2 - Basic Language Features

January 8, 2025

COMP2404

Darryl Hill


COMP
2404
virtual machines
interpreters
Garbage Collection

(Short) History of Programming Languages

1. Regional Assembly (1951)
2. FORTRAN (1957)
3. Lisp (1958)
4. COBOL (1959)
5. BASIC (1964)
6. B (1969)
7. Pascal (1970)
8. C (1972)
9. Prolog (1972)
10. Scheme (1975)

(Short) History of Programming Languages

11. C++ (1980) (C with classes, renamed in 1983)
12. Ada (1983)
13. Objective C (1986, Apple for MacOS and iOS)
14. Perl (1987)
15. Haskell (1990)
16. Python (1990)
17. R (1993)
18. Java (1995)
19. Javascript (1995)
20. PHP (1995)

Programming Languages

What is the most powerful language?

- the set of problems that are solvable on a computer is the same for all (Turing Complete) languages

- Then what is the difference?
  - Speed
  - Convenience
    - Libraries
  - Compiled vs Interpreted
  - Some are fast, some are expressive, some attempt to do both

C++ has high-level constructs and systems level control

- Ethos - the programmer should not be restricted from doing anything

All programming languages manipulate data

Data Types

Data types:

Primitive

- `int, float, double, char, bool`
- About a constant size (roughly the same as a memory address – more on this later)

Aggregate

- classes - `class Student{ };`
- structs - `struct Student{ };`

Memory address

- pointers - `Student* stu = new Student;`

Variable and Data Types

All memory looks the same

- 10010010010101110100 1101
- Is that a `double, int, char`?
- How does the compiler know?

Memory is pointed to by another piece of memory that has a label that tells the type

- label tells us how to interpret those bits – what value is contained
- `int, double, char`, etc
- This "pointer" memory has an address
- C++ lets us manipulate addresses as well as values

Variables

Variables have:

- Name
  - How the compiler and programmer identify a variable.
- Type
  - Variable points at a piece of memory with a bit value.
  - Type tells compiler (and programmer) how to interpret that value.
- Address
  - Where in memory the value is stored.
- Value
  - Technically a bit value, but interpreted according to the type

Memory

In C++ we manage our own memory

- Very important to understand how memory is used
- We will give a basic description here and come back to this model again and again...

Two main places for memory:

Function call stack

- Fast, convenient, but temporary

Heap

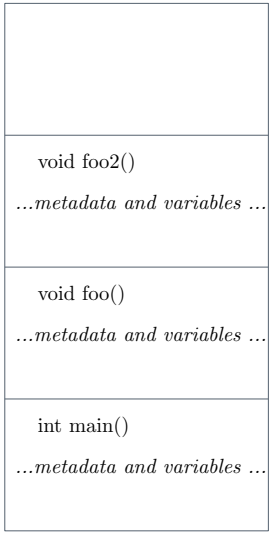
- Slower to allocate but lasts until `delete` is called
- Lifespan must be carefully managed to avoid segfaults and memory leaks

Function Call Stack

Every time you call a function, a new **Stack Frame** is put on the **Function Call Stack**

- Any local variables are stored in that stack frame
- These variables are deleted automatically when we return from a function

```
void foo2(){
    int x;
}
```



Heap

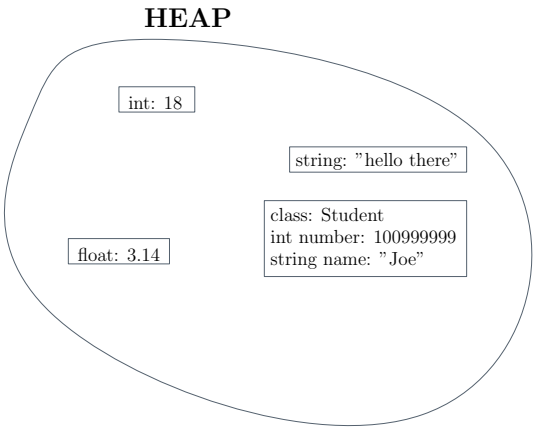
Variable declared with `new` goes on the **Heap**

- `new` keyword returns a pointer
- A pointer contains a memory address

```
int* x = new int;
*x = 18;
```

- must dereference (*) the pointer to access the int value
- variable remains in memory until `delete` is called on the pointer

```
delete x;
```



Declaring Variables

Two ways to declare a variable:

Statically allocated

```
int x;
float y;
```

- The variable has memory allocated **where it is declared**
  - could be on the **function call stack** or as data in a **struct** or **class**
  - the containing **struct** or **class** may be **statically** or **dynamically** allocated
  - this variable is deleted automatically

Dynamically allocated

```
int *x = new int;
float *y = new float;
```

- The variable has memory allocated **on the heap**
- deleted when `delete` is called
  - More on `delete` in the **Memory Management** section
  - For now we will focus on creation

Classes and Structs

Structs

- In C++ structs have **variables** *and* **functions**
- By default all members are **public**
  - Often (but not always) considered bad software engineering.
- **Do not use *structs* in place of *classes***.
  - You will lose marks.
- ***Structs*** have their place, but we will use ***classes*** almost exclusively.

Classes

- Contain **variables** and **functions**
- By default all members are **private**
  - (Mostly) considered good software engineering.
  - Prevents misuse or corruption of data.
  - Known as ***the principle of least privilege*** – allow least amount of access to data necessary.

Operators

We can manipulate data using operators.
They are the usual suspects, and you use them in ways you would expect

Types of operators:

arithmetic (don't forget BEDMAS!)	`+ - * / % ++ --`
relational (logical, returns `bool`)	`== <= >= != < >`
logical (returns `bool`)	`&& || !`
bitwise	`~ & | ^ << >>`
assignment operators	`= += -= *= /= %=`
conditional	`? :` (Like an `if` statement, but returns a value)

Programming example <p1>

Characteristics of Operators

Arity

- the number of arguments
- unary, binary, ternary

Precendence

- The order in which operators execute
- It is not simply left to right – BEDMAS

Associativity

- The order in which operators of the same precedence execute
- left-to-right or right-to-left

Characteristics of Operators

- Arithmetic operators have left to right associativity
- `a / b / c / d; // the order will affect the outcome`
- Assignment operators have right to left associativity
- `a = b = c = d; // all receive the value in d`
- Prefix vs postfix operators
  - `++x` vs `x++`
- prefix is slightly faster

C++ Terminology

Putting data and operators together gives us **Expressions**.

Expression:

- A sequence of operations that resolve to a value

```
2 + 2 - 3
3 * variable
2 * pow (5, 3)
2 < 3
```

Expressions are typically embedded into **Statements**

C++ Terminology

Statement

- An instruction or command terminated by a semi-colon
- Can be an expression – return value can be used or discarded
- Can be a function

```
v = 6 + 2;
getValue();
setValue(4);
```

Programming example <p2>

Control Flow Statements

Statements can also control the flow of the program.

Conditional

- `if-else`

Selective

- `switch`

Iterative

- `do-while, while, for`

Jump

- `break, continue`

C++ Terminology

How do statements control the program flow?

By dividing our code into **Blocks**

- been around since 1958 or so
- a sequence of statements between matching braces
  - functions
  - `if` statements, `for-loops`
  - blocks for no reason
- Blocks can have local variables

Blocks and Local Variables


```
int x = 8;

for (int i = 0; i < 10; i++){
    int x = i;
}

cout<<x<<endl;
```

```
int x = 8;

for (int i = 0; i < 10; i++){
    x = i;
}

{
    int x = 10;
    cout<<x<<endl;
}

cout<<x<<endl;
```

Scope

- Block scope
  - declared in a block
  - local to that block
  - "local variables"
- Global scope / file scope
  - declared outside of any block
  - "global variables"
  - we will learn to use them, but should NEVER be used in industrial grade applications

Programming example <p3>

Standard I/O Streams

C++ stream library

- `#include <iostream>`
  - allows standard input and output
  - in the `std` namespace
    - same namespace as `strings`
    - more on this later

- `cout` – member of the `ostream` class
  - an object with reference to standard output (console)
  - you can change the output stream associated with `cout` (to a `string` for example)
    - this does NOT change the standard output
    - `printf`, for example, will still print to the screen

Standard I/O Streams

C++ stream library

```
#include <iostream>
```

- `cin` – member of the `istream` class - standard input – i.e., from the keyboard
- `cerr` – member of the `ostream` class - standard error output (console, or you can set to a log file)

Programming example <p4>

Functions

Methods, subroutines, procedures, etc

- different languages have different names
- functionally (mostly) the same

In C++ there are two kinds of functions

- Global functions
  - belong to everyone
- Member functions
  - Belong to a class (or struct).
  - We will only refer to ***classes*** from now on.

Functions

Global functions

- C exclusively uses global functions.
- Defined outside of any block (except a namespace).
- Called from any function and any class in the program.
- Example: main().

Member functions

- Java exclusively uses member functions.
- Defined within a class – behaviour of that class.
- Called by an object of that class, however
- static (class) functions can be called on the class itself.

Functions

Function declaration:

- AKA function prototype or specification.
- Details of ***how*** to call your function.
  - Name, number and types of arguments, return type.
  - Compiler needs only the prototype to determine if you've used function correctly.
- Document your prototype to answer what your function does.
  - Descriptive variables and function names are often best, but when it doubt, provide comments.

Functions

Function implementation:

- Body of the function.
- Code that executes when the function is called.

Member (and some global) functions in C++:

- Declaration and implementation are stored in different files.
- Header has declaration.
- Source has implementation.

Functions

(Good) function design:

- Take in parameters (data), process data and return a result using
  - return value, or
  - output parameter(s).
- A function should be single purpose.
  - DON'T try and do everything in one function.
  - DO call another (helper) function if there is more to do.
- Single purpose functions mean:
  - Simple to write.
  - Simple to debug.
  - Simple to document.

Functions

(Good) function design:

- Encapsulate unnecessary details.
  - Prototypes are often enough for the user.
- Divide large tasks into helper functions.
- Functions should be reusable.
  - Anticipate other uses.
  - Make function as general as possible.

Functions

(Good) function design - Return values:

- Return values can be used to:
  - Indicate success or failure (bool).
  - To return a member (getters).
  - For *cascading* (which we'll see later).
  - To output the result of a function.
- In C++, results are sometimes returned using output parameters.
  - We want to modify a pre-existing object.
  - We want to return multiple values.
  - We want to add elements to a data structure.
- In this class we will use both return values and output parameters.

Functions

Return Values vs **Output Parameters**:

Return values:

- Allow for more elegant code (functions can be nested in expressions).
- Are simple and intuitive to use.
- Multiple values can be returned using `structs`

Output parameters:

- Allows for existing objects to be modified.
- Easier to return multiple values (don't have to create a struct)
- We can still return a pointer - double pointers
  - More on that later.

Function Design

Parameters

- Space is allocated on the function call stack.
  - Same as local variables.

Parameters can be passed in using

- pass-by-value, or
- pass-by-reference.

Function Design

Pass-by-value:

- The value passed in is copied into a local variable.
  - The value passed in can be a variable or expression.
- The function can only modify the local value - the original value is untouched.

Pass-by-reference:

- The memory address of a variable is ***copied*** and passed into the function.
  - Must be a variable or an expression that evaluates to a variable.
- This address "points" to the original value, allowing us to modify the original value.

Function Design

In C++, pass-by-reference can be done via ***pointers*** or ***references***.

- There are no references in C, only pointers.

To the compiler ***pointers*** and ***references*** are the same (or exceedingly similar).

- Both are addresses - same memory footprint.
- Use a different syntax.
- More on this shortly.

Parameter Roles

Input parameter:

- Value or information required by function.
- Sent in by calling function.

Output parameter:

- Result of the function – could be garbage.
- Parameter value is set by the function.

Input / Output parameter:

- A value we want changed by the function.

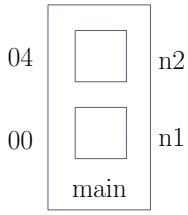
programming example <p5>

Pass by Reference/Value


```
someMath(int& num1, int num2){
    num2 = num2 - 1;
    num1 += num2;
}

int main(){
    int n1 = 10, n2 = 8;
    someMath(n1, n2);
}

someMath(                );
```

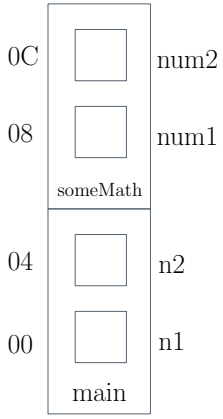


Pass by Reference/Value


```
someMath(int& num1, int num2){
    num2 = num2 - 1;
    num1 += num2;
}

int main(){
    int n1 = 10, n2 = 8;
    someMath(n1, n2);
}

someMath(                );
```

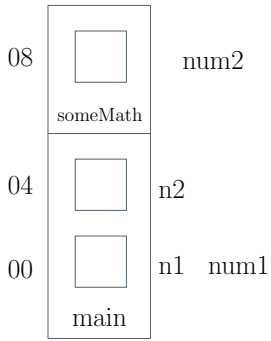


Pass by Reference/Value


```
someMath(int& num1, int num2){
    num2 = num2 - 1;
    num1 += num2;
}

int main(){
    int n1 = 10, n2 = 8;
    someMath(n1, n2);
}

someMath(                );
```



Pass by Reference

Essentially "pass by memory location". Two ways:

- **Pointers:**
  - Powerful.
  - Somewhat natural.
  - Easy to make mistakes with - segmentation fault, or worse, NULL reference (bad memory).

- **References:**
  - Same thing "under the hood" as pointers.
    - It is a memory location.
  - Syntactically treated as a variable.
  - Less versatile than pointers, but easier to use.
  - Supposed to protect against NULL references.
    - But you can do it if you try.

References

References can also be used outside of parameter passing.

- Must be assigned a memory address to a value on declaration.

```
int x = 10;
int& y = x;
```

- This memory location can never be changed.
  - This reference is permanently BOUND to the memory location of `x`.
  - We call it an ***alias*** for another variable.
    - It is another variable name accessing the same value.
  - The value stored in the memory location that is being pointed to CAN change.

- Most common usage is for passing parameters by reference.

programming example <p6>