CS 211 Notes

Introduction to Programming

Jaeden Bardati

Last modified December 16, 2021

0 Course Overview

September 10, 2021

0.1 Objectives

The course is intended to teach how to develop a computer program to solve a problem. C++ is a tools that will be used to develop these skills and logical thinking. These skills will be transferable to other languages.

1 Computer Organization

1.1 Hardware

September 11, 2021

1.1.1 Components

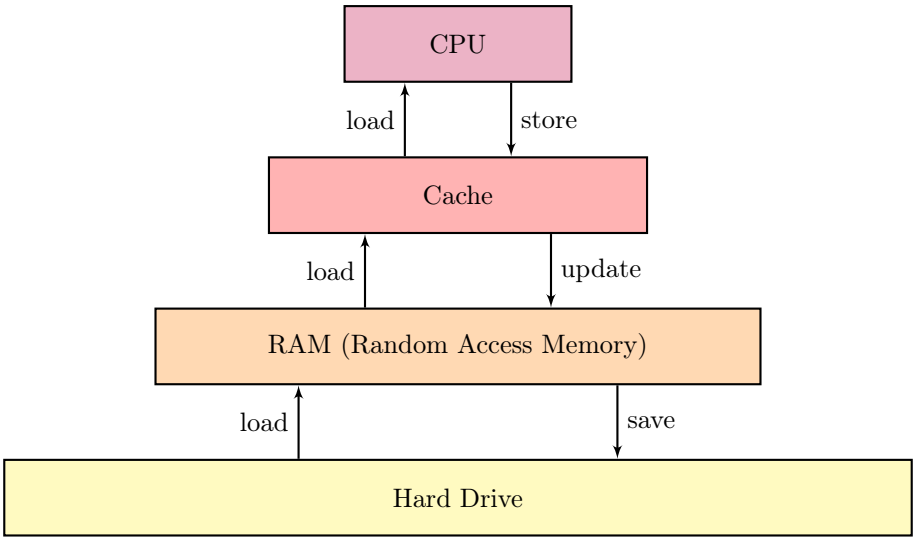
Modern computers are built using the **Von Neumann machine**. There are three aspects:

- **Architecture**: **I/O** (User interaction) + **Memory** (Storage) + **CPU** (CU: Control Unit, ALU: Arithmetic and Logic Unit). These are all connected by a shared bus.
- **Stored Programs**: All programs and data are stored in memory (binary).
- **Sequential Execution**: Also called the **fetch-decode-execute** cycle. Instructions are **fetched** from memory, **decoded** by the CU and then **executed** by the ALU. If there is a result, it is stored back in memory.

1.2 Memory

September 11, 2021

The memory is organized in a **hierarchy**. At the bottom of the hierarchy is the Hard Drive (in TB). At the top is the CPU. Since the hard drive is slow, when some data from the hard drive is needed, it is first loaded into **RAM (Random Access Memory)** (in GB). The RAM is still too slow for the RAM, so the data is stored in **cache** (in KB or MB). Yet still, this is not fast enough for the CPU, so **registers** (in Bytes) in the CPU itself are used to store variables.



As you **go up** the hierarchy, the **speed increases**, but the **size decreases** and the **cost increases**.

1.2.1 RAM

Random access memory is organized in an array of Bytes ("words").

Words in RAM are addressed with a byte themselves (e.g. 01101101 is an address). These are typically written in hexadecimal (e.g. 6D).

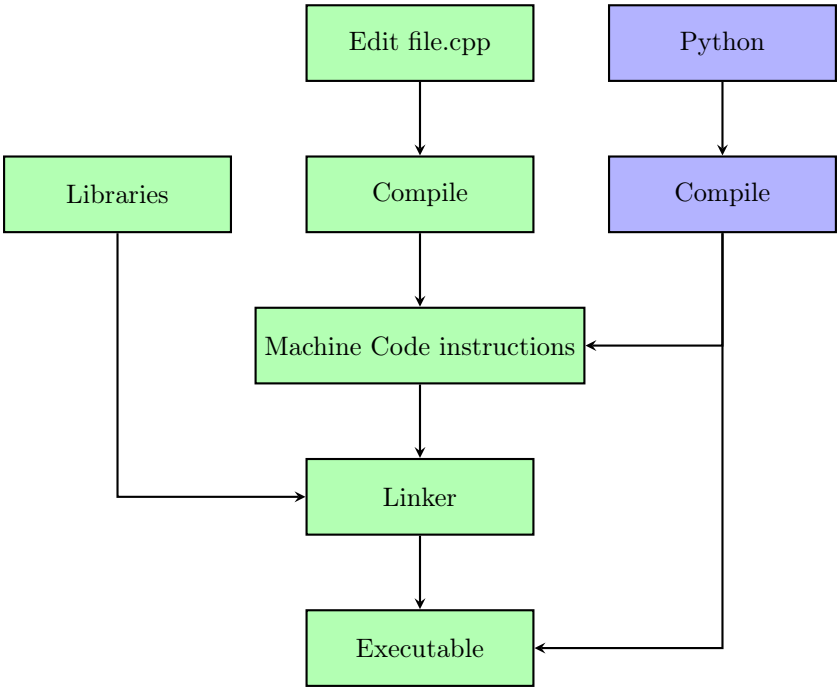
Words in RAM can be data or machine code instructions. Instructions contain a binary code for each operation (for example, addition). Instructions codes are dependent on the CPU.

2 C++ Programming

2.1 A Flow Chart: Program to Binary

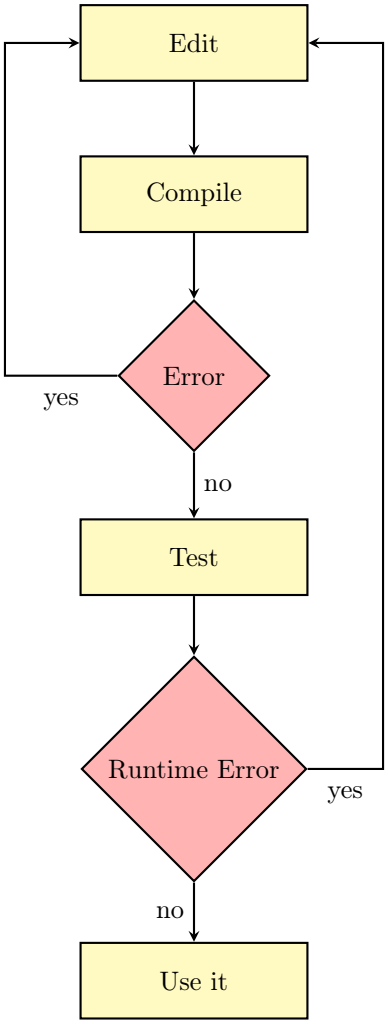
September 12, 2021

This is a flow chart of what is done by the computer when compiling a C++ file. In blue is the Python equivalent.



Note that the bottom of the flow chart is the same for all programming languages, because in all languages, CPU-specific machine code is needed to execute code.

The process of catching errors is as according to the following flow chart.



In this context, **errors** are caught by the compiler. This is opposed to **runtime errors**, which are not caught by the compiler. These can be something like division by zero or infinite loops.

2.2 First C++ Code

September 12, 2021

The following code is a hello world program in C++.

```
// helloworld.cpp
#include <iostream>   // include statement allows the use of C++ libraries
#include <stdio.h>    // this library contains getchar()

using namespace std; // a standard environment (input from keyboard, output is the
    screen)

int main() {

   cout << "Hello world!" << endl;  // Prints "Hello world!" to the screen

   getchar();     // wait for user to type a character

   return 0;     // 0 means that the execution was successful
```

```
}
```

2.3 Data types

September 12, 2021

Variables are referred to as identifiers. Identifiers are memory locations accessed and modified.

Inside a main function (as above), the following code declares a variable of integer type in C++.

```
int numYears;     // allocated space in memory to contain num of years
```

You can also initialize the variable with a value on declaration:

```
int number = 5;   // declare and initialize (give a value too)
```

Some rules to follow when naming variables are:

- Names have meanings
- Must be case sensitive (e.g. numYears is not numyears)
- Consists of letters, numbers and underscores
- First character cannot be a number

Some types of variables are:

- Integers (e.g. -5, 0, +2) [int]
- Real numbers (floating point numbers or doubles, e.g. 2.453, -4.1987e7) [float or double]
- Booleans (e.g. true, false) [bool]
- Characters [char]

You can assign a value to a variable after declaring it:

```
int width, height;
int area;

width = 5;
height = 3;

area = width * height;
```

Constants (denoted with the keyword "const") cannot be changed throughout the program. The convention is to use capital letters for constants.

```
const double PI = 3.14159265;

int radius = 6;
double area = PI * radius * radius;
```

You would get an error if you were to try to reassign a constant.

```
const double PI = 3.14159265;
PI = 3.14;  /// ERROR
```

2.4 Arithmetic operations

September 22, 2021

There are 5 basic arithmetic operations supported by C++

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus or "Remainder" (%)

For example,

```
int a = 5;
int b = 2;

cout << a + b / 2 << endl;    /// 6
cout << (a + b) / 2 << endl;  /// 3
cout << a / 2 + 3 << endl;    /// 5
cout << a / b / 3 << endl;    /// 0
```

This is done with integer operations since both arguments of each operation are integers (round down if the result of a division is a decimal). If one argument is a double/float, the result is a double/float. For example,

```
int a = 5;
int b = 2;

cout << a + b / 2 << endl;    /// 6
cout << (a + b) / 2 << endl;  /// 3.5
cout << a / 2 + 3 << endl;    /// 5
cout << a / b / 3 << endl;    /// 0.83333333
```

2.4.1 Assigning and operations

When you are assigning a variable and performing an operation to it at the same time you can do the following:

```
int x = 25;

x += 1     // equivalent to x = x + 1 (or x++)
x -= 2;    // equivalent to x = x - 2
x *= 3;    // equivalent to x = x * 3
x /= 4;    // equivalent to x = x / 4
x %= 5;    // equivalent to x = x / 5
```

2.4.2 Division when initializing or assigning a variable

Naively, to initialize a variable to the result of a division, you could try

```
int div = 7/3;  /// 2
```

However, this results in 2, not the desired result of 2.3333. You could also try changing the variable type

```
double div = 7/3;  /// 2
```

This does not work either, since the integer division happens before the result is assigned to the variable div. Instead, you have to force the compiler to recognize an argument as a float.

```
double div = 7.0/3;   /// 2.333333
```

If instead you had decided to make the variable type of div and integer and run it

```
int div = 7.0/3;   /// 2
```

It will perform the double operation, but save it in var as an integer.

A way of doing the float operation with two integers is by **casting** types. This can be done as follows.

```
double div = (double) 7/3;  /// 2.333333
double div2 = 7/(double)3;  /// 2.333333

int c = 7
double div3 = (double) c/3;   /// 2.333333
// Note, this does not change the type of c.
```

The remainder (%) operation returns the remainder of the division between two integers:

```
cout << 7 % 3 << endl;   /// 1
cout << 10 % 6 << endl;  /// 4
cout << 13 % 4 << endl;  /// 1
```

2.4.3 Type conversion

We can also convert types from doubles to integers by truncation or rounding:

```
double price = 2.55;

int sum = price;   /// truncate
int sum2 = price + 0.5  /// round to the nearest int
```

2.5 I/Os

September 22, 2021

To do this, we will use the iostream library and standard namespace.

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

2.5.1 Reading inputs

You can get input from the keyboard using cin. For example,

```
int number;

cout << "Please enter a value between 0 and 10." << endl;

cin >> number;  // reads integer

cout << "The entered number is " << number;
```

You can also get multiple variables with the same cin.

```
int length, width;

cout << "Please enter the length and width of the rectangle: ";

cin >> length >> width;

int area = length * width;
cout << "The area is " << area
```

2.5.2 Formatting outputs

When outputting the result of a float, we inevitably run into an issue:

```
double result = 123456789.1284567;  /// 1.23457e+008

cout << result << endl;
```

This returns 1.23457e+008. To see all the digits, we must first pass result to fixed before cout.

```
double result = 123456789.1284567;

cout << fixed << result << endl;  /// 123456789.128457
```

To properly format the output of doubles, we must include the iomanip library and use the setprecision(n) function.

```
#include <iomanip>

double result = 123456789.1284567;

cout << fixed << setprecision(2) << result << endl;  /// 123456789.13
cout << result << endl;                    /// 123456789.13
```

This rounds the double to the nearest nth decimal. Note that result will now format that way if outputted without the fixed and setprecision call. However, this does not actually change the value of the result variable.

```
double result2 = result + 1;

cout << setprecision(3) << result2 << endl;   /// 123456790.128
```

Finally, to set a limit to the number of characters that are displayed to an integer n, we can use the setw(n) function from the iomanip library like this

```
cout << fixed << setprecision(2) << setw(15) << left << result << "End of line" <<
    endl;
```

The result is "123456789.13 End of line". Note the exactly 15 characters until "End of line".

2.6 Strings

September 22, 2021

To use strings, you have to include the string library:

```
#include <string>
```

2.6.1 Declaration

You can declare, initialize, assign, and output strings as such

```
string lname;
string fname = "Jaeden";
lname = "Bardati";

cout << fname << " " << lname;   /// Jaeden Bardati
```

2.6.2 Concatenation

To combine strings together, we must use **concatenation**.

```
string name = fname + " " + lname;  /// Jaeden Bardati
```

This method does not work for constants

```
string greeting = "Hello" + " " + " World";  /// ERROR
```

An error occurs because we are trying to add the string " " to the constant string "Hello". A way to get around this is by adding a dummy variable before the concatenation.

```
string empt;    // empt = ""
string greeting = empt + "Hello" + " " + " World";  /// Hello World
```

2.6.3 Inputting strings

To read in a string, we can naively try

```
string myname;

cin >> myname;  /// Jaeden Bardati

cout << myname;    /// Jaeden
```

Only one word at a time is read in because cin treats things separated by spaces as distinct.

```
string myname;
string myfname, mylname;

cin >> myfname >> mylname;      /// Jaeden Bardati

myname = myfname + " " + mylname;

cout << myname << endl;         /// Jaeden Bardati
```

To get a full sentence (string) with cin, use the function getline(cin, sentence); This will get the whole line and put it in the variable sentence.

2.6.4 String functions

A useful quantity to know for a given string could be its length. You can get it as follows

```
int n = myname.length();

cout << "Your name has " << n-1 << " characters" << endl;
```

Here the number of characters is $n - 1$ since the space is not included.

You can also get substrings using substr(i, n) to get the n characters proceeding the character at location i. If only i in entered, it will extract all characters preceding the character at location i.

```
string sub = myname.substr(0, 6);  /// Jaeden
string sub2 = myname.substr(7);    /// Bardati
```

2.6.5 Characters

Characters are strings of length 1, however, they can be declared explicitly with the char keyword and are expressed using single quotes.

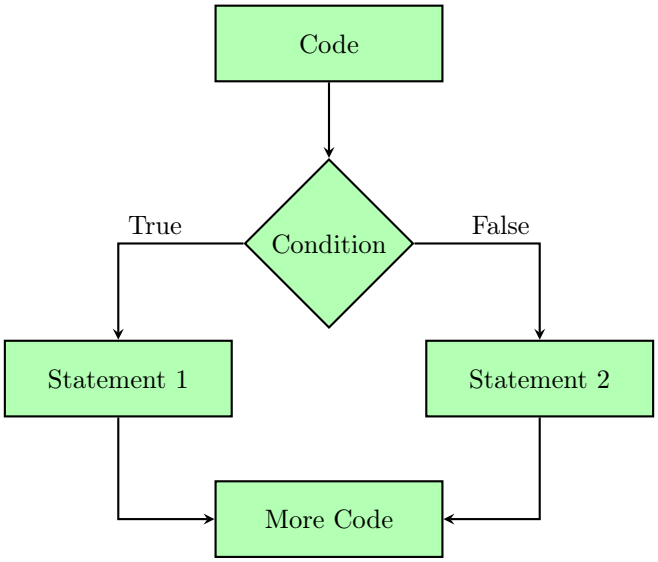
```
char c = 'Y';
```

2.7 The If Statement

September 23, 2021

We can also use conditional statements.

2.7.1 Flow Chart



2.7.2 Syntax

The if statement has the syntax

```
if (condition)
   statement1;
else
   statement2;
```

For multiple statements, we can use parentheses

```
if (condition) {
   statement;
   statement;
   ...
}
```

```
else {
   statement;
   statement;
   ...
}
```

2.7.3 The condition

A condition is a **boolean expression**. A boolean type can be **true** or **false**.

Boolean expressions are written using **relational operators**:

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal to
!=	Not equal to

You can also chain relational operators with **logic operators**:

||	OR
&&	AND
!	NOT

For example,

```
if (num < 20 || num % 3 == 1) // if num is < 20 or if the remainder of num / 3 is 1
   num = num * 2
else
   num = num / 2
```

2.7.4 Floating point precision error

Let us show an example of a floating point precision error. Note that here we are including the cmath library.

```
double root, number;

root = sqrt(3);
number = pow(root, 2);

cout << setprecision(18) << "Number is : " << number << endl; // 2.999999999999999956,
    not 3

if (number == 3)
   cout << "sqrt(3) * sqrt(3) = 3" << endl;
else
   cout << "What just happened?" << endl;
```

When running this in a main function, we obtain the "What just happened?" result. What just happened indeed? The issue is that floating point numbers are only precise to a certain number of digits. To account for this, we must set our own precision as follows:

```
double precision = 1E-14;

if (abs(number - 3) < precision)
   cout << "sqrt(3) * sqrt(3) = 3" << endl;
else
   cout << "What just happened?" << endl;
```

This returns the desired result of "sqrt(3) * sqrt(3) = 3".

2.7.5 String comparison (alphabetical order)

We can also use comparison operators for strings. For example,

```
string name1, name2;

cin >> name1 >> name 2;

if (name1 < name2)
   cout << name1 << endl << name2;
else
   cout << name2 << endl << name1;
```

This will return two entered strings in alphabetical order.

2.7.6 Nested ifs

Sometimes there are more than one options. To do this we can use nested ifs. You check conditions within conditions for multiple branchings.
For example, we can assign a letter grade for a given percentage grade.

A	80 <= grade <= 100
B	65 <= grade < 80
C	50 <= grade < 65
F	0 <= grade < 50

The implementation of this in code is

```
int grade;
char letter;

cout << "Please enter a numeric grade: ";
cin >> grade;

if (grade >= 50)
   if (grade >= 65)
      if (grade >= 80)
         letter = 'A';
      else
         letter = 'B';
   else
      letter = 'C';
else
   letter = 'F';

cout << "The letter grade is " << letter << endl;
```

2.8 The Switch Statement

September 30, 2021

The **switch statement** is an alternative to nested ifs. This statement can be used if it compares a boolean, integer or character against a constant of the same type. Each constant creates an **alternative** or **branch**.

The syntax is as follows:

```
int n;
cin >> n;

switch (n) {
   case 1: cout << "ONE"; break;
   case 2: cout << "TWO"; break;
   case 3: cout << "THREE"; break;
   case 4: cout << "FOUR"; break;
   case 5: cout << "FIVE"; break;
   default: cout << "Not a number between 1 and 5";
}
```

The general process of the switch statement is that it will check each "case" until it finds a match, then it will run what comes next. The "default" will run if no other case has run. Remember to put breaks after each case! No break is needed after the default case.

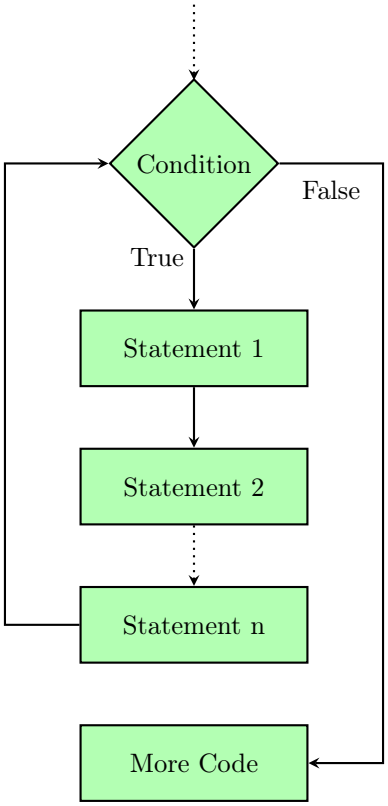
You can also stack cases as follows:

```
char c;
cin >> c;

switch (c) {
   case 'y': case 'Y': cout << "YES"; break;
   case 'n': case 'N': cout << "NO"; break;
   default: cout << "Not y or n";
}
```

2.9 The While Loop

September 30, 2021

The **while loop** can be used when a sequence of statements must be repeated multiple times so long as a condition is true. The following is a flow chart of the process.



The syntax for a while loop of *one statement* is

```
while (condition)
   statement;
```

Similarly, a while loop of *multiple statements* is made possible using parenthesis

```
while (condition) {
   statement;
   statement;
   ...
}
```

There are a few consideration we need to take into account:

- The condition must be initially true, other wise the loop will never execute
- If the condition is initially true, the loop executes at least once
- The loop executes until the condition becomes false
- The condition becomes false as a results of a statement in the loop
- If the condition never becomes false, the loop will be infinite

The following code is an example program using a while loop, to calculate how many years it takes for a bank account balance to double with a given rate:

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

int main() {
   // Calculates the number of years to double the balance of a bank account.
   const double RATE = 0.03;

   double bal;

   cout << "Please enter an initial balance: ";
   cin >> bal;

   double target = 2 * bal;
   int years = 0;

   while(bal < target){
      bal += (bal * RATE);
      years++;
   }

   cout << "You require " << years << " years to double your balance.";

   return 0;
}
```

Quick Aside: In order to get the last number of a digit, you can use mod 10. For example, to get the last number of 729, we find that 729 % 10 = 9. We can use integer division by 10 to make 729 become 72 and then get the last number of that in order to get the second number form the right (here, 2).

Another Quick Aside: If a variable is entered that is not of the right type in cin, it will fail. There is a function that checks if it fails: cin.fail(). It will return false if cin has not failed, and true if it has. After a cin fails, all the subsequent cins will fail. To allow for subsequent cins after failure, we can use cin.clear(). However, since the failed character is still in the buffer, you cin will fail again. Therefore, we need to also call cin.ignore(n, lastchar) after cin.clear() to ignore what is in the buffer that caused it to fail (up to n characters or up to lastchar). The following code

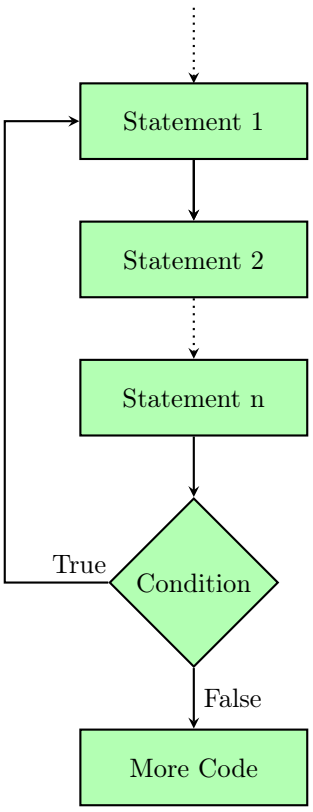
demonstrates this.

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

int main(){

   int input;

   cout << "Enter an integer value: " << endl;

   cin >> input;
   cout << "You have entered: " << input << " Status of cin: " << cin.fail() << endl;

   cin.clear();
   cin.ignore(10000, '\n');

   cout << "The cin status is now : " << cin.fail() << endl;

   cin >> input;
   cout << "You have entered: " << input << " Status of cin: " << cin.fail() << endl;

   return 0;
}
```

2.10 The Do-While Loop

October 6, 2021

The **do-while loop** can be used when a sequence of statements must be repeated multiple times so long as a condition is true. The following is a flow chart of the process.



This is different form the regular while loop because it always executes at least once. If the condition is initially false, there is one one execution. If the condition is initial true, the body of the loop is executed until the condition becomes false. The minimum number of iterations of a do-while loop is 1, whereas the minimum number of iterations for a while-loop is 0.

The syntax in C++ for one statement is

```
do
   statement;
while(condition);
```

For many statements, you need curly braces as usual.

```
do {
   statement1;
   statement2;
   ...
}
while(condition);
```

An example application of a do-while loop is in input validation:

```
/// input validation using a do-while
do {
   cout << "Please enter a value for 0 < n < 100 ";
   cin >> n;
} while(n <= 0 || n >= 100);
```

2.11 The For Loop

October 6, 2021

The for loop is yet another iterative statement. For loops are based on the use of counters. It has the syntax:

```
for(statement1;statement2;statement3)
   statement;
```

Or, for multiple statement loops

```
for(statement1;statement2;statement3) {
...
statement;
...
}
```

The statement1 is for counter initialization.
The statement2 is for the loop condition.
The statement3 is to update the counter.

To demonstrate this we will compare it to the while loop.

```
int i = 1;
while (i <= n) {
   sum = sum + i;
   i = i + 1;
}
```

```
for (int i=1; i<=n; i++)
   sum = sum + i
```

We can see that the for loop is much more compact. It is preferred to use a for loop when you are using counters.

2.11.1 Nested For Loops

Just as if statements can be nested, so can loops. Nested loops are usuful since sometimes it is necessary to loop over loops (or repeat loops a certain number of times).

For example, if we wanted to draw an n by n square of *s. We could use a nested loop:

```
int n = 4;

for(int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j++)
      cout << "*";
   cout << endl;
}
```

You can even change the number of iterations in the inner loop via nested loops. For example, if we wanted the pattern

```
*
**
***
****
```

Then, we could use the following code.

```
int n = 4;

for(int i = 1; i <= n; i++) {
   for (int j = 1; j <= i; j++)
   cout << "*";
   cout << endl;
}
```

2.12 Random Variables
October 6, 2021

Sometimes it is useful to have (pseudo)random numbers to use in a program. The library that we need to include is

```
#include <cstdlib>
```

Then, to get two random numbers, we can do

```
int r1, r2;

r1 = rand();
r2 = rand();

cout << r1 << endl << r2 << endl;
```

Running it again though, you can notice that it gives you the same numbers as before!

This is because the the number is not truly random, only pseudo-random (random-looking). They are really just values that are generated from a mathematical sequence.

This sequence has to start somewhere and that number is called the "seed". We can set the seed value using

```
srand(765); // 765 is some arbituary number
```

If we want this to change with time, we need to input the computer time. To do this we will import the following library:

```
#include <ctime>
```

We can get the time in seconds since January 1st 1970 using

```
cout << time(0);
```

We can use this as the seed to make the starting random numbers change over time.

```
srand(time(0));
```

What is the range of the outputted randome numbers by rand()? It is between 0 and RAND_MAX. This is a constant defined in the cstdlib library. We can get this value ourselves:

```
cout << RAND_MAX << endl;
```

If we want doubles instead, we can do

```
int r = rand();
double x = r*1.0/RAND_MAX;
```

The variable x is a random variable between 0 and 1.

What if we want to generate random variables between and arbitrary a and b? To do this, we need to follow this procedure:

- Generate r between 0 and RAND_MAX (via rand())
- Generate x between 0 and 1 (via x = r*1.0/RAND_MAX))
- Generate y = a + (b-a)*x

So, if x = 0, y = a + (b-a)*0 = a. If x = 1, y = a + (b-a)*1 = a + b - a = b.

In code, this is

```
int r = rand();
double x = r*1.0/RAND_MAX;
double y = -1 + 2 * x;

cout << y << endl;
```

2.13 Functions

2.13.1 Introduction

A function is a sequence of instructions with a name. It is almost like a mini-program.

Why do we use functions?

- Convenient for splitting the work into modules
- Hiding details from the main() and making main() easier to understand (e.g. library function)
- Avoid repetitions, when portions of the code have to be used more than once

A function can either

- Calculate and return a value
- Perform a task

2.13.2 Declaration and Call

The function is **declared**, **defined** and **called** with a statement, just like any library function.

For example,

```
#include <cmath>
double x = 50.75;
double y = sqrt(x);
double z = pow(x, 3);
double cube(x); //but cube is not declared
int n = incr_by_value(n, value) //same as above
```

To declare a function, we need a **function heading**. We also should comment and explain what the function does. Something like

```
return_type fnct_name*(type ident, type ident, ...) {
   \\ function b ody
   Local declaration
   Calculations
   Return statement
}
```

For example, a function that calculates the cube of a value for a given variable of type double could be

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

/*******************************************
Function cube calculates the volume of a cube.
Parameter side: The side of the cube.
*******************************************/

double cube(double s){
   double v = s*s*s;
   return v;
}


int main(){
   double side;
   cout << "Please enter the value of side: ";
   cin >> side;

   double volume = cube(side);

   cout << "The volume is : " << volume;

   return 0;
}
```

The parameter of the call are called the *actual parameters* (e.g. volume). The parameters of the function are called the *formal parameters* (e.g. s). It is important that the actual parameters match the formal parameters the formal parameters in type and number. The formal parameters and the actual parameters might have different names.

You must declare the function before it is called. For example, cube() must be before the function.

There is an exception to this, if you use **function prototyping**. This is where you first declare the function before it is used and then assign it later. In prototyping it is not necessary to put the parameter names. For example,

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

/*******************************************
Function cube calculates the volume of a cube.
Parameter side: The side of the cube.
*******************************************/

double cube(double); // prototyping

int main(){
   double side;
   cout << "Please enter the value of side: ";
   cin >> side;

   double volume = cube(side);

   cout << "The volume is : " << volume;

   return 0;
}

 double cube(double s){
   return s*s*s;
}
```

Another example of a function is

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


/*************************************
The function incr_by_val() increments
an integer v1 by another integer v2.
*************************************/

int incr_by_val(int v1, int v2){
   return v1 + v2;
}

int main(){
   int n = 4;
   int increment = 3;

   n = incr_by_val(n,2);
   cout << n << endl;
}
```

We can also make functions that only perform a task, which does not return a value. We can use **void** in the place of the return type. For example,

```
void printline(int length) {
   if (length >= 1) {
      for (int i=1; i<=length; i++) {
         cout << "-";
      }
   }
```

```
    else {
        return;
    }

    cout << "I am done with the function." << endl;
}
```

The following is a function that reads an integer between two values and keeps asking if it is not within that range.

```
int read_between(string what, int low, int high){
    int input;

    do{
        cout << "Please enter " << what << " between " << low << " and " << high << " :";
        cin >> input;
    } while(input < low || input > high);

    return input;
}
```

2.13.3 The scope of a variable

The scope of a variable is the portion fo the program where the variable is defined and can be used. These variables are **local**. For example,

```
int main() {
    int var = 5;  // scope of var is the main
    printline(var);
}

void printline(int n) { // the scope of n is printline()
    int index;    // The scope of index is printline()
}
```

You can only access variables in their scope. This is why you can use the same name in different scopes and the values of the variables can be different.

A variable declared in a block is visible in all the block, unless a var with the same name is declared in a n inner block.

```
int main() {
    int n = 25;
    if (choice == 'D') {
        int n = 0;
        cout << n; // This prints 0
    }
    cout << n; // This prints 25
}
```

2.13.4 Global Variables

October 22, 2021

Global variables are defined *outside of the main and all other functions in the program.*

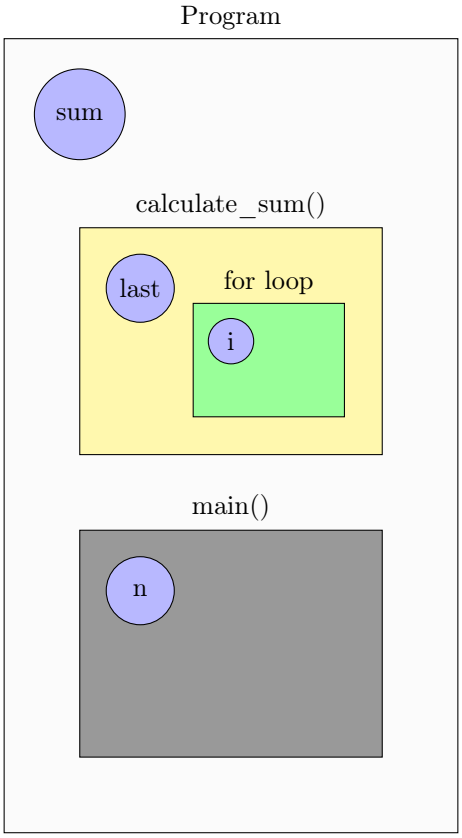
Their scope extends to all functions **defined after**. That is to say, they can be accessed and modified (read/write) from all functions (including the main) defined after.

Global variables are not recommended and are regarded as bad programming practice.

For example, the following program uses a global variable (passing by value) to calculate a Gauss sum.

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

int sum = 0;  // Global variable

void calculate_sum(int last) {
   for (int i=0;i<=last;i++) {
      sum += i;
   }
}

int main() {
   calculate_sum(5);

   cout << sum;
}
```

The following is the **scope diagram** for the variables in the above program.



If the above program were written using local variables (passing by value) instead, one could write:

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

int calculate_sum(int last){
   int result = 0;    // local variable

   for(int i = 0; i <= last; i++) {
      result += i;
   }

   return result;
}

int main(){
   int sum;   // local variable

   sum = calculate_sum(5);

   cout << sum;
}
```

2.13.5 Reference parameters

The parameters we've used so far are called **value parameters**. When a function is called, the **value** of the actual parameter **is copied** into the formal parameter. In that case, we say that parameters are **passed by value**.

When we passing by value, formal and actual parameters are **physically different** and only values can be passed. That is to say, both the formal and actual parameters have different locations in memory. Therefore, *changes to the formal parameter in the function will not affect the actual parameter*. For example,

```
void foo(int last){
   last = 2;
}

int main(){
   int n = 5;

   foo(n);

   // n is still 5, it has not been changed by foo()
}
```

An alternative solution is to pass parameters **by reference**. These are called **reference parameters**. In this case, the formal parameter (in the function header) *is not a local variable* within the function.

The function will instead have access to the *actual parameter* within the main. The function can use the value of the actual parameter from the main (a local variable to main) and change it directly.

To pass a parameter by reference in C++, we use the & symbol. For example,

```
void foo(int &last){
   last = 2;
}
```

```
int main(){
   int n = 5;

   foo(n);

   // n is now 2, it has been changed by foo()
}
```

Any change to last will affect n because of the use of the & character. We would say that the variable last is a **pointer** for the variable n. There is *no new variable physically* (in memory), last is simply another name for n.

When passing by value, a function can only calculate one value and return to the main. However, all parameters passed by reference can be changed by the function within the main. Therefore, *if we want to return multiple values*, we can use a function of type void and pass parameters by reference if we need to calculate multiple values by a function.

We will now repeat the local and global variable (passing by value) example, now adjusted to pass by parameter.

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

void calculate_sum(int &result, int last) {
   for(int i = 0; i <= last; i++) {
      result += i;
   }
}

int main(){
   int sum = 0;

   calculate_sum(sum, 5);

   cout << sum;
}
```

Note that the parameter result is passed by reference, but the parameter last is passed by value.

Note also that we can not pass a constant by reference, since it has no location in memory.

2.14 Arrays

An array is a **structure** for storing **multiple values** of the **same type**.

For example, suppose we are reading the numbers 5, 10, 3, 12, and 1.

We could find the largest of the values and print the result and it's index in the list (here, 12 fourth position). This is much more convenient for large data, when, for instance, reading each input in a different integer variable (a=5, b=10, etc.).

Instead, we could store all the values in one structure that is an array. The type of the array is int and the size is 5. All elemetns in the array are associateed with an incremental index that starts from zero.

index	0	1	2	3	4
element	5	10	3	12	1

2.14.1 Defining arrays

We can define an array of type int with size 34 as follows:

```
int marks[34]; // define but not initialized
```

Often it is useful to create a constant variable for the size of the array.

```
int SIZE = 34
int marks[SIZE]; // define but not initialized
```

We can also initialize arrays.

```
int numbers[5] = {6, 3, -2, 0, 10}; // defined and initialized
```

There is no need to put the size of the array if we initialize it.

```
int numbers[] = {6, 3, -2, 0, 10}; // defined and initialized
```

If we give a size that is different from what is initialized. If the given size is greater than the size of the array passed to be initialized, it will fill the rest with zeros. Otherwise, if the given size is less than the size of the array passed to be initialized, it will give an error.

```
int special[5] = {6, 3};         // special = [6, 3, 0, 0, 0]
int error[2] = {6, 3, -2, 0, 10};  // This results in a compile error
```

If we want an array of size 5 to be initialized with all 0s, we can do

```
int grades[5] = {};
```

You can also declare an array of other types.

```
string names[34];                    // names = ["", "", ...]
char vowels[] = {'a', 'e', 'i', 'o', 'u'};
```

2.14.2 Accessing array elements

We can access arrays using square brackets as follows. We can read/write as with regular variables.

```
int A[5];

A[3] = 25;            \\ set value at index 3 to 25
A[4] = A[3] + 1;      \\ set value at index 4 to the value at index 3 plus one
```

If we want to print the contents of an array, we must iterate through each of the elements and print them individually.

```
for (int i=0;i<5;i++){
   cout << A[i] << endl;
}
```

Since we have not initialized A[0], A[1], and A[2], they contain **garbage** (arbitrary, meaningless values).

The following is an example code to enter a sequence of grades using an array and calculate various properties of the set.

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

int main(){

   int Grades[5] = {};  // initialized with zeros
   int grade;
   int size = 0;


   // Get grades
   for(int i=0;i<5;i++){

      cout << "Please enter a grade or -1 to quit: ";
      cin >> grade;

      if(grade == -1){
         break;     // exits for loop
      } else {
         Grades[i] = grade;
         size++;
      }
   }


   // Calculate average of grades
   int sum = 0;

   for(int i=0; i<size; i++) {
      sum = sum + Grades[i];
   }

   double average = sum/(double)size;

   cout << "Average is: " << average;


   /// Find the maximum and minimum grades

   int max = Grades[0];
   int min = Grades[0];

   for(int i=1; i<size; i++) {
      if(Grades[i] > max) {
         max = Grades[i];
      }
      if(Grades[i] < min) {
         min = Grades[i];
      }
   }

   cout << "Max is: " << Max << endl;
   cout << "Min is: " << Min << endl;

   return 0;
}
```

2.15 Implementation of Algorithms

2.15.1 The Sequential Search Algorithm

November 2nd, 2021

Given a dataset of a given size N and given a target, is the target in the dataset?

For example, if the dataset is the list [13, 4, 5, -20, 45, 112] with $N = 6$. Is the target 45 in the dataset? Yes. Is the target 130 in the dataset? No.

To solve for much larger values of N, we can search sequentially through the list until we reach the target (the target is in the list), or the end of the list (the target is not in the list).

Let us assume we have a list of size N whose elements are L_1, L_2, ..., L_N and a target element called Target.

Algorithm 1
Sequential Search Algorithm

1: **get** L_1, L_2, ..., L_N, N, Target
2: **set** Found = No
3: **set** $i = 1$
4: **while** Found = No AND $i <= N$ **do**
5: **if** (L_i == Target) **then**
6: **set** Found = Yes
7: **else**
8: **set** $i = i + 1$
9: **if** Found = Yes **then**
10: **print** "Target in list."
11: **else**
12: **print** "Target not in list."
13: **stop**

We can implement the sequential search algorithm in C++ with the following code.

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

int main(){

   // Declarations
   const int N = 10;
   int data[N];    // data is L in the algorithm
   int target;

   // Read data
   cout << "Please enter " << N << " elements: ";
   for (int i=0; i<N; i++)
      cin >> data[i];

   cout << "Please enter a target: ";
   cin >> target;

   // Search
   bool found = false;
   int index = 0;
   while(!found && index<N) {
      if (data[index] == target)
         found = true;
      else
```

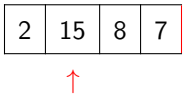
```
        index++;
    }

    if(found)
       cout << "Target is in the list at index " << index << endl;
    else
       cout << "Target is not in the list" << endl;

    return 0;
}
```

2.15.2 The Selection Sort Algorithm

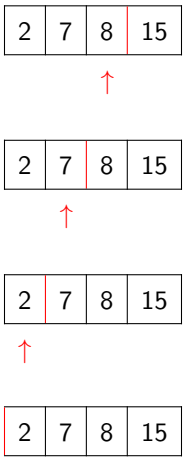
Given a list of n elements, we need to sort the list from smallest to largest.

For example, if the ($n = 5$) list = 5, 7, 2, 8, 3, then the sorted list should = 2, 3, 5, 7, 8.

Let's say we have the $n = 4$ list



We scan the list and find the largest (denoted by the red arrow). We also start with a marker (red line) equal to the length of the list. Then, we swap the element at that index with the last element and decrement the marker. Then, we repeat on all elements before the marker.



Now, the list is sorted. We can write the algorithm.

Algorithm 2
Selection Sort

1: **get** $n, L_1, ..., L_n$
2: **set** Marker = n
3: **while** (Marker $>$ 1) **do**
4: **set** largest = FindLargest($L_1, ..., L_{\text{Marker}}$)
5: Swap(largest, L_{Marker})
6: **set** Marker = Marker - 1
7: **stop**

We can implement the selection sort algorithm in C++ with the following code.

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

int main(){

   // Declarations
   const int N = 10;
   int data[N];     // data is L in the algorithm

   // Read data
   cout << "Please enter " << N << " elements: ";
   for (int i=0; i<N; i++)
      cin >> data[i];

   // Sort
   for(int marker = N-1; marker>0; marker--){

      // find index with the largest value
      int largest = 0;
      for (int i=1; i<=marker;i++) {
         if(data[i] > data[largest]) {
            largest = i;
         }
      }

      // swap L_largest and L_marker
      int temp = data[largest];
      data[largest] = data[marker];
      data[marker] = temp;

   }

   // Return list
   for (int i = 0; i<N; i++)
      cout << data[i] << " ";

   return 0;
}
```

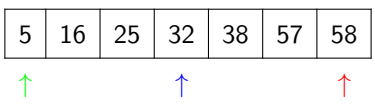
2.15.3 The Binary Search Algorithm

The objective of this algorithm is to search for an item in a list.

The *sequential search* is one way to do this. Recall that it has a complexity of $\Theta(n)$. *Can we make the search faster?*

The answer is **yes, but the list must be sorted**. If the list is sorted, then we can use the **binary search** (also called the *half interval search*).

For example, we will use the following list with $n = 7$.

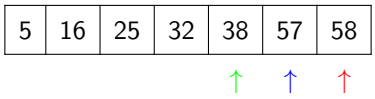


And let our be target = 57.

We will define Start = 1 and End = 7 and we will also define Mid = $\frac{\text{Start+End}}{2} = \frac{1+7}{2} = 4$. The value at the index Start is denoted by the green colored arrow, End by the red colored arrow, and

Mid by the blue colored arrow.

We then get the value at mid and check if it is less than target. It is, therefore, we set start = mid + 1 = 5. And we recalculate mid = $\frac{\text{Start+End}}{2} = \frac{5+7}{2} = 6$. Our new list is



Now, we get the value at mid which is 57. We check if it equals our target and it does, therefore, we say that our target is found (and at the index mid). We stop the algorithm there.

For this particular setup, it took 2 iterations. This is much faster than the 6 steps that it would have taken to do with sequential search.

Does this mean that the binary search is better than sequential search? Well, for finding a target it is. However, there is some time that is taken when initially sorting the list in the first place. If fast storage of data is required, perhaps the binary search is not the best option.

We will build now the algorithm.

Algorithm 3
Binary Search

1: **get** $n, L_1, ..., L_n$
2: **get** Target
3: **set** Found = No
4: **set** Start, End = 1, N
5: **while** (Found = No AND Start $\neq$ End) **do**
6: **set** Mid = $\frac{\text{Start+End}}{2}$
7: **if** (L_{Mid} = Target) **then**
8: **set** Found = Yes
9: **else**
10: **if** (Target < L_{Mid}) **then**
11: **set** End = Mid - 1
12: **else**
13: **set** Start = Mid + 1
14: **if** (Found = Yes) **then**
15: **print** "Target found"
16: **else**
17: **print** "Target not found"
18: **stop**

We can implement the binary search algorithm in C++ with the following code.

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

int main(){

   // Declarations
   const int N = 10;
   int data[N];    // data is L in the algorithm
   int target;

   // Read data
   cout << "Please enter " << N << " sorted elements: ";
   for (int i=0; i<N; i++)
   cin >> data[i];

   cout << "Please enter a target: ";
```

```
   cin >> target;

   // Search
   bool found = false;
   int start = 0;
   int last = N - 1;     // last is End in the algorithm
   int mid;

   while (!found && start < last){
      mid = (start + last)/2;
      if (data[mid] == target)
         found = true;
      else
         if(target < data[mid])
            last = mid - 1;
         else
            start = mid + 1;
   }

   if(found)
      cout << "Target is in the list at index " << mid << endl;
   else
      cout << "Target is not in the list" << endl;

   return 0;
}
```

2.16 Arrays and Functions

November 10th, 2021

Sometimes it is useful to use arrays in functions.

- An array can be passed as a parameter
- Arrays are passed by reference (their content can be changed by function)
- Formal parameter: no & (reserved for scalar values), empty [] instead
- Actual parameter: array name without []
- Cannot return an array (only a single value)

An example of using arrays in a function is

```
int sum(int values[], int size) {
   int total = 0;
   for (int i=0; i<size; i++)
      total += values[i]
   return total
}
```

A **bad example** (one that will give an error) is the following, if you try to return an array:

```
int[] getinfo(){
   cout << "How many: ";
   int size;
   cin >> size;
   int data[size];
   for (int i=0; i< size; i++)
      cin >> data[i];
   return data
}
```

Instead, you must pass by parameter:

```
void getinfo(int data[], int &size) {
   cout << "How many: ";
   cin >> size;
   for (int i=0; i<size; i++)
      cin >> data[i];
}
```

The following code is an example of various functions that involve arrays in the parameters or return.

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

int readarray(int data[]){
   int len;
   cout << "What is the length of the array (less than 100): ";
   cin >> len;

   for(int i=0; i< len; i++)
      cin >> data[i];

   return len;
}

void multiplyarray(int data[], int len, int factor){
   for(int i=0; i< len; i++)
      data[i] = data[i] * 2;
}

void printarray(int data[], int len){
   for(int i=0; i< len; i++)
      cout << data[i] << endl;
}

int main(){
   const int MAXSIZE = 100;

   int data[MAXSIZE];
   int len;

   len = readarray(data);
   multiplyarray(data, len, 2);
   printarray(data, len);
}
```

2.16.1 2D Arrays

We often need to use arrays within arrays. These are called **2D arrays**. An example of a 1D array of size 7 is

5	16	25	32	38	57	58

Whereas a 2D array with 3 rows and 7 columns could be

70	5	53	62	3	91	13
12	7	82	7	9	19	40
2	98	21	4	84	75	25

In C++ code, we can declare a 2D array with

```
const int maxrows = 4;
const int maxcols = 3;

int counts[maxrows][maxcols];    // declare but not initialize
```

We can also initialize 2D arrays as follows:

```
int grades[3][2] = {{85,69},{77,98},{87,83}};  // declare and initialize
```

Similar to the 1D array, when initializing, you can leave out the first size.

```
int marks[][2] = {{85,69},{77,98},{87,83}};
```

We will do an example problem where we need to store and count the number of medals (gold, silver, and bronze) won by 4 countries. For instance, we could have the case where the countries' medal count can be represented in the table

	gold	silver	bronze
country 1	5	2	6
country 2	3	9	8
country 3	2	3	1
country 4	2	0	7

The full code is

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

string country(int label) {
   switch(label) {
      case 0: return "Canada";
      case 1: return "USA";
      case 2: return "China";
      case 3: return "France";
      default: return "";
   }
}

string medal(int label) {
   switch(label) {
      case 0: return "Gold";
      case 1: return "Silver";
      case 2: return "Bronze";
      default: return "";
   }
}
```

```
int main() {
   // declare values and constants
   const int COUNTRIES = 4;
   const int MEDALS = 3;
   int counts[COUNTRIES][MEDALS];

   // fill array and calculate the sums of rows
   for(int i=0; i<COUNTRIES; i++) {
      cout << "Enter the medals counts for " << country(i) << ": ";
      for(int j=0;j<MEDALS;j++)
         cin >> counts[i][j];
   }
   cout << endl;

   // calculate the sums of rows
   for(int i=0; i< COUNTRIES; i++){
      int sum = 0;
      for(int j=0; j<MEDALS; j++)
         sum += counts[i][j];
      cout << "The total number of medals won by " << country(i) << " is " << sum <<
          endl;
   }
   cout << endl;

   // calculate the sums of columns
   for(int j=0; j< MEDALS; j++){
      int sum = 0;
      for(int i=0; i<COUNTRIES; i++)
         sum += counts[i][j];
      cout << "The total number of " << medal(j) << " medals is " << sum << endl;
   }
   cout << endl;

   /// display array
   for(int i=0; i<COUNTRIES;i++){
      for(int j=0;j<MEDALS;j++)
         cout << counts[i][j] << " ";
      cout << endl;
   }
   cout << endl;
}
```

You could also construct a function fillarray() which fills the array from a function instead of from the main. However, the number of rows and number of columns constants must be made globally so that the fillarray can access it.

```
const int COUNTRIES = 4;
const int MEDALS = 3;

void fillarray(int counts[][MEDALS]){
   for(int i=0; i<COUNTRIES; i++) {
      cout << "Enter the medals counts for " << country(i) << ": ";
      for(int j=0;j<MEDALS;j++)
         cin >> counts[i][j];
   }
   cout << endl;
}

int main() {
   int counts[COUNTRIES][MEDALS];
```

```
    fillarray(counts);

    ...
}
```

2.17 Vectors

Arrays are not the only data structure that can store multiple values, we can also use **vectors**.

- A vector collects a sequence (1D, 2D, etc.) of same type values just like an array does
- Unlike arrays, the size of a vector can change
- A vector is a higher abstraction of array: Array with a lot of useful built-in functions.

To use vectors in C++, we need to include the package.

```
#include <vector>
```

To declare a vector (let's say one that contains doubles), we can write

```
vector <double> marks;  // initial size is 0
```

We can declare an initial size with following parentheses

```
vector <double Marks(10);  // initial size is 10
```

If we want to get the **current size** of a vector, we can use

```
int size = Marks.size();
```

To fill a vector with inputted values, we can use

```
for(int=0; i < Marks.size(); i++)
    cin >> Marks[i];
```

The vector has a few more conveniences:

- Vectors can be used in functions like any other variable
- They can be passed by reference (with the &) or by value
- They can be returned by a function

To see the complete list of builtin functions for the vectors, visit the reference documentation at https://www.cplusplus.com/reference/vector/vector/.

As an example, let us make a vector that contains students marks. First, we can employ the following strategy if the size is known.

```
// declare
vector <double> marks(10);

// fill
for(int=0; i < marks.size(); i++)
    cin >> marks[i];

// display
for(int=0; i < marks.size(); i++)
    cout << marks[i] << endl;
```

However, if the size is unknown, we can instead do the following to fill the vector.

```
// The user enters values and stops when q is entered
double grade;
while (cin >> grade) { // 1. A value is read 2. returns true/false based on type of
    value
   marks.push_back(grade);  // increases the size by one and put grade in that location
}
cin.clear();
cin.ignore();
```

This shows you how to increase the size of a vector, but how do you *decrease* the size of a vector? We can pop the last value off of the vector and decrement the size by one. This is done with

```
marks.pop_back(); // removes last cell of the vector
```

To get the first and last elements of a vector, we can use the following methods

```
// First grade
cout << marks[0] << endl;
cout << marks.front() << endl;

// Last grade
cout << marks[marks.size() - 1] << endl;
cout << marks.back() << endl;
```

To calculate the average grade,

```
double total = 0;
double average;
for (int i=0; i < marks.size; i++)
   total += marks[i];

if (marks.size() != 0)
   average = total / marks.size();
else {
   cout << "No data. Program will stop";
   return -1;
}

cout << "Average is: " << average << endl;
```

To find and remove the smallest grade,

```
// find position of smallest
int smallpos = 0;
for(int i=0; i < marks.size(); i++) {
   if (marks[i] < marks[smallpos]) {
      smallpos = i;
   }
}

// copy back of vector to smallpos and remove duplicated back value
marks[smallpos] = marks.back();
marks.pop_back();
```

We can make the reading of a vector to be a function using the following

```
vector<double> vect_read() {
   vector <double> v;
```

```
   double element;
   while(cin >> element)
      v.push_back(element);
   cin.clear();
   cin.ignore();

   return v;
}
```

We can also make the finding of the average of the elements of a vector using

```
vector<double> vect_avg(vector<double> v) {
   double total = 0;

   for(int i=0; i< v.size();i++){
      total += v[i];
   }

   if (v.size()!=0)
      return total / v.size();

   return -1; // Error, no data
}
```

Finally, we can finding the smallest element of a vector with yet another function if we want:

```
int vect_minpos(vector<double> v) {
   int index = 0;
   for (int i=0; i<v.size();i++) {
      if (v[i] < v[index])
         index = i;
   }
   return index;
}
```

We can make a remove function if we would like also, to remove the element of a particular position

```
void vect_remove(vector<double> &results, int index) {
   results[index] = results.back();
   results.pop_back();
}
```

To append a vector (source) to another one (destination), we can use the function

```
void append(vector<int> src, vector<int> &dest) {
   for (int i=0; i<src.size(); i++) {
      dest.push_back(src[i]);
   }
}
```

We can print a vector with something like

```
void print(vector<int> v) {
   for (int i=0; i<v; i++) {
      cout << v[i] << " ";
   }
   cout << endl;
}
```

Something to note about arrays is that you cannot simply assign one to another, you must assign each element to another. Namely,

```
int A[2] = {1, 6};
int B[2];

B = A; // Gives Error

for (int i=0; i<2; i++) // Must iterate over all elements instead
   B[i] = A[i];
```

Whereas for vectors, you can directly assign one to the another without error.

```
vector <int> A(2);
A[0] = 1;
A[1] = 6;
vector <int> B;

B = A;  // This works now

for (int i=0; i<A.size(); i++) // This still also works
   B.push_back(A[i]);
```

2.18 Pointers

2.18.1 The reference operator: &

A variable is stored in the memory and a variable is associated with a *memory address* which points to a particular cell in memory.

If we have a variable, we can find the address of the number using the ampersand sign (&). For instance,

```
int number = 25;

cout << "number = " << number << endl;
cout << "Address of number = " << &number << endl;
```

This will give us a result in hexadecimal such as 0x6dfefc.

If we also pass it through a function **by value** as follows

```
int number = 25;

cout << "number = " << number << endl;
cout << "Address of number = " << &number << endl << endl;

display(number);
```

where display is defined as

```
void display(int number) {
   cout << "In a function:" << endl;
   cout << "number = " << number << endl;
   cout << "Address of number = " << &number << endl;
}
```

we will get a *different memory address*, but the same value for the variable (e.g. 0x6dfefc and 0x6dfee0). We can clearly see the locality of the variables this way.

If instead we pass number **by reference** to display with

```
void display(int &number) {
   cout << "In a function:" << endl;
   cout << "number = " << number << endl;
   cout << "Address of number = " << &number << endl;
}
```

Then we can see that they both have the *same address* (e.g. 0x6dfefc and 0x6dfefc).

The ampersand sign (&) is called the **reference operator** and it allows access to values' addresses.

2.18.2 Definition of Pointers

- A variable contains a value: int number = 25;
- The address of a variable is acessed by &: Address of number is &number
- A pointer is a special type of variable that contains an address.
- For example:

  ```
  int *p;
  ```

- p is a pointer to a variable that contains an int
- p will contain the address of a variable of type int.
- * is called the **dereference operator** (*p: value pointed by p)

We can define a pointer called p using

```
int *p;
```

If we want to initialize the pointer, we need to define another variable (say number) and then define it to be the address of that variable.

```
int number = 25; // initialize another variable
int *p;          // declare pointer
p = &number;     // assign pointer the address of another variable
```

If we write

```
*p = 35;
```

We take the number at the address pointed to by p to be 35. Thus, number becomes equal to 35.

If we declare new_number, if we want to set p to point to new_number instead of number, we must not use the *

```
int new_number;
p = &new_number;
```

Now p points to new_number and not to number anymore. So if we assign the value at p to be 40 with

```
*p = 40;
```

then new_number becomes 40, but number is still 35.

We will now do a few examples of this.

Example 1:

```
// Declaration and Assignment of Pointers
int number = 25;

cout << "number = " << number << endl;      // 25
cout << "address number = " << &number << endl;  // 0x6dfef8

int *p;
p = &number

cout << "p = " << p << endl;   // 0x6dfef8
cout << "*p = " << *p << endl;   // 25

double new_number = 56.0;
p = &new_number;   // Error, p must point to an int, new_number must instead be an int
```

Example 2:

```
// Manipulation with the pointers

int first = 5;
int sec = 15;

int *p1, *p2;
p1 = &first;
p2 = &sec;
// first = 5, second = 15, *p1 = 5, *p2 = 15

*p1 = 10;
// first = 10, second = 15, *p1 = 10, *p2 = 15

*p2 = *p1;
// first = 10, second = 10, *p1 = 10, *p2 = 10

p1 = p2;
// first = 10, second = 10, *p1 = 10, *p2 = 10
// p2 now points to first

*p1 = 20;
// first = 10, second = 20, *p1 = 20, *p2 = 20

cout << first << " " << sec << " " << *p1 << " " << *p2;
```

Example 3:

```
// Pointers initialization errors

int *p1;
*p1 = 25;     // fatal error
// This is because p1 does not point anywhere, and you cannot assign 25 to nowhere
```

```
// Must do the following instead
int number;
int *p1 = &number;
int *p1 = 25;
```

2.18.3 Functions with Pointer Arguments

Using pointers, we can use a different style for the following passing by reference example

```
void deposit(double &bal, double amt) {
   bal = bal + amt;
}

int main() {
   double balance = 1000;
   deposit(balance, 25);
   cout << "The new balance is " << balance;
}
```

We can rewrite this using a pointer as follows.

```
void deposit2(double *bal, double amt) {
   *bal = *bal + amt;
}

int main() {
   double balance = 1000;
   deposit2(&balance, 25);
   cout << "The new balance is " << balance;
}
```

2.18.4 Arrays with Pointers

Let's say we have an array

```
int a[5] = [6, 2, 4, 23, 0];
```

What is a? When we print

```
cout << a << endl;
```

We find that a contains an address (e.g. 0x6dfeec). Therefore, a is a pointer. Specifically, it is a pointer to the first value of the array. We can see this by noticing that

```
cout << &a[0] << endl;
```

gives the same value as when we printed just a. If we want, we can make an explicit pointer that points to the first value ourselves.

```
int *p = a;
```

Note that

```
cout << *p << endl;
cout << *a << endl;
```

both return the same address as before (of the first element of a).

We can iterate over the elements of a in a few different ways. We will demonstrate this my printing the array. Firstly,

```
for(int i=0;i<5;i++)
   cout << p[i] << endl;
```

Secondly,

```
for(int i=0;i<5;i++)
   cout << *(p+i) << endl;
```

Thirdly,

```
for(int i=0;i<5;i++)
   cout << *p << endl;
   p++;
```

Note that this third solution will no longer preserve the value of p.

We will now see a few examples of using arrays with pointers.For each of the examples we will assume an array a defined as

```
int a[] = {2, 3, 5};
```

Example 1:

```
int *p = a+1;      // p --> a[1]
*(p+1) = 0;        // p+1 --> a[2]
// a is now [2, 3, 0]
```

Example 2:

```
int *p = a;        // p --> a[0]
p++;             // p --> a[1]
*p = 0;
// a is now [2, 0, 5]
```

Example 3:

```
int *p = a;        // p --> a[0]
int *q = a+2;      // q --> a[2]
p++;             // p --> a[1]
q--;             // q --> a[1]
*p = *q;
// a is still [2, 3, 5]
```

Example 4:

```
cout << *a + 2 << endl;  // Returns a[0] + 2 = 4
cout << *(a+2) << endl;  // Returns a[0+2] = 5
```

2.18.5 Functions Returning Pointers

We can get around not being able to return an array from a function by instead returning a pointer. For example,

```
int * firstlast(int A[], int n){
   int result[2];

   result[0] = A[0];
   result[1] = A[n-1];

   return result;
}

int main() {
   int A[] = {5, 2, 6, 4, 7};
   int *B;

   B = firstlast(A, 5);

   cout << B[0] << " , " << B[1];
}
```

Will this work? No. The program compiles, but it does crashes on run. This error occurs because the local pointer variable result is lost when the function is exited. Instead, we must pass B by reference also.

```
void firstlast2(int A[], int B[], int n){
   B[0] = A[0];
   B[1] = A[n-1];
}

int main() {
   int A[] = {5, 2, 6, 4, 7};
   int B[2];

   firstlast2(A, B, 5);

   cout << B[0] << " , " << B[1];
}
```

Therefore, it is possible to return pointers, however, we cannot use that to return an array.

2.18.6 Dynamic Memory Allocation

We will consider the following array

```
int A[10];

for (int=0;i<10;i++)
   A[i] = 2*i;

int *p = A;
```

This is called **static array allocation**. When we declare the array A, it allocates memory space to the array (size 5) until the end of the program. However, if A is no longer needed, its memory space cannot be deallocated for other programs to use. Memory space for all variables declared statically remains allocated.

To make things more memory efficient, we can use dynamic memory allocation. In this case, the memory space of an array is allocated at run-time. The memory space can be deallocated or freed from the program once the array is no longer needed. Deallocation takes place before the program stops. Once deallocated the array can no longer be accessed. For example,

```
int n;

cout << "What is the size of the array? ";
cin >> n;

// Allocation
int *A = new int[n]; // A is a pointer to a dynamic array A

for (int i=0; i<n; i++)
   A[i] = n-i;

// Deallocation
delete[] A;

// A now contains garbage values
```

We can use dynamic memory allocation to return an array from a function.

```
int *firstlast3(int A[], int n){
   int *result = new int[2]; // dynamic allocation

   result[0] = A[0];
   result[1] = A[n-1];

   return result;
}

int main() {
   int A[] = {5, 2, 6, 4, 7};
   int *B;

   B = firstlast(A, 5);

   cout << B[0] << " , " << B[1];
}
```

This works. Therefore, you can return an array from an array from a function by returning a pointer to a dynamically allocated array.

2.19 Structures

2.19.1 Definition

When we have variables that require more than one variable type, it is useful to use a structure. A structure is used to group items of arbiturary types in a single construct.

The syntax for a structure is

```
struct Address (
   int number;
   string StName;
);
```

We can now declare a variable using Address as a variable type.

```
Address my_addr;
```

To initialize values in it we can do

```
my_addr.number = 2600;
my_addr.StName = "College";
```

We can use the structure as normal

```
cout << my_addr.number << " " << my_addr.StName << endl;
```

2.19.2 Nested Structures

We can define a structure within a structure. This is called a nested structure.

```
struct Exams (
   double mid;
   double fin;
);

struct Student (
   string name;
   Address StAddress;
   Exams grades;
);
```

Student is a nested structure. We can declare it

```
Student aStudent;
```

And initialize

```
aStudent.name = "Jaeden Bardati";
aStudent.StAddress.number = 2600;
aStudent.StAddress.StName = "College";
aStudent.grades.mid = 87.5;
aStudent.grades.fin = 92.1;
```

Once again, we can print them as usual

```
cout << "Final grade of " << aStudent.name << " is " << aStudent.grades.fin << endl;
```

2.19.3 Arrays of Structures

We can also declare arrays of structures. We treat this new Student structure as a variable type.

```
Student class211[32];

for (int i=0; i<32; i++) {
   cout << "Please enter info for student " << i << endl;
   cin >> class211[i].name;
   cin >> class211[i].StAddress.number;
   cin >> class211[i].StAddress.StName;
   cin >> class211[i].grades.mid;
   cin >> class211[i].grades.fin;
}
```

2.19.4 Vectors of Structures

Furthermore, we can create vectors of structures. This is particularly useful if there are varying numbers of this structure. In our example, this could mean that a student drops the class. We declare as usual:

```
vector<Student> class211;
```

However, in order to push to the vector of Students, we need to first construct a student structure for all desired students and then push to the vector.

```
Student aStudent;
bool more = true;
while(more) {
   cin >> aStudent.name;
   cin >> aStudent.StAddress.number;
   cin >> aStudent.StAddress.StName;
   cin >> aStudent.grades.mid;
   cin >> aStudent.grades.fin;

   class211.push_back(aStudent);

   cout << "Would you like to continue? (0/1) ";
   cin >> more;
}
```

2.19.5 Pointers of Structures

We can use pointers for structures also. We saw an example of this with the array earlier. However, we can declare a pointer explicitly.

```
Student *StudentPtr;
StudentPtr = &aStudent;

cout << "Final grade of " << (*StudentPtr).name << " is " << (*StudentPtr).grades.fin;
```

We can also use an arrow notation to achieve the same effect as above.

```
cout << "Final grade of " << StudentPtr->.name << " is " << StudentPtr->.grades.fin;
```

2.19.6 Structures with Pointer Members

Some members of the structures could also be pointers. Perhaps even pointers to other structures. This is useful when information is shared among structure values.
Let us create a new structure for the department that the student is part of.

```
struct Dept (
   string id; // CS, MATH, PHYS, etc.
   string chair; // chairperson of the department
);
```

And we will adjust the student structure to account for this

```
struct Student (
   string name;
   Address StAddress;
   Exams grades;
   Dept StdDep;
);
```

Use can now initialize two students

```
Student S1, S2;

S1.name = "Bob";
S1.StAddress.number = 2600;
S1.StAddress.StName = "College";
S1.grades.mid = 87.5;
S1.grades.fin = 92.1;
S1.StdDep.id = "CS";
S1.StdDep.chair = "Jane Doe";

S2.name = "Alice";              // Not shared
S2.StAddress.number = 245;      // Not shared
S2.StAddress.StName = "Queen";  // Not shared
S2.grades.mid = 80.7;           // Not shared
S2.grades.fin = 95.3;           // Not shared
S2.StdDep.id = "CS";            // Shared
S2.StdDep.chair = "Jane Doe";   // Shared
```

We have a couple values that will be shared for all our students. Is there a way of making this more efficient? Yes, we can use pointers. We can declare the following structures for the math and computer science departments.

```
Dept CSDept, MathDept;

CSDept.id = "CS"
CSDept.chair = "Jane Doe";

MathDept.id = "Math";
MathDept.chair = "John Doe";
```

Now, instead of the StdDep member of the Student structure being a value, we can declare it as a pointer.

```
struct Student (
   string name;
   Address StAddress;
   Exams grades;
   Dept* StdDep;
);
```

Now, we can use point to the particular department that each student is part of instead of re-declaring the department properties every time.

```
Student S1, S2;

S1.name = "Bob";
S1.StAddress.number = 2600;
S1.StAddress.StName = "College";
S1.grades.mid = 87.5;
S1.grades.fin = 92.1;
S1.StdDep = &CSDept;

S2.name = "Alice";              // Not shared
S2.StAddress.number = 245;      // Not shared
S2.StAddress.StName = "Queen";  // Not shared
S2.grades.mid = 80.7;           // Not shared
S2.grades.fin = 95.3;           // Not shared
S2.StdDep = &CSDept;            // Shared
```

If the chair of the department changes, we simply need to change one value total, not one value for every student. It is similarly useful is a student changes program.

2.19.7 Dynamic Arrays of Structures

Earlier, we showed that is was possible to declare static arrays of structures. We can similarly declare dynamic arrays of structures also.

```
Student *class211 = new Student[32];

for (int i=0; i<32; i++) {
   cout << "Please enter info for student " << i << endl;
   cin >> class211[i].name;
   cin >> class211[i].StAddress.number;
   cin >> class211[i].StAddress.StName;
   cin >> class211[i].grades.mid;
   cin >> class211[i].grades.fin;
}
```

2.20 Object-Oriented Programming

Object-Oriented Programming (**OOP**) is a programming style in which tasks are solved by collaborating **objects**.

Each object has

- Its own set of data
- A set of functions that act upon the data

For example, a **string** is an object that has data as an array of characters that can be manipulated by functions such as substr(), length(), etc.

Another example is a **vector**, which is an object that contains a list of elements that can be manipulated by functions such as pop_back(), push_back(), etc.

A **programmer** implements a class from which objects can be defined. An **object** is an instantiation of a class. For example, with the following code

```
string myname;
vector<int> data(10);
```

The **classes** are *string* and *vector*, whereas the **objects** are *myname* and *data*.

The **user** defines objects from a class implemented by a programmer. The user does not need to know how the class is implemented. Such an implementation is called **private**. The user only needs a **public interface** that explains how the class functions can be used. This concept is called **Encapsulation**.

We will demonstrate OOP through an example: We want to design a class that simulates a *cash register*. A cash register should be able to carry out te following operations:

- Clear cash register for new sale
- Add price of an item
- Get total of items
- Get total price

The interface is defined in the **class definition**. The following is the code to do so.

```
class CashRegister {
   public:
      void clear();                 // mutator
      void add_item(double price);  // mutator
      double get_total() const;     // accessor
      double get_count() const;     // accessor
   private:
      int item_count;
      double total_price;
}
```

In the **public** interface, we have **member functions**: functions that can access and manipulate the object's data. Of these, we have **mutators**, which can change an object's data, and we have **accessors**, which can only read an object's data (denoted with *const* keyword). Member functions are invoked using the **dot operator**:

```
CashRegister register1, register2;
register1.clear();
register2.clear();
```

In the **private** section, we have **data members**: variables declared in the class. The variables declared under the private keyword can only be accessed via class functions.

```
register1.item_count;  // error (it is private)
register2.get_count();  // this returns item_count (it is public)
```

2.20.1 Example: Cash Register

We will now implement the cash register class outside of the main function.

```
// Class definition of Cash Register

class CashRegister {
   public:
      void clear();                 // mutator
      void add_item(double price);  // mutator
      double get_total() const;     // accessor
      double get_count() const;     // accessor
   private:
      int item_count;
      double total_price;
}

// Class function members implementation

void CashRegister::clear() {
   item_count = 0;
   total_price = 0;
}

void CashRegister::add_item(double price) {
   item_count++;
   total_price += price;
}

double CashRegister::get_total() const {
```

```
   return total_price;
}

double CashRegister::get_count() const {
   return item_count;
}
```

Now we can use the following code inside the main function to use these methods.

```
CashRegister reg;

reg.clear();
reg.add_item(10.23);
reg.add_item(5.26);

cout << "You purchased " << reg.get_count() << " items" << endl;
cout << "The total price is " << reg.get_total() << "$" << endl;
```

Let us say that we want to make a function to display information about the Cash Register class. We can pass classes as a parameter by value or reference. Here, we will pass by value.

```
void display_cash(CashRegister reg) {
   cout << "You purchased " << reg.get_count() << " items" << endl;
   cout << "The total price is " << reg.get_total() << "$" << endl;
}
```

Let us say now that we want to add multiple items (of the same price) in one function call. We can update the public interface as follows

```
class CashRegister {
   public:
      void clear();
      void add_item(double price);
      void add_items(int quantity, double price);
      double get_total() const;
      double get_count() const;
   private:
      int item_count;
      double total_price;
}
```

And the function implementation is

```
void CashRegister::add_items(int quantity, double price){
   for(int i=0;i<quantity; i++)
      add_item(price);
}
```

Note that we can call class functions within another class function, and when we do, we do not use the dot operator notation to do so. We simply call the function as normal.

2.20.2 The Class Constructor

We will now introduce a new class function type. We can already seen the *mutator*, the *accessor*, and now we will see the **constructor**.

Notice that in the cash register example, we always needed to clear the data after instantiating the class. Often, we need to initialize class data after instantiating it. Can we do this in one step?

Yes, by using constructors.

What is a constructor? A **constructor** is a method or a function that will initialize your internal data upon declaration. In other words, it is not a function to be called, and is instead called automatically when declaring the class.

There is *no return type* for constructors and the name of the constructor must be the same as the name of the class. We declare it in the public section of the public interface. The **default constructor** is one where no parameters are passed. For example we can alter the public interface of the public interface for the cash register class as follows.

```
class CashRegister {
   public:
      CashRegister()  // Default constructor
      void clear();
      void add_item(double price);
      void add_items(int quantity, double price);
      double get_total() const;
      double get_count() const;
   private:
      int item_count;
      double total_price;
}
```

And the implementation of the default constructor is

```
CashRegister::CashRegister() {
   item_count = 0;
   total_price = 0;
}
```

We do not need to change anything to call the constructor, since it is called automatically when the object is declared.

```
CashRegister reg; // not need to clear anymore, constructor does so automatically

reg.add_item(10.23);
reg.add_item(5.26);

display(reg);
```

Let us say that we want to initialize the cash register with a balance (e.g. a coupon). We can make another constructor containing a parameter.

```
class CashRegister {
   public:
      CashRegister()  // Default constructor
      CashRegister(double coupon_value)
      void clear();
      void add_item(double price);
      void add_items(int quantity, double price);
      double get_total() const;
      double get_count() const;
   private:
      int item_count;
      double total_price;
}
```

And the implementation is

```
CashRegister::CashRegister(double coupon_value) {
   item_count = 0;
   total_price = coupon_value;
}
```

We can create another register to demonstrate a constructor with a parameter.

```
CashRegister reg;  // default contructor
CashRegister reg2(-10);  // constructor with a parameter

reg.add_item(10.23);
reg.add_item(5.26);

reg2.add_items(5, 1.0);

cout << "Register 1:" << endl;
display(reg);

cout << endl << "Register 2:" << endl;
display(reg2);
```

2.20.3 Dynamic Allocation of Objects

We can also create objects dynamically using the new keyword.

```
CashRegister *reg3 = new CashRegister; // dynamic allocation of a cash register object
```

Note that reg3 is a pointer. We can then use the object's methods using

```
(*reg3).add_items(2, 1.5);
display(*reg3);
```

We can also declare using the second constructor

```
CashRegister *reg4 = new CashRegister(-5); // dynamic allocation of a cash register
    object
```

If we want to delete the object, we can delete the pointer.

```
delete reg3;
```

We can also use the arrow notation as with structures:

```
reg4->additems(2, 1.5);
```