C++ Notes

Piotr Karamon

Contents

1 Building C++ application

Building an application means gathering all the text source files and generating an executable file. There are two main parts of it:

1. compiling
2. linking

1.1 Compiling

When you compile your source code, every file will be transformed into an `.obj` file. Which is an intermediate file format. Compiling involves lots of steps.

- Preprocessing The first one is preprocessing the code which means evaluating directives such as ==.
  - == this directive simply copies the code from specified file and pastes it. This is very simple mechanism can be confusing at times.
  - `= this directive will change every occurrence in this example =AGE` to 42
  - `= if =STH` is equal to 1 then the code inside will be kept otherwise it will not be included you can optionally use == as well.
- Creating an abstract syntax tree
- Convert the code into constant data or instruction

1.2 Linking

Linking is done by a program called the linker. It will take `.obj` files and link them together to create the final executable. The purpose of linking is finding where each symbol and function is and link them together.

1.2.1 Most common linker errors:

- **undefined reference / unresolved external symbol** → the linker could not find your symbol or function, you possibly forgot == or have spelling errors
- **multiple definition of** → when linker finds two identical symbols it throws an error because it does not know which one it should pick.

1.2.2 `static` keyword(not in class)

When you use the `= keyword it tells the linker that a function will only be used inside of the current file. Therefore if there are no references to that function inside of the file it will simply be deleted. Every file which includes({{{results(=)}}})` a static symbol will essentially have the own copy of it.

1.2.3 `inline` keyword

It will replace all the calls to the function with function body. Therefore there is no need to link an inline function, because it's not a function as far as **.obj** files are concerned.

2 Pointers

2.1 What is a pointer?

A pointer is a variable which holds a memory address.

2.2 Null pointer

Every memory address can be described as a integer value. However some integers are not valid memory addresses.

> If a pointer is equal to 0 we call it a **null pointer**. Null pointer often signifies absence, and it is not always a result of a human error.

In C++ we usually do not set a pointer to 0 instead we use

```
int* ptr1 = NULL;
int* ptr2 = nullptr;
cout << (ptr1 == ptr2) << endl;
```

```
1
```

2.3 Pointer example

Address	Value(32bit)
100	
101	
102	
103	
104	
105	
106	
...	...

```
int a = 5;
```

For the sake of the example the compiler used memory address location 102 to represent the variable `a`. We can say that whenever we see the variable name `a` what we really mean is go to somewhere in RAM and look up the address 102 and return the value.

`a = RAM[102]` → this is **direct** addressing.

Address	Value(32bit)
100	
101	
a 102	5
103	
104	
105	
106	
...	...

```
int *b = &a;
```

The == means get the address of the variable.

The compiler stores the value of b at the address 104.

Address	Value(32bit)
100	
101	
a 102	5
103	
b 104	102
105	
106	
...	...

```
int c = *b;
```

This is an example of indirect addressing. The compiler will first check what address is stored in variable b and then use that address to find a variable inside RAM and finally return that value.

Address	Value(32bit)
100	
101	
a 102	5
103	
b 104	102
105	
c 106	5

2.4 Pointer types

C++ does not have a `pointer` variable type. When we create a pointer we **must** specify to which type it points to.

Let's say we write:

```
int a = 5;
```

`int` is usually 32bits long so it will take up 4 memory address in RAM. Let's assume that the compiler decides to store `a` at address `100`

Address	Value(**8bit**)
100	A0
101	A1
102	A2
103	A3
104	
105	
106	
...	...

```
int* b = &a;
```

Why specify the type =? `Well, first of all the compiler will now know how many memory addresses to get when getting a value stored at =b` (4 in this example). Secondly it allows for *cleverer* pointer arithmetic.

```
// this
b = b+1;
// is equivalent to this
b = (int*)((long)b+sizeof(int));
```

2.5 Pointer examples with arrays

Remember an array is is really just a glorified pointer.

2.5.1 Basic pointer arithmetic using an array

```
int array[10];
int *pLocation6 = &array[6];
int *pLocation0 = &array[0];

cout << "pLocation6 = " << (long)pLocation6 << endl;
cout << "pLocation0 = " << (long)pLocation0 << endl;
cout << "Difference = " << pLocation6 - pLocation0 << endl;
```

```
pLocation6 = 140730167457448
pLocation0 = 140730167457424
Difference = 6
```

2.5.2 Iteration using indexes.

```
int array[10] = {3,6,9,12,15,18,21,24,27,30};
for (int i = 0; i < 10; i++) {
    cout << "element at index " << i << " equals " << array[i] << endl;

}
```

```
element at index 0 equals 3
element at index 1 equals 6
element at index 2 equals 9
element at index 3 equals 12
element at index 4 equals 15
element at index 5 equals 18
element at index 6 equals 21
element at index 7 equals 24
element at index 8 equals 27
element at index 9 equals 30
```

2.5.3 Iteration using pointer arithmetic.

```
int array[10] = {3,6,9,12,15,18,21,24,27,30};

for (int* p = array; p < array+10; p++) {
    cout << "memory address " << (long)p << " value " << *p << endl;

}
```

```
memory address 140733780624912 value 3
memory address 140733780624916 value 6
memory address 140733780624920 value 9
memory address 140733780624924 value 12
memory address 140733780624928 value 15
memory address 140733780624932 value 18
memory address 140733780624936 value 21
memory address 140733780624940 value 24
memory address 140733780624944 value 27
memory address 140733780624948 value 30
```

2.5.4 Function iterating over an array

When we want to write functions that deal with arrays we need to accept at least two things.

- **pointer** which tells the function where to start processing
- **amount of items** which tells how many elements to process

```
int find_max(int* array, int size){
    int max = *array;
    for(int i = 0; i < size; i++){
        int n = *(array+i);
        if(n > max) max = n;
    }
    return max;
}

int main(){
    int nums[] = {3,7,4,-10,3,5};
    cout << "max = " << find_max(nums, sizeof(nums) / sizeof(int)) <<
    ↪  endl;
    cout << "max of the last 4 elements = "
         << find_max(&nums[2], 4) << endl;
    return 0;
}
```

```
max = 7
max of the last 4 elements = 5
```

2.6 Arrays of characters

2.6.1 Introduction

An array of characters is really fundamentally no different to any other array. In reality an array of characters is just a continuous block of 8bit values. **Character arrays are used to describe text.** When you want to use a character array as a string of text **you must finish it with the null terminator** to signify the end of your text.

```
// this
char* text = "abc";
// is the same as this, null terminator='\0'
char* text = {'a', 'b', 'c', '\0'};
```

Many functions treat character arrays(pointers to chars,==) as text, meaning that often the type == get specially treated. For example

```
1 int a = 3;
2 int *b = &a;
3 char initials[3] = {'P', 'K', '\0'};
4 cout << b << endl;
5 cout << initials << endl;
```

```
0x7ffcbd644164
PK
```

The line 5 printed text to the console, instead of a raw address.

2.6.2 Basic operation and properties

```
char text[] = "hello!";
cout << "sizeof(char) = " << sizeof(char) << endl;
// prints 7 because there is also the null terminator
cout << "sizeof(text) = " << sizeof(text) << endl;
cout << "text = " << text << endl;
cout << "mem address (long)text = " << (long)text << endl;
char* substring = text+2;
cout << "substring = " << substring << endl;
```

```
sizeof(char) = 1
sizeof(text) = 7
text = hello!
mem address (long)text = 140724234495345
substring = llo!
```

2.6.3 Our own print function

```
#include <iostream>

using namespace std;

void print(const char* text) {
    while((*text) != '\0') {
        putchar(*text);
        text++; // go to next character
    }
```

```
}

int main(){
    print("hello world!");
    return 0;
}
```

```
hello world!
```

2.7 Structures

```
struct Point {
   int x;
   int y;
   char label;
};

cout << "sizeof(Point) = " << sizeof(Point) << endl;
Point point{2,3, 'a'};
printf("point info: %d %d %c\n", point.x, point.y, point.label);
cout << "(long)&point = " << (long)&point << endl;
cout << "(long)&point.x = " << (long)&point.x << endl;
cout << "(long)&point.y = " << (long)&point.y << endl;
cout << "(long)&point.label = " << (long)&point.label << endl;
```

```
sizeof(Point) = 12
point info: 2 3 a
(long)&point = 140722843212972
(long)&point.x = 140722843212972
(long)&point.y = 140722843212976
(long)&point.label = 140722843212980
```

```
struct Point {
   int x;
   int y;
   char label;
};
Point point {x:2, y:3, label:'a'};
long address = (long)(&point);
cout << "address = " << address << endl;
long addressOfLabel = address + 2*sizeof(int);
char* addr = (char*)(addressOfLabel);
cout << "addr = " << addr << endl;
```

```
address = 140720619819404
addr = a
```

2.8 More continuous memory examples

```
struct Point {
   int x;
   int y;
   char label;
};

Point points[10];
points[4].y = 123;
long pointer = (long)points;
int* pointer123 =(int*)(pointer + 4*sizeof(Point) + sizeof(int));
cout << "value: " << *pointer123 << endl;
```

```
value: 123
```

2.9 References

References in C++ are just syntactic sugar which cleans our code a bit. A reference is **not a new variable**. In reality it's just an alias for an existing variable. When we want to create a reference when need to specify the type, and add the `&` symbol next to the type. **It is not the dereference operator**. Internally references are implemented using pointers.

```
int  a = 3;
int& b = a;
b = 5;
cout << "b = " << b << endl;
cout << "a = " << a << endl;
cout << &a << endl;
cout << &b << endl;
```

```
b = 5
a = 5
0x7ffeaedd918c
0x7ffeaedd918c
```

```
void inc_ptr(int* x) {
    (*x)++;
}

void inc_ref(int& x) {
    x++;
}

int main(){
    int a = 3;
    inc_ptr(&a);
    cout << "a = " << a << endl;

    inc_ref(a);
    cout << "a = " << a << endl;
}
```

```
a = 4
a = 5
```

3 Allocating on stack vs heap

3.1 Introduction

Stack and heap are names referring to different areas inside of computer's memory(RAM).

3.1.1 Allocating on stack

- very fast
- quite small amount of space(around 2MB)
- the data gets pushed onto the stack
- you don't have to clear that memory yourself

- things allocated on the stack get cleared when you exit a scope (loop, if, function etc.)
- ideal for local variables and state

3.1.2 Allocating on heap

- you can store vast amounts of data on a heap
- you need to clear the memory yourself(usually)
- you allocate on the heap using the == keyword

3.2 Examples

```
int* get_counter(){
    int counter = 5;
    int *pcounter = &counter;
    cout << "get_counter pcounter = " << pcounter << endl;
    return &counter;
}

int main(){
    int* pcounter = get_counter();
    cout << "main pcounter = " << pcounter<< endl;
    return 0;
}
```

```
get_counter pcounter = 0x7ffca876fcfc
main pcounter = 0
```

Explanation: The `counter` variable got allocated on the stack, and well as the `pcounter` variable. When `get_counter` returned it cleared the allocated memory. Therefore the `pcounter` variable in `main` points to something that simply does not exist anymore.

Solution: The solution is to allocate on the heap which does not get cleared when you exit a scope.

3.3 Allocation on heap example

```
1  int* get_counter(){
2      int* counter = new int();
3      *counter = 5;
4      cout << "get_counter counter = " << counter << endl;
5      return counter;
6  }
7
8  int main(){
9      int* counter = get_counter();
10     cout << "main counter = " << counter<< endl;
11     (*counter)++;
12     cout << "final value = " << *counter << endl;
13     delete counter;
14     return 0;
15 }
```

```
get_counter counter = 0x55f7d6069eb0
main counter = 0x55f7d6069eb0
final value = 6
```

Now that we've allocated the `counter` variable on the stack the example works. **Notice** that we cleared the memory on line 13. **Never** try to use variables which have been deleted(you will get undefined trash).

3.4 Allocating an array

```
int *array = new int[5]{1,2,3,4,5};
cout << array << endl;
cout << array[3]<< endl;
delete[] array;
```

```
0x561ba624feb0
4
```

3.5 `new` keyword

3.5.1 introduction

The `new` keyword is used to allocate memory on the **heap**. When you use `new` you need to specify the datatype(int, string, custom class, etc.) in order

to determine the size of the allocation.

So, when you write `= the standard c library will be called which then in turn will call the OS in order to find 4 bytes of memory(=int` is usually 4 bytes). Once those 4 bytes of memory are found you get a pointer to that memory.

Using the `new` keyword is **often slow.**

3.5.2 usage

```
class Person {
private:
    string m_Name;
public:
    Person(): m_Name("unknown") {}
    Person(const string& name) : m_Name(name) {}
    const string& GetName() const {return m_Name;}
};

int main(){
    int* a = new int;
    Person* p = new Person();
    cout << a << endl;
    cout << p << endl;

    return 0;
}
```

```
0x55b9611b7eb0
0x55b9611b7ed0
```

4 Classes

4.1 What is a class?

A class is a grouping of data and behavior. **Example**: A `Player` class can contain:

- data: player's position, level, inventory, ...
- behavior: move to player to a position, attack an enemy, ...

An **object** is a variable of class type. Objects are also often called instances(of a class).

```cpp
class Player {
public:
    int x,y;
    int speed = 2;
    void Move(int xa, int ya){
        x += xa*speed;
        y += ya*speed;
    }
};

int main(){
    Player player;
    player.x = 0;
    player.y = 3;
    player.Move(2,3);
    cout << player.x << " " << player.y << endl;
}
```

```
4 9
```

4.2 `class` vs `struct`

4.2.1 Introduction

There is really no real difference in functionality. One small difference is visibility. In classes by default all the members are private. In structures however they are by default public.

> The functionality between them does not differ, but how we typically use them does.

The reason why `struct` keyword exists is backwards compatibility with c.

4.2.2 When to use `struct`?

- when we want to some data together into one type
- `struct` should be used when we create for example `DTOs` (Data transfer objects).
- simple groupings of data, with simple methods often for formatting, printing, etc.

4.2.3 When to use `class`?

- whenever inheritance is needed
- complex groupings of behavior and data

4.3 Log class example

```
#include <iostream>

using namespace std;

class Log {
public:
    const int LogLevelError = 0;
    const int LogLevelWarning = 1;
    const int LogLevelInfo = 2;
private:
    int loglevel = LogLevelInfo;

public:
    void SetLevel(int level){
        loglevel = level;
    }

    void Error(const char* message){
        if(loglevel >= LogLevelError)
            cout << "[ERROR]: " << message << endl;
    }

    void Warn(const char* message){
        if(loglevel >= LogLevelWarning)
            cout << "[WARNING]: " << message << endl;
    }

    void Info(const char* message){
        if(loglevel >= LogLevelInfo)
            cout << "[INFO]: " << message <<endl;
    }
};


int main(){
    Log log;
    log.SetLevel(log.LogLevelWarning);
    log.Error("error!");
    log.Warn("warn!");
    log.Info("info");
```

```
}
```

```
[ERROR]: error!
[WARNING]: warn!
```

4.4 Constructor

The constructor is a special method that gets executed whenever an object is created. It is primarily used to initialize memory, and setup the object to work correctly. Constructors can be overloaded(using different parameters) they can also be made private, in that case the object cannot be created.

```
class Point {
private:
    double x, y;
public:
    Point(){
        x = 0;
        y = 0;
    }
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    }
    void print(){
        cout << this->x << ", " << this->y  << endl;
    }
};

int main() {
    Point p; // using the default constructor
    p.print();
    Point a(2,3);
    a.print();
}
```

```
0, 0
2, 3
```

Sometimes you don't want people to create objects of your class:

- all methods are static, the class is just a namespace for related functions
- you're using the singleton pattern

Here are two ways of disabling the constructor

This makes the constructor private.

```
class SingletonClass {
private:
    SingletonClass(){
    }
};
```

This deletes the constructor.

```
class MathFuncs {
    MathFuncs() = delete;
};
int main(){
    MathFuncs mf;
}
```

4.5 Enumeration types

Enums are a way of grouping together constants. One should use enums whenever a certain parameter, return value, etc. has only a few possibilities. Enums allows us to eliminate *magic numbers* by giving clear names to constants. Example of comparison:

```
enum CompareResult {
    LESS_THAN = -1,
    EQUAL,
    GREATER_THAN,
};
```

Since we've set the `LESS_THAN` to -1 C++ will automatically set the values of `EQUAL` and `GREATER_THAN` for us by incrementing the first value(`LESS_THAN`).

This function returns a strict set of values that's why we are using an enum.

```
CompareResult compare(int x, int y){
   if(x == y) return EQUAL;
   if(x > y) return GREATER_THAN;
   else return LESS_THAN;
}
```

Enums works very well with switch statements especially if an enum has many possible values.

```
void printRelation(int x, int y){
    CompareResult res = compare(x, y);
    switch(res){
```

```
        case EQUAL:
            cout << x << " = " << y << endl;
            break;
        case GREATER_THAN:
            cout << x << " > " << y << endl;
            break;
        case LESS_THAN:
            cout << x << " < " << y << endl;
            break;
    }
}
```

Now we can put this all into a test:

```
int main(){
    cout << "compare(2, 3) = " << compare(2, 3) << endl;
    cout << "compare(3, 3) = " << compare(3, 3) << endl;
    cout << "compare(4, 3) = " << compare(4, 3) << endl;
    printRelation(2,3);
    printRelation(3,3);
    printRelation(4,3);
}
```

```
compare(2, 3) = -1
compare(3, 3) = 0
compare(4, 3) = 1
2 < 3
3 = 3
4 > 3
```

As you can see every constant is unique. Enums work very well with switch statements.

By default the underlying type of an enum is int. If you want to change it for example to char to save some memory you can do so by saying:

```
enum CompareResult : char {
    LESS_THAN = -1,
    EQUAL,
    GREATER_THAN,
};
```

When you create variables of type == you do so like this

```
CompareResult eq = EQUAL;
CompareResult lt = LESS_THAN;
```

```
CompareResult gt = GREATER_THAN;
```

4.6 Destructor

Whenever an object is destroyed(the memory allocated for it gets cleared) a special method called **destructor** gets called. Its signature is very similar to the constructor the difference is the ~ prefix. The destructor gets called for both stack and heap allocated objects.

```
class Point {
public:
    double x,y;
    ~Point() {
        cout << "the point " << x << ", " << y << " is being destroyed..."
        ↪  << endl;
    }
    void print(){
        cout << "(" <<x << ", " << y << ")"<< endl;
    }
};

int main(){
    if(true){
        Point p; // stack allocation
        p.x = 2;
        p.y = 3;
        p.print();
    }
    Point* p = new Point; // heap allocation
    p->x = 4;
    p->y = 5;
    p->print();
    delete p;
}
```

```
(2, 3)
the point 2, 3 is being destroyed...
(4, 5)
the point 4, 5 is being destroyed...
```

The destructor can be used for:

- closing opened files

- clearing allocated memory on the heap
- closing database/web connections
- commiting changes to servers/databases
- logging when objects get destoryed

The destructor can be called explicitly, althouth you probably should not do that. But if you want to:

```
Point p;
p.~Point();
Point q = new Point;
q->~Point();
```

4.7 Inheritance

Inheritance is a mechanism which allows to reduce code duplication. This is it's main purpose. Inheritance allows us also to create meaningful hierarchies, so for example(Player is a subclass of Character, and Character is a subclass of GameEntity). However **inheritance create very strong coupling** often using **composition** creates solutions which are more extensible and easier to modify.

```
class GameEntity {
public:
    float x,y;
    void move(int xa, int ya){
        x += xa;
        y += ya;
    }
};

class Player : public GameEntity {
public:
    const char* name;
    void printName(){
        cout << name << endl;
    }
};

void moveToOrign(GameEntity& ge){
    ge.x = 0;
    ge.y = 0;
}
```

```cpp
int main(){
    Player p;
    p.name = "Bob";
    p.printName(); // nothing new
    // but we can use everything GameEntity has
    p.x = 2;
    p.y = 3;
    p.move(1,2);
    cout << p.x << ", " << p.y << endl;
    moveToOrign(p);
    cout << p.x << ", " << p.y << endl;
}
```

```
Bob
3, 5
0, 0
```

The objects of the class `Player` have a type of `Player`, unsurprisingly. But they also have the type `GameEntity` meaning whenever `GameEntity` can be used a `Player` can be used as well.

4.8 Polymorphism

4.8.1 Introduction

Polymorphism allows us to create families of **interchangeable** objects. In such a way that a user does not have to be concerned about the specific details of the implementation. Polymorphism allows us to create systems which can be extendend not by modyfing existing code but rather by just adding new code. It allows allows us to **invert dependencies**, so for example a sql database will depend on business rules, and not the other way around.

4.8.2 The `virtual` methods

C++ does not have interfaces like Java or C#. Instead we use the `virtual` methods. Let's take a look why we even need them.

Here we have two classes, one the `Entity` is the base class. Why want every object to able to say what it is.

```cpp
class Entity {
public:
    void whoami(){
        cout << "i am entity" << endl;
```

```
    }
};

class Player: public Entity {
public:
    void whoami(){
        cout << "i am player" << endl;
    }
};
```

Let's see that that works

```
void identify(Entity& e) {
    e.whoami();
}

int main(){
    Entity e;
    Player p;
    cout << "in main:" << endl;
    e.whoami();
    p.whoami();

    cout << "through the method" << endl;
    identify(e);
    identify(p);
}
```

```
in main:
i am entity
i am player
through the method
i am entity
i am entity
```

You can see that once we in `identify` functions the whole thing breaks down. Essentially our `Player` is cast down to an `Entity`, and simply executes it's methods. To fix that we must specify the `whoami` method as `virtual`. You can provide a default implementation for it, but you don't have to. You can also say

```
virtual methodName(int someParameters) = 0;
```

This kind of method is known as **pure virtual**. **You cannot instantiate classes which have those methods**. They are like abstract classes

or interfaces in other languages. Now comes the *fixed* example:

```
1  class Entity {
2  public:
3      virtual void whoami() = 0;
4  };
5  class Player: public Entity {
6  public:
7      void whoami() override {
8          cout << "i am player" << endl;
9      }
10 };
11
12 void identify(Entity& e) {
13     e.whoami();
14 }
15
16 int main(){
17     Player p;
18     cout << "in main:" << endl;
19     p.whoami();
20     cout << "through the method" << endl;
21     identify(p);
22 }
```

```
in main:
i am player
through the method
i am player
```

Now we finally have the desired behavior. Notice that now on line 3 we mark the `whoami` method as pure virtual. Also notice the addition of `override` on line 7 this is not necessary but improves code quality.

4.9 Visibility(private, protected, public)

4.9.1 Example

```
class Entity {
public:
    double x, y;
private:
    const char* id;
protected:
    void changeId(const char* newid) {
        id = newid;
```

```
    }
};

class Player: public Entity {
    Player createCopy() {
        Player copy;
        copy.x = x;
        copy.y = y;
        copy.changeId("random string...");
        return copy;
    }
};
```

4.9.2 `private`

Private members of a class are accessable **only within that particular class**. So the variable `id` can only be accessed directly within `Entity` class.

```
Entity e;
e.id = "hello";  // throws an error because id is private
```

Even classes which inherit from `Entity` **cannot** access private variables.

4.9.3 `protected`

Protected members are similar to private ones. They are not visible to the outside world, however protected members can be accessed from subclasses. In the `Player` class we can access `changeId` method, but not in `main` for example.

```
Player p;
p.changeId("new id"); // throws an error
```

4.9.4 `public`

Public members are the simplest. Anyone can access them.

```
int main(){
    Player p;
    // this is allowed becuase x and y are public
    p.x = 2;
    p.y = 3;
    cout << p.x << " "<< p.y << endl;
}
```

```
2 3
```

4.9.5 TODO `friend` mechanism

4.10 Constructor member initializer list

4.10.1 Overview and example

Constructor member initializer list is a way of **assigning variables** in the constructor in a **more concise** manner.

This is the first way of initializing our variables, it's a standard in a lot of other programming languages:

```
class Person {
public:
    const char* m_Name;
    Person(){
        m_Name = "Unknown";
    }
    Person(const char* name){
       m_Name = name;
    }
};
int main(){
    Person p;
    cout << p.m_Name << endl;
    Person john("John");
    cout << john.m_Name << endl;
}
```

```
Unknown
John
```

In C++ we can however do something like this:

```
class Person {
public:
    const char* m_Name;
    int m_Score;
    Person(): m_Name("Unknown"), m_Score(0) {
    }
    Person(const char* name): m_Name(name){
    }
};
```

```
int main(){
    Person p;
    cout << p.m_Name << endl;
    Person john("John");
    cout << john.m_Name << endl;
    cout << p.m_Score << " " << john.m_Score << endl;
}
```

```
Unknown
John
0 121605768
```

Remeber to initialize variables **in the same order** they are declared.

4.10.2 Differences

Constructor member initializer list actually have a small performance benefit. When we use them the desired object is **created only once**. An example will show it:

1. First we create a `Friend` class which prints some stuff when it's created

```
class Friend {
public:
    Friend(){
        cout << "defualt Friend constructor" << endl;
    }
    Friend(const char* name) {
        cout << "created Friend with name " << name  << endl;
    }
};
```

1. Then we create our `Person`

```
class Person {
public:
    Friend m_Friend;
    Person(){
        m_Friend = Friend("Alfred");
    }
    Person(const char* name): m_Friend(name) {
    }
};
```

2. Now let's put it all to a test, you can see we use constructor initializator lists in the second constructor

```
int main(){
    cout << "john example start" << endl;
    Person john; // using the default constructor
    cout <<  "john example end" << endl << endl;;

    cout << "bob example start" << endl ;;
    Person bob("joe"); // using the second constructor
    cout <<  "bob example end" << endl;
}
```

```
john example start
defualt Friend constructor
created Friend with name Alfred
john example end

bob example start
created Friend with name joe
bob example end
```

We can clearly see that using the second constructor of `Person` creates **only one** object of `Friend`.

4.11 Implicit conversion

Say we have a class:

```
class Point1d {
public:
    double x;
    const char* label;
    Point1d(double x): x(x), label("NONE") {
    }
    Point1d(const char* label): x(0), label(label){}
};
```

Now we have a couple of ways of initializing it:

```
Point1d a(2);
Point1d a("a");
Point1d a = Point1d(2);
Point1d a = Point1d("a");
```

all of which are perfectly normal, we can do however something like this:

```
int main() {
    Point1d a = 3.14;
    cout << "x = " << a.x << " label = " << a.label << endl;
    Point1d b = "b";
    cout << "x = " << b.x << " label = " << b.label << endl;
}
```

```
x = 3.14 label = NONE
x = 0 label = b
```

Which is a bit suprising to say the list. When C++ sees something like `= it first recognizes the type on the right-hand side of the ==` operator(in this case it's `double`) and since `Point1d` has a constructor which only requires a single `double` it calls it and returns the object.

What this means is that code like this, works:

```
Point1d moveToN(Point1d p, double x) {
    Point1d copy = Point1d(p.label);
    copy.x = x;
    return copy;
}

int main(){
    Point1d a = moveToN("a", 3.14);
    cout << a.label << " " << a.x << endl;
}
```

```
a 3.14
```

4.12 `explicit` keyword

The `explicit` keyword disable implicit conversion.

```
class Person {
public:
    int m_age;
    const char* m_name;
    Person(int age): m_age(age), m_name("--none--"){}
    Person(const char* name): m_age(-1), m_name(name){}
```

```
};

int main() {
    Person p = 42; // this works
    Person john = "john"; // this throws an error
}
```

4.13 Operators

Operators are symbols like `+ * << & ^ new delete , {` and so on. Operators are not some weird magic, instead they are just functions with some fancy syntax.

Operator overloading is essentially giving/changing behaviour to operators.

```
class Vector2 {
public:
    double x, y;
    Vector2() : x(0), y(0){}
    Vector2(double x, double y) : x(x), y(y){}
    Vector2 operator+(const Vector2& other) const {
        return Vector2(x+other.x, y+other.y);
    }
    Vector2 operator*(const Vector2& other) const {
        return Vector2(x*other.x, y*other.y);
    }
    Vector2 operator*(const double& factor) const {
        return Vector2(x*factor, y*factor);
    }
    bool operator==(const Vector2& other) const {
        return x == other.x && y == other.y;
    }
    bool operator!=(const Vector2& other) const {
        return !(*this == other);
        // you could also do
        // return !(operator==(other));
    }
};

// the "equivalent" of toString() in other languages
std::ostream& operator<<(std::ostream& stream, const Vector2&vec) {
    stream << "(" << vec.x << ", " << vec.y << ")";
    return stream;
}

int main(){
    Vector2 pos(1,2);
```

```
    Vector2 speed(3, 4);
    Vector2 newpos = pos+speed;

    cout << "newpos: " << newpos.x << ", " << newpos.y << endl;
    Vector2 newspeed = speed*2.5;
    cout << "newspeed: " << newspeed.x << ", " << newspeed.y << endl;
    cout << newpos << endl;
    cout << (newpos == Vector2(4,6)) << endl;
    cout << (newpos != Vector2(4,6)) << endl;
}
```

```
newpos: 4, 6
newspeed: 7.5, 10
(4, 6)
1
0
```

4.14 `this` keyword

`this` is only available in member functions(methods of a particular class). `this` is a pointer to the current object instance that the method belongs to.

```
class Entity {
public:
    int x, y;

    Entity(int x, int y) {
        (*this).x = x; // using derefrencing
        this->y = y; // using the arrow operator
    }

    int getX() const {
        this->x = 3; // not allowed because of const
        return x;
    }

}
```

In a *normal* method(constructor, methods without `const` keyword) the type of `this` is equal to `= that means you can change the underlying data of the pointer but not the pointer itself. In a =const` method the type of this is `= meaning you cannot alter the pointer nor the underlying data. Templates are similar to generics, but more powerful. Template is a way for programmers to allow the compiler to /write code for`

them/. Creating a template is like creating a blueprint with some parameters, once you use your template(say something like {{{results(=)}}}) the compiler will take the parameters == and use them to write the code.

```cpp
template<typename T>
void print(T value) {
    cout << value << endl;
}

template<typename T, int N>
class Array {
private:
    T m_Array[N];
public:
    int getSize() const {
        return N;
    }
};

int main() {
    print<string>("hello");
    print<const char*>("henlo");
    print(3.14);
    print(2);
    Array<int, 5> array;
    cout << array.getSize() << endl;
}
```

```
hello
henlo
3.14
2
5
```

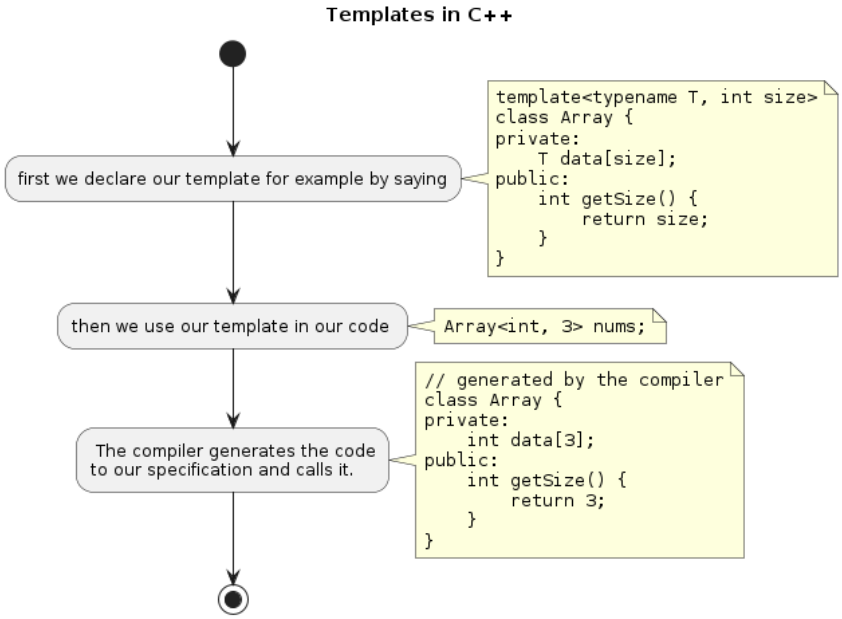
4.15 Templates

4.15.1 Introduction

- Templates are similar to generics in other programming languages, but they are much more powerful.
- Templates allow **the compiler to write code for us** under the rules which we specify.

- You can think of a template as a blueprint with some blanks, those blanks can be types, numbers, strings, or anything.
- When we use a template (call a function, instantiate a class) the compiler notices that and generates the code for us by essentially filling in the blanks in the template.
- Template parameters can be specified explicitly(for example `std::array<int,3>`) or the can be inferred from the usage.

Working with templates can be describe using a simple diagram:



4.15.2 Example of a template function

Say we what to have a function which prints limits of a signed integer number(char, short, int, long, long long). What we could do is write the function 5 times by overloading it, or we can use templates:

```
template<typename num_type>
void printLimits() {
    int amountOfBits = sizeof(num_type)*8;
    long long max = (1 << (amountOfBits-1)) - 1;
    long long min = -(1 << (amountOfBits-1));
    cout << "max: " << max << endl;
```

```
    cout << "min: " << min << endl;
}

int main() {
    printLimits<char>();
    printLimits<short>();
    printLimits<int>();

}
```

```
max: 127
min: -128
max: 32767
min: -32768
max: 2147483647
min: -2147483648
```

4.15.3 Example of a template class

This shows how we can write a *modern* array class, which is allocated on the stack.

```
template<typename T, int size>
class Array {
private:
    T data[size];
public:
    int getSize() {
        return size;
    }
    T& operator[](int index) {
        if(index < 0 || index >= size){
            throw "index out of bounds";
        }
        return data[index];
    }
};

int main() {
    Array<const char*, 3> names;
    names[0] = "Peter";
    names[1] = "Joe";
    cout << names[0] << endl;
    cout << names[1] << endl;
    cout << "last one equal to null? "  << (names[2] == nullptr) << endl;
```

```
}
```

```
Peter
Joe
last one equal to null? 0
```

5 Functions

5.1 `static` variable inside of a function

The `static` keyword has lots of meanings which all depend on where the keyword is placed. If we create a variable in a function using the `static` keyword the variable will not be deleted after the function returns. It will essentially live forever(until the program exits).

```
int next_integer() {
    static int a = -1;
    a++;
    return a;
}

int main(){
    cout << next_integer() << endl;
    cout << next_integer() << endl;
    cout << next_integer() << endl;
    cout << next_integer() << endl;
}
```

```
0
1
2
3
```

This usage of `static` keyword creates a *kind of global variable*. The good thing is that the scope of this variable is this function, this can prevent invalid modification of it.

5.2 Function pointers

5.2.1 Introduction

- Functions contain cpu instructions, which are stored in the memory, just like typical `int` or `double` variables. Functions **are also data and are stored as such**. Because of that we can create pointers to them.
- Function pointers allow us to **inject** some behavior into another function.
- That means we can create very useful functions, which change their behavior depending on what function we passed to them
- we can even return a pointer to a function from a function

5.2.2 Code Examples

1. Assigning a function pointer to a variable
    - often we use `auto` to create function pointers, because the types are *weird* looking
    - also we often use `typedef` to create an alias for a function pointer type like we did on line 12

```
1  void HelloWorld() {
2     cout << "hello world" << endl;
3  }
4  int main() {
5     void(*function)() = HelloWorld;
6     auto func = HelloWorld;
7
8     function();
9     func();
10    func();
11
12    typedef void(*HelloWorldFunction)();
13    HelloWorldFunction fun = HelloWorld;
14    fun();
15 }
```

```
hello world
hello world
hello world
hello world
```

2. Accepting a function pointer as an argument Here comes an example of a higher-order function, which takes a function pointer as an argument

```
1  int* map(int* array, int size, int(*mapper)(int)) {
2      int* mapped = new int[size];
3      for(int i = 0; i < size; i++) {
4          mapped[i] = mapper(array[i]);
5      }
6      return mapped;
7  }
8
9  int inc(int x) {
10     return x + 1;
11 }
12
13 int main() {
14     int nums[3] = {2,3,4};
15     int* incremented =  map(nums, 3, inc);
16
17     for(int i=0; i<3; i++ ){
18         cout << nums[i] << " -> " << incremented[i] << endl;
19     }
20     delete[] incremented;
21 }
22
```

```
2 -> 3
3 -> 4
4 -> 5
```

Lambda functions are usually small and are often used, where a function like `inc` is needed. Using lambda functions the line 15 would look like this:

```
int* incremented =  map(nums, 3, [](int a) {return a+1;});
```

3. Returning a function pointer

```cpp
int add(int x, int y){
    return x + y;
}

int multiply(int x, int y) {
    return x*y;
}

typedef int(*BinaryIntFunc)(int, int);
BinaryIntFunc getBinaryFunc(char symbol) {
    return (symbol == '*') ? multiply : add;
}

int main() {
    int a = 2, b = 3;
    auto op = getBinaryFunc('*');
    cout << op(a,b) << endl;
    cout << getBinaryFunc('+')(a,b) << endl;
}
```

```
6
5
```

5.2.3 Lambda

- Lambda is an anonymous function.
- Lambdas are often treated like *typical* variables like `int double` and so on.
- We can use a lambda whenever a function pointer is required
- Lambdas are a way for us to define a usually small function, and treat it like a typical variable.

```cpp
#include <iostream>
#include <functional>

typedef std::function<int(int,int)> binary_op;

binary_op returnFunc(){
    int bias = 3;

    return [=](int x, int y) mutable -> int {
```

```
        bias = 4;
        return x + y + bias;
    };
}

int main() {
    binary_op f = returnFunc();
    cout << f(1,2) << endl;
    cout << 'f' << 'u' << 'c' << 'k' << endl;
}
```

```
7
fuck
```

6 Strings

6.1 Introduction

A string is an array of characters. The important detail is that the last element of the array must be the **null terminator** (`=`). `The special string syntax({{{results(=)}}})` adds the null terminator for you.

6.2 Various ways of creating text

```
1 char hello[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
2 cout << hello << endl;
3 const char* world = "world!";
4 cout << world << endl;
```

```
hello
world!
```

Often in people's source code you can see `= as we did on line [[(world-variable-declar)]]` `the {{{results(=)}}}` keyword is used to ensure the string of text is read only, and it can prevent some nasty bugs.

6.3 `std::string`

The `std::string` is an layer of abstraction on top of character arrays, it provides lots of useful methods. Creating one is very simple

```cpp
string message = "hello world!";
cout << message << endl;
// you can also use message.append
message += "new text";
// number of bytes excluding '\0'
cout << "message length = " << message.length() << endl;
// we can change individual bytes
message[0] = 'H';
// this allows us to conver the string to c type
const char* cmessage = message.c_str();
cout << "cmessage " << cmessage << endl;
```

```
hello world!
message length = 20
cmessage Hello world!new text
```

6.4 Converting to and from strings

6.4.1 From `int` to `string`

```cpp
int x = stoi("  123 ");
cout << x*2 << endl;
```

```
246
```

6.4.2 List of conversion functions

function name	description
`stoi`	convert to `int`
`stod`	convert to `double`
`stold`	convert to `long double`
`stof`	convert to `float`
`stol`	convert to `long`
`stoll`	convert to `long long`
`stoul`	convert to `unsigned long`
`stoull`	convert to `unsigned long long`

6.4.3 What if conversion fails?

If you try to convert something like `"a2bc"` to an `int` we will get an exception we have to catch it otherwise the program exits. The specific exception in that case is the `std::invalid_argument`.

```
string notint = "a2bc";
try {
    int x = stoi(notint);
    cout << "x = " << endl;
} catch(invalid_argument& exception) {
   cout << "oh no error occured " << exception.what() << endl;
}
```

```
oh no error occured stoi
```

There is also a possibility that a value stored in the string can be out of range for `int`. In that case the conversion function throws `std::out_of_range` error.

```
string verybignum = "12345678912345678";
try {
    int x = stoi(verybignum);
    cout << "x = " << x << endl;
} catch(out_of_range exception) {
    cout << exception.what() << endl;
}
```

```
stoi
```

Those example show the usage of `stoi` function, the same rules apply to all the other conversion functions.

6.5 String literals

String literals allows us to work with character arrays in a *sane* manner. Apart from having to write text like this: `= string literals add the null terminator('\0'=)` for us.

So this kind of code:

```
const char text[5] = "hello";
```

throws an error because it's true that `"hello"` has 5 characters but we also need the null terminator, to signify the end of text. So use this:

```
const char text[6] = "hello";
```

or better still, this:

```
const char text[] = "hello";
```

6.6 C language string functions

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
    const char text[] = "hello!";
    // number of bytes without '\0'
    cout << strlen(text) << endl;
    // total number of bytes
    cout << sizeof(text) << endl; //
}
```

```
6
7
```

6.7 How are strings stored?

Text which has a type of `= is stored in a readonly part of the memory which means you cannot really change it. In other words this can and probably will give a =SEGFAULT`.

```
char* text = "hello!";
text[2] = 'a';
cout << text << endl;
```

If you want to modify strings you just need to use character arrays.

```
char text[] = "hello!";
text[2] = 'a';
cout << text << endl;
```

```
healo!
```

This works just fine. However we **don't** modify the text variable here. Because string literals are always stored in readonly memory, we create a new string and assign it to `text`.

6.8 Different types of `char`

```
// wchar takes 2 bytes on Win, 4 on Linux
const wchar_t* first = L"Hello "; // wstring
// char16 takes 2 bytes
const char16_t* second = u"world"; // u16string
// char32 takes 4 bytes
const char32_t* third = U" more text"; // u32string
```

```
0x55c865f11004
```

6.9 Appending strings the easy way

```
string fullname = std::string("Piotr") + " Karamon";
cout << fullname << endl;

using namespace std::string_literals;
string fullname2 = "Piotr"s + " Karamon";
cout << fullname2 << endl;
```

```
Piotr Karamon
Piotr Karamon
```

6.10 Raw strings

Raw strings are useful when creating regular expressions, urls or any other text which means to be interpreted literally without escaping characters(turning `'\n'` into a newline and so on).

```
const char* regex = R"(.*?[\t\n ]+.*?)";
cout << regex << endl;
```

```
.*?[^ ]+.*?
```

7 std::array

The `std::array` provides a more modern way of dealing with arrays, which is similar to the way things are in langauges like Java or C#. Probably the nicest thing is the addition of `.size()` method which means we don't have to track the size of the array on our own.

```
std::array<int, 4> numbers;
numbers[0] = 3;
numbers[2] = 9;
numbers[3] = 12;
cout << "first element " << numbers[0] << endl;
cout << "first element " << numbers.front() << endl;
cout << "last element " << numbers[numbers.size() -1] << endl;
cout << "last element " << numbers.back() << endl;
```

```
first element 3
first element 3
last element 12
last element 12
```

Modern iteration is also a big plus

```
std::array<int, 4> numbers = {1,2,3,4};
for(int& n : numbers){
    cout << "n " << n << endl;
}
```

```
n 1
n 2
n 3
n 4
```

8 const keyword

8.1 const with simple variables

The `const` keyword really is just a promise that something will not change. It strongly signifies that something should never change. The `const` keyword however does very little when it comes to machine code.

```cpp
// never change unless god help you, you're using Fahrenheit
const int WATER_BOILING_TEMPERATURE = 100;
cout << WATER_BOILING_TEMPERATURE << endl;
```

```
100
```

Something like the following, will not allows us to compile our program

```cpp
WATER_BOILING_TEMPERATURE = 123;
```

8.2 `const` with pointers

8.2.1 Introduction

When it comes to pointers you actually a few options. You can either make the address or the variable constant independently. By default you can change both the address of a pointer as well as the underlying data.

```cpp
int b = 3, c = 4;

int* a = &b;
cout << "a = " << a << endl;
cout << "*a = " << *a << endl;

a = &c;
*a = 5;
cout << "a = " << a << endl;
cout << "*a = " << *a << endl;
```

```
a = 0x7ffdef9dbf08
*a = 3
a = 0x7ffdef9dbf0c
*a = 5
```

8.2.2 Making underlying data constant

But using `=, meaning we use the =const` keyword before the type we make **the underlying data** constant. Meaning we can only point this variable to different `ints` but we cannot change the values of those `ints`.

So this works just fine:

```
int b = 3, c =4;
const int* a = &b;
a = &c; // this is allowed
cout << "a = " << a << endl;
cout << "*a = " << *a << endl;
```

But this will give you a compile error:

```
*a = c;
```

8.2.3 Making the address constant

On the other if we want a pointer to always point to the same variable we can use the `const` keyword after the type.

```
int b = 3, c = 4;
int * const a = &b;
*a = 5;
cout << "*a = " << *a << endl;
cout << "b = " << b << endl;
```

```
*a = 5
b = 5
```

If you try something like

```
a = &c;
```

you will get an error.

8.2.4 Making everything constant

To make both the address and the underlying data `const` we use

```
const int* const a = &b;
```

8.3 `const` in class methods

8.3.1 General overview

One can add the `const` keyword just before opening bracket of a method. This means that the method **will not** modify the data inside of the class. Such addition of the `const` keyword tells users of your class that no data inside of the it will change. Data modification can be **especially** important in concurrent/parallel systems.

```
int GLOBAL_INT = 123;
class Point {
public:
    double x;
    double y;
    void moveToOrign() const {
        // this will not let you compile
        x = 0;
        // so will this
        y = 0;
        // this however is okay
        GLOBAL_INT = 321;
    }
};
```

8.3.2 `mutable` keyword

If we mark a method in a class using the `const` keyword, we cannot modify any of its members, apart from those marked with `mutable` keyword. That's what `mutable` does, it lets you change specified variables even in `const` methods.

```
class Person {
public:
    mutable const char* name;
    int age;
    void changeName(const char* newname) const {
        name = newname;
    }
};

int main() {
    Person p;
    p.name = "Bob";
    p.age = 42;
    p.changeName("Joe");
    cout << p.name << endl;
}
```

```
Joe
```

The `const` and `mutable` keywords are also important when we use functions which accept constant references. Like this:

```
void prettyPrintPerson(const Person& p) {
    ...
}
```

In `prettyPrintPerson` function we can **only** use methods marked with `const` keyword, because *technically* they ensure that the value won't be modified(apart from variables marked with `mutable`).

9 Ternary operators

Ternary operator allows us to create more concise code but replacing `if-else` blocks with simple expressions. They are useful when the condition and possible outcomes are simple. One should not use the ternary operator if the logic is complex. **Never use nested ternary operators** because they are an abomination. The ternary operator has the following form

```
(condition) ? (what-to-return-if-true) :
↪   (what-to-return-if-false)
```

Some examples:

```
void printMessage(int age){
    const char* message =  (age >= 18) ?
        "you can buy beer" : "you can buy an orange juice";

    cout << "message: " << message << endl;
}

int main(){
   printMessage(17);
   printMessage(18);
   printMessage(19);
}
```

```
message: you can buy an orange juice
message: you can buy beer
message: you can buy beer
```

Now comes a demonstration which shows why nested ternary operators are so horrible:

```
int age = 16;
const char* message = age > 18 ?
```

```
    "you are an adult"
    : (age >= 13) ? "you are a teenager" : "you are a child";

cout << message << endl;
```

```
you are a teenager
```

Even though I tried to make it readable it's still a mess.

10 Macros

Using macros means using preprocessor directives such as `#define` to automate some code writing, basically it's a fancy find and replace.

```
#include <iostream>
#define HELLO cout << "hello world" << endl;
using namespace std;
int main() {
    HELLO;
}
```

```
hello world
```

Within that snippet of code, whenever the compiler encounters the symbol `HELLO` it replaces it with `cout << "hello world" << endl`

Macros also accept arguments

```
#include <iostream>
#define LOG(x) std::cout << x << std::endl;
int main() {
    LOG("hello world");
}
```

```
hello world
```

A good example of macros, is logging we want to log things only in development

```
// usually we don't define that in source files
// but in configuration files etc.
#define APP_DEBUG 1
#if APP_DEBUG == 1
    #define LOG(x) std::cout << x << std::endl;
```

```
#else
    #define LOG(x)
#endif

int main() {
    LOG("hello");
}
```

```
hello
```