C++ Part 3

Recap

- Non-primitive types, type casting, auto type
- cmath library and random numbers
- Strings are variable length sets of characters
- Arrays are fixed length sets of a single type
- Vectors are variable length sets of a single type
- For and while loops iterate and repeat code
- Arguments can be passed to main()

Memory allocation

- Variables are stored at some place in memory
- You can access the location in memory using the address-of operator (&)
- The address points to a particular place in memory, not the actual value

```
int myint = 10;
std::cout << &myint << std::endl;
```

Pointers

- Pointers store a memory location that can be referenced to get a value
    - Generally faster to use than complex data types
- Memory must be explicitly allocated (new) and deallocated (delete)
- Unallocated pointers usually should be initialized to nullptr (C++11)
- Use -> instead of . to call class methods on pointers
- Use dereference operator (*) to access object

```
std::string * mystring = new std::string("hello");
std::cout << *mystring << std::endl;
delete mystring;
```

```
std::vector<int> * myvec = nullptr;
myvec =  new std::vector<int>;
myvec->push_back(3);
std::cout << myvec->size() << std::endl;
std::cout << (*myvec).size() << std::endl;
delete myvec;
```

Smart pointers

- If pointers are not deleted, this can lead to memory leaks
  - Can cause jobs to crash if sufficiently complex code
  - Can be difficult and painful to track down
- Smart pointers (std::unique_ptr) provide easier memory management (C++11)
  - Need to include memory library
- Memory is automatically released when scope is exited
- After initialization, treat as a raw pointer

```
std::unique_ptr<int> myint(new int(7));
```

```
std::unique_ptr< std::vector<int> > myvec;
myvec.reset(new std::vector<int>);
myvec->push_back(3);
std::cout << myvec->size() << std::endl;
```

Functions - intro

- Functions enable allow more compact and cleaner code
- Reduce redundant code
- Modular - can be used in multiple places
- Fundamental aspect of class definitions (more next time)
- Functions must be declared or defined before they are called in main()

```
int myfunc() {
  return 7;
}
int main() {
  std::cout << myfunc() << std::endl;
  return 0;
}
```

```
int myfunc();
int main() {
  std::cout << myfunc() << std::endl;
  return 0;
}
int myfunc() {
  return 7;
}
```

Functions - arguments

- Functions can be defined using arguments
- Argument types and names are defined in function declaration/definition
- Passed values are copied into local variables within function
    - It is good practice to ensure all arguments are used

```
int sum(int x, int y) {
  return x + y;
}
int main() {
  std::cout << sum(8,5) << std::endl;
  return 0;
}
```

Functions - return values

- Functions should be terminated with a return statement
  - Not strictly necessary in all cases, but a good practice
  - void functions don’t need a return statement
  - return can be used with logical controls to terminate function early
- Function declaration defines what type of value is returned
  - Returned value must be castable into return type
- Only a single value can be returned by a function
- In case multiple outputs are needed from a function, there are options

Overloaded functions

- Functions can be overloaded to cover multiple use-cases
  - E.g., sum either 2 or 3 (or an arbitrary amount) values that could be float or int
  - Different function names could be used, but overloading can be easier for maintenance
- Declare multiple functions with same name but with different return type and/or set of arguments
- Compiler will automatically assign correct version (or complain if ambiguous)

```
int sum(int x1, int x2);
int sum(int x1, int x2, int x3);
int sum(std::vector<int> x);
float sum(float x1, float x2);
float sum(float x1, float x2, float x3);
float sum(std::vector<float> x);
```

Pairs

- A std::pair holds two variables of any type (need utility library)
  - Variable types are defined in declaration
- Useful when you want two return values from a function
- Initialized using std::make_pair(...)
- Elements accessed with first and second (note: no parentheses)

```
std::pair<char,int> mypair1("a",7);
std::pair<char,int> mypair2;
mypair2 = std::make_pair("b",4);
std::cout << mypair1.first << std::endl;
mypair2.second = 9;
```

Pass by reference

- Passing an argument to a function by reference can directly change variable
  - Address in memory is being passed, so actual location of variable, not just its value is used
- Use address-of operator (&) in function declaration

```
int myfunc(int &x) {
  x += 3;
  return 7;
}
int main() {
  int y = 9;
  int z = myfunc(y);
  std::cout << y << std::endl;
  std::cout << z << std::endl;
  return 0;
}
```

Recursion

- A function can recursively call itself
  - Generally return the function again with different arguments
- Often same functionality can be achieved with loops
- Important: define stopping conditions to return a default value!!

```
int factorial(int x) {
  if(x < 2) return x;
  return x * factorial(x-1);
}
int main() {
  int y = 9;
  int z = myfunc(y);
  std::cout << y << std::endl;
  std::cout << z << std::endl;
  return 0;
}
```

Resources

- https://www.w3schools.com/ - Great online learning resource
- https://www.youtube.com/@codebreakthrough - Excellent tutorial videos
- https://en.cppreference.com/w/ - Thorough documentation
- https://stackoverflow.com/ - Ask questions to experts