // accessed outside this function
    int age=18;
    return;
}
```

```
int main()
{
    cout << "Age is: " << age << endl;

    return 0;
}
```

Output:

```
Error: age was not declared in this scope
```

The above program displays an error saying “age was not declared in this scope”. The variable age was declared within the function func() so it is local to that function and not visible to portion of program outside this function.

**Problem Solved**: To correct the above error we have to display the value of variable age from the function someFunc() only. This is shown in the below program:

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void someFunc()
{
    // this variable is local to the
    // function someFunc() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}

int main()
{
    // We begin the output
    cout<<"Age is: ";
    // Then call the function to complete it
    someFunc();

    return 0;
}
```

In this case, things work just fine, since we are not attempting to access an unknown variable.

```
Age is: 18
```

### Global Variables

As the name suggests, Global Variables can be accessed from any part of the program.

- They are available through out the life time of a program.
- They are declared at the top of the program outside all of the functions or blocks.
- **Declaring global variables**: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

```
// CPP program to illustrate
// usage of global variables
#include<iostream>
using namespace std;

// global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    cout << global << endl;
}

// main function
int main()
{
    // Call display() function - value of global variable is 5
    display();

    // changing value of global
    // variable from main function
    global = 10;
        // Call display() function - value of global variable is 10
    display();
}
```

Global variables are accessible to any function throughout the file.

```
5
10
```

In the program, the variable “global” is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

### Duplicate names

What if there is a variable inside a function with the same name as that of a global variable and if the function tries to access the variable with that name, then which variable will be given precedence? Local variable or Global variable? Look at the below program to understand the question:

```
// CPP program to illustrate
// scope of local variables
// and global variables together
#include<iostream>
using namespace std;

// global variable
int twoNames = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable

    int twoNames = 2;
    cout << twoNames << endl;
}
```

Look at the above program. The variable “twoNames” declared at the top is global and stores the value 5 where as that declared within main function is local and stores a value 2. So, the question is when the value stored in the variable named “global” is printed from the main function then what will be the output? 2 or 5?

Here in the above program also, the local variable named “global” is given precedence. So the output is 2.

- Usually when two variable with same name are defined then the compiler produces a compile time error. But if the variables are defined in different scopes then the compiler allows it.
- Whenever there is a local variable defined with same name as that of a global variable then the **compiler will give precedence to the local variable**

### Scope Resolution

What if we want to do the opposite of above task. What if we want to access global variable when there is a local variable with same name?

To solve this problem we will need to use the **scope resolution operator**. Below program explains how to do this with the help of scope resolution operator.

```
// C++ program to show that we can access a global
// variable using scope resolution operator :: when
// there is a local variable with same name
#include<iostream>
using namespace std;

// Global
int newVal = 0;

int main()
{
   // Local x
   int newVal = 10;
   cout << "Value of global newVal is " << ::newVal;
   cout << "\nValue of local newVal is " << newVal;

   return 0;
}
```

Output:

```
Value of global newVal is 0
Value of local newVal is 10
```

## Definitions

**Global Scope**

Data storage defined outside of a function.

**Local Scope**

Data storage defined inside of a function.

**Data Area**

A part of an object code file used for storage of data.

**Stack**

A part of the computer's memory used for storage of data.

**Scope**

The area of a source code file where an identifier name is recognized.

Adapted from:
"Scope of Variables in C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 10.3: Arguments vs Parameters

## Argument vs Parameter in C++

### Argument

An **argument** is referred to the values that are passed within a function when the function is called. These values are generally the source of the function that require the arguments during the process of execution. These values are assigned to the variables in the definition of the function that is called. The type of the values passed in the function is the same as that of the variables defined in the function definition. These are also called **Actual arguments** .

**Example:** Suppose a sum() function is needed to be called with two numbers to add. These two numbers are referred to as the arguments and are passed to the sum() when it called from somewhere else.

```
// C++ code to illustrate Arguments
#include <iostream>
using namespace std;

// sum: Function defintion
int sum(int in1, int in2)
{
    // returning the addition
    return in1 + in2;
}

// main() code
int main()
{
    int num1 = 10, num2 = 20, results;

    // sum() is called with
    // num1 & num2 as ARGUMENTS.
    // ALSO NOTICE - the sum() funtion returns an integer value
    // and the variable results will now contain that value
    // because of the assignment statement
    results = sum(num1, num2);

    // Displaying the result
    cout << "The summation is " << res;
    return 0;
}
```

Make sure you take notice of the assignment of the returned value: results = sum(num1, num2); - we will discuss this more in a few lessons

```
The summation is 30
```

### Parameters

The parameter is referred to as the variables that are defined during a function declaration or definition. These variables are used to receive the arguments that are passed during a function call. These parameters within the function prototype are used during the execution of the function for which it is defined. These are also called Formal arguments or Formal Parameters. These are also called **Actual Parameters.**

```
// C++ code to illustrate Parameters

#include <iostream>
using namespace std;

// Function declaration
// the 2 ints are the parameters
int Mult(int, int);

// Driver code
int main()
{
    int num1 = 10, num2 = 20, results;

    // Mult() is called with
    // num1 & num2 as ARGUMENTS.
    results = Mult(num1, num2);

    // Displaying the result
    cout << "The multiplication is " << results;
    return 0;
}

// Mult: Function defintion
// numIn1 and numIn2 are the parameters
int Mult(int numIn1, int numIn2)
{
    // returning the results of the multiplication
    return numIn1 * numIn2;
}
```

Again - arguments are the values that are passed in the function call. See the 3 highlighted comments in the code above, including the function prototype declaration

```
The multiplication is 200
```

**Difference between Argument and Parameter**

| ARGUMENT | PARAMETER |
|---|---|
| When a function is called, the values that are passed during the call are called as arguments. | The values which are defined at the time of the function prototype or definition of the function are called as parameters. |
| These are used in function call statement to send value from the calling function to the receiving function. | These are used in function header of the called function to receive the value from the arguments. |

| ARGUMENT | PARAMETER |
|---|---|
| During the time of call each argument is always assigned to the parameter in the function definition. | Parameters are local variables which are assigned value of the arguments when the function is called. |
| They are also called Actual Parameters | They are also called Formal Parameters |
| **Example:**<br>`int num = 20;`<br>`someFunc(num)`<br>`// num is an argument` | **Example:**<br>`int someFunc(int rnum)`<br>`{`<br>`    cout << "The value is " << rnum << endl;`<br>`}`<br>`// rnum is parameter` |

Go

# 10.4: Pass by Value vs Pass by Reference

## Argument Passing Techniques in C/C++

There are different ways in which parameter can be passed into and out of methods and functions. Let us assume that a function *B()* is called from another function *A()*. In this case *A* is called the ***"caller function"*** and *B* is called the ***"called function or callee function"***. Also, the arguments which *A* sends to *B* are called *actual arguments* and the parameters of *B* are called *formal arguments*.

### Terminology

- **Formal Parameter :** A variable and its type as they appear in the prototype of the function or method.
- **Actual Parameter :** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.
- **Modes:**
  - **IN:** Passes info from caller to calle.
  - **OUT:** Callee writes values in caller.
  - **IN/OUT:** Caller tells callee value of variable, which may be updated by callee.

**Important methods of Parameter Passing**

1. **Pass By Value :** This method uses *in-mode* semantics. Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment.

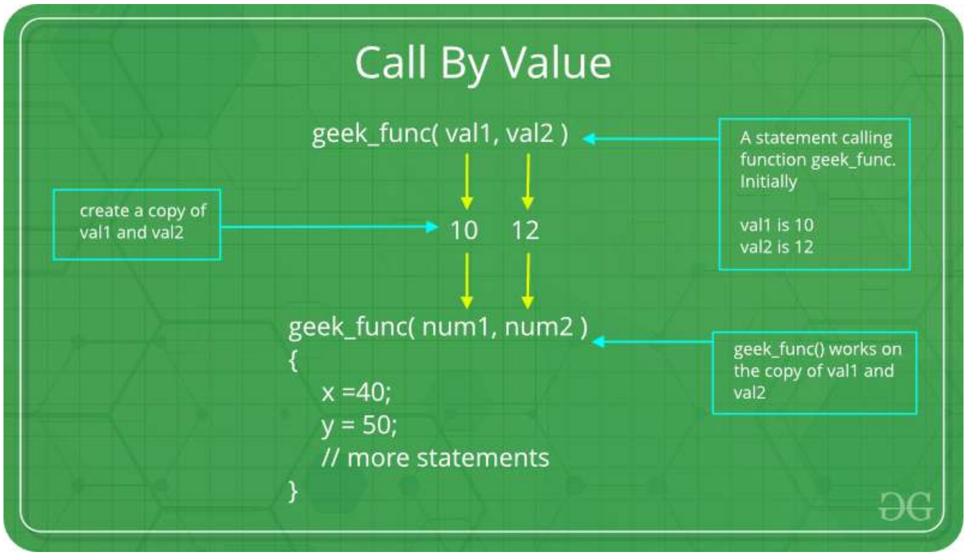

Call By Value
geek_func( val1, val2 )
A statement calling function geek_func. Initially
val1 is 10
val2 is 12
create a copy of val1 and val2
10
12
geek_func( num1, num2 )
{
x =40;
y = 50;
// more statements
}
geek_func() works on the copy of val1 and val2
GG

Figure 10.4.1: Call by Value. ("Call by Value" by Geeks for Geeks, Geek for Geeks is licensed under CC BY-SA 4.0)

This method is also called as ***call by value***. In the example to the left, the call to geek_func passes the value of 2 arguments, 10, and 12. The copies of the 2 values are known within the function itself as num1 and num2. Assigniing the value 40 to the variabl num1 and 50 to the variable num2 within the geek_func() only affect the variable num1 and num2. Those changes have no impact on the variable val1 or val2...they continue to maintain the values of 10 and 12. C++ is passing the VALUE of val1 and val2...therefore it is just a copy and impacts nothing else.

```
// C program to illustrate
// call by value
#include <iostream>
using namespace std;
```

```
void func(int inVal1, int inVal2)
{
    inVal1 += inVal2++;
    cout << "In func, inVal1 = " << inVal1 << " inVal2 = " << inVal2 << endl;
    return;
}

int main(void)
{
    int num1 = 5, num2 = 7;      // Passing parameters
    func(num1, num2);
    cout << "In main, num1 = " << num1 << " num2 = " << num2 << endl;
    return 0;
}
```

The two cout statements show the the changes in the function do NOT impact the original variables in the main() function.

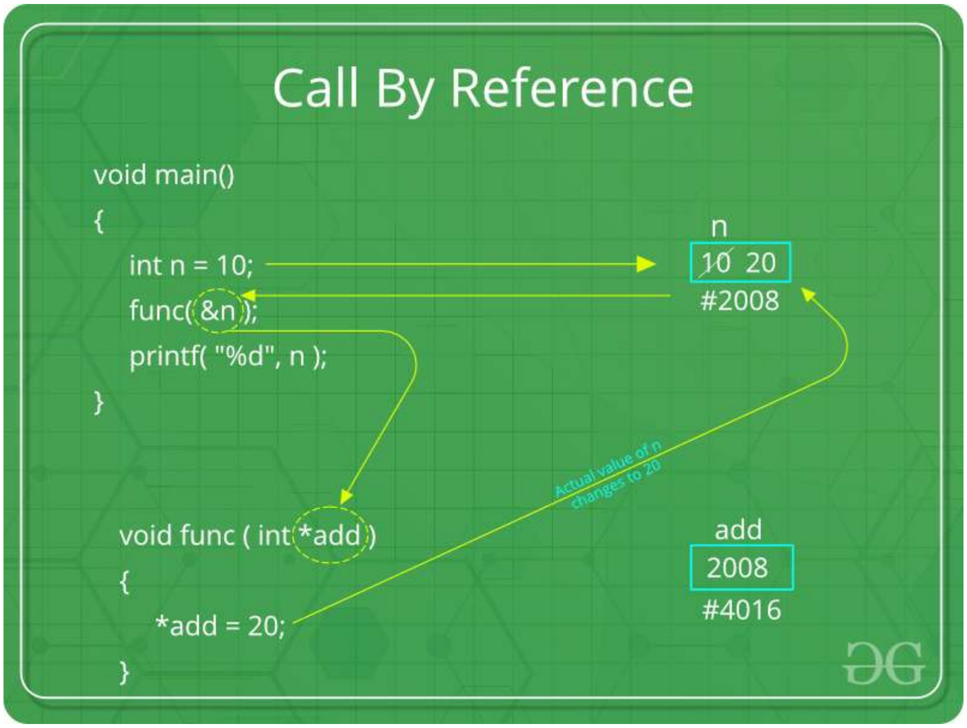
```
In func, inVal1 = 12 inVal2 = 8
In main, num1 = 5 num2 = 7
```

1. **Pass by reference(aliasing) :** This technique uses *in/out-mode* semantics. Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **call by reference**. This method is efficient in both time and space.

In the example to the right, we have an integer variable named `n`, with a value of 10. This variable is used in the function call func(). Notice the ampersand, &, in front of the argument `n`. This tells the compiler to pass the address of `n` - in our example that value is memory address 2008.

Now in the function func() we see the argument variable `add` has an asterisk in front of it. This syntax says that the variable `add` is a pointer, that is the value of that variable is an address. So, the statement `*add = 20`; says, "Set the memory address contained in the variable add, which is 2008, to 20". So the original variable `n`, which is at memory address 2008, is set to 20.

We will talk a lot more about addresses and pointers later in the semester...so if you don't understand this completely that's okay at this point.



## Call By Reference

```
void main()
{
    int n = 10;
    func( &n );
    printf( "%d", n );
}

void func ( int *add )
{
    *add = 20;
}
```

Figure 10.4.1: Pass by Reference. ("Call by Reference" by Geeks for Geeks, Geek for Geeks is licensed under CC BY-SA 4.0)

```
// C program to illustrate
// call by reference
#include <iostream>

void swapnum(int* input1, int* input2)
{
   int temp = *input1;
   *input1 = *input2;
   *input2 = temp;
   return;
}

int main(void)
{
   int val1 = 10, val2 = 20;

   // passing parameters
   swapnum(&val1, &val2);

   cout << "val1 is " << val1 << " and val2 is " << val2 << endl;
   return 0;
}
```

Output:

```
val1 is 20 and val2 is 10
```

Adapted from:
"Parameter Passing Techniques in C/C++" by Krishna Bhatia, Geeks for Geeks is licensed under CC BY-SA 4.0

# 10.5: Function Return Types

## Return Statement in C++

The **return statement** returns the flow of the execution to the location from where it is called. As soon as the return statement is executed, the **flow of the program stops immediately** and return the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void type of function, a return value must be returned.

There are various ways to use return statements. Few are mentioned below:

- **Methods not returning a value:** In C/C++ one cannot skip the return statement, when the methods are of return type. The return statement can be skipped only for void types
  - **Not using return statement in void return type function:** When a function does not return anything, the void return type is used. So if there is a void return type in the function definition, then there will be no return statement inside that function (generally).
  - Example:

```
// C++ code to show not using return
// statement in void return type function
#include <iostream>
using namespace std;

// void method
void Print()
{
    cout <<"Welcome to CSP 31A") << endl;
}

int main()
{
    // Calling print
    Print();

    return 0;
}
```

The function is called, and processes, when done the control of the program returns to where the function was called. The output of the above code example is as follows:

```
Welcome to CSP 31A
```

### Void function with a return statement

Now the question arises, what if there is a return statement inside a void return type function? Since we know that, if there is a void return type in the function definition, then there will be no return statement inside that function. But if there is a return statement inside it, then also there will be no problem if the syntax of it will be:

**Correct Syntax:**

```
void func()
{
    return;
}
```

This syntax is used in function to signify that the code is indeed to return to the calling location, and this is what the programmer intended.

```
// C++ code to show using return
// statement in void return type function
#include <iostream>
using namespace std;

// void method
void Print()
{
    printf("Welcome to CSP 31A");

    // void method using the return statement
    return;
}

// Driver method
int main()
{
    // Calling print
    Print();
    return 0;
}
```

There is no difference here, the return statement adds nothing to this code. Many organizations require a return statement, just for clarification that it is indeed the desire of the programmer to return at this point.

Output is the same.

```
Welcome to CSP 31A
```

### Invalid return from a void function

But if the return statement tries to return a value in a void return type function, that will lead to errors. So if we have a void function, and attempt to return a value like the example below

```
void func()
{
    return value;
}
```

When you attempt to compile this code you would receive a message stating that you can not do that. Some compilers give you a warning, others give you an error, which is correct, in that this is against the rules of C++.

```
warning: 'return' with a value, in function returning void
OR
```

```
error: return-statement with a value, in function returning 'void'
```

- **Methods returning a value:** For methods that define a return type, the return statement must be immediately followed by the return value of that specified return type.

  **Syntax:**

  ```
  return-type func()
  {
      return value;
  }
  ```

  When a function has a return type, the value being returned should be that type.  Any value that is of a different type will be forced into that type before it is returned. The easiest thing is to make sure you are returning the proper type of value.

  ```
  // C++ code to illustrate Methods returning
  // a value using return statement

  #include <iostream>
  using namespace std;

  // integer return type - the return value MUST be an int
  // function to calculate sum
  int SUM(int inputV1, int inputV2)
  {
      int s1 = inputV1 + inputV2;

      // method using the return statement to return a value
      // It will force any value to an integer before it is returned
      return s1;
  }

  // Driver method
  int main()
  {
      int num1 = 10;
      int num2 = 10;
      int sum_of;

      // The SUM() return value is being assigned to an int
      sum_of = SUM(num1, num2);
      cout << "The sum is " << sum_of;
      return 0;
  }
  ```

  **Output:**

  ```
  The sum is 20
  ```

Adapted from:
"return statement in C/C++ with Examples" by Chinmoy Lenka, Geeks for Geeks is licensed under CC BY-SA 4.0

# 10.6: Default Arguments

## Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn’t provide a value for the argument with a default value.

Following is a simple C++ example to demonstrate the use of default arguments. We don’t have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

All 3 of the function calls work because - in the first instance, where there are only 2 arguments, the defaults set the values of z and w. In the second instance the only defaults value in use is t=for the variable w. The last example - no default values are in use, as all 4 arguments are passed from the calling function.

Output:

```
25
50
80
```

When function overloading (we will talk about function overloading in the next sections) done along with default values. Then we need to make sure it will not be ambiguous.
The compiler will throw error if ambiguous. Following is the modified version of above program.

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}
int sum(int x, int y, float z=0, float w=0)
{
    return (x + y + z + w);
}
/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

In this code we now have an overloaded function, with the

**Error:**

```
prog.cpp: In function 'int main()':
prog.cpp:17:20: error: call of overloaded
'sum(int, int)' is ambiguous
  cout << sum(10, 15) << endl;
                    ^
prog.cpp:6:5: note: candidate:
int sum(int, int, int, int)
 int sum(int x, int y, int z=0, int w=0)
     ^
prog.cpp:10:5: note: candidate:
int sum(int, int, float, float)
 int sum(int x, int y, float z=0, float w=0)
     ^
```

**Key Points:**

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.
- Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

```
// Invalid because z has default value, but w after it
// doesn't have default value
int sum(int x, int y, int z=0, int w)
```

Adapted from:
"Default Arguments in C++" by Multiple contributors, Geeks for Geeks is licensed under CC BY-SA 4.0

# 10.7: Function Overloading

## Function Overloading in C++

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters and behave differently based on the types of arguments passed from the calling function.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

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

void print(int i)
{
   cout << " Here is int " << i << endl;
}

void print(double f)
{
   cout << " Here is float " << f << endl;
}

void print(char const *c)
{
   cout << " Here is char* " << c << endl;
}

int main()
{
   print(10);
   print(10.10);
   print("ten");

   return 0;
}
```

In the above code, the first call to the print() function passes an integer argument. The concept of overloading then ensures that the correct function is called with the appropriate parameter specified. So, that is how C++ knows exactly which function to call - by the argument list provided by the function call. The second call to print() has a float/double argument type - so the appropriate function gets called. The same is true with the thrid print() function call - a character string is passed in. The output of the above looks like the following then.

```
Here is int 10
Here is float 10.1
Here is char* ten
```

If we go back to our discussion of default parameters from the last lesson, when function overloading done along with default values we need to make sure it will not be ambiguous.

The compiler will throw error if ambiguous. Take a look at the following code, this will cause an error when compiled

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}
int sum(int x, int y, float z=0, float w=0)
{
    return (x + y + z + w);
}
/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

In this code we now have an overloaded function, with the parameters slightly changed between the two functions. The first call to the sum() function is ambiguous because C++ can not determine which of the function is being called. Both functions can handle the function call with 2 absolute parameters, but since the signature of both functions have 4 arguments it can not determine which one to call. So, in this instance we need to provide at least a thrid argument to the function call which will be either an integer or a float value, THEN we know which function is being called.

```
prog.cpp: In function 'int main()':
prog.cpp:17:20: error: call of overloaded
'sum(int, int)' is ambiguous
  cout << sum(10, 15) << endl;
                    ^
prog.cpp:6:5: note: candidate:
int sum(int, int, int, int)
 int sum(int x, int y, int z=0, int w=0)
     ^
prog.cpp:10:5: note: candidate:
int sum(int, int, float, float)
 int sum(int x, int y, float z=0, float w=0)
     ^
```

**Key Points:**

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.

- Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.
- **Once default value is used for an argument in function definition, all subsequent arguments to it must have default value.** It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

Also - the following gives us a error. the variable w NEEDS a default value according to the rules above.(highlighted statement)

```
// Invalid because z has default value, but w after it
// doesn't have default value
int sum(int x, int y, int z=0, int w)
```

Adapted from:
"Function Overloading in C++" by raHUL GUPTA 23, Geeks for Geeks is licensed under CC BY-SA 4.0
"Default Arguments in C++" by Multiple contributors, Geeks for Geeks is licensed under CC BY-SA 4.0

# 10.7.1: Function Overloading - Cannot be Overloaded

## Functions that cannot be overloaded in C++

In C++, following function declarations **cannot** be overloaded.

1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
#include<iostream>
int foo() {
return 10;
}

char foo() {
return 'a';
}

int main()
{
char x = foo();
getchar();
return 0;
}
```

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

```
#include<iostream>
class Test {
static void fun(int i) {}
void fun(int i) {}
};

int main()
{
Test t;
getchar();
return 0;
}
```

3) Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)()); // redeclaration of h(int())
```

5) Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error *"redefinition of `int f(int)' "*

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

using namespace std;

int f ( int x) {
    return x+10;
}

int f ( const int x) {
    return x+10;
}

int main() {
getchar();
return 0;
}
```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T,
"pointer to T," "pointer to const T," and "pointer to volatile T" are considered distinct parameter types, as are "reference to T," "reference to const T," and "reference to volatile T." For example, see the example in this comment posted by Venki.

6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *"redefinition of `int f(int, int)' "*

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

using namespace std;

int f ( int x, int y) {
    return x+10;
}

int f ( int x, int y = 10) {
    return x+y;
}

int main() {
```

```
getchar();
return 0;
}
```

Adapted from:
"Functions that cannot be overloaded in C++" by Author Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0

# CHAPTER OVERVIEW

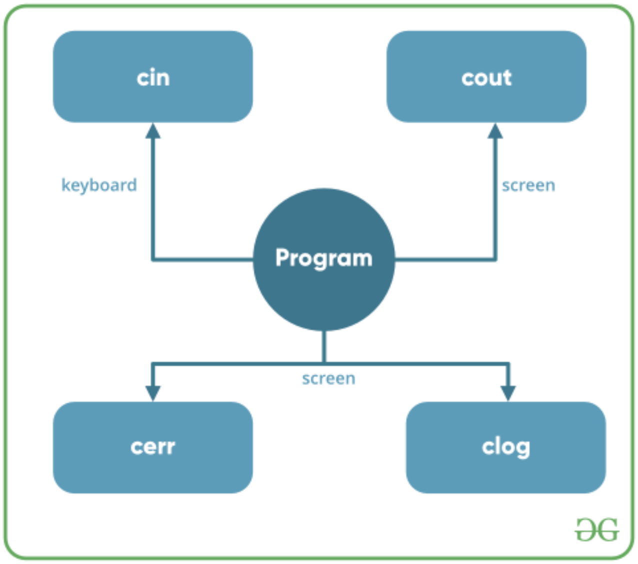
## 11: C++ Input and Output

# 11.1: C++ Input-Output Streams

## Basic Input / Output in C++

C++ comes with libraries which provides us with many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device( display screen ) then this process is called output.



**Header files available in C++ for Input/Output operations are:**

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.
2. **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The two keywords **cout in C++** and **cin in C++** are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.

This lesson mainly discusses the objects defined in the header file <iostream> like cin and cout.

cin and cout are NOT reserved words, they are actually variables, instances of classes, that have been declared in <iostream>. cout is a variable of type ostream. cin is a variable of type istream.

1. **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).*filter_none*

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

int main()
{
    char sample[] = "GeeksforGeeks";

    cout << sample << " - A computer science portal for geeks" << endl;

    return 0;
}
```

In the above program the insertion operator(<<) inserts the value of the string variable **sample** followed by the string “A computer science portal for geeks” in the standard output stream **cout** which is then displayed on screen. Notice that you have to insert spaces in the second string. The output of the above program would be:

```
GeeksforGeeks - A computer science portal for geeks
```

2. **standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.
The extraction operator(>>) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keboard.*filter_none*

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

int main()
{
    int age;

    cout << "Enter your age:";
    cin >> age;
    cout << "Your age is: " << age << endl;

    return 0;
}
```

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(>>) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

**Input :**

```
18
```

**Output:**

```
Enter your age:
Your age is: 18
```

The other 2 streams are unbuffered. The difference between buffered and unbuffered streams is that a buffered stream does not immediately send the data to the destination, but instead, it buffers incoming data and then sends it in blocks. An unbuffered stream, on the other hand, immediately sends the data to the destination. Buffering is usually done to improve performance, as certain destinations, such as files, perform better when writing bigger blocks at once

3. **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream which is used to output the errors. This is also an instance of the iostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display later.

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

```

int main()
{
    cerr << "An error occured";
    return 0;
}
```

cerr by defaults will write to the console.

**Output:**

```
An error occured
```

4. **buffered standard error stream (clog)**: This is also an instance of iostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. The error message will be displayed on the screen too.*filter_none*

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

int main()
{
    clog << "An error occured";

    return 0;
}
```

Again, this stream/s default device is the console.

**Output:**

```
An error occured
```

Adapted from:
"Basic Input / Output in C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 11.2: C++ Input- getline()

## getline (string) in C++

The C++ **getline()** is a standard library function that is used to read a string or a line from an input stream. It is a part of the **<*string*> header**. The getline() function extracts characters from the input stream and appends it to the string object until the delimiting character is encountered. If no delimiter is specified, the default is the newline character at the end of the line. The input value does NOT contain the delimiter. While doing so the previously stored value in the string object str will be replaced by the input string if any.

The getline() function can be represented in two ways:

1. **Syntax:**

   ```
   // inputStream is like cin
   string someString;
   char delimiter;

   getline(inputStream, someString, delimiter);
   ```

   **Parameters:**

   - **inputStream:** It is an object of istream class and tells the function about the stream from where to read the input from.
   - **someString:** It is a string object, the input is stored in this object after being read from the stream.
   - **delimiter:** It is the delimitation character which tells the function to stop reading further input after reaching this character.

   **Return Value:** This function returns the same input stream as *is* which is accepted as parameter.

2. **Syntax:**

   ```
   getline (inputStream, someString);
   ```

   The second declaration is almost the same as that of the first one. The only difference is, the latter have an delimitation character which is by default new line(\n)character.

   **Parameters:**

   - **inputStream:** It is an object of istream class and tells the function about the stream from where to read the input from.
   - **someString:** It is a string object, the input is stored in this object after being read from the stream.

   **Return Value:** This function returns the same input stream as *is* which is accepted as parameter.

Here is an example of using getlin() function.

```
// C++ program to demonstrate getline() function

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

int main()
{
    string str;

    cout << "Please enter your name: \n";
    getline(cin, str);
```

```
    cout << "Hello, " << str << " welcome to CSP 31A!" << endl;

    return 0;
}
```

In this code, we simply perform a getline()  using cin as out input stream and a variable named str to hold the input obtained by getline(), then we pass this variable to the cout routing.

**Input:**

```
Pat Mac
```

**Output:**

```
Hello, Pat Mac welcome to CSP 31A!
```

**Example 2:** We can use getline() function to split a sentence on the basis of a character. Let’s look at an example to understand how it can be done.

```
// C++ program to understand the use of getline() function
#include <bits/stdc++.h>
using namespace std;

int main()
{
    string myString, tempString;

    // Read some input from the keyboard
    getline(cin, myString);

    // This turns our string into a stream
    stringstream myStream(myString);

    // Using the newly created stream we read if up to the first space character
    // The next read starts at the character
    while (getline(myStream, tempString, ' '))
    {
        cout << tempString << ' ';
    }
    cout << endl;
    return 0;
}
```

We read from the keyboard and put those characters into the string myString. Then we turn that string into a stream using the stringstream() function. Lastly we use a while loop to take our input, and read up until the first space character - since that is the delimiter we specify in the getline() function. that word in placed in tempString. We output that string. Then we go back to the top of the loop and grab the next set of characters, until the next space, assigned that to tempString and output that string. we continue doing this until we reach the end of the input in myStream. When we reach the end of the input the condition becomes false and we stop the while loop.

Notice when we output each string we had to add the space, when we use a delimiter in the getline() function, that character gets thrown away.

```
Pat Mac - welcome to this glorious place

if we don't include the space at the end of the cout statement the output looks like

PatMac-welcometothisgloriousplace
```

Adapted from:
"getline (string) in C++" by Harsh Agarwal and Faisal Al Mamun, Geeks for Geeks is licensed under CC BY-SA 4.0

# 11.2.1: C++ Input- More getline()

## getline() function and character array

In C++, stream classes support line-oriented functions, getline() and write() to perform input and output functions respectively. getline() function reads whole line of text that ends with new line or until the maximum limit is reached. getline() is the member function of istream class and has the syntax:

The function does the following operations:
1. Extracts character up to the delimiter.
2. Stores the characters in the buffer.
3. Maximum number of characters extracted is size – 1.

Note that the terminator(or delimiter) character can be any character (like ‘ ‘, ‘, ‘ or any special character, etc.). The terminator character is read but not saved into a buffer, instead it is replaced by the null character.

```
// C++ program to show the getline() with
// character array
#include <iostream>
using namespace std;

int main()
{
    char str[20];
    cout << "Enter Your Name::";

    // see the use of getline() with array str
    // will read until '\n' or 20 characters
    cin.getline(str, 20);

    cout << "\nYour Name is:: " << str;
    return 0;
}
```

Notice that the getline() function looks almost like a method of cin. This example does NOT use a delimiter...it is proper to use a delimiter if you desire.

Input :

```
Pat Mac
```

Output :

```
 Your Name is:: Pat Mac
```

This code exhibits the write/put with cout. The cout.write will write out at most 9 characters. The cout.put will output a single character.

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

```
int main()
{
        char myStr[] = "This is a string of characters that is over 20 characters lon
        char mych = 'X';

        cout.write(myStr, 9);
        cout << endl;
        cout.put(mych);

        return 0;
}
```

When you run the code, the output will be:

```
This is a
X
```

Adapted from:
"getline() function and character array" by AdityaRakhecha, Geeks for Geeks is licensed under CC BY-SA 4.0

# 11.3: C++ Output Manipulators

## Manipulators in C++

**Manipulators** are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

For example, if we want to print the hexadecimal value of 100 then we can print it as:

```
cout << setbase(16) << 100
```

the setbase() is a manipulator - it manipulates the way the data is displayed. In this case, it takes the ouptu, the number 100, and converts it to base 16 and dislays it as 64

**Types of Manipulators**

There are various types of manipulators:

1. **Manipulators without arguments**: The most important manipulators defined by the **IOStream library** are provided below.
   - **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
   - **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
   - **ends**: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.
2. **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

   Here is an example of how to use manipulators in your code

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
    istringstream str("        Programmer");
    string line;

    // Ignore all the whitespace in string
    // str before the first word.
    getline(str >> std::ws, line);

    // you can also write str>>ws
    // After printing the output it will automatically
    // write a new line in the output stream.
    cout << line << endl;

    // without flush, the output will be the same.
    // flush simply forces the buffer to be sent in a buffered environment
    cout << "only a test" << flush;

    // Use of ends Manipulator
    cout << "\na";

    // NULL character will be added in the Output
    cout << "b" << ends;
    cout << "c" << endl;

    return 0;
}
```

**Output:**

```
Programmer
only a test
abc
```

3. **Manipulators with Arguments:** Some of the manipulators are used with the argument like setw (20), setfill (‘*’), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program.

   For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill (’*’);

   - **Some important manipulators in <*iomanip*> are:**
     1. **setw (val):** It is used to set the field width in output operations.
     2. **setfill (c):** It is used to fill the character ‘c’ on output stream.
     3. **setprecision (val):** It sets val as the new value for the precision of floating-point values.
     4. **setbase(val):** It is used to set the numeric base value for numeric values.
     5. **setiosflags(flag):** It is used to set the format flags specified by parameter mask.
     6. **resetiosflags(m):** It is used to reset the format flags specified by parameter mask.
   - **Some important manipulators in <ios> are:**
     1. **showpos:** It forces to show a positive sign on positive numbers.
     2. **noshowpos:** It forces not to write a positive sign on positive numbers.
     3. **showbase:** It indicates the numeric base of numeric values.
     4. **uppercase:** It forces uppercase letters for numeric values.
     5. **nouppercase:** It forces lowercase letters for numeric values.
     6. **fixed:** It uses decimal notation for floating-point values.
     7. **scientific:** It uses scientific floating-point notation.
     8. **hex:** Read and write hexadecimal values for integers and it works same as the setbase(16).
     9. **dec:** Read and write decimal values for integers i.e. setbase(10).
     10. **oct:** Read and write octal values for integers i.e. setbase(10).
     11. **left:** It adjusts output to the left.
     12. **right:** It adjusts output to the right.

**Example:**

```
#include <iomanip>
#include <iostream>
```

```
using namespace std;

int main()
{
    double A = 100;
    double B = 2001.5251;
    double C = 201455.2646;

    // We can use setbase(16) here instead of hex

    // turns on hexidecimal output, left justify
    // numerical base, and forces lowercase
    cout << hex << left << showbase << nouppercase;

    // actual printed part
    cout << (long long)A << endl;

    // We can use dec here instead of setbase(10)

    // reset base back to base 10, right justified
    // and width to 15.
    // fills empty space with underline
    // show positive sign, precision to 2 places
    cout << setbase(10) << right << setw(15)
        << setfill('_') << showpos
        << fixed << setprecision(2);

    // actual printed part
    cout << B << endl;

    // scientific formatting, uppercase of the 'E'
    // precision to 9 places
    cout << scientific << uppercase
        << noshowpos << setprecision(9);

    // actual printed part
    cout << C << endl;
}
```

The first line of output is the value 100 - it is printed in hexadecimal, base 16, as specified in the code. the 0x signifies that this value is base 16.

The second line of output is base 10, right justified, sets width to 15, fills empty space with underline, show positive sign, and precision to 2 places

The third line of output and the scientific formatting - we can tell because it has a lot of decimal places and a 'E+' near the end, precision is set to 9 places.

```
0x64
_______+2001.53
2.014552646E+05
```

Adapted from:
"Manipulators in C++ with Examples" by PulkitDahiya, Geeks for Geeks is licensed under CC BY-SA 4.0

# CHAPTER OVERVIEW

## 12: Pointers

# 12.1: Address Operator

## Address Operator in C++

"Every variable is assigned a memory location whose address can be retrieved using the address operator &. The address of a memory location is called a pointer. Every variable in an executing program is allocated a section of memory large enough to hold a value of that variable's type." Thus, whether the variables are **global scope** and use the data area for storage or **local scope** and use the stack for storage; you can ask the question at what address in the memory does this variable exist. Given an integer variable named age:

```
int age = 47;
```

We can use the **address operator** [which is the **ampersand** or &] to determine where it exists (or its address) in the memory by:

```
&age
```

This expression is a **pointer** data type. The concept of an address and a pointer are one in the same. A pointer points to the location in memory because the value of a pointer is the address were the data item resides in the memory.

The address operator is commonly used in two ways:

To do parameter passing by reference

To establish the value of pointers

Both of these items are covered in the supplemental links to this module.

You can print out the value of the address with the following code:

```
cout << &age;
```

This will by default print the value in hexadecimal. Some people prefer an integer value and to print it as an integer you will need to cast the address into a long data type:

```
cout << long(&age);
```

One additional tidbit, an array's name is by definition a pointer to the arrays first element. Thus:

```
int iqs[] = {122, 105, 131, 97};
```

establishes "iqs" as a pointer to the array.

## Definitions

**Address Operator**

The ampersand or &.

**Pointer**

A variable that holds an address as its value.

## Footnotes

Tony Gaddis, Judy Walters and Godfrey Muganda, Starting Out with C++ Early Objects Sixth Edition(United States of America: Pearson – Addison Wesley, 2008) 597.

# 12.2: Pointer Data Type

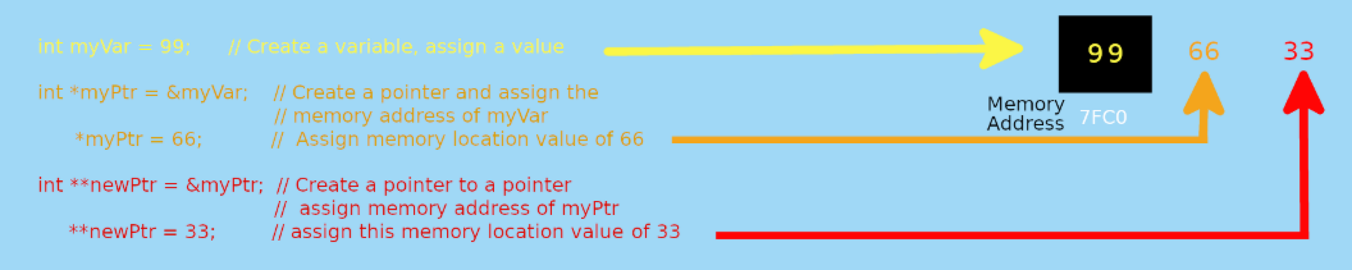
## Pointers in C/C++ with Examples

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It’s general declaration in C/C++ has the format:

**Syntax:**

```
datatype *var_name;

So, an integer point would be defined as:
int *ptr;   //ptr can point to an address which holds int data
```

**How to use a pointer?**



```
int myVar = 99;      // Create a variable, assign a value

int *myPtr = &myVar;    // Create a pointer and assign the
                        // memory address of myVar
    *myPtr = 66;        //  Assign memory location value of 66

int **newPtr = &myPtr;  // Create a pointer to a pointer
                        //  assign memory address of myPtr
    **newPtr = 33;      // assign this memory location value of 33
```

Figure 12.2.1: Copy and Paste Caption here. ("Pointers-in-C++.png" by Patrick McClanahan is in the Public Domain, CC0)

- Define a variable, myVar, and assign it the value of 99.
- Define a pointer variable - the variable newPtr is defined as type pointer with the asterisk - *newPtr.
- Assigning the address of myVar to the newPtr using the address operator (&) which returns the address of that variable - the variable myVar has a value of 99, and it is stored at memory address 7FC0 (this memory location varies each time the code is run). So we assign this address value to newPtr, and when we use the asterisk, *newPtr it is referring to an address - which now contains the value of 66.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand. The same concept applies to a pointer to a pointer - again, it performs address arithmetic to get the new value, 33, into the proper memory location.

The reason we associate data type to a pointer is **that it knows how many bytes the data is stored in**. When we increment a pointer, we increase the pointer by the size of data type to which it points. This is an important concept and we will come back to it later.

Let's take a look at some pointer code

```
// C++ program to illustrate Pointers in C++

#include <bits/stdc++.h>
using namespace std;

int main()
{
    int myVar = 20;

    //declare pointer variable
    int *myPtr;
    //note that data type of ptr and var must be same
    *myPtr = &myVar;

    // assign the address of a variable to a pointer
    cout << "Value at myPtr = " << myPtr << endl;
    cout << "Value at myVar = " << myVar << endl;
    cout << "Value at *myPtr = " << *myPtr << endl;

    return 0;
}
```

Our first cout outputs the value in the variable myPtr, which is 0x7ffcb9e9ea4c - the address assigned to the variable myVar.

The next cout simply outputs the value stored in the variable myVar.

The last cout, because it uses the asterisk, prints out the value at the address 0x7ffcb9e9ea4c. The asterisk says, "Look at the value that is found at the address contained in this variable.

Here is the output from this code (the address value will change if you run this code because it will probably occupy a different location)

```
Value at myPtr = 0x7ffcb9e9ea4c
Value at myVar = 20
Value at *myPtr = 20
```

**References and Pointers**

There are 3 ways to pass C++ arguments to a function:

- call-by-value
- call-by-reference with pointer argument
- call-by-reference with reference argument

```
// C++ program to illustrate call-by-methods in C++

#include <bits/stdc++.h>
using namespace std;

//Pass-by-Value
int square1(int n)
{
    //Address of n in square1() is not the same as n1 in main()
    cout << "address of n1 in square1(): " << &n << "\n";

    // clone modified inside the function
    n *= n;
    return n;
}
//Pass-by-Reference with Pointer Arguments
void square2(int *n)
{
    //Address of n in square2() is the same as n2 in main()
    cout << "address of n2 in square2(): " << n << "\n";

    // Explicit de-referencing to get the value pointed-to
    // Says: the value pointed to equals the value pointed to times the value pointed
    // *n = *n * *n
    *n *= *n;
}
//Pass-by-Reference with Reference Arguments
void square3(int &n)
{
    //Address of n in square3() is the same as n3 in main()
    cout << "address of n3 in square3(): " << &n << "\n";

    // Implicit de-referencing (without '*')
    n *= n;
}
int main()
{
    // Call-by-Value
    int n1=8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    // Notice how we call the function from within the cout statement
    cout << "Square of n1: " << square1(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";

    //Call-by-Reference with Pointer Arguments
    int n2=8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    //Call-by-Reference with Reference Arguments
    int n3=8;
    cout << "address of n3 in main(): " << &n3 << "\n";
    square3(n3);
    cout << "Square of n3: " << n3 << "\n";
    cout << "Change reflected in n3: " << n3 << "\n";

    return 0;
}
```

Call by value:

- We pass a copy of the value, 8
- **Notice** that the square1() function is called from WITHIN the cout statement
- The function squares that value AND returns that squared value

- **Remember**, when a function ends - or returns - it goes back to where it was called from. When it returns a value, then that value basically replaces the original functino call in the code.

Call-by-Reference with Pointer Arguments

- Calling the square2() function, we pass an address, &n2
- The function square2(), hen declares it as *n - which points to the address of n2
- **Notice** - there is NOT a return statement here...because we inserted a new value at the address that was passed from main()

Call-by-Reference with Reference Arguments

- We call square3() passing an argument, BUT, in square3() it is declared as &n
- When we use n *= n - it is understood that we are dealing with the address, because that is how it was declared in the function signature

Output:

```
Call-by-Value
address of n1 in main(): 0x7ffcdb2b4a44
address of n1 in square1(): 0x7ffcdb2b4a2c
Square of n1: 64
No change in n1: 8

Call-by-Reference with Pointer Arguments
address of n2 in main(): 0x7ffcdb2b4a48
address of n2 in square2(): 0x7ffcdb2b4a48
Square of n2: 64
Change reflected in n2: 64

Call-by-Reference with Reference Arguments
address of n3 in main(): 0x7ffcdb2b4a4c
address of n3 in square3(): 0x7ffcdb2b4a4c
Square of n3: 64
Change reflected in n3: 64
```

In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable. The changes are made into a clone made by the called function.

If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference. Pass-by-Reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

Adapted from:
"Pointers in C/C++ with Examples" by Abhirav Kariya, Geeks for Geeks is licensed under CC BY-SA 4.0

# 12.3: Indirection Operator

## Indirection Operator in C++

When we pass parameters to functions we usually pass by value; that is the calling function provides several values to the called function as needed. The called function takes these values which have **local scope** and stores them on the stack using them as needed for whatever processing the functions accomplishes. This is the preferred method when calling user defined specific task functions. The called function passes back a single value as the return item if needed. This has the advantage of a **closed communications model** with everything being neatly passed in as values and any needed item returned back as a parameter.

By necessity there are two exceptions to this closed communications model:

When we need more than one item of information returned by the function

When a copy of an argument cannot reasonably or correctly be made (example: file stream objects).

These exceptions could be handled by parameter passing by reference instead of passing a value. Although different syntax than parameter passing when using a **reference variable**; using a **pointer** variable and the **indirection operator** can accomplish the same effect. The indirection operator is the **asterisk** or the character that we also use for multiplication. The concept of indirection is also known as **dereferencing**, meaning that we are not interested in the pointer but want the item to which the address is referring or referencing.

```
Parameter passing with pointers
// prototype
void process_values(int qty_dimes, int qty_quarters, double *ptr_value_dimes, double

// variable definitions
int      dimes = 45;
int      quarters = 33;
double   value_dimes;
double   value_quarters;
double *ptr_value_dimes = &value_dimes;
double *ptr_value_quarters = &value_quarters;

// somewhere in the function main
process_values(dimes, quarters, ptr_value_dimes, ptr_value_quarters);

// definition of the function
void process_values(int qty_dimes, int qty_quarters, double *ptr_value_dimes, double
  {
  *ptr_value_dimes = dimes * 0.10;
  *ptr_value_quarters = quarters * 0.25;
  }
```

Note: The asterisk and must appear in both the prototype and the function definition when defining the pointer variables but it does not appear in the function call when the pointers are passed into the function.

The above example shows the basic mechanics of the indirection operator.

The use of pointers with indirection is often preferred for processing arrays. The **array index operator** is also known as the **array method of dereferencing**. The following couts are equivalent:

```
int ages[] = {47, 45, 18, 11, 9};
cout << ages[3];
cout << *(ages + 3);
```

The both say, "The name of an array is a pointer; take the pointer and calculate a new address that points to the 3$^{rd}$ offset by adding the correct number of bytes onto the pointer (integer data type is normally 4 bytes long – 3 offsets times 4 bytes is 12 bytes); then dereference that pointer (since this is an Rvalue context – fetch me the value that you are pointing at) and send it to the standard output device."

You should study the demonstration programs in conjunction with this module.

## Definitions

**Indirection Operator**

The asterisk used for dereferencing a pointer.

**Dereferencing**

The concept of using the item to which a pointer or address is pointing at.

# 12.4: Arrays, Pointers and Such

## Difference between pointer and array in C++?

Pointers are used for storing address of dynamically allocated arrays and for arrays which are passed as arguments to functions. In other contexts, arrays and pointer are two different things, see the following programs to justify this statement.

The behavior of the sizeof operator is shown below

```
/ 1st program to show that array and pointers are different
#include
using namespace std;

int main() {
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;

   // sizof(int) * (number of element in arr[]) is printed
   cout << "Size of arr[] "<< sizeof(arr)<<"\n";

   // sizeof a pointer is printed which is same for all type
   // of pointers (char *, void *, etc)
   cout << "Size of ptr "<< sizeof(ptr);

   return 0;

}
```

The array is 6 members * 4 bytes each, so we get 24 bytes for the variable arr.  The pointer ptr, and all pointers, are 8 bytes, because they hold addresses, which are 8 bytes, or 64 bits.

**Output:**

```
Size of arr[] 24
Size of ptr 8
```

Assigning any address to an array variable is not allowed.

```
// Second program to show that array and pointers are different
#include <iostream>
using namespace std;

int main()
{
    int arr[] = {10, 20}, x = 10;
    int *ptr = &x; // This is fine because ptr is to hold addresses
    arr = &x; // Compiler Error this is an integer array, not meant to hold addresses
    return 0;
}
```

Output:

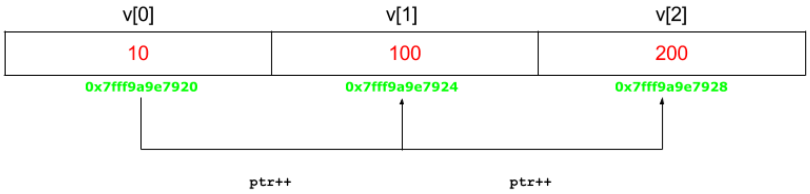
```
Compiler Error: incompatible types when assigning to
                type 'int[2]' from type 'int *'
```

Although array and pointer are different things, following properties of array make them look similar.

- Array name gives address of first element of array.
  Consider the following program for example.

```
// 1st program to show that array and pointers are different
#include <iostream>
using namespace std;

int main() {
   int arr[] = {10, 100, 200, 40, 50, 60};

   // Assigns address of array to ptr
   int *ptr = arr;
   cout << "Value of first element is " << *ptr;

   return 0;
}
```

In the above code the value of arr is actually an address. From that address all of the members of the array are calculated - an int is 4 bytes, so the value of 10, as shown in the image (the image uses an array named v, instead of arr), is at specific address. The next value 100 is at that same address PLUS 4 - because integers are 4 bytes long. So, by saying arr[1] we are saying - the value we want, 100, is at the address of arr + 4 more bytes - because the value of 10 takes up the 4 bytes from the address 0x7fff9a9e7920 to 0x7fff9a9e7923



So really we are simply performing address arithmatic - it all depends on the type of the array, which determines how much memory each value takes up.

**Output:**

```
Value of first element is 10
```

- Array members are accessed using pointer arithmetic.
  Compiler uses pointer arithmetic to access array element. For example, an expression like “arr[i]” is treated as *(arr + i) by the compiler. That is why the expressions like *(arr + i) work for array arr, and expressions like ptr[i] also work for pointer ptr.

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

int main() {
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;
   cout << "arr[2] = "<< arr[2] <<"\n";
   cout << "*(arr + 2) = "<< *(arr + 2)<<"\n";
   cout << "ptr[2] = "<< ptr[2]<< "\n";
   cout << "*(ptr + 2) = "<< *(ptr + 2)<< "\n";
   return 0;
}
```

- **Output:**

```
arr[2] = 30
*(arr + 2) = 30
ptr[2] = 30
*(ptr + 2) = 30
```

- Array parameters are always passed as pointers, even when we use square brackets.*filter_none*

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

int fun(int ptr[])
{
   int x = 10;

   // size of a pointer is printed
   cout << "sizeof(ptr) = " << sizeof(ptr) << endl;

   // This allowed because ptr is a pointer, not array
   ptr = &x;

   cout << "*ptr = " << *ptr << endl;

   return 0;
}

// Driver code
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   fun(arr);
   return 0;
}
```

**Output:**

```
sizeof(ptr) = 8
*ptr = 10
```

## Array Name as Pointers

An array name contains the address of first element of the array which acts like constant pointer. It means, the address stored in array name can't be changed.
For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

```
// C++ program to illustrate Array Name as Pointers in C++
#include <iostream>
using namespace std;
int main()
{
    //Declare an array
    int newArr[3] = { 5, 10, 20 };

    //declare pointer variable
    int *arrPtr;

    //Assign the address of newArr[0] to ptr
    // We could use arrPtr = &newArr[0];(both are same)
    arrPtr = newArr ;
    cout << "Elements of the array are: ";
    cout << arrPtr[0] << " " << arrPtr[1] << " " << arrPtr[2];

    return 0;
}
```

In looking at this code you may notice that when when we use the square brackets - [ ] - it is actually using the address. This is because the value stored in the newArr variable IS an address in memory.

```
Output:
Elements of the array are: 5 10 20
```

Adapted from:
"Pointers in C/C++ with Examples" by Abhirav Kariya, Geeks for Geeks is licensed under CC BY-SA 4.0
"Pointers in C and C++ | Set 1 (Introduction, Arithmetic and Array)" by Abhirav Kariya, Geeks for Geeks is licensed under CC BY-SA 4.0
"Difference between pointer and array in C?" by Abhay Rathi, Geeks for Geeks is licensed under CC BY-SA 4.0

# 12.4.1: Pointer to an Array - Array Pointer

## Pointer to Array

Consider the following program:

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

int main()
{
   int myArr[5] = { 1, 2, 3, 4, 5 };
   int *newPtr = myArr;

   cout << "Pointer is " << newPtr << endl;
   return 0;
}
```

In this program, we have a pointer newPtr that points to the 0th element of the array. Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array. This pointer is useful when talking about multidimensional arrays.

**Syntax:**

```
data_type (*var_name)[size_of_array];
```

**Example:**

```
int (*ptr)[10];
```

Here ptr is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is ‘pointer to an array of 10 integers’.

**Note :** The pointer that points to the 0th element of array and the pointer that points to the whole array are totally different. The following program shows this:

```
// C++ program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include <iostream>
using namespace std;

int main()
{
    // Pointer to an integer
    int *p;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];
    // Points to 0th element of the arr.
    p = arr;

    // Points to the whole array arr.
    ptr = &arr;

    cout << "p = " << p << ", ptr = " << ptr << endl;
    p++;
    ptr++;
    cout << "p = " << p << ", ptr = " << ptr << endl;

    return 0;
}
```

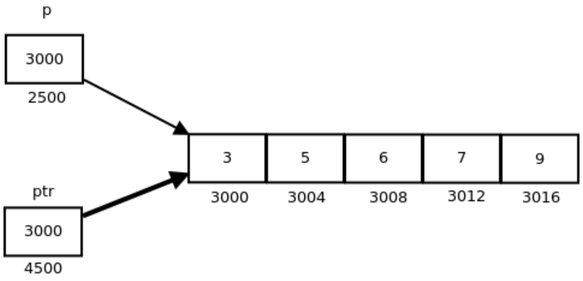
The following output shows that when the point p get incremented the address only increases by 4 - because an integer is 4 bytes long, and we therefore need to move to the beginning of the next member of the array. The variable ptr points to the entire array, which is 5 integers of 4 bytes each - so the array is 20 bytes long. When we increment the variable ptr it increases by 20 - remember looking at the output that we are dealing with hexadecimal values...

```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```

**p**: is pointer to $0^{th}$ element of the array arr, while **ptr** is a pointer that points to the whole array arr.

- The base type of p is int while base type of ptr is ‘an array of 5 integers’.
- We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer ptr will be shifted forward by 20 bytes.

The following figure shows the pointer p and ptr. Darker arrow denotes pointer to an array.



On dereferencing a pointer expression we get a value pointed to by that pointer expression. Pointer to an array points to an array, so on dereferencing it, we should get the array, and the name of array denotes the base address. So whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

```
// C++ program to illustrate sizes of
// pointer of array
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int arr[] = { 3, 5, 6, 7, 9 };
    int *p = arr;
    int (*ptr)[5] = &arr;

    cout << "p = "<< p <<", ptr = " << ptr << endl;
    cout << "*p = "<< *p <<", *ptr = " << *ptr << endl;

    cout << "sizeof(p) = " << sizeof(p) <<  ", sizeof(*p) = " << sizeof(*p) << endl;
    cout << "sizeof(ptr) = " << sizeof(ptr) << ", sizeof(*ptr) = " << sizeof(*ptr) <<

    return 0;
}
```

This is interesting output. The 2 pointers have the same address. The sizeof shows a couple of things: 1) the sizeof for both p and ptr shows 8 bytes - the amount of memory necessary to store an address; 2) the sizeof(*p) is 4, because this pointer points to a single integer value; 3) the sizeof(*ptr) is 20 because it points to the entire array of 5 integer values, or 20 bytes.

```
p = 0x7ffde1ee5010, ptr = 0x7ffde1ee5010
*p = 3, *ptr = 0x7ffde1ee5010
sizeof(p) = 8, sizeof(*p) = 4
sizeof(ptr) = 8, sizeof(*ptr) = 20
```

Adapted from:
"Pointer to an Array | Array Pointer" by Anuj Pratap Chauhan, Geeks for Geeks

# 12.5: Pointers to Pointers

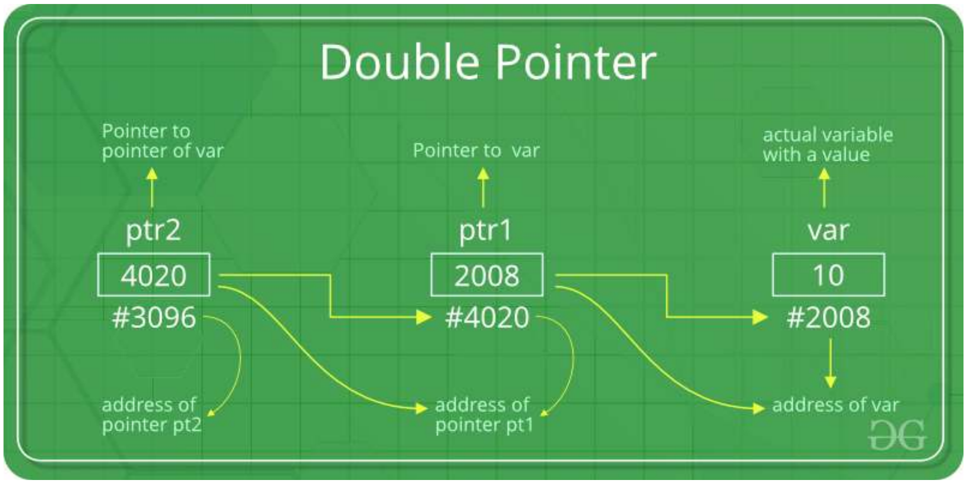
## Double Pointer (Pointer to Pointer) in C

We already know that a pointer points to a location in memory and thus used to store the address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as double pointers.

**How to declare a pointer to pointer in C++?**
Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.
**Syntax**:

```
int **ptr2;   // declaring double pointers
int *ptr1;    // pointer
int var;      // variable
```

Below diagram explains the concept of Double Pointers:



The above diagram shows the memory representation of a pointer to pointer. The first pointer ptr1 stores the address of the variable and the second pointer ptr2 stores the address of the first pointer.

Remember how the pointer works - we saw the next image in a previous section...just for a reminder.

Let us understand this more clearly with the help of the program below:

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

// C program to demonstrate pointer to pointer
int main()
{
    int newVar = 789;
    // pointer for newVar
    int *newPtr;
    // double pointer for newPtr
    int **dblPtr;
    // storing address of newVar in newPtr
    newPtr = &newVar;

    // Storing address of newPtr in dblPtr
    dblPtr = &newPtr;

    // Displaying address of all the varaibles
    cout << "Address of newVar: " << &newVar << endl;
    cout << "Address contained in newPtr: " << newPtr << endl;
    cout << "Address contained in dblPtr: " << *dblPtr << endl;

    // Displaying value of newVar using
    // both single and double pointers
    cout << "Value of newVar = " << newVar << endl;
    cout << "Value of newVar using single pointer, *newPtr = " << *newPtr << endl;
    cout << "Value of newVar using double pointer, **dblPtr = " << **dblPtr << endl;

return 0;
}
```

We have an integer variable, a pointer variable and a pointer to a pointer variable. Really, all we are doing is passing around the address of the integer variable, As you can see in the output below, the address is the same, and accessing the value is gives us that same value..

Here is the output:

```
Address of newVar: 0x7ffe33b7c834
Address contained in newPtr: 0x7ffe33b7c834
Address contained in dblPtr: 0x7ffe33b7c834

Value of newVar = 789
Value of newVar using single pointer, *newPtr = 789
Value of newVar using double pointer, **dblPtr = 789
```

Pointers to pointers can be useful...as long as you realize its just addresses, you can keep it straight in your mind

Adapted from:
"Double Pointer (Pointer to Pointer) in C" by Harsh Agarwal, Geeks for Geeks

# 12.6: Pointer Arithmetic

## Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers which are:

- incremented ( ++ )
- decremented ( -- )
- an integer may be added to a pointer ( + or += )
- an integer may be subtracted from a pointer ( – or -= )
- difference between two pointers (p1-p2)

(Note: Pointer arithmetic is meaningless unless performed on an array.)

```
// C++ program to illustrate Pointer Arithmetic in C++
#include <bits/stdc++.h>
using namespace std;
int main()
{
    //Declare an integer array
    int intArr[3] = {10, 100, 200};

    //declare an integer pointer variable
    int *arrPtr;

    //Assign the address of intArr[0] to arrPtr
    arrPtr = intArr;

    for (int count = 0; count < 3; count++)
    {
            cout << "Value at arrPtr = " << arrPtr << "\n";
            cout << "Value at *arrPtr = " << *arrPtr << "\n";

            // Increment pointer ptr by 1
            arrPtr++;
    }
    return 0;
}
```

(Note: Pointer arithmetic is meaningless unless performed on an array.)

In the code above we increment the pointer by 1. Look at the output of the addresses below - each address is 4 greater than the previous address!! How is that, we only added one, using the ++ operator, to the pointer variable?

```
Output:
Value at arrPtr = 0x7fff9a9e7920
Value at *arrPtr = 10
Value at arrPtr = 0x7fff9a9e7924
Value at *arrPtr = 100
Value at arrPtr = 0x7fff9a9e7928
Value at *arrPtr = 200
```

C++ knows that the variable - both the array and the pointer - are an integer type, so adding one to that address means we move forward the width of one integer, which is usually 4 bytes.

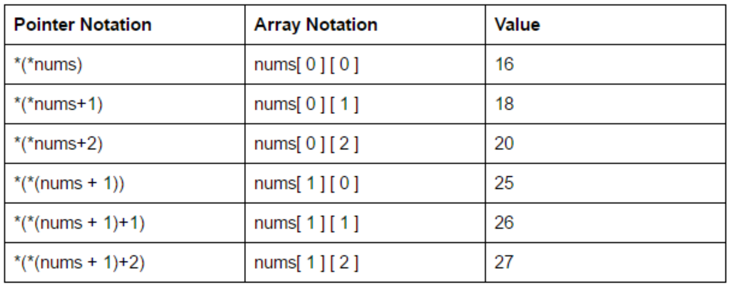
The subscript - that value in the [ ] - says to go a certain number of bytes past that address. So, the first element [0], says to go 0 bytes beyond the address. When we use [1] - C++ says, "okay, this is an integer (in our example), so I will go 1 integer width past the address in intArr. An integer, on this system, is 4 bytes long, so intArr[1] look 4 bytes past the address stored in intArr.

## Advanced Pointer Notation

Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

```
int nums[2][3]  =  { { 16, 18, 20 }, { 25, 26, 27 } };
```

**In general, nums[ i ][ j ] is equivalent to *(*(nums+i)+j)**



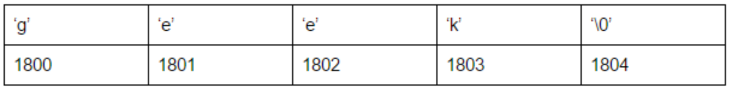
| Pointer Notation | Array Notation | Value |
|---|---|---|
| *(*nums) | nums[ 0 ] [ 0 ] | 16 |
| *(*nums+1) | nums[ 0 ] [ 1 ] | 18 |
| *(*nums+2) | nums[ 0 ] [ 2 ] | 20 |
| *(*(nums + 1)) | nums[ 1 ] [ 0 ] | 25 |
| *(*(nums + 1)+1) | nums[ 1 ] [ 1 ] | 26 |
| *(*(nums + 1)+2) | nums[ 1 ] [ 2 ] | 27 |

## Pointers and String literals

String literals are arrays containing null-terminated character sequences. String literals are arrays of type character plus terminating null-character, with each of the elements being of type const char (as characters of string can’t be modified).

```
const char * ptr = "geek";
```

This declares an array with the literal representation for “geek”, and then a pointer to its first element is assigned to ptr. If we imagine that “geek” is stored at the memory locations that start at address 1800, we can represent the previous declaration as:



| | | | | |
|---|---|---|---|---|
| ‘g’ | ‘e’ | ‘e’ | ‘k’ | ‘\0’ |
| 1800 | 1801 | 1802 | 1803 | 1804 |

As pointers and arrays behave in the same way in expressions, ptr can be used to access the characters of string literal. For example:

```
char x = *(ptr+3);
char y = ptr[3];
```

Here, both x and y contain k stored at 1803 (1800+3).

Adapted from:
"Pointers in C/C++ with Examples" by Abhirav Kariya, Geeks for Geeks is licensed under CC BY-SA 4.0
"Pointers in C and C++ | Set 1 (Introduction, Arithmetic and Array)" by Abhirav Kariya, Geeks for Geeks is licensed under CC BY-SA 4.0

# 12.7: Function Pointer in C

## Function Pointer in C

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <iostream>
using namespace std;
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    cout << "Value of a is " << a << endl;
}

int main()
{
    // Declare a void function pointer
    // with a single int argument
    void (*fun_ptr)(int);
    // Assign the function pointer the address of the function
    fun_ptr = &fun;

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

This is the same as what we have already learned about pointers...just that we are now pointing at functions. The output would be what we expect:

```
Value of a is 10
```

Why do we need an extra bracket around function pointers like fun_ptr in above example?
If we remove bracket, then the expression “void (*fun_ptr)(int)” becomes “void *fun_ptr(int)” which is declaration of a function that returns void pointer.

### Following are some interesting facts about function pointers.

**1)** Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

**2)** Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

**3)** A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing *, the program still works.

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

```
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    cout << "Value of a is " << a << endl;
}

int main()
{
    // Declare a void function pointer
    // with a single int argument
    void (*fun_ptr)(int);
    // Assign the function pointer the address of the function
    fun_ptr = fun;

    // Invoking fun() using fun_ptr
    fun_ptr(10);

    return 0;
}
```

Program works, output is the same:

```
Value of a is 10
```

**4)** Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

**5)** Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```
#include <iostream>
using namespace std;
void add(int a, int b)
{
    cout << "Addition is "  << a+b << endl;
}
void subtract(int a, int b)
{
    cout << "Subtraction is "  << a-b << endl;
}
void multiply(int a, int b)
{
    cout << "Multiplication is "  << a*b << endl;
}

int main()
```

```
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    for (int count = 0; count < 3; count++)
    {
        ch = unsigned (count);
       (*fun_ptr_arr[ch])(a, b);
    }
    return 0;
}
```

This is a bit more complex, the key is that the function pointer is an array - pointing at the 3 functions. In the for loop the value of ch is 0, 1 and 2 - thereby calling the addition function, fun_ptr_arr[0], the subtraction function, fun_ptr_arr[1], and the multiplication function, fun_ptr_arr[2].

Also, the assignment of ch= unsigned (count) - this is taking the integer count and forcing it to be an unsigned integer, then assigning it to ch. The variables a, and b, are also unsigned integers, but when these values are copied (call by value - remember) to the functions notice that the functions believe the arguments are integers.

```
Addition is 25
Subtraction is 5
Multiplication is 150
```

Adapted from:
"Function Pointer in C" by Abhay Rathi, Geeks for Geeks

CHAPTER OVERVIEW

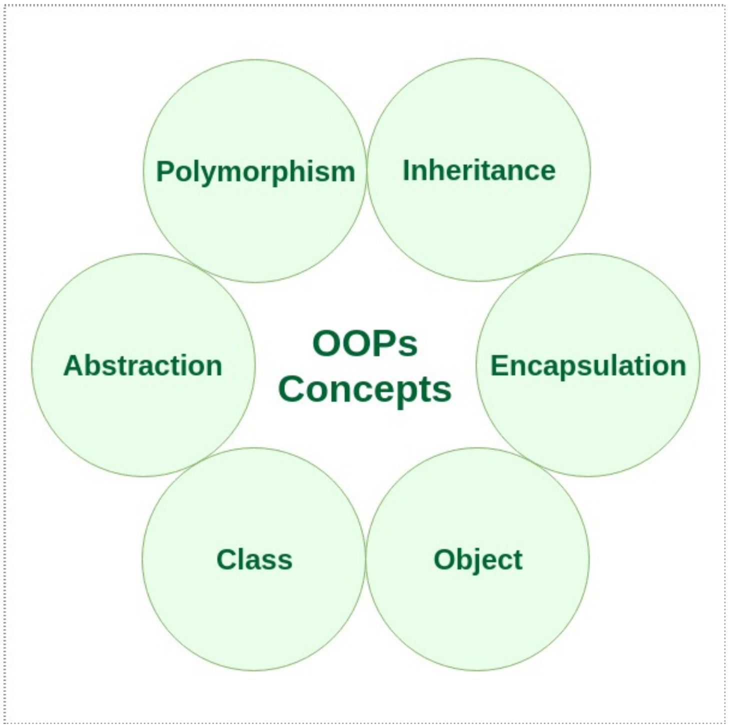
# 13: Object Oriented C++

# 13.1: Object Oriented Programming

## Object Oriented Programming in C++

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

**Characteristics of an Object Oriented Programming language**



We will discuss each of these concepts in greater detail in the following modules. For now, here are some very brief descriptions of each of the concepts shown in the above image.

**Class**: The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

**Object:** An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other’s data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

**Encapsulation**: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

**Abstraction**: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

**Polymorphism:** The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

**Inheritance**: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class**:The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Adapted from: "Object Oriented Programming in C++" by Vankayala Karunakar, Geeks for Geeks

# 13.2: Classes and Objects

## C++ Classes and Objects

**Class:** A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

### Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
include <iostream>
using namespace std;

// "class" is a resevered word in C++
// Junk is a user-defined name for this class definition
class Junk
{
    // Access specifier - other specifiers are: private, and protected
    public:
         // Data Members - these are public variables
         string junkName;
         int id;

        // Member Functions() - this function is ALSO public
        void printName()
        {
           cout << "My name is: " << junkName;
        }
};
```

We will talk about the Public keyword in a moment.

**Declaring Objects:** When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects. **THIS IS AN IMPORTANT CONCEPT**. The class definition simply defines what a class looks like. It is the object declaration that actually allocates memory space by assigning that space to the objectName - which is simply a variable.

**Syntax:**

```
ClassName objectName;
```

Using the class Junk that was defined in the example above, to create an instance of that class we use:

```
int main() {

    Junk myObj;
    myObj.junkName = "Antique";
    myObj.id=15;
}
```

In this code the declaration Junk myObj; declares the variable name myObj to point to an instance of the Junk class. The system allocates enough memory to contain an entire copy of the class Junk, as defined in the previous example. This gets a bit more complex since we have member functions, not simply regular typed varialbes.

Adapted from:
"Access Modifiers in C++" by Abhirav Kariya and Harsh Agarwal , Geeks for Geeks
"C++ Classes and Objects" by BabisSarantoglou, Geeks for Geeks is licensed under CC BY-SA 4.0

# 13.2.1: Classes and Objects - Data Members

## Accessing Data Members

There are 3 types of access modifiers available in C++:A Class is a user defined data-type which has data members and member functions.

1. **Public**
2. **Private**
3. **Protected**

**Note**: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Let us now look at each one these access modifiers in details:

1. **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
   **Example:**

```
// C++ program to demonstrate public
// access modifier
#include<iostream>
using namespace std;

// class definition
class Circle
{
    public:
        // a datamember that belongs to the class Circle
        double radius;

        // A method that belongs to this class
        double compute_area()
        {
            return 3.14*radius*radius;
        }
};


// main function
int main()
{
    // We declare a new instance of the class Circle
    Circle newObj;

    // accessing public data member outside class
    newObj.radius = 5.5;

    cout << "Radius is: " << newObj.radius << "\n";
    cout << "Area is: " << newObj.compute_area();
    return 0;
}
```

Several things in this code. We declare a class named Circle. Classes have data members, and functions (we will talk more about classes as we go along).

A data member is simply a variable that belongs to this class. The method is similar to a function, bu tit belongs to this class that it is defined in - in this case the Circle class.

Because the data member and the method are defined as public, we can access them both from the main() function. Notice that to access them we must prepend instance name followed by a period in front of the name: newObj.radius and newObj.compute_area().

```
Radius is: 5.5
Area is: 94.985
```

2. **Private**: The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.
   **Example:**

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        // member function can access private
        // data member radius
        double compute_area()
        {
            return 3.14*radius*radius;
        }
};

// main function
int main()
{
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    // THIS IS AN ERROR
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

The output of above program will be a compile error. In main() we attempt to access the obj.radius dta member outside of the class - but since this data member is now declared **private** we are not allowed to access it from outside of the class it is defined in, giving us the error:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
         double radius;
                ^
31:9: error: within this context
     obj.radius = 1.5;
         ^
```

However, we can access the private data members of a class indirectly bu using the public member functions of the class.So we alter the code to pass the argument to the public method computer_area() and the method can set the value of the private member radius, as well as calculate the area, and output the results.

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        void compute_area(double r)
        {   // member function CAN access private
            // data member radius
            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
        }
};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // Pass the argument to the method who CAN
    // access the private member from within the class
    obj.compute_area(1.5);

    return 0;
}
```

```
Radius is: 1.5
Area is: 7.065
```

3. **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.
   **Example:**

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

// base class
class myParent
{
    // protected data members
    protected:
    int id_protected;
};

// sub class or derived class
class thisChild : public myParent
{
    public:
       void setId(int id)
       {
           // Child class is able to access the inherited
           // protected data members of base class
           id_protected = id;
       }

```

```
        void displayId()
        {
            cout << "id_protected is: " << id_protected << endl;
        }
};

// main function
int main()
{
    thisChild newObj;

    // member function of the derived class can
    // access the protected data members of the base class

    newObj.setId(81);
    newObj.displayId();
    return 0;
}
```

**Output**:

```
id_protected is: 81
```

Adapted from:
"Access Modifiers in C++" by Abhirav Kariya and Harsh Agarwal , Geeks for Geeks
"C++ Classes and Objects" by BabisSarantoglou, Geeks for Geeks is licensed under CC BY-SA 4.0

# 13.2.2: Classes and Objects - Member Functions

## Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
// C++ program to demonstrate function
// declaration outside class

#include <bits/stdc++.h>
using namespace std;
class Courses
{
    public:
    string coursename;
    int id;

    // printname is not defined inside class definition
    void printname();

    // printid is defined inside class definition
    void printid()
    {
        cout << "Course id is: " << id;
    }
};

// Definition of printname using scope resolution operator ::
void Courses::printname()
{
    cout << "Course name is: " << coursename;
}

int main() {

    Courses thisObj;
    thisObj.coursename = "CSP 31A";
    thisObj.id=976;

    // call printname()
    // This is defined outside of the class Courses
    thisObj.printname();
    cout << endl;
    // call printid()
    // This is defined inside of the class Courses
    thisObj.printid();
    return 0;
}
```

In this code we have 2 separate ways to provide output. One is the method printid(), which is a method of the Courses class, it is defined inside of the class definition because it is between the opening an d closing brackets of the block of code which is the class definition.

We also have the method printname(), which is defined outside of the class definition. It is still a method of the class Courses, because as we can see it is specifically defined as void Courses::printname() using the scope resolution operator ::. Also, notice that we do declare printname() inside the class, but provide no body to the code.

Both of these methods belong to  the class Courses, and you have to call them with the object dot manner in your code.

```
// printname() method prints this statement
Geekname is: CSP 31A

// printid() method prints this statement
Geek id is: 976
```

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Note: Declaring a friend function is a way to give private access to a non-member function.

Adapted from:
"C++ Classes and Objects" by BabisSarantoglou, Geeks for Geeks is licensed under CC BY-SA 4.0

# 13.2.3: Classes and Objects - Constructor and Destructor

## Constructors

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. Constructors are usually used to setup the object that is being created. If you do NOT actually code and call a constructor then C++ will simply create an object.

There are 3 types of constructors:

- Default constructors
- Parametized constructors
- Copy constructors

```
// C++ program to demonstrate constructors


#include <bits/stdc++.h>
using namespace std;
class Courses
{
    public:
    int id;

    //Default Constructor
    Courses()
    {
        cout << "Default Constructor called" << endl;
        id=-1;
    }

    //Parametized Constructor
    Courses(int x)
    {
        cout << "Parametized Constructor called" << endl;
        id=x;
    }
};
int main() {

    // obj1 will call Default Constructor
    Courses courseObj1;
    cout << "Courses id is: " << courseObj1.id << endl;

    // obj1 will call Parametized Constructor
    Courses courseObj2(21);
    cout << "Courses id is: " << courseObj2.id << endl;
    return 0;
}
```

In this code we show both the default constructor, which has no arguments, and the parameterized constructor which takes arguments. Notice that the constructor method is simply the same as the name of the class we are constructing. Also, it is valid to have a default constructor with arguments.

The constructor gets called when the instance of a class is created. In our code above, we have the statement: Courses courseObj1; - at this point in our code the constructor method is called and memory is allocated for our object. This is shown in our output where the statement from within the constructor is output PRIOR to the statement in the code.

```
Default Constructor called
Geek id is: -1
Parametrized Constructor called
Geek id is: 21
```

A **Copy Constructor** is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
    ClassName (const ClassName &old_obj);
```

Here is an example of the copy constructor.

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

class Point
{
  private:
    int x, y;

  public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    // Copy constructor takes the input value
    Point(const Point &inputPt)
    {
        x = inputPt.x;
        y = inputPt.y;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    Point myP1(10, 15); // Normal constructor is called here
    Point myP2 = myP1;  // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "myP1.x = " << myP1.getX() << ", myP1.y = " << myP1.getY() << endl;
    cout << "myP2.x = " << myP2.getX() << ", myP2.y = " << myP2.getY() << endl;

    return 0;
}
```

The place we need to pay attention in the above code is where we declare myPt2 - this is where the copy constructor gets called. Our copy constructor simply copies the object variable values from the myPt1 object to the variables in the new myPt2 object. You can see from the output that both of these objects have the same variable values because of this copy. It is allowed to write copy constructors that do whatever you need them to do. The copy constructor is not restricted to simply

```
myP1.x = 10, myP1.y = 15
myP2.x = 10, myP2.y = 15
```

### Destructors

Destructor is another special member function that is called by the compiler when the scope of the object ends.

```
// C++ program to explain destructors


#include <iostream>
using namespace std;
class Courses
{
    public:
    int id;

    //Definition for Destructor
    ~Courses()
    {
        cout << "Destructor called for id: " << id <<endl;
    }
};
```

```
int main()
{
    Courses newObj1;
    newObj1.id=7;
    int i = 0;
    while ( i < 5 )
    {
        Courses newObj2;
        newObj2.id=i;
        i++;
    } // Scope for obj2 ends here


    return 0;
} // Scope for obj1 ends here
```

Just as we have a constructor, we have destructors that destroy the objects and release the memory back to the system.

One note in the code above is to look at newObj2. It is created within the block of code that is associated with the while loop. As we said when we discussed variable scope - a variable is only available within its block of code, or sub-blocks. So, newObj2 is created within this block, BUT, when we hit the bottom this loop newobj2 goes "out of scope" and the destructor method is called. Each time through the loop newObj2 is created again, and assigned an increasing id (we are incrementing i++). Then finally when the program ends newObj1 is destroyed as well, as you can see in the output below.

```
Destructor called for id: 0
Destructor called for id: 1
Destructor called for id: 2
Destructor called for id: 3
Destructor called for id: 4
Destructor called for id: 7
```

Adapted from:
"C++ Classes and Objects" by BabisSarantoglou, Geeks for Geeks is licensed under CC BY-SA 4.0

CHAPTER OVERVIEW

# 14: Overloading in C++

**Topic hierarchy**

# 14.1: Operator Overload

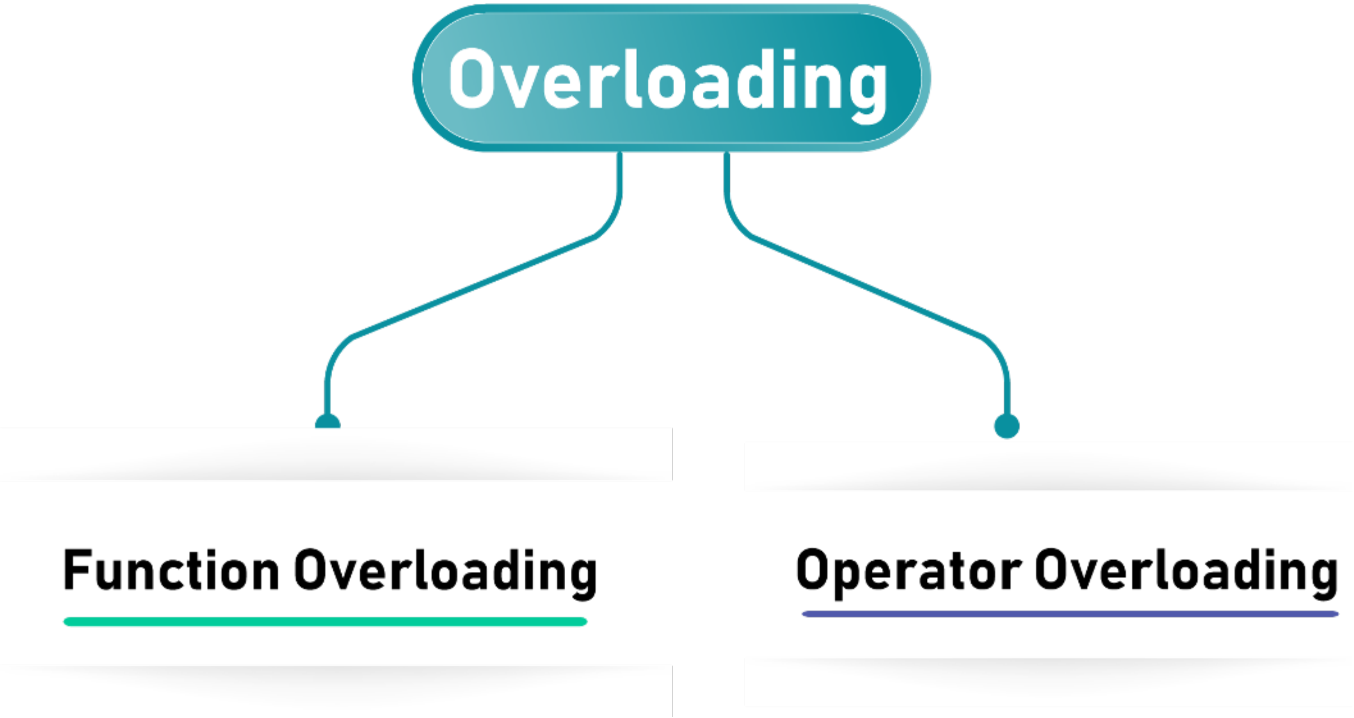
## What is overloading in C++?

Creating two or more objects that have the same name but are different in how they operate is known as **overloading**.

In C++, we can overload:

- Operators
- Methods - often referred to as function overloading

### Types of overloading



## Operator Overloading in C++

We will discuss operator overloading first, then in the next section we will discuss function overloading.

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using + as in the following example code snippet

```
String firstName = "Pat";
String lastName = "McClanahan";
String fullName;

// The + operator understands it is working on Strings
// so it concatenates them instead of attempting to add them
fullName = firstName + lastName
```

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

## What is the difference between operator functions and normal functions?

Operators methods are same as normal methods. The only differences are, name of an operator function is always **operator** keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

```
#include <iostream>
using namespace std;
class NewOp
{
    private:
       int myVal;
    public:
    void getvalue()
    {
        // The cin puts the entered value in the objects myVal variable.
        cin>>myVal;
    }
    NewOp operator + (NewOp inObj)
    {
        NewOp opObj;

        // opObj is defined in this function
        // inObj is the argument passed in - it is the obj on the right of the operat
        // myVal is the class variable associated with the object to the left of the
        opObj.myVal = myVal + inObj.myVal;
        return opObj;
    }
    NewOp operator - (NewOp inObj)
    {
        NewOp opObj;

        // opObj is defined in this function
        // inObj is the argument passed in - it is the obj on the right of the operat
        // myVal is the class variable associated with the object to the left of the
        opObj.myVal = myVal - inObj.myVal;
        return opObj;
    }

    int display()
    {
        return myVal;
    }
};
int main()
{

    NewOp obj1, obj2, sum, sub;
    cout << "enter an integer value for obj1: ";
    obj1.getvalue();
    cout << "Enter an integer value for obj2: ";
    obj2.getvalue();

    sum = obj1 + obj2;
    sub = obj1 - obj2;

    cout<<"Addition result is = "<<sum.display()<<endl;
    cout<<"Subtraction result is = "<<sub.display()<<endl;

    return 0;

}
```

In this code we overload the + and the - operator.  One thing to notice: in the operator overload functions there is a single argument, this argument is the object TO THE RIGHT of the operator...so in this example the function argument is the obj2 object. The other value is picked up from that object's myVal variable - that is the object to the right of the  operator, in out example this is obj1.

In case you are interested, the output from the sample code is:

```
enter an integer value for obj1: 10
Enter an integer value for obj2: 99
Addition result is = 109
Subtraction result is = -89
```

### Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

```
   . (dot)
   ::
   ?:
   sizeof
```

**Why can’t . (dot), ::, ?: and sizeof be overloaded?**
See this for answers from Stroustrup (the creator of C++) himself.

**Important points about operator overloading**
**1)** For operator overloading to work, at least one of the operands must be a user defined class object.

**2) Assignment Operator:** Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor). See this for more details.

**3) Conversion Operator:** We can also write conversion operators that can be used to convert one type to another type.

```
#include <iostream>
using namespace std;
class Fraction
{
    private:
        int num, den;
    public:
        Fraction(int n, int d)
        {
            // constructor sets a numerator and a denominiator for the fraction
            num = n; den = d;
        }


        // conversion operator: return float value of fraction
        operator float() const
        {
            return float(num) / float(den);
        }
};


int main() {
    Fraction frac(2, 5);
    float val = frac;
    cout << val;
    return 0;
}
```

When the code executes the statement `float val = frac`; we create an instance of the conversion operator, which calls the `float()` method. This method takes the two integer values, `num` and `den`, which were created and assigned values what we create an instance of the Fraction class: `Fraction frac(2,5);` and using the system float conversion to convert them to floating point values and performs the division. This overloading does a different conversion than the system conversion method which simply changes an integer value into a floating value.

The output is the result of a simple division:

```
0.4
```

Adapted from:
"Operator Overloading in C++" by Shun Xian Cai, Geeks for Geeks is licensed under CC BY-SA 4.0

# 14.2: Overload a Function

## Function Overloading in C++

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading “Function” name should be the same and the arguments should be different.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

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

// a char, an unsigned char, and short also use this function
void print(int inInt) {
  cout << " Here is int " << inInt << endl;
}

// a float argument is promoted to double and uses this function
void print(double  inFloat) {
  cout << " Here is float " << inFloat << endl;
}

// this function is a character pointer - and is used for string arguments
void print(char const *inChar) {
  cout << " Here is char* " << inChar << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

**Output:**

```
Here is int 10
Here is float 10.1
Here is char* ten
```

## How  Function Overloading works?

- Exact match:- (Function name and Parameter)
- If an exact argument type match is NOT found:
  - then a char, an unsigned char, and short are promoted to an int - and use the first print function.
  - a float argument is promoted to double - and uses the second print function
- If no match found:

- C++ tries to find a match through the standard conversion.
- *ELSE ERROR*

Adapted from:
"Function Overloading in C++" by devilchandra, Geeks for Geeks is licensed under CC BY-SA 4.0

# 14.2.1: Functions that cannot be overloaded in C++

## Overload NOT Allowed

In C++ there are 6 types of function that **CANNOT** be overloaded. We will review and discuss them below.

### Function declarations that differ only in the return type

In C++ (and Java), functions can not be overloaded if they differ only in the return type.

For example, the following program C++ programs will produce errors when compiled.

```
#include<iostream>

int foo() {
    return 10;
}

char foo() {  // compiler error; new declaration of foo()
    return 'a';
}

int main()
{
    char x = foo();
    getchar();
    return 0;
}
```

There are 2 foo() functions - and C++ cannot determine which one to use since they both have no arguments. The only difference between these two functions is the return type - therefore the compiler will flag this error.

## Function declarations with the same name and the name parameter-type-list

```
#include<iostream>
class Test {
   static void fun(int i) {}

   void fun(int i) {}
};

int main()
{
   Test myTest;
   myTest.fun();
   return 0;
}
```

If any of the member functions are declared as a static member function - then they cannot be overloaded. The compiler flags this as an error:

```
error: ‘void Test::fun(int)’ cannot be overloaded
void fun(int i) {}
       ^~~
error: with ‘static void Test::fun(int)’
static void fun(int i) {}
                ^~~
```

### Parameter declarations that differ only in a pointer * versus an array []

The pointer and array are seen by the compiler as equivalent.

```
#include<iostream>
class Test {
   int fun(int *ptr);
   int fun(int ptr[]); // redeclaration of fun(int *ptr)
};

int main()
{
   Test myTest;
   myTest.fun();
   return 0;
}
```

These declarations cause an error because the way that an array reference is passed is the same way that a pointer is passed.

```
error: ‘int Test::fun(int*)’ cannot be overloaded
int fun(int ptr[]); // redeclaration of fun(int *ptr)
    ^~~
error: with ‘int Test::fun(int*)’
int fun(int *ptr);
    ^~~
```

### Parameter declarations that differ only in one is a function type and the other is a pointer to the same function type

```
#include<iostream>
class Test {
     void fun(int ());
     void fun(int (*)()); // redeclaration of fun(int())
};

int main()
{
   Test myTest;
   myTest.fun();
   return 0;
}
```

C++ has difficulties determining between these two function calls...so the compiler flags it as an error.

```
error: ‘void Test::fun(int (*)())’ cannot be overloaded
void fun(int (*)()); // redeclaration of fun(int())
     ^
```

## Parameter declarations that differ only in the presence or absence of const and/or volatile

Again - C++ sees these types of arguments as the same and therefore cannot distinguish between them.

```
#include<iostream>
class Test {
    int fun ( int x) {
        return x+10;
    }

    int fun ( const int x) {
        return x+10;
    }
};

int main()
{
   Test myTest;
   myTest.fun();
   return 0;
}
```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations.

## Two parameter declarations that differ only in their default arguments

The two functions are the same - it is just that one is coded with a default value. This still causes C++ to be unable to distinguis between the two functions.

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

using namespace std;

int fun ( int x, int y) {
    return x+10;
}

int fun ( int x, int y = 10) {
    return x+y;
}
```

```
int main() {
   Test myTest;
   myTest.fun();
   return 0;
}
```

Adapted from:
"Function overloading and return type" by GeeksforGeeks, Geeks for Geeks is licensed under CC BY-SA 4.0
"Functions that cannot be overloaded in C++" by GeeksforGeeks, Geeks for Geeks is licensed under CC BY-SA 4.0

# 14.2.2: Function overloading and const keyword

## Overload and const

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

class Test
{
   protected:
     int num1;
   public:
     Test (int i):num1(i) { }
     void fun() const
     {
         cout << "fun() const called " << endl;
     }
     void fun()
     {
         cout << "fun() called " << endl;
     }
};

int main()
{
    Test t1 (10);
    const Test t2 (20);
    t1.fun();
    t2.fun();
    return 0;
}
```

Output: The above program compiles and runs fine, and produces following output.

```
fun() called
fun() const called
```

The two methods ‘void fun() const’ and ‘void fun()’ have same signature except that one is const and other is not. Also, if we take a closer look at the output, we observe that, ‘const void fun()’ is called on const object and ‘void fun()’ is called on non-const object.

C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function return reference or pointer. We can make one function const, that returns a const reference or const pointer, other non-const function, that returns non-const reference or pointer.

## What about parameters?

Rules related to const parameters are interesting. Let us first take a look at following two examples. The program 1 fails in compilation, but program 2 compiles and runs fine.

```
// PROGRAM 1 (Fails in compilation)
#include<iostream>
using namespace std;

void fun(const int Num1)
{
    cout << "fun(const int) called " << Num1;
}
void fun(int Num1)
{
    cout << "fun(int ) called " << Num1;
}
int main()
{
    const int myNum = 10;
    fun(myNum);
    return 0;
}
```

Output is:

```
error: redefinition of ‘void fun(int)’
void fun(int Num2)
     ^~~
```

Then the second piece of code:

```
// PROGRAM 1 (Fails in compilation)
#include<iostream>
using namespace std;

void fun(const int i)
{
    cout << "fun(const int) called ";
}
void fun(int i)
{
    cout << "fun(int ) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

which produces the outpu:

```
const fun() GeeksforGeeks
```

C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer. That is why the program 1 failed in compilation, but the program 2 worked fine. This rule actually makes sense. In program 1, the parameter ‘myNum’ is passed by value, so ‘Num1’ in fun() is a copy of ‘myNum’ in main(). Hence fun() cannot modify ‘myNum’ of main(). Therefore, it doesn’t matter whether ‘Num1’ is received as a const parameter or normal parameter. When we pass by reference or pointer, we can modify the value referred or pointed, so we can have two versions of a function, one which can modify the referred or pointed value, other which can not.

Adaprted from:
"Function overloading and const keyword" by GeeksforGeeks, Geeks for Geeks is licensed under CC BY-SA 4.0

CHAPTER OVERVIEW

# 15: Polymorphism

**Topic hierarchy**

# 15.1: Polymorphism

## Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
For instance, a real life example of polymorphism would be a man who can be a father, a husband, an employee all at the same moment in time. So the same person posses different behavior in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object Oriented Programming.

**In C++ polymorphism is mainly divided into two types:**

- Compile time Polymorphism
- Runtime Polymorphism

1. **Compile time polymorphism**: This type of polymorphism is achieved by method overloading or operator overloading.

```
// C++ program for method overloading
#include <iostream>


using namespace std;
class CSP31
{
    public:

        // method with 1 int parameter
        void outFunc(int x)
        {
            cout << "value of x is " << x << endl;
        }

        // method with same name but 1 double parameter
        void outFunc(double x)
        {
            cout << "value of x is " << x << endl;
        }

        // method with same name and 2 int parameters
        void outFunc(int x, int y)
        {
            cout << "value of x and y is " << x << ", " << y << endl;
        }
};


int main()
{
    CSP31 myObj1;
    // Which method is called will depend on the parameters passed
    // The first 'outFunc' is called
    myObj1.outFunc(7);

    // The second 'outFunc' is called
    myObj1.outFunc(9.132);

    // The third 'outFunc' is called
    myObj1.outFunc(85,64);
    return 0;
}
```

In the above example, a single method named outFunc() acts differently in three different situations which is the property of polymorphism.

**Method Overloading**: When there are multiple methods with same name but different parameters then these methods are said to be **overloaded**. methods can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

The output of the above code is:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

**Operator Overloading**: C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i =0)
        {
            real = r; imag = i;
        }

        // This is automatically called when '+' is used with
        // between two Complex objects
        Complex operator + (Complex const &obj)
        {
            Complex res;
            res.real = real + obj.real;
            res.imag = imag + obj.imag;
            return res;
        }
        void print()
        {
            cout << real << " + i" << imag << endl;
        }
};


int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

In the above example the operator ‘+’ is overloaded. The operator ‘+’ is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. Notice that in the '+' method there is a keyword "operator" - this is required whenever you are overloading one of the C++ mathematical operators.

The output from the above code is below. If you do not understand real and imaginary numbers don't worry - we will not be delving into that topic in this course.

```
12 + i9
```

2. **Runtime polymorphism**: This type of polymorphism is achieved by method Overriding.
    - **Method overriding** occurs when a derived class has a definition for one of the member methods of the base class. That base method is said to be **overridden**.

```
// C++ program for method overriding
#include <iostream>
using namespace std;

class base
{
    public:
        virtual void print ()
        {
            cout<< "print base class" <<endl;
        }

        void show ()
        {
            cout<< "show base class" <<endl;
        }
};


class derived:public base
{
    public:
        //print () is already virtual method in derived class
        // we could also declared as virtual void print () explicitly
        void print ()
        {
            cout<< "print derived class" <<endl;
        }

        void show ()
        {
            cout<< "show derived class" <<endl;
        }
};


//main method
int main()
{
    base *bptr;
    derived deriv;
    // Assigne the address of the derived class instance
    bptr = &deriv;

    // print() is a virtual method in the parent class
    // bound at runtime (Runtime polymorphism)
    bptr->print();

    // show() is a non-virtual method in the parent class
    // bound at compile time
    bptr->show();


    return 0;
}
```

Output:

```
print derived class
show base class
```

## Inheritance and Polymorphism.

There are some differences between Inheritance and Polymorphism shown in the table below

| | INHERITANCE | POLYMORPHISM |
|---|---|---|
| 1. | Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class). | Whereas polymorphism is that which can be defined in multiple forms. |
| 2. | It is basically applied to classes. | Whereas it is basically applied to functions or methods. |
| 3. | Inheritance supports the concept of reusability and reduces code length in object-oriented programming. | Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding). |
| 4. | Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance. | Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding). |
| 5. | It is used in pattern designing. | While it is also used in pattern designing. |

Adapted from:
"Polymorphism in C++" by Harsh Agarwal, Geeks for Geeks
"Difference between Inheritance and Polymorphism" by MKS075, Geeks for Geeks

# CHAPTER OVERVIEW

## 16: Inheritance

# 16.1: Inheritance

## Inheritance in C++

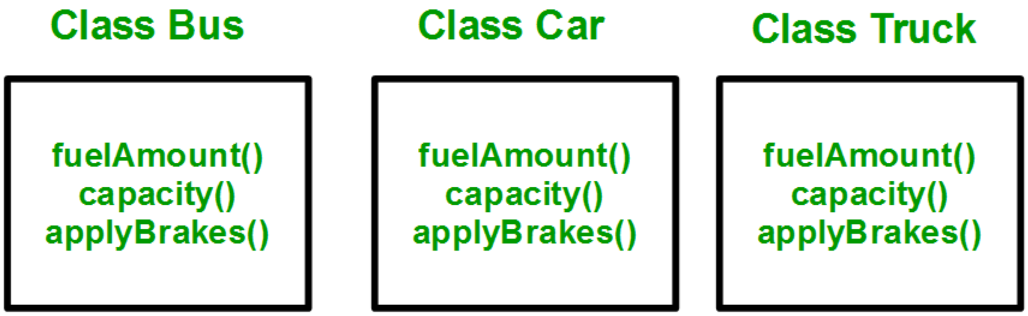
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.
**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class - also often referred to as the child class.
**Super Class:**The class whose properties are inherited by sub class is called Base Class or Super class - also often referred to as the parent class.

The article is divided into following subtopics:

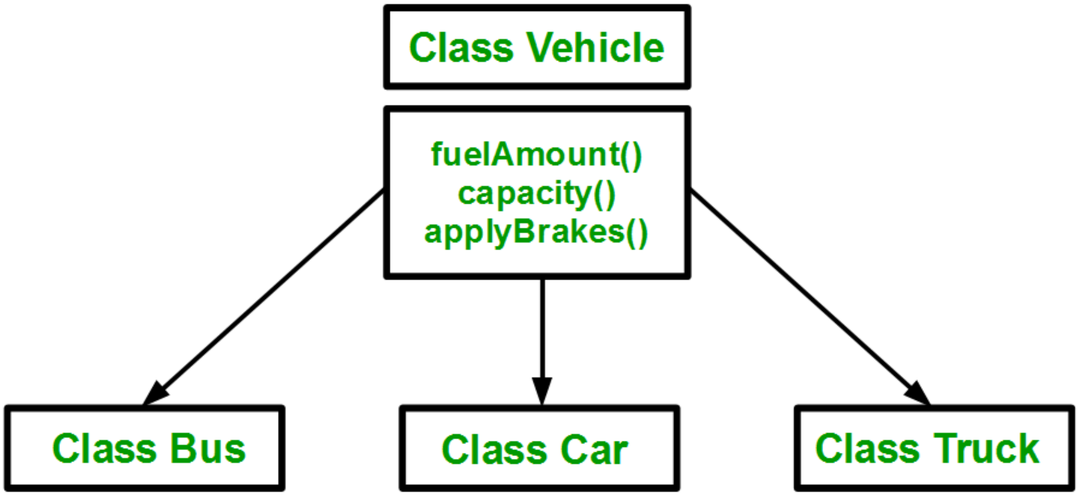
1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

### Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

**Implementing inheritance in C++**: For creating a sub-class which is inherited from the base class we have to follow the below syntax.

```
class child_class_name : access_mode parent_class_name
{
  //body of child_class
};
```

Here, **child_class_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **parent_class_name** is the name of the base class from which you want to inherit the sub class.

**Note**: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
// C++ program to demonstrate implementation of Inheritance
#include <bits/stdc++.h>
using namespace std;

//Base class
class School
{
    public:
        string name_p;
};

// Sub class inheriting from Base Class(School)
class Department : public School
{
    public:
        string name_c;
};

//main function
int main()
{
        Department deptObj;

        // An object of class child has all data members
        // and member functions of class parent
        deptObj.name_c = "Computer Science";
        deptObj.name_p = "Delta College";

        cout << "Child name is " << deptObj.name_c << endl;
        cout << "Parent name is " << deptObj.name_p << endl;

        return 0;
}
```

In the code there is a parent class called School, and the child class which is Department. We could have a lot more variable and methods in the parent class, but we are trying to keep this simple. The child class inherits the methods and variables of the parent class - EXCEPT - the child class can not access a parents private variables or methods.

Output:

```
Child name is Computer Science
Parent name is Delta College
```

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

Adapted from:
"Inheritance in C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 16.1.1: Inheritance - Modes of Inheritance

**Modes of Inheritance**

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

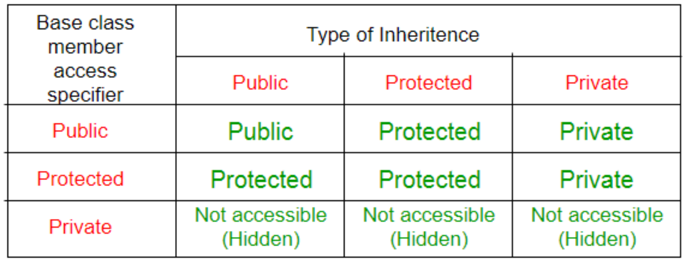
**Note :** The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

*filter_none*

```
// C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object
class A
{
   public:
      int x;
   protected:
      int y;
   private:
    int z;
};


class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};


   class C : protected A
   {
    // x is protected
    // y is protected
    // z is not accessible from C
   };


   class D : private A // 'private' is default for classes
   {
    // x is private
    // y is private
    // z is not accessible from D
   };
```

*brightness_4*

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



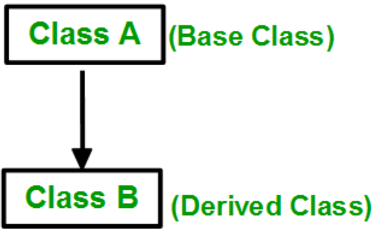
| Base class member access specifier | Type of Inheritence | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not accessible (Hidden) | Not accessible (Hidden) | Not accessible (Hidden) |

Adapted from:
"Inheritance in C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

# 16.1.2: Inheritance - Types of Inheritance

## Types of Inheritance in C++

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



**Syntax**:

```
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

*filter_none*

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle()
    {
    cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

*play_arrow*

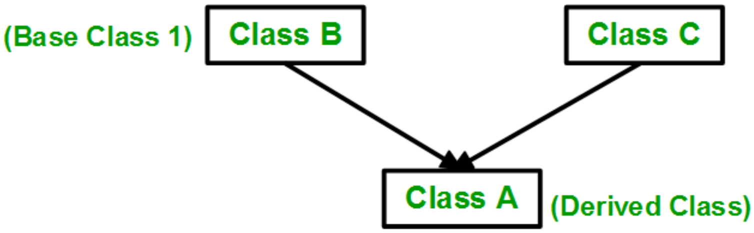
*brightness_4*

```
// C++ program to explain
// Single inheritance
#include <iostream>
using  namespace  std;

// base class
class  Vehicle {
    public  :
      Vehicle()
      {
        cout <<  "This is a Vehicle"  << endl;
      }
};

// sub class derived from two base classes
class  Car:  public  Vehicle{

};

// main function
int  main()
{
      // creating object of sub class will
      // invoke the constructor of base classes
      Car obj;
      return  0;
}
```

Output:

```
This is a vehicle
```

2. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



**Syntax**:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

*filter_none*

*edit*

*play_arrow*

*brightness_4*

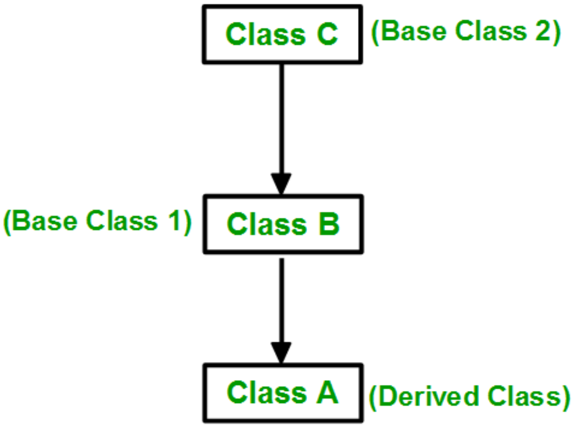
```
// C++ program to explain
// multiple inheritance
#include <iostream>
using  namespace  std;

// first base class
class  Vehicle {
    public  :
      Vehicle()
      {
        cout <<  "This is a Vehicle"  << endl;
      }
};

// second base class
class  FourWheeler {
    public  :
      FourWheeler()
      {
        cout <<  "This is a 4 wheeler Vehicle"  << endl;
      }
};

// sub class derived from two base classes
class  Car:  public  Vehicle,  public  FourWheeler {

};

// main function
int  main()
{
      // creating object of sub class will
      // invoke the constructor of base classes
      Car obj;
      return  0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

Please visit this link to learn multiple inheritance in details.

3. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



*filter_none*

*edit*

*play_arrow*

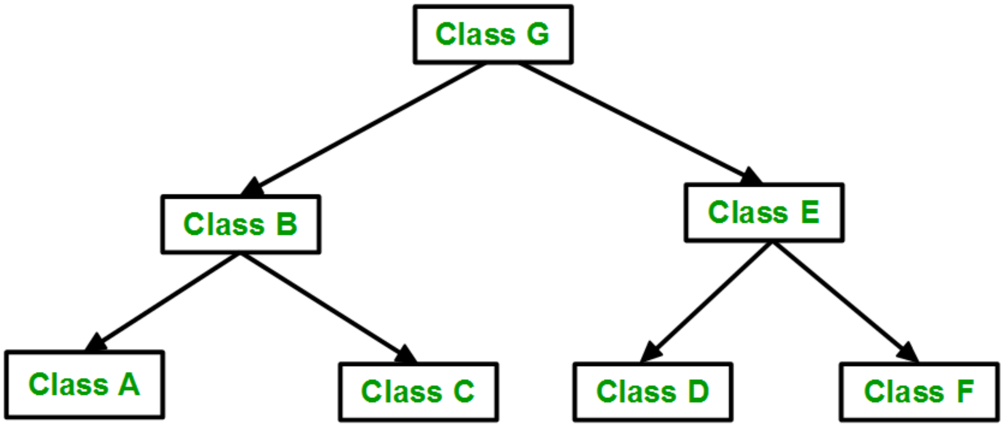
*brightness_4*

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using  namespace  std;

// base class
class  Vehicle
{
    public  :
      Vehicle()
      {
        cout <<  "This is a Vehicle"  << endl;
      }
};
class  fourWheeler:  public  Vehicle
{   public  :
      fourWheeler()
      {
        cout<<  "Objects with 4 wheels are vehicles"  <<endl;
      }
};
// sub class derived from two base classes
class  Car:  public  fourWheeler{
     public  :
       car()
       {
         cout<<  "Car has 4 Wheels"  <<endl;
       }
};

// main function
int  main()
{
      //creating object of sub class will
      //invoke the constructor of base classes
      Car obj;
      return  0;
}
```

output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

4. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



*filter_none*

*edit*

*play_arrow*

*brightness_4*

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using  namespace  std;

// base class
class  Vehicle
{
    public  :
      Vehicle()
      {
        cout <<  "This is a Vehicle"  << endl;
      }
};


// first sub class
class  Car:  public  Vehicle
{

};

// second sub class
class  Bus:  public  Vehicle
{

};

// main function
int  main()
{
      // creating object of sub class will
      // invoke the constructor of base class
      Car obj1;
      Bus obj2;
      return  0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

Adapted from: "Inheritance in C++" by Harsh Agarwal, Geeks for Geeks is licensed under CC BY-SA 4.0

CHAPTER OVERVIEW

# 19: Handling Errors

# 19.1: Errors and Exceptions

# 19.2: How Exceptions Work

# 19.3: Throwing Exceptions

# 19.4: Error Handling Issues

# this is a test

# Index

# Glossary

**Sample Word 1** | Sample Definition 1

# Detailed Licensing

## Overview

**Title:** C++ Programming I (McClanahan)

**Webpages:** 147

**All licenses found:**

- CC BY-SA 4.0: 88.4% (130 pages)
- Undeclared: 11.6% (17 pages)

## By Page

- C++ Programming I (McClanahan) - *CC BY-SA 4.0*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
    - Front Matter - *Undeclared*
      - TitlePage - *Undeclared*
      - InfoPage - *Undeclared*
      - Table of Contents - *Undeclared*
  - 1: Building and Running C++ Code - *CC BY-SA 4.0*
    - 1.1: Building a C++ Program - *CC BY-SA 4.0*
    - 1.2: Understanding Preprocessing - *CC BY-SA 4.0*
      - 1.2.1: Understanding Preprocessing- File Inclusion - *CC BY-SA 4.0*
      - 1.2.2: Understanding Preprocessing- Conditional Compilation - *CC BY-SA 4.0*
    - 1.3: Header Files - *CC BY-SA 4.0*
    - 1.4: Compiling - *CC BY-SA 4.0*
    - 1.5: Compiling- Analysis Phase - *CC BY-SA 4.0*
    - 1.6: Intermediate steps - *CC BY-SA 4.0*
    - 1.7: Linking and Libraries - *CC BY-SA 4.0*
    - 1.8: Standard Libraries - *CC BY-SA 4.0*
  - 2: C++ Basics - *CC BY-SA 4.0*
    - 2.1: The "main" concern.... - *CC BY-SA 4.0*
    - 2.2: The Importance of Good Looking Code - *CC BY-SA 4.0*
    - 2.3: Comments are Required - *CC BY-SA 4.0*
    - 2.4: Do I Need to Decompose a Bit - *CC BY-SA 4.0*
    - 2.5: Naming Concepts - *CC BY-SA 4.0*
    - 2.6: The Great Bracket Debate - *CC BY-SA 4.0*
    - 2.7: Spaces and Such - *CC BY-SA 4.0*
  - 3: Program Planning and Design - *CC BY-SA 4.0*
    - 3.1: What is Software Development - *CC BY-SA 4.0*
    - 3.2: Why is Design Important - *CC BY-SA 4.0*
    - 3.3: Program Design - *CC BY-SA 4.0*
    - 3.4: Psuedocode - *CC BY-SA 4.0*
    - 3.5: Test Data - *CC BY-SA 4.0*
    - 3.6: Two Good Ideas to Keep in Mind - *CC BY-SA 4.0*
  - 4: Data and Operators - *CC BY-SA 4.0*
    - 4.1: We Begin to Code... - *CC BY-SA 4.0*
    - 4.2: Data Types in C++ - *CC BY-SA 4.0*
    - 4.3: Identifier Names - *CC BY-SA 4.0*
    - 4.4: Constants and Variables - *CC BY-SA 4.0*
      - 4.4.1: Literals and Constants - Integers - *CC BY-SA 4.0*
      - 4.4.2: Literals and Constants - Floating Point - *CC BY-SA 4.0*
      - 4.4.3: Literals and Constants - Characters - *CC BY-SA 4.0*
      - 4.4.4: Literals and Constants - Strings and Booleans - *CC BY-SA 4.0*
    - 4.5: Data Manipulation - *CC BY-SA 4.0*
    - 4.6: Assignment Operator - *CC BY-SA 4.0*
    - 4.7: L Value and R Value - *CC BY-SA 4.0*
    - 4.8: Sizeof Operator - *CC BY-SA 4.0*
    - 4.9: Arithmetic Operators - *CC BY-SA 4.0*
    - 4.10: Data Type Conversions - *CC BY-SA 4.0*
    - 4.11: Operator Overloading - *CC BY-SA 4.0*
    - 4.12: Unary Positive and Negative Operators - *CC BY-SA 4.0*
    - 4.13: Bitwise Operators - *CC BY-SA 4.0*
  - 5: Common Data Types - *CC BY-SA 4.0*
    - 5.1: Integer Data Type - *CC BY-SA 4.0*
    - 5.2: Floating-Point Data Type - *CC BY-SA 4.0*
    - 5.3: String Data Type - *CC BY-SA 4.0*
    - 5.4: Character Data Type - *CC BY-SA 4.0*
    - 5.5: Interger Division and Modulus - *CC BY-SA 4.0*
    - 5.6: Typedef - An Alias - *CC BY-SA 4.0*
    - 5.7: Sequence Operator - *CC BY-SA 4.0*
  - 6: Conditional Execution - *CC BY-SA 4.0*
    - 6.1: Conditional Execution - *CC BY-SA 4.0*
    - 6.2: If - Else - *CC BY-SA 4.0*
      - 6.2.1: Conditional Execution - If-Else - *CC BY-SA 4.0*
    - 6.3: Boolean Data Type - *CC BY-SA 4.0*