C++ Fundamentals

1. Language Basics

    - ✓ Variables: Fundamental types
    - ✓ Variables: User defined types
    - ✓ if-else, while, do-while, for, switch
    - ✓ Functions
    - ✓ Operators

2. Templates
    - ✓ Template Function
    - ✓ Template Class
    - ✓ Template Specialization

3. Indirection

    - ✓ Pointers
    - ✓ References
    - ✓ Polymorphism
    - ✓ Memory Management

Contents

Modern C++

- https://isocpp.org – C++ Timeline
- https://en.cppreference.com/

- String
- Collection
- Smart Pointer
- File and Screen IO

Building Exe:

C++ can be used to create a huge variety of applications

- Phone and other client apps (including Windows) including games
- Console applications
- Services
- Servers
- Libraries

Console applications are the simplest

- Write text to the screen
- Read text from the keyboard
- No graphics, no controls

Fundamental Types

C++ is a strongly typed language
- Variables can hold only certain types of values
- Must be declared before they're used, and can't change type
- "The compiler is your friend"

Fundamental types built into the language
- Numbers, boolean, single characters

User defined types
- Strings, dates, business objects
- Structs and classes

User defined types are full participants in the language

Fundamental Types

Integers short, long, int	**Real numbers** float, double
Character char, unsigned char	**Boolean (true/false)** bool

Auto

Asks the compiler to deduce the type

Variable is still strongly typed

Useful for ugly declarations

```
int i1 = 1;

int i2;
i2 = 2;

int i3(3);

int i4{ 4 };


float f = 2.2f;

double d = 3.3;

bool b = true;

char c = 'c';
```

```
auto a1 = 1;
auto a2 = 2.2;
auto a3 = 'c';
auto a4 = "s";
auto a5 = true;
auto a6 = 3L;
auto a7 = 1'000'000'000'000;
auto a8 = 0xFF;     //255
auto a9 = 0b111;    //7
```

Casting

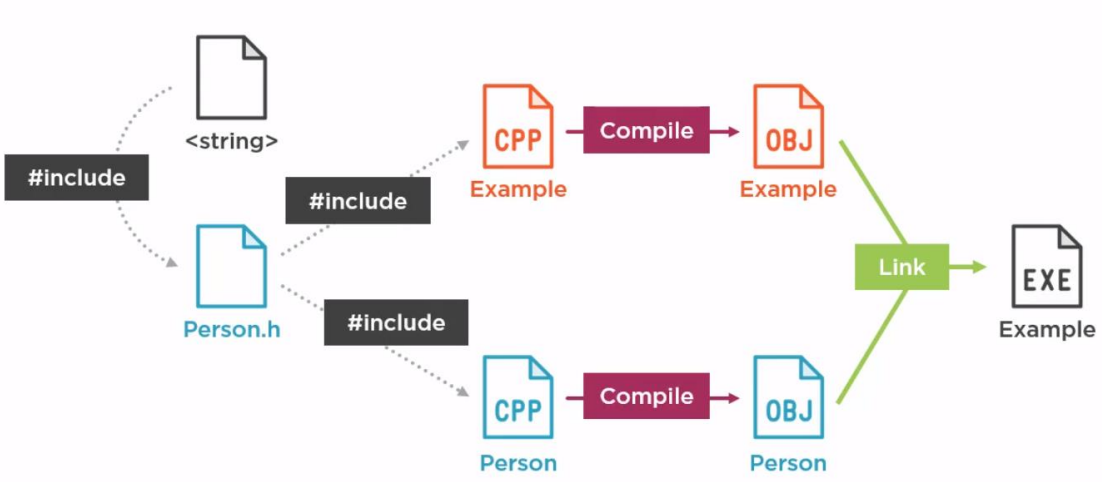
Compiler will convert types	By casting, you make your intention clear	This can backfire
Always use safe casts	**Suffixes to show type of a literal**	

```
int i5 = static_cast <int> (d1);
a1 = static_cast<int> (a2);
```

User Defined Types

You can make the compiler a promise	Linker will ensure it is kept	#include only a convenience	Put each class in its own file

#include pastes one file into another



Class keyword

private and public sections

Declare like fundamental types

Access members with dot .

```
class Person{
private:
            std::string firstname;
            std::string lastname;
            int arbitrarynumber;
public:
            std::string getname();
};
```

```
#include “Person.h”

std::string Person::getname(){

      return firstname + “ ” + lastname ;
}
```

Member variables are private generally

Member functions are public generally

Class declaration may not contain function code, just declaration

- Object – Instance of Class
- Has own copy of member variables
- Member function operate on member variables and arguments
- Initialized with **Constructor**

```
class Person{
      Person(std::string first,
            std::string last,
            int arbitrary);
}
```

```
Person::Person( std::string first,
             std::string last,
             int arbitrary)
            :
firstname(first),
lastname(last),
arbitrarynumber(arbitrary)

{

}
```

Constructor

Uses same name as class name.
Takes arguments which have type and value
Constructors do not return value.
Can have 0 arguments also.
Constructor with 0 argument is **Default Constructor**.
When no constructor declared, compiler adds one default constructor.
If at least one constructor declared, mandatory to add one default constructor too.

Default Constructor

Type 1: Person ( ) { }
Type 2: **Person ( ) = default;**

```
class Person {
private:
      std::string firstname;
      std::string lastname;
      int arbitraryNumber;
public:
      Person(std::string first,
            std::string last,
            int arbitrary);
      Person();
      ~Person();
      std::string getName();
};
```

Destructor

Used to free resources
Destructor does not take arguments
Destructor does not have return value.

Destructor name = ~classname.

```
Person::~Person(){

}
```

Scope

- Object Declared -> Constructor is called

- Memory is allocated for the object
- Memory is on the stack
- So, Objects have stack semantics

- Object has scope -> within block { }
- As control reaches close brace, memory is freed, destructor is called.

- Constructor-Destructor pair concept -> Resource Acquisition Is Initialization (RAII)
- Pattern -> acquire some resource in constructor, release in destructor.
- Prevents dangling resources.


```
int main() {
	Person p1("Steve", "Jobs", 123);
	{
		Person p2;
	}
	std::cout << "after innermost block" << std::endl;
	std::string name = p1.getName();

	return 0;
}
```

struct

User defined type

- struct – default is public
- class – default is private
- union – default is public

struct can have member data
struct can have member function
struct can have constructor, destructor

In c++ if public and private are explicitly used, there is no difference between struct and class.

Use struct only to group data

Use class to group both data and function

Namespace

Namespace groups classes/ modules together

Prevent name collisions

Separate from class name with ::

```
using std::cout;
using std::endl;
```

Never use "using" statement in header file. Always call full name in .h files.

Inheritance

Object Oriented Design

Derived class – Add or override member variables/functions

```
class Tweeter :
    public Person
{
private:
    std::string twitterhandle;
public:
    Tweeter(std::string first,
        std::string last,
        int arbitrary,
        std::string handle);
    ~Tweeter();
};
```

```
Tweeter::Tweeter(std::string first,
        std::string last,
        int arbitrary,
        std::string handle)
    :
    Person(first,last, arbitrary),
    twitterhandle(handle)
{
    std::cout << "constructing tweeter" <<
        twitterhandle << std::endl;
}
```

Sequence of constructor-destructor

1. Constructor of base class
2. Constructor of derived class
3. Destructor of derived class
4. Destructor of base class

Enum

Give names to set of constants

Starts at value 0

Names have to be unique

```
enum Status{
      Pending,
      Approved,
      Cancelled
}
```

Or

```
enum Status{
      Pending = 1,
      Approved = 7,
      Cancelled
}
```

Scoped Enums

Allows underlying type other than int

Names don't have to be unique

Use fully qualified names

Should use scoped enums always

```
enum class FileError {
      notfound,
      ok
};

enum class NetworkError{
      disconnected,
      ok
};
```

PreProcessor

- Line starts with #
- Control what is compiled
- Including header files makes header files compiled right there
- Can also be used to compile slightly different code, e.g. debug build.

#pragma once

- Technically a non-standard one
- All the major compilers support it
- Include whatever to include and use #pragma once as include-guard

Flow of Control

Flow of control changes, e.g. If, while, Function call, return.

If, Immediate If, Switch

For, While, do-while

Break, Continue, Goto

If

```
if (x > y ){
      cout << "x is larger";
}
else {
      cout << " x is smaller";
}
```

For:

```
for (int i = 1; i < 10; i++){
      cout << i;
}
```

While

```
int i = 2;
while ( i < x/i){
      int factor = x / i ;
      if ( factor*i == x)
            break;
      i = i + 1;
}
```

Switch


```
switch(x){
      case 1: break;
      case 2: break;
      case 3: break;
      default: ;
}
```

- Some compilers will warn if fall through is put.
  Fall through: putting multiple case statements without break within switch.
  In that case, we need to put **special compiler-attribute** to mention that it is deliberate.

- In C++17 a new capability has been added to restrict scope to the switch itself.

```
switch( Thing t = somefunction(stuff); t.getStatus() ) {

      case value1: // . . . some action . . .
                   break;

      case value2: // . . . some action . . .
                   break;
}
```

Initializer clause

t will go out of scope when switch finishes.

Immediate If / Ternary Operator

```
result = some condition ? 7 : 302 ;
```

Functions

Free Functions – Function that is not part of any class

- Pass arguments by value, Pass arguments by reference
- Pass by const reference – value cannot be changed

```
bool IsPrime(int const& x) { }
```

```
It is possible to return a reference

int& BadFunction(){
      int a = 3;
      return a;
}
```

This returns dangling reference.

- Return by value is good safe way.

Member Functions

Declare in header file, Implement in .cpp file

Can implement "inline" where declared

Mark as "const" unless you can't.

```
string Person::GetName() const
{
    return firstname + " " + lastname;
}
```

```
class Person
{
private:
    std::string firstname;
    std::string lastname;
    int arbitrarynumber;

public:
    Person(std::string first,
        std::string last,
        int arbitrary);
    ~Person();
    std::string GetName() const;
    int GetNumber() const {return arbitrarynumber;}
    void SetNumber(int number) {arbitrarynumber = number;}

};
```

Understanding Error Messages

compiler	linker
Have you declared that function?	Have you implemented that function?
usually in a .h	usually in a .cpp
making a promise	keeping the promise

Operators

Arithmetic: +, - , *, /

Shortcuts: +=, -=, *=, /=

Increment and Decrement: i++, ++i, i--, --i

Module: %

No Exponential operator

Comparison: < > <= >=

==, !=

&&, | | ----- uses shortcut

!

Bitwise: &, | , ^ , ~

Bit-shift operator: <<, >>

Operator Overloading

int i = j + 3 ;

Order newOrder = oldOrder + newItem;

Write a function that defines the operator

Usually a member function, occasionally a free function

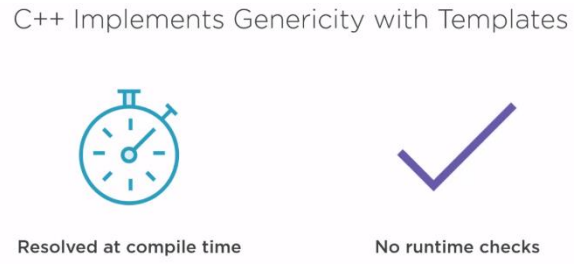
Writing an Overload

MyObject < Something	Something < MyObject
`bool MyClass::operator<(OtherType something)`	`bool operator<(OtherType something, MyClass mc)`
Can operate on two MyClass objects or different types	Free function Use MyClass public functions Or be declared a friend

```
class Person{
private:
	std::string firstname;
	std::string lastname;
	int arbitraryNumber;
	friend bool operator<(int i, Person const& p);
public:
	Person(std::string first, std::string last, int arbitrary);
	~Person();
	std::string GetName() const;
	int GetNumber() const { return arbitraryNumber;}
	void SetNumber(int number) { arbitrarynumber = number;}
	bool operator<(Person const& p) const;
	bool operator<(int i) const;
};

bool operator<(int i, Person const& p);
```

Template



Since Templates are resolved at compile time, it slows down compilation. **But speeds up runtime.**

Template Function:

```
template <class T>
T max ( T const& t1, T const& t2){
      return t1 < t2 ? t2 : t1 ;
}
```

Compiler will deduce which version to call:
max (33, 44)
max (x, 0)
max (s1, s2)

Specifically requesting compiler to call the double version: max<double>(33,2.0)

Note: If you don't put cost reference, you won't be able to pass a literal e.g. 0.

Template Class:

```
template <class t>
class Accum{
private:
      T total;
public:
      Accum(T start): total (start) { };
      T operator+= ( T const& t) {
            return total = total + t;
      }
      T GetTotal() const {
            return total;
      }
};
```

```
Accum <int> integers(0);
Accum <string> strings("");
```

C++ 17

```
Accum integers(0);
Accum strings("");
```

Template Specialization

Sometimes a template won't work for a particular class:

1. operator/function missing
2. Cannot add
3. Logic in the operator won't work for the class

- First Choice: Add the operator/function with the right logic
- Second Choice: Specialize the template

```
Accum <Person> people ( start )
Person P1("Barshan", "Das",123)
Person P2("Someone", "Else", 456 )
people += P1;
people += P2;
```

Error:

+ not implemented for Person

1. Could write + for Person class
2. Template Specialization

```
template <>
class Accum<Person>{
private:
      int total;
public:
      Accum(int start) : total(start) {};
      int operator+=(Person const& t){
            return total = total + t.GetNumber();
      }
      int GetTotal() const { return total; }
};
```

Indirection

Reference


```
int& rA = a;
rA = 5;

Person P;
Person& rP = P;
rP.SetNumber(345);
```

Pointer


```
int* pA = &A;

*pA = 5;
int b = 100;
pA = &b;

Person P("Barshan", "Das",123);
Person* pA = &P;
name = (*pA).GetName();
name = pA->GetName();
```

Null Pointer – Not pointing to anything

0, NULL

Modern C++ - nullptr

const : A way to commit to compiler something won't change

1. Declare a variable: int const zero = 0
2. As a function parameter :
   a. value: int foo (int const i)
   b. reference: int somefunction (Person const& p)
3. Modifier on a member function: int GetName() const ;

Note: Const after or before

1. const after, const right or east const.
   int const ci = 3;
   this is easy to understand
2. Another style:
   const int ci = 3;

Compiler considers **both same**.

const with Indirection

- ✓ Reference cannot retarget.
- ✓ const reference means, we cannot change the value. int const& rI;

const int

```
int i =3;
int const ci = 3
i = 4;      // valid
ci = 4;     // Invalid.
```

const cannot be changed.

const reference

```
int const& cri = i;
cri = 6 ; // Invalid.
```

cannot change value through const reference

Non-const reference cannot refer const.

```
int& ncri = ci ; // not allowed
```

Function parameters – int vs int reference vs int const reference

```
int j = 10;
int DoubleJ = DoubleIt(j);
int DoubleTen = DoubleIt(10);
```

```
// value parameter
int DoubleIt(int x) {
      return x * 2;
}
```

```
// This cannot take constant literal as argument
int DoubleIt(int& x ) {
      return x * 2;
}
```

```
// const& can take constant literal
int DoubleIt(int const& x )
{
      return x * 2;
}
```

```
Person P("Barshan", "Das", "234");
P.SetNumber(235);
Person const cP = P;

//cost cannot call non-const member function
cP.SetNumber(236); // Invalid

//can call only member functions marked as const
int number = cP.GetNumber(); // valid
```

```
class Person
{
private:
    std::string firstname;
    std::string lastname;
    int arbitrarynumber;

public:
    Person(std::string first,
        std::string last,
        int arbitrary);
    ~Person();
    std::string GetName() const;
    int GetNumber() const {return arbitrarynumber;}
    void SetNumber(int number) {arbitrarynumber = number;}
};
```

Pointers can be const – in two ways

1. Pointer is const but the value is not.
   Example: int * const cpI = &x; ------ means we cannot change it to point somewhere else

   cpI = &y; ----- This won't compile

```
int * const cpI = pI ; // const pointer
*cpI = 4;   // valid
cpI = &j;   // error
```

2. Pointer points to a const. Pointer is changeable but its pointing at something that's not.
   Example:

   int const * cpI = &x; ----- we cannot use this cpI to change the value of target

   ***cpI = 2; ----- This won't compile**

```
int const * pcI = pI; // pointer to a const
*pcI = 4; // error
pcI = &j; // valid
```

3. Can be done both ways too ------ the pointer is const and it points at something const.
   Example:

```
int const * const crazy = pI;
*crazy = 4; // error
crazy = &j; // error
j = *crazy  // valid. Only dereferencing
```

We cannot change the pointer to point somewhere else or use it to change the value of target.

Note: **Type safety** – Pointers are of some type. Cannot point to anything. int* will point only to int.

```
int* pI = &i;
int* pII = &pI; // error
```

Indirection and Inheritance

Base class Reference

- Base class reference can refer to derived class Instance
- Base class reference, referring to derived class instance, can call base class function.
  - Virtual Function – Derived class function executes. This is polymorphism.
  - Non- virtual function – Base class function executes

- Base class reference cannot call derived class functions (which are only in derived class)
- Derived class reference cannot refer base class instance.
  - Derived class may have extra member that base class reference may not know.

Base class pointer

- Base class pointer can point to derived class instance
- Base class pointer can call base class function.
  - Virtual function – derived class function executes
  - Non-virtual function – base class function executes
- Derived class pointer cannot point to base class instance.

Smart pointer and Inheritance – **Same rule** for smart pointers. Smart pointers **behave same** like regular pointers and that includes **polymorphism**.

- ✓ **Virtual Destructor -** If you have atleast **one virtual function**, make sure **destructor is also virtual.**
- ✓ Why not make destructor virtual always? Because that will use **virtual table** always. Code becomes slow. Unnecessary.

Slicing –

- If you copy object around, slicing may occur
  - Copy a derived object into a base object – extra member variables fall away
  - Cannot copy a base object into a derived object
- Same rules apply when passing to a function by value
  - A copy is made
  - Slicing will happen

Use reference or pointers to avoid slicing.

Casting with indirection – casting base class pointer to derived class

1. C style cast

   Dangerous

```
Tweeter *pt = (Tweeter * ) p;
```

2. static_cast<type>

   Compile time only

   Up to programmer to be sure.

```
Tweeter *pt = static_cast<Tweeter*> p;
```

> Note:
> **const_cast** – For casting away const. not modern c++.
>
> **reinterpret_cast** – For bit twilddling

3. dynamic_cast<type>

   Runtime check.

   Works only when casting a pointer to a class with virtual table, Has at least one virtual function.

   Returns null if cast fails

   Slower but safe

```
Tweeter *pt = dynamic_cast<Tweeter*> p;
```

Memory Management

- Local variables are on stack.
- Dynamically allocated memory is from Heap - the Free Store.
- Memory allocated by new operator.
  - Returns pointer to the object or instance.
  - Uses constructor to initialize object.
- Release memory by delete operator. Triggers destructor for clean-up.

For raw array --> new [ ] and delete[ ]
In modern C++, we don't use raw array.

Problem of manual memory management
Managing pointers:

Delete soon	Delete twice
Delete late	Never Delete

Rule of Three:

Rule of Five:

Rule of three +

Move Constructor, Move copy constructor

Manual Memory Management

```
int main()
{
	Person P("Barshan", "Das", 123);
	P.AddResource();
	P.AddResource();
	Person P2 = P;
	return 0;
}
```

```
class Person
{
private:
	std::string firstname;
	std::string lastname;
	int arbitraryNumber;
	Resource *pResource;

public:
	Person();
	Person(std::string first, std::string last, int arbitrary);
	Person(Person const& p);
	~Person();
	Person& operator=(Person const& p);
	std::string GetName();
	void SetName(std::string first, std::string last);
	void AddResource();
};
```

Best: Rule of zero – Easy memory management

```
void Person::AddResource() {

	pResource.reset();


	pResource =
	std::make_shared<Resource>(
	"Resource for " + GetName()
	);

}
```

```
class Person
{
private:
	std::string firstname;
	std::string lastname;
	int arbitraryNumber;
	std::shared_ptr<Resource> pResource;

public:
	Person();
	Person(std::string first, std::string last, int arbitrary);
	std::string GetName();
	void SetName(std::string first, std::string last);
	void AddResource();
};
```

Smart Pointer:

1. **auto_ptr**
   - manages an object obtained via new expression
   - deletes the object when auto_ptr itself is destroyed.
   - deprecated as of C++11.

```
auto_ptr<A> p1(new A);
p1->show();

// returns the memory address of p1
cout << p1.get() << endl;

// copy constructor called, makes p1 empty.
auto_ptr<A> p2(p1);
p2->show();

// p1 is empty now
cout << p1.get() << endl;

// p1 gets copied in p2
cout << p2.get() << endl;
```

Why is auto_ptr deprecated?
It takes ownership of the pointer in a way that no two pointers should contain the same object. Assignment transfers ownership and resets the rvalue auto pointer to a null pointer. Thus, they can't

2. **unique_ptr**
   - can not be copied
   - use std::move to move

```
std::unique_ptr<int> p1 = std::make_unique<int>(42);
std::unique_ptr<int> p2;

p2 = std::move(p1); // contents of p1 moved to p2
```

3. **shared_ptr**
   - maintains count of reference object

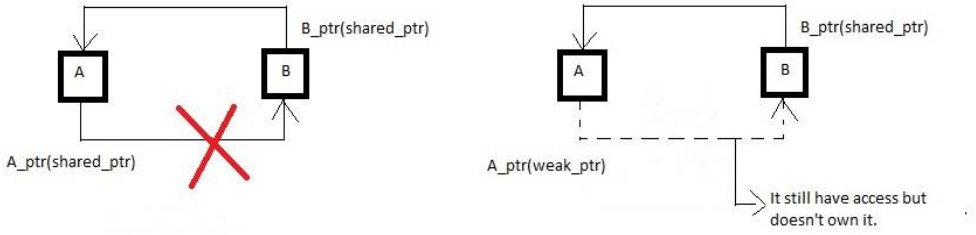
```
std::shared_ptr<Resource> pResource;

pResource.reset();

pResource = std::make_shared<Resource>( );
```

4. **weak_ptr**
   - Allows to peek at a shared_ptr without bumping the refernece count
   - Created as copy of shared_ptr.
   - Provides access to an object that is owned by one or more shared_ptr instances.
   - Does not participate in reference counting.
   - Existence/destruction of weak_ptr has no effect on the shared_ptr or its other copies.
   - Required in some cases to break circular references between shared_ptr instances.

```
std::weak_ptr<int> gw;
auto sp = std::make_shared<int>(42);
gw = sp;
```



When to use weak_ptr?
When you do want to refer to your object from multiple places – for those references for which it's okay to ignore and deallocate (so it will be just noted the object is gone when we try to dereference).