ENGG1340

C++ Notes for HKU · Spring 2024

Author: Jax

Contact: enhanjax@connect.hku.hk

MORE notes on my website!

Contents

1 C++ basics

Basic differences to Python:

1. C++ is a *compiled* language, which means that the code needs to be compiled before it can be run.
2. C++ is a *statically typed* language, which means that the type of a variable is known at compile time.
3. C++ expressions end with a semicolon `;`.

1.1 Initialization

For a `.cpp` file, we must include the following contents:

```cpp
// include different libraries
#include <iostream>  // iostream provides basic io functions
using namespace std; // use the "std" namespace
int main()
{
    // code to be ran when program starts
}
```

For this document, all codeblocks are assumed to be in the `std` namespace.

Without `using namespace std;`, we can use the `std::` prefix to access the objects and functions in the standard namespace.

1.2 Types

C++ is a statically typed language, which means that the type of a variable is known at compile time. The following are the basic types in C++:

```cpp
char 'b' // only one character
int 1 // integer
float 1.0 // floating point number
double 1.0 // double precision floating point number
bool true // boolean

const int a = 1; // constant variable that cannot be changed
```

1.2.1 Chars

We use single quotes to represent a `char`. Note that there are no built-in string types in C++. A string is simple an array of characters.

`char` stores a single character as a number, and the ASCII value of the character is stored in the `char` variable. If we assign a char to an interger, the ASCII value of the character is stored in the integer variable.

This means that if we perform arithmetic operations on a `char`, the ASCII value of the character is used, and the result is stored as an integer.

```
char letter = 'a';
int val = letter; // 97
cout << 'b' - 'a'; // 98 - 97 = 1
letter++ // 'b'
```

```
isdigit('1') // 1
isalpha('a') // 1
isalnum('1') // 1
isupper('A') // 1
islower('a') // 1
toupper('a') // 'A'
tolower('A') // 'a'
```

1.3 Variable declaration

```
int a = 1, b = 2, c = 3; // declare vars with vales
double a, b, c; // empty declaration
```

1.4 Operators

```
n = ++i;      // equiv to i = i + 1, n = i;
n = i++;      // equiv to n = i, i = i + 1;
n = 1 && 0;   // logical and
n = 1 || 0;   // logical or
n = !3;       // logical not, converts any number > 0 to 0 (!3 => 0)

// C++ contains the usual operators: +, -, *, /, %
10 / 3 // If both types are integers, the result is an integer (3)
10.0 / 3 // If either type is a float, the result is a float (3.3333)
```

1.5 cmath library

The `cmath` library contains a set of functions that can be used to perform mathematical operations.

```
#include <cmath>
int a = pow(2, 3); // 2^3 = 8
int b = sqrt(16); // square root of 16 = 4
int c = abs(-10); // absolute value of -10 = 10
```

1.6 cstdlib library

The `cstdlib` library provides the `rand()` function, which returns a random number.

```
rand() % N; // random number between 1 and N-1
```

2 Streams

Streams are a sequence of bytes that represent a flow of data. In C++, we use the `iostream` library to work with streams. The `iostream` library contains the `cin` and `cout` objects, which are used to read and write data to the console, respectively.

2.1 Character I/O Stream

```
cin >> name >> age; // read multiple inputs to variables
cout << "Hello " << name << endl; // Hello <name>
```

1. `>>` is the *extraction operator*
2. `<<` is the *insertion operator*
3. We can use `\` to escape special characters

2.2 File I/O Stream

We can also use the `fstream` library to work with files. The `fstream` library contains the `ifstream` and `ofstream` objects, which are used to read and write data to files, respectively.

2.2.1 Reading from a file

```
#include <fstream>
#include <string>

ifstream fin; // file input stream
fin.open("input.txt"); // open file for reading
if (fin.is_open()) { // check if file is open
    string line;
    while (getline(fin, line)) { // read line by line
        cout << line << endl;
    }
    fin.close();
} else {
    cout << "Unable to open file for reading.";
}
```

2.2.2 Write to a file

This following code creates `output.txt` if it doesn't exists and writes to it. If it does exist, it will be overwritten.

```cpp
ofstream fout; // file output stream
fout.open("output.txt");
if (fout.is_open()) {
    fout << "Hello World!" << endl;
    fout.close();
} else {
    cout << "Unable to open file for writing.";
}
```

We can use the `ios::app` flag to append to the file instead of overwriting it.

```cpp
fout.open("output.txt", ios::app); // open file for appending
```

2.3 String Input Stream

We can use the `sstream` library to work with strings. The `sstream` library contains the `istringstream`, which is used to read data from a string.

The following code separates the string `str` by spaces and stores the separated values in the `word` variable, then prints them line by line.

```cpp
#include <sstream>
#include <string>

string str = "Hello World!";
istringstream line_in(str); // string stream

while (line_in >> word) {
    cout << word << endl; // "Hello\nWorld!"
}
```

2.4 Manipulators

Manipulators are special functions that can be used to format the output of the `cout` object. They are presistent, meaning that they will affect all subsequent output until they are changed.

2.4.1 End line

The keyword `endl` is a manipulator that represents a new line. It is equivalent to the `\n` character.

2.4.2 Decimal points

We can fix the number of decimal points by using the `fixed` and `setprecision` functions from the `iomanip` library.

```
#include <iomanip>                  // include iomanip library
cout << fixed << setprecision(2);  // set precision to 2 decimal places
cout << 3.14159;                   // 3.14
```

2.4.3 Width and alignment

We can fix the width of the output by using the `setw` function from the `iomanip` library.

```
cout << setw(6) << "123" << endl; // "   123"
cout << setfill('%') << setw(6) << "123" << endl; // "%%%123"
cout << left << setw(6) << "123" << endl; // "123   "
```

3 Control flow and functions

3.1 Basisc conditionals

```
if (a == 1) {
    // do something
} else if (a == 2) {
    // do something else
} else {
    // do something else
}
```

3.2 Switches

```
switch (ctrl_expr) {
    case const_1: // if ctrl_expr == const_1
        break; // man
    case const_2: // if ctrl_expr != const_1 && ctrl_expr == const_2
        break;
    default: // if no case matches (optional)
        break;
}
```

3.3 Tentary operators

```
(condition ? if_true : else_false) // returns the corresponding expression
```

3.4 Loops

```
for (int i = 0; i < 10; i++) { // initialize, check conditions, increment
    // do something 10 times
    continue;   // skip the rest of the loop
    break;      // break out of the loop
}

while (a < 10) { // check conditions first
    // do something
}

do { // execute first
```

```
    // do something
} while (a < 10); //then check conditions
```

3.5 Functions

```
//return_type func_name (type var1, type var2)
double func_name (int a, int b) {
    return a + b;
}
```

Note that nested function declaration is not allowed.

3.5.1 Pass by reference

Pass-by-reference means to pass the **reference of a variable** as the parameter of a function. The called function can **modify the value of the variable** by using its reference passed in.

This example uses reference arguments:

```
void swapnum(int &i, int &j) {     // pass by reference
  int temp = i;
  i = j;
  j = temp;
}

int main() {
  int a = 10, b = 20;
  swapnum(a, b);                   // a = 20, b = 10
}
```

This other example uses pointer arguments, so referenes are passed in:

```
void swapnum(int *i, int *j) {     // pass by reference
  int temp = *i;
  *i = *j;
  *j = temp;
}
int main() {
  int a = 10, b = 20;
  swapnum(&a, &b);                   // a = 20, b = 10
}
```

More in reference operators.

4 Arrays

An array is a collection of elements of **the same type**. The elements of an array are stored in contiguous memory locations. C++ is a zero-indexed language, which means that the first element of an array is at index 0.

Arrays are static, so arrays cannot be variable-lengthed**. Additionally, only one element of an array can be changed at a time. **You cannot assign entire arrays to each other.**

*** not part of the standard, but some compilers allow it.* Also check out dynamic arrays.

```
int a[10]; // array of 10 integers
int a[] = {1, 2, 3, 4, 5}; // array of 5 integers
int a[2][2] = {{1, 2}, {3, 4}}; // 2D array of 2x2 integers
char a[] = "Hello"; // array of 5 characters
```

4.1 Function array parameters

We can tell the function to expect an array as a parameter by using the square brackets `[]`.

Arrays passed into functions are **passed by reference**, so any changes made to the array in the function will be reflected in the original array.

```
void zeroize(int a[], int n) {
    for (int i = 0; i < n; i++) {
        a[i] = 0;
    }
}
int[5] a = {1, 2, 3, 4, 5};
zeroize(a, 5); // a = {0, 0, 0, 0, 0}
```

For 2D arrays, we must specify the number of columns in the function parameter.

```
void zeroize(int a[][5], int n) {
...}

char letter = 'a';
int val = letter; // 97
cout << 'b' - 'a'; // 98 - 97 = 1
letter++ // 'b'
isdigit('1') // 1
isalpha('a') // 1
```

4.2 Character arrays (C-string)

A string is simply an array of characters. The last character of a string is the null character `\0`, which is used to denote the end of a string. The null character is automatically added to the end of a string when it is declared.

```
char hi[] = "Hello"; // hi[6] = {'H', 'e', 'l', 'l', 'o', '\0'}
```

This also means that you cannot directly assign a string to the variable after declaration, as the same of arrays.

```
hi = "World"; // raises compilation error
```

4.3 The string class

You can also use the string library to work with strings. The library provides the `string` class, as well as functions for working with strings.

4.3.1 String concatenation

We can use the `+` operator to concatenate strings. Note that at least one of the operands must be a string object.

```
#include <string>
string hi = "Hello";
string who = "World";
string message = hi + " " + who; // "Hello World"
string message = "Hello" + " World" // raises compilation error
```

4.3.2 String comparison

String comparisons are carried out character by character, and stops when a difference is found. The comparison is by ANSCII value.

```
string msg1 = "Apple", msg2 = "apple";
string msg3 = "apples", msg4 = "orange";
bool c1 = msg1 == msg2; // false
bool c2 = msg3 != msg4; // true
bool c3 = msg1 < msg2; // true (A < a)
bool c4 = msg2 < msg3; // true (0 < s)
bool c5 = msg4 > msg3; // true (o > a)
```

4.3.3 Member functions

```
a = "Hello World!";

a.length(); // 12
a.empty(); // false

a.erase(6, 6); // (from, n) "Hello "
a.insert(5, "New "); // (after, string) "Hello New World!"
a.replace(6, 5, "Universe"); // (from, n, string) "Hello Universe!"

a.substr(6, 5); // (from, n) "World"
a.substr(6); // "World!"

a.find("World"); // (string, start) 6 (starting index)
a.find("World", 7); // -1 (not found)
a.rfind("o"); // 7 (search from the end)
```

4.4 Vectors

Vectors are a sequence container that encapsulates dynamic size arrays.

```
#include <vector>

vector<int> v; // empty vector
vector<int> v(5); // vector of 5 integers
vector<int> v(5, 1); // vector of 5 integers with value 1
v.push_back(1); // add 1 to the end of the vector
v.pop_back(); // remove the last element
v.size(); // 5
v[0]; // 1
v.clear(); // remove all elements
v.front(); // first element
v.back(); // last element
```

We use vectors for dynamic arrays, and arrays for static arrays.

5 Structures

A structure is a user-defined data type that groups related variables of different types. A member is a variable within the structure.

```
struct Point {
    int x, y;
};
```

Structure variables is declared and initialized similar to how you would for variables. We use the dot operator to access the members of a structure.

```
Point p1 = {1, 2};
Point p2 = {1}; // p2.x = 1, p2.y = 0
Point p3 = {1, 2, 3} // raises compilation error
cout << p1.x; // 1
```

Note that we can only assign to a structure, and not perform operations on it.

Structures can be nested, and we can define member functions for a structure.

```
struct Circle {
    Point center;
    int radius;

    double area() {
        return 3.14 * radius * radius;
    }
};

Circle c1 = {{1, 2}, 3};
```

6 Dynamic memory management

6.1 Pointers

A pointer is a variable that stores the *memory address* of another variable.

6.1.1 The reference operator

`&` after the type name during delcaration indicates the variable serves as an alias to a variable (providing a **reference**).

```
int n = 0;
int & i = 0; // reference to j
n++;
cout << i; // 1
```

6.1.2 Pointer declaration and usage

`*` after the type name during declaration indicates the variable is a pointer.

`*` before the variable name **dereferences** the pointer and gets the value stored in the memory address.

`&` before the variable name gives the **memory address** of the variable.

```
int i = 0;
int * iPtr = &i; // iPtr = address of i
cout << *iPtr; // 0 (dereference -> value of i)
```

6.1.3 Possible errors

Note that pointers **must have the same type** as the variable they point to.

```
int i = 0;
char * badPtr = &i; // raises compile-time error
```

We must initialize a pointer to `nullptr` (same as 0) or a valid memory address before using it. If we don't, the pointer will be a **dangling pointer**.

```
int * a = nullptr, * b = 0; // declare multiple empty pointers
```

Dereferencing a dangling pointer raises a **compile-time error**.

6.1.4 Pointer operations

We can carry out *addition* and *subtraction* on pointers, which will move the pointer to next or previous memory addresses. We can also carry out *subtraction* on two pointers, which will give the number of elements between the two pointers.

```
int a[2] = {0, 1};
int * aPtr = a;
cout << *aPtr << ' ';
cout << *(aPtr+1) << endl; // 0 1
```

We can also compare pointers with `==` and `!=`.

6.1.5 Member access operator

We can use the `->` operator to access the members of a structure that a pointer points to.

```
struct Data {
    int a = 1;
}
Data data;
Data * dataPtr = & data;
cout << dataPtr->a; // 1
```

6.1.6 Array pointers

The name of an array is a **pointer to the first element** of the array. Therefore, we can use pointers to access the elements of an array as well.

```
char x[10];
char * xPtr = x; // pointer to x
char * xPtr = &x[0]; // reference to x[0], same as above
xPtr[0] = 'a'; // x[0] = 'a'
```

6.2 Dynamic variables and memory allocation

Dynamic variables are variables that are allocated memory at runtime. We use the `new` operator to allocate memory for a dynamic variable, and the statement returns the memory address of the allocated memory. We can then store the address in a pointer to access the dynamic variable.

```
int * i = new int (42); // allocate memory for an integer
cout << *i; // 42
```

A dynamic variable is deallocated using the `delete` operator. It is important to deallocate memory when it is no longer needed to prevent memory leaks. Additionally, the pointer to the memory should be set to `0` after deallocation to prevent the pointer from becoming a *dangling pointer*.

```
delete i; // free memory
i = 0;
```

6.2.1 Dynamic arrays

Unlike static arrays, dynamic arrays can have their size changed at runtime.

```
int n = 10;
int * a = new int [n]; // allocate memory for an array of 10 integers
delete [] a; // free memory
```

6.3 Linked lists

A linked list is a data structure that consists of a sequence of nodes (elements), where **each node points to the next node** in the sequence. The last node points to `nullptr`. Therefore, items **need not be stored in contiguous memory locations**.

This allows for **efficient insertion and deletion** of nodes, as we only need to change the pointers of the nodes around the element we want to insert or delete.

However, a linked list is a **sequential access** data structure, so to access the nth element, we must traverse the list from the first element. This is different from an array, where you can access the nth element directly.

6.3.1 Implementation

```
struct Node
{
    int data;
    Node * next;
};

Node * head = new Node;
Node * node1 = new Node;

head->data = 1;
head->next = node1;
node1->data = 2;
node1->next = nullptr;
```

We tranverse through the linked list from `head` to access the data stored in the elements.

```
cout << head->data << ' ' << head->next->data << endl; // 1 2
```

We can also use a while loop to do so:

```
Node * current = head; // remember head is a pointer
while (current != nullptr) {
    cout << current->data << ' ';
    current = current->next;
```

```
} // 1 2
```

7 Recursion

A recursion function can be defined by the following:

1. $f(b) = k$ A basecase and it's result
2. $f(a) = f(b)$ A recursive case which breaks the problem down into smaller problems for when it's not the basecase.

We can implement a recursive function as followed:

```
int recursion(int var) {
    if (basecase) {
        return var;
    } else {
        return recursion(var - 1);
    }
}
```