FINM 326: Computing for Finance in C++

Lecture 2

Chanaka Liyanarachchi

January 13, 2023

References, Pointers and Arrays

Control Structures

Currency Converter

References, Pointers and Arrays

Functions: Passing Arguments

- Let's look at the following example:

```
void IncrementByOne(int n)
{
   n = n + 1;
}


int main()
{
   int x = 5;

   IncrementByOne(x);

   cout << "x  =  " << x << endl;
}
```

- What is the output of this program?

- We want to answer 2 questions:
  1. Why `x` did not change?
  2. What changes do we need to make to change `x`?

- We can pass an argument to a function:
  - by value
  - by reference
- The `IncrementByOne()` *passes the argument (x) by value*:
  - a copy of the variable (`x`) is passed to the function
  - we're changing the copy; the original value remains unchanged
- When we want a function to use the same variable (object) inside the function, we *pass the argument by reference*.

References

- A *reference* is an alias (another name) for a variable (object).
- To use a reference, it has to be bound to another existing variable.
- Once a reference is initialized by binding to a variable, you cannot rebind it to a different variable.
- We use the & to declare a reference.

References: Example

- Let's look at some examples:
  `int x = 10;`
- `refx` is a reference to (i.e. another name for) `x`:
  `int& refx = x;`
- Now we can change `x` using refx:
  `refx = 20;`
- This line will not compile since it is not bound to another variable:
  `int& refy;` error

Pass by Reference: Example

```
void IncrementByOne(int& n)
{
   n = n+1;
}


int main()
{
   int x = 5;

   IncrementByOne(x);

   cout << "x = " << x << endl;
}
```

What's the output of this program now?

Const Reference

- We can bind a reference to a const value:

```
const int x = 1;
const int& refx = x;
```

- Trying to change the value is an error:

```
x = 10; error
refx = 10; error
```

- We can also bind a const reference to a non const variable:

```
int y = 2;
const int& refy = y;
```

- Now, we cannot change the variable using the reference:

```
y = 10;
refy = 10; error
```

- Suppose we have the following function.

```
int DotProduct(BigMatrix m1,
    BigMatrix m2)
{
    //dot product using m1 and m2

}
```

- Passing by value has an overhead due to copying:
  - time to copy m1 and m2
  - store multiple copies of m1 and m2 in memory
- Should we pass by reference?

- Suppose, we pass by reference:

```
int DotProduct(BigMatrix& m1,
      BigMatrix& m2)
{
    //dot product using m1 and m2
    //should never change m1 or m2
    //what if we change m1 or m2 by mistake?
}
```

- It allows the original values to be changed.
- Changing the original can be dangerous.
- Should not pass by reference to improve performance by avoiding copies.

Passing Arguments by Const Reference

- Solution is to pass by const-reference:

```
int DotProduct(const BigMatrix& m1,
       const BigMatrix& m2)
{
    //dot product using m1 and m2
    //cannot change m1 and m2
}
```

- Now, we cannot intentionally or accidentally change m1 or m2 in `DotProduct()`.
- We're using type safety to write "correct" code.
- Improves readability/clarity: using const reference clearly shows our intention for using a reference.

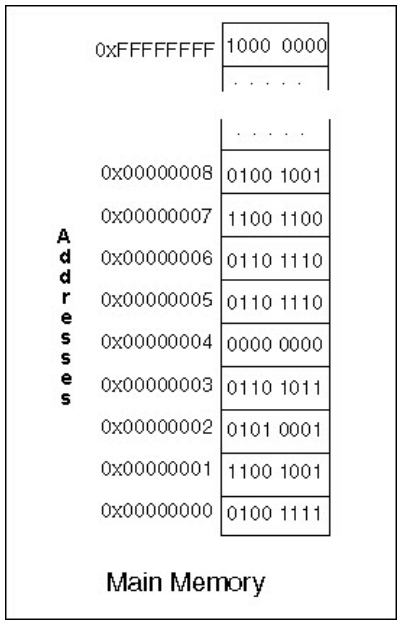
Quiz

1. What is a reference?
2. Why would you use a reference?[1]

[1] We will discuss a third reason to use references, when we discuss OOP later in the course.

Computer Memory

- Physical memory is arranged in blocks of bytes (8 bits).
- Simplified computer memory diagram shown below:[2]



- Programs use memory locations to store values (we store values at a location using 1s and 0s).
- Each memory location has an address.
- We can use the memory address to access a memory location/variable.

[2] Image source: Wikipedia

Pointers in C++

- A pointer is a variable used to store a memory address of a variable/object:
  - we say the pointer is *"pointing to the variable/object"*.
  - we can access the variable/object using the pointer.
- We use `*` to declare a pointer.
- E.g. we can declare a pointer to an integer variable as:

```
int* pi;
```

- We use use two important operators with pointers:
  - `&` : address-of operator
  - `*` : dereference operator
- We use the address-of operator to get the address of a variable/object.
- We use the dereference operator to access the variable/object, a pointer is *pointing-to*.

Pointers: Examples

- We have an integer variable `x` which is initialized to 10:
  ```
  int x = 10;
  ```
- We can use a pointer (`px`) to store the address of `x`:
  ```
  int* px = &x;
  ```
- We can access/read the variable (`x`) using the pointer:

  ```
  cout << *px;
  ```
- We can change the variable (`x`) using the pointer:
  ```
  *px = 20;
  ```
- Pointers are used to access variables/objects indirectly.

- The pointer above can point it to (store the address of) a different variable:

```
int x2 = 10;

px = &x2;
```

- We can have an uninitialized pointer:

```
int* px2; this is legal, but bad practice
```

- An uninitialized pointer can point to anything (garbage), we should not use uninitialized pointers.
- We use `nullptr` to indicate that the variable does not point to an object:

```
int* px2 = nullptr; good practice
```

Pointers and const

- You can have a pointer to a `const` variable/object.
- Then you cannot use the pointer to change the variable/object.

```
const int i = 10;
const int* ip = &i;
```

- You can also have a `const` pointer.
- Now, the pointer itself is `const`, but what is pointed to is not const.

```
int i = 10;
int* const  ip = &i;
```

Pointers and const: Examples

- Remember: pointer is also a variable; we use a pointer to store an address (of a variable).
- Case 1: non-const variable and non-const pointer to a non-const variable:

```
int x = 10;

int* ptr = &x;  //initialize

*ptr = 32;        //ok - variable is non-const

int y = 3;

ptr = &y;      //ok - pointer is non-const
```

- Case 2: const variable and non-const pointer to a const variable:

```
const int x = 10;

const int* ptr = &x;  //initialize

*ptr = 32;        //not ok - variable is const

const int y = 3;

ptr = &y;     //ok - pointer is non-const
```

- Case 3: non-const variable and const pointer to a non-const variable:

```
int x = 10;

int* const ptr = &x;  //initialize

*ptr = 32;         //ok - variable is non-const

int y = 3;

ptr = &y;      //not ok -  pointer is const
```

- Case 4: const variable and const pointer to a const variable:

```
const int x = 10;

const int* const ptr = &x;  //initialize

*ptr = 32;          // not ok - variable is const

const int y = 3;

ptr = &y;      //not ok - pointer is const
```

Quiz

1. Write a function to swap (exchange) values of two variables.
2. What are the similarities and differences between pointers and references?

C-Style Arrays

- A container is used to store/organize data in a program.
- The array is a fundamental container in C++.
- C++ inherits the array from C programming language.
- An array is a fixed collection of similar type of items that are stored in a contiguous (consecutive) block in memory.

- We define the size of the array at the creation time:
  `int myarray[5];`

  defines an array of 5 integers, i.e. a block of 5 consecutive integers named `myarray[0]`, `myarray[1]`, `... myarray[4]`.
- We use an index to access the elements in an array. In C++, the array index starts at 0 (zero).
- The code snippet below shows how to read and write to an array using an index[3].

```
myarray[0] = 3;
myarray[1] = 2;
myarray[2] = 5;
myarray[3] = 9;
myarray[4] = 1;

int sum = myarray[0] + myarray[1] + myarray[2]
          + myarray[3] + myarray[4];
```

[3] You'll very soon see that we don't access individual elements like this. We use loops (next topic).

- We can use an initializer-list to initialize an array:
  `int myarray[] {2, 3, 1, 14, 8};`

- When we use arrays, we need to:
  1. use a non negative index (no underflow)
  2. make sure index < array size (no overflow)
- They are 2 most common mistakes associated with using arrays in C++.

Arrays and Pointers

- In C and C++ arrays and pointers have a very strong relationship.
- The name of the array is the same as the location of its first element (i.e. the address of element at index 0).
- Let's declare an array of integers (the name of the array is `myarray`):
  ```
  int myarray[10];
  ```
- The first element of the array:
  `myarray[0]`
- The address of the first element:
  `&myarray[0]`
- Memory address/location of the first element is also given by:
  `myarray`

- Why is this important?
  - I can get the 1^{st} element (index 0):
    ```
    int elem_1 = *myarray;
    ```
  - And, the 2^{nd} element (index 1):
    ```
    int elem_2 = *(myarray +1);
    ```
  - And, the n^{th} element (index n-1):
    ```
    int elem_n = *(myarray + n - 1);
    ```
- This is the power of arrays. We can access an array using a pointer increment, which is an extremely efficient operation.

C-Style Strings

- In C the term string refers to a variable length array of characters.
  `char msg[3];`

- C++ inherits the strings from C.
- These strings are commonly referred to as C-style strings to distinguish from the string class available in the C++ Standard Library.
- We will use the `std::string` type in the Standard Library when we work with strings.

Control Structures

Control Structures

- Useful programs in real life usually do not execute every statement in a program sequentially (one by one); they:
  - Branch based on a condition
  - Run some code in a loop until a condition is met:
- We use control structures to achieve branching and loops.
- Branching:
  1. if/else
  2. switch
- Loops:
  1. while
  2. do/while
  3. for

Using if/else

The `if/else` keywords are used to achieve conditional structure in code.

- Using `if`:

```
if (condition == true)
{
   statement1;
   statement2;
}
```

- Using `if/else`:

```
if (condition == true)
{
   statement1;
}
else //condition is false
{
  statement2;
}
```

- Using nested `if/else`:

```
if (condition1 == true)
{
   statement1;
}
else if (condition2 == true)
{
  statement2;
}
else // no condition is true
{
  statement3;
}
```

Example1: Using if

```
int main()
{
   cout << "Enter Number: ";

   int x = 0;

   cin >> x;

   if (x > 0)
   {
      cout << ''positive value";
   }
   if (x < 0)
   {
      cout << "negative value";
   }
   if (x==0)
   {
     cout << "zero";
   }
}
```

Example 2: Using if/else

Here's the same example, written differently, using nested if/else.

```
int main()
{
   cout << "Enter Number: ";

   int x = 0;

   cin >> x;

   if (x > 0)
   {
      cout <<  ''positive value";
   }
   else if (x < 0)
   {
     cout << "negative value";
   }
   else
   {
     cout << "zero";
   }
}
```

Homework

1. What (if any) are the differences between the two examples?
2. Run the two code segments in the debugger and observe similarities/differences.
3. Use the table below to find the letter grade for a given score (0-100).

Range	Grade
95-100	A
90-94	A^-
85-89	B^+
80-84	B
75-79	B^-
65-74	C
0-64	F

Using switch

- With the `switch` you don't need to write deeply nested `if/else` branches.
- `switch` checks an expression against a set of **constants**:
  - selects the matching case
  - if no matches, the default is chosen
  - we use `break` to exit a `case` block
- The expression must evaluate to an integral (can be expressed as an integer value), or, enumeration value.

```
switch(expression)
{
   case const_value1:
       statement1;
       break;
   case const_value2:
      statement2;
      break;
   default:
      statement3;
}
```

Example: switch

```
cout << "Enter number: ";
int value = 0;
cin >> value;

switch(value)
{
  case 0: cout << "input: zero" << endl;
          break;

  case 1: cout << "input: one" << endl;
          break;

  case 2: cout << "input: two" << endl;
          break;

  default: cout << "input is not 0, 1 or 2";

}
```

Homework:

1. What happens if you remove the `break`?

Iterative (loop) Statements

- `for`, `while`, and `do/while` keywords are used to achieve repeated execution (loops) in code.

```
while(condition == true)
{
     statement1;

     statement2;
}
```

Example: while

Let's look at an example:

```
int n = 0;

while (n < 10)
{
    cout << " n : " << n << endl;

    n = n + 1;
}
```

What's the output of the code snippet above?

do/while

- `do/while` is similar to `while` statement.
- The condition is evaluated after the statement body is completed.

```
do
{
     statement1;

     statement2;

} while(condition == true);
```

Example: do/while

Example below shows how to use do/while to read an input:

```
do
{
     cout << "Enter number (0 to end): ";

     cin >> n;

     cout << "You entered: " << n << endl;

 } while (n != 0);
```

Homework:

1. Write the same program using a `while` loop.
2. Compare `while` and `do/while` structures.

for Loop

- `for` is another way to achieve repeated execution of a code block:

```
for (initializer; condition; expression)
{
    statement1;

    statement2;
}
```

- Example:

```
for (int n=0; n<10; n=n+1)
{
    cout << "n : " << n << endl;
}
```

- We usually write this as:

```
for (int n=0; n<10; ++n)
{
    cout << "n : " << n << endl;
}
```

Example

The factorial of a non negative integer n, denoted by n! is defined as:

```
n! = n * (n-1) * (n-2) * ..... 1
n! = n * (n-1)!
```

E.g.:

```
5! =5 * 4! =5 * 4 * 3! =5 * 4 * 3 * 2! =5 * 4 * 3 * 2 * 1
```

Write a function to find the factorial of any non negative number.

Example

Write a function to find if a given number is a prime number:

- A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.
- How do we write our function body?
  - use % operator to find if a number is a factor of another.
  - take each number from 2, upto one less than the given number, and see if each number is a factor.
  - use a loop such as `for` or `while`.

Homework

1. Write a function to find Fibonacci numbers: (`https://en.wikipedia.org/wiki/Fibonacci_number`).
2. Write a function to find the square root of a number using the Babylonian method: (`https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method`).
3. Sorting:
   - We use *sorting* to arrange elements in a certain order.
   - The order can be ascending (in increasing order, from smallest to largest) or descending (in decreasing order, from largest to smallest).
   - Implement the *Bubble Sort* algorithm described here: `https://en.wikipedia.org/wiki/Bubble_sort`

Currency Converter

Currency Converter

Objective: To write a program to convert a given amount from one currency to another. We will write this program in two steps:

1. Write program to convert a value in USD to another currency (class discussion).
   - user inputs:
     - value in USD
     - foreign currency
2. Modify the program to convert a value from any currency to another (Assignment 1).
   - user inputs:
     - value
     - local currency
     - foreign currency

Exchange Rates

- We're going to use the following exchange rates:

Currency Pair	Value
USD/EUR	0.88
USD/GBP	0.73
USD/CAD	1.25
USD/AUD	1.39

- E.g. The quote USD/EUR (0.88) identifies the number of Euros per US Dollar.

Reading the Inputs

- First, let's write code to read the user inputs.
- We read the initial amount as a `double` value, and the foreign currency symbol as a `std::string`:

```
cout << "Enter amount in USD: " ;

double amount;

cin >> amount;

cout <<"Enter foreign currency  (EUR/GBP/CAD/AUD): ";

string foreignCurrency;

cin >> foreignCurrency;
```

Choosing the Rate using if/else

- We have to select the appropriate exchange rate based on the user input.
- This is clearly a conditional structure.
- We can use `if/else` to select the appropriate exchange rate.

```
if (foreignCurrency == "EUR") rate = 0.88;

else if (foreignCurrency == "GBP") rate = 0.73;

.....
```

- What can we say about this solution?

- We have a nested `if/else` block here.
- If we add more currencies this block gets deeper.
- Can we `switch` instead?

enum

- We cannot use a `string` in our `switch` statement. A `switch` statement tests an integral or enum value against a set of constants.
- One solution is to use an `enum` here.
- The `enum` type is used to define a **collection of named integer constants**.
- The keyword `enum` is short for *enumerated*.

```
enum CurrencyType { USD=0, EUR=1, GBP=2, CAD=3, AUD=4};
or,
enum CurrencyType { USD=0, EUR, GBP, CAD, AUD};
```

- Now we can use this `enum` type in our `switch` statement.

Choosing the Right Rate - Using switch

```
enum CurrencyType { USD=0, EUR=1, GBP=2, CAD=3, AUD=4};

.....................

.....................

switch (currencyType)
{
    case EUR:
        rate = 0.88;
        break;

   ..............................
   ..............................

     default:
        rate = 1.0;
}
```

Reading Input CurrencyType

- We have a drawback – we need to read the input currency type as an int:

```
cout <<"Enter foreign currency (USD=0;EUR=1;GBP=2;CAD=3;AUD=4): ";

int foreignCurrency;

cin >> foreignCurrency;
```

Assignment 1 (Graded)

Write a Currency Converter program:

1. Should handle 7 currencies (including USD).
2. The user should be able to select the base currency and the foreign currency.
3. The user should be able to perform any number of currency conversions in a single program run (without restarting the program).
4. Handle graceful program shutdown based on a user input.
5. **Use functions** whenever appropriate.
6. You are not expected to test and validate the inputs (we will handle that later).

- Individual Assignment.
- Due: Saturday, January 21, by midnight (CST).
- Hint: You can convert from any currency to any other currency in two steps: first convert to USD, then convert from USD.

Procedural Programming

Advantages:

- Follows a *natural* approach to problem solving.
- Easy for the new programmers to understand.
- Allows quick completion.
- In general, procedural programming languages rely on static type checking, which ensures better performance than dynamic type checking (dynamic type checking will be discussed later in the course).

Procedural Programming

Limitations:

- Offers some limited code reusability but cannot extend easily.
- Emphasis was only laid on the actions or functions, however the actual purpose for which computer programs are made is the storage and management of data.
- Difficult to use for large scale projects.